

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，包括了 dmPython 的安装、dmPython 接口、dmDjango 驱动、dmSQLAlchemy 方言包、DBUtils 包的相关说明。

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
HarmonyOS Sans SC Light	表示正文。
Poppins Light	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。标题、警告、注意、小窍门、说明等内容均采用粗体。
< >	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

1 dmPython 简介.....	1
2 dmPython 安装.....	2
2.1 安装	2
2.1.1 安装 Python	2
2.1.2 安装 dmPython	2
2.2 使用	5
3 dmPython 接口详解.....	8
3.1 MODULE dmPython.....	8
3.1.1 接口	8
3.1.2 常量.....	36
3.2 Connection	41
3.2.1 接口	41
3.2.2 属性.....	44
3.3 Cursor.....	58
3.3.1 接口	58
3.3.2 属性.....	68
3.4 大字段.....	74
3.4.1 举例说明.....	75
3.5 exBFILE.....	76
3.5.1 接口	76
3.5.2 举例说明.....	77

3.6 Object.....	77
3.6.1 属性.....	77
3.6.2 接口.....	79
3.6.3 举例说明.....	80
4 dmDjango 驱动.....	86
4.1 简介及安装.....	86
4.2 举例说明.....	87
4.2.1 创建 Django 项目.....	87
4.2.2 数据库配置.....	87
4.2.3 创建 dm 应用.....	88
4.2.4 创建模型.....	88
4.2.5 进行数据迁移.....	89
4.2.6 对数据库进行操作.....	89
5 dmSQLAlchemy 方言包.....	97
5.1 简介及安装.....	97
5.2 engine 的配置.....	98
5.3 类型映射.....	98
5.4 特殊说明.....	99
5.5 暂不支持功能.....	100
6 DBUtils 包.....	101
6.1 简介.....	101
6.2 使用方法.....	101
6.2.1 PersistentDB.....	102
6.2.2 PooledDB.....	103

手册版本	105
------------	-----

1 dmPython 简介

dmPython 是 DM 依据 Python DB API version 2.0 中 API 使用规定开发的数据库访问接口，Python 应用程序能够通过这些 API 对 DM 数据库进行访问。因 dmPython 通过调用 DM DPI 接口完成 Python 模块扩展，在使用过程中，除 Python 标准库以外，还需要 DPI 的运行环境。

后文中依此介绍了如何安装 dmPython，如何使用 dmPython 支持的接口，如何使用 dmDjango 驱动建立 Django 和 DM 数据库的连接，如何使用 dmSQLAlchemy 方言包建立 SQLAlchemy 与 DM 数据库连接，及如何使用 DBUtils 包进行 DM 数据库连接池的管理。

2 dmPython 安装

2.1 安装

dmPython 可以运行在任何安装了 Python 的平台上。用户需先安装 Python，再安装 dmPython。

2.1.1 安装 Python

Python 软件请用户自行前往 Python 官网下载。安装任何一个版本 Python 均可。如果 Python 版本大于等于 3.12，请务必确保 Python 同时安装了 setuptools 库。

成功安装 Python 之后，可查看 Python 版本号：

```
C:\Users\dameng>python --version  
  
Python 3.7.4
```

2.1.2 安装 dmPython

dmPython 目前提供离线和在线两种安装方式。选择其中一种安装方式即可。



注意：

使用第三方加密算法，dpi 默认是从

DM_HOME/bin/external_crypto_libs 或者 dm_svc.conf 配置文件中

CIPHER_PATH 参数指定的路径加载第三方依赖库，用户需要确保第三方依赖库存在。

2.1.2.1 离线安装

第一步，安装达梦数据库或者下载达梦驱动压缩包。

安装达梦数据库的具体的安装步骤可参考《DM8 安装手册》。本章以 Windows 安装为

例，安装目录为 D:\dmdbms。

或者可以选择下载达梦驱动压缩包并解压。本章以 Window 为例，解压目录为 D:\drivers，后续该目录简称为 drivers 目录。

第二步，配置环境变量。

dmPython 的运行需要使用 DPI 动态库。因此用户需提前配置好环境变量，使用环境变量指定 DPI 的位置。

Linux 环境下，用户须手动将 DPI 动态库所在目录（即 DM 安装目录中的 BIN 目录或者 drivers 目录中的 dpi 目录）加入到 LD_LIBRARY_PATH 环境变量中。并设置环境变量 DM_HOME 等于 DM 安装目录或者 drivers 目录。

Windows 环境下，当 Python 版本小于等于 3.7 时，需要将 DPI 所在 DM 安装目录中的 BIN 目录或者 drivers 目录中的 dpi 目录加入环境变量 PATH, 并将 DM 安装目录或者 drivers 目录加入到环境变量 DM_HOME 中。当 Python 版本大于 3.7 时，仅需配置 DM_HOME 即可。

成功配置之后，可查看环境变量。下面以 Windows 为例。

```
//查询环境变量 DM_HOME  
C:\Users\dameng>echo %DM_HOME%  
  
d:\dmdbms  
  
//查询环境变量 PATH  
C:\Users\dameng>echo %PATH%  
  
..... d:\dmdbms\bin.....
```

第三步，安装 dmPython。

用户既可以使用 DM 软件中的源码安装 dmPython, 也可使用安装文件安装 dmPython。选择其中一种即可。


 方法一 使用源码安装。

dmPython 源码位于达梦安装目录 dmdbms\drivers\python\dmPython 中，安装程序为 setup.py。

进入到 setup.py 所在的源码目录，执行命令：python setup.py install。可以在 Windows 或 Linux 操作系统下安装 dmPython。

```
D:\dmdbms\drivers\python\dmPython>python setup.py install
```

在 Windows 操作系统上, 如果 Python 版本大于等于 3.6, 编译并安装 dmPython 时, 可能会因为 vc++ 版本不够报错, 可通过安装 vs2015 或更高版本的 vs 等途径满足高版本 Python 对 vc++14 的需求。

 方法二 使用安装文件安装。

需要先生成安装文件再进行安装。

可以使用 Python 生成 whl 文件再使用进行安装, 操作如下:

```
//进入到 setup.py 所在的源码目录 (D:\dmdbms\drivers\python\dmPython>), 执行以下命令:

python setup.py bdist_wheel

//如果遇到报错 invalid command 'bdist_wheel', 需更新打包所需工具版本, 去 pypi 官网下载合适版本的 wheel 包, 或者使用 pip 在线更新, 执行命令如下:

pip install wheel

pip install --upgrade setuptools

//生成 whl 文件后, 在当前目录的 dist 文件夹下可以找到生成的 whl 文件。进入 dist 目录, 执行如下安装命令:

pip install dmPython-2.4.5.whl
```

在 Windows 操作系统下生成 exe 文件的操作如下, 之后只需要直接执行 exe 文件即可安装:

```
//进入到 setup.py 所在的源码目录 (D:\dmdbms\drivers\python\dmPython>), 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，生成 rpm 包的操作如下所示：

//进入到 setup.py 所在的源码目录（D:\dmdbms\drivers\python\dmPython>），执行以下命令：

```
python setup.py bdist_rpm
```

Linux 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

第四步，查看 dmPython 版本号。

安装完成后，通过 pip 查看 dmPython 的版本。

```
pip show dmPython
```

2.1.2.2 在线安装

在拥有网络的环境下安装 dmPython，可选择在线安装方式。在线安装方式目前支持的 python 版本为 2.7,3.4-3.12，支持的平台为 win_amd64, x86_64。

第一步，安装 dmPython。

在命令行工具中输入命令。

```
pip install dmPython
```

第二步，查看 dmPython 版本号。

安装完成后，通过 pip 查看 dmPython 的版本。

```
pip show dmPython
```

2.2 使用

在 Python 软件中使用 dmPython 接口时，用户既可以在交互界面逐行输入命令，也可以通过文件批量执行命令。

例 1 在 Python 交互界面逐行输入命令。

```
C:\Users\dameng>python

Python 3.7.4 (tags/v3.7.4:e0935912e, Jul 8 2019, 20:34:20) [MSC v.1916 64 bit (AMD64)] on
win32

Type "help", "copyright", "credits" or "license" for more information.

>>> import dmPython

>>> conn = dmPython.connect('SYSDBA', 'Dmsys_123', 'localhost:5236')

>>> dmPython.apilevel

'2.0'

>>>
```

例 2 通过文件批量执行命令。将待执行的 dmPython 命令全部存在一个文件中，然后一次性执行文件。达到批量执行命令的效果。

文件 test.py 包含多条命令。内容如下：

```
from datetime import date

d = date(2024,6,10)

print (d)

import dmPython

conn = dmPython.connect('SYSDBA/Dmsys_123')

cursor = conn.cursor()

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

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

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

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

cursor.execute("select * from test_date_xxx ")

print(cursor.description)
```

```
print(cursor.fetchall())
```

执行 test.py 文件。

```
2024-06-10
```

```
[('CI', <class 'dmPython.DATE'>, 10, 10, 10, 0, 1)]
```

```
[(datetime.date(2024, 6, 10),), (datetime.date(2024, 6, 10),), (datetime.date(2024, 6, 10),)]
```

3 dmPython 接口详解

本章详细介绍 dmPython 的接口用法。

接口中有一些参数和 dm_svc.conf 中的参数功能一致，如下表所示。如果同一个参数在接口中和在 dm_svc.conf 配置项（全局配置区、服务配置区）中均有设置，但值却不同，则优先顺序为：全局配置区<接口设置<服务配置区。

表 3.1 dmPython 接口和 dm_svc.conf 中功能一致的参数

dmPython 参数	dm_svc.conf 参数
login_timeout	CONNECT_TIMEOUT
compress_msg	COMPRESS_MSG
ssl_path	SSL_CONFIG
ssl_pwd	SSL_CONFIG
rwseparate	RW_SEPARATE
rwseparate_percent	RW_PERCENT
local_code	CHAR_CODE
lang_id	LANGUAGE

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 3.2 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，为 dminit 初始化时，系统强制要求设置的登陆口令，本文档中以“Dmsys_123”为示例密码，实际运行中需要用户自行替换为数据库初始化时设定的密码	是
dsn	包含主机地址和端口号的字符串，格式为“主机地址:端口号”。如主库地址为 ipv6 地址，则应包含在[]中。	否
host/server	主库地址，包括 IP 地址、localhost 或者服务名，默认 localhost，注意 host 和 server 关键字只允许指定其中一个，含义相同。如主库地址为 ipv6 地址，则应包含在[]中。	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(ms)，默认为 5000	否

DM8_dmPython 使用手册

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，压缩算法加载成功时为 TRUE，否则 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
ukey_name	UKEY 登录验证的 UKEY 名，只允许在连接前设置，不允许读取	否
ukey_pin	UKEY 登录验证时的 UKEY 密钥，只允许在连接前设置，不允许读取	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。 取值如下： 1:PG_UTF8 2:PG_GBK 3:PG_BIG5 4:PG_ISO_8859_9 5:PG_EUC_JP	否

DM8_dmPython 使用手册

	<p>6:PG_EUC_KR</p> <p>7:PG_KOI8R</p> <p>8:PG_ISO_8859_1</p> <p>9:PG_SQL_ASCII</p> <p>10:PG_GB18030</p> <p>11:PG_ISO_8859_11</p> <p>因为 PG_ISO_8859_9、PG_EUC_JP、PG_EUC_KR、PG_KOI8R、PG_ISO_8859_1、PG_SQL_ASCII、PG_ISO_8859_11 不支持编码中文，所以使用这些编码方式时会强制将 lang_id 改为英文</p>	
cursorclass	<p>兼容 mysql 用法，表示游标返回结果集形式。取值</p> <p>dmPython.DictCursor 时，结果集为字典类型；取值为</p> <p>dmPython.TupleCursor 为列表类型，默认为列表类型</p>	否
schema	<p>连接默认模式。未设置时，模式为登录用户的默认模式。</p> <p>schema 参数只在创建 cursor 时起作用，即创建 cursor</p> <p>时设置当前 cursor 所在的模式</p>	否
shake_crypto	客户端传输加密算法所在的加密引擎名	否
dmsvc_path	<p>指定 dm_svc.conf 所在目录，不包含文件名，配置后只</p> <p>读取连接所用的服务配置信息，其他配置信息不影响全局</p> <p>设置；设置后优先使用，若在该路径下找不到</p> <p>dm_svc.conf 或者该文件中没有指定的服务配置项，则</p> <p>转向全局配置文件中查找</p>	否

举例说明：

例 1:

```
import dmPython

conn = dmPython.connect(user='SYSDBA', password='Dmsys_123', server='localhost', port=5236,
autoCommit=True)
```

例 2:

```
import dmPython

properties = { 'user' : 'SYSDBA', 'password' : 'Dmsys_123', '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', 'Dmsys_123', 'localhost:5236')
```

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

```
import dmPython

conn = dmPython.connect('SYSDBA/Dmsys_123@localhost:5236/schl')
```

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='Dmsys_123', schema='schl')
```

方式 2 中，连接串内只允许出现"user/password@ip:port/schema"这五个关键字，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/Dmsys_123@localhost:5236/schl')

conn = dmPython.connect('SYSDBA/Dmsys_123@localhost:5236')

conn = dmPython.connect('SYSDBA/Dmsys_123@localhost')

conn = dmPython.connect('SYSDBA/Dmsys_123')

conn = dmPython.connect('SYSDBA/Dmsys_123@localhost:5236', autoCommit=True)

conn = dmPython.connect('SYSDBA/Dmsys_123@localhost:5236', schema='schl')
```

例 4:

使用 ipv6 地址进行连接

```
import dmPython

conn = dmPython.connect(user='SYSDBA', password='Dmsys_123', server=
'fd15:4ba5:5a2b:1008:36cb:3920:f89c:c329', port=5236)
```

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('SYSDBA/Dmsys_123')

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('SYSDBA/Dmsys_123')

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

conn = dmPython.connect('SYSDBA/Dmsys_123')
```

```
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

conn = dmPython.connect('SYSDBA/Dmsys_123')

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('SYSDBA/Dmsys_123')

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]

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

strb = dmPython.StringFromBytes(b)

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('SYSDBA/Dmsys_123')

cursor = conn.cursor()

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('SYSDBA/Dmsys_123')

cursor = conn.cursor()

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，此外，ROWID 也可以作为表中列的数据类型。

例如:

```
import dmPython

conn = dmPython.connect('SYSDBA/Dmsys_123')

cursor = conn.cursor()

cursor.execute("create table test_rowid(c1 rowid)")

i1 = 'AAAAAAAAAAAAAAAAAAAE'

i2 = 'AAAAAAAAAAAAAAAAAAAF'

Seq_params = [(i1), (i2)]

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

cursor.execute("select c1,rowid from test_rowid")

cursor.description

cursor.fetchall()
```

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

conn = dmPython.connect('SYSDBA/Dmsys_123')

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

conn = dmPython.connect('SYSDBA/Dmsys_123')

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

from decimal import Decimal

i = Decimal('123.45');

conn = dmPython.connect('SYSDBA/Dmsys_123')

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

conn = dmPython.connect('SYSDBA/Dmsys_123')

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

conn = dmPython.connect('SYSDBA/Dmsys_123')

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

conn = dmPython.connect('SYSDBA/Dmsys_123')

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

conn = dmPython.connect('SYSDBA/Dmsys_123')

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

conn = dmPython.connect('SYSDBA/Dmsys_123')

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

conn = dmPython.connect('SYSDBA/Dmsys_123')

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('SYSDBA/Dmsys_123')

cursor = conn.cursor()

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 用于在用户获取大字段对象后，在外部操作大字段对象的描述类型，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

conn = dmPython.connect('SYSDBA/Dmsys_123')

cursor = conn.cursor()

cursor.execute('drop table t_intv;')

cursor.execute('create table t_intv(fl 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('SYSDBA/Dmsys_123')

cursor = conn.cursor()

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

conn = dmPython.connect('SYSDBA/Dmsys_123')

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

conn = dmPython.connect('SYSDBA/Dmsys_123')

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 3.1.1.27 例子中的表）。

```
import dmPython

conn = dmPython.connect('SYSDBA/Dmsys_123')

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 3.3 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
schema	字符串	对应获取 OBJECT 对象所属模式名, 仅 CLASS 类

关于 Object 的详细描述请看 3.6 节。

下面是一个简单的示例, 在本例中, CLSI 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('SYSDBA/Dmsys_123')

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

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 版本号。

例如：

```
>>> dmPython.apilevel
'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。

例如：

```
>>> dmPython.version
'2.4.4'
```

3.1.2.5 dmPython.buildtime

dmPython 的创建时间。

例如：

```
>>> dmPython.buildtime  
'Aug 23 2022 09:29:55'
```

3.1.2.6 ShutdownType 命令类型常量

服务器关闭 shutdown 命令类型常量，具体有以下常量：

- 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 Debug 类型常量

服务器 debug 命令类型常量，具体有以下常量：

- 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 会话事务隔离级别常量

会话事务隔离级别的常量，具体有以下常量：

- `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 连接访问属性常量

连接访问属性值，有以下常量：

- `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 自动提交属性常量

语句是否自动提交属性值，有以下常量：

- `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 编码方式常量

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。`dmPython` 执行语句、参数的编码，必须与设置的编码方式一致。访问方式与其他常量一样，如访问 UTF8 编码：

```
>>> import dmPython

>>> dmPython.PG_UTF8

1
```

支持的编码方式如下对照表。

表 3.4 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 语言类型常量

支持语言类型常量，用于设置连接上 lang_id 属性，具体有以下常量：

- dmPython.LANGUAGE_CN：中文
- dmPython.LANGUAGE_EN：英文
- dmPython.LANGUAGE_CNT_HK：繁体中文

3.1.2.13 bool 类型常量

DM DPI 支持的 bool 类型的表达常量，具体有以下常量：

- dmPython.DSQL_TRUE：对应 bool 类型的 TRUE

- `dmPython.DSQL_FALSE`: 对应 `bool` 类型的 `FALSE`

3.1.2.14 读写分离开关的相关属性常量

DM DPI 支持的关于读写分离开关的相关属性常量，具体有以下常量：

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

3.1.2.15 事务状态的相关属性常量

DM DPI 支持的关于事务状态的相关属性常量，具体有以下常量：

- `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 MPP 登陆方式的相关属性常量

DM DPI 支持的关于 MPP 登陆方式的相关属性常量，具体有以下常量：

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

3.1.2.17 回滚后游标状态的相关属性常量

DM DPI 支持的关于回滚后游标状态的相关属性常量，具体有以下常量：

- `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.explain(sql)`

说明:

返回指定 SQL 语句的执行计划。

例如：

```
import dmPython

conn = dmPython.connect('SYSDBA/Dmsys_123')

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)
```

3.2.1.8 Connection.__enter__

语法：

Connection.__enter__()

说明：

返回当前 Connection 对象。__enter__ 是上下文管理器的一部分，用来在进入 with 语句块时执行获取资源操作，无需手动调用。

3.2.1.9 Connection.__exit__

语法:

```
Connection.__exit__()
```

说明:

关闭连接并回滚未提交事务。__exit__是上下文管理器的一部分，用来在离开 with 语句块时执行清理操作，无需手动调用。

3.2.1.10 Connection.ping

语法:

```
Connection.ping(reconnect)
```

说明:

检查当前连接是否可用，reconnect 控制是否尝试重连，reconnect 默认为 false。当设置 reconnect 为 false 时，如果连接不可用会引发一个错误；当设置 reconnect 为 true 并且检查到连接不可用时，会尝试重连数据库，如果重连失败会引发一个错误。

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('SYSDBA/Dmsys_123')

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：

- dmPython.DSQL_TRUE：压缩
- dmPython.DSQL_FALSE：不压缩

例如：

```
import dmPython

conn = dmPython.connect('SYSDBA/Dmsys_123')

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`，仅能在创建连接时通过关键字 `rwseparate`、`rwseparate_percent` 进行设置。这两个属性在连接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，读写分离的具体介绍可参考《DM8 数据守护与读写分离集群》。

`Connection.rwseparate` 可设置的属性：

- `dmPython.DSQL_RWSEPARATE_ON`：开启读写分离
- `dmPython.DSQL_RWSEPARATE_OFF`：关闭读写分离

例如：

```
>>> import dmPython

>>> dmPython.DSQL_RWSEPARATE_ON
1

>>> dmPython.DSQL_RWSEPARATE_OFF
0

>>> conn = dmPython.connect('SYSDBA/Dmsys_123')

>>> 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(user='SYSDBA', password='Dmsys_123', server='RW',
autoCommit=True, rwseparate=dmPython.DSQL_RWSEPARATE_OFF, rwseparate_percent=50)

>>> conn2.rwseparate

0

>>> conn2 = dmPython.connect(user='SYSDBA', password='Dmsys_123', server='RW',
autoCommit=True, rwseparate=dmPython.DSQL_RWSEPARATE_ON, rwseparate_percent=50)

>>> conn2.rwseparate

1

>>> conn2.rwseparate_percent

50
```

上述示例中，服务名“RW”对应数据守护环境的连接串。开启读写分离时，也可通过 ip:port 的方式直连主库，但不能通过 ip:port 的方式直连备库或单机。

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('SYSDBA/Dmsys_123')

>>> 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('SYSDBA/Dmsys_123')

>>> 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。

设置 lang_id 为 LANGUAGE_CN 时，需要保证 local_code 等于 PG_UTF8、PG_GBK、PG_BIG5 或者 PG_GB18030。

设置 lang_id 为 LANGUAGE_CNT_HK 时，需要保证 local_code 等于 PG_UTF8、PG_BIG5 或者 PG_GB18030。

例如：

```
>>> import dmPython

>>> conn = dmPython.connect('SYSDBA/Dmsys_123')

>>> conn.lang_id

0

>>> conn.DSQL_ATTR_LANG_ID

0

>>> dmPython.LANGUAGE_EN

1

>>> dmPython.LANGUAGE_CN

0

>>> dmPython.LANGUAGE_CNT_HK

2
```



```
>>> conn2 = dmPython.connect(lang_id=dmPython.LANGUAGE_EN)

>>> conn2.lang_id

1

>>> conn2.DSQL_ATTR_LANG_ID

1

>>> conn3 = dmPython.connect(lang_id=dmPython.LANGUAGE_CNT_HK, local_code =
dmPython.PG_UTF8)

>>> conn3.lang_id

2

>>> conn3.DSQL_ATTR_LANG_ID

2

>>>
```

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('SYSDBA/Dmsys_123')

>>> 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('SYSDBA/Dmsys_123')  
  
>>> 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('SYSDBA/Dmsys_123')  
  
>>> 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

执行超时时间，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('SYSDBA/Dmsys_123')

>>> 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，以毫秒为单位，对应 DPI 属性 DSQL_ATTR_LOGIN_TIMEOUT。

例如：

```
>>> import dmPython

>>> conn = dmPython.connect('SYSDBA/Dmsys_123')

>>> conn.login_timeout

5000

>>> conn.DSQL_ATTR_LOGIN_TIMEOUT

5000

>>> conn.login_timeout = 3

>>> conn.DSQL_ATTR_LOGIN_TIMEOUT

3000
```

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='Dmsys_123')

>>> 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

登录服务器的主机，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('SYSDBA/Dmsys_123')

>>> 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('SYSDBA/Dmsys_123')

>>> 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('SYSDBA/Dmsys_123')

>>> 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，名称大小写敏感，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，名称大小写敏感，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。以下操作不允许 prepare 执行：修改数据库状态、修改数据库模式、修改数据库归档模式、修改数据库归档

配置、数据库归档文件切换。

例如：

```
>>> import dmPython

>>> conn = dmPython.connect('SYSDBA/Dmsys_123')

>>> 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。

以下操作不允许 prepare 执行：修改数据库状态、修改数据库模式、修改数据库归档模式、修改数据库归档配置、数据库归档文件切换。

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。在内存允许范围内，对批量绑定的参数无数量限制；若内存不足，则报错。

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 dict 或 tuple 类型。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。返回类型由各行数据的 dict 或 tuple 类型组成的 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('SYSDBA/Dmsys_123')

>>> 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 dict 或 tuple 类型组成的 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('SYSDBA/Dmsys_123')

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。

3.3.1.14 Cursor.var

语法:

```
Cursor.var(typ[, size, arraysize, inconverter, outconverter, typename, encoding_errors,
bypass_decode])
```

说明:

根据指定特征创建变量。

typ: 指定变量或变量中元素的 Python 类型。

size: 当变量或变量中元素为 str 或 bytes 类型时, 指定其数据长度。str 类型默认长度为 4094 字节, bytes 类型默认长度为 8188 字节。其他类型忽略该参数。

arraysize: 指定变量中元素的个数。

其他参数暂不支持, 仅作兼容使用。

Cursor.var 支持的 Python 类型及其对应的数据库类型如下表所示:

表 3.5 Cursor.var 支持的类型列表

Python 类型	数据库类型
bool	dmPython.BOOLEAN
bytes	dmPython.BINARY
datetime.date	dmPython.DATE
datetime.datetime	dmPython.TIMESTAMP
datetime.timedelta	dmPython.INTERVAL
decimal.Decimal	dmPython.DECIMAL
float	dmPython.REAL
int	dmPython.BIGINT
str	dmPython.STRING

3.3.1.15 Cursor.__enter__

语法:

```
Cursor.__enter__()
```

说明:

返回当前 Cursor 对象。__enter__ 是上下文管理器的一部分, 用来在进入 with 语句块时执行获取资源操作, 无需手动调用。

3.3.1.16 Cursor.__exit__

语法:

`Cursor.__exit__()`

说明：

关闭 Cursor 对象。__exit__ 是上下文管理器的一部分，用来在离开 with 语句块时执行清理操作，无需手动调用。

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('SYSDBA/Dmsys_123')

>>> 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.DmdbContext 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('SYSDBA/Dmsys_123')

>>> 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 操作可以查询到 lastrowid 值，其他操作返回 None。

例如：

```
>>> import dmPython
```

```
>>> conn = dmPython.connect('SYSDBA/Dmsys_123')
```

```
>>> 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
```

```
'AAAAAAAAAAAAAAAAAAB'
```

3.3.2.6 Cursor.execid

服务器 sql 语句的执行号。

例如：

```
>>> import dmPython

>>> conn = dmPython.connect('SYSDBA/Dmsys_123')

>>> cursor = conn.cursor()

>>> cursor.execute('select 1 from dual')

>>> cursor.execid

500

>>> cursor.execute('select 2 from dual')

>>> cursor.execid

501
```

3.3.2.7 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>
```

3.3.2.8 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.DmdbContext on <dmPython.Connection to SYSDBA@localhost:5236>>

>>> print (cursor.description)

[(('C1', <class 'dmPython.NUMBER'>, 11, 10, 10, 0, 1), ('C2', <class 'dmPython.STRING'>, 8188, 8188, 8188, 0, 1))]
```

3.3.2.9 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.DmdbContext on <dmPython.Connection to SYSDBA@localhost:5236>>

>>> print (cursor.column_names)

['C1', 'C2']
```

3.3.2.10 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('SYSDBA/Dmsys_123')

>>> cursor = conn.cursor()

>>> cursor.execute('insert into t1 values(3)')

>>> cursor.rownumber

-1
```

```
>>> cursor.rowcount
1
>>> cursor.execute('select * from t1')
<builtins.DmdbContext 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
```

3.3.2.11 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。

3.3.2.12 Cursor.output_stream

输出参数是否可以返回多行。默认值为 0（不可以返回多行）。

由于 dmPython 查询语句的结果集是在用户手动调用 fetchone 或者 fetchall 函数时才进行处理的，目前无法先处理结果集，再处理输出参数。所以当 output_stream=1 并且语句中有输出参数时，会默认抛弃语句的查询结果集。

例如：

```
>>> import dmPython

>>> conn = dmPython.connect('SYSDBA/Dmsys_123')

>>> cursor = conn.cursor()

>>> cursor.execute("drop table if exists test_output;")

>>> cursor.execute("create table test_output(id int,name varchar(100));")

>>> cursor.output_stream = 1

>>> cursor.execute("insert into test_output values(1,'qwe'),(2,'asd'),(3,'zxc') return id ,name
into ?,?",[None,None])

[[1, 2, 3], ['qwe', 'asd', 'zxc']]
```

3.4 大字段

大字段的查询、插入、更新以及绑定参数等操作与字符串一样，都是利用 cursor.execute 接口。DM 大字段数据类型最大支持长度为 100G-1 字节，超过该长度则报错。

Python 版本 3.0 及以上，BLOB 类型数据查询结果为 bytes 类型数据，CLOB 类型数据查询结果为 str 类型数据。

Python 版本 3.0 及以下，BLOB 类型数据查询结果为 str 类型数据，CLOB 类型数据查询结果为 unicode 类型数据。

3.4.1 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('SYSDBA/Dmsys_123')

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

if sys.version_info[0] >= 3 :

    type(blob)

    type(clob)

blob

clob
```

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('SYSDBA/Dmsys_123')

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 3.6 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 3.7 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： <pre>>>> obj.type.attributes [<dmPython.ObjectAttribute <class 'dmPython.VARCHAR'>>, <dmPython.ObjectAttribute <class 'dmPython.VARCHAR'>>, <dmPython.ObjectAttribute <class 'dmPython.NUMBER</pre> <pre>'>>]</pre>

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

表 3.8 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： <pre>>>> obj.type.attributes[0].type <dmPython.ObjectType <class</pre>

		'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('SYSDBA/Dmsys_123')

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

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('SYSDBA/Dmsys_123')

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('SYSDBA/Dmsys_123')

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('SYSDBA/Dmsys_123')

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('SYSDBA/Dmsys_123')

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 dmDjango 驱动

4.1 简介及安装

Django是基于Python的Web应用程序框架，dmDjango是DM提供的Django连接DM数据库的驱动，两者的版本对应关系如下：

表 4.1 dmDjango 版本对应表

dmDjango 版本	Django 版本
dmDjango2.0.x	Django1.9 、 Django1.10 、 Django1.11 、 Django2.0 、 Django2.1 、 Django2.2、 Django3.0
dmDjango3.0.x	Django3.1、 Django3.2、 Django4.0、 Django4.1及以上版本

dmDjango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。

可以使用如下命令很方便地在 Windows 和 Linux 操作系统下编译并安装 dmDjango：

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

也可以先生成安装文件再进行安装。

在Windows操作系统下生成exe文件的操作如下，之后只需要直接执行exe文件即可安装：

```
//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，生成rpm包的操作如下所示：

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

安装和卸载命令参考如下:

```
安装: rpm -ivh dmDjango-1.0-1.noarch.rpm
```

```
卸载: rpm -e dmDjango-1.0-1.noarch.rpm
```

4.2 举例说明

4.2.1 创建 Django 项目

进入创建项目所在目录, 例如 D:\django_dmPython, 在该目录下使用如下命令创建mysite 项目:

```
django-admin startproject mysite
```

Django将会创建一个mysite目录, 该目录包含mysite文件夹与manage.py文件。

4.2.2 数据库配置

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': 'dmDjango',  
  
        'NAME': 'DAMENG',  
  
        'USER': 'SYSDBA',  
  
        'PASSWORD': 'Dmsys_123',  
  
        '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](#)

4.2.3 创建 dm 应用

在 Django 中, 项目是一个网站使用的配置和应用的集合, 一个项目可以包含很多个应用。例如, 使用如下命令创建一个 dm 应用:

```
cd mysite

python manage.py startapp dm
```

4.2.4 创建模型

编辑 dm\models.py 文件。

```
from django.db import models

# Create your models here.

class Book(models.Model):

    class Meta:

        db_table = 'PRODUCT'

    PRODUCTID = models.AutoField(primary_key=True, db_column='PRODUCTID')

    NAME = models.CharField(max_length=100, null=True)

    AUTHOR = models.CharField(max_length=25, null=True)

    PUBLISHER = models.CharField(max_length=50, null=True)

    PUBLISHTIME = models.DateField(null=True)
```

```
PRODUCTNO = models.CharField(max_length=25, null=True)

SATETYSTOCKLEVEL = models.IntegerField(null=True)

ORIGINALPRICE = models.DecimalField(max_digits=5, decimal_places=2, null=True)

NOWPRICE = models.DecimalField(max_digits=10, decimal_places=4, null=True)

DISCOUNT = models.DecimalField(max_digits=2, decimal_places=1, null=True)

DESCRIPTION = models.TextField(null=True)

TYPE = models.CharField(max_length=5, null=True)

PAPERTOTAL = models.IntegerField(null=True)

WORDTOTAL = models.IntegerField(null=True)

SELLSTARTTIME = models.DateField(null=True)

SELLENDTIME = models.DateField( null=True)
```

同时需要在 settings.py 文件中的 INSTALLED_APPS 中添加设置'dm.apps.DmConfig', 将 dm 应用安装到 Django 项目里。

4.2.5 进行数据迁移

在 D:\django_dmPython\mysite 目录下打开命令行工具，执行以下命令，为模型的改变生成迁移文件：

```
python manage.py makemigrations dm

python manage.py migrate
```

4.2.6 对数据库进行操作

(1) 在 dm 目录中新建一个 services.py 文件，在该文件中实现增、删、改、查四个基本操作。

```
from .models import Book
```


#查询操作

```
def fun_get_books():  
  
    books = Book.objects.all() #返回 QuerySet  
  
    total_rows=books.count()  
  
    print('查询到'+str(total_rows)+'条记录')  
  
    for book in books:  
  
        print(book.NAME, book.AUTHOR, book.PUBLISHER)  
  
    print("")
```

#增加操作

```
def fun_insert_book():  
  
    NewBook = Book()  
  
    NewBook.NAME = '水浒传'  
  
    NewBook.AUTHOR = '施耐庵, 罗贯中'  
  
    NewBook.PUBLISHER = '中华书局'  
  
    NewBook.PUBLISHTIME = '2005-4-1'  
  
    NewBook.PRODUCTNO = '9787101046137'  
  
    NewBook.SATETYSTOCKLEVEL = '10'  
  
    NewBook.ORIGINALPRICE = '19'  
  
    NewBook.NOWPRICE = '14.3'  
  
    NewBook.DISCOUNT = '7.5'  
  
    NewBook.DESCRPTION = ""
```

《水浒传》是宋江起义故事在民间长期流传基础上产生出来的，吸收了民间文学的营养。

《水浒传》是我国人民最喜爱的古典长篇白话小说之一。它产生于明代，是在宋、元以来有关水浒的故事、话本、戏曲的基础上，由作者加工整理、创作而成的。

全书以宋江领导的农民起义为主要题材，艺术地再现了中国古代人民反抗压迫、英勇斗争的悲壮画卷。

作品充分暴露了封建统治阶级的腐朽和残暴，揭露了当时尖锐对立的社会矛盾和“官逼民反”的残酷现实，成功地塑造了鲁智深、李逵、武松、林冲、阮小七等一批英雄人物。

小说故事情节曲折，语言生动，人物性格鲜明，具有高度的艺术成就。但作品歌颂、美化宋江，鼓吹“忠义”和“替天行道”，表现出严重的思想局限。

```
'''
NewBook.PHOTO = ''

NewBook.TYPE = '16'

NewBook.PAPERTOTAL = '922'

NewBook.WORDTOTAL = '912000'

NewBook.SELLSTARTTIME = '2006-03-20'

NewBook.SELLENDTIME = '2021-07-12'

NewBook.save()

print('插入成功')

#更新操作

def fun_update_book():

    try:

        obj = Book.objects.get(PRODUCTID=1) #返回结果有且只有一个

        obj.NAME='水浒'

        obj.save()

        print('更新成功')

    except Book.DoesNotExist:
```

```
        print('PRODUCTID=1 的记录不存在，未更新任何数据')

#删除操作

def fun_delete_book():

    obj = Book.objects.filter(NAME='水浒') #返回 QuerySet

    del_rows=obj.count()

    obj.delete()

    print('删除'+str(del_rows)+'条记录')
```

(2) 编写 dm\views.py 文件。

```
from django.shortcuts import render

# Create your views here.

from django.http import HttpResponse

from .services import fun_get_books

from .services import fun_insert_book

from .services import fun_update_book

from .services import fun_delete_book


def index(request):

    # 查询

    fun_get_books()


    # 增加

    fun_insert_book()

    fun_get_books()
```

```
# 更新

fun_update_book()

fun_get_books()


# 删除

fun_delete_book()

fun_get_books()

return HttpResponse("执行结束，请到控制台查看输出结果。")
```

(3) 在 dm 目录里新建 urls.py 文件。

```
from . import views

from django.urls import path

urlpatterns = [

    path("", views.index, name='index'),

]
```

(4) 在 mysite\urls.py 文件的 urlpatterns 列表中指定创建的 dm.urls 模块。

```
from django.contrib import admin

from django.urls import path, include

urlpatterns = [

    path('dm/', include('dm.urls')),

    path('admin/', admin.site.urls),

]
```

(5) 在命令行工具中输入以下命令，启动 Django 自带的简易服务器：

```
python manage.py runserver
```

默认使用浏览器访问 <http://127.0.0.1:8000/dm/>，即可查看输出结果。

4.3 类型映射

Django 中的列类型通常使用 Field 指定，与数据库中的类型进行对应映射，dmDjango 支持当前 Django 的类型，类型映射对照关系如下：

表 4.2 dmDjango 类型映射对照关系

Django 类型	数据库类型
SmallAutoField	INTEGER IDENTITY(1,1)
AutoField	INTEGER IDENTITY(1,1)
BigAutoField	BIGINT IDENTITY(1,1)
BinaryField	BLOB
BooleanField	TINYINT
CharField	VARCHAR(max_length)
CommaSeparatedIntegerField	VARCHAR(max_length)
DateField	DATE
DateTimeField	TIMESTAMP
DecimalField	NUMBER(max_digits, decimal_places)
DurationField	INTERVAL DAY(9) TO SECOND(6)
FileField	VARCHAR(max_length)
FilePathField	VARCHAR(max_length)
FloatField	DOUBLE PRECISION
IntegerField	INTEGER

INTEGER	BIGINT
IPAddressField	VARCHAR(15)
GenericIPAddressField	VARCHAR(39)
NullBooleanField	TINYINT
OneToOneField	INTEGER
PositiveIntegerField	INTEGER
PositiveBigIntegerField	BIGINT
PositiveSmallIntegerField	SMALLINT
SlugField	VARCHAR(max_length)
SmallIntegerField	SMALLINT
TextField	TEXT
TimeField	TIMESTAMP
URLField	VARCHAR(max_length)
UUIDField	VARCHAR(32)
JSONField	JSON

4.4 特殊说明

1. 按照 Django 规范，使用 dmDjango 创建的用户不会默认赋予 SOI 角色权限，因此新创建的用户默认无法访问 SYS 模式下的系统表。

5 dmSQLAlchemy 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。dmSQLAlchemy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.dmSQLAlchemy方言包的软件生成与安装。

dmSQLAlchemy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\dmSQLAlchemy\dmSQLAlchemyx.x` 目录中。

可以使用如下命令很方便地在 Windows 和 Linux 操作系统下编译并安装 dmSQLAlchemy:

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

```
python setup.py install
```

也可以先生成安装文件再进行安装, 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dmSQLAlchemy-1.1.10.win-amd64.exe) 位于 `drivers\python\dmSQLAlchemy\dmSQLAlchemyx.x\dist` 目录中。点击安装包安装即可。

DM提供的dmSQLAlchemy与SQLAlchemy的版本对应关系如下表:

表 5.1 dmSQLAlchemy 版本对应表

dmSQLAlchemy版本	SQLAlchemy版本
----------------	--------------

1.1.x	1.3.x
1.4.x	1.4.x
2.0.x	2.0.x

5.2engine 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:Dmsys_123@localhost:5236/',connect_args={'local_code':1,'co
nnection_timeout':15})

或

engine =

create_engine('dm+dmPython://SYSDBA:Dmsys_123@localhost:5236/',connect_args={'local
_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。

5.3类型映射

SQLAlchemy支持了数据库中使用的大部分类型，根据映射关系，`dmSQLAlchemy`方言包支持达梦数据库的下列类型：

ARRAY, BFILE, BIGINT, BINARY, BIT, BLOB, CHAR, CHARACTER, CLOB, DATE, DATETIME, DEC, DECIMAL, DOUBLE, DOUBLE PRECISION, FLOAT, JSON, JSONB, INT,

INTEGER, INTERVAL YEAR, INTERVAL MONTH, INTERVAL DAY, INTERVAL HOUR, INTERVAL MINUTE, INTERVAL SECOND, INTERVAL YEAR TO MONTH, INTERVAL DAY TO HOUR, INTERVAL DAY TO MINUTE, INTERVAL DAY TO SECOND, INTERVAL HOUR TO MINUTE, INTERVAL HOUR TO SECOND, INTERVAL MINUTE TO SECOND, IMAGE, ROWID, LONG, NCLOB, NUMBER, NUMERIC, NVARCHAR2, REAL, SMALLINT, TEXT, TIME, TIME WITH TIME ZONE, TIMESTAMP, TIMESTAMP WITH LOCAL TIME ZONE, TIMESTAMP WITH TIME ZONE, TINYINT, VARBINARY, VARCHAR, VARCHAR2。

其中由于SQLAlchemy类型支持问题，DECIMAL，DEC，NUMBER与NUMERIC类型在SQLAlchemy中均被映射为NUMBER类型，时间间隔数据类型（INTERVAL YEAR等）均被映射为INTERVAL类型，CHARACTER类型被映射为CHAR类型，TINYINT类型被映射为SMALLINT类型，CLOB类型与LONGVARCHAR类型被映射为TEXT类型TIME WITH TIME ZONE类型被映射为TIME类型。

5.4特殊说明

- 1、dmSQLAlchemy当前版本，主键integer primary_key不会再自动形成自增列了。
- 2、只有当dmPython版本大于2.5.9时才可以正常使用executemany返回多行数据集。
- 3、由于表名标准化问题，克隆或迁移表时，全大写表名将无法识别，请避免使用此类表名。
- 4、由于当python中的True或False作为JSON类型的值时，选取时使用的json_value函数在匹配时如果期望匹配True或False需要使用字符串'true'或'false'匹配。
- 5、由于SQLAlchemy对于类型返回值的检测问题，导致JSON类型无法返回为正常的dict，因此暂时将从数据库中查询到的JSON或者JSONB类型列的值映射为字符串类型返回。

5.5暂不支持功能

1、由于数据库语法暂不支持LATERAL功能，因此在dmSQLAlchemy中当前暂不支持使用SQLAlchemy提供的lateral方法构建子查询语句。

2、由于语法限制暂不支持使用bulk_insert_mappings, supports_default_values, supports_empty_insert等空值插入方法。

6 DBUtils 包

6.1 简介

DBUtils 是一套用于管理数据库连接池的Python包，可以自动管理连接对象的创建和释放。并允许对非线程安全的数据库接口进行线程安全包装。DBUtils可为高频度高并发的数据库访问提供更好的性能。

DBUtils提供两种外部接口：PersistentDB和PooledDB。

PersistentDB提供线程专用的数据库连接，并自动管理连接。为每个线程创建一个连接，线程即使调用了close方法也不会关闭连接，只是把连接重新放到连接池，供自己线程再次使用。当线程终止时，连接自动关闭。

PooledDB提供线程间可共享的数据库连接，并自动管理连接。创建一批连接并放到连接池中，供所有线程共享使用。

PersistentDB和PooledDB都是为了重用数据库连接来提高性能，并保持数据库的稳定性。PersistentDB将会保持一定数量的连接供频繁使用，在这种情况下总是保持固定数量的连接。如果应用程序频繁的启动和关闭线程，最好使用PooledDB。

6.2 使用方法

dmpython支持DBUtils1.3-DBUtils3.02之间的版本。其中，DBUtils1.3-DBUtils2.0版本和DBUtils2.0以上-DBUtils3.02版本的导入模块用法略有不同，具体的细节在下面的章节中会进行介绍。

DBUtils的用法分为两步：一是创建连接池；二是从连接池中获取连接。

下面对PersistentDB和PooledDB分别进行介绍。

6.2.1 PersistentDB

— 使用 PersistentDB创建连接池。

```
import dmPython

#如果是 DBUtils 1.3 -DBUtils 2.0 版本使用如下语句导入模块

from DBUtils.PersistentDB import PersistentDB

#如果是 DBUtils2.0 以上版本，则将上面一句替换为下面的语句

#from dbutils.persistent_db import PersistentDB

POOL = PersistentDB(

    creator=dmPython, # 使用链接数据库的模块

    maxusage=None, # 一个链接最多被重复使用的次数，None 表示无限制

    session=[], # 开始会话前执行的命令列表。如: ["set datestyle to ...", "set time zone ..."]

    ping=0,

    # ping DM 服务端，检查是否服务可用。# 如: 0 = None = never, 1 = default = whenever it is
    requested, 2 = when a cursor is created, 7 = always

    closeable=False,

    # 如果为 False 时， conn.close() 实际上被忽略，供下次使用，再线程关闭时，才会自动关闭链接。
    # 如果为 True 时， conn.close()则关闭链接，那么再次调用 pool.connection 时就会报错，因为已经真的
    关闭了连接 (pool.steady_connection()可以获取一个新的链接)

    threadlocal=None, # 表示 thread-local 数据的类。

    host='127.0.0.1', # 主机号

    port=5236, # 端口号

    user='SYSDBA', # 用户名

    password='Dmsys_123', # 密码
```

)

二 从连接池获取连接的方法。

获取连接的方法为 `conn = POOL.connection()`。

获取到的连接的使用方法可以参考dmPython使用手册中Connection接口的使用方法，

具体操作实例如下：

```
def func():

    conn = POOL.connection()

    cursor = conn.cursor()

    cursor.execute('select * from tbl')

    result = cursor.fetchall()

    cursor.close()

    conn.close()
```

6.2.2 PooledDB

一 使用PooledDB创建连接池。

```
import dmPython

#如果是 DBUtils 1.3 -DBUtils 2.0 版本使用如下语句导入模块

from DBUtils.PooledDB import PooledDB

#如果是 DBUtils2.0 以上版本，则将上面一句替换为下面的语句

#from dbutils.pooled_db import PooledDB

POOL = PooledDB(

    creator=dmPython, # 使用链接数据库的模块

    maxconnections=6, # 连接池允许的最大连接数，0 和 None 表示不限制连接数

    mincached=2, # 初始化时，连接池中至少创建的空闲的连接，0 表示不创建
```

```
maxcached=5, # 链接池中最多闲置的链接, 0 和 None 不限制

maxshared=3, # 链接池中最多共享的链接数量, 0 和 None 表示全部共享

blocking=True, # 连接池中如果没有可用连接后, 是否阻塞等待。True, 等待; False, 不等待然后报错

maxusage=None, # 一个链接最多被重复使用的次数, None 表示无限制

setsession=[], # 开始会话前执行的命令列表。如: ["set datestyle to ...", "set time zone ..."]

ping=0,

# pingDM 服务端, 检查是否服务可用。# 如: 0 = None = never, 1 = default = whenever it is
requested, 2 = when a cursor is created, 7 = always

host='127.0.0.1', # 主机号

port=5236, # 端口号

user='SYSDBA', # 用户名

password='Dmsys_123', # 密码

)
```

二 从连接池获取连接的方法。

获取连接的方法为 `conn=POOL.connection()`。具体用法和 `PersistentDB` 一样。

手册版本

版本号：V8

发版日期：2025年4月

咨询热线：400-991-6599

技术支持：dmtech@dameng.com

官网网址：www.dameng.com



武汉达梦数据库股份有限公司
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
