达梦数据库

达梦技术手册

DM8_dmPython 使用手册

Service manual of DM8_dmPython

前言

概述

本文档主要介绍 DM 数据库的 Python 接口 dmPython。包括 dmPython 的各个接口、属性的介绍及其使用说明与示例。

读者对象

本文档主要适用于 DM 数据库的：

- 开发工程师
- 测试工程师
- 技术支持工程师

通用约定

在本文档中可能出现下列标志，它们所代表的含义如下：

表 0.1 标志含义

标志	说明
 警告：	表示可能导致系统损坏、数据丢失或不可预知的结果。
 注意：	表示可能导致性能降低、服务不可用。
 小窍门：	可以帮助您解决某个问题或节省您的时间。
 说明：	表示正文的附加信息，是对正文的强调和补充。

在本文档中可能出现下列格式，它们所代表的含义如下：

表 0.2 格式含义

格式	说明
宋体	表示正文。
黑体	标题、警告、注意、小窍门、说明等内容均采用黑体。
Courier new	表示代码或者屏幕显示内容。
粗体	表示命令行中的关键字（命令中保持不变、必须照输的部分）或者正文中强调的内容。
<>	语法符号中，表示一个语法对象。
::=	语法符号中，表示定义符，用来定义一个语法对象。定义符左边为语法对象，右边为相应的语法描述。
|	语法符号中，表示或者符，限定的语法选项在实际语句中只能出现一个。
{ }	语法符号中，大括号内的语法选项在实际的语句中可以出现 0…N 次(N 为大于 0 的自然数)，但是大括号本身不能出现在语句中。
[ ]	语法符号中，中括号内的语法选项在实际的语句中可以出现 0…1 次，但是中括号本身不能出现在语句中。
关键字	关键字在 DM_SQL 语言中具有特殊意义，在 SQL 语法描述中，关键字以大写形式出现。但在实际书写 SQL 语句时，关键字既可以大写也可以小写。

访问相关文档

如果您安装了 DM 数据库，可在安装目录的“\doc”子目录中找到 DM 数据库的各种手册与技术丛书。

您也可以通过访问我们的网站 www.dameng.com阅读或下载 DM 的各种相关文档。

联系我们

如果您有任何疑问或是想了解达梦数据库的最新动态消息，请联系我们：

网址：www.dameng.com

技术服务电话：400-991-6599

技术服务邮箱：dmtech@dameng.com

目录

1dmPython 简介

dmPython 是 DM 提供的依据 Python DB API version 2.0 中 API 使用规定而开发的数据库访问接口。dmPython 实现这些 API，使 Python 应用程序能够对 DM 数据库进行访问。

dmPython 通过调用 DM DPI 接口完成 python 模块扩展。在其使用过程中，除 Python 标准库以外，还需要 DPI 的运行环境。

dmPython 接口当前版本号为 2.3，下表指明了 dmPython 接口的版本与服务器版本和 python 的版本之间的对应情况。

表 1.1 dmPython 版本对照表

dmPython 版本	DM Server 版本	Python 版本
2.3	7.0.0.9 以上	2.6 及以上

2dmPython 安装

dmPython 可以运行在任何安装了 python 的平台上。可以使用安装包安装，也可以直接用源码安装。另外，需要保证 dpi 和 dmPython 版本一致，都是 32 位或都是 64 位。

dmPython 的运行需要使用 dpi 动态库，用户应将 dpi 所在目录（一般为 DM 安装目录中的 bin 目录）加入系统环境变量。

在 Windows 操作系统下安装 dmPython 只需要直接执行 exe 文件即可。Windows 操作系统下生成 exe 文件操作如下：

```
//进入到 setup.py 所在的源码目录，执行以下命令：
python setup.py bdist_wininst
```

在 Linux 操作系统下使用 rpm 包安装 dmPython。安装和卸载命令参考如下：

```
安装：rpm -ivh dmPython-2.1-7.1-py33-1.x86_64.rpm --nodeps
卸载：rpm -e dmPython-2.1-1.x86_64
```

Linux 操作系下生成 rpm 包操作如下：

```
//进入到 setup.py 所在的源码目录，执行以下命令：
python setup.py bdist_rpm
```

3 dmPython 接口详解

3.1 MODULE dmPython

3.1.1 接口

3.1.1.1 dmPython.connect

语法：

```
dmPython.connect(*args, **kwargs)
dmPython.Connect(*args, **kwargs)
```

说明：

创建与数据库的连接，这两个方法完全等效，返回一个 connection 对象。参数为连接属性，所有连接属性都可以用关键字指定，在 connection 连接串中，没有指定的关键字都按照默认值处理。

连接属性 property 列表如下：

表 3.1 连接属性表

关键字	描述	是否必填
user	登录用户名，默认 SYSDBA	否
password	登录密码，默认 SYSDBA	否
dsn	包含主库地址和端口号的字符串，格式为“主库地址:端口号”	否
host / server	主库地址，包括 IP 地址、localhost 或者主库名，默认 localhost，注意 host 和 server 关键字只允许指定其中一个，含义相同。	否
port	端口号，服务器登录端口号，默认 5236	否
access_mode	连接的访问模式，默认为读写模式	否
autoCommit	DML 操作是否自动提交，默认 TRUE	否
connection_timeout	执行超时时间(s)，默认 0 不限制	否
login_timeout	登录超时时间(s)，默认为 5	否
txn_isolation	事务隔离级，默认使用服务器的隔离级	否

app_name	应用程序名	否
compress_msg	消息是否压缩，默认 FALSE	否
use_stmt_pool	是否开启语句句柄缓存池，默认 TRUE	否
ssl_path	SSL 证书所在的路径，默认为空	否
ssl_pwd	SSL 加密密码，只允许在连接前设置，不允许读取	否
mpp_login	是否以 LOCAL 方式登录 MPP 系统，默认 FALSE 以 GLOBAL 方式登录 MPP 系统	否
crypto_name	加密方式，只允许在连接前设置，不允许读取。必须与 certificate 配合使用	否
certificate	加密密钥，只允许在连接前设置，不允许读取。必须与 crypto_name 配合使用	否
rwseparate	是否启用读写分离方式，默认为 FALSE	否
rwseparate_percent	读写分离比例(%)，默认为 25	否
cursor_rollback_behavior	回滚后游标的状态，默认为不关闭游标	否
lang_id	错误消息的语言，默认为中文	否
local_code	客户端字符编码方式，默认当前环境系统编码方式	否

举例说明：

例 1：

```
import dmPython
conn = dmPython.connect(user='SYSDBA', password='SYSDBA', server='localhost',
port=5236, autoCommit=True)
```

例 2：

```
import dmPython
properties = { 'user' : 'SYSDBA', 'password' : 'SYSDBA', 'server' : '127.0.0.1',
'port' : 5236, 'autoCommit' : True, }
conn = dmPython.connect(**properties)
```

例 3：

如果不需指定除了 user、password、ip、port 以外的其他连接属性，还可以使用以下两种方式创建连接：

方式 1：dmPython.connect(user, password, "ip:port")

```
import dmPython
conn = dmPython.connect('SYSDBA', 'SYSDBA', 'localhost:5236')
```

方式 2：dmPython.connect("user/password@ip:port")

```
import dmPython

conn = dmPython.connect('SYSDBA/SYSDBA@localhost:5236')
```

注意：

如果 connect 接口中又重复指定 host/server 或 port，则忽略重复指定的值。

方式 1 给出的是省略关键字的写法，先后顺序不能打乱，如果指定有关键字，则不要求先后顺序。举例如下：

```
import dmPython

conn = dmPython.connect(dsn='localhost:5236', user='SYSDBA', password='SYSDBA')
```

方式 2 中，连接串内只允许出现"user/password@ip:port"这四个关键字，如果要省略的话，只能按照从后往前的关键字顺序依次省略，不允许省略掉中间某个字段，省略掉的关键字按照默认值处理，也不允许在串内再拼接其他关键字，如果需要指定，则需要单独使用对应的关键字来赋值，举例如下：

```
import dmPython

conn = dmPython.connect('SYSDBA/SYSDBA@localhost:5236')

conn = dmPython.connect('SYSDBA/SYSDBA@localhost')

conn = dmPython.connect('SYSDBA/SYSDBA')

conn = dmPython.connect('SYSDBA')

conn = dmPython.connect('SYSDBA/SYSDBA@localhost:5236', autoCommit=True)
```

3.1.1.2 dmPython.Date

语法：

```
dmPython.Date(year, month, day)
```

说明：

同标准 datetime.date(year, month, day)。

3.1.1.3 dmPython.DATE

说明：

日期类型对象，用于描述列属性。

例如，下面的例子说明了日期类型数据的插入与查询。

```
from datetime import date

d = date(2015,6,10)

print (d)

import dmPython

conn = dmPython.connect()

cursor = conn.cursor()

cursor.execute("create table test_date(c1 date)")

cursor.execute("insert into test_date values(?)", d)

Seq_params = [(d,), (d,)]

cursor.executemany("insert into test_date values(?)", Seq_params)

cursor.execute("select * from test_date")

cursor.description

cursor.fetchall()
```

3.1.1.4 dmPython.DateFromTicks

语法：

```
dmPython.DateFromTicks(ticks)
```

说明：

指定 ticks(从新纪元开始的秒值)构造日期类型对象。

3.1.1.5 dmPython.Time

语法：

```
dmPython.Time(hour[,minute[,second[,microsecond[,tzinfo]]]])
```

说明：

同标准 datetime.time(hour[, minute[, second[, microsecond[, tzinfo]]]])。

3.1.1.6 dmPython.TIME

说明：

时间类型对象，用于描述列属性。

例如，下面的例子说明了时间类型数据的插入与查询。

```
from datetime import time

t = time(12,13,14)

print (t)

import dmPython

conn = dmPython.connect()

cursor = conn.cursor()

cursor.execute("create table test_time(c1 time)")

cursor.execute("insert into test_time values(?)", t)

Seq_params = [(t,), (t,)]

cursor.executemany("insert into test_time values(?)", Seq_params)

cursor.execute("select * from test_time")

cursor.description

cursor.fetchall()

>>>
```

3.1.1.7 dmPython.TimeFromTicks

语法：

```
dmPython.TimeFromTicks(ticks)
```

说明：

指定 ticks(从新纪元开始的秒值)构造时间类型对象。

3.1.1.8 dmPython.Timestamp

语法：

```
dmPython.Timestamp(year,month,day[,hour[,minute[,second[,microsecond[,tzinfo]
]]]])
```

说明：

同标准 datetime.datetime(year, month, day[, hour[, minute[, second[, microsecond[, tzinfo]]]]])。

3.1.1.9 dmPython.TIMESTAMP

说明：

时间戳类型对象，用于描述列属性，对应达梦数据库中的 TIMESTAMP 和 TIMESTAMP WITH LOCAL TIME ZONE 本地时区类型。

例 1，下面的例子说明了时间戳类型数据的插入与查询。

```
from datetime import datetime
ts = datetime(2015,6,10,17,51,52,53)
print(ts)
import dmPython
dmPython.TIMESTAMP
conn = dmPython.connect()
cursor = conn.cursor()
cursor.execute("create table test_timestamp(c1 timestamp)")
cursor.execute("insert into test_timestamp values(?)", ts)
Seq_params = [(ts,), (ts,)]
cursor.executemany("insert into test_timestamp values(?)", Seq_params)
cursor.execute("select * from test_timestamp")
cursor.description
cursor.fetchall()
```

例 2，下面的例子说明了 TIMESTAMP WITH LOCAL TIME ZONE 类型数据的插入与查询。

```
import dmPython
dmPython.TIMESTAMP
```

```
conn = dmPython.connect()
cursor = conn.cursor()
i = '2002-12-12 09:10:21 +8:00';
cursor.execute("create table test_timestamplocaltz(C1 TIMESTAMP(3) WITH LOCAL
TIME ZONE)")
cursor.execute("insert into test_timestamplocaltz values(?)", i)
i1 = '2011-11-11 02:10:21 -8:00';
i2 = '2015-06-17 15:12:15 +2:00';
Seq_params = [(i1,), (i2,)]
cursor.executemany("insert into test_timestamplocaltz values(?)", Seq_params)
cursor.execute("select * from test_timestamplocaltz")
cursor.description
cursor.fetchall()
```

3.1.1.10 dmPython.TimestampFromTicks

语法：

```
dmPython.TimestampFromTicks(ticks)
```

说明：

指定 ticks(从新纪元开始的秒值)构造日期时间类型对象。

3.1.1.11 dmPython.StringFromBytes

语法：

```
dmPython.StringFromBytes(bytes)
```

说明：

将二进制字节串转换为相应的字符串表示。

低于 3 的 Python 版本中将二进制串也认为是常规字符串，而二进制串和字符串绑定到 SQL 类型为 BINARY 参数时，DM 数据库服务器内部处理不一样，因此，对于将二进制串直接绑定到 SQL 类型为 BINARY 参数的需求将无法满足，如将 BLOB 对象中读取的二进制串的绑定插入等。可以通过使用本函数，在低于 3 的 Python 版本中，手动将已经获取的二进制

串转换为相应的字符后，再执行插入即可。

例如：

```
import dmPython

conn = dmPython.connect()

cursor = conn.cursor()

cursor.execute("drop table t_blob")

cursor.execute("create table t_blob(c1 blob)")

cursor.execute("insert into  t_blob values (0x456)")

cursor.execute("select  * from t_blob")

row =cursor.fetchone()

b =row[0].read()

cursor.execute("insert into  t_blob values (?)", b)  #直接绑定会报错

b

strb = dmPython.StringFromBytes(b)

strb

cursor.execute("insert into  t_blob values (?)", strb)
```

3.1.1.12 dmPython.NUMBER

说明：

用于描述达梦数据库中的 BYTE/TINYINT/SMALLINT/INT/INTEGER 类型。

例如，下面是一个 INT 类型的使用示例。

```
import dmPython

conn = dmPython.connect()

cursor = conn.cursor()

dmPython.NUMBER

i = 1234

cursor.execute("create table test_int(c1 int)")

cursor.execute("insert into test_int values(?)", i)

i1 = 5678
```

```
i2 = 9001

Seq_params = [(i1,), (i2,)]

cursor.executemany("insert into test_int values(?)", Seq_params)

cursor.execute("select * from test_int")

cursor.description

cursor.fetchall()
```

3.1.1.13 dmPython.BIGINT

说明：

用于描述达梦数据库中的 BIGINT 类型。

例如：

```
import dmPython

conn = dmPython.connect()

cursor = conn.cursor()

dmPython.BIGINT

i = 9991011119

cursor.execute("create table test_bigint(c1 bigint)")

cursor.execute("insert into test_bigint values(?)", i)

i1 = -9223372036854775808

i2 = 9223372036854775807

Seq_params = [(i1,), (i2,)]

cursor.executemany("insert into test_bigint values(?)", Seq_params)

cursor.execute("select * from test_bigint")

cursor.description

cursor.fetchall()
```

3.1.1.14 dmPython.ROWID

说明：

用于描述 DM 数据库中的 ROWID，ROWID 列在达梦中是伪列，用来标识数据库基表中每

一条记录的唯一键值，实际上在表中并不存在。允许查询 ROWID 列，不允许增删改操作。

例如：

```
import dmPython

dmPython.ROWID
```

3.1.1.15 dmPython.DOUBLE

说明：

用于描述 DM 数据库中的 FLOAT/DOUBLE/DOUBLE PRECISION 类型。

例如：

```
import dmPython

dmPython.DOUBLE

conn = dmPython.connect()

cursor = conn.cursor()

i = 1.2345

cursor.execute("create table test_float(c1 float)")

cursor.execute("insert into test_float values(?)", i)

i1 = 2.3456

i2 = 5.6789

Seq_params = [(i1,), (i2,)]

cursor.executemany("insert into test_float values(?)", Seq_params)

cursor.execute("select * from test_float")

cursor.description

cursor.fetchall()
```

3.1.1.16 dmPython.REAL

说明：

用于描述 DM 数据库中的 REAL 类型（映射为 C 语言中的 float 类型），由于 Python 不支持单精度浮点数类型(float)，查询到的结果转换为 double 输出后，可能会和实际值在小数位上有出入。

例如：

```
import dmPython

dmPython.REAL

conn = dmPython.connect()

cursor = conn.cursor()

i = 9.8765

cursor.execute("create table test_real(c1 real)")

cursor.execute("insert into test_real values(?)", i)

i1 = 6.789016

i2 = 5.432156

Seq_params = [(i1,), (i2,)]

cursor.executemany("insert into test_real values(?)", Seq_params)

cursor.execute("select * from test_real")

cursor.description

cursor.fetchall()
```

3.1.1.17 dmPython.DECIMAL

说明：

用于描述 DM 数据库中的 NUMERIC/NUMBER/DECIMAL/DEC 类型，用于存储零、正负定点数。

例如：

```
import dmPython

dmPython.DECIMAL

from decimal import Decimal

i = Decimal('123.45');

conn = dmPython.connect()

cursor = conn.cursor()

cursor.execute("create table test_decimal(c1 decimal(5, 2))")

cursor.execute("insert into test_decimal values(?)", i)
```

```
i1 = Decimal('89.98')

i2 = Decimal('-89.90')

Seq_params = [(i1,), (i2,)]

cursor.executemany("insert into test_decimal values(?)", Seq_params)

cursor.execute("select * from test_decimal")

cursor.description

cursor.fetchall()
```

3.1.1.18 dmPython.STRING

说明：

用于描述 DM 数据库中的变长字符串类型(VARCHAR/VARCHAR2)。

例如：

```
import dmPython

dmPython.STRING

conn = dmPython.connect()

cursor = conn.cursor()

vch = 'varchartest'

cursor.execute("create table test_varchar(c1 varchar)")

cursor.execute("insert into test_varchar values(?)", vch)

vch1 = 'testmany' * 2

vch2 = 'testmany' * 5

Seq_params = [(vch1,), (vch2,)]

cursor.executemany("insert into test_varchar values(?)", Seq_params)

cursor.execute("select * from test_varchar")

cursor.description

cursor.fetchall()
```

3.1.1.19 dmPython.FIXED_STRING

说明：

用于描述达梦数据库中的定长字符串类型（CHAR/ CHARACTER）。

例如：

```
import dmPython

dmPython.FIXED_STRING

conn = dmPython.connect()

cursor = conn.cursor()

ch = 'test'

cursor.execute("create table test_char(c1 char(10))")

cursor.execute("insert into test_char values(?)", ch)

ch1 = 'testmany'

Seq_params = [(ch1,), (ch1,)]

cursor.executemany("insert into test_char values(?)", Seq_params)

cursor.execute("select * from test_char")

cursor.description

cursor.fetchall()
```

3.1.1.20 dmPython.UNICODE _STRING

说明：

Python2.x 版本中 dmPython 支持的类型，表示变长的 UNICODE 字符串。

例如：

```
import dmPython

dmPython.UNICODE_STRING

conn = dmPython.connect()

cursor = conn.cursor()

u = u'abcd'

cursor.execute('create table test_unicode_string(c1 varchar)')

cursor.execute('insert into test_unicode_string values(?)', u)

u1 = u'中文'

u2 = u'test 测试'
```

```
seq = [(u1,), (u2,)]
cursor.executemany('insert into test_unicode_string values(?)', seq)
cursor.execute('select * from test_unicode_string')
cursor.description
cursor.fetchall()
```

3.1.1.21 dmPython.FIXED_UNICODE _STRING

说明：

Python2.x 版本中 dmPython 支持的类型，表示定长的 UNICODE 字符串。

例如：

```
import dmPython
dmPython.FIXED_UNICODE_STRING
conn = dmPython.connect()
cursor = conn.cursor()
u = u'abcd'
cursor.execute('create table test_fixed_unicode(c1 char(15))')
cursor.execute('insert into test_fixed_unicode values(?)', u)
u1 = u'中文'
u2 = u'test 测试'
seq = [(u1,), (u2,)]
cursor.executemany('insert into test_fixed_unicode values(?)', seq)
cursor.execute('select * from test_fixed_unicode')
cursor.fetchall()
```

3.1.1.22 dmPython.BINARY

说明：

用于描述 DM 数据库中的变长二进制类型（VARBINARY），以十六进制显示。

例如：

```
import dmPython
```

```
dmPython.BINARY

conn = dmPython.connect()

cursor = conn.cursor()

b = b'12'

cursor.execute("create table test_varbinary(c1 varbinary)")

cursor.execute("insert into test_varbinary values(?)", b)

b1 = b'ABCD56ccaadd'

b2 = b'abcd34EFFDA'

Seq_params = [(b1,), (b2,)]

cursor.executemany("insert into test_varbinary values(?)", Seq_params)

cursor.execute("select * from test_varbinary")

cursor.description

cursor.fetchall()
```

3.1.1.23 dmPython.FIXED_BINARY

说明：

用于描述 DM 数据库中的定长二进制类型（BINARY），以十六进制显示。

例如：

```
import dmPython

dmPython.FIXED_BINARY

conn = dmPython.connect()

cursor = conn.cursor()

b = b'12'

cursor.execute("create table test_binary(c1 binary(10))")

cursor.execute("insert into test_binary values(?)", b)

b1 = b'abcd56'

b2 = b'34efgh'

Seq_params = [(b1,), (b2,)]

cursor.executemany("insert into test_binary values(?)", Seq_params)
```

```
cursor.execute("select * from test_binary")
cursor.description
cursor.fetchall()
```

3.1.1.24 dmPython.BOOLEAN

说明：

用于描述 DM 数据库中的 BIT 类型，对应 Python 中的 True/False。

例如：

```
import dmPython
conn = dmPython.connect()
cursor = conn.cursor()
dmPython.BOOLEAN
i = True
cursor.execute("create table test_bit(c1 bit)")
cursor.execute("insert into test_bit values(?)", i)
i1 = True
i2 = False
Seq_params = [(i1,), (i2,)]
cursor.executemany("insert into test_bit values(?)", Seq_params)
cursor.execute("select * from test_bit")
cursor.description
cursor.fetchall()
```

3.1.1.25 dmPython.BLOB、dmPython.CLOB、dmPython.LOB

说明：

用于描述 DM 数据库中大字段数据类型。其中，dmPython.BLOB 和 dmPython.CLOB 分别用于描述 BLOB 和 CLOB 数据类型；dmPython.LOB 用于描述用户获取大字段对象后，用于在外部操作大字段对象类型，拥有自己的操作方法，详见 3.4 节。

其定义如下：

```
import dmPython
dmPython.BLOB
dmPython.CLOB
dmPython.LOB
```

3.1.1.26 dmPython.BFILE、dmPython.exBFILE

说明：

用于描述 DM 数据库中 BFILE 数据类型。其中，dmPython.BFILE 用于描述 BFILE 数据类型；dmPython.exBFILE 用于描述用户获取 BFILE 对象后，用于在外部操作 BFILE 对象类型，拥有自己的操作方法，详见 3.5 节。

其定义如下：

```
import dmPython
dmPython.BFILE
dmPython.exBFILE
```

3.1.1.27 dmPython.INTERVAL

说明：

日期间隔类型对象（年月间隔类型不包括在内），用于描述列属性。

例如，下面的例子说明了日期间隔类型数据的插入与查询。

```
from datetime import timedelta
d = timedelta(days = 10, seconds = 0, microseconds=10)
print(d)
import dmPython
dmPython.INTERVAL
conn = dmPython.connect()
cursor = conn.cursor()
cursor.execute('drop table t_intv;')
cursor.execute('create table t_intv(f1 interval day to second);')
cursor.execute('insert into t_intv values(?)', d)
```

```
Seq_params = [(d,), (d,)]
cursor.executemany("insert into t_intv values(?)", Seq_params)
cursor.execute('select * from t_intv')
cursor.description
cursor.fetchall()
```

3.1.1.28 dmPython.YEAR_MONTH_INTERVAL

说明：

日期间隔类型中的年月间隔类型，用于描述列属性。由于 Python 没有提供具体的年月间隔接口，插入时需要使用字符串方式。

例如：

```
import dmPython
conn = dmPython.connect()
cursor = conn.cursor()
dmPython.YEAR_MONTH_INTERVAL
ym = "INTERVAL '05-05' YEAR TO MONTH"
cursor.execute('create table test_iym(c1 INTERVAL YEAR TO MONTH);')
cursor.execute('insert into test_iym values(?)', ym)
ym1 = "INTERVAL '32-01' YEAR TO MONTH"
ym2 = "INTERVAL '15-00' YEAR TO MONTH"
Seq_params = [(ym1,), (ym2,)]
cursor.executemany("insert into test_iym values(?)", Seq_params)
cursor.execute('select * from test_iym')
cursor.description
cursor.fetchall()
```

3.1.1.29 dmPython.TIME_WITH_TIMEZONE

说明：

带时区的 TIME 类型，用于描述 DM 数据库中的 TIME WITH TIME ZONE 类型，是标

准时区类型。由于 Python 没有提供具体的时区类型接口，插入时需要使用字符串方式。

例如：

```
import dmPython
dmPython.TIME_WITH_TIMEZONE
conn = dmPython.connect()
cursor = conn.cursor()
i = '09:10:21 +8:00';
cursor.execute("create table test_timetz(c1 time(2) with time zone)")
cursor.execute("insert into test_timetz values(?)", i)
i1 = '02:10:21 -8:00';
i2 = '15:12:15 +2:00';
Seq_params = [(i1,), (i2,)]
cursor.executemany("insert into test_timetz values(?)", Seq_params)
cursor.execute("select * from test_timetz")
cursor.description
cursor.fetchall()
```

3.1.1.30 dmPython.TIMESTAMP_WITH_TIMEZONE

说明：

带时区的 TIMESTAMP 类型，用于描述 DM 数据库中的 TIMESTAMP WITH TIME ZONE 类型，为标准时区类型，由于 Python 没有提供具体的时区类型接口，插入时需要使用字符串方式。

例如：

```
import dmPython
dmPython.TIMESTAMP_WITH_TIMEZONE
conn = dmPython.connect()
cursor = conn.cursor()
i = '2002-12-12 09:10:21 +8:00';
cursor.execute("create table test_timestamptz(c1 timestamp(2) with time zone)")
```

```
cursor.execute("insert into test_timestamptz values(?)", i)

i1 = '2011-11-11 02:10:21 -8:00';

i2 = '2015-06-17 15:12:15 +2:00';

Seq_params = [(i1,), (i2,)]

cursor.executemany("insert into test_timestamptz values(?)", Seq_params)

cursor.execute("select * from test_timestamptz")

cursor.description

cursor.fetchall()
```

3.1.1.31 dmPython.CURSOR

说明：

游标类型，支持使用游标作为存储过程或存储函数的绑定参数，以及存储函数的返回值类型。

例如，下面的例子创建存储函数，并使用游标作为输入输出参数及返回值类型（借用 3.1.1.26 例子中的表）。

```
import dmPython

conn = dmPython.connect()

cursor = conn.cursor()

cursor.execute('CREATE OR REPLACE function CURSOR_func(cs1 in out CURSOR)'

    '\nreturn CURSOR'

    '\nAS'

    '\nCURSOR C1 IS SELECT * FROM t_intv;'

    '\nCURSOR C2 IS SELECT * FROM t_intv;'

    '\nBEGIN'

    '\nOPEN C1;'

    '\nOPEN C2;'

    '\ncs1 := C1;'

    '\nreturn C2;'

    '\nEND;')
```

```
cs1 = conn.cursor()

rets = cursor.callfunc('CURSOR_func', cs1)

rets

rets[0].description

rets[0].fetchall()

rets[1].description

rets[1].fetchall()

rets[0].close()

rets[1].close()
```

3.1.1.32 dmPython.Error

说明：

dmPython 的错误类型，保存 dmPython 模块执行中的异常。

例如，下面的例子捕获执行过程中发生的异常，并打印出来。

```
try:
   import dmPython
   conn=dmPython.connect()
   cursor=conn.cursor()
   cursor.execute('create table t(c1 int)')
   print(cursor.description)
except dmPython.Error as e:
   sqlError =  "Error:%s" % str(e)
   print(sqlError)
```

3.1.1.33 dmPython.objectvar

语法：

```
dmPython.objectvar(connection,name[pkgname,schema])
```

说明：

构造 OBJECT 对象，可以是数组（ARRAY/SARRAY），也可以是结构体（CLASS、RECORD）。

各参数说明见下表。

表 3.2 dmPython.objectvar 参数介绍

参数	参数类型	描述
Connection	Connection 对象	对应用于获取 OBJECT 对象描述信息的指定连接
name	字符串	对应获取 OBJECT 对象名称
pkgname	字符串	对应获取 OBJECT 对象所属的包名，若对应数据类型 DSQL_CLASS，则无包，直接输入 None
schema	字符串	对应获取 OBJECT 对象所属模式名，仅 CLASS 类型有效

关于 Ojbect 的详细描述请看 3.6 节。

下面是一个简单的示例，在本例中，CLS1 为 DSQL_CLASS 类型，T_ARR_ARR 为定义在包 CMP_PKG_T 中 DSQL_ARRAY 类型，均创建在当前 SYSDBA 模式下。

```
import dmPython

conn = dmPython.connect()

obj = dmPython.objectvar(conn, 'CLS1', None, 'SYSDBA')

obj

obj_arr_arr = dmPython.objectvar(conn, 'T_ARR_ARR', 'CMP_PKG_T', 'SYSDBA')
```

3.1.2 常量

3.1.2.1 dmPython.apilevel

支持的 Python DB API 版本。当前使用‘2.0’。

3.1.2.2 dmPython.threadsafety

支持线程的安全级别。当前值为 1，线程可以共享模块，但不能共享连接。

3.1.2.3 dmPython.paramstyle

支持的标志参数格式。当前值为‘qmark’，支持‘？’按位置顺序绑定，不支持按名称绑定参数。

例如：

```
("insert into test(c1, c2) values(?, ?)", 1, 2)
```

3.1.2.4 dmPython.version

dmPython 的版本号。当前为“2.3”。

3.1.2.5 dmPython.buildtime

扩展属性，记录 dmPython 创建时间。

例如：

```
>>> dmPython.buildtime
'May 15 2015 13:40:58'
```

3.1.2.6 dmPython.ShutdownType

服务器关闭 shutdown 命令类型常量类，给出所有 shutdown 命令类型常量，具体有以下取值：

- dmPython.SHUTDOWN_DEFAULT：默认值，正常关闭
- dmPython.SHUTDOWN_ABORT：强制关闭
- dmPython.SHUTDOWN_IMMEDIATE：立即关闭
- dmPython.SHUTDOWN_TRANSACTIONAL：等待事务都完成后关闭
- dmPython.SHUTDOWN_NORMAL：正常关闭

3.1.2.7 dmPython.DebugType

服务器 debug 命令类型常量类，给出所有 debug 命令类型常量，具体有以下取值：

- dmPython.DEBUG_CLOSE: 关闭服务器调试
- dmPython.DEBUG_OPEN：打开服务器调试，记录 SQL 日志为非切换模式，输出的日志为详细模式
- dmPython.DEBUG_SWITCH：打开服务器调试，记录 SQL 日志为切换模式，输出的日志为详细模式
- dmPython.DEBUG_SIMPLE：打开服务器调试，记录 SQL 日志为非切换模式，输

出日志为简单模式

3.1.2.8 dmPython.TransIsoType

会话事务隔离级别的常量类，给出所有隔离级别常量，具体有以下取值：

- dmPython.ISO_LEVEL_READ_DEFAULT：默认隔离级，即服务器的隔离级是读提交(ISO_LEVEL_READ_COMMITTED)
- dmPython.ISO_LEVEL_READ_UNCOMMITTED：未提交可读
- dmPython.ISO_LEVEL_READ_COMMITTED：读提交
- dmPython.ISO_LEVEL_REPEATABLE_READ：重复读，暂不支持
- dmPython.ISO_LEVEL_SERIALIZABLE：串行化。

3.1.2.9 dmPython.accessMode

连接访问属性值，有以下取值：

- dmPython.DSQL_MODE_READ_ONLY：以只读的方式访问数据库
- dmPython.DSQL_MODE_READ_WRITE：以读写的方式访问数据库

3.1.2.10 dmPython.autoCommit

语句是否自动提交属性值，有以下取值：

- dmPython.DSQL_AUTOCOMMIT_ON：打开自动提交开关
- dmPython.DSQL_AUTOCOMMIT_OFF：关闭自动提交开关

3.1.2.11 dmPython.code

支持编码方式常量，用于连接上服务器和本地编码方式，与 code_map.h 中支持的编码方式一致。访问方式与其他常量一样，如访问 UTF8 编码：

```
>>> import dmPython
>>> dmPython.PG_UTF8
1
>>> dmPython.PG_UTF9
Traceback (most recent call last):
```

```
  File "<stdin>", line 1, in <module>
AttributeError: 'module' object has no attribute 'PG_UTF9'
>>>
```

支持的编码方式如下对照表。

表 3.3 dmPython.code 取值对照表

参数值	编码方式	code_mape.h	dmPython 中常量
1	UTF8	PG_UTF8	PG_UTF8
2	GBK	PG_GBK	PG_GBK
3	BIG5	PG_BIG5	PG_BIG5
4	ISO_8859_9	PG_ISO_8859_9	PG_ISO_8859_9
5	EUC_JP	PG_EUC_JP	PG_EUC_JP
6	EUC_KR	PG_EUC_KR	PG_EUC_KR
7	KOI8R	PG_KOI8R	PG_KOI8R
8	ISO_8859_1	PG_ISO_8859_1	PG_ISO_8859_1
9	SQL_ASCII	PG_SQL_ASCII	PG_SQL_ASCII
10	GB18030	PG_GB18030	PG_GB18030
11	ISO_8859_11	PG_ISO_8859_11	PG_ISO_8859_11

3.1.2.12 dmPython.lang_id

支持语言类型常量，用于设置连接上 lang_id 属性，具体有以下取值：

- dmPython.LANGUAGE_CN：中文
- dmPython.LANGUAGE_EN：英文

3.1.2.13 dmPython.bool

DM DPI 支持的 bool 类型的表达常量，具体有以下取值：

- dmPython.DSQL_TRUE：对应 bool 类型的 TRUE
- dmPython.DSQL_FALSE：对应 bool 类型的 FALSE

3.1.2.14 dmPython.rwseparate

DM DPI 支持的关于读写分离开关的相关属性常量，具体有以下取值：

- dmPython.DSQL_RWSEPARATE_ON：打开读写分离
- dmPython.DSQL_RWSEPARATE_OFF：关闭读写分离

3.1.2.15 dmPython.trx_state

DM DPI 支持的关于事务状态的相关属性常量，具体有以下取值：

- dmPython.DSQL_TRX_ACTIVE：事务处于活动状态
- dmPython.DSQL_TRX_COMPLETE：事务执行完成

3.1.2.16 dmPython.use_stmt_pool

DM DPI 支持的关于语句句柄缓存池的相关属性常量，具体有以下取值：

- dmPython.DSQL_TRUE：开启语句句柄缓存池
- dmPython.DSQL_FALSE：关闭语句句柄缓存池

3.1.2.17 dmPython.mpp_login

DM DPI 支持的关于 mpp 登陆方式的相关属性常量，具体有以下取值：

- dmPython.DSQL_MPP_LOGIN_GLOBAL：全局登陆
- dmPython.DSQL_MPP_LOGIN_LOCAL：本地登陆

3.1.2.18 dmPython.cursor_rollback_behavior

DM DPI 支持的关于回滚后游标状态的相关属性常量，具体有以下取值：

- dmPython.DSQL_CB_PRESERVE：回滚后不关闭游标
- dmPython.DSQL_CB_CLOSE：回滚后关闭游标

3.2 Connection

3.2.1 接口

3.2.1.1 Connection.cursor

语法：

```
Connection.cursor()
```

说明：

构造一个当前连接上的 cursor 对象，用于执行操作。

3.2.1.2 Connection.commit

语法：

```
Connection.commit()
```

说明：

手动提交当前事务。如果设置了非自动提交模式，可以调用该方法手动提交。

3.2.1.3 Connection.rollback

语法：

```
Connection.rollback()
```

说明：

手动回滚当前未提交的事务。

3.2.1.4 Connection.close 、Connection.disconnect

语法：

```
Connection.close()

Connection.disconnect()
```

说明：

关闭与数据库的连接。

3.2.1.5 Connection.debug

语法：

```
Connection.debug([debugType])
```

说明：

打开服务器调试，可以指定 dmPython.DebugType 的一种方式打开，不指定则使用默认方式 dmPython.DEBUG_OPEN 打开。

3.2.1.6 Connection.shutdown

语法：

```
Connection.shutdown([shutdownType])
```

说明：

关闭服务器，可以指定 dmPython.ShutdownType 的一种方式关闭，不指定则使用默认方式 dmPython.SHUTDOWN_DEFAULT 关闭。

3.2.1.7 Connection.explain

语法：

```
Connection.explan(sql)
```

说明：

返回指定 SQL 语句的执行计划。

例如：

```
import dmPython

conn = dmPython.connect()

info = conn.explain('select * from t_mixtp')

print(info)



1   #NSET2: [0, 4, 100]
```

```
2    #PRJT2: [0, 4, 100]; exp_num(6), is_atom(FALSE)

3      #CSCN2: [0, 4, 100]; INDEX33555574(T_MIXTP)
```

```
info = conn.explain('select * from dual')

print(info)



1  #NSET2: [0, 1, 1]

2    #PRJT2: [0, 1, 1]; exp_num(1), is_atom(FALSE)

3      #CSCN2: [0, 1, 1]; SYSINDEXSYSDUAL(SYSDUAL as DUAL)
```

3.2.2 属性

3.2.2.1 Connection.access_mode

连接访问模式，对应 DPI 属性 DSQL_ATTR_ACCESS_MODE，可以设置为 dmPython.accessMode 的一种连接访问模式。

例如：

```
import dmPython

conn = dmPython.connect()

conn.access_mode

conn.DSQL_ATTR_ACCESS_MODE

conn.DSQL_ATTR_ACCESS_MODE = dmPython.DSQL_MODE_READ_ONLY

conn.DSQL_ATTR_ACCESS_MODE
```

3.2.2.2 Connection.async_enable

允许异步执行，读写属性，对应 DPI 属性 DSQL_ATTR_ASYNC_ENABLE，暂不支持。

3.2.2.3 Connection.auto_ipd

是否自动分配参数描述符，只读属性，对应 DPI 属性 DSQL_ATTR_AUTO_IPD。

3.2.2.4 Connection.compress_msg

消息是否压缩，对应 DPI 属性 DSQL_ATTR_COMPRESS_MSG，仅能在创建连接时通过关键字 compress_msg 进行设置：

- DSQL_TRUE：压缩
- DSQL_FALSE：不压缩

例如：

```
import dmPython
conn = dmPython.connect()
conn.compress_msg
conn.DSQL_ATTR_COMPRESS_MSG
conn2 = dmPython.connect(compress_msg=dmPython.DSQL_TRUE)
conn2.compress_msg
```

3.2.2.5 Connection.rwseparate、Connection.rwseparate_percent

读写分离相关属性，分别对应 DPI 属性 DSQL_ATTR_RWSEPARATE 和 DSQL_ATTR_RWSEPARATE_PERCENT。Connection.rwseparate 可以设置为 dmPython.rwseparate 的取值。这两个属性都应在连接创建之前设置，创建后只能进行读访问。由于 DM 数据库提供的读写分离功能需要其他环境配置，因此，如果仅设置了读写分离属性，连接后返回的属性值也不一定与设置相同。

例如：

```
>>> import dmPython
>>> dmPython.DSQL_RWSEPARATE_ON
1
>>> dmPython.DSQL_RWSEPARATE_OFF
0
>>> conn = dmPython.connect()
>>> conn.rwseparate
0
```

```
>>> conn.DSQL_ATTR_RWSEPARATE
0
>>> conn.rwseparate=dmPython.DSQL_RWSEPARATE_ON
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
dmPython.DatabaseError: 连接已打开
>>> conn2 =
dmPython.connect(rwseparate=dmPython.DSQL_RWSEPARATE_ON,rwseparate_p
ercent=50)
>>> conn2.rwseparate
0
>>> conn2 =
dmPython.connect(rwseparate=dmPython.DSQL_RWSEPARATE_ON,rwseparate_p
ercent=50)
>>> conn2.rwseparate
0
>>> conn2.rwseparate_percent
50
```

3.2.2.6 Connection.server_version

服务器版本号，只读属性。

3.2.2.7 Connection.current_schema

当前模式，只读属性，对应 DPI 属性 DSQL_ATTR_CURRENT_SCHEMA。用户可通过执行 SQL 语句 set schema 来更改当前模式。

例如：

```
>>> import dmPython
>>> conn = dmPython.connect()
>>> conn.current_schema
```

```
'SYSDBA'
>>> conn.DSQL_ATTR_CURRENT_SCHEMA
'SYSDBA'
>>> cursor = conn.cursor()
>>> cursor.execute("create schema user_sch")
>>> cursor.execute("set schema user_sch")
>>> conn.current_schema
'USER_SCH'
>>> conn.DSQL_ATTR_CURRENT_SCHEMA
'USER_SCH'
>>>
```

3.2.2.8 Connection.server_code

服务器端编码方式，只读属性，对应 DPI 属性 DSQL_ATTR_SERVER_CODE。

3.2.2.9 Connection.local_code

客户端本地的编码方式，对应 DPI 属性 DSQL_ATTR_LOCAL_CODE。

例如：

```
>>> import dmPython
>>> conn = dmPython.connect()
>>> conn.local_code
10
>>> conn.DSQL_ATTR_LOCAL_CODE
10
>>> conn.local_code = dmPython.PG_UTF8
>>> conn.DSQL_ATTR_LOCAL_CODE
1
```

3.2.2.10 Connection.lang_id

错误消息的语言，仅能在创建连接时通过关键字 lang_id 进行设置。对应 DPI 属性 DSQL_ATTR_LANG_ID。

例如：

```
>>> import dmPython
>>> conn = dmPython.connect()
>>> conn.lang_id
0
>>> conn.DSQL_ATTR_LANG_ID
0
>>> dmPython.LANGUAGE_EN
1
>>> dmPython.LANGUAGE_CN
0
>>> conn2 = dmPython.connect(lang_id=dmPython.LANGUAGE_EN)
>>> conn2.lang_id
1
>>> conn2.DSQL_ATTR_LANG_ID
1
>>>
```

3.2.2.11 Connection.app_name

应用程序名称，仅能在连接创建时通过关键字 app_name 设置目标应用名称。对应 DPI 属性 DSQL_ATTR_APP_NAME。

例如：

```
>>> import dmPython
>>> conn = dmPython.connect()
>>> conn.app_name
'python'
```

```
>>> conn.DSQL_ATTR_APP_NAME
'python'
>>> conn2 = dmPython.connect(app_name='dmPython')
>>> conn2.app_name
'dmPython'
```

3.2.2.12 Connection.txn_isolation

会话的事务隔离级别，对应 DPI 属性 DSQL_ATTR_TXN_ISOLATION。

例如：

```
>>> import dmPython
>>> conn = dmPython.connect()
>>> conn.txn_isolation
1
>>> conn.DSQL_ATTR_TXN_ISOLATION
1
>>> conn.txn_isolation = dmPython.ISO_LEVEL_READ_UNCOMMITTED
>>> conn.DSQL_ATTR_TXN_ISOLATION
0
```

3.2.2.13 Connection.autoCommit

DML 语句是否自动提交，可以设置为 dmPython.autoCommit 的取值。与 DPI 属性 DSQL_ATTR_AUTOCOMMIT 对应。

例如：

```
>>> import dmPython
>>> conn = dmPython.connect()
>>> conn.autoCommit
1
>>> conn.DSQL_ATTR_AUTOCOMMIT
1
```

```
>>> conn.DSQL_ATTR_AUTOCOMMIT = dmPython.DSQL_AUTOCOMMIT_OFF
>>> conn.DSQL_ATTR_AUTOCOMMIT
0
```

3.2.2.14 Connection.connection_dead

检查连接是否存活，对应 DPI 属性 DSQL_ATTR_CONNECTION_DEAD，尚未支持。

3.2.2.15 Connection.connection_timeout

连接超时时间，以秒为单位，0 表示不限制。对应 DPI 属性 DSQL_ATTR_CONNECTION_TIMEOUT。

例如：

```
>>> import dmPython
>>> conn = dmPython.connect()
>>> conn.connection_timeout
0
>>> conn.connection_timeout = 100
>>> conn.connection_timeout
100
>>> conn.DSQL_ATTR_CONNECTION_TIMEOUT
100
```

3.2.2.16 Connection.login_timeout

登录超时时间，以秒为单位，对应 DPI 属性 DSQL_ATTR_LOGIN_TIMEOUT。

例如：

```
>>> import dmPython
>>> conn = dmPython.connect()
>>> conn.login_timeout
5
>>> conn.DSQL_ATTR_LOGIN_TIMEOUT
```

```
5
>>> conn.login_timeout = 3
>>> conn.DSQL_ATTR_LOGIN_TIMEOUT
3
```

3.2.2.17 Connection.packet_size

网络数据包大小，对应 DPI 属性 DSQL_ATTR_PACKET_SIZE，暂不支持。

3.2.2.18 Connection.dsn

当前连接的 IP 和端口号，仅允许在建立连接时进行设置，连接建立后，只允许读。

例如：

```
>>> import dmPython
>>> conn = dmPython.connect(dsn='192.168.0.91:5236', user='SYSDBA',
password='SYSDBA')
>>> conn.dsn
'192.168.0.91:5236'
>>>
>>> conn.dsn = 'localhost:5236'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: readonly attribute
```

3.2.2.19 Connection.user

当前登录的用户名，只读属性，对应 DPI 属性 DSQL_ATTR_LOGIN_USER。

3.2.2.20 Connection.port

当前登录数据库服务器的端口号，仅允许在创建连接时进行设置，连接创建后，只可读。对应 DPI 属性 DSQL_ATTR_LOGIN_PORT。

3.2.2.21 Connection.server

登录服务器的主库，只读属性，对应 DPI 属性 DSQL_ATTR_LOGIN_SERVER。

3.2.2.22 Connection.inst_name

当前登录服务器的实例名称，只读属性，对应 DPI 属性 DSQL_ATTR_INSTANCE_NAME。

3.2.2.23 Connection.mpp_login

MPP 登陆方式，仅允许在创建连接时进行设置，可设置为 dmPython.mpp_login 的取值，连接创建后，只可读。对应 DPI 属性 DSQL_ATTR_MPP_LOGIN。

例如：

```
>>> import dmPython
>>> conn = dmPython.connect()
>>> conn.mpp_login
0
>>> conn.mpp_login = '1'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
dmPython.DatabaseError: 连接已打开
>>>
>>> conn2 = dmPython.connect(mpp_login = 1)
>>> conn2.mpp_login
1
```

3.2.2.24 Connection.str_case_sensitive

字符大小写是否敏感，只读属性，对应 DPI 属性 DSQL_ATTR_STR_CASE_SENSITIVE。

3.2.2.25 Connection.max_row_size

行最大字节数，只读属性，对应 DPI 属性 DSQL_ATTR_MAX_ROW_SIZE。

3.2.2.26 Connection.server_status

DM 服务器的模式和状态，只读属性。

3.2.2.27 Connection.warning

最近一次警告信息，只读属性。

例如：

```
>>> import dmPython
>>> conn = dmPython.connect()
>>> cursor = conn.cursor()
>>>
>>> cursor.execute("select * from t1")
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
dmPython.DatabaseError: 第 1 行附近出现错误:
无效的表或视图名[T1]
>>>
>>> print (conn.warning)
>>>第 1 行附近出现错误:
无效的表或视图名[T1]
```

3.2.2.28 Connection.current_catalog

当前连接的数据库实例名,只读属性,对应 DPI 属性 DSQL_ATTR_CURRENT_CATALOG。

3.2.2.29 Connection.trx_state

事务状态，只读属性，对应 DPI 属性 DSQL_ATTR_TRX_STATE。

3.2.2.30 Connection.use_stmt_pool

是否开启语句句柄缓存池，仅允许在创建连接时进行设置，可设置为

dmPython.use_stmt_pool 的取值，连接创建后，只可读，对应 DPI 属性 DSQL_ATTR_USE_STMT_POOL。

3.2.2.31 Connection.ssl_path

SSL 证书所载的路径，仅允许在创建连接时进行设置，连接创建后，只可读，对应 DPI 属性 DSQL_ATTR_SSL_PATH。

例如：

```
>>> import dmPython
>>> conn = dmPython.connect()
>>> conn.ssl_path
''
>>> conn.ssl_path = 'D:/client_ssl/SYSDBA'
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
dmPython.DatabaseError: 连接已打开
>>>
>>> conn2 = dmPython.connect(ssl_path = 'D:/client_ssl/SYSDBA')
>>> conn2.ssl_path
'D:/client_ssl/SYSDBA'
```

3.2.2.32 Connection.cursor_rollback_behavior

回滚后游标的状态，仅允许在创建连接时进行设置，可设置为 dmPython.cursor_rollback_behavior 的取值，连接创建后，只可读，对应 DPI 属性 DSQL_ATTR_CURSOR_ROLLBACK_BEHAVIOR。

3.3 Cursor

3.3.1 接口

3.3.1.1 Cursor.callproc

语法：

```
Cursor.callproc(procname, *args)
```

说明：

调用存储过程，返回执行后的所有输入输出参数序列。如果存储过程带参数，则必须为每个参数键入一个值，包括输出参数。

procname：存储过程名称，字符串类型

args：存储过程的所有输入输出参数

例如：下面的例子说明了如何调用无参数的存储过程。

```
>>> cursor.execute('create or replace procedure test_proc_1() as begin print
true;end;')
>>> print(cursor.callproc('test_proc_1'))
[]
```

下面的例子则说明了如何调用带参数的存储过程。

```
>>> cursor.execute('create or replace procedure test_proc_2(p1 int, p2 out int)
as begin p2 = p1 + 1;end;')
>>> print(cursor.callproc('test_proc_2', 10000, 0))
[10000, 10001]
```

3.3.1.2 Cursor.callfunc

语法：

```
Cursor.callfunc(funcname, *args)
```

说明：

调用存储函数，返回存储函数执行的返回值以及所有参数值。返回序列中第一个元素为函数返回值，后面的是函数的参数值。如果存储函数带参数，则必须为每个参数键入一个值

包括输出参数。

funcname：存储函数名称，字符串类型

args：存储函数的所有参数

例如：下面的例子说明了如何调用无参数的存储函数。

```
>>> cursor.execute('create or replace function test_func_1() return int as begin
return 123;end;')
>>> print(cursor.callfunc('test_func_1'))
[123]
```

下面的例子则说明了如何调用带参数的存储函数。

```
>>> cursor.execute('create or replace function test_func_2(p1 int, p2 out int)
return int as begin p2 = p1 + 1;return 456;end;')
>>> print(cursor.callfunc('test_func_2', 10000, 0))
[456, 10000, 10001]
```

3.3.1.3 Cursor.prepare

语法：

```
Cursor.prepare(sql)
```

说明：

准备给定的 SQL 语句。后续可以不指定 sql，直接调用 execute。

例如：

```
>>> import dmPython
>>> conn = dmPython.connect()
>>> cursor = conn.cursor()
>>> cursor.prepare('insert into t_fetch values(?)')
>>>
>>> cursor.execute(None, 1)
>>> cursor.execute(None, 2)
>>> cursor.executemany(None, [[3,],[4,]])
```

3.3.1.4 Cursor.execute

语法：

```
Cursor.execute(sql[,parameters]|[,**kwargsParams])
```

说明：

执行给定的 SQL 语句，给出的参数值和 SQL 语句中的绑定参数从左到右一一对应.如果给出的参数个数小于 SQL 语句中需要绑定的参数个数或者给定参数名称绑定时未找到，则剩余参数按照 None 值自动补齐。若给出的参数个数多于 SQL 语句中需要绑定参数个数，则自动忽略。

例如：

```
cursor.execute("create table test(c1 int, c2 varchar)")


#按位置动态绑定
cursor.execute("insert into test values(?, ?)", 1, 'abcdefg')


#按位置绑定数组
values = (99, 'today')
cursor.execute("insert into test values(?, ?)", values)


#按名称动态绑定
cursor.execute("insert into test values(:cp1, :cp2)", cp1=1, cp2='abcdefg')


#按名称绑定字典
params_map = {'cp1':'99','cp2':'today'}
cursor.execute("insert into test values(:cp1, :cp2)", params_map)


#若按名称动态绑定参数前面，出现其他类型绑定，则忽略名称动态绑定，下面例子以 values 绑定为准
values = (99, 'today')
cursor.execute("insert into test values(?, ?)", values, cp1=1, cp2='abcdefg')
```

3.3.1.5 Cursor.executedirect

语法：

```
Cursor.executedirect(sql)
```

说明：

执行给定的 SQL 语句，不支持参数绑定。

例如：

```
cursor.executedirect("create table test(c1 int, c2 varchar)")
cursor.executedirect("insert into test values(1, 'abcdefg')")
```

3.3.1.6 Cursor.executemany

语法：

```
Cursor.executemany(sql, sequence_of_params)
```

说明：

对给定的 SQL 语句进行批量绑定参数执行。参数用各行的 tuple 组成的序列给定。

例如：

```
Seq_params = [(1, 'abcdefg'), (2, 'uvwxyz')]
cursor.executemany("insert into test values(?, ?)", Seq_params)
```

3.3.1.7 Cursor.close

语法：

```
Cursor.close()
```

说明：

关闭 Cursor 对象。

3.3.1.8 Cursor.fetchone、Cursor.next

语法：

```
Cursor.fetchone()
```

```
Cursor.next()
```

说明：

获取结果集的下一行，返回一行的各列值，返回类型为 tuple。如果没有下一行返回 None。

3.3.1.9 Cursor.fetchmany

语法：

```
Cursor.fetchmany([rows=Cursor.arraysize])
```

说明：

获取结果集的多行数据，获取行数为 rows，默认获取行数为属性 Cursor.arraysize 值。返回类型为由各行数据的 tuple 组成的 list，如果 rows 小于未读的结果集行数，则返回 rows 行数据，否则返回剩余所有未读取的结果集。

例如：

```
>>> import dmPython
>>> conn = dmPython.connect()
>>> cursor = conn.cursor()
>>>
>>> cursor.execute('drop table test')
>>> cursor.execute('create table test(c1 int, c2 varchar)')
>>> cursor.executedirect("insert into test values(1, 'abcdefg')")
>>>
>>> Seq_params = [(1, 'abcdefg'), (2, 'uvwxyz')]
>>> cursor.executemany("insert into test values(?, ?)", Seq_params)
>>> cursor.executemany("insert into test values(?, ?)", Seq_params)
>>> cursor.executemany("insert into test values(?, ?)", Seq_params)
>>>
>>> cursor.execute('select * from test')
<builtins.DmdbCursor on <dmPython.Connection to SYSDBA@localhost:5236>>
>>>
```

```
>>> cursor.arraysize
50
>>> cursor.fetchmany(rows = 1)
[(1, 'abcdefg')]
>>> cursor.fetchmany(rows = 3)
[(1, 'abcdefg'), (2, 'uvwxyz'), (1, 'abcdefg')]
>>> cursor.fetchmany()
[(2, 'uvwxyz'), (1, 'abcdefg'), (2, 'uvwxyz')]
>>> cursor.fetchmany()
[]
```

3.3.1.10 Cursor.fetchall

语法：

```
Cursor.fetchall()
```

说明：

获取结果集的所有行。返回所有行数据，返回类型为由各行数据的 tuple 组成的 list。

3.3.1.11 Cursor.nextset

语法：

```
Cursor.nextset()
```

说明：

获取下一个结果集。如果不存在下一个结果集则返回 None，否则返回 True。可以使用 fetchXXX()获取新结果集的行值。

例如：

```
>>> cursor.execute("begin select 1; select 2;end")
>>> print(cursor.fetchall())
[(1,)]
>>> print(cursor.nextset())
True
```

```
>>> print(cursor.fetchall())
[(2,)]
>>> print(cursor.nextset())
None
```

3.3.1.12 Cursor.setinputsizes

语法：

```
Cursor.setinputsizes(sizes)
```

说明：

在执行操作(executeXXX,callFunc,callProc)之前调用，为后续执行操作中所涉及参数预定义内存空间，每项对应一个参数的类型对象，若指定一个整数数字，则认为对应字符串类型最大长度。

例如：

```
from datetime import timedelta
intv = timedelta(days = 10, seconds = 0, microseconds=10)

from decimal import *
dec = Decimal(12.3)

import dmPython
conn = dmPython.connect()
cursor = conn.cursor()
cursor.setinputsizes(int, 200, float, dmPython.DECIMAL, dmPython.INTERVAL)
cursor.execute('create table t_mixtp (fint int, f2 varchar(200), ffloat float,
fdouble double, finter INTERVAL DAY TO SECOND)')
cursor.execute('insert into t_mixtp values (?, ?, ?, ?, ?)', 1, 'string200', 12.3,
dec, intv)
cursor.execute('insert into t_mixtp (fint, ffloat, finter) values (?, ?, ?)', 1,
dec, intv)
```

3.3.1.13 Cursor.setoutputsize

语法：

```
Cursor.setoutputsize(size[,column])
```

说明：

为某个结果集中的大字段(BLOB/CLOB/LONGVARBINARY/LONGVARCHAR)类型设置预定义缓存空间。若未指定 column，则 size 对所有大字段值起作用。对于大字段类型，dmPython 均以 LOB 的形式返回，故此处无特别作用，仅按标准实现。

3.3.2 属性

3.3.2.1 Cursor.bindarraysize

与 setinputsizes 结合使用，用于指定预先申请的待绑定参数的行数。

例如：

```
>>> import dmPython
>>> conn = dmPython.connect()
>>> cursor = conn.cursor()
>>> cursor.bindarraysize
1
>>> cursor.bindarraysize = 10
>>> cursor.bindarraysize
10
```

3.3.2.2 Cursor.arraysize

fetchmany()一次获取结果集的行数，默认值为 50。

例如：

```
>>> print cursor.arraysize
50
>>> cursor.arraysize = 10
```

```
>>> print cursor.arraysize
10
```

3.3.2.3 Cursor.statement

最近一次执行的 sql 语句，只读属性。

例如：

```
>>> cursor.execute('select * from t3')
<builtins.DmdbCursor on <dmPython.Connection to SYSDBA@localhost:5236>>
>>> cursor.statement
'select * from t3'
```

3.3.2.4 Cursor.with_rows

是否存在非空结果集，只读属性，True 表示非空结果集，False 表示空结果集。

例如：

```
>>> import dmPython
>>> conn = dmPython.connect()
>>> cursor = conn.cursor()
>>> cursor.with_rows
False
>>> cursor.execute('select * from dual')
>>> cursor.fetchall()
[(1,)]
>>> cursor.with_rows
True
>>> cursor.execute('create table t3 (f3 varchar(100))')
>>> cursor.with_rows
False
>>> cursor.execute('select * from t3')
>>> cursor.with_rows
```

```
False
```

3.3.2.5 Cursor.lastrowid

最近一次操作影响的行的 rowid，只读属性。对于 INSERT/UPDATE/DELETE 操作可以查询到 lastrowid 值，其他操作返回 None。

例如：

```
>>> import dmPython
>>> conn = dmPython.connect()
>>> cursor = conn.cursor()
>>> cursor.lastrowid
>>> print(cursor.lastrowid)
None
>>> cursor.execute("create table test(c1 int, c2 varchar)")
>>> cursor.execute("insert into test values(1, 'test')")
>>> cursor.lastrowid
1
```

3.3.2.6 Cursor.connection

当前 Cursor 对象所在的数据库连接，只读属性。

例如：

```
>>> cursor.connection
<dmPython.Connection to SYSDBA@localhost:5236>
```

3.3.2.7 Cursor.description

结果集所有列的描述信息，只读属性。描述信息格式为：tuple(name, type_code, display_size, internal_size, precision, scale, null_ok)。

例如：

```
>>> cursor.execute("select c1, c2 from test")
<builtins.DmdbCursor on <dmPython.Connection to SYSDBA@localhost:5236>>
```

```
>>>
>>> print (cursor.description)
[('C1', <class 'dmPython.NUMBER'>, 11, 10, 10, 0, 1), ('C2', <class 'dmPython.ST
RING'>, 8188, 8188, 8188, 0, 1)]
```

3.3.2.8 Cursor.column_names

当前结果集的所有列名序列，只读属性。

例如：

```
>>> cursor.execute("select c1, c2 from test")
<builtins.DmdbCursor on <dmPython.Connection to SYSDBA@localhost:5236>>
>>> print (cursor.column_names)
['C1', 'C2']
```

3.3.2.9 Cursor.rowcount

最后一次执行查询产生的结果集总数，或者执行插入和更新操作影响的总行数，只读属性。若无法确定，则返回-1。

例如：

```
>>> import dmPython
>>> conn = dmPython.connect()
>>> cursor = conn.cursor()
>>> cursor.execute('insert into t1 values(3)')
>>> cursor.rownumber
-1
>>> cursor.rowcount
1
>>> cursor.execute('select * from t1')
<builtins.DmdbCursor on <dmPython.Connection to SYSDBA@localhost:5236>>
>>> cursor.rowcount
7
```

```
>>> cursor.rownumber
0
>>> cursor.next()
(None,)
>>> cursor.rownumber
1
>>> cursor.rowcount
7
>>> cursor.execute('create table t2 (f1 int)')
>>> cursor.rowcount
-1
>>> cursor.rownumber
-1
```

3.3.2.10 Cursor.rownumber

当前所在结果集的当前行号，从 0 开始，只读属性。若无法确定，则返回-1。

3.4 Lob

Lob 是允许用户独立操作的 LOB 对象描述，包括 CLOB 和 BLOB 对象，对应 dmPython.LOB。

3.4.1 接口

3.4.1.1 Lob.read

语法：

```
Lob.read([offset[, length]])
```

说明：

读取 LOB 对象从偏移 offset 开始的 length 个值，并返回。若为 CLOB 数据对象，则 offset 对应字符偏移位置，length 对应字符个数，返回数据为字符串；若为 BLOB 数据

对象，则 offset 对应字节偏移位置，length 对应字节个数，返回数据为二进制字节对象。offset 默认为 1，length 默认为 LOB 数据的总长度。

3.4.1.2 Lob.write

语法：

```
Lob.write(value[, offset])
```

说明：

向 LOB 对象中从偏移位置 offset 开始写入数据 value，并返回实际写入数据的字节个数。若为 CLOB 数据对象，则 offset 对应字符偏移位置；若为 BLOB 数据对象，offset 对应字节偏移位置。offset 默认为 1。

3.4.1.3 Lob.size

语法：

```
Lob.size()
```

说明：

返回 LOB 数据对象数据长度。若为 CLOB 对象，返回字符个数；BLOB 对象，返回字节个数。

3.4.1.4 Lob.truncate

语法：

```
Lob.truncate([newSize])
```

说明：

截断 LOB 对象，使截断后数据大小为 newSize。对于 CLOB 对象，newSize 对应字符个数；对于 BLOB 对象，newSize 对应字节个数。newSize 默认为 0。

3.4.1.5 Lob.__reduce__

语法：

```
Lob.__reduce__ ()
```

说明：

LOB 对象的归纳操作，包括 LOB 对象数据类型和数据内容，对于每个字段值最多显示 1000 个字符，BLOB 前导字符的"0x"除外。也可以通过 str 或者 print 方法查看每个字段的字符串值，最多也是 1000 个字符。

例如：

```
import sys
longstring = ""
longstring += 'ABCDEF0123456789' * 500
cvalue = longstring


if sys.version_info[0] >= 3 :
   bvalue = longstring.encode("ascii")
else :
  bvalue = longstring



import dmPython
conn = dmPython.connect()
cursor = conn.cursor()
cursor.execute('drop table t_lob')
cursor.execute('create table t_lob(c1 blob, c2 clob)')
cursor.execute('insert into t_lob values(?, ?)', bvalue, cvalue)
cursor.execute('select * from t_lob')
rows = cursor.fetchall()
[blob, clob] = rows[0]
str_b = str(blob)
str_b
len(str_b)
str_c = str(clob)
str_c
```

```
len(str_c)

print(blob)

print(clob)

blob.__reduce__()

clob.__reduce__()
```

3.4.2 举例说明

下面的例子创建一个含有 BLOB 和 CLOB 类型字段的表，向表中插入数据，并执行查询返回数据。

```
import sys

longstring = ""

longstring += 'ABCDEF0123456789' * 500

cvalue = longstring


if sys.version_info[0] >= 3 :

   bvalue = longstring.encode("ascii")

else :

  bvalue = longstring


import dmPython

conn = dmPython.connect()

cursor = conn.cursor()

cursor.execute('drop table t_lob')

cursor.execute('create table t_lob(c1 blob, c2 clob)')

cursor.execute('insert into t_lob values(?, ?)', bvalue, cvalue)

cursor.execute('select * from t_lob')


cursor.description

row = cursor.fetchone()
```

```
(blob, clob) = row

blob

clob

blob.size()

clob.size()

clob.read(100, 10)

clob.write('中国', 100)

clob.read(100, 10)

clob.size()

clob.truncate(500)

clob.size()

blob.truncate(100)

blob.size()
```

3.5 exBFILE

exBFILE 是允许用户独立操作的 BFILE 对象描述，对应 dmPython.exBFILE。

3.5.1 接口

3.5.1.1 exBFILE.read

语法：

```
exBFILE.read([offset[, length]])
```

说明：

读取 exBFILE 对象从偏移 offset 开始的 length 个值，并返回。offset 必须大于等于 1。

3.5.1.2 exBFILE.size

语法：

```
exBFILE.size()
```

说明：

返回 BFILE 数据对象数据长度。

3.5.2 举例说明

下面的例子 BFILE 类型字段的表，向表中插入数据，并执行查询返回数据。

```
import dmPython

conn = dmPython.connect()

cursor = conn.cursor()

cursor.execute("create or replace directory TEST as '/opt/dmdata';")

cursor.execute("select bfilename('TEST','test.txt') FROM DUAL;")

value=cursor.fetchone()[0]

value.size()

value.read(1,5)
```

3.6 Object

3.6.1 属性

Object 的属性见下表。

表 3.4 Object 属性列表

属性名	类型	说明
type	dmPython.ObjectType	只读属性，Object对象的类型描述。如： >>>obj.type <dmPython.ObjectType SYSDBA.CLS1>
valuecount	数字	只读属性，Object 对象所能容纳的数字个数或者已经存在的数字个数（数组类型）

其中，dmPython.ObjectType 也是一个可访问对象，其中也包含相关属性信息，见下表。

表 3.5 dmPython.ObjectTyp 属性列表

属性名	类型	说明

schema	字符串	只读属性，Object 对象所属模式名，若无，则返回空
name	字符串	只读属性，Object 对象的名称
attributes	dmPython.ObjectAttribute 的 List	只读属性，给出 OBJECT 对象各属性描述。如： >>> obj.type.attributes [<dmPython.ObjectAttribute <class 'dmPython.VARCHAR'>>, <dmPython.ObjectAttribute <class 'dmPython.VARCHAR'>>, <dmPython.ObjectAttribute <class 'dmPython.NUMBER '>>]

dmPython.ObjectAttribute 是属性对象类型，同样具有类似属性。其属性见下表。

表 3.6 dmPython.ObjectAttribute 属性列表

属性名	类型	说明
type	dmPython.ObjectType	只读属性，OBJECT 对象中某个属性类型描述。如： >>> obj.type.attributes[0].type <dmPython.ObjectType <class 'dmPython.VARCHAR'>>

3.6.2 接口

3.6.2.1 Object.getvalue

语法：

```
Object.getvalue()
```

说明：

以链表方式返回当前 Object 对象的数据值。若当前对象尚未赋值，则返回空。

例如：

```
import dmPython

conn = dmPython.connect()

obj = dmPython.objectvar(conn, 'CLS1')

obj.getvalue()
```

```
tp = ['test1','test2',8098]

obj.setvalue(tp)

obj.getvalue()
```

3.6.2.2 Object.setvalue

语法：

```
Object.setvalue(value)
```

说明：

为 Object 对象设置值 value。执行后，若 Object 原存在值，则覆盖原对象值。

3.6.3 举例说明

3.6.3.1 简单 CLASS 示例

```
import dmPython

conn = dmPython.connect()

cursor = conn.cursor()

cursor.execute('create or replace class cls as c1 varchar(50); c2 varchar(50);

c3 int; end;')

tp1 = ['ptest','ptest1', 123]

obj1 = dmPython.objectvar(conn, 'CLS')

obj1.setvalue(tp1)

tp2 = ['testobj2','obj2test', 7897]

obj2 = dmPython.objectvar(conn, 'CLS')

obj2.setvalue(tp2)

cursor.execute('create table t_cls (f1 cls)')

cursor.execute('insert into t_cls values(?)', obj1)

cursor.execute('insert into t_cls values(?)', obj2)

cursor.execute('select * from t_cls')

rows = cursor.fetchall()
```

```
cursor.rowcount

rows

rows[0][0].type

rows[0][0].getvalue()

rows[1][0].type

rows[1][0].getvalue()
```

3.6.3.2 简单 ARRAY 示例

```
import dmPython

conn = dmPython.connect()

cursor = conn.cursor()

cursor.execute('create or replace package cmp_pkg as type t_arr is array int[];

end cmp_pkg;')

cursor.execute('''create or replace procedure pro_arr(

    num int,

    p2 in out cmp_pkg.t_arr)

    as

    begin

           for i in 1..num loop

                  print p2[i];

                p2[i] := p2[i] * 2;

           end loop;

    end;''')

tp = [1,2,3,4,5]

arr = dmPython.objectvar(conn, 'T_ARR', 'CMP_PKG')

arr.setvalue(tp)

res = cursor.callproc('pro_arr', 5, arr)

res

res[1]
```

```
res[1].getvalue()
```

3.6.3.3 CLASS 中嵌套 ARRAY 和 CLASS 示例

```
import dmPython
conn = dmPython.connect()
cursor = conn.cursor()
cursor.execute('''create or replace package cmp_pkg
as
type t_arr is array int[];
type t_rec is record( id int, name varchar(50));
type t_arr_rec is record( ids t_arr, names t_rec);
end cmp_pkg;''')
cursor.execute('''create or replace function fun_arr_rec(
    p2 in out cmp_pkg.t_arr_rec
)
return cmp_pkg.t_arr_rec
as
begin
   for i in 1 .. 5 loop
     print p2.ids[1];
    p2.ids[1] := p2.ids[1] * 2;
   end loop;

  print p2.names.id;
  print p2.names.name;

    p2.names.id := 13;
    p2.names.name := 'mod+dameng';
    return p2;
```

```
end;''')

tp_arr = [1, 2, 3, 4, 5]

tp_obj = [1, 'dameng']

tp_mix = [tp_arr, tp_obj]

mix_obj = dmPython.objectvar(conn, 'T_ARR_REC', 'CMP_PKG')

mix_obj.setvalue(tp_mix)

cursor = conn.cursor()

res = cursor.callfunc('fun_arr_rec', mix_obj)
```

3.6.3.4 ARRAY 中嵌套 CLASSS 和 ARRAY 示例

```
import dmPython

conn = dmPython.connect()

cursor = conn.cursor()

cursor.execute('''create or replace package cmp_pkg

as

type t_arr is array int[];

type t_rec is record( id int, name varchar(50));

type t_rec_arr is array t_rec[];

type t_arr_arr is array t_arr[];

type t_rec_mix is record(rec_arr t_rec_arr, arr_arr t_arr_arr);

end cmp_pkg;''')

cursor.execute('''create or replace function fun_mix_arr(

    arr_size int,

    rec_arr in out cmp_pkg.t_rec_arr,

    arr_arr in out cmp_pkg.t_arr_arr

)

return cmp_pkg.t_rec_mix

as

declare p_out cmp_pkg.t_rec_mix;
```

```
begin
   for i in 1 .. arr_size loop
      print rec_arr[i].id;
      print rec_arr[i].name;

    rec_arr[i].id := rec_arr[i].id * 2;
    rec_arr[i].name := rec_arr[i].name || '-mod';
   end loop;

   for i in 1 .. arr_size loop
    for j in 1 .. arr_size loop
        print arr_arr[i][j];
        arr_arr[i][j] := arr_arr[i][j] * 2;
    end loop;
   end loop;

    p_out.rec_arr := rec_arr;
    p_out.arr_arr := arr_arr;
    return p_out;
end;''')
tp_arr = [1, 2, 3, 4, 5]
tp_obj = [1, 'dameng']
tp_arr_arr = [tp_arr, tp_arr, tp_arr, tp_arr, tp_arr]
tp_rec_arr = [tp_obj, tp_obj, tp_obj, tp_obj, tp_obj]
obj_arr_arr = dmPython.objectvar(conn, 'T_ARR_ARR', 'CMP_PKG')
obj_arr_arr.setvalue(tp_arr_arr)
obj_rec_arr = dmPython.objectvar(conn, 'T_REC_ARR', 'CMP_PKG')
obj_rec_arr.setvalue(tp_rec_arr)
cursor = conn.cursor()
res = cursor.callfunc('fun_mix_arr', 5, obj_rec_arr, obj_arr_arr)
```

4 django_dmPython 驱动

4.1 简介及安装

Django是基于Python的Web应用程序框架，django_dmPython是DM提供的Django连接DM数据库的驱动。

django_dmPython可以运行在任何安装了python的平台上，可以使用安装包安装，也可以直接用源码安装。

在Windows操作系统下安装django_dmPython只需要直接执行exe文件即可。

Windows操作系统下生成exe文件操作如下：

```
//进入到 setup.py 所在的源码目录，执行以下命令：
python setup.py bdist_wininst
```

在Linux操作系统下使用rpm包安装django_dmPython。安装和卸载命令参考如下：

```
安装：rpm -ivh django_dmPython-1.0-1.noarch.rpm
卸载：rpm -e django_dmPython-1.0-1.noarch.rpm
```

Linux 操作系下生成 rpm 包操作如下：

```
//进入到 setup.py 所在的源码目录，执行以下命令：
python setup.py bdist_rpm
```

4.2 配置

Django配置数据库默认为sqlite3，这是一个小型数据库。要连接DM数据库，需修改settings.py中的DATABASES元组。配置方法如下：

```
DATABASES = {
   'default': {
      'ENGINE': 'django_dmPython',
      'NAME': 'DAMENG',
      'USER': 'SYSDBA',
      'PASSWORD': 'SYSDBA',
```

```
        'HOST': 'localhost',
        'PORT': '5236',
        'OPTIONS': {'local_code':1,'connection_timeout':5}
    }
}
```

OPTIONS:是各个驱动都支持的选项，只要在 OPTIONS 中以字典对象的方式配置 dmPython.connect 支持的选项即可，例如：'local_code':1。可以包含多个字典对象，用逗号分隔。dmPython.connect 请参考 3.1.1.1 dmPython.connect。

5 sqlalchemy_dm 方言包

5.1 简介及安装

SQlAlchemy是python下的开源软件，提供了SQL工具包及对象关系映射（ORM）工具，让应用程序开发人员使用上SQL的强大功能和灵活性。sqlalchemy_dm方言包是DM提供用于SQlAlchemy连接DM数据库的方法。

1.SQLALchemy软件的安装。例如SQLAlchemy-1.1.10.win-amd64-py2.7.exe。

2.sqlalchemy_dm方言包的软件生成与安装。

sqlalchemy_dm可以运行在任何安装了Python的平台上。生成工具setup.py位于drivers\python\sqlalchemy目录中。不同平台生成安装包的命令如下：

```
Windows：python setup.py bdist_wininst

Linux：python setup.py bdist_rpm
```

生成之后的安装包（例如sqlalchemy_dm-1.1.10.win-amd64.exe）位于drivers\python\sqlalchemy\dist目录中。点击安装包安装即可。

5.2 engine 的配置

create_engine()返回一个数据库引擎，下面是 DM 数据库的配置方法。

```
from sqlalchemy import create_engine

engine =

create_engine('dm://SYSDBA:SYSDBA@localhost:5236/',connect_args={'local_code'

:1,'connection_timeout':15})

    或

engine =

create_engine('dm+dmPython://SYSDBA:SYSDBA@localhost:5236/',connect_args={'lo

cal_code':1,'connection_timeout':15})
```

其中，connect_args是字典选项，只要在connect_args中以字典对象的方式配置dmPython.connect支持的选项即可。可以包含多个字典对象，用逗号分隔。

dmPython.connect请参考3.1.1.1 dmPython.connect。其他配置参考SQlAlchemy官网http://docs.sqlalchemy.org文档。

咨询热线：400-991-6599

技术支持：dmtech@dameng.com

官网网址：www.dameng.com

武汉达梦数据库有限公司
Wuhan Dameng Database Co.,Ltd.

地址：武汉市东湖新技术开发区高新大道999号未来科技大厦C3栋16—19层

16th-19th Floor, Future Tech Building C3, No.999 Gaoxin Road, Donghu New Tech Development Zone,Wuhan,Hubei Province,China

电话：(+86) 027-87588000　传真：(+86) 027-87588810