Python API User Manual

DolphinDB

Contents

Chapter 1. 快速上手 5

1.1 介绍 5

如何安装 DolphinDB Python API 6

DolphinDB Python API 快速上手指南 7

DolphinDB Python API 的常用操作 14

Chapter 2. 基础操作 21

如何构建 Session 21

如何使用 Connect 方法建立连接 26

Session 相关的常用方法 29

构造 DBConnectionPool 39

方法介绍 42

TableAppender 50

TableUpserter 53

PartitionedTableAppender 57

订阅 59

Session 异步提交 63

MultithreadedTableWriter 66

BatchTableWriter 80

Chapter 3. 进阶操作 85

PROTOCOL_DDB 85

PROTOCOL_PICKLE 102

PROTOCOL_ARROW 105

强制类型转换 108

类型转换 109

多种写入方案 111

流订阅模式 116

3.4.1 Table 123

3.4.2 Database 142

3.5.1 强制终止进程 148

Chapter 4. 参考阅读 151

4.1 流数据应用 151

4.2 动量交易策略 153

4.3 时间序列计算 155

Index ..... a

Glossary ii

Chapter 1. 快速上手

1.1 介绍

如何安装 DolphinDB Python API

DolphinDB Python API 快速上手指南

DolphinDB Python API 的常用操作

本章介绍如何以相对便捷的方式对 DolphinDB 开发的 Python API 快速上手。

1.1 介绍

dolphindb 是 DolphinDB 数据库的 Python API，主要实现了 Python 客户端和 DolphinDB 服务端之间的消息传递和数据转换协议，允许用户在 Python 环境中与 DolphinDB 数据库建立连接，并实现数据的双向传输和脚本执行。该 Python 库具有以下特点：

1. 支持 Python 3.6 - 3.10
2. 支持 Linux(x86_64, arm)、Windows、MacOS(arm64, x86_64) 平台
3. 提供多种接口函数，包括连接 DolphinDB 服务器、执行 DolphinDB 脚本、发送 DolphinDB 消息等。
4. 使用 Pybind11 编写 C++ 库，提供后台多线程处理，优化数据交互性能。
5. 支持数据的批量处理和异步执行。
6. 支持多种数据类型交互，例如 pandas.DataFrame、arrow.Table。

用户可以使用 DolphinDB Python API 在 Python 环境中对 DolphinDB 数据库进行数据处理、分析和建模等操作，同时还可以利用 DolphinDB 的高性能计算和存储能力来加速数据处理和分析，优化整体业务性能。

dolphindb 是 DolphinDB 数据库的官方 Python API。它提供了多种接口函数，例如连接 DolphinDB 服务器、执行 DolphinDB 脚本、发送 DolphinDB 消息等，以实现 Python 客户端和 DolphinDB 服务端之间的消息传递和数据转换协议，允许用户在 Python 环境中与 DolphinDB 数据库建立连接，并实现数据的双向传输和脚本执行。

该库支持 Linux(x86_64, arm)、Windows、MacOS(arm64, x86_64) 平台和 Python 3.6 - 3.10 版本。使用 Pybind11 编写 C++ 库，该库提供后台多线程处理和数据交互性能的优化。此外，DolphinDB Python API 还支持数据的批量处理和异步执行，并支持多种数据类型交互，例如 pandas.DataFrame、arrow.Table。

用户可以利用 DolphinDB Python API 在 Python 环境中对 DolphinDB 数据库进行数据处理、分析和建模等操作，同时还可以利用 DolphinDB 的高性能计算和存储能力来加速数据处理和分析，优化整体业务性能。

如何安装 DolphinDB Python API

在安装 DolphinDB Python API 前，请确定已部署 Python 执行环境。若无，推荐使用 Anaconda Distribution 下载 Python 及常用库。

目前仅支持通过 `pip` 指令安装 DolphinDB Python API，暂不支持 `conda` 指令。安装示例如下：

```
$ pip install dolphindb
```

支持版本

DolphinDB Python API 的详细版本列表和离线下载链接请参照 pypi dolphindb。下表展示 DolphinDB Python API (最新发布版本：1.30.21.1) 在不同操作系统中对应支持的 Python 版本号。

操作系统	Python 版本号
Windows(amd64)	Python 3.6-3.10
Linux(x86_64)	Python 3.6-3.10
Linux(aarch64)	Conda 环境下的 Python 3.7-3.10
Mac(x86_64)	Conda 环境下的 Python 3.6-3.10
Mac(arm64)	Conda 环境下的 Python 3.8-3.10

注意：为保证正常使用 DolphinDB Python API，您需要同时安装以下依赖库：

- future
- NumPy（建议使用版本 1.18~1.23.4）
- pandas（建议使用 0.25.1及以上版本，暂不支持版本 1.3.0）

常见问题

Q1：安装失败，或安装成功后无法导入 dolphindb 包。

建议检查并重装 whl 包。具体操作如下：

1. 通过 PyPI 确认是否存在支持当前操作系统（例如 Linux arm 架构、Mac M1 等）的 DolphinDB API 安装包。若存在，则将该 whl 包下载至本地。
2. 通过如下命令查看适合当前系统环境支持的 whl 包后缀。

```
pip debug --verbose
```

3. 根据 Compatible tags 的显示信息，将 DolphinDB 的 whl 包名修改为适合系统架构的名称。以 Mac(x86_64) 系统为例：安装包名为 "dolphindb-1.30.19.2-cp37-cp37m-macosx_10_16_x86_64.whl"。但查询到 pip 支持的当前系统版本为10.13，则将 whl 包名中的“10_16”替换成“10_13”。
4. 尝试安装更名后的 whl 包。

若执行完上述操作后，仍无法安装或导入，可在 DolphinDB 社区中进行反馈。

DolphinDB Python API 快速上手指南

本节将展示通过 DolphinDB Python API 连接、使用及维护单节点 DolphinDB 服务器的完整操作。

通过阅读，您将了解到如何使用 DolphinDB Python API 连接 DolphinDB 并与之交互、以及如何维护 DolphinDB 服务器。

1. 使用前准备

1.1 Python 基础知识

在学习使用 DolphinDB Python API 前，请确保您已掌握 Python 的基本知识。具体内容可参考 Python 官方教程。

1.2 搭建 DolphinDB 服务器

在 DolphinDB 官网下载 DolphinDB 服务器，并参考单节点部署教程启动 DolphinDB 服务。

有关快速部署 DolphinDB 服务器, 请参考用新服务器从零开始部署 DolphinDB。

2. 建立连接

Session（会话控制）可以实现客户端与服务器之间的信息交互。DolphinDB Python API 通过 Session 在 DolphinDB 服务器上执行脚本和函数，同时实现双向的数据传递。

注： DolphinDB Python API 自1.30.22.1版本调整 session 类名为 Session，同时增加别名 session 以确保兼容性。

Session 连接及使用示例

在如下脚本中，先通过 `import` 语句导入 dolphindb ，在 Python 中创建一个 Session，然后使用指定的域名（或 IP 地址）和端口号把该会话连接到 DolphinDB 服务器。最后展示一个简单的计算示例，执行 `1+1` 的脚本，得到返回值 `2`。

```
>>> import dolphindb as ddb
>>> s = ddb.Session()
>>> s.connect("localhost", 8848)
True
>>> s.run("1+1;")
2
>>> s.close()
```

注意：

- 建立连接前，须先启动 DolphinDB 服务器。
- 若当前 Session 不再使用，建议立即调用 `close()` 关闭会话。否则可能出现因连接数过多，导致其它会话无法连接服务器的情况。

3. 数据交互

DolphinDB Python API 目前支持多种数据交互的方法，本小节仅介绍xxxx，更多交互方式可参考xxx章节。

下表为 DolphinDB 的各种数据形式与 Python 对象的对应关系。

DolphinDB DataForm	Python
Scalar	int/str/float/...
Vector	numpy.ndarray
Pair	list
Matrix	
Set	set
Dictionary	dict
Table	pandas.Data-Frame

注意：

- 由于 DolphinDB 中的 Matrix 对象可以设置行名和列名，故下载的 Matrix 形式的数据将转换成 list 类型的 Python 对象。该 list 对象中将包含一个表示数据的二维数组和两个分别表示行名、列名的一维数组。
- 若 API 使用不同的下载方式，同样的 DolphinDB 数据将会对应不同的 Python 对象。详情请参考章节x。

3.1 下载数据

在下载数据前，首先导入 dolphindb，然后创建一个 Session 并连接到 DolphinDB 服务端。

```
>>> import dolphindb as ddb
>>> s = ddb.Session()
>>> s.connect("localhost", 8848)
True
```

使用 `s.run` 的方式直接下载数据。成功执行后将返回具体数据。

以下依次创建并下载数据形式分别为标量（Scalar）、向量（Vector）、数据对（Pair）、矩阵（Matrix）、集合（Set）、字典（Dictionary）和表（Table）的数据。

标量（Scalar）

```
>>> s.run("1;")
1
```

向量（Vector）

```
>>> s.run("1..10;")
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10], dtype=int32)
```

数据对（Pair）

```
>>> s.run("1:5;")
[1, 5]
```

矩阵（Matrix）

```
>>> s.run("1..6$2:3;")
[array([[1, 3, 5],
        [2, 4, 6]], dtype=int32), None, None]
```

集合（Set）

```
>>> s.run("set([1,5,9]);")
{1, 5, 9}
```

字典（Dictionary）

```
>>> s.run("dict(1 2 3, 4.5 7.8 4.3);")
{2: 7.8, 1: 4.5, 3: 4.3}
```

表（Table）

```
>>> s.run("table(`XOM`GS`AAPL as id, 102.1 33.4 73.6 as x);")
     id      x
0   XOM  102.1
1    GS   33.4
2  AAPL   73.6
```

3.2 上传数据

同样地，在上传数据前，确定已先导入 dolphindb，创建一个 Session 并连接到 DolphinDB 服务端。此外，导入库 numpy 和 pandas，以完整代码示例如下：

```
>>> import dolphindb as ddb
>>> s = ddb.Session()
>>> s.connect("localhost", 8848)
True
>>> import numpy as np
>>> import pandas as pd
```

通过 `s.upload` 的方式上传数据，输入参数`变量名称: 数据`，成功执行后将返回该变量在服务器端的地址。

以下将依次展示上传 DolphinDB 数据类型为标量（Scalar）、向量（Vector）、矩阵（Matrix）、集合（Set）、字典（Dictionary）和表（Table）的数据。注意，Python API 暂不支持上传数据对（Pair）形式的数据。

成功上传后，脚本中将展示通过 `typestr` 函数查询数据的 Python 类型名称，以及使用 `s.run` 的方式重新下载数据。

标量（Scalar）

```
>>> s.upload({'scalar_sample': 1})
62776640
>>> s.run("typestr(scalar_sample);")
'LONG'
>>> s.run("scalar_sample;")
1
```

向量（Vector）

```
>>> s.upload({'vector_sample': np.array([1, 3])})
65583680
>>> s.run("typestr(vector_sample);")
'FAST LONG VECTOR'
>>> s.run("vector_sample;")
array([1, 3])
```

矩阵（Matrix）

```
>>> s.upload({'matrix_sample': np.array([[1, 2, 3], [4, 5, 6]])})
65484832
>>> s.run("typestr(matrix_sample);")
'FAST LONG MATRIX'
>>> s.run("matrix_sample;")
[array([[1, 2, 3],
       [4, 5, 6]]), None, None]
```

集合（Set）

```
>>> s.upload({'set_sample': {1, 4, 7}})
65578432
>>> s.run("typestr(set_sample);")
'LONG SET'
>>> s.run("set_sample;")
{1, 4, 7}
```

字典（Dictionary）

```
>>> s.upload({'dict_sample': {'a': 1}})
58318576
>>> s.run("typestr(dict_sample);")
'STRING->LONG DICTIONARY'
>>> s.run("dict_sample;")
{'a': 1}
```

表（Table）

```
>>> df = pd.DataFrame({'a': [1, 2, 3], 'b': ['a', 'b', 'c']})
>>> s.upload({'table_sample': df})
63409760
```

```
>>> s.run("typestr(table_sample);")
'IN-MEMORY TABLE'
>>> s.run("table_sample;")
   a  b
0  1  a
1  2  b
2  3  c
```

4. 服务器运维

Python API 支持通过如下两种方式维护 DolphinDB 服务器：

- 方式一：通过 `Session.run()` 执行 DolphinDB 脚本。
- 方式二：使用 Python API 封装的方法操作服务器。

以下展示两种方式的使用示例。

方式一：执行 DolphinDB 脚本

在以下脚本中，先导入 dolphindb，创建一个 Session 并连接到 DolphinDB 服务端。由于后续脚本中涉及到执行数据库相关的操作，须在连接服务端的同时登录具有相应权限的账户。

在完成上述操作后，通过 `Session.run()` （脚本中对应 `s.run()`）执行创建变量、数据库 db 和数据表 pt,并向表 pt 中写入数据等操作，最后执行 SQL 语句返回表 pt 的行数 `1000000`。

```
import dolphindb as ddb
s = ddb.Session()
s.connect("localhost", 8848, "admin", "123456")
s.run("""
    n=1000000
    ID=rand(10, n)
    x=rand(1.0, n)
    t=table(ID, x)
    db=database(directory="dfs://hashdb", partitionType=HASH, partitionScheme=[INT, 2])
    pt = db.createPartitionedTable(t, `pt, `ID)
    pt.append!(t);
""")
re = s.run("select count(x) from pt;")
print(re)
```

```
# output
   count_x
0  1000000
```

方式二：使用 API 接口

在以下脚本中，首先导入 pandas, numpy, dolphindb 和 dolphindb.settings，然后创建一个 Session 并连接到 DolphinDB 服务端，同时登录具有相应权限的账户。

然后创建变量 n 并定义表结构 df。使用 API 接口执行 `s.table()`, `s.database()` 和 `db.createPartitionedTable()` 以创建数据库 db 和数据表 pt， 再通过 `pt.append()` 向表 pt 中写入数据。最后使用 `toDF()` 下载表 t 的数据。

```
import pandas as pd
import numpy as np
import dolphindb as ddb
import dolphindb.settings as keys
s = ddb.Session("192.168.1.113", 8848, "admin", "123456")
n = 1000000
df = pd.DataFrame({
    'ID':   np.random.randint(0, 10, n),
    'x':    np.random.rand(n),
})
t = s.table(data=df)
db = s.database("db", dbPath="dfs://hashdb", partitionType=keys.HASH, partitions=[keys.DT_INT, 2])
pt:ddb.Table = db.createPartitionedTable(table=t, tableName="pt", partitionColumns=["ID"])
pt.append(t)
print(pt.toDF())

# output
        ID         x
0        4  0.320935
1        8  0.426056
2        8  0.505221
3        4  0.692984
4        4  0.709175
...     ..       ...
999995   5  0.479531
999996   3  0.805629
999997   5  0.873164
999998   7  0.158090
```

```
999999   5  0.530824

[1000000 rows x 2 columns]
```

5. 参考链接

- DolphinDB 用户手册-数据形式
- DolphinDB 用户手册-数据库操作

DolphinDB Python API 的常用操作

1. 连接管理

DolphinDB Python API 支持以下两种创建连接的方式：

- 方式一：在构造 Session 的同时传入相应参数，在构造的同时创建连接。注意：如果连接服务器失败，仍能正常创建 Session 对象，只会打印出相关内容，并不会抛出异常。
- 方式二：在构造 Session 后，通过 `connect` 方法建立连接。注意：如果连接成功，则会返回 `True`；否则将返回 `False`。

以下展示创建连接、登录账号和关闭连接的使用示例。

创建连接

以下脚本使用方式一即在构造 Session 的同时创建连接，并演示创建失败和创建成功的例子。

```
>>> import dolphindb as ddb
# 输入错误ip/port，创建连接失败，但仍正常构造s1
>>> s1 = ddb.Session("errorip", 8848)
# 输入正确参数，创建连接成功
>>> s1 = ddb.Session("localhost", 8848)
```

以下脚本使用方式二即在构造 Session 后创建连接，并演示创建失败和创建成功的例子。

```
>>> import dolphindb as ddb
# 构造Session，此时不创建连接
>>> s2 = ddb.Session()
# 输入错误ip/port，connect函数创建连接失败，返回False
>>> s2.connect("localhost", 1234)
False
```

```
# 输入正确参数，connect函数创建连接成功，返回Ture
>>> s2.connect("localhost", 8848)
True
```

登录账号

登录账号需要输入正确的用户名和对应密码。DolphinDB Python API 支持三种登录方式，以下分别展示三种登录方式。

方式一：在构造 Session 时创建连接并登录。

```
>>> s1 = ddb.Session("localhost", 8848, "admin", "123456")
```

方式二：在使用 `connect` 方法建立连接时进行登录。

```
>>> s2 = ddb.Session()
>>> s2.connect("localhost", 8848, "admin", "123456")
True
```

方式三：通过 `login` 方法进行登录。

```
>>> s3 = ddb.Session()
>>> s3.connect("localhost", 8848)
True
>>> s3.login("admin", "123456")
```

关闭连接

DolphinDB Python API 支持调用 `Session.close()` 关闭 Session 连接。当 Session 关闭后，`s.run()` 将无法成功运行脚本，系统会报出异常。以下为示例脚本。

```
>>> s = ddb.Session()
>>> s.connect("localhost", 8848)
True
>>> s.run("1+1;")
2
>>> s.close()
>>> s.run("1+1;")
# RuntimeError: <Exception> in run: Couldn't send script/function to the remote host because the connection has been closed
```

注意：若当前 Session 不再使用，建议立即调用 `Session.close()` 关闭会话。否则可能出现因连接数过多，导致其它会话无法连接服务器的情况。

2. 数据库管理

DolphinDB Python API 中最常用的数据库管理方式是使用 `Session.run()` 方法执行指定脚本。此外，也可以使用 Python API 封装的函数来操作服务器。下表汇总了 Python API 已封装好的服务端管理函数。

函数名	功能	语法	说明
Session.database	创建数据库	`db = s.database(dbName="db", dbPath="dfs://hashdb", partitionType=keys.HASH, partitions=[keys.DT_INT, 2])`	详细使用方式和传入参数可参考 DolphinDB 用户手册-database
Session.existsDatabase	判断是否存在数据库	`s.existsDatabase(path)`	
Session.dropDatabase	根据给定的 dbPath 删除数据库	`s.dropDatabase("dfs://hashdb2")`	
Session.dropPartition	删除指定分区	`s.dropPartition(dbPath="dfs://valuedb", partitionPaths="0", tableName="pt")`	具体参数使用可以参考 DolphinDB 用户手册-dropPartition
Database.createPartitionedTable	在数据库中创建分区表	`db.createPartitionedTable(t, "pt", partitionColumns="ID")`	该方法也需要传入Table对象作为生成表的结构参考。需要传入一个字符串或者字符串列表，用于表示分区列。
Database.createTable	在数据库中创建维度表	`db.createTable(t, "pt")`	传入参数table是一个Python API的Table对象，该对象将作为生成表的结构参考。

3. 数据表操作

函数名	功能	使用示例	说明
Session.existsTable	判断是否存在表	`s.existsTable("dfs://hashdb", "pt")`	
Session.dropTable	删除指定分区数据库中的表	`s.dropTable("dfs://valuedb", "pt")`	

Session.table

获取服务端指定表的句柄，并返回Python Table对象。如果指定数据库路径，也可以获取数据库中表的句柄。

```
tb1 = s.table(dbPath="dfs://valuedb", data="pt")
tb2 = s.table(data="t")
```

也可以将本地数据（字典、DataFrame等）上传到服务端，并保存为指定名字的内存表。

```
dict1 = {
    'a': [1, 2, 3],
    'b': ['a', 'b', 'c'],
}
df = pd.DataFrame(dict1)
tb1 = s.table(data=dict1)
tb2 = s.table(data=df, tableAliasName="tb2")
```

Session.loadText

将数据文件加载到DolphinDB的内存表中，并返回内存表的句柄为Table对象。使用时需要指定远程文件的路径以及文件分隔符。

```
t1 = s.loadText("/home/DolphinDB/Data/stock.csv")
t2 = s.loadText("/home/DolphinDB/Data/stock.csv", "-")
```

注： 加载时使用的路径为DolphinDB服务器所在设备的路径，而非本地文件路径。使用细节请参考DolphinDB用户手册-loadText

Session.ploadText

启动多线程将数据文件加载到DolphinDB的内存表中，使用方法与loadText一致。

```
t = s.ploadText("/home/DolphinDB/Data/stock.csv")
```

4. 一个例子

```
import dolphindb as ddb
import dolphindb.settings as keys
import numpy as np
import pandas as pd

n = 1000
df = pd.DataFrame({
    'ID':   np.random.randint(0, 3, n),
    'x':    np.random.rand(n),
})

if s.existsDatabase("dfs://valuedb"):
```

```
    print("exists dfs://valuedb")
    s.dropDatabase("dfs://valuedb")
db:ddb.Database = s.database(dbName="db", dbPath="dfs://valuedb", partitionType=keys.VALUE, partitions=[0, 1, 2])
t = s.table(data=df)
if not s.existsTable("dfs://valuedb", "pt"):
    pt = db.createPartitionedTable(t, "pt", partitionColumns="ID")
    print("create pt Table")
else:
    raise RuntimeError("dfs://valuedb has table pt.")
pt.append(t)
print(pt.toDF().shape)
s.dropPartition("dfs://valuedb", 0, "pt")
print(pt.toDF().shape)

# output
exists dfs://valuedb
create pt Table
(1000, 2)
(673, 2)
```

5. SQL查询

DolphinDB Python API 支持使用面向对象的 SQL 查询语句。在使用时，须先获得服务端数据表的句柄和 Python 中对应的 Table 对象。获取 Table 对象的方式可以参考数据表操作部分内容。详细使用教程，请参考章节3.4.1。以下为示例脚本。

```
df = pd.DataFrame({
    'ID':   [1, 2, 3, 4, 5, 6, 7, 8, 9, 10],
    'x':    ['a', 'b', 'b', 'c', 'a', 'c', 'a', 'b', 'b', 'a'],
})
t = s.table(data=df)     # Table对象
```

例子

```
res = s.select(["ID", "x"]).where("ID>=5").executeAs("res")
print(res.toDF())

# output
   ID  x
0   5  a
1   6  c
```

```
2   7  a
3   8  b
4   9  b
5  10  a
```

Chapter 2. 基础操作

- 如何构建 Session
- 如何使用 Connect 方法建立连接
- Session 相关的常用方法
- 构造 DBConnectionPool
- 方法介绍
- TableAppender
- TableUpserter
- PartitionedTableAppender
- 订阅
- Session 异步提交
- MultithreadedTableWriter
- BatchTableWriter

如何构建 Session

Session（会话控制）可以实现 API 客户端与 DolphinDB 之间的信息交互。DolphinDB Python API 通过 Session 在 DolphinDB 上执行脚本和函数，同时实现双向的数据传递。

注意： DolphinDB Python API 自1.30.22.1版本起调整 session 类名为 Session，同时增加别名 session 以确保兼容性。

如下展示创建一个 Session 的完整示例。

```
Session(host=None, port=None, userid="", password="",
        enableSSL=False, enableASYNC=False,
        keepAliveTime=30, enableChunkGranularityConfig=False, compress=False,
        enablePickle=None, protocol=PROTOCOL_DEFAULT, python=False)
```

由上述脚本可知，Session 的创建涉及到多个参数。以下内容将对参数进行详细说明。

1. 连接参数 *host*, *port*, *userid*, *password*

- *host* 表示所连接服务器的地址。
- *port* 表示所连接服务器的端口。
- *userid* 表示登录时的用户名。
- *password* 表示登录时用户名对应的密码。

用户可以使用指定的域名（或 IP 地址）和端口号把该会话连接到 DolphinDB，并且在建立连接的同时登录账号。使用示例如下：

```
import dolphindb as ddb

# 创建Session，同时连接地址为localhost，端口为8848的DolphinDB
s = ddb.Session("localhost", 8848)

# 创建Session，同时连接地址为localhost，端口为8848的DolphinDB，登录用户名为admin，密码为123456的账户
s = ddb.Session("localhost", 8848, "admin", "123456")
```

注意：

- 在构造 Session 时，可以不指定参数 *host*, *port*, *userid*, *password*，之后通过 connect 建立连接时再进行指定。
- 如果在构造时输入错误的参数值，将无法连接 DolphinDB，但是仍能正常创建 Session 对象。

2. 加密参数 *enableSSL*

- *enableSSL* 表示是否支持加密通讯，默认值为 False。

API 端设置脚本示例如下：

```
# 开启加密通讯
s = ddb.Session(enableSSL=True)
```

注意：

- DolphinDB 自1.10.17与1.20.6版本起支持加密通讯参数 *enableSSL*。
- DolphinDB 须同时设置配置项 `enableHTTPS=true` 方可启动 SSL 通讯。详情可参考集群通信与连接。

3. 异步参数 *enableASYNC*

- *enableASYNC* 表示是否支持异步通讯，默认值为 False。

开启异步通讯后，API 端只能通过 `Session.run()` 方法与 DolphinDB 端进行通讯，该情况下无返回值。该模式适用于异步写入数据，可节省 API 端检测返回值的时间。使用示例如下：

```
# 开启异步通讯
s = ddb.Session(enableASYNC=True)
```

注意： DolphinDB 自1.10.17与1.20.6版本起支持异步通讯参数 *enableASYNC*。

4. 保活参数 *keepAliveTime*

- *keepAliveTime* 表示在 TCP 连接状态下两次保活传输之间的持续时间，默认参数为60，单位秒（s）。

使用示例如下：

```
# 设置保活时间为120秒
s = ddb.Session(keepAliveTime=120)
```

注意：

- 该参数在 Linux、Windows、MacOS 平台均可生效。
- TCP 超时时间 Timeout 设置可参考章节2.1.3。仅为标记作用，此处需要放链接

5. 压缩参数 *compress*

- *compress* 表示是否开启压缩，默认参数为 False。

该模式适用于大数据量的写入或查询。压缩数据后再传输，这可以节省网络带宽，但会增加 DolphinDB 和 API 端的计算量。使用示例如下：

```
import dolphindb.settings as keys


# api version >= 1.30.21.1，开启压缩
s = ddb.Session(compress=True, protocol=keys.PROTOCOL_DDB)
```

```
# api version <= 1.30.19.4，开启压缩
s = ddb.Session(compress=True, enablePickle=False)
```

注意：

- DolphinDB 自1.30.6版本起支持压缩参数 *compress*。
- 开启压缩时，若 API 为1.30.21.1版本前，须指定 `enablePickle=False`；若 API 为1.30.21.1版本及之后，须指定协议参数 *protocol* 为 PROTOCOL_DDB。（下一小节将介绍协议参数）

6. 协议参数 *protocol*, *enablePickle*

- *protocol* 表示 API 与 DolphinDB 交互时使用的数据格式协议，默认值为 PROTOCOL_DEFAULT，表示使用 PROTOCOL_PICKLE。
- *enablePickle* 表示 API 与 DolphinDB 交互时是否使用 PROTOCOL_PICKLE 作为数据格式协议，默认值为 True。

目前 DolphinDB 支持三种协议：PROTOCOL_DDB, PROTOCOL_PICKLE, PROTOCOL_ARROW。使用不同的协议，会影响 API 在执行 DolphinDB 脚本后接收到的数据格式。有关协议的详细说明请参考章节3.1类型转换。仅为标记作用，此处需要放链接

在1.30.21.1版本前，API 仅支持使用 *enablePickle* 来指定数据格式协议，可设置使用协议 PROTOCOL_PICKLE, PROTOCOL_DDB。使用示例如下：

```
# 使用协议 PROTOCOL_PICKLE
s = ddb.Session(enablePickle=True)


# 使用协议 PROTOCOL_DDB
s = ddb.Session(enablePickle=False)
```

在1.30.21.1版本及之后，API 支持使用 *protocol* 来指定数据格式协议，可设置使用协议 PROTOCOL_DDB, PROTOCOL_PICKLE, PROTOCOL_ARROW。使用示例如下：

```
import dolphindb.settings as keys


# 使用协议 PROTOCOL_DDB
s = ddb.Session(protocol=keys.PROTOCOL_DDB)


# 使用协议 PROTOCOL_PICKLE
s = ddb.Session(protocol=keys.PROTOCOL_PICKLE)


# 使用协议 PROTOCOL_ARROW
s = ddb.Session(protocol=keys.PROTOCOL_ARROW)
```

注意： 在设置 *protocol* 时，建议不要同时设置参数 *enablePickle*，以避免产生冲突。

7. 其他参数 *enableChunkGranularityConfig*

- *enableChunkGranularityConfig* 表示是否支持在使用 `Session.database()` 创建数据库时允许配置 *chunkGranularity* 参数，默认值为 False。

该参数会影响 `Session.database()` 函数的正常使用。Session 中必须指定 `enableChunkGranularityConfig=True`，否则 `Session.database()` 的参数 *chunkGranularity* 将会失效。

在如下脚本中，设置参数 *enableChunkGranularityConfig* 为 True，并展示参数 *chunkGranularity* 已生效：

```
import dolphindb as ddb
import dolphindb.settings as keys

# 设置参数enableChunkGranularityConfig为True，即允许配置Session.database()中的chunkGranularity参数
s = ddb.Session("localhost", 8848, "admin", "123456", enableChunkGranularityConfig=True)

# 以下部分仅为展示参数chunkGranularity已生效
if s.existsDatabase("dfs://testdb"):
    s.dropDatabase("dfs://testdb")
db = s.database("db", partitionType=keys.VALUE, partitions=[1, 2, 3], dbPath="dfs://testdb", chunkGranularity="DATABASE")
print(s.run("schema(db)")["chunkGranularity"])

# 输出结果为 DATABASE
```

在如下脚本中，设置参数 *enableChunkGranularityConfig* 为 False，并展示参数 *chunkGranularity* 已失效：

```
import dolphindb as ddb
import dolphindb.settings as keys

# 设置参数enableChunkGranularityConfig为False，即不允许配置Session.database()中的chunkGranularity参数
s = ddb.Session("localhost", 8848, "admin", "123456", enableChunkGranularityConfig=False)

# 以下部分仅为展示参数chunkGranularity已失效
if s.existsDatabase("dfs://testdb"):
    s.dropDatabase("dfs://testdb")
db = s.database("db", partitionType=keys.VALUE, partitions=[1, 2, 3], dbPath="dfs://testdb", chunkGranularity="TABLE")
print(s.run("schema(db)")["chunkGranularity"])

# 输出结果为 TABLE
```

8. 其他参数 *python*

- *python* 表示是否启用 python parser 特性。

使用示例如下：

```
# 启用 python parser 特性
s = ddb.Session(python=True)


# 不启用 python parser 特性
s = ddb.Session(python=False)
```

注意： 仅支持 DolphinDB 2.10 版本。（暂未发布，敬请期待）

如何使用 Connect 方法建立连接

DolphinDB Python API 支持以下两种创建连接的方式：

- 方式一：在构造 Session 的同时传入相应参数，在构造的同时创建连接。
- 方式二：在构造 Session 后，通过 `connect` 方法建立连接。

本节将介绍方式二即在构造 Session 后通过 `connect` 方法建立连接。如下展示创建一个 Connect 的完整示例：

```
connect(host, port,
        userid=None, password=None, startup=None,
        highAvailability=False, highAvailabilitySites=None,
        keepAliveTime=None, reconnect=False)
```

由上述脚本可知，使用 Connect 方法建立连接涉及到多个参数。以下内容将对参数进行详细说明。

1. 连接参数 *host, port, userid, password*

- *host* 表示所连接服务器的地址。
- *port* 表示所连接服务器的端口。
- *userid* 表示登录时的用户名。
- *password* 表示登录时用户名对应的密码。

用户可以使用指定的域名（或 IP 地址）和端口号把该会话连接到 DolphinDB，并且在建立连接的同时登录账号。使用示例如下：

```
import dolphindb as ddb
s = ddb.Session()


# 连接地址为localhost，端口为8848的DolphinDB
s.connect("localhost", 8848)


# 连接地址为localhost，端口为8848的DolphinDB，登录用户名为admin，密码为123456的账户
s.connect("localhost", 8848, "admin", "123456")
```

若 Session 过期，或者初始化会话时没有指定登录信息（用户名与密码），可使用 `login` 方法登录 DolphinDB。

```
import dolphindb as ddb
s = ddb.Session()
# 连接地址为localhost，端口为8848的DolphinDB，未指定登录信息（用户名与密码）
s.connect("localhost", 8848)
# 使用login函数登录 DolphinDB
s.login("admin","123456")
```

2. 高可用参数 *highAvailability*, *highAvailabilitySites*

- *highAvailability* 表示是否开启 API 高可用，默认值为 False。
- *highAvailabilitySites* 表示所有可用节点的地址和端口，格式为 `ip:port`。

highAvailability 和 *highAvailabilitySites* 都是 API 高可用的相关配置参数。在高可用模式下，Python API 在连接集群节点时会查询负载最小的节点，并与其建立连接。当使用单线程方式有一定延迟地（注1）创建多个 Session 时，Python API 可以保证所有可用节点上连接的负载均衡；但在使用多线程方式同时创建多个 Session 时，由于同时建立连接，每个 Session 建立时查询的最小负载节点可能为同一个，不能保证节点的负载均衡。

下例中为简化问题使用节点连接数代替节点负载：

1. 假如当前三个节点的负载值分别为 ，则单线程依次、有延迟地建立20个连接后，每次建立连接都会向服务器查询当前最小负载的节点，因此最后负载值分别为 。
2. 如果多线程**同时**建立20个连接，或单线程快速创建，则同一时刻服务器查询得到的最小负载节点为同一个，最后负载值分别为 。

节点负载的计算公式为：

```
load = (connectionNum + workerNum + executorNum)/3.0
```

- *connectionNum*：连接到本地节点的连接数。
- *workerNum*：常规作业的工作线程的数量。
- *executorNum*：本地执行线程的数量。

上述变量可在任意节点通过执行 `rpc(getControllerAlias(), getClusterPerf)` 获得，相关函数介绍请参考 DolphinDB 用户手册-getClusterPerf。

若要开启 API 高可用，则需要指定 *highAvailability* 参数为 True，同时通过 *highAvailabilitySites* 指定所有可用节点的地址和端口。示例脚本如下：

```
import dolphindb as ddb
s = ddb.Session()
# 创建向量，包含所有可用节点的地址和端口
sites = ["192.168.1.2:24120", "192.168.1.3:24120", "192.168.1.4:24120"]
# 创建连接；开启高可用，并指定sites为所有可用节点的ip:port
s.connect(host="192.168.1.2", port=24120, userid="admin", password="123456", highAvailability=True, highAvailabilitySites=sites)
```

注1： 如果连续建立多个 Session，服务端集群间可能尚未同步负载信息，此时查询到的结果可能始终为同一个最小负载节点，无法保证节点的负载均衡。

注2： 若开启高可用后不指定 *highAvailabilitySites*，则默认高可用组为集群全部节点。

3. 保活参数 *keepAliveTime*

- *keepAliveTime* 表示在 TCP 连接状态下两次保活传输之间的持续时间，默认参数为60，单位秒（s）。

通过配置 *keepAliveTime* 参数可以设置 TCP 的存活检测机制的检测时长，以实现即便在网络不稳定的条件下，仍可及时释放半打开的 TCP 连接。若不指定保活参数 *keepAliveTime*，则默认使用构造 Session 时使用的 *keepAliveTime* 仅为标注，之后放前一节的链接。指定参数的示例如下：

```
import dolphindb as ddb
s = ddb.Session()
# 创建连接；设置保活时间为120秒
s.connect(keepAliveTime=120)
```

4. 重连参数 *reconnect*

- *reconnect* 表示在不开启高可用的情况下，是否在 API 检测到连接异常时进行重连，默认值为 False。

若开启高可用模式，则 API 在检测到连接异常时将自动进行重连，不需要设置参数 *reconnect*。若未开启高可用，通过配置 `reconnect = True`，即可实现 API 在检测到连接异常时进行重连。使用示例如下：

```
import dolphindb as ddb
s = ddb.Session()
```

```
# 创建连接；开启重连
s.connect(host="localhost", port=8848, reconnect=True)
```

5. 其他参数

- *startup* 表示启动脚本。

该参数可以用于执行一些预加载任务。包含加载插件、加载分布式表、定义并加载流数据表等脚本。

```
import dolphindb as ddb
s = ddb.Session()
# 创建连接；设置启动脚本 "clearAllCache();"
s.connect(host="localhost", port=8848, startup="clearAllCache();")
```

Session 相关的常用方法

本章节将介绍Session中常用的方法。

1. 执行脚本

在Session中执行脚本可以调用Session.run方法，方法接口如下：

```
run(script, *args, **kwargs)
```

1.1 传参

除了运行脚本之外，`run` 命令可以直接在远程 DolphinDB 服务器上执行 DolphinDB 内置或用户自定义函数。对这种用法，`run` 方法的第一个参数是 DolphinDB 中的函数名，之后的参数是该函数的参数。

下面的示例展示 Python 程序通过 `run` 调用 DolphinDB 内置的 `add` 函数。`add` 函数有 x 和 y 两个参数。根据参数是否已在 DolphinDB server 端被赋值，有以下三种调用方式：

- 所有参数均已在 DolphinDB server 端被赋值

若变量 x 和 y 已经通过 Python 程序在 DolphinDB server 端被赋值，

```
>>> s.run("x = [1,3,5];y = [2,4,6]")
```

那么在 Python 端要对这两个向量做加法运算，只需直接使用 `run(script)` 即可：

```
>>> s.run("add(x,y)")
array([3, 7, 11], dtype=int32)
```

- 仅有一个参数 DolphinDB server 端被赋值

若仅变量 x 已通过 Python 程序在服务器端被赋值：

```
>>> s.run("x = [1,3,5];")
```

而参数 y 要在调用 `add` 函数时一并赋值，需要使用 “部分应用” 方式把参数 x 固化在 `add` 函数内。具体请参考 DolphinDB用户手册-部分应用。

```
>>> import numpy as np
>>> y = np.array([1,2,3])
>>> result = s.run("add{x,}", y)
>>> result
array([2,5,8])
>>> result.dtype
dtype('int64')
```

- 两个参数都待由 Python 客户端赋值

```
>>> import numpy as np
>>> x = np.array([1.5,2.5,7])
>>> y = np.array([8.5,7.5,3])
>>> result = s.run("add", x, y)
>>> result
array([10., 10., 10.])
>>> result.dtype
dtype('float64')
```

通过 `run` 调用 DolphinDB 的内置函数时，客户端上传参数的数据结构可以是标量 (scalar)，列表 (list)，字典 (dict)，NumPy 的对象，pandas 的 DataFrame 和 Series 等等。

“

需要注意：

1. NumPy array 的维度不能超过 2。
2. pandas 的 DataFrame 和 Series 若有 index，在上传到 DolphinDB 以后会丢失。如果需要保留 index 列，则需要使用 pandas 的 DataFrame 函数 reset_index。
3. 如果 DolphinDB 函数的参数是时间或日期类型，Python 客户端上传时，参数应该先转换为 numpy.datetime64 类型。

”

1.2 分段读取

对于大数据量的表，API 提供了分段读取方法 (仅适用于 DolphinDB 1.20.5 及以上版本，Python API 1.30.0.6 及以上版本) 在 Python 客户端执行以下代码创建一个大数据量的表：

```
>>> s = ddb.Session()
>>> s.connect("localhost", 8848, "admin", "123456")
True
>>> script = """
... rows=100000;
... testblock=table(take(1,rows) as id,take(`A,rows) as symbol,take(2020.08.01..2020.10.01,rows) as date, rand(50,rows) as
 size,rand(50.5,rows) as price);
... """
>>> s.run(script)
```

在 run 方法中使用参数 fetchSize 指定分段大小，会返回一个 BlockReader 对象，可通过 read() 方法一段段的读取数据。需要注意的是 fetchSize 取值不能小于 8192，示例如下：

```
>>> script1 = "select * from testblock"
>>> block= s.run(script1, fetchSize = 8192)
>>> total = 0
>>> while block.hasNext():
...     tem = block.read()
...     total+=len(tem)
...
>>> total
100000
```

使用上述分段读取的方法时，若数据未读取完毕，需要调用 skipAll 方法来放弃读取后续数据，才能继续执行后续代码。否则会导致套接字缓冲区滞留数据，引发后续数据的反序列化失败。示例代码如下：

```
>>> block= s.run(script1, fetchSize = 8192)
>>> re = block.read()
>>> block.skipAll()
>>> s.run("1+1;") # 若没有调用 skipAll，执行此代码会抛出异常。
2
```

1.3 其他参数

Session.run方法还有以下不定长关键字参数：

- **clearMemory**：使用run方法时，有时候希望 server 能在执行完毕后，自动释放 run 语句中创建的变量，以减少内存占用。这可以通过指定 Session 以及 DBConnectionPool 对象的 run 方法的参数 clearMemory=True 来实现。

```
>>> s.run("t=1", clearMemory = True)
>>> s.run("t")
```

由于t在倒数第二行执行完毕后就被清除了，所以最后一行脚本会抛出异常:

```
RuntimeError: <Exception> in run: Server response: 'Syntax Error: [line #1] Cannot recognize the token t' script: 't'
```

- **pickleTableToList**：当Session构造时指定的protocol为PROTOCOL_DDB或者PROTOCOL_PICKLE。开启该参数后，如果返回值为Table，则对应Python对象为一个numpy.ndarray的列表，列表中每一个元素表示原Table中的一列。有关数据格式的相关内容，请参考章节3.1类型转换。

```
>>> import dolphindb.settings as keys
>>> s = ddb.Session(protocol=keys.PROTOCOL_DDB)
>>> s.connect("localhost", 8848)
True
>>> s.run("table(1..3 as a)")
   a
0  1
1  2
2  3
>>> s.run("table(1..3 as a)", pickleTableToList=True)
[array([1, 2, 3], dtype=int32)]
```

1.4 相关方法

- **runFile**：读取文件所有内容作为脚本执行，可以像run方法一样传入不定长位置参数和不定长关键字参数。**注：** 该文件路径为客户端路径。

```
>>> with open("./test.dos", "w+") as f:
...     f.write("""
...         t = table(1..3 as a);
...         t;
...     """)
...
47
>>> s.runFile("./test.dos")
    a
0   1
1   2
2   3
```

2. 加载数据

2.1 table

```
table(dbPath=None, data=None, tableAliasName=None, inMem=False, partitions=None)
```

- **data**：字符串或字典、DataFrame。如果data为字符串，表示服务端数据表表名；如果data为字典或DataFrame，则表示将本地数据作为临时表上传到服务器。
- **dbPath**：字符串，表示待加载数据表所在的数据库地址。
- **tableAliasName**：用于指定待加载表的别名。
- **inMem**：是否将数据从服务器磁盘加载到服务器内存中。
- **partitions**：表示要加载的分区。

注： table函数在data为字符串时，实际封装了DolphinDB loadTable函数，从指定数据库中加载对应表，并获取其句柄。**inMem**、**partitions**参数详细含义请参考DolphinDB用户手册-loadTable

对数据表句柄的操作以及面向对象的SQL查询，请参考章节3.4面向对象操作。

2.2 loadTable

```
loadTable(tableName,  dbPath=None, partitions=None, memoryMode=False)
```

与table方法相似，但该方法仅用于加载服务器端指定表，获取其句柄。

2.3 loadTableBySQL

```
loadTableBySQL(tableName, dbPath, sql)
```

该方法封装DolphinDB loadTableBySQL函数，将满足SQL查询中筛选条件的记录加载为内存中的分区表，返回内存表句柄给API。函数详细说明请参考DolphinDB用户手册-loadTableBySQL。

- **tableName/dbPath**：根据tableName和dbPath加载sql中使用到的分区表。
- **sql**：SQL查询的元代码。它可以用 WHERE 子句来过滤分区或记录行，也可以用 SELECT 语句选择包括计算列在内的列，但不能包含 TOP 子句、GROUP BY 子句、 ORDER BY 子句、CONTEXT BY 子句和 LIMIT 子句。

2.4 loadText

```
loadText(remoteFilePath, delimiter=",")
```

可使用 `loadText` 方法把远程文本文件导入到服务端的内存表中。该方法会在 Python 中返回一个 DolphinDB 内存表句柄。可使用 `toDF` 方法把 Python 中的内存表句柄对象 Table 转换为 pandas.DataFrame。

请注意，使用 `loadText` 方法时，载入的内存表数据量必须小于DolphinDB服务器可用内存。

```
>>> WORK_DIR = "C:/DolphinDB/Data"
>>> trade = s.loadText(WORK_DIR+"/example.csv")
```

将返回的 DolphinDB 表对象转化为 pandas DataFrame。表的数据传输发生在此步骤。

```
>>> trade.toDF()
      TICKER        date       VOL        PRC        BID       ASK
0       AMZN  1997.05.16   6029815   23.50000   23.50000   23.6250
1       AMZN  1997.05.17   1232226   20.75000   20.50000   21.0000
2       AMZN  1997.05.20    512070   20.50000   20.50000   20.6250
3       AMZN  1997.05.21    456357   19.62500   19.62500   19.7500
4       AMZN  1997.05.22   1577414   17.12500   17.12500   17.2500
5       AMZN  1997.05.23    983855   16.75000   16.62500   16.7500
...
13134   NFLX  2016.12.29   3444729  125.33000  125.31000  125.3300
13135   NFLX  2016.12.30   4455012  123.80000  123.80000  123.8300
```

`loadText` 函数导入文件时的默认分隔符是','。用户也可指定其他符号作为分隔符。例如，导入'\t'分割的表格形式文本文件：

```
>>> t1 = s.loadText(WORK_DIR+"/t1.tsv", '\t')
```

注： `loadText`/`ploadText`/`loadTextEx` 都是将远程文件加载到服务端，并非加载本地文件。

2.5 ploadText

`ploadText` 函数可以并行加载远程文本文件到内存分区表中。它的加载速度要比 `loadText` 函数快。

```
>>> trade = s.ploadText(WORK_DIR+"/example.csv")
>>> trade.rows
13136
```

2.6 loadTextEx

```
loadTextEx(dbPath, tableName,  partitionColumns=None, remoteFilePath=None, delimiter=",")
```

可使用函数 `loadTextEx` 把远程文本文件导入到分区数据库的分区表中。如果分区表不存在，函数会自动生成该分区表并把数据追加到表中。如果分区表已经存在，则直接把数据追加到分区表中。

函数 `loadTextEx` 的各个参数如下：

- **dbPath** 表示数据库路径
- **tableName** 表示分区表的名称
- **partitionColumns** 表示分区列
- **remoteFilePath** 表示文本文件在 DolphinDB 服务器上的绝对路径。
- **delimiter** 表示文本文件的分隔符（默认分隔符是逗号）

下面的例子使用函数 `loadTextEx` 创建了分区表 trade，并把 example.csv 中的数据加载到表中。

```
>>> import dolphindb.settings as keys
>>> if s.existsDatabase("dfs://valuedb"):
...     s.dropDatabase("dfs://valuedb")
...
>>> s.database(dbName='mydb', partitionType=keys.VALUE, partitions=["AMZN","NFLX", "NVDA"], dbPath="dfs://valuedb")
>>> trade = s.loadTextEx(dbPath="mydb", tableName='trade',partitionColumns=["TICKER"], remoteFilePath=WORK_DIR + "/example.csv")
>>> trade.toDF()
      TICKER       date       VOL      PRC      BID      ASK
0       AMZN 1997-05-15   6029815   23.500   23.500   23.625
1       AMZN 1997-05-16   1232226   20.750   20.500   21.000
2       AMZN 1997-05-19    512070   20.500   20.500   20.625
3       AMZN 1997-05-20    456357   19.625   19.625   19.750
4       AMZN 1997-05-21   1577414   17.125   17.125   17.250
...      ...        ...       ...      ...      ...      ...
13131   NVDA 2016-12-23  16193331  109.780  109.770  109.790
13132   NVDA 2016-12-27  29857132  117.320  117.310  117.320
13133   NVDA 2016-12-28  57384116  109.250  109.250  109.290
13134   NVDA 2016-12-29  54384676  111.430  111.260  111.420
13135   NVDA 2016-12-30  30323259  106.740  106.730  106.750

[13136 rows x 6 columns]
```

返回表中的行数：

```
>>> trade.rows
13136
```

返回表中的列数：

```
>>> trade.cols
6
```

展示表的结构：

```
>>> trade.schema
     name typeString  typeInt comment
0  TICKER     SYMBOL       17
1    date       DATE        6
2     VOL        INT        4
3     PRC     DOUBLE       16
4     BID     DOUBLE       16
5     ASK     DOUBLE       16
```

2.7 database

```
database(dbName=None, partitionType=None, partitions=None, dbPath=None, engine=None, atomic=None, chunkGranularity=None)
```

如果需要持久保存导入数据，或者需要导入的文件超过可用内存，可将数据导入 DFS 分区数据库保存。下面将使用一个例子来介绍如何创建分区数据库。

本节例子中会使用数据库 valuedb。首先检查该数据库是否存在，如果存在，将其删除：

```
>>> if s.existsDatabase("dfs://valuedb"):
...     s.dropDatabase("dfs://valuedb")
...
```

使用 `database` 方法创建值分区（VALUE）的数据库，使用股票代码作为分区字段。参数 partitions 表示分区方案。下例中，我们先导入 DolphinDB 的关键字，再创建数据库。

```
>>> import dolphindb.settings as keys
>>> s.database(dbName='mydb', partitionType=keys.VALUE, partitions=['AMZN','NFLX', 'NVDA'], dbPath='dfs://valuedb')
```

上述语句等同于在DolphinDB中执行脚本`db=database('dfs://valuedb', VALUE, ['AMZN','NFLX','NVDA'])`

除了值分区（VALUE），DolphinDB 还支持哈希分区（HASH）、范围分区（RANGE）、列表分区（LIST）与组合分区（COMPO），具体请参见 DolphinDB 用户手册-database。

创建了分区数据库后，不可更改分区类型，一般亦不可更改分区方案，但是值分区或范围分区（或者复合分区中的值分区或范围分区）创建后，DolphinDB脚本中可以分别使用 `addValuePartitions` 与 `addRangePartitions` 函数添加分区。

`database`方法的详细参数以及使用方法，请参考章节3.4面向对象操作。

3. 数据库表管理

特别的，Python API 封装部分服务端常用数据库表管理函数，作为 Session 的方法，可以调用这些方法对数据库表进行管理。

3.1 existsDatabase

```
>>> s.existsDatabase(dbUrl="dfs://testDB")
False
```

该函数用于判断 DolphinDB 服务端是否存在 dbUrl 对应的数据库，函数使用请参考 DolphinDB 用户手册-existsDatabase。

3.2 existsTable

```
>>> s.existsTable(dbUrl="dfs://valuedb", tableName="trade")
True
```

检查指定表是否存在于指定数据库中，函数使用请参考 DolphinDB 用户手册-existsTable。

3.3 dropDatabase

```
>>> s.dropDatabase(dbPath="dfs://valuedb")
```

删除指定数据库的所有物理文件，函数使用请参考 DolphinDB 用户手册-dropDatabase。

3.4 dropPartition

```
>>> s.dropPartition(dbPath="dfs://valuedb", partitionPaths="AMZN", tableName="trade")
```

删除数据库中指定分区的数据。如果指定了 tableName 则删除指定表中符合指定条件的分区数据。否则，删除指定数据库所有表中符合指定条件的分区数据。函数使用请参考 DolphinDB 用户手册-dropPartition。

注意： 如果创建数据库时指定分区粒度为 DATABASE，则可以不填 tableName，否则必须指定表名。

3.5 dropTable

```
>>> s.dropTable(dbPath="dfs://valuedb", tableName="trade")
```

删除指定数据库中的数据表，函数使用请参考 DolphinDB 用户手册-dropTable。

4. 其他方法

除了上述方法，Session中还提供封装了一些常用的函数。

4.1 undef/undefAll

```
>>> s.undef("t1", "VAR")
>>> s.undefAll()
```

`undef` 方法释放 Session 中的指定对象；`undefAll` 方法释放 Session 中的全部对象。`undef` 支持的对象类型包括："VAR"(变量)、"SHARED"(共享变量) 与 "DEF"(函数定义)。默认类型为变量 "VAR"。"SHARED" 指内存中跨 Session 的共享变量，例如流数据表。

假设 Session 中有一个 DolphinDB 的表对象 t1，可以通过 `undef("t1","VAR")` 将该表释放掉。释放后，并不一定能够看到内存在服务端马上释放。这与 DolphinDB 的内存管理机制有关。DolphinDB 从操作系统申请的内存，释放后不会立即还给操作系统，因为这些释放的内存在 DolphinDB 中可以立即使用。申请内存首先从 DolphinDB 内部的池中申请内存，不足才会向操作系统去申请。配置文件 (dolphindb.cfg) 中参数 *maxMemSize* 设置的内存上限会尽量保证。譬如说设置为 8GB，那么 DolphinDB 会尽可能利用 8GB 内存。所以如果用户需要反复 undef 内存中的一个变量以释放内存，为后面程序腾出更多内存空间，则需要将 *maxMemSize* 调整到一个合理的数值，否则当前内存没有释放，而后面需要的内存超过了系统的最大内存，DolphinDB 的进程就有可能被操作系统杀掉或者出现 out of memory 的错误。

4.2 clearAllCache

```
>>> s.clearAllCache()
>>> s.clearAllCache(dfs=True)
```

clearAllCache方法会调用服务端同名方法来清理服务端缓存，如果dfs参数为True，将会在所有节点上清理缓存；否则只会在连接节点上清理。

4.3 setTimeout

与Session建立连接时使用的参数keepAliveTime不同，setTimeout是Session的类方法，用于设置TCP连接TCP_USER_TIMEOUT选项。可以设置用户允许TCP连接在没有端到端连接的情况下的生存时间（单位 秒/s）。参考 Linux Socket options。

```
>>> ddb.Session.setTimeout(3600)
```

注： 本方法仅在Linux系统生效。默认时间为30秒。

4.4 流订阅相关

DolphinDB Python API提供流数据订阅接口，可以从服务器订阅流数据表，并获取其数据，详细介绍请参考章节3.3流订阅。 相关方法：`enableStreaming` / `subscribe` / `unsubscribe` / `getSubscriptionTopics`

构造 DBConnectionPool

DBConnectionPool （连接池）可以实现并发执行脚本。由前一章节的内容可知，Session（会话控制）可以实现 API 客户端与 DolphinDB 之间的信息交互。Python API 通过 Session 在 DolphinDB 上执行脚本和函数，同时实现双向的数据传递。但由于 Session 只能调用 `run()` 方法来串行执行脚本，且无法在多线程中使用同一 Session 执行脚本。因此，若需要并发地执行脚本，建议使用 DBConnectionPool 以提高任务运行的效率。

DBConnectionPool 通过创建多个线程以实现并发执行任务。如下展示创建一个 DBConnectionPool 的完整示例：

```
DBConnectionPool(host, port, threadNum=10, userid=None, password=None,
                 loadBalance=False, highAvailability=False, compress=False,
                 reConnect=False, python=False, protocol=PROTOCOL_DEFAULT)
```

通过调用方法函数 `getSessionId()` 来获取 DBConnectionPool 对象创建的所有线程会话的 Session id。若不再使用当前 DBConnectionPool，API 会在析构时自动释放连接。

以下内容将对创建 DBConnectionPool 的相关参数进行详细说明。

1. 连接参数 *host*, *port*, *threadNum*, *userid*, *password*

   - *host* 表示所连接服务器的地址。
   - *port* 表示所连接服务器的端口。
   - *threadNum* 表示建立连接的数量，默认为10。
   - *userid* 表示登录时的用户名。
   - *password* 表示登录时用户名对应的密码。

用户可以使用指定的域名（或 IP 地址）和端口号把 DBConnectionPool 连接到 DolphinDB，并且在建立连接的同时登录账号。使用示例如下：

```
import dolphindb as ddb

# 连接地址为localhost，端口为8848的DolphinDB，连接数为10
pool = ddb.DBConnectionPool("localhost", 8848)

# 连接地址为localhost，端口为8848的DolphinDB，登录用户名为admin，密码为123456的账户，连接数为8
pool = ddb.DBConnectionPool("localhost", 8848, 8, "admin", "123456")
```

注意：

- 在构造 DBConnectionPool 时，必须指定参数 *host*, *port*。
- 如果在构造时输入错误的参数值，将无法连接 DolphinDB，也无法正常创建 DBConnectionPool 对象。

2. 负载均衡参数 *loadBalance*

- *loadBalance*：连接池负载均衡相关配置参数，默认值为 False。

该参数的默认值为 False，表示不开启负载均衡。若要开启负载均衡，则将参数设置为 True。示例脚本如下：

```
import dolphindb as ddb

# 创建连接池；开启负载均衡
pool = ddb.DBConnectionPool("localhost", 8848, 8, loadBalance=True)
```

注意，在负载均衡模式下：

- 如果开启高可用，则可连接节点为集群中所有数据节点。此时负载均衡参数无效。
- 如果不开启高可用模式，则 DBConnectionPool 会向所有可连接的数据节点均匀建立连接。例如，集群中有3个节点，当前连接数分别为，DBConnectionPool 的连接数为6，则在建立连接后，集群中3个节点的连接数分别为，即每个节点均增加2个连接数。

3. 高可用参数 *highAvailability*

- *highAvailability* 表示是否在集群所有节点上进行高可用配置，默认值为 False。

在高可用模式下，如果不启用负载均衡模式，DBConnectionPool 会和当前集群中负载最小的节点建立连接。但由于 DBConnectionPool 中的连接为同时建立，故无法保证节点资源的负载均衡。

示例脚本如下：

```
import dolphindb as ddb

# 创建连接池；开启高可用，使用集群所有节点作为高可用节点
pool = ddb.DBConnectionPool("localhost", 8848, 8, "admin", "123456", highAvailability=True)
```

4. 压缩参数 *compress*

- *compress* 表示当前连接是否开启压缩，默认参数为 False。

该模式适用于大数据量的写入或查询。压缩数据后再传输，这可以节省网络带宽，但会增加 DolphinDB 和 API 端的计算量。使用示例如下：

```
import dolphindb as ddb
import dolphindb.settings as keys


# api version >= 1.30.21.1，开启压缩，需指定协议为PROTOCOL_DDB
pool = ddb.DBConnectionPool("localhost", 8848, 8, compress=True, protocol=keys.PROTOCOL_DDB)


# api version <= 1.30.19.4，开启压缩，默认使用协议为PROTOCOL_DDB，即enablePickle=False
pool = ddb.DBConnectionPool("localhost", 8848, 8, compress=True)
```

注意：

- DolphinDB 自1.30.6版本起支持压缩参数 *compress*。
- 目前仅在配置协议参数 *protocol* 为 PROTOCOL_DDB 的情况下支持开启压缩。（API version<=1.30.19.4 时，默认协议使用PROTOCOL_DDB，支持开启压缩）

5. 重连参数 *reconnect*

- *reconnect* 表示在不开启高可用的情况下，是否在 API 检测到连接异常时进行重连，默认值为 False。

若开启高可用模式，则 API 在检测到连接异常时将自动进行重连，不需要设置参数 *reconnect*。若未开启高可用，通过配置 `reconnect = True`，即可实现 API 在检测到连接异常时进行重连。使用示例如下：

```
import dolphindb as ddb


# 创建连接池；开启重连
pool = ddb.DBConnectionPool("localhost", 8848, 8, reconnect=True)
```

6. 协议参数 *protocol*

- *protocol* 表示 API 与 DolphinDB 交互时使用的数据格式协议，默认值为 PROTOCOL_DEFAULT，表示使用 PROTOCOL_PICKLE。

目前 DolphinDB 支持三种协议：PROTOCOL_DDB, PROTOCOL_PICKLE, PROTOCOL_ARROW。使用不同的协议，会影响 API 执行 DolphinDB 脚本后接收到的数据格式。有关协议的详细说明请参考章节3.1类型转换。仅为标记作用，此处需要放链接

```
import dolphindb.settings as keys


# 使用协议 PROTOCOL_DDB
pool = ddb.DBConnectionPool("localhost", 8848, 10, protocol=keys.PROTOCOL_DDB)


# 使用协议 PROTOCOL_PICKLE
```

```
pool = ddb.DBConnectionPool("localhost", 8848, 10, protocol=keys.PROTOCOL_PICKLE)


# 使用协议 PROTOCOL_ARROW
pool = ddb.DBConnectionPool("localhost", 8848, 10, protocol=keys.PROTOCOL_ARROW)
```

注意： 在1.30.21.1版本及之后，API 支持使用 *protocol* 来指定数据格式协议。1.30.19.4版本及之前，默认API内部使用PROTOCOL_DDB，即enablePickle=False。

7. 其他参数

- *python* 表示是否启用 python parser 特性。

指定该参数后，可以在 `DBConnectionPool.run` 执行脚本时启用 python parser 特性.使用示例如下：

```
import dolphindb as ddb


# 启用 python parser 特性
pool = ddb.DBConnectionPool("localhost", 8848, 10, python=True)
```

注意： 仅支持 DolphinDB 2.10 版本。（暂未发布，敬请期待）

方法介绍

本节将介绍连接池 DBConnectionPool 的三类常用方法，该三类方法兼可用于异步执行脚本，用户可根据实际需求选用不同的方法。

- run 是一个协程函数，其内部维护递增的 taskId，可以使用协程方式进行调用。
- addTask，isFinished，getData 方法是将脚本任务提交给 DBConnectionPool，由 DBConnectionpool 直接维护任务的异步进行和获得返回值，在使用时需要用户自行指定 taskId。
- runTaskAsync 是在 DBConnectionPool 的内部维护一个事件循环，被调用后将使用 run 方法在该内部事件循环中执行脚本任务，并返回一个 `concurrent.futures.Future` 对象。

注： 由于第一种和第三种方法的 taskId 由系统自动生成，而第二种方法的 taskId 由用户自行指定，故为了避免 taskId 冲突，建议用户在实际使用中不要将第二种方法和第一/三种方法混用。

1. run

```
run(script, *args, **kwargs)
```

- script：待执行的 DolphinDB 脚本。
- *args：传递给 DolphinDB 函数的参数。
- **kwargs：
  - *clearMemory*：是否在查询后释放变量。默认值为 True，表示释放。
  - *pickleTableToList*：是否将结果中的 Table 类型对象转换为 list 类型对象。True 表示转换为 list 类型对象，False 表示转换为 DataFrame 类型对象，该参数默认值为 False。

为了提高效率，DBConnectionPool 中的 run 方法被包装成了协程函数，通过 run 方法将脚本传入线程池中调用线程运行，因此在 Python 中调用 run 方法时需要使用协程以进行使用。

示例1

以下内容介绍一个简单的固定任务示例。

首先，创建一个最大连接数为8的连接池 DBConnectionPool。和通常连接池有所不同，当不再使用连接时，API 不会立刻销毁该连接，而是直到析构 DBConnectionPool 时才进行销毁，或者手动执行 `shutDown()` 关闭 DBConnectionPool 时才会销毁连接。

```
import dolphindb as ddb
import time
import asyncio

pool = ddb.DBConnectionPool("localhost", 8848, 8)
```

创建一个协程任务函数，使用 sleep 模拟一段运行的时间。

```
async def test_run(i):
    try:
        return await pool.run(f"sleep(2000);1+{i}")
    except Exception as e:
        print(e)
```

定义任务列表，并创建一个事件循环对象，运行任务列表直到完成全部任务。

```
tasks = [
    asyncio.ensure_future(test_run(1)),
    asyncio.ensure_future(test_run(3)),
    asyncio.ensure_future(test_run(5)),
    asyncio.ensure_future(test_run(7)),
```

```
]

loop = asyncio.get_event_loop()
try:
    time_st = time.time()
    loop.run_until_complete(asyncio.wait(tasks))
    time_ed = time.time()
except Exception as e:
    print("catch e:")
    print(e)
```

任务结束后，打印执行时间和各个任务的结果，并关闭连接池对象。

```
print("time: ", time_ed-time_st)

for task in tasks:
    print(task.result())

pool.shutDown()
```

期望的输出结果如下所示：

```
time:  2.0017542839050293
2
4
6
8
```

上述例子展示了已固定脚本任务调用 DBConnectionPool 的用法，在 Python 中只有一个主线程，但使用了协程创建子任务并调用 DBConnectionPool 以实现运行。须注意，实际上 Python API 的底层实现是通过使用 C++ 线程以维护每一个连接。若提交任务数超出实际线程数，则可能出现任务迟迟没有执行的情况，与通常的协程并发有一定区别。

此外，用户也可以自定义传入脚本的对象，可参考下述示例2。

示例2

下例定义了一个可以传入自定义脚本作为参数的类，并配合 Python 的多线程机制动态添加子任务。

```
import dolphindb as ddb
import time
import asyncio
import threading
```

```
# 在该例子中主线程负责创建协程对象传入自定义脚本并调用自定义的对象去运行，并新起子线程运行事件循环防止阻塞主线程。
class DolphinDBHelper(object):
    pool = ddb.DBConnectionPool("localhost", 8848, 10)
    @classmethod
    async def test_run(cls,script):
        print(f"run script: [{script}]")
        return await cls.pool.run(script)

    @classmethod
    async def runTest(cls,script):
        start = time.time()
        task = loop.create_task(cls.test_run(script))
        result = await asyncio.gather(task)
        print(f"""[{time.strftime("%Y-%m-%d %H:%M:%S", time.localtime())}] time: {time.time()-start} result: {result}""")
        return result

#定义一个跑事件循环的线程函数
def start_thread_loop(loop):
    asyncio.set_event_loop(loop)
    loop.run_forever()

if __name__=="__main__":
    start = time.time()
    print("In main thread",threading.current_thread())
    loop = asyncio.get_event_loop()
    # 在子线程中运行事件循环，让它 run_forever
    t = threading.Thread(target= start_thread_loop, args=(loop,))
    t.start()
    task1 = asyncio.run_coroutine_threadsafe(DolphinDBHelper.runTest("sleep(1000);1+1"),loop)
    task2 = asyncio.run_coroutine_threadsafe(DolphinDBHelper.runTest("sleep(3000);1+2"),loop)
    task3 = asyncio.run_coroutine_threadsafe(DolphinDBHelper.runTest("sleep(5000);1+3"),loop)
    task4 = asyncio.run_coroutine_threadsafe(DolphinDBHelper.runTest("sleep(1000);1+4"),loop)

    end = time.time()
    print("main thread time: ", end - start)
```

运行结果如下：

```
In main thread <_MainThread(MainThread, started 139838803788160)>
main thread time:  0.00039839744567871094
run script: [sleep(1000);1+1]
run script: [sleep(3000);1+2]
run script: [sleep(5000);1+3]
run script: [sleep(1000);1+4]
[2023-03-14 16:46:56] time: 1.0044968128204346 result: [2]
[2023-03-14 16:46:56] time: 1.0042989253997803 result: [5]
[2023-03-14 16:46:58] time: 3.0064148902893066 result: [3]
[2023-03-14 16:47:00] time: 5.005709409713745 result: [4]
```

上述例子中，在主线程中创建子线程开启事件循环，并指定该事件循环一直保持运行。随后向该事件循环中加入四个脚本执行任务，每个任务分别需要耗时1s、3s、5s和1s。从主线程打印 `main thread time: 0.00039839744567871094` 可以看出，四个事件放入事件循环后实现了异步执行，随后每个协程都打印自身执行的结束时间和时长。由于任务1和任务4耗时一致，因此同时打印结果；2s后任务2执行结束；再过2s后任务3也执行结束。由结果可知，四个任务的执行结果符合并发执行的预期。

2. addTask, isFinished, getData

不同于协程函数 run，addTask 会将用户脚本任务按照 taskId 直接提交给 DBConnectionPool 执行。用户可以通过 `isFinished` 判断线程池中的任务是否结束，并使用 `getData` 获取任务的返回结果。下述内容将依次介绍三个函数。

2.1 addTask

```
addTask(script, taskId, *args, **kwargs)
```

- script：待执行的 DolphinDB 脚本。
- taskId：指定的任务 Id。
- *args：传递给 DolphinDB 函数的参数。
- **kwargs：
  - *clearMemory*：是否查询后释放变量。True 表示释放，默认值为 True。
  - *pickleTableToList*：是否将结果的 Table 类型对象转换为 list 类型对象。True 表示转换为 list 类型对象，False 表示转换为 DataFrame 类型对象，默认值为 False。

addTask 根据 taskId 将任务提交至 DBConnectionPool 的连接池中，，由连接池分配连接执行脚本任务。如下所示，调用 `addTask` 向连接池中添加一个 taskId 为12的任务。

```
pool.addTask("sleep(1000);1+2", taskId=12)
```

2.2 isFinished

```
isFinished(taskId)
```

- taskId：查询的任务 Id。

该函数通过 taskId 来查询对应任务是否已经完成。如果已完成，则返回 True；反之返回 False。简单使用示例如下：

```
if pool.isFinished(taskId=12):
    print("task has done!")
```

2.3 getData

```
getData(taskId)
```

- taskId：查询的任务 Id。

该函数通过 taskId 来查询对应任务的返回结果。简单使用示例如下：

```
res = pool.getData(taskId=12)
```

注意： 每次执行 `addTask` 指定 taskId 并创建任务后，使用 `getData` 方法只能对该 taskID 对应任务的返回结果执行一次查询。若在创建任务后未调用 `getData` 方法，则在下次使用 `addTask` 指定同一 taskId 并创建任务时，其执行结果将覆盖掉前一次 该 taskId 对应任务的执行结果。

2.4 综合示例

在如下脚本中，首先创建一个 DBConnectionPool 连接池对象，然后调用 `addTask` 向连接池中添加一个 taskId 为12的任务，随后通过 `isFinished` 方法判断任务是否执行完毕，执行完毕后跳出循环，调用 `getData` 方法获取任务结果。

```
import dolphindb as ddb
import time

pool = ddb.DBConnectionPool("localhost", 8848, 8)
taskid = 12
pool.addTask("sleep(1500);1+2", taskId=taskid)
while True:
    if pool.isFinished(taskId=taskid):
        break
    time.sleep(0.01)

res = pool.getData(taskId=taskid)
print(res)
```

```
# output:
3
```

3. runTaskAsync

```
runTaskAsync(script, *args, **kwargs)
```

- script：待执行的 DolphinDB 脚本。
- *args：传递给 DolphinDB 函数的参数。
- **kwargs：
  - *clearMemory*：是否查询后释放变量。True 表示释放，默认值为 True。
  - *pickleTableToList*：是否将结果中的 Table 类型对象转换为 list 类型对象。True 表示转换为 list 类型对象，False 表示转换为 DataFrame 类型对象，默认值为 False。

注1： 在1.30.17.4及以前的版本中，该函数的名称为 runTaskAsyn。 **注2：** 若使用该方法异步执行脚本，在任务结束后，用户需要手动调用 `pool.shutDown()` 才能正确析构连接池对象。

除了使用 run 和 addTask 的方法来执行脚本，DBConnectionPool 还提供了 runTaskAsync 的方法以实现异步执行脚本。

用户可以调用 runTaskAsync 方法向连接池中添加任务，返回一个 `concurrent.futures.Future` 对象。然后调用这个对象的 `result(timeout=None)` 方法获得结果 (*timeout*，单位为秒)。如果在 `result()` 方法中设置了 *timeout* 参数，任务还未完成，则继续等待 *timeout* 时间；在 *timeout* 时间内若任务完成，则将返回结果，否则将抛出 timeoutError 异常。下面演示如何使用 runTaskAsync 创建异步任务。

```
import dolphindb as ddb
import time
pool = ddb.DBConnectionPool("localhost", 8848, 10)

t1 = time.time()
task1 = pool.runTaskAsync("sleep(1000); 1+0");
task2 = pool.runTaskAsync("sleep(2000); 1+1");
task3 = pool.runTaskAsync("sleep(4000); 1+2");
task4 = pool.runTaskAsync("sleep(1000); 1+3");
t2 = time.time()
print(task1.result())
t3 = time.time()
print(task2.result())
```

```
t4 = time.time()
print(task4.result())
t5 = time.time()
print(task3.result())
t6 = time.time()

print(t2-t1)
print(t3-t1)
print(t4-t1)
print(t5-t1)
print(t6-t1)
pool.shutDown()
```

输出结果如下：

```
Task1 Result:  1
Task2 Result:  2
Task4 Result:  4
Task3 Result:  3
Add Tasks:  0.0015881061553955078
Get Task1:  1.0128183364868164
Get Task2:  2.0117716789245605
Get Task4:  2.0118134021759033
Get Task3:  4.012163162231445
```

如上示例中，首先创建一个 DBConnectionPool 连接池对象，调用 `runTaskAsync` 方法执行 4 个耗时不同的脚本，并分别返回 4 个 `concurrent.futures.Future` 对象。再调用其 `result` 方法依次阻塞地获得各个任务的返回值，并将耗时打印出来。如下为打印说明：

- t2 - t1：表示添加任务所耗时间。
- t3 - t1：表示获取task1结果的总耗时。
- t4 - t1：表示获取task2结果的总耗时。
- t5 - t1：表示获取task4结果的总耗时。
- t6 - t1：表示获取task3结果的总耗时。

Task1 执行耗时 1s，因此 t3 - t1 = 1s；Task2 执行耗时 2s，因此 t4 - t1 = 2s；Task4 执行耗时 1s，在等待获取 Task2 结果的时候，Task4 任务已经执行结束，因此 t5-t1 = 2s；Task3 执行耗时 4s，因此 t6-t1 = 4s。

4. 其他方法

4.1 shutDown

```
pool.shutDown()
```

该方法用于关闭不再使用的 DBConnectionPool，停止线程池中使用的事件循环，并且中止所有的异步任务。调用 shutDown 方法后，关闭的线程池不可继续使用。

注意： 如果使用了 `runTaskAsync()` 的方式创建异步任务，必须在不使用 DBConnectionPool 时调用该函数。

4.2 getSessionId

```
sessionids = pool.getSessionId()
```

该方法用于获得当前线程池中所有 Session 会话中的 Session Id。

TableAppender

由于 Python 与 DolphinDB 的数据类型并不是一一对应的，故部分类型数据无法直接进行上传、写入等操作。举例来说，Python 中的 DataFrame 对象中仅支持 datetime64 时间类型，若直接从 API 上传 DataFrame 数据到 DolphinDB，所有的时间类型列均将转换为 NANOTIMESTAMP 类型。并且每次向内存表或分布式表追加一个带有时间类型列的 DataFrame 时，用户需要专门对时间列进行类型转换，该类情况影响了用户的使用体验。

针对以上情况，Python API 提供了 TableAppender 对象，在通过 append 方法向内存表或者分布式表中添加本地 DataFrame 数据时，能够实现对部分类型进行自动转换，用户无须再进行额外的手动转换。

注1： 1.30.19.4及之前版本的 API 仅支持使用 TableAppender 对时间类型数据进行自动转换；1.30.21.1及之后版本的 API 支持使用 TableAppender 对所有类型数据的自动转换。

注2： 1.30.16、2.00.4及之后版本的 DolphinDB 支持使用 tableInsert 对时间类型数据进行自动转换。

注3： 1.30.22.1及之后版本的 API，调整 tableAppender 类名为 TableAppender，兼容原有 tableAppender。

1. 接口说明

```
TableAppender(dbPath=None, tableName=None, ddbSession=None, action="fitColumnType")
```

- dbPath: 分布式数据库的地址。若待写入表为内存表，可以不指定该参数。
- tableName: 分布式表或内存表的表名。
- dbSession: 已连接 DolphinDB 的 Session 对象。
- action：指定 append 表时的行为。目前仅支持 fitColumnType，表示对列类型进行转换。

TableAppender 类仅支持 append 方法，接口如下：

```
append(table)
```

- table：待写入数据，通常为 pandas.DataFrame 类型的本地数据。

2. 示例

下例创建了一个共享表 t，通过 TableAppender 向表 t 中添加数据。

```
import pandas as pd
import dolphindb as ddb
import numpy as np
s = ddb.Session()
s.connect("localhost", 8848, "admin", "123456")


s.run("share table(1000:0, `sym`timestamp`qty, [SYMBOL, TIMESTAMP, INT]) as t")
appender = ddb.TableAppender(tableName="t", ddbSession=s)
data = pd.DataFrame({
    'sym': ['A1', 'A2', 'A3', 'A4', 'A5'],
    'timestamp': np.array(['2012-06-13 13:30:10.008', 'NaT','2012-06-13 13:30:10.008', '2012-06-13 15:30:10.008', 'NaT'],
 dtype="datetime64[ms]"),
    'qty': np.arange(1, 6),
})
num = appender.append(data)
print("append rows: ", num)
t = s.run("t")
print(t)
schema = s.run("schema(t)")
print(schema["colDefs"])
```

输出结果如下：

```
append rows:  5
  sym              timestamp  qty
0  A1 2012-06-13 13:30:10.008    1
1  A2                    NaT    2
2  A3 2012-06-13 13:30:10.008    3
3  A4 2012-06-13 15:30:10.008    4
4  A5                    NaT    5
       name typeString  typeInt  extra comment
0       sym     SYMBOL       17    NaN
```

```
1  timestamp  TIMESTAMP       12    NaN
2        qty        INT        4    NaN
```

共享表 t 中的 timestamp 列被定义为 TIMESTAMP 类型，但是在 API 端写入的 pd.DataFrame 对象中该列的数据类型对应为 datetime64。由上述结果可知，通过 TableAppender 将表 t 从 API 上传至 DolphinDB 后，该列的数据类型将自动转换为TIMESTAMP。

3. 常见问题

3.1 自动转换的类型

1.30.16、2.00.4 之后版本的 DolphinDB 支持向内存表写入数据时**自动转换时间类型**，因此用户可以不使用本文介绍的 TableAppender 方法，而是直接使用 run 方法执行 `tableInsert` 将上传的 DataFrame 本地时间类型数据插入到指定表中。

此外，由于在 Python API 中 DolphinDB 的 UUID、INT128、IPADDR、BLOB 等类型对应 Python 的 str 类型，故无法直接上传这些类型的数据，进而无法插入到指定表中。在1.30.19.4及之后版本的 Python API 中，TableAppender 对象支持**自动识别指定表的类型**，如果上传数据中包含 UUID 等特殊类型，TableAppender 对象将自动识别 str 类型，并将其转换为表中对应的类型。写入数据时的类型转换相关内容，请参考章节3.1类型转换。仅为标注，此处要加跳转链接

简单示例如下：

```
import dolphindb as ddb
import dolphindb.settings as keys
import numpy as np
import pandas as pd

s = ddb.Session(protocol=keys.PROTOCOL_DDB)
s.connect("192.168.1.113", 8848, "admin", "123456")

s.run("share table(1000:0, `sym`uuid`int128`ipaddr`blob, [SYMBOL, UUID, INT128, IPADDR, BLOB]) as t")
appender = ddb.TableAppender(tableName="t", ddbSession=s)
data = pd.DataFrame({
    'sym': ["A1", "A2", "A3"],
    'uuid': ["5d212a78-cc48-e3b1-4235-b4d91473ee87", "b93b8253-8d5e-c609-260a-86522b99864e", ""],
    'int128': [None, "073dc3bc505dd1643d11a4ac4271d2f2", "e60c84f21b6149959bcf0bd6b509ff6a"],
    'ipaddr': ["2c24:d056:2f77:62c0:c48d:6782:e50:6ad2", "", "192.168.1.0"],
    'blob': ["testBLOB1", "testBLOB2", "testBLOB3"],
})

appender.append(data)
```

```
t = s.run("t")
print(t)
```

注： 上例中，由于要下载并展示 BLOB 类型的数据，而 Session 默认的数据传输协议 PROTOCOL_PICKLE 并不支持 BLOB 类型数据，故须指定其数据传输协议为 PROTOCOL_DDB。

输出结果如下：

```
  sym                                  uuid                             int128                                ipaddr       blob
0  A1  5d212a78-cc48-e3b1-4235-b4d91473ee87  00000000000000000000000000000000  2c24:d056:2f77:62c0:c48d:6782:e50:6ad2  testBLOB1
1  A2  b93b8253-8d5e-c609-260a-86522b99864e  073dc3bc505dd1643d11a4ac4271d2f2                               0.0.0.0  testBLOB2
2  A3  00000000-0000-0000-0000-000000000000  e60c84f21b6149959bcf0bd6b509ff6a                           192.168.1.0  testBLOB3
```

3.2 Pandas 警告

在1.30.19.4及之后版本的 Python API 中，用户在使用 TableAppender 类的 append 方法写入数据时，可能会收到如下警告：

```
UserWarning: Pandas doesn't allow columns to be created via a new attribute name - see https://pandas.pydata.org/pandas-docs/stable/indexing.html#attribute-access
```

该警告并不会对程序执行造成任何影响，如需屏蔽，可以使用如下方法：

```
import warnings
warnings.filterwarnings("ignore","Pandas doesn't allow columns to be created via a new attribute name - see
 https://pandas.pydata.org/pandas-docs/stable/indexing.html#attribute-access", UserWarning)
```

TableUpserter

Python API 提供 TableUpserter 对象，可以通过 upsert 方式向索引内存表、键值内存表以及分布式表中追加数据。与 TableAppender 对象类似，使用 TableUpserter 对象向表中添加本地的 DataFrame 数据，能够对时间类型进行自动转换，用户无须再进行额外的手动转换。

注： 1.30.19.4及之后版本的 API 同时支持使用 TableUpserter 进行 UUID 等特殊类型的自动转换。

接口说明

```
TableUpserter(dbPath=None, tableName=None, ddbSession=None, ignoreNull=False, keyColNames=[], sortColumns=[])
```

- dbPath: 分布式数据库地址。若待写入表为内存表，可以不指定该参数。
- tableName: 分布式表或索引内存表、键值内存表的表名。
- ddbSession: 已连接 DolphinDB 的 Session 对象。
- ignoreNull：布尔值。若追加的新数据中某元素为 NULL 值，是否对目标表中的相应数据进行更新。默认值为 False。

- keyColNames：字符串列表。由于 DFS 表不包含键值列，在更新 DFS 表时，会将该参数指定的列视为键值列。
- sortColumns：字符串列表。设置该参数后，更新的分区内的所有数据会根据指定的列进行排序。排序在每个分区内部进行，不会跨分区排序。

该类仅支持 upsert 方法，接口如下：

```
upsert(table)
```

- table：待写入数据，通常为 pandas.DataFrame 类型的本地数据。

示例1

下例创建了一个以 id 列为 key 的共享键值内存表 keyed_t，然后构造 TableAppender 对象，调用其 upsert 方法向表 keyed_t 中添加数据。在构造的数据中，id 列在0-9之间循环，text 列则不断递增。最后查询写入的数据，仅保留每个 id 下最后一条写入的数据。

```
import dolphindb as ddb
import dolphindb.settings as keys
import numpy as np
import pandas as pd
s = ddb.Session()
s.connect("localhost", 8848, "admin", "123456")


script_KEYEDTABLE = """
    testtable=keyedTable(`id,1000:0,`date`text`id,[DATETIME,STRING,LONG])
    share testtable as keyed_t
    """
s.run(script_KEYEDTABLE)
upserter = ddb.TableUpserter(tableName="keyed_t", ddbSession=s)
dates=[]
texts=[]
ids=[]
for i in range(1000):
    dates.append(np.datetime64('2012-06-13 13:30:10.008'))
    texts.append(f"test_i_{i}")
    ids.append(i%10)
df = pd.DataFrame({
    'date': dates,
    'text': texts,
    'id': ids,
})
```

```
upserter.upsert(df)
keyed_t = s.run("keyed_t")
print(keyed_t)
```

输出结果符合预期：

```
                date        text  id
0 2012-06-13 13:30:10  test_i_990   0
1 2012-06-13 13:30:10  test_i_991   1
2 2012-06-13 13:30:10  test_i_992   2
3 2012-06-13 13:30:10  test_i_993   3
4 2012-06-13 13:30:10  test_i_994   4
5 2012-06-13 13:30:10  test_i_995   5
6 2012-06-13 13:30:10  test_i_996   6
7 2012-06-13 13:30:10  test_i_997   7
8 2012-06-13 13:30:10  test_i_998   8
9 2012-06-13 13:30:10  test_i_999   9
```

示例2

若要写入没有键值列的分区表或者内存表，则需要在构造 TableUpserter 时指定键值列。

下例中，首先定义一个 VALUE 分区方式的分区表 p_table，然后构造 TableUpserter 对象并指定 id 为键值列，调用其 upsert 方法向表 p_table 中添加数据。最后查询写入的数据。

```
import dolphindb as ddb
import dolphindb.settings as keys
import numpy as np
import pandas as pd
import datetime

s = ddb.Session()
s.connect("localhost", 8848, "admin", "123456")
script_DFS_VALUE = """
    if(existsDatabase("dfs://valuedb")){
        dropDatabase("dfs://valuedb")
    }
    db = database("dfs://valuedb", VALUE, 0..9)
    t = table(1000:0, `date`text`id`flag, [DATETIME, STRING, LONG, LONG])
    p_table = db.createPartitionedTable(t, `pt, `flag)
"""
s.run(script_DFS_VALUE)
upserter = ddb.TableUpserter(dbPath="dfs://valuedb", tableName="pt", ddbSession=s, keyColNames=["id"])
```

```
for i in range(10):
    dates = [np.datetime64(datetime.datetime.now()) for _ in range(100)]
    texts = [f"test_{i}_{_}" for _ in range(100)]
    ids = [ _ % 10 for _ in range(100)]
    flags = [ _ % 10 for _ in range(100)]
    df = pd.DataFrame({
        'date': dates,
        'text': texts,
        'id': ids,
        'flag': flags,
    })
    upserter.upsert(df)

p_table = s.run("select * from p_table")
print(p_table)
```

输出结果如下所示：

```
                 date       text  id  flag
0  2023-03-16 10:09:33  test_9_90   0     0
1  2023-03-16 10:09:26  test_0_10   0     0
2  2023-03-16 10:09:26  test_0_20   0     0
3  2023-03-16 10:09:26  test_0_30   0     0
4  2023-03-16 10:09:26  test_0_40   0     0
..                 ...        ...  ..   ...
95 2023-03-16 10:09:26  test_0_59   9     9
96 2023-03-16 10:09:26  test_0_69   9     9
97 2023-03-16 10:09:26  test_0_79   9     9
98 2023-03-16 10:09:26  test_0_89   9     9
99 2023-03-16 10:09:26  test_0_99   9     9

[100 rows x 4 columns]
```

由上述结果可知，当键值列在某分区中值不唯一时，执行 `upsert` 时仅会覆盖分区中当前键值列下该键值对应的第一个数据。

注： TableUpserter 实际上调用了 DolphinDB 的 upsert! 函数，同时传入 pandas.DataFrame 作为参数以实现其功能。upsert! 函数的详细使用方式请参考 DolphinDB用户手册-upsert!。

PartitionedTableAppender

与 TableAppender 对象类似，使用 PartitionedTableAppender 对象向表中追加时间类型数据时，能够实现对时间类型数据的自动转换和多线程并行写入。其基本原理是在构造时接收一个 DBConnectionPool 对象作为参数，再调用 append 方法，将数据按照分区拆分后传入连接池以实现并发追加。需注意，一个分区在同一时间只能由一个连接写入。

接口说明

```
PartitionedTableAppender(dbPath=None, tableName=None, partitionColName=None, dbConnectionPool=None)
```

- dbPath: 分布式数据库名字。如为内存表则不需要指定。
- tableName: 分区表表名。
- partitionColName: 字符串，表示分区字段。
- dbConnectionPool: DBConnectionPool，连接池对象。

注： 指定参数 *partitionColName* 时，如果分区表中仅包含一个分区列，则该参数必须指定为该分区列；如果存在多个分区，则该参数可指定为任意分区列。在指定该参数后，API 将根据分区字段进行数据分配。

该类仅支持 append 方法，接口如下：

```
append(table)
```

- table：待写入数据，通常为 pandas.DataFrame 类型的本地数据。

示例

下例创建了一个分布式数据库 dfs://valuedb 和一个分布式表 pt，同时创建了连接池 pool 并传入 PartitionedTableAppender，然后使用 append 方法向分布式表并发写入本地数据。最后查询写入的数据。

```
import dolphindb as ddb
import pandas as pd
import numpy as np
import random

s = ddb.Session()
s.connect("localhost", 8848, "admin", "123456")
script = """
    dbPath = "dfs://valuedb"
    if(existsDatabase(dbPath)){
```

```
        dropDatabase(dbPath)
    }
    t = table(100:0, `id`date`vol, [SYMBOL, DATE, LONG])
    db = database(dbPath, VALUE, `APPL`IBM`AMZN)
    pt = db.createPartitionedTable(t, `pt, `id)
"""
s.run(script)

pool = ddb.DBConnectionPool("localhost", 8848, 3, "admin", "123456")
appender = ddb.PartitionedTableAppender(dbPath="dfs://valuedb", tableName="pt", partitionColName="id", dbConnectionPool=pool)
n = 100

dates = []
for i in range(n):
    dates.append(np.datetime64(
        "201{:d}-0{:1d}-{:2d}".format(random.randint(0, 9), random.randint(1, 9), random.randint(10, 28))))

data = pd.DataFrame({
    "id": np.random.choice(['AMZN', 'IBM', 'APPL'], n),
    "time": dates,
    "vol": np.random.randint(100, size=n)
})
re = appender.append(data)

print(re)
print(s.run("pt = loadTable('dfs://valuedb', 'pt'); select * from pt;"))
```

输出结果如下：

```
100
     id       date  vol
0   AMZN 2017-01-22   60
1   AMZN 2014-08-12   37
2   AMZN 2012-09-10   68
3   AMZN 2012-03-14   48
4   AMZN 2016-07-12    1
..   ...        ...  ...
95   IBM 2016-05-15   25
96   IBM 2012-06-19    6
97   IBM 2010-05-10   96
98   IBM 2017-07-10   32
99   IBM 2012-09-23   68
```

```
[100 rows x 3 columns]
```

订阅

Python API 支持流数据订阅的功能，以下介绍如何启用流数据，订阅流数据，获取订阅主题和取消订阅流数据。

1. 启用流数据

使用 Session 提供的 `enableStreaming` 方法启用流数据功能，使用示例如下：

```
enableStreaming(port=0)
```

port：指定开启数据订阅的端口，用于订阅服务器端发送的数据。（1.30.21、 2.00.9 之后版本的 DolphinDB 无需指定该参数）

使用时须注意，API 进行订阅时，其订阅行为根据 DolphinDB 的版本有所不同，详细说明如下：

DolphinDB Server 版本	Python API 版本	是否需要指定端口
1.30.21, 2.00.9 之前的版本	与 DolphinDB Server 版本对应的版本	是
1.30.21, 2.00.9 及之后的版本	与 DolphinDB Server 版本对应的版本	否

- 1.30.21, 2.00.9 之前的版本在订阅端提交订阅请求后，发布端需要向 API 端指定端口重新发起一个 TCP 连接用于传输数据。
- 1.30.21, 2.00.9 及之后的版本支持发布端通过订阅端的请求连接推送数据。因此，订阅端无需指定端口（默认值为0）；如果指定，该参数无效，会被 API 忽略。

使用不同版本 API 启用流数据功能的脚本示例如下：

```
import dolphindb as ddb
s = ddb.Session()
# 1.30.21、2.00.9 之前的版本，开启订阅，指定端口8000
s.enableStreaming(8000)
# 1.30.21、2.00.9 及之后的版本，开启订阅，无需指定端口
s.enableStreaming()
```

注意：若同时升级 API 和 Server 至1.30.21, 2.00.9及之后的版本，须在升级前先取消订阅，完成升级后再重新订阅。

2. 订阅

使用 `subscribe` 方法订阅 DolphinDB 中的流数据表，接口示例如下：

```
subscribe(host, port, handler, tableName, actionName=None, offset=-1, resub=False,
          filter=None, msgAsTable=False, batchSize=0, throttle=1.0,
          userName=None, password=None, streamDeserializer=None)
```

2.1 参数介绍

连接参数： *host*, *port*, *userName*, *password*

- *host*： 字符串，必填，表示发布端节点的 IP 地址。
- *port*： 字符串，必填，表示发布端节点的端口号。
- *userName*： 字符串，可选，表示所连接 DolphinDB 的登录用户名。
- *password*： 字符串，可选，表示所连接 DolphinDB 的登录用户名对应的密码。

回调参数： *handler*

- *handler*： 用户自定义的回调函数，用于处理每次流入的数据。

下例定义了一个简单的回调函数：

```
def handler(msg):
    print(msg)
```

订阅参数： *tableName*, *actionName*, *offset*, *resub*

- *tableName*：表示发布表的名称。
- *actionName*：表示订阅任务的名称。
  - 订阅时，订阅主题由订阅表所在节点的别名、流数据表名称和订阅任务名称组合而成，使用“/”分隔。注意，如果订阅主题已经存在，将会订阅失败。
- *offset*：整数，表示订阅任务开始后的第一条消息所在的位置。消息即为流数据表中的行。
  - 若该参数未指定，或设为-1，订阅将会从流数据表的当前行开始。
  - 若该参数设为-2，系统会获取持久化到磁盘上的 offset，并从该位置开始订阅。其中，offset 与流数据表创建时的第一行对应。如果某些行因为内存限制被删除，在决定订阅开始的位置时，这些行仍然考虑在内。

- *resub*：布尔值，表示订阅中断后，API 是否进行自动重订阅。默认值为 False，表示不会自动重订阅。
- *filter*：一个数组，表示过滤条件。在流数据表过滤列中，只有符合 *filter* 过滤条件的数据才会发布到订阅端。

模式参数： *msgAsTable*, *batchSize*, *throttle*, *streamDeserializer*

- *msgAsTable*：布尔值。若该参数设置为 True，表示订阅的数据将会转换为 DataFrame 格式。须注意，只有设置了参数 *batchSize*，参数 *msgAsTable* 才会生效。**注意**：若设置了 *streamDeserializer*，则参数 *msgAsTable* 必须设置为 False。
- *batchSize*：整数，表示批处理的消息的数量。
  - 如果该参数是正数：
    - 设置 `msgAsTable = False` 时，直到消息的数量达到 *batchSize* 时，*handler* 才会处理进来的 *batchSize* 条消息，并返回一个 list，其中每一项都是单条数据。
    - 设置 `msgAsTable = True` 时，回调参数 *handler* 将基于消息块（由 DolphinDB 中的参数 *maxMsgNumPerBlock* 进行配置）处理消息。当收到的记录总数大于等于 *batchSize* 时，*handler* 将处理所有达到条件的消息块。举例说明：若设置 `batchSize = 1500`，API 收到 DolphinDB 发送的第一个消息块（包含1024条记录），1024<1500，*handler* 将不处理消息；API 收到第2个消息块（包含500条记录），此时，1024+500>1500，即两个消息块中包含的记录数大于 *batchSize*，*handler* 将开始处理这两个消息块中的1524条记录。
  - 如果该参数是非正数或者未被指定，消息到达之后，*handler* 将立刻逐条处理消息，返回的数据将为单条数据组成的 list。
- *throttle*：浮点数，表示 *handler* 处理到达的消息之前等待的时间。单位为秒，默认值为1.0。如果未指定 *batchSize*，参数 *throttle* 将无法产生作用。
- *streamDeserializer*：表示订阅的异构流表对应的反序列化器。

注意： 订阅流表时，参数 *batchsize*、*msgAsTable*、*streamDeserializer* 的值都将影响传入回调函数 *handler* 中变量的格式，各种参数的订阅示例请参考章节3.3。`仅为标注，之后补充跳转链接`

2.2 订阅示例

先在 DolphinDB 中创建共享的流数据表，指定进行过滤的列为 sym，并向5个 symbol 各插入2条记录，共10条记录：

```
share streamTable(10000:0,`time`sym`price`id, [TIMESTAMP,SYMBOL,DOUBLE,INT]) as trades
setStreamTableFilterColumn(trades, `sym)
insert into trades values(take(now(), 10), take(`000905`600001`300201`000908`600002, 10), rand(1000,10)/10.0, 1..10)
```

在 Python API 中，首先指定订阅端口号并启用流订阅（本例中使用1.30.21, 2.00.9之前版本的 DolphinDB，故需要指定订阅端口号），然后定义回调函数 *handler*。再调用 subscribe 方法开启流订阅，其中设置 `offset=-1`，`filter=np.array(["000905"])`。最后通过 Event().wait()的方式阻塞主线程，保持进程不退出。

```
import dolphindb as ddb
import numpy as np
s = ddb.Session()
s.enableStreaming(0) # DolphinDB 的版本小于 1.30.21 和 2.00.9 时，需指定端口

def handler(lst):
    print(lst)

s.subscribe("192.168.1.113", 8848, handler, "trades", "action", offset=-1, filter=np.array(["000905"]))

from threading import Event
Event().wait()          # 阻塞主线程，保持进程不退出
```

执行脚本，发现此时没有流数据被打印出来。该情况是由于 offset 设置为-1，表示订阅将会从流数据表的当前行开始，因此 DolphinDB `insert into` 命令中包含的数据将不会发送到流订阅客户端。

在 DolphinDB 中再次执行同一脚本：

```
insert into trades values(take(now(), 10), take(`000905`600001`300201`000908`600002, 10), rand(1000,10)/10.0, 1..10)
```

此时 Python 进程开始打印收到的流数据，输出内容如下：

```
[numpy.datetime64('2023-03-17T10:11:19.684'), '000905', 69.3, 1]
[numpy.datetime64('2023-03-17T10:11:19.684'), '000905', 96.5, 6]
```

在 API 收到的数据中，由于 *filter* 参数生效，故打印结果中仅保留了 `sym="000905"`的数据，其余数据都被过滤掉。

3. 获取订阅主题

通过 `getSubscriptionTopics` 方法可以获取当前 Session 中的所有订阅主题。主题的构成方式是：`host/port/tableName/actionName`，每个 Session 的主题互不相同。使用示例如下：

```
s.getSubscriptionTopics()
```

示例的输出结果如下所示：

```
['192.168.1.113/8848/trades/action']
```

4. 取消订阅

使用 `unsubscribe` 方法可以取消订阅，接口如下：

```
unsubscribe(host, port, tableName, actionName=None)
```

取消示例中的订阅：

```
s.unsubscribe("192.168.1.113", 8848, "trades", "action")
```

Session 异步提交

在高吞吐率的场景下，尤其是典型的高速小数据写入，使用 API 的异步调用功能可以有效提高 API 的任务吞吐量。异步方式提交有如下几个特点：

- API 客户端提交任务后，DolphinDB 接到任务后客户端即认为任务已完成。
- API 客户端无法得知任务在 DolphinDB 执行的情况和结果。
- API 客户端的异步任务提交时间取决于提交参数的序列化及其网络传输时间。

注意：异步方式不适用于前后任务之间有依赖的场景。比如有两个任务，前一个任务向分布式数据库写入数据，后一个任务将新写入的数据结合历史数据做分析。像这类后一个任务对前一任务有依赖的场景，不能使用异步提交的方式。

Python API 开启 ASYNC（异步）模式的操作可以参照 xx 节仅为标注，之后添加跳转链接建立 DolphinDB 连接的部分，即设置 Session 的 *enableASYNC* 参数为 True。通过这种方式异步写入数据可以节省 API 端检测返回值的时间。

```
s = ddb.Session(enableASYNC=True)
```

以追加数据到分布式表为例，在 Python 中可以参考如下脚本使用异步方式追加数据。

```
import dolphindb as ddb
import dolphindb.settings as keys
import numpy as np
import pandas as pd

s = ddb.Session(enableASYNC=True) # 打开异步模式
s.connect("localhost", 8848, "admin", "123456")
dbPath = "dfs://testDB"
tbName = "tb1"


script = """
    dbPath="dfs://testDB"
    tbName=`tb1
    if(existsDatabase(dbPath))
        dropDatabase(dbPath)
    db=database(dbPath, VALUE, ["AAPL", "AMZN", "A"])
```

```
    testDictSchema=table(5:0, `id`ticker`price, [INT,STRING,DOUBLE])
    tb1=db.createPartitionedTable(testDictSchema, tbName, `ticker)
"""
s.run(script)    #此处脚本可以在服务器端运行

tb = pd.DataFrame({
    'id': [1, 2, 2, 3],
    'ticker': ['AAPL', 'AMZN', 'AMZN', 'A'],
    'price': [22, 3.5, 21, 26],
})

s.run(f"append!{{loadTable('{dbPath}', `{tbName})}}", tb)
```

注意：异步通讯的条件下，只能通过 `Session.run()` 方法与 DolphinDB 实现通讯，**并无返回值。**

示例1

由于在数据吞吐量较高的情况下使用异步的效果更佳，下面给出一个 Python API 写入流数据表的示例。

```
import dolphindb as ddb

import numpy as np
import pandas as pd
import random
import datetime

s = ddb.Session(enableASYNC=True)
s.connect("localhost", 8848, "admin", "123456")

n = 100

script = """
    share streamTable(10000:0,`time`sym`price`id, [TIMESTAMP,SYMBOL,DOUBLE,INT]) as trades
"""
s.run(script) # 此处的脚本可以在服务端直接运行

# 生成一个 DataFrame
time_list = [np.datetime64(datetime.date(2020, random.randint(1, 12), random.randint(1, 20))) for _ in range(n)]
sym_list = np.random.choice(['IBN', 'GTYU', 'FHU', 'DGT', 'FHU', 'YUG', 'EE', 'ZD', 'FYU'], n)
price_list = [round(np.random.uniform(1, 100), 1) for _ in range(n)]
```

```
id_list = np.random.choice([1, 2, 3, 4, 5], n)

tb = pd.DataFrame({
    'time': time_list,
    'sym': sym_list,
    'price': price_list,
    'id': id_list,
})

for _ in range(50000):
    s.run("tableInsert{trades}", tb)
```

Linux 系统中可以使用 time 命令来统计代码执行的时间，Windows 系统中则可以使用 Measure-Command 命令。在本例中使用 time 来统计代码执行时间。

同步模式下，即 `enableAsync=False` 时，测试上述代码的执行时间为 8.39 秒；异步模式下，即 `enableAsync=True` 时，测试上述代码的执行之间为 4.89 秒。对比可见，在同步模式下，频繁写入大量小数据，会有较大的网络 IO 开销；而使用异步模式则能解决这一问题，只需要将数据上传至服务端，而不需要返回值。在网络性能较差、任务数量较多的情况下，使用异步模式能够显著提升总体性能，但其缺点是无法保证数据能够正常写入，没有异常报错，因此难以发现问题。

示例2

由于异步模式的特殊性，在异步追加数据时，如果追加的数据需要进行时间类型的转换，不能直接调用 upload 提交数据到 DolphinDB，然后再在 DolphinDB 用 SQL 脚本进行类型转换，因为异步可能导致数据提交还未完成却已经开始执行 SQL 的情况。为了解决这个问题，建议先在 DolphinDB 中定义好函数视图，后续操作只需要调用该函数视图即可。以下为简单示例。

首先，在 DolphihinDB 中定义一个视图函数 appendStreamingData：

```
login("admin","123456")
share streamTable(10000:0,`time`sym`price`id, [DATE,SYMBOL,DOUBLE,INT]) as trades
def appendStreamingData(mutable data){
    tableInsert(trades, data.replaceColumn!(`time, date(data.time)))
}
addFunctionView(appendStreamingData)
```

然后在 Python API 异步追加数据：

```
import dolphindb as ddb
import numpy as np
import pandas as pd
import random
import datetime

s = ddb.Session(enableASYNC=True)
```

```
s.connect("localhost", 8848, "admin", "123456")

n = 100

# 生成一个 DataFrame
time_list = [np.datetime64(datetime.date(2020, random.randint(1, 12), random.randint(1, 20))) for _ in range(n)]
sym_list = np.random.choice(['IBN', 'GTYU', 'FHU', 'DGT', 'FHU', 'YUG', 'EE', 'ZD', 'FYU'], n)
price_list = [round(np.random.uniform(1, 100), 1) for _ in range(n)]
id_list = np.random.choice([1, 2, 3, 4, 5], n)

tb = pd.DataFrame({
    'time': time_list,
    'sym': sym_list,
    'price': price_list,
    'id': id_list,
})

for _ in range(50000):
    s.run("appendStreamingData", tb)
```

使用函数视图可以将数据作为函数参数传递给 DolphinDB，这也将极大方便 DolphinDB 中的数据清洗、类型转换等任务，结合使用异步模式可以进一步提高总体的性能。

MultithreadedTableWriter

针对单条数据批量写入的场景，DolphinDB Python API 提供 MultithreadedTableWriter（**推荐**）和 BatchTableWrite （**不推荐**） 两个类对象用于批量异步追加数据，并在客户端维护了一个数据缓冲队列。当服务器端忙于网络 I/O 时，客户端的写入线程仍然可以将数据持续写入缓冲队列（该队列由客户端维护），写入队列后即可返回，从而避免了写线程的忙等。目前，`BatchTableWrite` 支持批量写入数据到内存表、分区表；而 `MultithreadedTableWriter` 支持批量写入数据到内存表、分区表和维度表。

1. 工作原理

MultithreadedTableWriter，以下简称 **MTW**，主要用于批量写入单条数据。构造 MTW 时，需要指定后台写入线程数 *threadCount*，MTW 将会创建 *threadCount+1* 个后台 C++线程，包含1个转换线程和 *threadCount* 个写入线程。以下为简要的使用说明：

1. MTW 提供接口 insert 方法，该方法一次接收一条数据，将输入的 Python 对象放到待转换队列中，然后返回 ErrorCodeInfo。该过程中仅会校验写入数据列数是否和待写入表一致，**并不会进行类型检查。**如遇到写入数据列数不匹配、或写入线程已退出等情况，`ErrorCodeInfo.hasError()` 将会返回 True，表示执行 insert 时发生错误。
2. MTW 在构造时，会创建1个转换线程，用于将 Python 对象转换为 C++ 对象，减弱 Python GIL 锁在 C++ 多线程中的影响。在转换过程中，如果发生转换失败等错误，将会立刻结束所有后台线程（包括写入线程），并将错误信息记录至 MultithreadedTableWriterStatus，可以通过 MTW 的 getStatus 方法获得。转换成功的 C++对象，将分发至不同的写入线程中；转换失败的 Python 对象将会被收集起来，可以通过 MTW 的 getUnwrittenData 方法获得。
3. 后台创建的 *threadCount* 个写入线程按照构造 MTW 时指定的 *batchSize* 和 *throttle* 参数，将数据和写入脚本发送至 DolphinDB。同样的，如果在此过程中发生错误，也会终止所有后台线程，并将错误信息记录至 MultithreadedTableWriterStatus。
4. 调用 MTW 的 waitForThreadCompletion 方法时，类似于线程的 join，会阻塞等待后台写入任务全部执行完毕后，结束工作线程。**注意：调用结束后，原 MTW 将无法再次使用 insert 方法写入数据。**

2. MultithreadedTableWriter

```
MultithreadedTableWriter(host, port, userId, password, dbPath, tableName, useSSL=False,
                         enableHighAvailability=False, highAvailabilitySites=[], batchSize=1,
                         throttle=1, threadCount=1, partitionCol="", compressMethods=[],
                         mode="", modeOption=[])
```

参数介绍详见下方内容。

2.1 连接参数

- *host*：字符串，必填，表示所连接的服务器的 IP 地址。
- *port*：整数，必填，表示所连接的服务器的端口号。
- *userName*：字符串，可选，表示登录时的用户名。
- *password*：字符串，可选，表示登录时用户名对应的密码。
- *dbPath*：字符串，表示分布式数据库地址。内存表时该参数为空。**请注意，若使用1.30.17.4及以下版本的 API 向内存表写入数据时，该参数须填写内存表表名。**
- *tableName*：字符串，表示分布式表或内存表的表名。**请注意，若使用1.30.17.4及以下版本的 API 向内存表写入数据时，该参数须为空。**

- *useSSL*：布尔值，默认值为 False。表示是否启用加密通讯。和 Session 类似，当 MultithreadedTableWriter 启用加密通讯时，DolphinDB 需配置 `enableHTTPS=true`。
- *enableHighAvailability / highAvailabilitySites*：高可用配置参数。配置方式与 Session 一致，请参考章节xxx仅为标记，之后加跳转链接。

2.2 配置参数

- *batchSize*：整数，表示批处理的消息的数量，默认值是1，表示客户端写入数据后就立即发送给服务器。如果该参数大于 1，表示数据量达到 *batchSize* 时，客户端才会将数据发送给服务器。
- *throttle*：大于0的浮点数，单位为秒。若客户端有数据写入，但数据量小于 *batchSize*，则等待 *throttle* 的时间再发送数据。
- *threadCount*：整数，表示创建的工作线程数量，默认为1，表示单线程。若写入对象为内存表，则该参数值必须为1。
- *partitionCol*：字符串类型，默认为空，仅在 *threadCount* 大于1时起效。若写入对象为分区表，必须指定为分区字段名；若写入对象为流表，必须指定为表的字段名；若写入对象为维度表，该参数不起效。
- *compressMethods*：列表，用于指定每一列采用的压缩传输方式，若该参数为空则表示不压缩。每一列可选的压缩方式（大小写不敏感）包括：
  - "LZ4"： LZ4 压缩。
  - "DELTA"： DELTAOFDELTA 压缩。
- *mode*：字符串，表示数据写入的方式，可选值为 "Upsert" 或 "Append"。"Upsert" 表示以 upsert! 方式更新表数据；"Append" 表示以 tableInsert 方式更新表数据。
- *modeOption*：字符串，表示 upsert! 可选参数组成的 list，仅当 *mode* 指定为 "Upsert" 时有效。

2.3 构造示例

```
writer = ddb.MultithreadedTableWriter(
    "localhost", 8848, "admin", "123456", "dfs://testMTW", "pt", batchSize=2, throttle=0.01,
    threadCount=3, partitionCol="date", compressMethods=["LZ4", "DELTA", "LZ4", "DELTA"],
    mode="Upsert", modeOption=["true", "`index"]
)
```

以上示例展示了如何构造一个 MultithreadedTableWriter，其中设置了3个写入线程，用于写入数据库 dfs://testMTW 中的分区表 pt。设置参数 `batchSize=2`、`throttle=0.01`，一旦当前待写入队列的长度超过2或者等待时间超过0.01秒，就会立即触发写入线程，将写入队列中的所有数据发送到 DolphinDB。其中 *partitionCol* 被指定为 date 列，当数据分配到待写入队列时，将根据 date 列将数据分别分配到三个写入线程对应的队列中。由于待写入表有四列数据，分别指定了压缩方式为 LZ4、DELTA、LZ4、DELTA。写入模式被指定为 Upsert，并传入了 upsert! 函数对应的参数 ，即每次写入时调用的 upsert! 函数的 ignoreNull 参数为 true，keyColNames 为 `index。

2.4 insert

```
res:ErrorCodeInfo = writer.insert(*args)
```

功能说明：

插入单行数据。

返回一个 ErrorCodeInfo 对象，包含 errorCode 和 errorInfo，分别表示错误代码和错误信息。

“

ErrorCodeInfo 介绍

- *errorCode*：`""` 表示没有错误发生；如果有错误发生，值为错误码，例如：`"A2"`。
- *errorInfo*：表示错误信息。当没有错误发生时，返回 `None`；发生错误时，返回包含错误信息的字符串。
- hasError()：判断插入待转换队列时是否发生错误。如果发生错误，返回 `True`；反之，返回 `False`。
- succeed()：判断是否成功插入待转换队列。如成功插入，返回 `True`；反之，返回 `False`。

”

ErrorCodeInfo 类提供了 hasError 和 succeed 方法用于获取数据是否成功放入数据队列的结果。在插入一行数据后，推荐调用 hasError 来判断是否成功将数据插入到待转换队列。

参数说明：

- *args*：不定长位置参数，代表插入的一行数据。

示例：

下面展示一个简单的写入示例，在 insert 执行结束后，再使用 hasError 方法判断数据是否成功写入后台待转换队列。

```
import numpy as np
import pandas as pd
import dolphindb as ddb
import random

s = ddb.Session()
s.connect("localhost", 8848, "admin", "123456")

script = """
    t = table(1000:0, `date`ticker`price, [DATE,SYMBOL,LONG]);
    share t as tglobal;
```

```
"""
s.run(script)

writer = ddb.MultithreadedTableWriter(
    "localhost", 8848, "admin", "123456", dbPath="", tableName="tglobal",
    batchSize=10, throttle=1, threadCount=5, partitionCol="date"
)

for i in range(10):
    if i == 3:
        res = writer.insert(np.datetime64(f'2022-03-2{i%6}'), random.randint(1,10000))
    else:
        res = writer.insert(np.datetime64(f'2022-03-2{i%6}'), "AAAA", random.randint(1,10000))
    if res.hasError():
        print("insert error: ", res.errorInfo)

writer.waitForThreadCompletion()
print(s.run("""select count(*) from tglobal"""))

# output
"""
insert error:  Column counts don't match 3
   count
0      9
"""
```

在上例中，共写入10条数据，其中包含一条列数为2的数据，在插入这条数据时，`res.hasError()` 返回 True，表示插入的数据有误，打印后显示插入数据与待写入表列数不匹配。查询 tglobal 中数据条数为9，和预期一致。

2.5 getUnwrittenData

```
data:list = writer.getUnwrittenData()
```

功能说明：

返回一个嵌套列表，表示未写入服务器的数据，包含未转换的数据以及待发送的数据两部分。

注意：通过该方法获取到数据资源后，MultithreadedTableWriter 将释放这些数据资源。

2.6 insertUnwrittenData

```
res:ErrorCodeInfo = insertUnwrittenData(unwrittenData)
```

功能说明：

将数据插入数据表。

返回值同 insert 方法。与 insert 方法的区别在于，insert 只能插入单行数据，而 insertUnwrittenData 可以同时插入多行数据。通常 insertUnwrittenData 用于批量写回之前未成功写入的数据。

参数说明：

unwrittenData：需要再次写入的数据。一般通过方法 getUnwrittenData 获取该对象。

2.7 getStatus

```
status:MultithreadedTableWriterStatus = writer.getStatus()
```

功能说明：

获取 `MultithreadedTableWriter` 对象当前的运行状态。返回 `MultithreadedTableWriterStatus` 类对象。

“

MultithreadedTableWriterStatus 介绍

- *isExiting*：表示写入线程是否正在退出。
- *errorCode*：表示错误码。
- *errorInfo*：表示错误信息。
- *sentRows*：表示成功发送（写入表中）的总记录数。
- *unsentRows*：表示待发送的总记录数（包含待转换和已经转换但还未进入发送队列的数据）。
- *sendFailedRows*：表示发送失败的总记录数（发送队列中的数据）。
- *threadStatus*：表示写入线程状态列表。
  - *threadId*：表示线程 Id。
  - *sentRows*：表示该线程成功发送的记录数。
  - *unsentRows*：表示该线程待发送的记录数。
  - *sendFailedRows*：表示发送失败的记录数。
- hasError()：判断写入中是否发生错误，如果发生错误，返回 `True`；反之，返回 `False`。
- succeed()：判断是否写入成功，如写入成功，返回 `True`；反之，返回 `False`。

”

以一个 MultithreadedTableWriterStatus 打印输出为例：

```
errorCode     : A1
 errorInfo     : Data conversion error: Cannot convert double to LONG
 isExiting     : True
 sentRows      : 2493
 unsentRows    : 0
 sendFailedRows: 7507
 threadStatus  :
        threadId        sentRows        unsentRows      sendFailedRows
               0               0                 0                7507
        3567691520           415                 0                   0
        3489658624           831                 0                   0
        3481265920           416                 0                   0
        3472873216           416                 0                   0
        3464480512           415                 0                   0
<dolphindb.session.MultithreadedTableWriterStatus object at 0x7f0102c76d30>
```

以上输出内容显示，在本次 MTW 写入流程中发生了异常，异常代码是 `A1`，异常信息为 `Data conversion error: Cannot convert double to LONG`。`isExiting=True` 表示当前线程正在退出。`sentRows=2493` 表示当前有 2493 条数据已经写入到DolphinDB服务端；`unsentRows=0/sendFailedRows=7507` 则表示当前仍在API的未成功写入DolphinDB 服务端的数据一共有 `0+7507=7507` 条。在后面的表格中，列出了各个后台线程的处理情况，`threadId=0`表示所有线程的统计总数。通常而言，成功写入的结果（sentRows）会分别显示在各个线程中，例如上述输出中的 sentRows 列；写入失败的结果（unsentRows、sendFailedRows）会集中显示在 `threadId=0` 行，表示整个过程中的写入失败条数。

2.8 waitForThreadCompletion

```
writer.waitForThreadCompletion()
```

功能说明：

调用此方法后，MTW 会进入等待状态，待后台工作线程全部完成后再退出等待状态。

3. 常规流程

MultithreadedTableWriter 的常规处理流程如下：

```
# 准备数据 data
# 插入数据或其他任务
writer.insert(data)
...
# 等待 MTW 工作完成
writer.waitForThreadCompletion()
```

```
# 获取 MTW 工作状态，通过 writeStatus.hasError() 和 writeStatus.succeed() 判断是否正常执行完成
writeStatus=writer.getStatus()
if writeStatus.hasError():
    # 获取并修正失败数据后将失败数据重新写入MTW
    unwrittendata = writer.getUnwrittenData()
    unwrittendata = revise(unwrittendata)
    newwriter.insertUnwrittenData(unwrittendata)
else
    print("Write successfully!")
```

4. 使用示例

下例以写入分布式表为例，介绍 MultithreadedTableWriter 插入数据的总体流程：

步骤一： 创建 DolphinDB 数据库分布式表

```
import numpy as np
import pandas as pd
import dolphindb as ddb
import time
import random

s = ddb.Session()
s.connect("localhost", 8848, "admin", "123456")

script = """
    dbName = 'dfs://valuedb3';
    if(exists(dbName)){
        dropDatabase(dbName);
    }
    datetest=table(1000:0,`date`symbol`id,[DATE,SYMBOL,LONG]);
    db = database(directory=dbName, partitionType=HASH, partitionScheme=[INT, 10]);
    pt=db.createPartitionedTable(datetest,'pdatetest','id');
"""
s.run(script)
```

步骤二： 创建 MultithreadedTableWriter 对象

```
writer = ddb.MultithreadedTableWriter(
    "localhost", 8848, "admin", "123456", dbPath="dfs://valuedb3", tableName="pdatetest",
```

```
    batchSize=10000, throttle=1, threadCount=5, partitionCol="id", compressMethods=["LZ4","LZ4","DELTA"]
)
```

在这一步中，创建一个包含5个线程的 MTW 对象，同时设置 `batchSize=10000`，`throttle=1`，指定分区列为 "id"、各列压缩方式分别为 LZ4、LZ4、DELTA，向 dfs://valuedb3 分布式数据库中的分区表 pdatetest 写入数据。

步骤三： 写入数据

```
try:
    # 插入100行正确数据
    for i in range(100):
        res = writer.insert(random.randint(1,10000),"AAAAAAAB", random.randint(1,10000))
        if res.hasError():
            print("MTW insert error: ", res.errorInfo)
except Exception as ex:
    # MTW 抛出异常
    print("MTW exit with exception: ", ex)
# 等待 MTW 插入完成
writer.waitForThreadCompletion()
writeStatus = writer.getStatus()
if writeStatus.succeed():
    print("Write successfully!")
print("writeStatus: \n", writeStatus)
print(s.run("select count(*) from pt"))
```

调用 `writer.insert()` 方法向 writer 中写入数据，并通过 `writer.getStatus()` 获取 writer 的状态。

输出结果

```
Write successfully!
writeStatus:
 errorCode     : None
 errorInfo     :
 isExiting     : True
 sentRows      : 100
 unsentRows    : 0
 sendFailedRows: 0
 threadStatus  :
       threadId        sentRows        unsentRows      sendFailedRows
              0               0                 0                   0
       1677719296            22                 0                   0
       1669326592            20                 0                   0
       1660933888            19                 0                   0
       1652541184            14                 0                   0
```

```
       1644148480           25                0                 0
<dolphindb.session.MultithreadedTableWriterStatus object at 0x7fe896260d30>
   count
0    100
```

由上述结果可知，MultithreadedTableWriter 对象中的所有数据均成功写入分布式表，此时 errorCode 的输出为 "None"。

步骤四：错误处理

下述示例将介绍如果写入过程中发生错误，该如何通过 getStatus 和 try-catch 进行错误处理。

注1： 此时分区表中已有 100 条记录。

注2： 因为步骤三中 MTW 已经调用 waitForThreadCompletion 方法结束后台线程，所以在后续代码中需要重新构造一个新的 MTW 对象，才能继续写入数据。

```
writer = ddb.MultithreadedTableWriter(
    "localhost", 8848, "admin", "123456", dbPath="dfs://valuedb3", tableName="pdatetest",
    batchSize=10000, throttle=1, threadCount=5, partitionCol="id", compressMethods=["LZ4","LZ4","DELTA"]
)

try:
    # 插入100行正确数据 （类型和列数都正确），MTW正常运行
    for i in range(100):
        res = writer.insert(np.datetime64('2022-03-23'),"AAAAAAAB", random.randint(1,10000))
    # 插入10行类型错误数据，此时 MTW 并不会进行类型判断，这些数据能够进入 MTW 待转换队列
    # 直到转换线程对这些数据进行转换时，检测到类型不匹配，就会立刻终止 MTW 所有后台线程
    for i in range(10):
        res = writer.insert(np.datetime64('2022-03-23'),222, random.randint(1,10000))
        if res.hasError():
            # 此处不会执行到
            print("Insert wrong format data:\n", res)
    # 插入1行数据(列数不匹配)，MTW 立刻发现待插入数据列数与待插入表的列数不匹配，立刻返回错误信息，
    # 本条数据并不会进入待转换队列
    res = writer.insert(np.datetime64('2022-03-23'),"AAAAAAAB")
    if res.hasError():
        # 数据错误，插入列数不匹配数据
        print("Column counts don't match:\n", res)
    # sleep 1秒，等待 MTW 转换线程处理数据直至检测到第2次插入的10行数据类型不匹配
    # 此时 MTW 立刻终止所有线程，并修改状态为错误状态
    time.sleep(1)

    # 再插入1行正确数据，MTW 会因为工作线程终止而抛出异常，且不会写入该行数据
```

```
    res = writer.insert(np.datetime64('2022-03-23'),"AAAAAAAB", random.randint(1,10000))
    print("MTW has exited")
except Exception as ex:
    # MTW 抛出异常
    print("MTW exit with exception %s" % ex)
# 等待 MTW 插入完成
writer.waitForThreadCompletion()
writeStatus = writer.getStatus()
if writeStatus.hasError():
    print("Error in writing:")
print(writeStatus)
print(s.run("select count(*) from pt"))
"""
Column counts don't match:
 errorCode: A2
 errorInfo: Column counts don't match 3
<dolphindb.session.ErrorCodeInfo object at 0x7f0d52947040>
MTW exit with exception <Exception> in insert: thread is exiting.
Error in writing:
errorCode     : A1
 errorInfo     : Data conversion error: Cannot convert long to SYMBOL
 isExiting     : True
 sentRows      : 0
 unsentRows    : 4
 sendFailedRows: 106
 threadStatus  :
        threadId        sentRows        unsentRows      sendFailedRows
               0               0                 0                 106
        511690496              0                 2                   0
        520083200              0                 1                   0
        528475904              0                 0                   0
        618751744              0                 0                   0
        536868608              0                 1                   0
<dolphindb.session.MultithreadedTableWriterStatus object at 0x7f0d25d68910>
   count
0    100
"""
```

步骤五：重新写入

当 MTW 发生类型转换错误、写入线程写入失败等错误时，API 将会终止所有线程。此时可以通过 `writer.getUnwrittenData()` 方法获取失败数据，通过 insertUnwrittenData(unwrittendata) 重新写入失败数据。因为原有 MTW 对象的所有线程已经终止，无法再次被用来写入数据，因此需要重新构造一个新的 MTW 对象，将失败数据重新写入新的 MTW 对象中。

```
if writeStatus.hasError():
    print("Error in writing:")
    unwrittendata = writer.getUnwrittenData()
    print("Unwrittendata: %d" % len(unwrittendata))
    # 重新构造新的 MTW 对象
    newwriter = ddb.MultithreadedTableWriter(
        "localhost", 8848, "admin", "123456", dbPath="dfs://valuedb3", tableName="pdatetest",
        batchSize=10000, throttle=1, threadCount=5, partitionCol="id", compressMethods=["LZ4","LZ4","DELTA"]
    )
    try:
        # 修正失败数据后将失败数据重新写入 MTW
        for row in unwrittendata:
            row[1]="aaaaa"
        res = newwriter.insertUnwrittenData(unwrittendata)
        if res.hasError():
            print("Failed to write data again: \n", res)
    except Exception as ex:
        # MTW 抛出异常
        print("MTW exit with exception %s" % ex)
    finally:
        # 确保 newwriter工作线程结束运行
        newwriter.waitForThreadCompletion()
        writeStatus = newwriter.getStatus()
        print("Write again:\n", writeStatus)
else:
    print("Write successfully:\n", writeStatus)

print(s.run("select count(*) from pt"))
"""
Error in writing:
Unwrittendata: 110
Write again:
 errorCode     : None
 errorInfo     :
 isExiting     : True
```

```
 sentRows      : 110
 unsentRows    : 0
 sendFailedRows: 0
 threadStatus  :
        threadId        sentRows        unsentRows      sendFailedRows
               0               0                 0                   0
        618751744             17                 0                   0
        528475904             21                 0                   0
        520083200             24                 0                   0
        511690496             25                 0                   0
        503297792             23                 0                   0
<dolphindb.session.MultithreadedTableWriterStatus object at 0x7f0d52947040>
   count
0    210
"""
```

5. Python 多线程调用

MTW 内部使用 C++ 多线程完成数据转换和写入任务，同时也保证 Python 多线程调用 MTW 的线程安全。具体示例如下：

```
import numpy as np
import pandas as pd
import dolphindb as ddb
import time
import threading
import random

s = ddb.Session()
s.connect("localhost", 8848, "admin", "123456")

script = """
    dbName = 'dfs://valuedb3';
    if(exists(dbName)){
        dropDatabase(dbName);
    }
    datetest=table(1000:0,`date`symbol`id,[DATE,SYMBOL,LONG]);
    db = database(directory=dbName, partitionType=HASH, partitionScheme=[INT, 10]);
    pt=db.createPartitionedTable(datetest,'pdatetest','id');
"""
s.run(script)
```

```
writer = ddb.MultithreadedTableWriter(
    "localhost", 8848, "admin", "123456", dbPath="dfs://valuedb3", tableName="pdatetest",
    batchSize=10000, throttle=1, threadCount=5, partitionCol="id", compressMethods=["LZ4","LZ4","DELTA"]
)

def insert_MTW(writer):
    try:
        # 插入100行正确数据
        for i in range(100):
            res = writer.insert(random.randint(1,10000),"AAAAAAAB", random.randint(1,10000))
    except Exception as ex:
        # MTW 抛出异常
        print("MTW exit with exception %s" % ex)

# 创建 10 个线程，在线程中将数据写入MTW
threads = []
for i in range(10):
    threads.append(threading.Thread(target=insert_MTW, args=(writer,)))

for thread in threads:
    thread.start()

# 完成其他任务，此处用 sleep 模拟
time.sleep(10)

# 现在需要结束任务
# 1 - 等待线程退出
for thread in threads:
    thread.join()
# 2 - 等待 MTW 线程结束
writer.waitForThreadCompletion()
# 3 - 检查写入结果
writeStatus = writer.getStatus()
print("writeStatus:\n", writeStatus)
print(s.run("select count(*) from pt"))
```

```
writeStatus:
 errorCode     : None
 errorInfo     :
 isExiting     : True
```

```
sentRows      : 1000
unsentRows    : 0
sendFailedRows: 0
threadStatus  :
       threadId        sentRows        unsentRows      sendFailedRows
              0               0                 0                   0
       1309443840           194                 0                   0
       1301051136           188                 0                   0
       1292658432           215                 0                   0
       1284265728           208                 0                   0
       1275873024           195                 0                   0
<dolphindb.session.MultithreadedTableWriterStatus object at 0x7ff2802bf9a0>
   count
0   1000
```

本例中演示了如何在 Python 多线程中使用 MTW 对象写入数据。其中开启10个 Python 线程，并且在每个线程中写入 100 条数据。从结果可以看出，成功写入 1000 条数据。

BatchTableWriter

“

注意：BatchTableWriter 现已不再维护，不推荐使用。

”

1. 工作原理

BatchTableWriter 以下简称 **BTW**，用于批量写入单条数据。以下为简要的使用说明：

1. BTW 提供接口 addTable 方法，用于添加一个写入表，执行该方法后，将会创建一个写入队列和一个 C++ 工作线程。
2. 写入数据时，需要调用 insert 方法，并指定写入表的信息和待写入的一行数据。将数据转换为 C++ 对象后放入后台写入队列中。
3. 工作线程每隔100 ms 取出写入队列中的所有数据，一次性将这一批数据全部写入 DolphinDB。
4. 如果写入数据时（insert）发生类型转换错误或者列数不匹配等错误，会立刻抛出异常信息，该条数据不会放入写入队列中；如果在工作线程中将数据发送到 DolphinDB 的过程中发生错误，则会将当前写入队列中的数据和发送失败的数据保存起来，调用 getUnwrittenData 即可获得。
5. 如果需要查看当前各表的写入状态，可以调用 getAllStatus 方法，该方法将返回一张表，表示各个待写入表的工作状态，包含写入队列长度、发送行数、是否已销毁、是否结束。也可以调用 getStatus，传入表名和数据库名，就可以获得指定表当前的写入状态。
6. 不需要再往表中写数据时，可以调用 removeTable。该方法将停止工作线程，然后销毁相关的工作线程、连接、写入队列。

2. BatchTableWriter

```
BatchTableWriter(host, port, userid=None, password=None, acquireLock=True)
```

2.1 参数说明

- *host*：字符串，必填，表示所连接的服务器的 IP 地址。
- *port*：整数，必填，表示所连接的服务器的端口号。
- *userName*：字符串，可选，表示登录时的用户名。
- *password*：字符串，可选，表示登录时用户名对应的密码。
- *acquireLock*: 布尔值，表示在使用过程中是否对 BTW 内部加锁。默认为 True，表示需要加锁。若在并发调用 API 的场景下，建议加锁。

构造示例：

```
writer = ddb.BatchTableWriter("localhost", 8848, "admin", "123456", acquireLock=True)
```

2.2 addTable

```
writer.addTable(dbPath=None, tableName=None, partitioned=True)
```

功能说明：

向 BTW 中添加一个待写入的表，如果是分区表则需要设置 `partitioned=True`。

参数说明：

- *dbName*：表示数据库名。若待写入表为磁盘表时，须填写该参数；若为内存表，则不需要填写该参数。
- *tableName*：表示数据表的表名。
- *partitioned*：表示添加的表是否为分区表。若设置该参数为 True，则表示为分区表。如果添加的表是磁盘未分区表，须设置 `partitioned=False`。

注意:

- 如果添加的是内存表，则需要共享该表。
- 表名不可重复添加。如果重复添加，则需要先移除之前添加的表，否则会抛出异常。

2.3 insert

```
writer.insert(dbPath=None, tableName=None, *args)
```

功能说明：

向指定表中插入单行数据。

参数说明：

- *dbPath*：表示数据库名。若待写入表为磁盘表时，须填写该参数；若为内存表，则不需要填写该参数。
- *tableName*：表示数据表的表名。
- *args*：变长参数，表示插入的一行数据。

注意：

- 在调用 insert 前须先使用 addTable 添加表，否则会抛出异常。
- 变长参数的个数和数据类型必须与 insert 表的列数及类型匹配。
- 若在插入过程中出现异常进而导致后台线程退出，此时再次调用 insert 将会抛出异常。建议使用 getUnwrittenData 来获取之前所有已经写入缓冲队列但是没有成功写入服务器的数据（不包括本次 insert 的数据），然后使用 removeTable 释放资源。如果需要再次插入数据，则须重新调用 addTable。
- 在移除指定表的过程中调用 insert 仍能成功插入数据，但插入的这部分数据并不会发送到服务器。该操作属于未定义行为，不建议用户进行类似操作。

2.4 removeTable

```
writer.removeTable(dbPath=None, tableName=None)
```

功能说明：

释放由 addTable 添加的表所占用的资源。

第一次调用该函数，若该函数返回即表示后台线程已退出。再次写入需要重新调用 addTable。

2.5 getUnwrittenData

```
data:pandas.DataFrame = writer.getUnwrittenData(dbPath=None, tableName=None)
```

功能说明：

获取还未写入的数据。

当写入出现错误时，使用该函数可以获取剩余未写入的数据。但是这些未写入的数据不会尝试重写，若需要重新写入，则需要使用 removeTable 后重新调用 addTable，再通过 insert 写入数据。

2.6 getStatus

```
res:list = writer.getStatus(dbPath=None, tableName=None)
```

功能说明：

获取当前的写入状态。详细信息可参考下方返回值说明。

返回值说明：

返回值是由一个整型和两个布尔型组合的列表，分别表示当前写入队列的深度、当前表是否被移除（True: 表示正在被移除），以及后台写入线程是否因为出错而退出（True: 表示后台线程因出错而退出）。

2.7 getAllStatus

```
res:pandas.DataFrame = writer.getAllStatus()
```

功能说明：

获取所有当前存在的表的信息，不包含被移除的表。

返回值说明：

该函数的返回值是一个2*6的表格，包含数据库名（DatabaseName）、数据表名（TableName）、写入队列长度（WriteQueueDepth）、发送行数（sendedRows）、是否已销毁（Removing）和是否结束（Finished）。返回值的示例如下表：

Database-Name	Table-Name	Write-QueueDepth	sended-Rows	Removing	Finished
0	tglobal	0	5	False	False

3. 使用示例

本例中，首先定义一个共享表 tglobal，然后构造 BTW，调用 addTable 将 tglobal 加入待写入表。随后调用 insert 方法写入五条数据，然后**立刻**调用 getUnwritternData、getStatus、getAllStatus 获取尚未写入的数据、当前表的写入状态和所有表的写入状态。最后，使用 Session 查询待写入表中的当前写入数据。

注意： 示例中，执行 insert 后立刻调用 getUnwrittenData，写入队列中的数据全部被取出，因此，此时 BTW 的写入队列中没有数据，仅有之前已经从写入队列取出、进入工作线程的数据成功写入 DolphinDB。

```
import dolphindb as ddb
import numpy as np
```

```
import pandas as pd

s = ddb.Session()
s.connect("localhost", 8848, "admin", "123456")

script = """
    t = table(1000:0,`id`date`ticker`price, [INT,DATE,SYMBOL,DOUBLE]);
    share t as tglobal;
"""
s.run(script)

writer = ddb.BatchTableWriter("localhost", 8848)
writer.addTable(tableName="tglobal")
writer.insert("","tglobal", 1, np.datetime64("2019-01-01"),'AAPL', 5.6)
writer.insert("","tglobal", 2, np.datetime64("2019-01-01"),'GOOG', 8.3)
writer.insert("","tglobal", 3, np.datetime64("2019-01-02"),'GOOG', 4.2)
writer.insert("","tglobal", 4, np.datetime64("2019-01-03"),'AMZN', 1.4)
writer.insert("","tglobal", 5, np.datetime64("2019-01-05"),'AAPL', 6.9)

print(writer.getUnwrittenData(dbPath="", tableName="tglobal"))
print(writer.getStatus(tableName="tglobal"))
print(writer.getAllStatus())

print("rows:", s.run("tglobal.rows()"))
print(s.run("select * from tglobal"))
```

输出结果如下：

```
   id       date ticker  price
0   2 2019-01-01   GOOG    8.3
1   3 2019-01-02   GOOG    4.2
2   4 2019-01-03   AMZN    1.4
3   5 2019-01-05   AAPL    6.9
[0, False, False]
  DatabaseName TableName  WriteQueueDepth  SendedRows  Removing  Finished
0               tglobal                0           1     False     False
rows: 1
   id       date ticker  price
0   1 2019-01-01   AAPL    5.6
```

Chapter 3. 进阶操作

PROTOCOL_DDB

PROTOCOL_PICKLE

PROTOCOL_ARROW

强制类型转换

类型转换

多种写入方案

流订阅模式

3.4.1 Table

3.4.2 Database

3.5.1 强制终止进程

PROTOCOL_DDB

PROTOCOL_DDB 作为 DolphinDB 自定义的一套数据序列化、反序列化方案，该协议广泛使用于 Python API、C++ API、Java API 等 API 中，且其支持的**数据形式**和**数据类型**最为全面。

注1：数据形式指 DolphinDB 类型系统中的 DATAFORM，表示数据结构的形式，通常包含 Scalar、Vector、Table 等。

注2：数据类型指 DolphinDB 类型系统中的 DATATYPE，表示数据的具体类型，通常包含 INT、DOUBLE、DATETIME 等。

注3：以下简称 Python 库 NumPy 为 **np**，pandas 为 **pd**。

1. 启用 PROTOCOL_DDB

在以下示例中，Session 和 DBConnectionPool 通过设置参数 *protocol* 指定启用 PROTOCOL_DDB 协议。

```
import dolphindb as ddb
import dolphindb.settings as keys

s = ddb.Session(protocol=keys.PROTOCOL_DDB)
s.connect("localhost", 8848, "admin", "123456")

pool = ddb.DBConnectionPool("localhost", 8848, "admin", "123456", 10, protocol=keys.PROTOCOL_DDB)
```

2. PROTOCOL_DDB 数据形式支持表

PROTOCOL_DDB 支持的数据形式如下表展示：

附加参数	数据形式	序列化	反序列化
pickleTableTo-List=False	Scalar	支持	支持
pickleTableTo-List=False	Vector	支持	支持
pickleTableTo-List=False	Pair	支持	支持
pickleTableTo-List=False	Matrix	支持	支持
pickleTableTo-List=False	Set	支持	支持
pickleTableTo-List=False	Dict	支持	支持
pickleTableTo-List=False	Table	支持	支持
pickleTableTo-List=True	Table	不支持	支持

3. 序列化 Python->DolphinDB

本节以 upload 函数为例，介绍在使用协议 PROTOCOL_DDB 上传 Scalar、Vector，Pair 等类型数据时分别对应的 DolphinDB 数据类型。

3.1 **Scalar**

下表展示上传 Scalar 型数据时，DolphinDB 与 Python 对应的数据类型与示例数据。

Python 类型	Python 示例数据	DolphinDB 类型	DolphinDB 示例数据
NoneType	None	VOID	NULL
bool	True	BOOL	true
np.int8	np.int8(12)	CHAR	char(12)
np.int16	np.int16(12)	SHORT	short(12)
np.int32	np.int32(12)	INT	int(12)
np.int64	np.int64(12)	LONG	long(12)
int	12	LONG	long(12)
np.datetime64	np.datetime64("2012-01-02", "D")	DATE	date(2012.01.02)
np.datetime64	np.datetime64("2012-01", "M")	MONTH	month(2012.01)
---	---	TIME	---
---	---	MINUTE	---
---	---	SECOND	---
np.datetime64	np.datetime64("2012-01-02T01:02:03", "s")	DATETIME	datetime(2012.01.02T01:02:03)
np.datetime64	np.datetime64("2012-01-02T01:02:03.123", "ms")	TIMESTAMP	timestamp(2012.01.02T01:02:03.123)
---	---	NANOTIME	---
np.datetime64	np.datetime64("2012-01-02T01:02:03.123456789", "ns")	NANOTIMES-TAMP	nanotimestamp(2012.01.02T01:02:03.123456789)

Python 类型	Python 示例数据	DolphinDB 类型	DolphinDB 示例数据
np.datetime64	np.datetime64("")	NANOTIMES-TAMP	nanotimestamp(NULL)
pd.Timestamp	pd.Timestamp("2012-01-02T01:02:03.123456789")	NANOTIMES-TAMP	nanotimestamp(2012.01.02T01:02:03.123456789)
pd.NaTType	pd.NaT	NANOTIMES-TAMP	nanotimestamp(NULL)
np.float32	np.float32(1.1)	FLOAT	float(1.1)
np.float64	np.float64(1.2)	DOUBLE	double(1.2)
float	1.2	DOUBLE	double(1.2)
float	np.nan	DOUBLE	double(NULL)
str	"abc"	STRING	"abc"
str	""	STRING	""
---	---	SYMBOL	---
---	---	UUID	---
np.datetime64	np.datetime64("2012-01-02T01", "h")	DATEHOUR	2012.01.02T01
---	---	IPADDR	---
bytes	bytes("abc", encoding="UTF-8")	BLOB	"abc"
---	---	DECIMAL32	---
Decimal	decimal.Decimal("-10.21")	DECIMAL64	decimal64(-10.21, 2)
Decimal	decimal.Decimal("NaN")	DECIMAL64	decimal64(NULL, 2)

3.2 Vector

和 Scalar 类似，由于 DolphinDB 与 Python 的类型系统并不是一一对应的关系，因此 API 无法直接向 DolphinDB 上传 TIME、UUID 等类型的 Vector。

类型转换的处理逻辑如下：

1. 判断当前对象是否为 Vector 型数据，如 tuple、list、np.ndarray、pd.Series等。
2. 如果当前对象为 np.ndarray 等类型确定的对象时，且 np.ndarray 的 dtype 不为 object，则直接进行类型转换。该种情况下的数据转换效率较高。
3. 如果当前对象为 list 等类型不确定的对象，或者是 dtype 为 object 的 np.ndarray 对象，则需要遍历一遍上传的数据后再进行判断。在此过程中，
    - 若数据中包含空值（None, np.nan, pd.NaT），则将根据第一个非空值类型确认该 Vector 型数据的类型。
    - 若遍历过程中发现有 **两种及两种以上（注1）** 的数据类型数据，或者包含 Vector 型数据，则将会判断为 Any Vector。该种情况下需要逐个对数据进行类型转换，且需要额外遍历，转换效率较差。
4. 在判断 Vector 型数据的元素类型时，基本遵循和 Scalar 一致的判断规则，因此，Python API 也不支持直接上传 MINUTE、UUID 等类型。
5. 如果一个 np.ndarray 对象中的元素都为长度相等的 Vector 型数据，则会优先判断为 Matrix 类型。
6. 如果上传的对象不是 np.ndarray，但是却包含长度一致的 Vector 型数据，则会作为 Any Vector 数据上传。
7. 如果上传的 Vector 中全为空值，且包含不同类型的空值，则将按照空值处理规则进行转换，如下表展示：

“

空值组合情况	DolphinDB 列字段类型
全部为 None	STRING
np.nan 与 None 组合	DOUBLE
pd.NaT 与 None 组合	NANOTIMESTAMP
np.nan 与 pd.NaT 组合	NANOTIMESTAMP
None, np.nan 与 pd.NaT 组合	NANOTIMESTAMP
None, pd.NaT 或 np.nan 与非空值组合	非空值类型

”

注1： 此处“两种及两种以上”特指 DolphinDB 类型系统中的类型，例如数据为 np.array() ，则将被视为仅包含一种数据类型。

注2： 目前版本的 DolphinDB Python API 并不支持 pd.array 的 Vector 型数据。

注3： 在 list 或是 dtype 为 object 的 np.ndarray 对象中的空值类型，将全部被视为同一种空值，例如 None/np.nan/pd.NaT 等，但不包含 decimal.Decimal('NaN')。

注4： 目前不支持直接上传 Array Vector 型数据，形如 np.array([,]) 的数据会作为 Any Vector 数据上传。

注5： 上传 Decimal 类型的数据时，须确保 DECIMAL 类型列的所有数据具有相同的小数位数。否则将根据第一个非空 Decimal 数据来确定精度。可使用下述方式对齐小数位数：

“

```
>>> b = decimal.Decimal("1.23")
>>> b
Decimal('1.23')
>>> b = b.quantize(decimal.Decimal("0.000"))
>>> b
Decimal('1.230')
```

”

下面给出3个常见的上传 Vector 型数据的例子：

示例1 上传不包含空值的 BOOL Vector、INT Vector、DOUBLE Vector、STRING Vector、DATE Vector

相关代码示例：

```
>>> s.upload({'bool_v': np.array([True, False, False], dtype="bool")})
>>> s.upload({'int_v': np.array([1, 2, 4], dtype="int32")})
>>> s.upload({'double_v': [1.2, 2.456]})
>>> s.upload({'string_v': np.array(["abc", "123"], dtype="object")})
>>> s.upload({'date_v': np.array(["2012-01-02"], dtype="datetime64[D]")})
```

可使用 typestr 方法查看已上传变量的数据类型：

```
>>> s.run("typestr(bool_v)")
FAST BOOL VECTOR
>>> s.run("typestr(int_v)")
FAST INT VECTOR
>>> s.run("typestr(double_v)")
FAST DOUBLE VECTOR
>>> s.run("typestr(string_v)")
STRING VECTOR
```

```
>>> s.run("typestr(date_v)")
FAST DATE VECTOR
```

示例2 上传包含空值的 BOOL Vector、INT Vector、DOUBLE Vector、STRING Vector、DATE Vector

相关代码示例：

```
>>> s.upload({'bool_v': [True, None, False]})
>>> s.upload({'int_v': np.array([None, np.int32(2), np.int32(12)], dtype="object")})
>>> s.upload({'double_v': np.array([1.1, np.nan, 3.456])})
>>> s.upload({'string_v': ["", "abc", "123"]})
>>> s.upload({'date_v': [pd.NaT, None, np.nan, np.datetime64("2012-01-03", "D")]})
```

可使用 typestr 方法查看上传变量的数据类型：

```
>>> s.run("typestr(bool_v)")
FAST BOOL VECTOR
>>> s.run("typestr(int_v)")
FAST INT VECTOR
>>> s.run("typestr(double_v)")
FAST DOUBLE VECTOR
>>> s.run("typestr(string_v)")
STRING VECTOR
>>> s.run("typestr(date_v)")
FAST DATE VECTOR
```

示例3 上传 Any Vector

上传 Any Vector 有两种方式。一种是在构造 np.ndarray 时指定 `dtype=object`；另一种则是上传一个 list/tuple 对象。须注意，这两种方式都需要包含 **两种及两种以上** 类型的数据或者包含 Vector 型的数据。

相关代码示例：

```
>>> s.upload({'list_v': [1.2, "abc"])})
>>> s.upload({'array_v': np.array([1, 1.2], dtype="object"})
>>> s.upload({'list_av': [[1, 2], [3]]})
>>> s.upload({'array_av': np.array([[1], [2, 3]], dtype="object")})
```

使用 typestr 方法查看上传变量的数据类型：

```
>>> s.run("typestr(list_v)")
ANY VECTOR
```

```
>>> s.run("typestr(array_v)")
ANY VECTOR
>>> s.run("typestr(list_av)")
ANY VECTOR
>>> s.run("typestr(array_av)")
ANY VECTOR
```

3.3 Pair

在使用 PROTOCOL_DDB 协议时，DolphinDB Python API 暂不支持直接上传 Pair 形式的数据。

3.4 Matrix

如果上传的 np.ndarray 对象中包含长度一致的 Vector 型数据，则将会作为 Matrix 形式的数据上传。但是，如果上传的对象不是 np.ndarray，却包含长度一致的 Vector 型数据，则将会作为 Any Vector 数据上传。

上传 Matrix 型数据的相关代码示例：

```
>>> s.upload({'int_m': np.array([[1, 2], [2, 3], [3, 4]])})
>>> s.run("typestr(int_m)")
FAST LONG MATRIX
>>> s.upload({'any_vec': [[1, 2], [2, 3], [3, 4]]})
>>> s.run("typestr(any_vec)")
ANY VECTOR
```

3.5 Set

上传 set 类型的 Python 对象，数据将会转换为 DolphinDB 中的 Set 形式。在转换时 API 将会遍历 set 中的所有元素，将各元素作为 Scalar 数据进行逐个转换，转换后的数据类型以对应的 DolphinDB 数据类型为准。

上传 Set 型数据的相关代码示例：

```
>>> s.upload({'long_set': {1, 2}})
>>> s.run("typestr(long_set)")
LONG SET
>>> s.upload({'double_set': {1.2, np.double(5.5), pd.NaT}})
>>> s.run("typestr(double_set)")
DOUBLE SET
```

注1： DolphinDB 的 Set 数据形式目前不支持元素有多种数据类型，也不支持 Vector 作为组成元素。

注2： 在转换时，若 Set 中包含空值，则空值将不被视为任何一种数据类型。此外，DolphinDB 也不支持构造全为空值的 Set。

3.6 Dict

和 Set 类似，在转换 Dict 型数据时 API 会遍历所有元素，按照对应的数据形式逐个进行类型转换，最后数据整体作为一个 Dictionary 上传至 DolphinDB。

上传 Dict 型数据的相关代码示例：

```
>>> s.upload({'long_dict': {'a': None, 'b': 1}})
>>> s.run("typestr(long_dict)")
STRING->LONG DICTIONARY
>>> s.upload({'any_dict1': {'a': 1, 'b': [1.1, 2.4], 'c': np.array([1, "a"], dtype="object")}})
>>> s.run("typestr(any_dict1)")
STRING->ANY DICTIONARY
>>> s.upload({'any_dict2': {1: [[1], [2, 3]], 2: [[1.1, np.nan], [3.3]]}})
>>> s.run("typestr(any_dict2)")
STRING->ANY DICTIONARY
```

注1： DolphinDB 不支持多种数据类型作为键，支持多种数据类型作为值。

注2： 在 Dict 中转换时，Vector 型数据中如果包含 Vector 型元素，将作为 Any Vector 进行转换，而非 Array Vector。

注3： Dict 中的空值在转换时不视为任何一种数据类型。DolphinDB 也不支持构造全为空值的 Dict。

3.7 Table

Table 型数据在 Python 中对应的数据类型为 pd.DataFrame，在转换时，按列进行处理，每列作为 Vector 型数据进行转换。特别的，当 pd.DataFrame 的元素中出现 Vector 型数据时，将作为 Array Vector 进行处理，且不支持 Any Vector 作为列类型，这点与 Vector 型数据的转换逻辑有较大不同。

当 Table 型数据中包含空值时，处理逻辑与 Vector 型数据保持一致。

注意： pd.DataFrame 中仅有一种时间类型 datetime64，对应 np.datetime64，因此，如果直接上传时间类型数据，在 DolphinDB 服务器端仅能得到 NANOTIMESTAMP 类型的数据。

示例1：

```
>>> df1 = pd.DataFrame({
...     'int_v': [1, 2, 3],
...     'long_v': np.array([None, 3, np.int64(3)], dtype="object"),
...     'float_v': np.array([np.nan, 1.2, 3.3], dtype="float32")
... })
>>> s.upload({'df1': df1})
>>> s.run("schema(df1)")['colDefs']
      name typeString  typeInt  extra comment
```

```
0    int_v       LONG        5    NaN
1   long_v       LONG        5    NaN
2  float_v      FLOAT       15    NaN
```

当上传的数据 `dtype=object` 时，会逐个判断数据类型，空值（None、pd.NaT、np.nan）不视为任何一种数据类型，并根据第一个非空值的数据类型作为该列的数据类型。

本例中，int_v 列没有空值，但是在 Python pandas 中，Python 的 int 会被转换为 int64，对应 DolphinDB 的 LONG；long_v 列 `dtype=object`，且第一个非空值为 Python int 3，对应 DolphinDB LONG，因此该列作为 LONG 进行类型转换； float_v 列 dtype=float，因此无论是否包含空值，都会作为 FLOAT 处理。

注意： numpy 和 pandas 中，dtype=bool/int8/int16/int32/int64 时，不能包含空值，因此，如果需要上传包含空值的列，则需要在构造 pd.DataFrame 时，指定该列类型为 object。

示例2：

```
>>> df2 = pd.DataFrame({
...     'day_v': np.array(["2012-01-02", "2022-02-05"], dtype="datetime64[D]"),
...     'month_v': np.array([np.datetime64("2012-01", "M"), None], dtype="datetime64[M]"),
... })
>>> s.upload({'df2': df2})
>>> s.run("schema(df2)")['colDefs']
      name     typeString  typeInt  extra comment
0    day_v  NANOTIMESTAMP       14    NaN
1  month_v  NANOTIMESTAMP       14    NaN
```

对于时间类型，pandas 中仅有一种时间类型 datetime64，因此无法直接上传 DATE、MONTH等类型，需要指定 __DolphinDB_Type__ 进行强制类型转换，才能上传。

```
>>> import dolphindb.settings as keys
>>> df2.__DolphinDB_Type__ = {
...     "day_v": keys.DT_DATE,
...     "month_v": keys.DT_MONTH,
... }
...
>>> s.upload({'df2': df2})
>>> s.run("schema(df2)")['colDefs']
      name typeString  typeInt  extra comment
0    day_v       DATE        6    NaN
1  month_v      MONTH        7    NaN
```

示例3：

```
>>> df3 = pd.DataFrame({
...     'long_av': [[1, None], [3]],
...     'double_av': np.array([[1.1], [np.nan, 3.3]], dtype="object")
... })
>>> s.upload({'df3': df3})
>>> s.run("schema(df3)")['colDefs']
        name typeString  typeInt  extra comment
0    long_av     LONG[]       69    NaN
1  double_av   DOUBLE[]       80    NaN
```

如上例所示，当上传的 pd.DataFrame 中某一列包含 Vector 型数据时，该列将作为 Array Vector 进行类型转换，而非 Any Vector。

4. 反序列化 DolphinDB -> Python(设置 pickleTableToList=False)

以下示例以 run 函数为例，介绍不同 DolphinDB 数据类型和数值通过该方式下载时对应的 Python 对象。

注： 由于 numpy 和 pandas 具有 dtype，故下文中，Python 类型中的 np.datetime64，表示该对象的类型为 numpy.datetime64，且 dtype=datetime64。

4.1 **Scalar**

DolphinDB类型	DolphinDB数据	Python类型	Python数据
VOID	NULL	NoneType	None
INT	int(NULL)	NoneType	None
STRING	string(NULL)	NoneType	None
BOOL	true	bool	True
CHAR	'a'	int	97
SHORT	224h	int	224
INT	16	int	16
LONG	3000l	int	3000
DATE	2013.06.13	np.datetime64	2013-06-13
MONTH	2012.06M	np.datetime64	2012-06

DolphinDB类型	DolphinDB数据	Python类型	Python数据
TIME	13:30:10.008	np.datetime64	1970-01-01T13:30:10.008
MINUTE	13:30m	np.datetime64	1970-01-01T13:30
SECOND	13:30:10	np.datetime64	1970-01-01T13:30:10
DATETIME	2012.06.13T13:30:10	np.datetime64	2012-06-13T13:30:10
TIMESTAMP	2012.06.13T13:30:10.008	np.datetime64	2012-06-13T13:30:10.008
NANOTIME	13:30:10.008007006	np.datetime64	1970-01-01T13:30:10.008007006
NANOTIMES-TAMP	2012.06.13T13:30:10.008007006	np.datetime64	2012-06-13T13:30:10.008007006
FLOAT	2.1f	float	2.0999999046325684
DOUBLE	2.1	float	2.1
SYMBOL	---	---	---
STRING	"Hello"	str	"Hello"
UUID	uuid("5d212a78-cc48-e3b1-4235-b4d91473ee87")	str	"5d212a78-cc48-e3b1-4235-b4d91473ee87"
DATEHOUR	datehour(2012.06.13T13:30:10)	np.datetime64	2012-06-13T13
IPADDR	ipaddr("192.168.1.13")	str	"192.168.1.13"
INT128	int128("e1671797c52e15f763380b45e841ec32")	str	"e1671797c52e15f763380b45e841ec32"
BLOB	blob("xxxyyyzzz")	str	"xxxyyyzzz"
DECIMAL32	decimal32(1.111, 4)	decimal.Decimal	1.1110
DECIMAL64	decimal64(1.123456789, 5)	decimal.Decimal	1.12345

注1： DolphinDB 中的空值不仅有 VOID 类型，还有 INT、STRING 等类型的空值，这些空值 Scalar 都对应 Python API 的空值 None。

注2： DolphinDB 中不存在 Scalar 形式的 SYMBOL 数据。

4.2 **Vector**

DolphinDB 中的 Vector 数据一般对应 Python 中的 numpy.ndarray。特别的，Any Vector 对应 Python 中的 list （1.30.17.1及之前版本的 API 对应 numpy.ndarray）。

下表给出各类型 Vector 所对应 numpy.ndarray 的 dtype：

DolphinDB类型	np.dtype
BOOL（不含空值）	bool
CHAR（不含空值）	int8
SHORT（不含空值）	int16
INT（不含空值）	int32
LONG（不含空值）	int64
DATE	datetime64
MONTH	datetime64
TIME、TIMESTAMP	datetime64
MINUTE	datetime64
SECOND、DATETIME	datetime64
NANOTIME、NANOTIMESTAMP	datetime64
FLOAT	float32
DOUBLE、CHAR（含空值）、SHORT（含空值）、INT（含空值）、LONG（含空值）	float64
DATEHOUR	datetime64
BOOL（含空值）、SYMBOL、STRING、UUID、IPADDR、INT128、BLOB、DECIMAL32、DECIMAL64、Array Vector	object

注1： numpy.ndarray 中没有整型的空值，因此如果 DolphinDB 整型的 Vector 中包含空值，则会强制转换为 float64，空值变为 np.nan。

注2： BOOL Vector 中如果包含空值，则会转换为 `dtype=object` 的 np.ndarray，而非 `dtype=bool`。

注3： DolphinDB Array Vector 下载到 Python 所对应的 dtype 为 object，其中每一项元素都是原始 ArrayVector 的一个元素转换而成的 np.ndarray。

注4： 若使用1.30.17.2及之后版本的 Python API，则 DolphinDB Any Vector 下载到 Python 对应的数据类型为 list；若使用1.30.17.2及之前版本的 Python API，则 DolphinDB Any Vector 下载到 Python 对应的数据类型为 np.ndarray。

注5： 当前版本暂不支持 Decimal32 Array Vector 和 Decimal64 Array Vector。

下载 Vector 型数据的相关代码示例：

```
>>> re = s.run("[true, false]")
>>> re
[ True False]
>>> type(re)
<class 'numpy.ndarray'>
>>> re.dtype
bool

>>> re = s.run("[true, None]")
>>> re
[True None]
>>> re.dtype
object
```

上例中，首先下载一个不含空值的 BOOL Vector，可以看到对应的 Python 对象为 np.ndarray，且 `dtype=bool`；如果下载的 BOOL Vector 中包含空值，则对应的 dtype 变为 object。和 BOOL 不同，如果下载的整型向量中包含空值，则 `dtype=float64`。

```
>>> re = s.run("[1, 2, 3, NULL]")
>>> re
[ 1.  2.  3. nan]
>>> re.dtype
float64
```

如果下载的 Vector 为 Array Vector，则 API 会遍历每个元素，将所有元素逐个按照 Vector 的转换规则进行转换。示例中下载的 INT Array Vector 某一元素中包含空值，因此，其他不含空值的元素对应 dtype=int32，包含空值的元素则对应 dtype=float64。

```
>>> s.run("arrayVector(2 3 4, [1, 2, 3, NULL])")
[array([1, 2], dtype=int32) array([3], dtype=int32) array([nan])]

>>> re = s.run('''(1, 2, [12, "aaa"])''')
>>> re
[1, 2, [12, 'aaa']]
```

```
>>> type(re)
<class 'list'>
>>> type(re[2])
<class 'list'>
```

如果下载的 Vector 为 Any Vector，则 API 会遍历每个元素，按照每个元素的转换规则进行转换。示例中的 Any Vector 嵌套了另一个 Any Vector，两者都被转换为 list 类型。如果 Python API 版本等于或低于 1.30.17.1，则 Any Vector 数据将会被转换为 np.ndarray，且 dtype=object。

4.3 Pair

DolphinDB 中的 Pair 类型对应 Python 中的 list，其中每一个元素都将按照 Scalar 的规则进行转换。

上传 Pair 型数据的相关代码示例：

```
>>> s.run("100:0")
[100, 0]
```

4.4 Matrix

DolphinDB 中的 Matrix 类型对应 Python 中的 np.ndarray。不同数据类型与 np.dtype 对应关系如下表所示：

DolphinDB类型	np.dtype
BOOL（不含空值）	bool
CHAR（不含空值）	int8
SHORT（不含空值）	int16
INT（不含空值）	int32
LONG（不含空值）	int64
DATE	datetime64
MONTH	datetime64
TIME、TIMESTAMP	datetime64
MINUTE	datetime64
SECOND、DATETIME	datetime64

DolphinDB类型	np.dtype
NANOTIME、NANOTIMESTAMP	datetime64
FLOAT	float32
DOUBLE、CHAR（含空值）、SHORT（含空值）、INT（含空值）、LONG（含空值）	float64
DATEHOUR	datetime64
BOOL（含空值）	object

下载 Matrix 型数据的相关代码示例：

```
>>> s.run("""
...     mtx = 1..12$4:3;
...     mtx.rename!(1 2 3 4, `c1`c2`c3);
...     mtx
... """)
[array([[ 1,  5,  9],
       [ 2,  6, 10],
       [ 3,  7, 11],
       [ 4,  8, 12]], dtype=int32), array([1, 2, 3, 4], dtype=int32), array(['c1', 'c2', 'c3'], dtype=object)]
```

与上传数据时不同，虽然 Matrix 直接对应二维 np.ndarray，但是 API 在下载 Matrix 型数据时会包含其行名和列名的信息。如果 Matrix 数据中不包含行名（或列名），则使用 None 代替。

注1： 时间类型的 Matrix 在 PROTOCOL_DDB 协议中的对应规则与 Vector 类似，但在 PROTOCOL_PICKLE 协议中则全部对应 datetime64，请参考xxxxx。仅为标注，之后加跳转链接

4.5 Set

DolphinDB 中的 Set 数据形式对应 Python set 类型，在转换时，API 会将 Set 中的每个元素作为 Scalar 进行转换。有关具体每种数据类型对应的 Python 类型，请参考4.1 Scalar 的转换规则。

注意： 目前仅支持 CHAR、SHORT、INT、LONG、FLOAT、DOUBLE、STRING、SYMBOL Set 转换为 Python 对象。

下载 Set 型数据的相关代码示例：

```
>>> re = s.run("set(1..5)")
>>> re
{1, 2, 3, 4, 5}
```

```
>>> type(re)
<class 'set'>
```

4.6 Dict

DolphinDB 中的 Dict 数据形式对应 Python 中的 dict 类型。在转换时，API 会遍历 Dict 中的所有键值对，并将键作为 Scalar 进行转换；同时根据值本身的数据形式、数据类型进行转换，转换规则请参考本文中对应数据形式的规则。

下载 Dict 型数据的相关代码示例：

```
>>> re = s.run('''{"a": 123, "b": [1.1, 2.2]}''')
>>> re
{'b': array([1.1, 2.2]), 'a': 123}
>>> type(re)
<class 'dict'>
```

4.7 Table

DolphinDB 中的 Table 数据形式对应 Python 中的 pandas.DataFrame 类型。在转换时，API 会将 Table 型数据的每一列作为 Vector 类型进行处理。

特别的，不同于 Vector 的处理，当 Table 型数据为时间类型时，Python pandas 仅支持一种时间类型 datetime64，因此下载的时间类型都会转换为 datetime64。

注意： 目前 API 仅支持 Array Vector 列的下载，不支持 Any Vector 列的下载。

5. 反序列化 DolphinDB -> Python(设置 pickleTableToList=True)

开启附加参数 pickleTableToList 后，如果执行脚本的返回值数据形式为 Table，则对应的 Python 对象为 list 而非 pd.DataFrame。其中，list 中的每一元素（np.ndarray）都表示 Table 中的一列。

PROTOCOL_DDB 协议的附加参数仅在 API 端做处理，不会作为 flag 的一部分发送至 DolphinDB。API 收到数据后，不会再将数据拼接为 pd.DataFrame，而是将每一列放入 list 中。

Table

和 Vector 型数据的转换流程不同，每一列在转换时仍旧视为是 Table 中的一列，而非单独的 Vector，因此时间类型会被转换为 datetime。

注1： 如果下载 Table 型数据的数据列为 Array Vector 列，须确保每个元素的长度一致，其对应数据类型为二维 np.ndarray。

下载 Table 型数据的相关代码示例：

```
>>> re = s.run("table([1, NULL] as a, [2012.01.02, NULL] as b)", pickleTableToList=True)
>>> re
[array([ 1., nan]), array(['2012-01-02T00:00:00.000000000',                          'NaT'],
      dtype='datetime64[ns]')]
>>> type(re)
<class 'list'>
>>> re[0].dtype
float64
>>> re[1].dtype
datetime64[ns]

>>> s.run("table(arrayVector(1 2 3, [1, 2, 3]) as a)", pickleTableToList=True)
[array([[1],
       [2],
       [3]], dtype=int32)]
```

上例中，指定附加参数 pickleTableToList 后下载的 Table 数据，转换为每个元素都是 np.ndarray 的 list。如果下载的 Table 中整型数据列中包含空值，则对应 dtype=float64；如果下载时间类型列，则全部对应 datetime64；如果下载 Array Vector 数据列，则需要其中的每个元素都长度相等。

PROTOCOL_PICKLE

Pickle 协议是一种对 Python 对象进行序列化和反序列化的方式，允许使用者将复杂的 Python 对象转换为可以存储或传输的字节流，再将该字节流转换为原始的 Python 对象。DolphinDB 中提供了基于 Python Pickle 协议特化的反序列化方案 PROTOCOL_PICKLE，该协议仅限在 DolphinDB Python API 中进行使用，其支持的**数据形式**和**数据类型**相对较少。

注1：数据形式指 DolphinDB 类型系统中的 DATAFORM，通常包含 Scalar、Vector、Table 等，表示数据结构的形式。

注2：数据类型指 DolphinDB 类型系统中的 DATATYPE，通常包含 INT、DOUBLE、DATETIME 等，表示数据的具体类型。

注3：以下简称 Python 库 NumPy 为 **np**，pandas 为 **pd**。

1. 启用 PROTOCOL_PICKLE

在以下示例中，Session 和 DBConnectionPool 通过设置参数 *protocol* 指定启用 PROTOCOL_PICKLE 协议。在当前版本中 PROTOCOL_DEFUALT 等同于 PROTOCOL_PICKLE，故默认使用 PROTOCOL_PICKLE 作为序列化、反序列化协议。

```
import dolphindb as ddb
import dolphindb.settings as keys

s = ddb.Session(protocol=keys.PROTOCOL_PICKLE)
```

```
s.connect("localhost", 8848, "admin", "123456")

pool = ddb.DBConnectionPool("localhost", 8848, "admin", "123456", 10, protocol=keys.PROTOCOL_PICKLE)
```

2. PROTOCOL_PICKLE 数据形式支持表

PROTOCOL_PICKLE 支持的数据形式如下表展示：

附加参数	数据形式	序列化	反序列化
pickleTableTo-List=False	Matrix	不支持	支持
pickleTableTo-List=False	Table	不支持	支持
pickleTableTo-List=True	Table	不支持	支持

3. 反序列化 DolphinDB -> Python(设置 pickleTableToList=False)

3.1 **Matrix**

DolphinDB 中的 Matrix 对应 Python 中的 np.ndarray，不同数据类型与 np.dtype 的对应关系如下表所示：

DolphinDB类型	np.dtype
BOOL（不含空值）	bool
CHAR（不含空值）	int8
SHORT（不含空值）	int16
INT（不含空值）	int32
LONG（不含空值）	int64
DATE、MONTH、TIME、TIMESTAMP、MINUTE、SECOND、DATETIME、NANOTIME、NANOTIMESTAMP、DATE-HOUR	datetime64

DolphinDB类型	np.dtype
FLOAT	float32
DOUBLE、CHAR（含空值）、SHORT（含空值）、INT（含空值）、LONG（含空值）	float64
BOOL（含空值）	object

和 PROTOCOL_DDB 一致，API 通过 PROTOCOL_PICKLE 协议下载的 Matrix 对应着包含三个元素的 list。list 中的第一个元素为 np.ndarray，表示实际数据；第二、三个元素分别对应 Matrix 的行名和列名，如果未设置行名、列名，则用 None 替代。如下为示例代码：

```
>>> s.run("date([2012.01.02, 2012.02.03])$1:2")
[array([['2012-01-02T00:00:00.000000000', '2012-02-03T00:00:00.000000000']],
      dtype='datetime64[ns]'), None, None]
```

注意： 若指定协议为 PROTOCOL_DDB，则下载时间类型 Matrix 对应的 dtype 为 datetime64/datetime64/datetime64/...；若指定协议为 PROTOCOL_PICKLE，则下载时间类型 Matrix 对应的 dtype 都为 datetime64。

3.2 Table

PROTOCOL_PICKLE 协议中 Table 列类型对应的 np.dtype 如下表所示：

DolphinDB 类型	np.dtype
BOOL（不含空值）	bool
CHAR（不含空值）	int8
SHORT（不含空值）	int16
INT（不含空值）	int32
LONG（不含空值）	int64
DATE、MONTH、TIME、TIMESTAMP、MINUTE、SECOND、DATETIME、NANOTIME、NANOTIMESTAMP、DATEHOUR	datetime64
FLOAT	float32
DOUBLE、CHAR（含空值）、SHORT（含空值）、INT（含空值）、LONG（含空值）	float64

DolphinDB 类型	np.dtype
BOOL（含空值）、SYMBOL、STRING、UUID、IPADDR、INT128、Array Vector	object

注1： PROTOCOL_PICKLE 暂不支持 BLOB、DECIMAL32、DECIMAL64 类型的数据列。

注2： PROTOCOL_PICKLE 暂不支持 UUID、IPADDR、INT128 类型的 Array Vector 数据列。

下载 Table 型数据的相关代码示例：

```
>>> re = s.run("table([1, NULL] as a, [2012.01.02, 2012.01.05] as b)")
>>> re
     a          b
0  1.0 2012-01-02
1  NaN 2012-01-05
>>> re['a'].dtype
float64
>>> re['b'].dtype
datetime64[ns]
```

4. 反序列化 DolphinDB -> Python(设置 pickleTableToList=True)

Table

指定使用 PROTOCOL_PICKLE 协议，在执行 run 方法时，若指定额外参数 `pickleTableToList=True`，则下载 Table 型数据将得到一个 list 数据，且 list 的每个元素都是 np.ndarray。如果下载 Table 型数据的数据列为 Array Vector 列，须确保每个元素的长度一致，其对应数据类型为二维 np.ndarray。

本节详细的类型转换规则和 PROTOCOL_DDB 中章节仅为标注，之后加跳转链接一致。 开启附加参数 pickleTableToList 后，如果执行脚本的返回值数据形式为 Table，则对应的 Python 对象为 list 而非 pd.DataFrame。其中，list 中的每一元素（np.ndarray）都表示 Table 中的一列。

和 PROTOCOL_DDB 协议的附加参数稍有不同，PROTOCOL_DDB 协议的附加参数会作为flag的一部分发送至服务端。

PROTOCOL_ARROW

Apache Arrow 协议是一种用于对大型数据集进行序列化和反序列化的协议，可以跨平台、跨语言地进行高效数据传输。DolphinDB 提供的 formatArrow 插件在 Apache Arrow 协议的基础上进行类型适配，实现 DolphinDB 和 API 之间通过 Apache Arrow 协议进行数据传输。

1. PROTOCOL_ARROW 数据形式支持表

对于 API 而言，PROTOCOL_ARROW 协议目前仅支持 Table 型数据的反序列化，且不支持开启压缩模式。

附加参数	数据形式	序列化	反序列化
无	Table	不支持	支持

2. 启用 PROTOCOL_ARROW

目前 API 使用 Apache Arrow 4.0 (待确认具体版本)，如果使用 PROTOCOL_ARROW，须在 Python 环境中安装 9.0.0 以上版本的 pyarrow。

在以下示例中，Session 和 DBConnectionPool 通过设置参数 *protocol* 指定启用 PROTOCOL_ARROW 协议。

```
import dolphindb as ddb
import dolphindb.settings as keys

s = ddb.Session(protocol=keys.PROTOCOL_ARROW)
s.connect("localhost", 8848, "admin", "123456")

pool = ddb.DBConnectionPool("localhost", 8848, "admin", "123456", 10, protocol=keys.PROTOCOL_ARROW)
```

3. 反序列化 DolphinDB -> Python

Table

使用 PROTOCOL_ARROW 协议时，DolphinDB 中的 Table 对应 Python 中的 pyarrow.Table。详细类型转换的对照信息如下表所示：

DolphinDB类型	Arrow类型
BOOL	boolean
CHAR	int8
SHORT	int16
INT	int32
LONG	int64
DATE	date32
MONTH	date32
TIME	time32(ms)

DolphinDB类型	Arrow类型
MINUTE	time32(s)
SECOND	time32(s)
DATETIME	timestamp(s)
TIMESTAMP	timestamp(ms)
NANOTIME	time64(ns)
NANOTIMES-TAMP	timestamp(ns)
DATEHOUR	timestamp(s)
FLOAT	float32
DOUBLE	float64
SYMBOL	dictionary(int32, utf8)
STRING	utf8
IPADDR	utf8
UUID	fixed_size_binary(16)
INT128	fixed_size_binary(16)
BLOB	large_binary
DECIMAL32(X)	decimal128(38, X)
DECIMAL64(X)	decimal128(38, X)

注1： PROTOCOL_ARROW 协议同时支持以上除了 DECIMAL32/DECIMAL64 外的 Array Vector 类型。

注2： 使用 PROTOCOL_ARROW 协议获取 pyarrow.Table 数据后，如果需要将数据转换为 pandas.DataFrame，由于 DolphinDB NANOTIME 数据类型对应 Arrow 的 time64(ns) 类型，因此要求进行转换的小数数值必须为0.001的倍数，否则会提示 `Value xxxxxxx has non-zero nanoseconds`。

代码示例：

```
>>> s.run("table(1..3 as a)")
pyarrow.Table
a: int32
----
a: [[1,2,3]]
```

强制类型转换

在使用 upload 接口上传 pandas.DataFrame 时，由于 DolphinDB 类型系统与 Python 类型系统不是一一对应的关系，所以无法直接上传部分类型的数据，例如 UUID、IPADDR、SECOND 等类型。

自1.30.22.1版本起，Python API 支持强制类型转换。在使用强制类型转换时，需要在待上传的 pandas.DataFrame 上增加属性 __DolphinDB_Type__，该属性是一个 Python 字典对象，键为列名，值为指定的类型。示例如下：

```
import dolphindb as ddb
import pandas as pd
import numpy as np

s = ddb.Session()
s.connect("localhost", 8848, "admin", "123456")
df = pd.DataFrame({
    'cint': [1, 2, 3],
    'csymbol': ["aaa", "bbb", "aaa"],
    'cblob': ["a1", "a2", "a3"],
})

s.upload({"df_wrong": df})
print(s.run("schema(df_wrong)")['colDefs'])
```

输出如下：

```
      name typeString  typeInt  extra comment
0     cint       LONG        5    NaN
1  csymbol     STRING       18    NaN
2    cblob     STRING       18    NaN
```

参考章节 3.1.1 可知，如果直接上传 `df`，此时 `cint` 列的 dtype 为 int64，仍会作为 LONG 类型上传；而由于 SYMBOL、BLOB 没有对应的类型，故直接上传的 str 型数据会被视作 STRING 类型。

导入 dolphindb.settings，为待上传的 pandas.DataFrame 添加属性，其字典键为需要指定类型的列名。

```
import dolphindb.settings as keys

df.__DolphinDB_Type__ = {
    'cint': keys.DT_INT,
    'csymbol': keys.DT_SYMBOL,
    'cblob': keys.DT_BLOB,
}

s.upload({"df_true": df})
print(s.run("schema(df_true)")['colDefs'])
```

输出如下：

```
      name typeString  typeInt  extra comment
0     cint        INT        4    NaN
1  csymbol     SYMBOL       17    NaN
2    cblob       BLOB       32    NaN
```

再次上传后，由输出结果可知 pandas.DataFrame 的各列都被正确转换为指定的类型。

类型转换

DolphinDB 与 Python API 的交互过程始终遵循 API交互协议。该协议规定了通信双方在交互过程中使用的报文信息格式。在 API 与 DolphinDB 的交互过程中，通过指定 *protocol* 参数，即可选择交互过程中使用的传输数据格式协议。

Python 中包含多套常用类型系统，其与 DolphinDB 的类型系统并非一一对应。为更好地兼容各类型系统与 DolphinDB 之间的数据交互，DolphinDB Python API 自1.30.21.1版本起，Session（session）和 DBConnectionPool 新增 *protocol* 参数。目前已支持的协议包括 PROTOCOL_DDB、PROTOCOL_PICKLE 和 PROTOCOL_ARROW，默认使用 PROTOCOL_PICKLE。此外，对于 UUID、IPADDR、SECOND 等无法直接上传的类型， Python API 支持强制类型转换。

下例展示了 Session 如何指定使用的传输协议。

```
import dolphindb as ddb
import dolphindb.settings as keys

# 使用协议 PROTOCOL_DDB
s = ddb.Session(protocol=keys.PROTOCOL_DDB)

# 使用协议 PROTOCOL_PICKLE
s = ddb.Session(protocol=keys.PROTOCOL_PICKLE)

# 使用协议 PROTOCOL_ARROW
s = ddb.Session(protocol=keys.PROTOCOL_ARROW)
```

PROTOCOL_DDB、PROTOCOL_PICKLE 和 PROTOCOL_ARROW 协议支持不同的类型系统，提供各自特色的序列化方式以适应不同应用场景，最终实现更高效的数据传输。以下内容为用户选择传输协议提供些许参考建议：

- PROTOCOL_DDB 协议是 DolphinDB 自定义的一套数据序列化、反序列化方案，被广泛使用于 Python API、C++ API、Java API 等 API 中。其支持的数据形式和数据类型最为全面。
- PROTOCOL_PICKLE 协议基于 Python 原生的 Pickle 协议，同时进行了部分适配 DolphinDB 的修改，是 *protocol* 参数默认指定的协议。
- PROTOCOL_ARROW 协议是基于 Apache Arrow 通用序列化方案、用于对大型数据集进行序列化和反序列化的协议，可以跨平台、跨语言地进行高效数据传输。

从使用场景的角度来看：

- PROTOCOL_ARROW 协议适用于从上游数据库到下游消费、且全部使用 Apache Arrow 格式作为中间格式的场景，可以方便地在各个组件之间传递数据。而用户只需要付出从数据库取出数据这一次序列化开销，后续不再需要进行序列化和反序列化。此外，使用该协议可以较高效地利用网络带宽，目前有较多的行情服务商使用 Apache Arrow 协议。
- PROTOCOL_PICKLE 协议针对数据传输类的场景，PROTOCOL_DDB 协议支持的数据形式和数据类型最为全面。如果场景中使用 pandas 的 DataFrame 较多，则更推荐使用 PROTOCOL_PICKLE 协议和 PROTOCOL_DDB 协议。由于一般的数据类型用 PROTOCOL_PICKLE 协议的数据传输速度相对更快，因此 *protocol* 参数默认开启 PROTOCOL_PICKLE 协议。

从数据类型的角度来看：

- PROTOCOL_PICKLE 协议和 PROTOCOL_DDB 协议在某些类型的性能上各有优劣。`？这里需要继续展开讲一下吗?`

交互流程

API 与 DolphinDB 的交互流程可以简化为 Session 建立阶段和单次 run 请求阶段。

Session 建立阶段

API 先向 DolphinDB 发起连接请求，发送的报文中包含一组 flag。该 flag 中包含协议参数对应的标志位，表示建立 Session 时 API 选用的序列化、反序列化协议。目前版本 Python API 支持的可选协议有 PROTOCOL_DDB、PROTOCOL_PICKLE、PROTOCOL_ARROW。

协议参数的生效周期和 Session 保持一致。在 Session 连接期间，如果上传、查询涉及到的数据形式属于该协议支持的数据形式，则会使用该协议对应的序列化、反序列化方案进行类型转换。如果不属于该协议支持的数据形式，则默认使用 PROTOCOL_DDB 的方式进行处理。举例来说，Python API 中的协议 PROTOCOL_PICKLE 仅支持 Matrix 和 Table 数据形式的反序列化。在下载一个 Vector 数据时，即便指定了使用 PROTOCOL_PICKLE 协议，但由于 PROTOCOL_PICKLE 不支持 Vector 数据形式，因此仍会使用默认的 PROTOCOL_DDB 协议的反序列化流程。

单次 run 请求阶段

与建立 Session 类似，单次执行 run 时发送的请求同样附带一组 flag。该 flag 中包含协议参数和附加参数，例如协议参数 *PROTOCOL_PICKLE* 和 *PROTOCOL_DDB* 都具有附加参数 *pickleTableToList*。如果在执行 run 方法时指定 `pickleTableToList=True`，则将改变当次请求的序列化、反序列化流程。

但与建立 Session 不同的是，单次 run 请求的 flag 参数生效周期和一次查询一致。

多种写入方案

Python API 提供多种写入方案，可以适配不同场景的写入需求，下面将详细介绍各种写入方案之间的区别。

1. upload & table & tableInsert

1.1 **upload**

最直接的上传方式，就是调用 Session.upload 将数据直接上传，这种方案适合上传各种数据形式，例如 TABLE、DICTIONARY、VECTOR 等，调用 upload 时，需要指定数据上传后的变量名。在 upload 上传数据时，因为 DolphinDB 的数据类型和 Python 的原生数据类型、numpy 以及 pandas 的数据类型无法一一对应，因此会出现某些数据类型无法直接上传的情况，例如 UUID、MINUTE 等数据类型。

1.30.22.1及之后版本的 Python API，新增强制类型转换，可以在调用 upload 上传 pd.DataFrame 时添加 __DolphinDB_Type__，指定待上传列的类型。

代码示例：

```
>>> data = pd.DataFrame({
...     'clong': [1, 2, 3],
... })
...
>>> s.upload({'data': data})
```

1.2 **table**

Session.table 方法可以传入一个本地数据对象，例如 pandas.DataFrame/dict/...，将该数据对象作为一个临时表上传到 DolphinDB 服务端，其生存周期由 Python API 进行维护。该种方法仅支持上传 TABLE 数据形式的对象。内部实现调用了 Session.upload，因此也可以指定 __DolphinDB_Type__ 实现强制类型转换。

和 upload 方法稍有不同，upload 方法上传的变量需要手动析构上传变量的生存周期，处理不当可能会导致 Session 占用内存过大。table 方法返回一个 Python API 定义的 Table 类对象，在析构时会同时析构 DolphinDB 服务端的该临时表，不需要手动维护生存周期。

代码示例如下：

```
>>> data = pd.DataFrame({
...     'clong': [1, 2, 3],
... })
...
>>> tb = s.table(data=data)
>>> tb
<dolphindb.table.Table object at 0x7faf42e67a00>
```

1.3 tableInsert

和前两种方法不同，tableInsert 并非 Python API 提供的方法，而是通过 run 方法上传参数的功能实现的写入方式。从数据序列化的角度，该方法和 upload、table 没有区别。从使用的角度，在某些不需要指定上传变量名的流程中，作为 run 方法的参数上传，更为简单、直接。例如写入表时，无需先上传临时表到服务端，再调用 tableInsert 写入，直接作为 tableInsert 的参数上传并写入，简化流程。

同时，如果代码写入部分涉及到访问权限问题、写入时有较长步骤，可以将这些内容封装为 functionview，将需要上传的内容作为 functionview 的参数上传至服务端。

代码示例如下：

```
>>> data = pd.DataFrame({
...     'clong': [1, 2, 3],
... })
...
>>> s.run("t = table(100:0, [`clong], [LONG])")
>>> s.run("tableInsert{t}", data)
3
```

参考链接：

- DolphinDB 用户手册-部分应用
- DolphinDB 用户手册-函数视图

1.4 upload & table & tableInsert 对比

这三种方式本质上都是同一种写入流程，即先判断待上传变量的数据形式、类型，类型转换后作为函数/变量的参数上传至服务端。其中 table 方法实现上封装了 upload 方法，tableInsert 方法不仅仅适用于 tableInsert 一种服务器函数，同样适用于任何需要传入参数的服务器函数、函数视图。

2. TableAppender & TableUpserter & PartitionedTableAppender

2.1 TableAppender

内部实现等价于 run("tableInsert{tableName}", data)，和直接调用不同，TableAppender 在构造时获得待写入表的列类型，在类型转换时，能够根据列类型进行自动转换。

代码示例：

```
>>> s.run("t = table(100:0, `csymbol`cvalue, [SYMBOL, LONG])")
>>> tbAppender = ddb.tableAppender(tableName="t", ddbSession=s)
>>> data = pd.DataFrame({
...     'csymbol': ["aaa", "bbb", "aaa"],
...     'cvalue': [1, 2, 3],
... })
...
>>> tbAppender.append(data)
3
```

2.2 TableUpserter

与 TableAppender 一致，TableUpserter 同样会在构造时获取待更新表的列类型，在类型转换时，能够根据列类型进行自动转换。内部实现等价于 upsert! 方法，构造 TableUpserter 时需指定键值列。

代码示例：

```
>>> s.run("t = keyedTable(`csymbol, 100:0, `csymbol`cvalue, [SYMBOL, LONG])")
>>> tbUpserter = ddb.TableUpserter(tableName="t", ddbSession=s, keyColNames=["csymbol"])
>>> data = pd.DataFrame({
...     'csymbol': ["aaa", "bbb", "aaa"],
...     'cvalue': [1, 2, 3],
... })
...
>>> tbUpserter.upsert(data)
```

参考链接：

- DolphinDB 用户手册-upsert!

2.3 PartitionedTableAppender

TableAppender 和 TableUpserter 都基于 Session 进行数据写入，PartitionedTableAppender 构造时需要传入 DBConnectionPool 对象，写入时可以将数据并发地写入分区表中。同样的，PartitionedTableAppender 也支持写入数据的自动类型转换。

```
>>> if s.existsDatabase("dfs://test"):
...     s.dropDatabase("dfs://test")
...
>>> db = s.database(dbPath="dfs://test", partitionType=keys.VALUE, partitions=[1, 2, 3])
>>> s.run("schema_table = table(100:0, `cindex`cvalue, [INT, DOUBLE]);")
>>> schema_table = s.table(data="schema_table")
>>> tb = db.createPartitionedTable(table=schema_table, tableName="pt", partitionColumns="cindex")
>>> pool = ddb.DBConnectionPool("localhost", 8848, 3, "admin", "123456")
>>> ptableAppender = ddb.PartitionedTableAppender(dbPath="dfs://test", tableName="pt", partitionColName="cindex", dbConnectionPool=pool)
>>> data = pd.DataFrame({
...     'cindex': [1, 2, 3, 4, 5],
...     'cvalue': [1.1, 2.2, 3.3, 4.4, 5.5]
... })
...
>>> ptableAppender.append(data)
5
```

2.4 TableAppender & TableUpserter & PartitionedTableAppender 对比

这三种方式都能够将 pandas.DataFrame 形式的数据自动类型转换后写入到指定表中，但是三者的适用场景有一定区别。TableAppender 适用于所有表的写入；TableUpserter 适用于键值表、索引表、分区表的更新写入；PartitionedTableAppender 则适用于分区表等支持同时写入的表。

3. MTW & BTW & Async tableInsert

3.1 MultithreadedTableWriter

MTW 在后台启用多个 C++ 线程，异步地进行类型转换和上传写入。对于每个表，都需要构造一个对应的 MTW 对象进行写入。在写入时，前台调用 insert 后，并不立刻将数据进行转换，而是先将数据放入待转换队列，等待转换线程将数据转换完毕后放入写入队列。最后由多个写入队列向服务端写入数据。

代码示例：

```
>>> if s.existsDatabase("dfs://test"):
...     s.dropDatabase("dfs://test")
>>> db = s.database(dbPath="dfs://test", partitionType=keys.VALUE, partitions=[1, 2, 3])
>>> s.run("schema_table = table(100:0, `cindex`cvalue, [INT, DOUBLE])")
>>> schema_table = s.table(data="schema_table")
>>> pt = db.createPartitionedTable(table=schema_table, tableName="pt", partitionColumns="cindex")
>>> writer = ddb.MultithreadedTableWriter("localhost", 8848, "admin", "123456", dbPath="dfs://test", tableName="pt", threadCount=1)
>>> for i in range(100):
...     writer.insert(i, i*1.1)
```

```
>>> writer.waitForThreadCompletion()
>>> res = writer.getStatus()
>>> if res.succeed():
...     print("Data successfully written.")
...
Data successfully written.
```

3.2 BatchTableWriter

BTW 仅为每张表创建一个写入线程，不同于 MTW，BTW 在 insert 时进行类型转换，总体性能较差。

注意： 目前已经停止维护 BTW。

3.4 Async tableInsert

和 tableInsert 方法类似，Async tableInsert 并非 API 提供的方法，而是在异步模式 Session 中调用 run 方法，将待上传数据作为参数上传的一种方式。参考章节 2.5.1，该方法的工作原理是 Session 的异步模式执行脚本时，仅需将脚本发送至服务端，方法立刻返回，而无需等待脚本执行完毕再返回。

代码示例：

```
>>> s = ddb.Session(enableASYNC=True)
>>> s.connect("localhost", 8848, "admin", "123456")
>>> s.run("t = table(100:0, `cindex`cvalue, [INT, DOUBLE]);")
>>> data = pd.DataFrame({
...     'cindex': [1, 2, 3, 4, 5],
...     'cvalue': [1.1, 2.2, 3.3, 4.4, 5.5]
... })
...
>>> for i in range(100):
...     s.run("tableInsert{t}", data)
...
```

3.5 MTW & BTW & Async tableInsert 对比

这三种写入方式都是异步写入，工作原理上稍有不同。Async tableInsert 本质上利用了 Session 的异步模式，适用于网络带宽资源紧张的情况，可以有效降低网络占用和等待时间，其缺点是将写入压力转移至服务端，可能会造成服务端资源占用过多，如果单次写入仅为一条数据，非批量数据，则会造成服务器资源的浪费。MTW 和 BTW 都采用后台 C++ 写入线程的处理方式，BTW 每隔 100 ms 将待写入数据发送至服务端，MTW 则提供 batchSize、throttle 参数用于指定批数据处理的粒度和等待时间，适用于更多场景。受制于 Python 本身的全局解释器锁，MTW 在类型转换时难以利用多线程提速，但在写入阶段，后台多个 C++ 工作线程可以有效进行数据分流和批量上传，降低网络状况带来的影响。从类型转换角度，MTW 和 BTW 将每条待写入数据中的每个元素逐个进行类型转

换，BTW 无法根据待写入表的列类型进行自动类型转换， MTW 则可以根据列类型进行自动转换。当前版本，BTW 已经停止维护，MTW 基本可以替代所有 BTW 的写入场景。

4. 总结

通常而言，如果需要上传一个变量，则使用 upload 上传最为简单、直接，类型转换更加自由。如果需要执行服务端函数时附带参数，则调用 run 方法，并将 Python 对象作为参数直接传入。如果需要面向对象地在 Python 端操作服务端数据库、数据表，可以使用 table 相关方法，能够较为便捷地在服务端和 Python 端进行数据交互。

如果待写入数据为批量数据，数据结构可以较为便捷地转换为 pandas.DataFrame，则可以使用 TableAppender、TableUpserter、PartitionedTableAppender 来写入数据。该场景适用于文件内容上传等场景。针对不同的写入表，可以选择使用不同的类进行写入。

如果待写入数据为流式数据，例如股市数据等，可以使用 MTW 来进行流式写入。如果API端资源较为紧张，服务器资源较为充裕，也可以考虑使用异步模式 Session 的写入方式，将写入压力转移至服务端，通常不推荐该方式写入。

流订阅模式

在 Python API 中，共推荐使用四种流订阅模式：单条订阅、批量订阅（设置 `msgAsTable=False`）、批量订阅（设置 `msgAsTable=True`）和异构流表订阅。下面将通过四个示例来分别介绍如何使用这四种订阅模式，以及各种订阅之间的区别。有关流订阅相关参数的介绍，请参考章节2.4。仅为标注，之后添加跳转链接

1. 单条订阅

使用单条订阅模式，不需要指定 *batchSize*，此时 *msgAsTable* 应为 False，*throttle* 参数无效。

下例中，首先通过 Session.run 执行脚本来构造流表，然后调用 Session.enableStreaming 方法启用流订阅，再定义回调函数 handler。开始订阅后，调用 Session.run 执行写入脚本，API 立刻收到消息并将结果打印出来。等待3秒后，调用 unsubscribe 取消订阅。

```
import dolphindb as ddb
import numpy as np
import time

s = ddb.Session()
s.connect("192.168.1.113", 8848, "admin", "123456")

s.run("""
share streamTable(10000:0,`time`sym`price`id, [TIMESTAMP,SYMBOL,DOUBLE,INT]) as trades
""")
```

```
s.enableStreaming()

def handler(lst):
    print(lst)

s.subscribe("192.168.1.113", 8848, handler, "trades", "SingleMode", offset=-1)

s.run("insert into trades values(take(now(), 6), take(`000905`600001`300201`000908`600002, 6), rand(1000,6)/10.0, 1..6)")

time.sleep(3)

s.unsubscribe("192.168.1.113", 8848, "trades", "SingleMode")
```

输出结果如下所示：

```
[numpy.datetime64('2023-03-17T12:06:30.439'), '000905', 36.7, 1]
[numpy.datetime64('2023-03-17T12:06:30.439'), '600001', 80.7, 2]
[numpy.datetime64('2023-03-17T12:06:30.439'), '300201', 68.7, 3]
[numpy.datetime64('2023-03-17T12:06:30.439'), '000908', 52.2, 4]
[numpy.datetime64('2023-03-17T12:06:30.439'), '600002', 45.1, 5]
[numpy.datetime64('2023-03-17T12:06:30.439'), '000905', 55.1, 6]
```

在流订阅中，API 内部仅使用 PROTOCOL_DDB 协议进行反序列化。在单条订阅模式下，DolphinDB 发送的数据由 API 接收后，每一行数据将从 AnyVector 转换为 list。有关 AnyVector 转换的详细说明，请参考章节3.1.1。仅为标注，此处需要补充跳转链接

2. 批量订阅（设置 `msgAsTable=False`）

若要使用批量订阅模式，则须指定参数 *batchSize* 和 *throttle*，表示当接收到的消息条数超过 *batchSize*，或者处理消息前的等待时间超过 *throttle*，则会触发一次回调，将数据传递给 *handler*，并且按批次来处理数据。当指定 `msgAsTable=False` 时，收到的一批数据将是一个列表 list，其中每一项都是单条数据，结构和单条订阅模式中的一条数据一致。

在下例中，分别指定 `batchSize=2`，`throttle=0.1`，表示在0.1秒时间内，如果收到了2条数据，则立刻调用回调函数传入这2条数据；如果等待的0.1秒仅收到1条数据，则会在等待结束后调用回调函数传入这1条数据。与单条订阅模式相似，批量订阅模式下通过 PROTOCOL_DDB 协议进行数据类型转换，每条数据将从 AnyVector 转为 list。

```
import dolphindb as ddb
import numpy as np
import time

s = ddb.Session()
s.connect("192.168.1.113", 8848, "admin", "123456")
```

```
s.run("""
share streamTable(10000:0,`time`sym`price`id, [TIMESTAMP,SYMBOL,DOUBLE,INT]) as trades
""")

s.enableStreaming()

def handler(lsts):
    print(lsts)

s.subscribe("192.168.1.113", 8848, handler, "trades", "MultiMode1", offset=-1, batchSize=2, throttle=0.1, msgAsTable=False)

s.run("insert into trades values(take(now(), 6), take(`000905`600001`300201`000908`600002, 6), rand(1000,6)/10.0, 1..6)")

time.sleep(3)

s.unsubscribe("192.168.1.113", 8848, "trades", "MultiMode1")
```

输出结果如下：

```
[[numpy.datetime64('2023-03-17T14:46:27.358'), '000905', 21.2, 1], [numpy.datetime64('2023-03-17T14:46:27.358'), '600001', 39.8, 2]]
[[numpy.datetime64('2023-03-17T14:46:27.358'), '300201', 84.0, 3], [numpy.datetime64('2023-03-17T14:46:27.358'), '000908', 26.2, 4]]
[[numpy.datetime64('2023-03-17T14:46:27.358'), '600002', 25.1, 5], [numpy.datetime64('2023-03-17T14:46:27.358'), '000905', 42.7, 6]]
```

3. 批量订阅（设置 `msgAsTable=True`）

开启批量订阅时，如果指定 `msgAsTable=True`，则每一批数据将基于消息块（由 DolphinDB 中的参数 *maxMsgNumPerBlock* 进行配置）处理消息。当收到的记录总数大于等于 *batchSize* 时，*handler* 会对所有达到条件的消息块进行处理。

下例中，在开启批量订阅模式后，调用 `Session.run` 执行脚本，向流表中写入1500条数据，此时 DolphinDB 中的参数 *maxMsgNumPerBlock* 为默认值 1024，因此 API 接收到1024条数据后，消息条数恰好超过 `batchSize=1000`，立刻调用回调函数；随后收到剩下的476条数据，等待0.1秒仍无新数据，再次调用回调函数。因此最后的输出结果为两个长度分别为1024和476的 DataFrame。

```
import dolphindb as ddb
import numpy as np
import time

s = ddb.Session()
s.connect("192.168.1.113", 8848, "admin", "123456")

s.run("""
```

```
share streamTable(10000:0,`time`sym`price`id, [TIMESTAMP,SYMBOL,DOUBLE,INT]) as trades
""")

s.enableStreaming()

def handler(lsts):
    print(lsts)

s.subscribe("192.168.1.113", 8848, handler, "trades", "MultiMode2", offset=-1, batchSize=1000, throttle=0.1, msgAsTable=True)

s.run("n=1500;insert into trades values(take(now(), n), take(`000905`600001`300201`000908`600002, n), rand(1000,n)/10.0, 1..n)")

time.sleep(3)

s.unsubscribe("192.168.1.113", 8848, "trades", "MultiMode2")
```

如果修改上述示例中的 *batchSize* 为1500，发送的数据为3000条，服务端发送第一个消息块（长度为1024）后，不触发回调函数；服务端发送第二个消息块（长度为1024）后，API 收到的数据条数共为2048，超过 `batchSize=1500`，立刻触发回调函数，通过 PROTOCOL_DDB 协议将收到的消息从 Table 转换为 pandas.DataFrame；服务端发送第三个消息块（长度为952）后，经过0.1秒，仍没有接收到新数据，此时触发回调函数。在这种情况下，回调函数中收到的数据，长度分别为2048和952。

4. 异构流表订阅

DolphinDB 自 1.30.17 及 2.00.5 版本开始，支持通过 replay 函数将多个结构不同的流数据表回放（序列化）到一个流表里，这个流表被称为异构流表。Python API 自 1.30.19 版本开始新增 streamDeserializer 类，用于构造异构流表反序列化器，以实现对异构流表的订阅和反序列化操作。

4.1 异构流表反序列化器

Python API 通过 streamDeserializer 类来构造异构流表反序列化器，接口定义如下：

```
streamDeserializer(sym2table, session=None)
```

- *sym2table*：字典对象，其结构与 replay 回放到异构流表的输入表结构保持一致。streamDeserializer 将根据 *sym2table* 指定的结构对注入的数据进行反序列化。
- *session*：已连接 DolphinDB 的 Session 对象，默认为 None。如果不指定，将会在订阅时自动获取当前连接。

下例构造一个简单的异构流表反序列化器：

```
sd = ddb.streamDeserializer({
    'msg1': ["dfs://test_StreamDeserializer_pair", "pt1"],
```

```
    'msg2': "pt2",
}, session=s)
```

其中，sym2table 的键为不同输入表的标记，用于区分不同输入表的数据；sym2table 的值为表名，或由分区数据库地址和表名组成的列表（或元组）。订阅时，会通过构造时传入的 Session 调用 schema 方法获得 sym2table 键值对应的表的结构，因此并不一定需要填输入表名，只需要和输入表结构一致即可。

关于构造 DolphinDB 异构流表的具体脚本，请参照异构回放示例。

注意：

1. 在 DolphinDB 中构造异构流表时，字典中 key 对应的表应为内存表或 replayDS 定义的数据源，请参考 replay。
2. API 端构造异构流表反序列化器时，*sym2table* 的值对应的表（可以为分区表、流表或者内存表）结构需要和 DolphinDB 中构造异构流表使用的表结构一致。
3. 订阅异构流表时，*msgAsTable* 不能为 True，可以指定 *batchSize* 和 *throttle*。

4.2 订阅示例 1 （分区表数据源作为输入表）

下例中，首先在 DolphinDB 中定义由两个分区表组合而成的异构流表，然后在 Python 客户端定义异构流表反序列化器 sd，再根据 sd 中指定表的结构反序列化数据。在输出结果中，每条数据的末尾都增加了一个字段，用于标识当前数据的 symbol。

构造异构流表

首先在 DolphinDB 中定义输出表，即要订阅的异构流表。

```
try{dropStreamTable(`outTables)}catch(ex){}
share streamTable(100:0, `timestampv`sym`blob`price1,[TIMESTAMP,SYMBOL,BLOB,DOUBLE]) as outTables
```

然后定义两张输入表，均为分布式分区表。

```
n = 6;
dbName = 'dfs://test_StreamDeserializer_pair'
if(existsDatabase(dbName)){
    dropDB(dbName)}
db = database(dbName,RANGE,2012.01.01 2013.01.01 2014.01.01 2015.01.01 2016.01.01 2017.01.01 2018.01.01 2019.01.01)
table1 = table(100:0, `datetimev`timestampv`sym`price1`price2, [DATETIME, TIMESTAMP, SYMBOL, DOUBLE, DOUBLE])
table2 = table(100:0, `datetimev`timestampv`sym`price1, [DATETIME, TIMESTAMP, SYMBOL, DOUBLE])
tableInsert(table1, 2012.01.01T01:21:23 + 1..n, 2018.12.01T01:21:23.000 + 1..n, take(`a`b`c,n), rand(100,n)+rand(1.0, n), rand(100,n)+rand(1.0, n))
tableInsert(table2, 2012.01.01T01:21:23 + 1..n, 2018.12.01T01:21:23.000 + 1..n, take(`a`b`c,n), rand(100,n)+rand(1.0, n))
pt1 = db.createPartitionedTable(table1,'pt1',`datetimev).append!(table1)
pt2 = db.createPartitionedTable(table2,'pt2',`datetimev).append!(table2)
```

将分区表转为数据源后进行回放。

```
re1 = replayDS(sqlObj=<select * from pt1>, dateColumn=`datetimev, timeColumn=`timestampv)
re2 = replayDS(sqlObj=<select * from pt2>, dateColumn=`datetimev, timeColumn=`timestampv)
d = dict(['msg1', 'msg2'], [re1, re2])
replay(inputTables=d, outputTables=`outTables, dateColumn=`timestampv, timeColumn=`timestampv)
```

订阅异构流表

```
import dolphindb as ddb

# 异构流表反序列化器返回的数据末尾为异构流表反序列化器中 sym2table 指定的 key
def streamDeserializer_handler(lst):
    if lst[-1]=="msg1":
        print("Msg1: ", lst)
    elif lst[-1]=='msg2':
        print("Msg2: ", lst)
    else:
        print("Error: ", lst)

s = ddb.Session()
s.connect("192.168.1.113", 8848, "admin", "123456")
s.enableStreaming()

# 填入分区表数据库路径和表名的 list，以获取对应表结构
sd = ddb.streamDeserializer({
    'msg1': ["dfs://test_StreamDeserializer_pair", "pt1"],
    'msg2': ["dfs://test_StreamDeserializer_pair", "pt2"],
}, session=s)
s.subscribe(host="192.168.1.113", port=8848, handler=streamDeserializer_handler, tableName="outTables", actionName="action", offset=0,
 resub=False,
            msgAsTable=False, streamDeserializer=sd, userName="admin", password="123456")

from threading import Event
Event().wait()
```

输出结果如下所示：

```
Msg2:  [numpy.datetime64('2012-01-01T01:21:24'), numpy.datetime64('2018-12-01T01:21:23.001'), 'a', 18.43745171907358, 'msg2']
Msg1:  [numpy.datetime64('2012-01-01T01:21:24'), numpy.datetime64('2018-12-01T01:21:23.001'), 'a', 65.69160503265448, 41.17562178615481, 'msg1']
Msg2:  [numpy.datetime64('2012-01-01T01:21:25'), numpy.datetime64('2018-12-01T01:21:23.002'), 'b', 93.68146854126826, 'msg2']
Msg1:  [numpy.datetime64('2012-01-01T01:21:25'), numpy.datetime64('2018-12-01T01:21:23.002'), 'b', 22.181119214976206, 38.162505637388676, 'msg1']
Msg2:  [numpy.datetime64('2012-01-01T01:21:26'), numpy.datetime64('2018-12-01T01:21:23.003'), 'c', 51.19852650281973, 'msg2']
Msg1:  [numpy.datetime64('2012-01-01T01:21:26'), numpy.datetime64('2018-12-01T01:21:23.003'), 'c', 16.937458558939397, 36.79589221812785, 'msg1']
Msg2:  [numpy.datetime64('2012-01-01T01:21:27'), numpy.datetime64('2018-12-01T01:21:23.004'), 'a', 0.812068443512544, 'msg2']
```

```
Msg1:  [numpy.datetime64('2012-01-01T01:21:27'), numpy.datetime64('2018-12-01T01:21:23.004'), 'a', 34.11729482654482, 29.094212289899588, 'msg1']
Msg2:  [numpy.datetime64('2012-01-01T01:21:28'), numpy.datetime64('2018-12-01T01:21:23.005'), 'b', 93.43341179518029, 'msg2']
Msg1:  [numpy.datetime64('2012-01-01T01:21:28'), numpy.datetime64('2018-12-01T01:21:23.005'), 'b', 9.413380537647754, 32.449754945002496, 'msg1']
Msg2:  [numpy.datetime64('2012-01-01T01:21:29'), numpy.datetime64('2018-12-01T01:21:23.006'), 'c', 65.18307867064141, 'msg2']
Msg1:  [numpy.datetime64('2012-01-01T01:21:29'), numpy.datetime64('2018-12-01T01:21:23.006'), 'c', 83.58133838768117, 54.27990723075345, 'msg1']
```

4.3 订阅示例 2 （内存表作为输入表）

下例中，在 DolphinDB 中定义了一个由两个内存表构成的异构流表，并在 Python 端使用共享内存表的表名构造反序列化器，最后指定 `batchSize=4` 进行批量订阅。可以看出，在总数据条数为6*2=12的情况下，数据首先按总条数分3批传入回调函数，在每批数据中，每条数据可能来自不同的输入表。因此，共调用回调函数3次，每次输出4条数据构成的一批数据。

构造异构流表

```
try{dropStreamTable(`outTables)}catch(ex){}
// 构造输出流表
share streamTable(100:0, `timestampv`sym`blob`price1,[TIMESTAMP,SYMBOL,BLOB,DOUBLE]) as outTables

n = 6;
table1 = table(100:0, `datetimev`timestampv`sym`price1`price2, [DATETIME, TIMESTAMP, SYMBOL, DOUBLE, DOUBLE])
table2 = table(100:0, `datetimev`timestampv`sym`price1, [DATETIME, TIMESTAMP, SYMBOL, DOUBLE])
tableInsert(table1, 2012.01.01T01:21:23 + 1..n, 2018.12.01T01:21:23.000 + 1..n, take(`a`b`c,n), rand(100,n)+rand(1.0, n), rand(100,n)+rand(1.0, n))
tableInsert(table2, 2012.01.01T01:21:23 + 1..n, 2018.12.01T01:21:23.000 + 1..n, take(`a`b`c,n), rand(100,n)+rand(1.0, n))
share table1 as pt1
share table2 as pt2

d = dict(['msg1', 'msg2'], [pt1, pt2])
replay(inputTables=d, outputTables=`outTables, dateColumn=`timestampv, timeColumn=`timestampv)
```

订阅异构流表

```
import dolphindb as ddb

def streamDeserializer_handler(lsts):
    print(lsts)

s = ddb.Session()
s.connect("192.168.1.113", 8848, "admin", "123456")
s.enableStreaming()

sd = ddb.streamDeserializer({
    'msg1': "pt1",
    'msg2': "pt2",
}, session=s)
```

```
s.subscribe(host="192.168.1.113", port=8848, handler=streamDeserializer_handler, tableName="outTables", actionName="action", offset=0,
 resub=False, batchSize=4,
            msgAsTable=False, streamDeserializer=sd, userName="admin", password="123456")

from threading import Event
Event().wait()
```

输出结果如下所示：

```
[[numpy.datetime64('2012-01-01T01:21:24'), numpy.datetime64('2018-12-01T01:21:23.001'), 'a', 87.90784921264276, 'msg2'], [numpy.datetime64('2012-01-01T01:21:24'),
 numpy.datetime64('2018-12-01T01:21:23.001'), 'a', 14.867915444076061, 92.22166634746827, 'msg1'], [numpy.datetime64('2012-01-01T01:21:25'),
 numpy.datetime64('2018-12-01T01:21:23.002'), 'b', 80.60459423460998, 'msg2'], [numpy.datetime64('2012-01-01T01:21:25'), numpy.datetime64('2018-12-01T01:21:23.002'), 'b',
 10.429520844481885, 29.480175042990595, 'msg1']]
[[numpy.datetime64('2012-01-01T01:21:26'), numpy.datetime64('2018-12-01T01:21:23.003'), 'c', 12.45058359648101, 'msg2'], [numpy.datetime64('2012-01-01T01:21:26'),
 numpy.datetime64('2018-12-01T01:21:23.003'), 'c', 55.05597074679099, 88.84371786634438, 'msg1'], [numpy.datetime64('2012-01-01T01:21:27'),
 numpy.datetime64('2018-12-01T01:21:23.004'), 'a', 27.357952459948137, 'msg2'], [numpy.datetime64('2012-01-01T01:21:27'), numpy.datetime64('2018-12-01T01:21:23.004'), 'a',
 57.705578718334436, 25.98224212951027, 'msg1']]
[[numpy.datetime64('2012-01-01T01:21:28'), numpy.datetime64('2018-12-01T01:21:23.005'), 'b', 63.73548944480717, 'msg2'], [numpy.datetime64('2012-01-01T01:21:28'),
 numpy.datetime64('2018-12-01T01:21:23.005'), 'b', 65.34572763741016, 0.6374575316440314, 'msg1'], [numpy.datetime64('2012-01-01T01:21:29'),
 numpy.datetime64('2018-12-01T01:21:23.006'), 'c', 89.62549424753524, 'msg2'], [numpy.datetime64('2012-01-01T01:21:29'), numpy.datetime64('2018-12-01T01:21:23.006'), 'c',
 98.75018240674399, 46.55078419903293, 'msg1']]
```

3.4.1 Table

在 Python API 中，可以使用 DolphinDB Python API 的原生方法来创建、使用数据库及数据表，本节将介绍如何创建数据表、使用 SQL 操作数据表。

Table & Session.table

Python API 将 DolphinDB 服务端的数据表对象句柄，在 API 包装为 Table 类，封装实现部分功能。通常使用 Session.table 或 Session.loadTable方法构造，也可以通过 loadText 等函数获得。

接口如下：

```
Session.table(dbPath=None, data=None, tableAliasName=None, inMem=False, partitions=None)
```

- *dbPath*：数据库路径，内存表或流表无需指定该参数。
- *data*：数据表的数据，可以为 dict、pd.DataFrame 或 DolphinDB 服务端数据表名。
- *tableAliasName*：表的别名。
- *inMem*：是否加载表数据到 DolphinDB 服务端内存中。
- *partitions*：将被加载到 DolphinDB 服务端内存中的分区。

上传数据为临时表

如果 data 参数传入 dict 或 pd.DataFrame，则表示上传该对象到 DolphinDB 服务端为临时表，此时无需指定 dbPath、inMem、partitions。

代码示例如下：

```
data1 = pd.DataFrame({
    'a': [1, 2, 3],
    'b': [4, 5, 6],
})
t1 = s.table(data=data1)
print(t1, t1.tableName())
data2 = {
    'a': ['a', 'b', 'c'],
    'b': [1, 2, 3],
}
t2 = s.table(data=data2)
print(t2, t2.tableName())
```

输出结果如下：

```
<dolphindb.table.Table object at 0x7fbd5f02bd60> TMP_TBL_3cc57246
<dolphindb.table.Table object at 0x7fbd3205fc70> TMP_TBL_dbae4978
```

data1 和 data2 都作为临时表上传至服务端，对应表名分别为 TMP_TBL_3cc57246 和 TMP_TBL_dbae4978。

获取服务端数据表句柄

data 参数传入字符串，则表示获取服务端数据表句柄。

1. 如果同时指定 dbPath 和 data，执行函数则表示从 dbPath 对应的数据库中加载表名为 data 的数据表。

```
dbPath = "dfs://testTable"
if s.existsDatabase(dbPath):
    s.dropDatabase(dbPath)
db = s.database(partitionType=keys.VALUE, partitions=[1, 2, 3], dbPath=dbPath, engine="TSDB")
s.run("schema_t = table(100:0, `ctime`csymbol`price`qty, [TIMESTAMP, SYMBOL, DOUBLE, INT])")
schema_t = s.table(data="schema_t")
db.createTable(schema_t, "pt", ["csymbol"])
pt = s.table(dbPath=dbPath, data="pt")
print(pt, pt.tableName())
print(pt.toDF())
```

输出结果如下：

```
<dolphindb.table.Table object at 0x7f5036bcd040> pt
Empty DataFrame
Columns: [ctime, csymbol, price, qty]
Index: []
```

1. 如果仅指定 data，执行函数则表示获取名为 data 的内存表句柄。

```
s.run("test_t = table(100:0, `ctime`csymbol`price`qty, [TIMESTAMP, SYMBOL, DOUBLE, INT])")
t = s.table(data="test_t")
print(t, t.tableName())
print(t.toDF())
```

输出结果如下：

```
<dolphindb.table.Table object at 0x7f11ffb3c070> test_t
Empty DataFrame
Columns: [ctime, csymbol, price, qty]
Index: []
```

tableAliasName

如果指定该参数，则加载表时不会使用随机表民作为句柄名称。

1. 上传本地变量至服务端时指定该参数。

```
data1 = pd.DataFrame({
    'a': [1, 2, 3],
    'b': [4, 5, 6],
})
t1 = s.table(data=data1, tableAliasName="data1")
print(t1, t1.tableName())
data2 = {
    'a': ['a', 'b', 'c'],
    'b': [1, 2, 3],
}
t2 = s.table(data=data2, tableAliasName="data2")
print(t2, t2.tableName())
```

输出结果如下：

```
<dolphindb.table.Table object at 0x7f167ecb69d0> data1
<dolphindb.table.Table object at 0x7f1651d0bc40> data2
```

1. 获取 DolphinDB 服务端数据库中数据表时指定该参数。

```
dbPath = "dfs://testTable"
if s.existsDatabase(dbPath):
    s.dropDatabase(dbPath)
db = s.database(partitionType=keys.VALUE, partitions=[1, 2, 3], dbPath=dbPath, engine="TSDB")
s.run("schema_t = table(100:0, `ctime`csymbol`price`qty, [TIMESTAMP, SYMBOL, DOUBLE, INT])")
schema_t = s.table(data="schema_t")
db.createTable(schema_t, "pt", ["csymbol"])
pt = s.table(dbPath=dbPath, data="pt", tableAliasName="tmp_pt")
print(pt, pt.tableName())
print(pt.toDF())
```

输出结果如下：

```
<dolphindb.table.Table object at 0x7f3350edc040> tmp_pt
Empty DataFrame
Columns: [ctime, csymbol, price, qty]
Index: []
```

1. 获取 DolphinDB 服务端内存表句柄时指定该参数。

```
s.run("test_t = table(100:0, `ctime`csymbol`price`qty, [TIMESTAMP, SYMBOL, DOUBLE, INT])")
t = s.table(data="test_t", tableAliasName="test_t2")
print(t, t.tableName())
print(t.toDF())
```

输出结果如下：

```
<dolphindb.table.Table object at 0x7f9fb55b4070> test_t
Empty DataFrame
Columns: [ctime, csymbol, price, qty]
Index: []
```

从上述例子中可以看出，如果加载分区表或上传本地数据时，可以通过指定别名来避免使用临时表名；如果直接使用表名加载数据表句柄，则该参数无效。

inMem & partitions

该参数仅在加载磁盘数据库中的数据表时有效，参数详细使用方式请参考 DolphinDB 用户手册-loadTable。

Session.loadTable

该函数和 Session.table 方法类似，返回值为 Table，但无法上传本地数据，仅能获取 DolphinDB 服务端数据表句柄。

```
Session.loadTable(tableName, dbPath=None, partitions=None, memoryMode=False)
```

- *tableName*：内存表名或数据库中数据表表名。
- *dbPath*：数据库路径。
- *partitions*：将被加载到 DolphinDB 服务端内存中的分区。
- *memoryMode*：是否加载表数据到 DolphinDB 服务端内存中。

加载内存表

代码示例如下：

```
s.run("test_t = table(100:0, `ctime`csymbol`price`qty, [TIMESTAMP, SYMBOL, DOUBLE, INT])")
t = s.loadTable("test_t")
print(t, t.tableName())
print(t.toDF())
```

输出结果如下：

```
<dolphindb.table.Table object at 0x7fd1c90a4c10> test_t
Empty DataFrame
Columns: [ctime, csymbol, price, qty]
Index: []
```

加载数据库表

代码示例如下：

```
dbPath = "dfs://testTable"
if s.existsDatabase(dbPath):
    s.dropDatabase(dbPath)
db = s.database(partitionType=keys.VALUE, partitions=[1, 2, 3], dbPath=dbPath, engine="TSDB")
s.run("schema_t = table(100:0, `ctime`csymbol`price`qty, [TIMESTAMP, SYMBOL, DOUBLE, INT])")
schema_t = s.table(data="schema_t")
db.createTable(schema_t, "pt", ["csymbol"])
pt = s.loadTable("pt", dbPath=dbPath)
print(pt, pt.tableName())
print(pt.toDF())
```

输出结果如下：

```
<dolphindb.table.Table object at 0x7fdaf7885eb0> pt_TMP_TBL_0dfdc80a
Empty DataFrame
Columns: [ctime, csymbol, price, qty]
Index: []
```

上传的数据表的生命周期

`table` 和 `loadTable` 方法返回一个 Python 本地变量，如果上传一个本地数据对象到服务端，且未指定别名，则使用随机表名作为该变量的句柄名。下例中，上传本地 data 对象到服务端，对应的 Python 本地变量为 t。服务端表名可以通过 Table.tableName 方法获取。

```
data = pd.DataFrame({
    'a': [1, 2, 3],
    'b': [4, 5, 6],
})
t = s.table(data=data)
print(t.tableName())
```

数据结果如下：

```
TMP_TBL_e03723c9
```

其中 TMP_TBL_ 开头，表示该句柄为临时表的句柄，会随着 Python 端 Table 对象的析构而析构。

此时释放 DolphinDB 服务端对象有三种方法：

1. `undef` 方法

```
s.undef(t.tableName(), "VAR")
```

1. 将服务端对象置空

```
s.run(f"{t.tableName()}=NULL")
```

1. 析构本地变量以取消本地对象对服务端对象的引用。

```
del t
```

注意： 如果在获取句柄时指定别名，或者获取服务端已存在的非临时表句柄，则 Python 端对象的析构并不会影响服务端数据。

如果一个表对象只是一次性使用，尽量不要使用上传机制，可以直接通过函数调用来完成，将表对象作为函数的一个参数。函数调用不会缓存数据。函数调用结束后，所有数据都释放，而且只有一次网络传输，降低网络延迟。

表操作

rows & cols & colNames & schema

调用 rows/cols 属性可以获取当前表的行数和列数。

```
>>> s.run("t = table(1..5 as a, 2..6 as b)    ")
>>> t = s.table(data="t")
>>> t.rows
5
>>> t.cols
2
```

调用 colNames 属性则可以获取当前表的列名。

```
>>> t.colNames
['a', 'b']
```

调用 schema 属性返回一个 pd.DataFrame，表示当前表的结构（返回结果与服务端函数 schema 的结果中的 colDefs 属性一致）。

```
>>> t.schema
  name typeString  typeInt  extra comment
0    a        INT        4    NaN
1    b        INT        4    NaN
```

rename

调用该函数可以给表重新设置名称。

```
>>> t.tableName()
t
>>> t.rename("xx")
>>> t.tableName()
xx
```

注意： 如果给临时表重新设置名称，会导致临时表无法被及时析构，导致内存泄漏。

drop

```
>>> trade = s.loadText(WORK_DIR+"/example.csv")
>>> trade.colNames
['TICKER', 'date', 'VOL', 'PRC', 'BID', 'ASK']
>>> t1 = trade.drop(['ask', 'bid'])
```

```
>>> t1.colNames
['TICKER', 'date', 'VOL', 'PRC']
```

select

该方法类似 SQL 语句中的 select 子句，用于选取部分列。下例中使用 toDF 方法获取表对应的 pd.DataFrame 对象，详细使用请参考 toDF 方法。

1. 使用一系列的列名作为输入内容。

```
>>> trade=s.loadText(WORK_DIR+"/example.csv")
>>> trade.select(["ticker", "date"]).toDF()
      ticker       date
0       AMZN 1997-05-15
1       AMZN 1997-05-16
2       AMZN 1997-05-19
3       AMZN 1997-05-20
4       AMZN 1997-05-21
...
```

1. 使用字符串作为输入内容。

```
>>> trade.select("ticker, date, bid").toDF()
      ticker       date        bid
0       AMZN 1997-05-15   23.50000
1       AMZN 1997-05-16   20.50000
2       AMZN 1997-05-19   20.50000
3       AMZN 1997-05-20   19.62500
4       AMZN 1997-05-21   17.12500
...
```

showSQL

可以使用 `showSQL` 来展示 SQL 语句：

```
>>> trade=s.loadText(WORK_DIR+"/example.csv")
>>> trade.select(["ticker", "date"]).showSQL()
select ticker,date from TMP_TBL_fb11c541
```

toDF & toList

toDF 和 toList 功能类似，都是执行缓存在 Python 端的 SQL 语句（可通过 showSQL 方法获取），并将执行结果返回。两者区别在于，toDF 与 Session.run(sql) 一致，toList 则与 Session.run(sql, pickleTableToList=True) 行为一致。Session构造时，如果指定 protocol=PROTOCOL_PICKLE/PROTOCOL_DDB，则 toDF 返回一个 pd.DataFrame，toList 返回一个 np.ndarray 构成的 list，每个 np.ndarray 表示表中的一列。详细区别请参考章节 3.1.1 和 3.1.2。

exec

select 子句总是生成一张表，即使只选择一列亦是如此。若需要生成一个标量或者一个向量，可使用 exec 子句。

`exec` 只选择一列时生成一个 dolphinDB 的向量。在 Python 中，使用 toDF() 加载该对象，可以打印出一个 `np.ndarray` 对象：

```
>>> trade = s.loadText(WORK_DIR+"/example.csv")
>>> trade.exec("ticker").toDF()
['AMZN' 'AMZN' 'AMZN' ... 'NFLX' 'NFLX' 'NFLX']
```

如果 `exec` 语句选择了多列，那么结果和 `select` 语句一致，生成一个 dolphinDB 的 table 类型。在 Python 中，使用 toDF() 加载该对象，可以打印出一个 `pd.DataFrame` 对象：

```
>>> trade.exec(["ticker", "date", "bid"]).toDF()
      ticker       date        bid
0       AMZN 1997-05-15   23.50000
1       AMZN 1997-05-16   20.50000
2       AMZN 1997-05-19   20.50000
3       AMZN 1997-05-20   19.62500
4       AMZN 1997-05-21   17.12500
...
```

where

`where` 子句用于过滤数据。

1. 多个条件过滤

```
>>> trade = s.loadText(WORK_DIR+"/example.csv")
>>> t1 = trade.select(['date','bid','ask','prc','vol']).where('TICKER=`AMZN').where('bid!=NULL').where('ask!=NULL').where('vol>10000000')
>>> t1.toDF()
         date        bid        ask        prc       vol
0  1998-09-01   79.93750   80.25000   79.95313  11321844
1  1998-11-17  148.68750  149.00000  148.50000  10279448
2  1998-11-20  179.62500  179.75000  180.62500  11314228
```

```
3   1998-11-23  217.75000  217.81250  218.00000  11559042
4   1998-11-24  214.25000  214.62500  214.50000  13820992
...
>>> t1.rows
765
>>> t1.showSQL()
select date,bid,ask,prc,vol from TMP_TBL_03a070d8 where TICKER=`AMZN and bid!=NULL and ask!=NULL and vol>10000000
```

1. 使用字符串作为输入内容

`select` 的输入内容可以是包含多个列名的字符串，`where` 的输入内容也可以是包含多个条件的字符串。

```
>>> t1 = trade.select("ticker, date, vol").where("bid!=NULL, ask!=NULL, vol>50000000")
>>> t1.toDF()
  ticker       date        vol
0    AMZN 1999-09-29   80380734
1    AMZN 2000-06-23   52221978
2    AMZN 2001-11-26   51543686
3    AMZN 2002-01-22   57235489
4    AMZN 2005-02-03   60580703
...
>>> t1.rows
41
```

execute （暂时存疑，该方法需要修改）

执行 SQL 查询。如果执行 update、delete、groupby、contextby、pivotby等操作，必须执行 execute 才能确保在服务端正确执行，返回结果为 Table。

executeAs

`executeAs` 可以把结果保存为 server 端的表对象，表名由参数 *newTableName* 指定，执行后返回一个 `Table` 对象管理新创建的表。

注意： 执行该方法创建的表，生存周期不受 Python 端控制，与 Session 生存周期一致。

```
>>> trade = s.loadText(WORK_DIR+"/example.csv")
>>> t1 =
 trade.select(['date','bid','ask','prc','vol']).where('TICKER=`AMZN').where('bid!=NULL').where('ask!=NULL').where('vol>10000000').executeAs("
AMZN")
>>> t1.tableName()
AMZN
```

update

`update` 更新表后，必须和 `execute` 一起使用才能将 Python 端修改同步至服务端。

```
>>> trade = s.loadText(WORK_DIR+"/example.csv")
>>> t1 = trade.update(["VOL"],["999999"]).where("TICKER=`AMZN").where(["date=2015.12.16"]).execute()
>>> t2 = t1.where("ticker=`AMZN").where("VOL=999999")
>>> t2.toDF()
  TICKER       date     VOL        PRC        BID        ASK
0   AMZN 2015-12-16  999999  675.77002  675.76001  675.83002
```

delete

`delete` 必须与 `execute` 一同使用来删除表中的记录。

```
>>> trade = s.loadText(WORK_DIR+"/example.csv")
>>> trade.rows
13136
>>> t = trade.delete().where('date<2013.01.01').execute()
>>> trade.rows
3024
```

groupby

`groupby` 后面需要使用聚合函数，如 `count`, `sum`, `agg` 或 `agg2` 等。

准备数据库

```
>>> dbPath = "dfs://valuedb"
>>> if s.existsDatabase(dbPath):
...     s.dropDatabase(dbPath)
>>> s.database(dbName='mydb', partitionType=keys.VALUE, partitions=["AMZN","NFLX","NVDA"], dbPath=dbPath)
>>> trade = s.loadTextEx(dbPath=dbPath, partitionColumns=["TICKER"], tableName='trade', remoteFilePath=WORK_DIR+"/example.csv")
```

分别计算每个股票的 vol 总和与 prc 总和：

```
>>> trade.select(['sum(vol)','sum(prc)']).groupby(['ticker']).toDF()
  ticker       sum_vol       sum_prc
0   AMZN  33706396492  772503.81377
1   NFLX  14928048887  421568.81674
2   NVDA  46879603806  127139.51092
```

`groupby` 与 `having` 一起使用：

```
>>> trade.select('count(ask)').groupby(['vol']).having('count(ask)>1').toDF()
       vol  count_ask
0   579392          2
1  3683504          2
2  5732076          2
3  6299736          2
4  6438038          2
5  6946976          2
6  8160197          2
7  8924303          2
```

contextby

`contextby` 与 `groupby` 相似，区别在于 `groupby` 为每个组返回一个标量，但是 `contextby` 为每个组返回一个向量，向量的长度与该组的行数相同。

```
>>> trade.contextby('ticker').top(3).toDF()
  TICKER       date      VOL      PRC      BID      ASK
0   AMZN 1997-05-15  6029815  23.5000  23.5000  23.6250
1   AMZN 1997-05-16  1232226  20.7500  20.5000  21.0000
2   AMZN 1997-05-19   512070  20.5000  20.5000  20.6250
3   NFLX 2002-05-23  7507079  16.7500  16.7500  16.8500
4   NFLX 2002-05-24   797783  16.9400  16.9400  16.9500
5   NFLX 2002-05-28   474866  16.2000  16.2000  16.3700
6   NVDA 1999-01-22  5702636  19.6875  19.6250  19.6875
7   NVDA 1999-01-25  1074571  21.7500  21.7500  21.8750
8   NVDA 1999-01-26   719199  20.0625  20.0625  20.1250
>>> trade.select("TICKER, month(date) as month, cumsum(VOL)").contextby("TICKER,month(date)").toDF()
      TICKER      month  cumsum_VOL
0       AMZN 1997-05-01     6029815
1       AMZN 1997-05-01     7262041
2       AMZN 1997-05-01     7774111
3       AMZN 1997-05-01     8230468
4       AMZN 1997-05-01     9807882
...
```

`contextby` 与 `having` 一起使用：

```
>>> trade.contextby('ticker').having("sum(VOL)>40000000000").toDF()
     TICKER       date       VOL       PRC       BID       ASK
0      NVDA 1999-01-22   5702636   19.6875   19.6250   19.6875
```

```
1      NVDA 1999-01-25   1074571   21.7500   21.7500   21.8750
2      NVDA 1999-01-26    719199   20.0625   20.0625   20.1250
3      NVDA 1999-01-27    510637   20.0000   19.8750   20.0000
4      NVDA 1999-01-28    476094   19.9375   19.8750   20.0000
...
```

pivotby

`pivotby` 是 DolphinDB 的独有功能，是对标准 SQL 语句的拓展。它将表中一列或多列的内容按照两个维度重新排列，亦可配合数据转换函数使用。详细使用请参考 DolphinDB 用户手册-pivotby。

`pivotby` 与 `select` 子句一起使用时返回一个表。

```
>>> trade = s.table("dfs://valuedb", "trade")
>>> t1 = trade.select("VOL").pivotby("TICKER", "date")
>>> t1.toDF()
  TICKER  1997.05.15  1997.05.16  ...  2016.12.28  2016.12.29  2016.12.30
0   AMZN   6029815.0   1232226.0  ...     3301025     3158299     4139451
1   NFLX         NaN         NaN  ...     4388956     3444729     4455012
2   NVDA         NaN         NaN  ...    57384116    54384676    30323259
```

`pivotby` 和 `exec` 语句一起使用时返回一个 DolphinDB 的矩阵对象。

```
>>> trade.exec("VOL").pivotby("TICKER", "date").toDF()
[array([[ 6029815.,  1232226.,   512070., ...,  3301025.,  3158299.,
         4139451.],
       [      nan,       nan,       nan, ...,  4388956.,  3444729.,
         4455012.],
       [      nan,       nan,       nan, ..., 57384116., 54384676.,
        30323259.]]), array(['AMZN', 'NFLX', 'NVDA'], dtype=object), array(['1997-05-15T00:00:00.000000000', '1997-05-16T00:00:00.000000000',
       '1997-05-19T00:00:00.000000000', ...,
       '2016-12-28T00:00:00.000000000', '2016-12-29T00:00:00.000000000',
       '2016-12-30T00:00:00.000000000'], dtype='datetime64[ns]')]
```

sort & csort

可使用 `csort` 关键字排序。

```
>>> trade = s.loadTable("trade", "dfs://valuedb")
>>> trade.contextby('ticker').csort('date desc').toDF()
      TICKER       date      VOL        PRC        BID        ASK
```

```
0       AMZN 2016-12-30  4139451  749.87000  750.02002  750.40002
1       AMZN 2016-12-29  3158299  765.15002  764.66998  765.15997
2       AMZN 2016-12-28  3301025  772.13000  771.92999  772.15997
3       AMZN 2016-12-27  2638725  771.40002  771.40002  771.76001
4       AMZN 2016-12-23  1981616  760.59003  760.33002  760.59003
...
```

除了在排序函数 `sort` 和 `csort` 中指定 asc 和 desc 关键字来决定排序顺序外，也可以通过传参的方式实现。

```
sort(by, ascending=True)
csort(by, ascending=True)
```

参数 ascending 表示是否进行升序排序，默认值为 True。可以通过传入一个 list 来定义多列的不同排序方式。如以下脚本：

```
>>> trade.select("*").contextby('ticker').csort(["TICKER", "VOL"], True).limit(5).toDF()
   TICKER       date    VOL      PRC      BID     ASK
0    AMZN 1997-12-26  40721  54.2500  53.8750  54.625
1    AMZN 1997-08-12  47939  26.3750  26.3750  26.750
2    AMZN 1997-07-21  48325  26.1875  26.1250  26.250
3    AMZN 1997-08-13  49690  26.3750  26.0000  26.625
4    AMZN 1997-06-02  49764  18.1250  18.1250  18.375
5    NFLX 2002-09-05  20725  12.8500  12.8500  12.950
6    NFLX 2002-11-11  26824   8.4100   8.3000   8.400
7    NFLX 2002-09-04  27319  13.0000  12.8200  13.000
8    NFLX 2002-06-10  35421  16.1910  16.1900  16.300
9    NFLX 2002-09-06  54951  12.8000  12.7900  12.800
10   NVDA 1999-05-10  41250  17.5000  17.5000  17.750
11   NVDA 1999-05-07  52310  17.5000  17.3750  17.625
12   NVDA 1999-05-14  59807  18.0000  17.7500  18.000
13   NVDA 1999-04-01  63997  20.5000  20.1875  20.500
14   NVDA 1999-04-19  65940  19.0000  19.0000  19.125
>>> trade.select("*").contextby('ticker').csort(["TICKER", "VOL"], [True, False]).limit(5).toDF()
   TICKER       date        VOL       PRC     BID       ASK
0    AMZN 2007-04-25  104463043   56.8100   56.80   56.8100
1    AMZN 1999-09-29   80380734   80.7500   80.75   80.8125
2    AMZN 2006-07-26   76996899   26.2600   26.17   26.1800
3    AMZN 2007-04-26   62451660   62.7810   62.77   62.8300
4    AMZN 2005-02-03   60580703   35.7500   35.74   35.7300
5    NFLX 2015-07-16   63461015  115.8100  115.85  115.8600
6    NFLX 2015-08-24   59952448   96.8800   96.85   96.8800
7    NFLX 2016-04-19   55728765   94.3400   94.30   94.3100
```

```
8    NFLX 2016-07-19   55685209   85.8400   85.81   85.8300
9    NFLX 2016-01-20   53009419  107.7400  107.73  107.7800
10   NVDA 2011-01-06   87693472   19.3300   19.33   19.3400
11   NVDA 2011-02-17   87117555   25.6800   25.68   25.7000
12   NVDA 2011-01-12   86197484   23.3525   23.34   23.3600
13   NVDA 2011-08-12   80488616   12.8800   12.86   12.8700
14   NVDA 2003-05-09   77604776   21.3700   21.39   21.3700
```

top & limit

`top` 用于取表中的前 n 条记录。

```
>>> trade = s.table("dfs://valuedb", "trade")
>>> trade.top(5).toDF()
  TICKER       date      VOL     PRC     BID     ASK
0   AMZN 1997-05-15  6029815  23.500  23.500  23.625
1   AMZN 1997-05-16  1232226  20.750  20.500  21.000
2   AMZN 1997-05-19   512070  20.500  20.500  20.625
3   AMZN 1997-05-20   456357  19.625  19.625  19.750
4   AMZN 1997-05-21  1577414  17.125  17.125  17.250
```

`limit` 子句和 `top` 子句功能类似。两者的区别在于：

- top 子句中的整型常量不能为负数。在与 context by 子句一同使用时，limit 子句标量值可以为负整数，返回每个组最后指定数目的记录。其他情况 limit 子句标量值为非负整数。
- 可使用 limit 子句从某行开始选择一定数量的行。

详细使用请参考 DolphinDB 用户手册-limit。

```
>>> trade.select("*").contextby('ticker').limit(-2).toDF()
  TICKER       date       VOL        PRC        BID        ASK
0   AMZN 2016-12-29   3158299  765.15002  764.66998  765.15997
1   AMZN 2016-12-30   4139451  749.87000  750.02002  750.40002
2   NFLX 2016-12-29   3444729  125.33000  125.31000  125.33000
3   NFLX 2016-12-30   4455012  123.80000  123.80000  123.83000
4   NVDA 2016-12-29  54384676  111.43000  111.26000  111.42000
5   NVDA 2016-12-30  30323259  106.74000  106.73000  106.75000
```

```
>>> trade.select("*").limit([2, 5]).toDF()
  TICKER       date      VOL     PRC     BID     ASK
0   AMZN 1997-05-19   512070  20.500  20.500  20.625
```

```
1   AMZN 1997-05-20   456357  19.625  19.625  19.750
2   AMZN 1997-05-21  1577414  17.125  17.125  17.250
3   AMZN 1997-05-22   983855  16.750  16.625  16.750
4   AMZN 1997-05-23  1330026  18.000  18.000  18.125
```

merge & merge_asof & merge_window & merge_cross

`merge` 用于内部连接 (ej)、左连接 (lj)、左半连接 (lsj) 和外部连接 (fj)，`merge_asof` 为 asof join，`merge_window` 为窗口连接，`merge_cross` 为交叉连接。

`merge`

如果连接列名称相同，使用 *on* 参数指定连接列，如果连接列名称不同，使用 *left_on* 和 *right_on* 参数指定连接列。可选参数 *how* 表示表连接的类型。默认的连接类型为内部连接(ej)。

```
>>> trade = s.table("dfs://valuedb", "trade")
>>> t1 = s.table(data={
...     'TICKER': ['AMZN', 'AMZN', 'AMZN'],
...     'date': np.array(['2015-12-31', '2015-12-30', '2015-12-29'], dtype='datetime64[D]'),
...     'open': [695, 685, 674],
... })
...
>>> t1 = t1.select("TICKER, date(date) as date, open")
>>> trade.merge(t1,on=["TICKER","date"]).toDF()
  TICKER       date      VOL        PRC        BID        ASK  open
0   AMZN 2015-12-29  5734996  693.96997  693.96997  694.20001   674
1   AMZN 2015-12-30  3519303  689.07001  689.07001  689.09998   685
2   AMZN 2015-12-31  3749860  675.89001  675.85999  675.94000   695
```

当连接列名称不相同时，需要指定 *left_on* 参数和 *right_on* 参数。

```
>>> trade = s.table("dfs://valuedb", "trade")
>>> t1 = s.table(data={
...     'TICKER': ['AMZN', 'AMZN', 'AMZN'],
...     'date': np.array(['2015-12-31', '2015-12-30', '2015-12-29'], dtype='datetime64[D]'),
...     'open': [695, 685, 674],
... })
...
>>> t1 = t1.select("TICKER as TICKER1, date(date) as date1, open")
>>> trade.merge(t1, left_on=["TICKER","date"], right_on=["TICKER1", "date1"]).toDF()
  TICKER       date      VOL        PRC        BID        ASK  open
0   AMZN 2015-12-29  5734996  693.96997  693.96997  694.20001   674
```

```
1   AMZN 2015-12-30  3519303  689.07001  689.07001  689.09998   685
2   AMZN 2015-12-31  3749860  675.89001  675.85999  675.94000   695
```

左连接时，把 *how* 参数设置为'left'。

```
>>> trade = s.table("dfs://valuedb", "trade")
>>> t1 = s.table(data={
...     'TICKER': ['AMZN', 'AMZN', 'AMZN'],
...     'date': np.array(['2015-12-31', '2015-12-30', '2015-12-29'], dtype='datetime64[D]'),
...     'open': [695, 685, 674],
... })
...
>>> t1 = t1.select("TICKER, date(date) as date, open")
>>> trade.merge(t1,how="left", on=["TICKER","date"]).where('TICKER=`AMZN').where('2015.12.23<=date<=2015.12.31').toDF()
  TICKER       date      VOL        PRC        BID        ASK   open
0   AMZN 2015-12-23  2722922  663.70001  663.48999  663.71002    NaN
1   AMZN 2015-12-24  1092980  662.78998  662.56000  662.79999    NaN
2   AMZN 2015-12-28  3783555  675.20001  675.00000  675.21002    NaN
3   AMZN 2015-12-29  5734996  693.96997  693.96997  694.20001  674.0
4   AMZN 2015-12-30  3519303  689.07001  689.07001  689.09998  685.0
5   AMZN 2015-12-31  3749860  675.89001  675.85999  675.94000  695.0
```

外部连接时，把 *how* 参数设置为'outer'。分区表只能与分区表进行外部链接，内存表只能与内存表进行外部链接。

```
>>> t1 = s.table(data={'TICKER': ['AMZN', 'AMZN', 'NFLX'], 'date': ['2015.12.29', '2015.12.30', '2015.12.31'], 'open': [674, 685, 942]})
>>> t2 = s.table(data={'TICKER': ['AMZN', 'NFLX', 'NFLX'], 'date': ['2015.12.29', '2015.12.30', '2015.12.31'], 'close': [690, 936, 951]})
>>> t1.merge(t2, how="outer", on=["TICKER","date"]).toDF()
  TICKER        date   open tmp_TICKER    tmp_date  close
0   AMZN  2015.12.29  674.0       AMZN  2015.12.29  690.0
1   AMZN  2015.12.30  685.0                           NaN
2   NFLX  2015.12.31  942.0       NFLX  2015.12.31  951.0
3                       NaN       NFLX  2015.12.30  936.0
```

`merge_asof`

`merge_asof` 对应 DolphinDB 中的 asof join (aj)。asof join 与 left join 非常相似，主要有以下区别：

1. asof join 的最后一个连接列通常是时间类型。对于左表中某行的时间 t，在右表最后一个连接列之外的其它连接列一致的记录中，如果右表没有与 t 对应的时间，asof join 会取右表中 t 之前的最近时间对应的记录；如果有多个相同的时间，会取最后一个时间对应的记录。
2. 如果只有一个连接列，右表必须按照连接列排好序。如果有多个连接列，右表必须在其它连接列决定的每个组内根据最后一个连接列排好序。如果右表不满足这些条件，计算结果将会不符合预期。右表不需要按照其他连接列排序，左表不需要排序。

本节与下节的例子使用了 trades.csv 和 quotes.csv，它们含有 NYSE 网站下载的 AAPL 和 FB 的 2016 年 10 月 24 日的交易与报价数据。

```
>>> dbPath = "dfs://tickDB"
>>> if s.existsDatabase(dbPath):
...     s.dropDatabase(dbPath)
...
>>> s.database(partitionType=keys.VALUE, partitions=["AAPL","FB"], dbPath=dbPath)
>>> trades = s.loadTextEx(dbPath, tableName='trades', partitionColumns=["Symbol"], remoteFilePath=WORK_DIR+"/trades.csv")
>>> quotes = s.loadTextEx(dbPath, tableName='quotes', partitionColumns=["Symbol"], remoteFilePath=WORK_DIR+"/quotes.csv")
>>> trades.top(5).toDF()
                         Time  Exchange  Symbol  Trade_Volume  Trade_Price
0 1970-01-01 08:00:00.022239        75    AAPL           300        27.00
1 1970-01-01 08:00:00.022287        75    AAPL           500        27.25
2 1970-01-01 08:00:00.022317        75    AAPL           335        27.26
3 1970-01-01 08:00:00.022341        75    AAPL           100        27.27
4 1970-01-01 08:00:00.022368        75    AAPL            31        27.40
>>> quotes.where("second(Time)>=09:29:59").top(5).toDF()
                         Time  Exchange  Symbol  Bid_Price  Bid_Size  Offer_Price  Offer_Size
0  1970-01-01 09:30:00.005868        90    AAPL      26.89         1        27.10           6
1  1970-01-01 09:30:00.011058        90    AAPL      26.89        11        27.10           6
2  1970-01-01 09:30:00.031523        90    AAPL      26.89        13        27.10           6
3  1970-01-01 09:30:00.284623        80    AAPL      26.89         8        26.98           8
4  1970-01-01 09:30:00.454066        80    AAPL      26.89         8        26.98           1
>>>
 trades.merge_asof(quotes,on=["Symbol","Time"]).select(["Symbol","Time","Trade_Volume","Trade_Price","Bid_Price", "Bid_Size","Offer_Price", "
Offer_Size"]).top(5).toDF()
  Symbol                         Time          Trade_Volume  Trade_Price  Bid_Price  Bid_Size  \
0   AAPL  1970-01-01 08:00:00.022239                   300        27.00       26.9         1
1   AAPL  1970-01-01 08:00:00.022287                   500        27.25       26.9         1
2   AAPL  1970-01-01 08:00:00.022317                   335        27.26       26.9         1
3   AAPL  1970-01-01 08:00:00.022341                   100        27.27       26.9         1
4   AAPL  1970-01-01 08:00:00.022368                    31        27.40       26.9         1

  Offer_Price   Offer_Size
```

```
0       27.49           10
1       27.49           10
2       27.49           10
3       27.49           10
4       27.49           10
```

`merge_window`

`merge_window` 对应 DolphinDB 中的 window join(wj)，它是 asof join 的扩展。*leftBound* 参数和 *rightBound* 参数用于指定窗口的边界 w1 和 w2，对左表中最后一个连接列对应的时间为 t 的记录，在右表中选择 (t+w1) 到 (t+w2) 的时间并且其他连接列匹配的记录，然后对这些记录使用指定的聚合函数。

window join 和 prevailing window join 的唯一区别是，如果右表中没有与窗口左边界时间（即 t+w1）匹配的值，prevailing window join 会选择右表中 (t+w1) 之前的最近时间的记录作为 t+w1 时的记录。如果要使用 prevailing window join，需将 *prevailing* 参数设置为 True。

```
>>> trades.merge_window(quotes, -5000000000, 0, aggFunctions=["avg(Bid_Price)","avg(Offer_Price)"],
 on=["Symbol","Time"]).where("Time>=07:59:59").top(10).toDF()
 Time                        Exchange Symbol  Trade_Volume  Trade_Price  avg_Bid_Price  avg_Offer_Price
0 1970-01-01 08:00:00.022239        75   AAPL           300        27.00          26.90            27.49
1 1970-01-01 08:00:00.022287        75   AAPL           500        27.25          26.90            27.49
2 1970-01-01 08:00:00.022317        75   AAPL           335        27.26          26.90            27.49
3 1970-01-01 08:00:00.022341        75   AAPL           100        27.27          26.90            27.49
4 1970-01-01 08:00:00.022368        75   AAPL            31        27.40          26.90            27.49
5 1970-01-01 08:00:02.668076        68   AAPL          2434        27.42          26.75            27.36
6 1970-01-01 08:02:20.116025        68   AAPL            66        27.00            NaN              NaN
7 1970-01-01 08:06:31.149930        75   AAPL           100        27.25            NaN              NaN
8 1970-01-01 08:06:32.826399        75   AAPL           100        27.25            NaN              NaN
9 1970-01-01 08:06:33.168833        75   AAPL            74        27.25            NaN              NaN

[10 rows x 6 columns]
```

ols

`ols` 函数用于计算最小二乘回归系数，返回的结果是一个字典。

```
>>> trade = s.loadTable(tableName="trade",dbPath="dfs://valuedb")
>>> z = trade.select(['bid','ask','prc']).ols('PRC', ['BID', 'ASK'])
>>> z["ANOVA"]
    Breakdown     DF            SS            MS             F  Significance
0  Regression      2  2.689281e+08  1.344640e+08  1.214740e+10           0.0
1    Residual  13133  1.453740e+02  1.106937e-02           NaN           NaN
2       Total  13135  2.689282e+08           NaN           NaN           NaN
```

```
>>> z["RegressionStat"]
           item    statistics
0            R2      0.999999
1    AdjustedR2      0.999999
2      StdError      0.105211
3  Observations  13136.000000
>>> z["Coefficient"]
      factor      beta  stdError      tstat    pvalue
0  intercept  0.003710  0.001155   3.213171  0.001316
1        BID  0.605306  0.010517  57.552527  0.000000
2        ASK  0.394712  0.010515  37.537920  0.000000
>>> z["Coefficient"].beta[1]
0.6053064998964696
```

3.4.2 Database

在 Python API 中，可以使用 DolphinDB Python API 的原生方法来创建、使用数据库及数据表，本节将介绍如何创建数据库、通过数据库创建数据表。

Database & Session.database

Python API 将 DolphinDB 服务端的数据库对象句柄，在 API 包装为 Database 类，封装实现部分功能。通常使用 Session.database 方法构造。该方法部分参数可以参考 DolphinDB 用户手册-database。

接口如下：

```
Session.database(dbName=None, prititionType=None, parititions=None, dbPath=None, engine=None, atomic=None, chunkGranularity=None)
```

- *dbName*: 数据库句柄名称，创建数据库时可以不指定该参数。
- *partitionType*: 分区类型，可选项为 keys.SEQ/keys.VALUE/keys.RANGE/keys.LIST/keys.COMPO/keys.HASH。
- *partitions*: 描述如何进行分区，通常为 list 或者 np.ndarray。
- *dbPath*: 保存数据库的目录的路径。
- *engine*: 数据库存储引擎。
- *atomic*: 写入事务的原子性层级。
- *chunkGranularity*: 分区粒度，可选值为 "Table"/"DATABASE"。

数据库句柄 *dbName*

当加载已有数据库或创建数据库时，可以指定该参数，表示将数据库加载到内存后的句柄名称。如果不指定该参数，将会自动生成随机字符串作为句柄名称，可以通过 _getDbName() 方法获取。

例1：创建数据库时不指定 dbName

```
dbPath = "dfs://dbName"
if s.existsDatabase(dbPath):
    s.dropDatabase(dbPath)
db = s.database(partitionType=keys.VALUE, partitions=[1, 2, 3], dbPath=dbPath)

dbName = db._getDbName()
print(dbName)
print(s.run(dbName))
```

输出结果如下：

```
TMP_DB_15c2bf85DB
DB[dfs://dbName]
```

例2：创建数据库时指定 dbName

```
dbPath = "dfs://dbName"
if s.existsDatabase(dbPath):
    s.dropDatabase(dbPath)
db = s.database(dbName="testDB", partitionType=keys.VALUE, partitions=[1, 2, 3], dbPath=dbPath)

dbName = db._getDbName()
print(dbName)
print(s.run(dbName))
```

输出结果如下：

```
testDB
DB[dfs://dbName]
```

数据库路径 *dbPath* 和 分区参数 *partitionType/partitions*

调用 Session.database 创建数据库时，必须指定分区相关参数 partitionType/partitions。如果创建的数据库为内存数据库，则不需要指定 dbPath，如果创建的数据库为分区数据库，则必须指定 dbPath。

各种分区数据库创建方式如下：

准备环境：

```
import dolphindb as ddb
import dolphindb.settings as keys
import numpy as np
import pandas as pd

s = ddb.Session()
s.connect("localhost", 8848, "admin", "123456")
```

创建基于 VALUE 分区的数据库

按 date 分区：

```
dbPath="dfs://db_value_date"
if s.existsDatabase(dbPath):
    s.dropDatabase(dbPath)
dates=np.array(pd.date_range(start='20120101', end='20120110'), dtype="datetime64[D]")
db = s.database(dbName='mydb', partitionType=keys.VALUE, partitions=dates,dbPath=dbPath)
```

按 month 分区：

```
dbPath="dfs://db_value_month"
if s.existsDatabase(dbPath):
    s.dropDatabase(dbPath)
months=np.array(pd.date_range(start='2012-01', end='2012-10', freq="M"), dtype="datetime64[M]")
db = s.database(partitionType=keys.VALUE, partitions=months,dbPath=dbPath)
```

创建基于 RANGE 分区的数据库

按 INT 类型分区：

```
dbPath="dfs://db_range_int"
if s.existsDatabase(dbPath):
    s.dropDatabase(dbPath)
db = s.database(partitionType=keys.RANGE, partitions=[1, 11, 21], dbPath=dbPath)
```

创建基于 LIST 分区的数据库

按 SYMBOL 类型分区：

```
dbPath="dfs://db_list_sym"
if s.existsDatabase(dbPath):
```

```
    s.dropDatabase(dbPath)
db = s.database(partitionType=keys.LIST, partitions=[['IBM', 'ORCL', 'MSFT'], ['GOOG', 'FB']],dbPath=dbPath)
```

创建基于 HASH 分区的数据库

按 INT 类型分区：

```
dbPath="dfs://db_hash_int"
if s.existsDatabase(dbPath):
    s.dropDatabase(dbPath)
db = s.database(partitionType=keys.HASH, partitions=[keys.DT_INT, 3], dbPath=dbPath)
```

创建基于 COMPO 分区的数据库

以下脚本创建基于 COMPO 分区的数据库及数据表：第一层是基于 VALUE 的 date 类型分区，第二层是基于 RANGE 的 int 类型分区。

注意： 创建 COMPO 的子分区数据库的 dbPath 参数必须设置为空字符串或不设置。

```
db1 = s.database(partitionType=keys.VALUE, partitions=np.array(["2012-01-01", "2012-01-06"], dtype="datetime64[D]"))
db2 = s.database(partitionType=keys.RANGE, partitions=[1, 6, 11])
dbPath="dfs://db_compo_test"
if s.existsDatabase(dbPath):
    s.dropDatabase(dbPath)
db = s.database(partitionType=keys.COMPO, partitions=[db1, db2], dbPath=dbPath)
```

数据库引擎 *engine*

默认使用 OLAP 引擎创建数据库，如果希望使用其他引擎创建数据库，可以指定该参数。

创建 TSDB 引擎下的数据库

TSDB 引擎数据库的创建方法和 OLAP 几乎一致，只需要在 database 函数中指定 engine = "TSDB"，并在调用建表函数 createTable，createPartitionedTable 时指定 sortColumns。

```
dates = np.array(pd.date_range(start='20120101', end='20120110'), dtype="datetime64[D]")
dbPath = "dfs://tsdb"
if s.existsDatabase(dbPath):
    s.dropDatabase(dbPath)
db = s.database(partitionType=keys.VALUE, partitions=dates, dbPath=dbPath, engine="TSDB")
```

事务原子性层级 *atomic*

该参数表示写入事务的原子性层级，决定了是否允许并发写入同一分区，可选值为 "TRANS" 和 "CHUNK"，默认值为 "TRANS"。

- 设置为 "TRANS"，写入事务的原子性层级为事务，即一个事务写入多个分区时，若某个分区被其他写入事务锁定而出现写入冲突，则该事务的写入全部失败。因此，该设置下，不允许并发写入同一个分区。
- 设置为 "CHUNK"，写入事务的原子性层级为分区。若一个事务写入多个分区时，某分区被其它写入事务锁定而出现冲突，系统会完成其他分区的写入，同时对之前发生冲突的分区不断尝试写入，尝试数分钟后仍冲突才放弃。此设置下，允许并发写入同一个分区，但由于不能完全保证事务的原子性，可能出现部分分区写入成功而部分分区写入失败的情况。同时由于采用了重试机制，写入速度可能较慢。

分区粒度 *chunkGranularity*

该参数表示分区粒度，可选值为 "TABLE" 和 "DATABASE"。

- "Table"：表级分区，设置后支持同时写入同一分区的不同表。
- "DATABASE"：数据库级分区，设置后只支持同时写入不同分区。

注意： 指定该参数前，需要构造 Session 时设置 enableChunkGranularityConfig=True，否则该参数无效。

createTable

```
Database.createTable(table, tableName, sortColumns=None)
```

- *table*：Table 类对象，将根据该表的表结构在数据库中创建一个空的维度表。
- *tableName*：字符串，表示维度表的名称。
- *sortColumns*：字符串或字符串列表，用于指定表的排序列。写入的数据将按照 sortColumns 列进行排序。系统默认 sortColumns （指定多列时）排序列的最后一列为时间类型，其余列字段作为排序的索引列，称作 sort key。

该方法与 DolphinDB 服务器同名函数使用限制一致，请参阅 DolphinDB 用户手册-createTable。

下面的代码示例将在 TSDB 引擎数据库中创建一张基于 schema_t 表的结构、按 csymbol 列排序的维度表，表名为 pt。

```
dbPath = "dfs://createTable"
if s.existsDatabase(dbPath):
    s.dropDatabase(dbPath)
db = s.database(partitionType=keys.VALUE, partitions=[1, 2, 3], dbPath=dbPath, engine="TSDB")
s.run("schema_t = table(100:0, `ctime`csymbol`price`qty, [TIMESTAMP, SYMBOL, DOUBLE, INT])")
schema_t = s.table(data="schema_t")
pt = db.createTable(schema_t, "pt", ["csymbol"])
schema = s.run(f'schema(loadTable("{dbPath}", "pt"))')
print(schema["colDefs"])
```

输出结果如下：

```
     name typeString  typeInt  extra comment
0   ctime  TIMESTAMP       12    NaN
1 csymbol     SYMBOL       17    NaN
2   price     DOUBLE       16    NaN
3     qty        INT        4    NaN
```

createPartitionedTable

```
Database.createPartitionedTable(
    table, tableName, partitionColumns, compressMethods={}, sortColumns=None,
    keepDuplicates=None, sortKeyMappingFunction=None
)
```

- *table*：Table 类对象，将根据该表的表结构在数据库中创建一个空的分区表。
- *tableName*：字符串，表示分区表的名称。
- *partitionColumns*：字符串或字符串列表，表示分区列。
- *compressMethods*：字典，用于指定各列使用的压缩方法，键值分别为列名和压缩方法。
- *sortColumns*：字符串或字符串列表，用于指定表的排序列。写入的数据将按照 sortColumns 列进行排序。系统默认 sortColumns （指定多列时）排序列的最后一列为时间类型，其余列字段作为排序的索引列，称作 sort key。
- *keepDuplicates*：指定在每个分区内如何处理所有 sortColumns 之值皆相同的数据，提供以下选项：
  - "ALL"：保留所有数据，为默认值。
  - "LAST"：仅保留最新数据。
  - "FIRST"：仅保留第一条数据。
- *sortKeyMappingFunction*：DolphinDB 服务端函数名字符串列表，其长度与索引列一致，用于指定各索引列使用的排序方法。

该方法与 DolphinDB 服务器同名函数使用限制一致，请参阅 DolphinDB 用户手册-createPartitionedTable。

例1：下面的代码示例将在 TSDB 引擎数据库中根据 schema_t 表的结构创建一张分区列为TradeDate、索引列为 sortColumns 的分区表，并指定排序列为 SecurityID 和 TradeDate，其中 SecurityID 的排序函数使用 hashBucket{,5}，每个分区排序列值相同时的处理策略为 "ALL"。

```
dbPath = "dfs://createPartitionedTable"
if s.existsDatabase(dbPath):
    s.dropDatabase(dbPath)
```

```
dates = np.array(pd.date_range(start='20220101', end='20220105'), dtype="datetime64[D]")
db = s.database(partitionType=keys.VALUE, partitions=dates, dbPath=dbPath, engine="TSDB")
s.run("schema_t = table(100:0, `SecurityID`TradeDate`TotalVolumeTrade`TotalValueTrade, [SYMBOL, DATE, INT, DOUBLE])")
schema_t = s.table(data="schema_t")
pt = db.createPartitionedTable(schema_t, "pt", partitionColumns="TradeDate", sortColumns=["SecurityID", "TradeDate"], keepDuplicates="ALL",
 sortKeyMappingFunction=["hashBucket{,5}"])
schema = s.run(f'schema(loadTable("{dbPath}", "pt"))')
print(schema["colDefs"])
```

输出结果如下：

```
              name typeString  typeInt  extra comment
0       SecurityID     SYMBOL       17    NaN
1        TradeDate       DATE        6    NaN
2  TotalVolumeTrade        INT        4    NaN
3   TotalValueTrade     DOUBLE       16    NaN
```

例2：下面的代码示例将在 OLAP 引擎数据库中根据 schema_t 的表结构创建一张分区列为 symbol 的分区表，并指定 timestamp 列压缩方式为 delta。

```
dbPath = "dfs://createPartitionedTable"
if s.existsDatabase(dbPath):
    s.dropDatabase(dbPath)
db = s.database(partitionType=keys.VALUE, partitions=["IBM", "MS"], dbPath=dbPath)
s.run("schema_t = table(100:0, `timestamp`symbol`value, [TIMESTAMP, SYMBOL, DOUBLE])")
schema_t = s.table(data="schema_t")
pt = db.createPartitionedTable(schema_t, "pt", partitionColumns="symbol", compressMethods={'timestamp': "delta"})
schema = s.run(f'schema(loadTable("{dbPath}", "pt"))')
print(schema["colDefs"])
```

输出结果如下：

```
       name typeString  typeInt  extra comment
0  timestamp  TIMESTAMP       12    NaN
1     symbol     SYMBOL       17    NaN
2      value     DOUBLE       16    NaN
```

3.5.1 强制终止进程

Session 对象中提供静态方法 `enableJobCancellation()`，用于开启强制终止进程的功能。此功能默认关闭。开启后，可通过 “Ctrl+C” 按键终止 API 进程中所有 Session 提交的正在运行的作业。目前，该功能仅在 Linux 系统生效。

示例：

```
ddb.Session.enableJobCancellation()
```

Chapter 4. 参考阅读

4.1 流数据应用

4.2 动量交易策略

4.3 时间序列计算

4.1 流数据应用

4.1.1 流数据应用

本节介绍实时 K 线计算的三个步骤。

(1) 使用 Python 接收实时数据，并写入 DolphinDB 流数据表

- DolphinDB 中建立流数据表

```
share streamTable(100:0, `Symbol`Datetime`Price`Volume,[SYMBOL,DATETIME,DOUBLE,INT]) as Trade
```

- Python 程序从数据源 trades.csv 文件中读取数据写入 DolphinDB。

实时数据中 Datetime 的数据精度是秒，由于 pandas DataFrame 中仅能使用 DateTime 即 nanatimestamp 类型，所以下列代码在写入前有一个数据类型转换的过程。这个过程也适用于大多数数据需要清洗和转换的场景。

```
import dolphindb as ddb
import pandas as pd
import numpy as np

csv_file = WORK_DIR + "/trades.csv"
csv_data = pd.read_csv(csv_file,parse_dates=['Datetime'], dtype={'Symbol':str})
csv_df = pd.DataFrame(csv_data)
s = ddb.session()
s.connect("192.168.1.103", 8921,"admin","123456")
#上传 DataFrame 到 DolphinDB，并对 Datetime 字段做类型转换
s.upload({"tmpData":csv_df})
s.run("data = select Symbol, datetime(Datetime) as Datetime, Price, Volume from tmpData;tableInsert(Trade,data)")
```

这个方法的缺点是，s.upload 和 s.run 涉及两次网络数据传输，有可能会出现网络延迟。可以考虑先在 Python 端中处理数据，然后再单步 `tableInsert` 到服务器端，减少网络传输次数。

```
csv_df=csv_df[['Symbol', 'Datetime', 'Price', 'Volume']]
s.run("tableInsert{Trade}", csv_df)
```

(2) 实时计算 K 线

本例中使用时序聚合引擎实时计算 K 线数据，并将计算结果输出到流数据表 OHLC 中。

计算 K 线数据，按照计算时间窗口是否存在重合分为两种计算场景：一是时间窗口不重合，比如每隔 5 分钟计算一次过去 5 分钟的 K 线数据；二是时间窗口部分重合，比如每隔 1 分钟计算过去 5 分钟的 K 线数据。

可通过设定 `createTimeSeriesAggregator` 函数的 windowSize 和 step 参数以实现这两个场景。场景一 windowSize 与 step 相等；场景二 windowSize 是 step 的倍数。

首先定义输出表:

```
share streamTable(100:0, `datetime`symbol`open`high`low`close`volume,[DATETIME, SYMBOL, DOUBLE,DOUBLE,DOUBLE,DOUBLE,LONG]) as OHLC
```

根据应用场景的不同，在以下两行代码中选择一行，以定义时序聚合引擎：

场景一：

```
tsAggrKline = createTimeSeriesAggregator(name="aggr_kline", windowSize=300, step=300, metrics=<[first(Price),max(Price),min(Price),last(Price),sum(volume)]>, dummyTable=Trade,
 outputTable=OHLC, timeColumn=`Datetime, keyColumn=`Symbol)
```

场景二：

```
tsAggrKline = createTimeSeriesAggregator(name="aggr_kline", windowSize=300, step=60, metrics=<[first(Price),max(Price),min(Price),last(Price),sum(volume)]>, dummyTable=Trade,
 outputTable=OHLC, timeColumn=`Datetime, keyColumn=`Symbol)
```

最后，定义流数据订阅。若此时流数据表 Trade 中已经有实时数据写入，那么实时数据会马上被订阅并注入聚合引擎：

```
subscribeTable(tableName="Trade", actionName="act_tsaggr", offset=0, handler=append!{tsAggrKline}, msgAsTable=true)
```

(3) 在 Python 中展示 K 线数据

在本例中，聚合引擎的输出表也定义为流数据表，客户端可以通过 Python API 订阅输出表，并将计算结果展现到 Python 终端。

以下代码使用 Python API 订阅实时聚合计算的输出结果表 OHLC，并将结果通过 print 函数打印出来。

```
from threading import Event
import dolphindb as ddb
import pandas as pd
```

```
import numpy as np
s=ddb.session()
#设定本地端口 20001 用于订阅流数据
s.enableStreaming(20001)
def handler(lst):
    print(lst)
# 订阅 DolphinDB(本机 8848 端口) 上的 OHLC 流数据表
s.subscribe("192.168.1.103", 8921, handler, "OHLC")
Event().wait()

# output
[numpy.datetime64('2018-09-03T09:31:00'), '000001', 10.13, 10.15, 10.1, 10.14, 586160]
[numpy.datetime64('2018-09-03T09:32:00'), '000001', 10.13, 10.16, 10.1, 10.15, 1217060]
[numpy.datetime64('2018-09-03T09:33:00'), '000001', 10.13, 10.16, 10.1, 10.13, 1715460]
[numpy.datetime64('2018-09-03T09:34:00'), '000001', 10.13, 10.16, 10.1, 10.14, 2268260]
[numpy.datetime64('2018-09-03T09:35:00'), '000001', 10.13, 10.21, 10.1, 10.2, 3783660]
...
```

也可通过 Grafana 等可视化系统来连接 DolphinDB database，对输出表进行查询并将结果以图表方式展现。

4.2 动量交易策略

11.1 动量交易策略

下面的例子使用动量交易策略进行回测。最常用的动量因子是过去一年扣除最近一个月的收益率。本例中，每天调整 1/5 的投资组合，并持有新的投资组合 5 天。为了简化起见，不考虑交易成本。

Create server session

```
import dolphindb.settings as keys
s=ddb.session()
s.connect("localhost",8921, "admin", "123456")
```

步骤 1：加载股票交易数据，对数据进行清洗和过滤，然后为每只股票构建过去一年扣除最近一个月收益率的动量信号。注意，必须使用 `executeAs` 把中间结果保存到 DolphinDB 服务器上。数据集 "US" 包含了美国股票 1990 到 2016 年的交易数据。

```
if s.existsDatabase("dfs://US"):
 s.dropDatabase("dfs://US")
s.database(dbName='USdb', partitionType=keys.VALUE, partitions=["GFGC","EWST", "EGAS"], dbPath="dfs://US")
US=s.loadTextEx(dbPath="dfs://US", partitionColumns=["TICKER"], tableName='US', remoteFilePath=WORK_DIR + "/US.csv")
```

```
US = s.loadTable(dbPath="dfs://US", tableName="US")
def loadPriceData(inData):
    s.loadTable(inData).select("PERMNO, date, abs(PRC) as PRC, VOL, RET, SHROUT*abs(PRC) as MV").where("weekday(date) between 1:5,
 isValid(PRC), isValid(VOL)").sort(bys=["PERMNO","date"]).executeAs("USstocks")
    s.loadTable("USstocks").select("PERMNO, date, PRC, VOL, RET, MV, cumprod(1+RET) as
 cumretIndex").contextby("PERMNO").executeAs("USstocks")
    return s.loadTable("USstocks").select("PERMNO, date, PRC, VOL, RET, MV, move(cumretIndex,21)/move(cumretIndex,252)-1 as
 signal").contextby("PERMNO").executeAs("priceData")

priceData = loadPriceData(US.tableName())
# US.tableName() returns the name of the table on the DolphinDB server that corresponds to the table object "US" in Python.
```

步骤 2：为动量策略生成投资组合

```
def genTradeTables(inData):
    return s.loadTable(inData).select(["date", "PERMNO", "MV", "signal"]).where("PRC>5, MV>100000, VOL>0,
 isValid(signal)").sort(bys=["date"]).executeAs("tradables")

def formPortfolio(startDate, endDate, tradables, holdingDays, groups, WtScheme):
    holdingDays = str(holdingDays)
    groups=str(groups)
    ports = tradables.select("date, PERMNO, MV, rank(signal,,"+groups+") as rank, count(PERMNO) as symCount, 0.0 as wt").where("date between
 "+startDate+":"+endDate).contextby("date").having("count(PERMNO)>=100").executeAs("ports")
    if WtScheme == 1:
        ports.where("rank=0").contextby("date").update(cols=["wt"], vals=["-1.0/count(PERMNO)/"+holdingDays]).execute()
        ports.where("rank="+groups+"-1").contextby("date").update(cols=["wt"], vals=["1.0/count(PERMNO)/"+holdingDays]).execute()
    elif WtScheme == 2:
        ports.contextby("date").update(cols=["wt"], vals=["-MV/sum(MV)/"+holdingDays]).where("rank=0").execute()
        ports.contextby("date").update(cols=["wt"], vals=["MV/sum(MV)/"+holdingDays]).where("rank="+groups+"-1").execute()
    else:
        raise Exception("Invalid WtScheme. valid values:1 or 2")
    return ports.select("PERMNO, date as tranche, wt").where("wt!=0.0").sort(bys=["PERMNO","date"]).executeAs("ports")

tradables=genTradeTables(priceData.tableName())
startDate="2016.01.01"
endDate="2017.01.01"
holdingDays=5
groups=10
ports=formPortfolio(startDate=startDate,endDate=endDate,tradables=tradables,holdingDays=holdingDays,groups=groups,WtScheme=2)
dailyRtn=priceData.select("date, PERMNO, RET as dailyRet").where("date between "+startDate+":"+endDate).executeAs("dailyRtn")
```

步骤 3：计算投资组合中每只股票接下来 5 天的利润或损失。在投资组合形成后的 5 天后关停投资组合。

```
def calcStockPnL(ports,inData, dailyRtn, holdingDays, endDate):
    s.table(data={'age': list(range(1,holdingDays+1))}).executeAs("ages")
    ports.select("tranche").sort("tranche").executeAs("dates")
    s.run("dates = sort distinct dates.tranche")
    s.run("dictDateIndex=dict(dates,1..dates.size())")
    s.run("dictIndexDate=dict(1..dates.size(), dates)")
    inData.select("max(date) as date").groupby("PERMNO").executeAs("lastDaysTable")
    s.run("lastDays=dict(lastDaysTable.PERMNO,lastDaysTable.date)")
    ports.merge_cross(s.table(data="ages")).select("dictIndexDate[dictDateIndex[tranche]+age] as date, PERMNO, tranche, age,
 take(0.0,age.size()) as ret, wt as expr, take(0.0,age.size()) as pnl").where("isValid(dictIndexDate[dictDateIndex[tranche]+age]),
 dictIndexDate[dictDateIndex[tranche]+age]<=min(lastDays[PERMNO],"+endDate+")").executeAs("pos")
    t1= s.loadTable("pos")
    # t1.merge(dailyRtn, on=["date","PERMNO"], merge_for_update=True).update(["ret"],["dailyRet"]).execute()
    t1.merge(dailyRtn, on=["date","PERMNO"]).update(["ret"],["dailyRet"]).execute()

    t1.contextby(["PERMNO","tranche"]).update(["expr"], ["expr*cumprod(1+ret)"]).execute()
    t1.update(["pnl"],["expr*ret/(1+ret)"]).execute()
    return t1


stockPnL = calcStockPnL(ports=ports,inData=priceData, dailyRtn=dailyRtn, holdingDays=holdingDays, endDate=endDate)
```

步骤 4：计算投资组合的利润或损失。

```
portPnl = stockPnL.select("sum(pnl)").groupby("date").sort(bys=["date"]).executeAs("portPnl")
print(portPnl.toDF())
```

4.3 时间序列计算

下面的例子计算 "101 Formulaic Alphas" by Kakushadze (2015) 中的 98 号因子。

```
def alpha98(t):
    t1 = s.table(data=t)
    t1.contextby(["date"]).update(cols=["rank_open","rank_adv15"], vals=["rank(open)","rank(adv15)"]).execute()
    t1.contextby(["PERMNO"]).update(["decay7", "decay8"], ["mavg(mcorr(vwap, msum(adv5, 26), 5), 1..7)",\
               "mavg(mrank(9 - mimin(mcorr(rank_open, rank_adv15, 21), 9), true, 7), 1..8)"]).execute()
    # from previous update the server's schema is changed, so you may reload it
    t1 = s.table(data=t)
```

```
    return t1.select("PERMNO, date,decay7, decay8, rank(decay7)-rank(decay8) as A98")\
        .contextby(["date"])\
        .executeAs("alpha98")


US = s.loadTable(tableName="US", dbPath="dfs://US")\
    .select("PERMNO, date, PRC as vwap, PRC+rand(1.0, PRC.size()) as open, mavg(VOL, 5) as adv5\
            , mavg(VOL,15) as adv15")\
    .where("2007.01.01<=date<=2016.12.31")\
    .contextby("PERMNO")\
    .executeAs("US")
result=alpha98(US.tableName()).where('date>2007.03.12').executeAs("result")
print(result.toDF())
s.close()
```

Contents

Chapter 5. 快速上手 5

1.1 介绍 5

如何安装 DolphinDB Python API 6

DolphinDB Python API 快速上手指南 7

DolphinDB Python API 的常用操作 14

Chapter 6. 基础操作 21

如何构建 Session 21

如何使用 Connect 方法建立连接 26

Session 相关的常用方法 29

构造 DBConnectionPool 39

方法介绍 42

TableAppender 50

TableUpserter 53

PartitionedTableAppender 57

订阅 59

Session 异步提交 63

MultithreadedTableWriter 66

BatchTableWriter 80

Chapter 7. 进阶操作 85

PROTOCOL_DDB 85

PROTOCOL_PICKLE 102

PROTOCOL_ARROW 105

强制类型转换 108

类型转换 109

多种写入方案 111

流订阅模式 116

3.4.1 Table 123

3.4.2 Database 142

3.5.1 强制终止进程 148

Chapter 8. 参考阅读 151

4.1 流数据应用 151

4.2 动量交易策略 153

4.3 时间序列计算 155

Index ..... a

Glossary ii

Index

P

PythonAPI
see API , Python

Glossary

WMD

Nuclear, biological, or chemical weapons able to cause widespread devastation and loss of life.

Weapons of Mass Destruction (WMD)