pyflink-docs

Release release-1.15

PyFlink

Nov 23, 2022

CONTENTS

1 How to build docs locally **3**

1.1 Getting Started . . . 3
1.1.1 Installation . . . 3
1.1.1.1 Preparation . . . 3
1.1.1.2 Local . . . 6
1.1.1.3 Standalone . . . 7
1.1.1.4 YARN . . . 8
1.1.1.5 Kubernetes . . . 11
1.1.2 QuickStart . . . 12
1.1.2.1 QuickStart: Table API . . . 12
1.1.2.2 QuickStart: DataStream API . . . 19
1.2 User Guide . . . 22
1.2.1 RealTime Feature . . . 22
1.2.1.1 Coming Soon. . . . 22
1.2.2 PyFlink + Flink ML . . . 22
1.2.2.1 Coming Soon. . . . 22
1.3 Frequently Asked Questions (FAQ) . . . 22
1.3.1 Installation issues . . . 22
1.3.1.1 O1: Scala Dependency . . . 22
1.3.1.2 O2: Java gateway process exited before sending its port number . . . 22
1.3.2 Usage issues . . . 24
1.3.2.1 O1: How to prepare Python Virtual Environment . . . 24
1.3.2.2 O2: How to add Python Files . . . 25
1.3.3 JDK issues . . . 25
1.3.3.1 O1: InaccessibleObjectException: Unable to make field private final byte[] java.lang.String.value accessible: module java.base does not "opens java.lang" to unnamed module @4e4aea35 . . . 25
1.3.4 Connector issues . . . 26
1.3.4.1 O1: Could not find any factory for identifier 'xxx' that implements 'org.apache.flink.table.factories.DynamicTableFactory' in the classpath . . . 26
1.3.4.2 O2: ClassNotFoundException: com.mysql.cj.jdbc.Driver . . . 29
1.3.4.3 O3: NoSuchMethodError: org.apache.flink.table.factories.DynamicTableFactory$Context.getCatalogTable()
1.3.5 Runtime issues . . . 30
1.3.5.1 Q1: OverflowError: timeout value is too large . . . 30
1.3.5.2 Q2: An error occurred while calling z:org.apache.flink.client.python.PythonEnvUtils.resetCallbackClient 30
1.3.6 Data type issues . . . 31
1.3.6.1 Q1: 'tuple' object has no attribute '_values' . . . 31
1.3.6.2 Q2: AttributeError: 'int' object has no attribute 'encode' . . . 32
1.3.6.3 Q3: Types.BIG_INT() VS Types.LONG() . . . 32
1.4 API reference . . . 32

PyFlink is a Python API for Apache Flink that allows you to build scalable batch and streaming workloads, such as real-time data processing pipelines, large-scale exploratory data analysis, Machine Learning (ML) pipelines and ETL processes. If you're already familiar with Python and libraries such as Pandas, then PyFlink makes it simpler to leverage the full capabilities of the Flink ecosystem. Depending on the level of abstraction you need, there are two different APIs that can be used in PyFlink: PyFlink Table API and PyFlink DataStream API.

CHAPTER

ONE

HOW TO BUILD DOCS LOCALLY

1. Install dependency requirements

```
python3 -m pip install -r dev/requirements.txt
```

2. Conda install pandoc

```
conda install pandoc
```

3. Build the docs

```
python3 setup.py build_sphinx
```

4. Open the `pyflink-docs/build/sphinx/html/index.html` in the Browser

1.1 Getting Started

This page summarizes the basic steps required to setup and get started with PyFlink.

There are live notebooks where you can try PyFlink out without any other step:

- Live Notebook: Table
- Live Notebook: DataStream

The list below is the contents of this quickstart page:

1.1.1 Installation

1.1.1.1 Preparation

This page shows you how to install PyFlink using pip, conda, installing from the source, etc.

Python Version Supported

PyFlink Version	Python Version Supported
PyFlink 1.16	Python 3.6 to 3.9
PyFlink 1.15	Python 3.6 to 3.8
PyFlink 1.14	Python 3.6 to 3.8

You could check your Python version as following:

```
python3 --version
```

Create a Python virtual environment

Virtual environment gives you the ability to isolate the Python dependencies of different projects by creating a separate environment for each project. It is a directory tree which contains its own Python executable files and the installed Python packages.

It is useful for local development to create a standalone Python environment and also useful when deploying a PyFlink job to production when there are massive Python dependencies. It's supported to use Python virtual environment in your PyFlink jobs, see PyFlink Dependency Management for more details.

Create a virtual environment using virtualenv

To create a virtual environment using virtualenv, run:

```
python3 -m pip install virtualenv

# Create Python virtual environment under a directory, e.g. venv
virtualenv venv

# You can also create Python virtual environment with a specific Python version
virtualenv --python /path/to/python/executable venv
```

The virtual environment needs to be activated before to use it. To activate the virtual environment, run:

```
source venv/bin/activate
```

That is, execute the activate script under the bin directory of your virtual environment.

Create a virtual environment using conda

To create a virtual environment using conda (suppose miniconda), run:

```
# Download and install miniconda, the latest miniconda installers are available in https:
↪//repo.anaconda.com/miniconda/

# Suppose the name of the downloaded miniconda installer is miniconda.sh
chmod +x miniconda.sh
# install miniconda
./miniconda.sh -b -p miniconda

# Activate the miniconda environment
source miniconda/bin/activate

# Create conda virtual environment under a directory, e.g. venv
conda create --name venv python=3.8 -y
```

The conda virtual environment needs to be activated before to use it. To activate the conda virtual environment, run:

```
conda activate venv
```

Install PyFlink

You could then install the latest PyFlink package into your virtual environment. Note that the Flink version and PyFlink version need to be consistent. For example, if you are using Flink 1.15, then you should use PyFlink 1.15

Installing using PyPI

PyFlink could be installed using PyPI as following:

```
python3 -m pip install apache-flink
```

Installing from Source

To install PyFlink from source, you could refer to Build PyFlink.

Check the installed package

You could then perform the following checks to make sure that the installed PyFlink package is ready for use:

```
curl -L https://raw.githubusercontent.com/apache/flink/master/flink-python/pyflink/
↪examples/table/word_count.py -o word_count.py
python3 word_count.py
# You will see outputs as following:
# Use --input to specify file input.
# Printing result to stdout. Use --output to specify output path.
# +I[To, 1]
# +I[be,, 1]
# +I[or, 1]
# +I[not, 1]
# +I[to, 1]
# +I[be,--that, 1]
# ...
```

If there are any problems, you could perform the following checks.

Check the logging messages in the log file to see if there are any problems:

```
# Get the installation directory of PyFlink
python3 -c "import pyflink;import os;print(os.path.dirname(os.path.abspath(pyflink.__
↪file__)))"
# It will output a path like the following:
# /path/to/python/site-packages/pyflink

# Check the logging under the log directory
ls -lh /path/to/python/site-packages/pyflink/log
# You will see the log file as following:
```

(continues on next page)

(continued from previous page)

```
#  -rw-r--r--  1 dianfu  staff    45K 10 18 20:54 flink-dianfu-python-B-7174MD6R-1908.
↪local.log
```

Besides, you could also check if the files of the PyFlink package are consistent. It may happen that you have installed an old version of PyFlink before and multiple PyFlink versions exist at the same time for some reason.

```
# List the jar packages under the lib directory
ls -lh /path/to/python/site-packages/pyflink/lib
# It will output a list of jar packages as following:
#  -rw-r--r--  1 dianfu  staff   190K 10 18 20:43 flink-cep-1.15.2.jar
#  -rw-r--r--  1 dianfu  staff   475K 10 18 20:43 flink-connector-files-1.15.2.jar
#  -rw-r--r--  1 dianfu  staff    93K 10 18 20:43 flink-csv-1.15.2.jar
#  -rw-r--r--  1 dianfu  staff   110M 10 18 20:43 flink-dist-1.15.2.jar
#  -rw-r--r--  1 dianfu  staff   171K 10 18 20:43 flink-json-1.15.2.jar
#  -rw-r--r--  1 dianfu  staff    20M 10 18 20:43 flink-scala_2.12-1.15.2.jar
#  -rw-r--r--  1 dianfu  staff    10M 10 18 20:43 flink-shaded-zookeeper-3.5.9.jar
#  -rw-r--r--  1 dianfu  staff    15M 10 18 20:43 flink-table-api-java-uber-1.15.2.jar
#  -rw-r--r--  1 dianfu  staff    35M 10 18 20:43 flink-table-planner-loader-1.15.2.jar
#  -rw-r--r--  1 dianfu  staff   2.9M 10 18 20:43 flink-table-runtime-1.15.2.jar
#  -rw-r--r--  1 dianfu  staff   203K 10 18 20:43 log4j-1.2-api-2.17.1.jar
#  -rw-r--r--  1 dianfu  staff   295K 10 18 20:43 log4j-api-2.17.1.jar
#  -rw-r--r--  1 dianfu  staff   1.7M 10 18 20:43 log4j-core-2.17.1.jar
#  -rw-r--r--  1 dianfu  staff    24K 10 18 20:43 log4j-slf4j-impl-2.17.1.jar
```

Please make sure that the versions of all the Flink jar packages are consistent, e.g. 1.15.2 in the above example.

1.1.1.2 Local

This page shows you how to set up PyFlink development environment in your local machine. This is usually used for local execution or development in an IDE.

Set up Python environment

It requires Python 3.6 or above with PyFlink pre-installed to be available in your local environment. It's suggested to use Python virtual environments to set up your local Python environment. See *Create a Python virtual environment* for more details on how to prepare Python virtual environments with PyFlink installed.

Execute PyFlink jobs in terminal

You could execute PyFlink jobs locally as following:

```
curl -L https://raw.githubusercontent.com/apache/flink/master/flink-python/pyflink/
↪examples/table/word_count.py -o word_count.py
python3 word_count.py
```

If there any any problems, you could check the logging messages in the log file as following:

```
# Get the installation directory of PyFlink
python3 -c "import pyflink;import os;print(os.path.dirname(os.path.abspath(pyflink.__
↪file__)))"
```

(continues on next page)

(continued from previous page)

```
# It will output a path like the following:
# /path/to/python/site-packages/pyflink

# Check the logging under the log directory
ls -lh /path/to/python/site-packages/pyflink/log
# You will see the log file as following:
#  -rw-r--r--  1 dianfu  staff    45K 10 18 20:54 flink-dianfu-python-B-7174MD6R-1908.
↪local.log
```

Execute PyFlink jobs in IDE

You need firstly configure the Python virtual environment for your IDE. See Configure a virtual environment for more details on how to configure the Python virtual environment in IntelliJ IDEA.

Right click on the job file and execute it. If there are any problems, you could check the logging messages in the log file which resides under the log directory of PyFlink installation directory as following:

```
# Check the logging under the log directory of PyFlink
ls -lh /path/to/python/site-packages/pyflink/log
```

1.1.1.3 Standalone

The standalone mode is the most barebone way of deploying Flink. This page shows you how to set up Python environment and execute PyFlink jobs in a standalone Flink cluster.

Set up Python environment

It requires Python 3.6 or above with PyFlink pre-installed to be available on the nodes of the standalone cluster. It's suggested to use Python virtual environments to set up the Python environment. See *Create a Python virtual environment* for more details on how to prepare Python virtual environments with PyFlink installed.

Once the Python virtual environment is available, it needs to be deployed on the cluster. There are the following options to deploy it:

- Install Python virtual environments on all the cluster nodes in advance

You could install Python virtual environments on all the cluster nodes with PyFlink pre-installed before submitting PyFlink jobs. Note that if you have a lot of jobs which use different Python versions and Flink versions, you could create multiple Python virtual environments to isolate them. For each PyFlink job, it could choose one of these Python virtual environments to use.

```
./bin/flink run \
      --jobmanager <jobmanagerHost>:8081 \
      -pyclientexec /path/to/venv/bin/python3 \
      -pyexec /path/to/venv/bin/python3 \
      -py word_count.py
```

In the above example, it assumes that there is already a Python virtual environment available at /path/to/venv on all the cluster nodes of the standaone cluster. It should be noted that options **-pyclientexec** and **-pyexec** are also required to specify to use the given Python virtual environment at client side (for job compiling) and server side (for Python UDF execution) separately.

- Specify the Python virtual environments during submitting PyFlink jobs

It also supports to distribute the Python virtual environment during submitting PyFlink jobs. In this way, the Python virtual environment will be distributed to the cluster nodes where PyFlink jobs are running on during job starting up. This is more flexible and useful when it's not possible to set up the Python environments in advance on the cluster nodes or when there are some special requirements where the pre-installed Python environments could not meet.

```
./bin/flink run \
      --jobmanager <jobmanagerHost>:8081 \
      -pyarch /path/to/venv.zip \
      -pyclientexec /path/to/venv/bin/python3 \
      -pyexec venv.zip/venv/bin/python3 \
      -py word_count.py
```

In the above example, the Python virtual environment is specified via option **-pyarch**. It will be distributed to the cluster nodes during job execution. It should be noted that option **-pyexec** is also required to specify the Python virtual environment to use at server side (for Python UDF execution).

For the Python virtual environment at client side (for job compiling), option **-pyclientexec** could be used. If it's not specified, it will use the Python environment of the current shell.

- Mix use of the above options

You could also mix use of the above options, that is, pre-install a few commonly used Python virtual environments on the cluster nodes of the standalone cluster and use custom Python virtual environment when there are some special requirements.

Submit PyFlink jobs to a standalone Flink cluster

You could submit PyFlink jobs to a standalone Flink cluster as following:

```
./bin/flink run \
          --jobmanager <jobmanagerHost>:8081 \
          -pyarch /path/to/venv.zip \
          -pyexec venv.zip/venv/bin/python3 \
          -py word_count.py
```

See Submitting PyFlink jobs for more details.

1.1.1.4 YARN

Apache Hadoop YARN is a cluster resource management framework for managing the resources and scheduling jobs in a Hadoop cluster. It's supported to submit PyFlink jobs to YARN for execution.

Set up Python environment

It requires Python 3.6 or above with PyFlink pre-installed to be available on the nodes of the YARN cluster. It's suggested to use Python virtual environments to set up the Python environment. See *Create a Python virtual environment* for more details on how to prepare Python virtual environments with PyFlink installed.

Once the Python virtual environment is available, it needs to be deployed on the cluster. There are the following options to deploy it:

- Install Python virtual environments on all the cluster nodes in advance

You could install Python virtual environments on all the cluster nodes with PyFlink pre-installed before submitting PyFlink jobs. Note that if you have a lot of jobs which use different Python versions and Flink versions, you could create multiple Python virtual environments to isolate them. For each PyFlink job, it could choose one of these Python virtual environments to use.

```
./bin/flink run-application -t yarn-application \
      -Djobmanager.memory.process.size=1024m \
      -Dtaskmanager.memory.process.size=1024m \
      -Dyarn.application.name=<ApplicationName> \
      -pyclientexec /path/to/venv/bin/python3 \
      -pyexec /path/to/venv/bin/python3 \
      -py word_count.py
```

In the above example, it assumes that there is already a Python virtual environment available at /path/to/venv on all the cluster nodes. It should be noted that options **-pyclientexec** and **-pyexec** are also required to specify to use the given Python virtual environment at client side (for job compiling) and server side (for Python UDF execution) separately.

- Specify the Python virtual environments during submitting PyFlink jobs

It also supports to distribute the Python virtual environment during submitting PyFlink jobs. In this way, the Python virtual environment will be distributed to the cluster nodes where PyFlink jobs are running on during job starting up. This is more flexible and useful when it's not possible to set up the Python environments in advance on the cluster nodes or when there are some special requirements where the pre-installed Python environments could not meet.

```
./bin/flink run-application -t yarn-application \
      -Djobmanager.memory.process.size=1024m \
      -Dtaskmanager.memory.process.size=1024m \
      -Dyarn.application.name=<ApplicationName> \
      -Dyarn.ship-files=/path/to/shipfiles \
      -pyarch shipfiles/venv.zip \
      -pyclientexec venv.zip/venv/bin/python3 \
      -pyexec venv.zip/venv/bin/python3 \
      -py shipfiles/word_count.py
```

In the above example, the Python virtual environment is specified via option **-pyarch**. It will be distributed to the cluster nodes during job execution. It should be noted that options **-pyclientexec** and **-pyexec** are also required to specify to use the given Python virtual environment at client side (for job compiling) and server side (for Python UDF execution) separately.

Note: It assumes that the Python dependencies needed to execute the job are already placed in the directory /path/to/shipfiles. For example, it should contain venv.zip and word_count.py for the above example.

As it executes the job on the JobManager in YARN application mode, the paths specified in **-pyarch** and **-py** are paths relative to shipfiles which is the directory name of the shipped files.

The archive files specified via **-pyarch** will be distributed to the TaskManagers through blob server where the file size limit is 2 GB. If the size of an archive file is more than 2 GB, you could upload it to a distributed file system and then use the path in the command line option **-pyarch**.

- Mix use of the above options

You could also mix use of the above options, that is, pre-install a few commonly used Python virtual environments on the cluster nodes and use custom Python virtual environment when there are some special requirements.

Submit PyFlink jobs to YARN cluster

It supports to execute PyFlink jobs in application mode, per-job mode and session mode in YARN deployment.

You could execute PyFlink jobs in application mode as following:

```
./bin/flink run-application -t yarn-application \
      -Djobmanager.memory.process.size=1024m \
      -Dtaskmanager.memory.process.size=1024m \
      -Dyarn.application.name=<ApplicationName> \
      -Dyarn.ship-files=/path/to/shipfiles \
      -pyarch shipfiles/venv.zip \
      -pyclientexec venv.zip/venv/bin/python3 \
      -pyexec venv.zip/venv/bin/python3 \
      -py shipfiles/word_count.py
```

You could execute PyFlink jobs in per-job mode as following:

```
./bin/flink run -t yarn-per-job \
      -Djobmanager.memory.process.size=1024m \
      -Dtaskmanager.memory.process.size=1024m \
      -Dyarn.application.name=<ApplicationName> \
      -Dyarn.ship-files=/path/to/shipfiles \
      -pyarch shipfiles/venv.zip \
      -pyclientexec /path/to/venv/bin/python3 \
      -pyexec venv.zip/venv/bin/python3 \
      -py shipfiles/word_count.py
```

Note: Per-Job mode has been deprecated since Flink 1.15 and may be dropped in the future releases. It's suggested to use Application mode. See YARN Per-Job Mode for more details.

It should be noted that there are some differences compared with the application mode. For option **-pyclientexec**, it should point to a path on the client node (node executing the above command) as the job is compiled at the client side in per-job mode. If it's not specified, it will use the Python environment of the current shell environment.

You could also execute PyFlink jobs in session mode as following:

```
./bin/flink run -t yarn-session \
          -Djobmanager.memory.process.size=1024m \
          -Dtaskmanager.memory.process.size=1024m \
          -Dyarn.application.id=<application_XXXX_YY> \
          -Dyarn.ship-files=/path/to/shipfiles \
          -pyarch shipfiles/venv.zip \
          -pyclientexec /path/to/venv/bin/python3 \
          -pyexec venv.zip/venv/bin/python3 \
          -py shipfiles/word_count.py
```

See Session Mode for more details.

Note: Same as the per-job mode, the option **-pyclientexec** should point to a path on the client node (node executing the above command) as the job is compiled at the client side in per-job mode. If it's not specified, it will use the Python environment of the current shell environment.

1.1.1.5 Kubernetes

Kubernetes is a popular container-orchestration system for automating computer application deployment, scaling, and management. This page shows you how to set up Python environment and exeucte PyFlink jobs in a Kubernetes cluster.

Build PyFlink Image

It requires Python 3.6 or above with PyFlink pre-installed to be available in the docker container. It's suggested to use Python virtual environments to set up the Python environment. See *Create a Python virtual environment* for more details on how to prepare Python virtual environments with PyFlink installed.

You need install Python environments in the docker image with PyFlink pre-installed in advance.

To build a custom image which has Python and PyFlink prepared, you can refer to the following Dockerfile:

```
FROM flink:1.15.2

# install python3: it has updated Python to 3.9 in Debian 11 and so install Python 3.7␣
↪from source
# it currently only supports Python 3.6, 3.7 and 3.8 in PyFlink officially.

RUN apt-get update -y && \
apt-get install -y build-essential libssl-dev zlib1g-dev libbz2-dev libffi-dev && \
wget https://www.python.org/ftp/python/3.7.9/Python-3.7.9.tgz && \
tar -xvf Python-3.7.9.tgz && \
cd Python-3.7.9 && \
./configure --without-tests --enable-shared && \
make -j6 && \
make install && \
ldconfig /usr/local/lib && \
cd .. && rm -f Python-3.7.9.tgz && rm -rf Python-3.7.9 && \
ln -s /usr/local/bin/python3 /usr/local/bin/python && \
apt-get clean && \
rm -rf /var/lib/apt/lists/*

# install PyFlink
RUN pip3 install apache-flink==1.15.2
```

Execute PyFlink jobs in application mode with Native Kubernetes

You could execute PyFlink jobs in application mode as following:

This is useful when there is already a Kubernetes cluster available and you want to execute each Flink job in a separate Flink cluster. Flink is responsible for talking with Kubernetes and allocating and de-allocating TaskManagers depending on the required resources.

```
./bin/flink run-application \
      --target kubernetes-application \
      --parallelism 8 \
      -Dkubernetes.cluster-id=<ClusterId> \
      -Dtaskmanager.memory.process.size=4096m \
      -Dkubernetes.taskmanager.cpu=2 \
      -Dtaskmanager.numberOfTaskSlots=4 \
```

(continues on next page)

(continued from previous page)

```
    -Dkubernetes.container.image=<PyFlinkImageName> \
    --pyModule word_count \
    --pyFiles /opt/flink/examples/python/table/word_count.py
```

Execute PyFlink jobs in session mode with Native Kubernetes

You could also starting a Flink session cluster on Kubernetes and then submit PyFlink jobs to the session cluster.

The session cluster could be started as following:

```
./bin/kubernetes-session.sh -Dkubernetes.cluster-id=my-first-flink-cluster
```

Then you could submit PyFlink jobs to the session cluster as following:

```
./bin/flink run \
    --target kubernetes-session \
    -Dkubernetes.cluster-id=my-first-flink-cluster \
    -pyarch /path/to/venv.zip \
    -pyexec venv.zip/venv/bin/python3
    -py word_count.py
```

Note: Option **-pyclientexec** could be used to specify a local Python executable as the job will be compiled at the client side. Otherwise, if it's not specified, it will use the Python environment of the current shell environment.

See Session Mode for more details about session mode of Kubernetes.

Execute PyFlink jobs with Flink Kubernetes Operator

See PyFlink Example for more details on how to execute PyFlink jobs with Flink Kubernetes Operator.

1.1.2 QuickStart

1.1.2.1 QuickStart: Table API

This document is a short introduction to the PyFlink Table API, which is used to help novice users quickly understand the basic usage of PyFlink Table API.

You can run the latest version of these examples by yourself in 'Live Notebook: Table' at the quickstart page.

For advanced usage, you can refer to the latest version of PyFlink Table API doc

TableEnvironment Creation

`TableEnvironment` is the entry point and central context for creating Table and SQL API programs. Flink is an unified streaming and batch computing engine, which provides unified streaming and batch API to create a `TableEnvironment`. `TableEnvironment` is responsible for:

- `Table` management: `Table` Creation, listing `Tables`, Conversion between `Table` and `DataStream`, etc.
- User-defined function management: User-defined function registration, dropping, listing, etc.
- Executing SQL queries
- Job configuration
- Python dependency management
- Job submission

For more details of how to create a `TableEnvironment`, you can refer to the latest version Create a TableEnvironment

[1]:
```
# Create a batch TableEnvironment
from pyflink.table import EnvironmentSettings, TableEnvironment

env_settings = EnvironmentSettings.in_batch_mode()
table_env = TableEnvironment.create(env_settings)
table_env
```

[1]:
```
<pyflink.table.table_environment.TableEnvironment at 0x7fcd16342ac8>
```

[2]:
```
# Create a streaming TableEnvironment
env_settings = EnvironmentSettings.in_streaming_mode()
table_env = TableEnvironment.create(env_settings)
table_env
```

[2]:
```
<pyflink.table.table_environment.TableEnvironment at 0x7fcd1ad0c0f0>
```

Table Creation

`Table` is a core component of the Python Table API. A `Table` object describes a pipeline of data transformations. It does not contain the data itself in any way. Instead, it describes how to read data from a table source, how to add some compute on data and how to eventually write data to a table sink. The declared pipeline can be printed, optimized, and eventually executed in a cluster. The pipeline can work with bounded or unbounded streams which enables both streaming and batch scenarios.

A `Table` is always bound to a specific `TableEnvironment`. It is not possible to combine tables from different `TableEnvironments` in same query, e.g., to join or union them.

Firstly, you can create a `Table` from a Python `List` Object

[3]:
```
table = table_env.from_elements([(1, 'Hi'), (2, 'Hello')])
table.get_schema()
```

[3]:
```
root
 |-- _1: BIGINT
 |-- _2: STRING
```

Create a `Table` with an explicit schema.

[4]:
```
from pyflink.table import DataTypes
table = table_env.from_elements([(1, 'Hi'), (2, 'Hello')],
                                DataTypes.ROW([DataTypes.FIELD("id", DataTypes.
↪TINYINT()),
                                               DataTypes.FIELD("data", DataTypes.
↪STRING())]))
table.get_schema()
```

[4]:
```
root
 |-- id: TINYINT
 |-- data: STRING
```

Create a Table from a Pandas DataFrame

[5]:
```
import pandas as pd
df = pd.DataFrame({'id': [1, 2], 'data': ['Hi', 'Hello']})
table = table_env.from_pandas(df)
table.get_schema()
```

```
/Users/duanchen/sourcecode/flink/flink-python/dev/.conda/lib/python3.7/site-packages/
↪pyflink/table/utils.py:55: FutureWarning: Schema passed to names= option, please pass␣
↪schema= explicitly. Will raise exception in future
  return pa.RecordBatch.from_arrays(arrays, schema)
```

[5]:
```
root
 |-- id: BIGINT
 |-- data: STRING
```

Create a Table from DDL statements

[6]:
```
table_env.execute_sql("""
    CREATE TABLE random_source (
        id TINYINT,
        data STRING
    ) WITH (
        'connector' = 'datagen',
        'fields.id.kind' = 'sequence',
        'fields.id.start' = '1',
        'fields.id.end' = '2',
        'fields.data.kind' = 'random'
    )
""")
table = table_env.from_path("random_source")
table.get_schema()
```

[6]:
```
root
 |-- id: TINYINT
 |-- data: STRING
```

Create a Table from TableDescriptor

[7]:
```
from pyflink.table import DataTypes
from pyflink.table.schema import Schema
from pyflink.table.table_descriptor import TableDescriptor
```

(continues on next page)

(continued from previous page)

```
schema = (Schema.new_builder()
          .column('id', DataTypes.TINYINT())
          .column('data', DataTypes.STRING())
          .build())

table = table_env.from_descriptor(
    TableDescriptor
        .for_connector('datagen')
        .option('fields.id.kind', 'sequence')
        .option('fields.id.start', '1')
        .option('fields.id.end', '2')
        .option('fields.data.kind', 'random')
        .schema(schema)
        .build())
table.get_schema()
```

[7]:
```
root
 |-- id: TINYINT
 |-- data: STRING
```

Create a Table from a DataStream

[8]:
```
from pyflink.common import Types
from pyflink.datastream import StreamExecutionEnvironment
from pyflink.table import StreamTableEnvironment

# create a StreamExecutionEnvironment which is the entry point of `DataStream` program.
env = StreamExecutionEnvironment.get_execution_environment()
t_env = StreamTableEnvironment.create(env)

ds = env.from_collection([(1, 'Hi'), (2, 'Hello')],
                         type_info=Types.ROW_NAMED(
                             ["id", "data"],
                             [Types.BYTE(), Types.STRING()]))

table = t_env.from_data_stream(ds,
                               Schema.new_builder()
                               .column("id", DataTypes.TINYINT())
                               .column("data", DataTypes.STRING())
                               .build())
table.get_schema()
```

[8]:
```
root
 |-- id: TINYINT
 |-- data: STRING
```

Create a Table from Catalog

[9]:
```
# prepare the catalog
# register Table API tables in the catalog
old_table = table_env.from_elements([(1, 'Hi'), (2, 'Hello')], ['id', 'data'])
table_env.create_temporary_view('source_table', old_table)
```

(continues on next page)

(continued from previous page)

```
# create Table API table from catalog
table = table_env.from_path('source_table')
table.get_schema()
```

[9]:
```
root
 |-- id: BIGINT
 |-- data: STRING
```

Viewing Data on Table

You can get the schema of `Table` as follows:

[10]:
```
table.get_schema()
```

[10]:
```
root
 |-- id: BIGINT
 |-- data: STRING
```

[11]:
```
table.print_schema()
```

```
(
  `id` BIGINT,
  `data` STRING
)
```

`Table.execute()` collects the contents of the current `Table` to local client.

[12]:
```
list(table.execute().collect())
```

[12]:
```
[<Row(1, 'Hi')>, <Row(2, 'Hello')>]
```

[13]:
```
table.execute().print()
```

```
+----+----------------------+--------------------------------+
| op |                   id |                           data |
+----+----------------------+--------------------------------+
| +I |                    1 |                             Hi |
| +I |                    2 |                          Hello |
+----+----------------------+--------------------------------+
2 rows in set
```

PyFlink Table also provides the conversion back to a pandas DataFrame to leverage pandas API.

[14]:
```
table.to_pandas()
```

[14]:
```
   id   data
0   1     Hi
1   2  Hello
```

Selecting and Accessing Data on Table

PyFlink Table is lazily evaluated and simply selecting a column does not trigger the computation but it returns a Column `Expression` instance.

[15]:
```
from pyflink.table.expressions import col
type(table.id)==type(col('id'))
```

[15]:
```
True
```

These Column `Expressions` can be used to select the columns from a `Table`. For example, `Table.select()` takes the column `Expression` instances that returns another `Table`.

[16]:
```
table.select(table.id).to_pandas()
```

[16]:
```
   id
0   1
1   2
```

[17]:
```
table.select(col('id')).to_pandas()
```

[17]:
```
   id
0   1
1   2
```

Assign new Column `Expression` instance.

[18]:
```
table.add_columns(col('data').upper_case.alias('upper_data')).to_pandas()
```

[18]:
```
   id   data upper_data
0   1     Hi         HI
1   2  Hello      HELLO
```

To select a subset of rows, use `Table.filter()`.

[19]:
```
table.filter(col('id') == 1).to_pandas()
```

[19]:
```
   id data
0   1   Hi
```

Applying a Function on Table

PyFlink supports various UDFs and APIs to allow users to execute Python native functions. See also the latest User-defined Functions and Row-based Operations.

The first example is UDFs used in `Table API & SQL`

[20]:
```
from pyflink.table.udf import udf

# create a general Python UDF
@udf(result_type=DataTypes.BIGINT())
def plus_one(i):
  return i + 1

table.select(plus_one(col('id'))).to_pandas()
```

[20]:
```
   _c0
0    2
1    3
```

[21]:
```
# create a general Python UDF
@udf(result_type=DataTypes.BIGINT(), func_type='pandas')
def pandas_plus_one(series):
  return series + 1
table.select(pandas_plus_one(col('id'))).to_pandas()
```

```
/Users/duanchen/sourcecode/flink/flink-python/dev/.conda/lib/python3.7/site-packages/
↪pyflink/table/utils.py:55: FutureWarning: Schema passed to names= option, please pass␣
↪schema= explicitly. Will raise exception in future
  return pa.RecordBatch.from_arrays(arrays, schema)
```

[21]:
```
   _c0
0    2
1    3
```

[ ]:
```
# use the Python function in SQL API
table_env.create_temporary_function("plus_one", plus_one)
table_env.sql_query("SELECT plus_one(id) FROM {}".format(table)).to_pandas()
```

Another example is UDFs used in `Row-based Operations`

[23]:
```
from pyflink.common.types import Row
@udf(result_type=DataTypes.ROW([DataTypes.FIELD("id", DataTypes.BIGINT()),
                                DataTypes.FIELD("data", DataTypes.STRING())]))
def func(data: Row):
  return Row(data.id, data.data * 2)
table.map(func).execute().print()
```

```
+----+----------------------+--------------------------------+
| op |                   id |                           data |
+----+----------------------+--------------------------------+
| +I |                    1 |                           HiHi |
| +I |                    2 |                     HelloHello |
+----+----------------------+--------------------------------+
2 rows in set
```

Emits Results of Table

There are many connectors and formats available in Flink. See also the latest Table & SQL Connectors.

[24]:
```
# create a `Print` connector
schema = (Schema.new_builder()
          .column('id', DataTypes.BIGINT())
          .column('data', DataTypes.STRING())
          .build())

table.execute_insert(
    TableDescriptor
```

(continues on next page)

(continued from previous page)

```
        .for_connector('print')
        .schema(schema)
        .build())
```

```
2> +I[1, Hi]
2> +I[2, Hello]
```

[24]:
```
<pyflink.table.table_result.TableResult at 0x7fcd1ba83be0>
```

1.1.2.2 QuickStart: DataStream API

Apache Flink offers a DataStream API for building robust, stateful streaming applications. It provides fine-grained control over state and timer, which allows for the implementation of advanced event-driven systems.

You can run the latest version of these examples by yourself in 'Live Notebook: DataStream' at the quickstart page.

For advanced usage, you can refer to the latest version of PyFlink DataStream API doc

StreamExecutionEnvironment Creation

`StreamExecutionEnvironment` is the entry point and central concept for creating DataStream API programs. Flink is an unified streaming and batch computing engine, which provides unified streaming and batch API to create a `StreamExecutionEnvironment`.

`StreamExecutionEnvironment` is responsible for:

- `DataStream` Creation
- Python dependency management
- Job configuration
- Job submission

[1]:
```
from pyflink.datastream import StreamExecutionEnvironment
from pyflink.datastream import RuntimeExecutionMode

env = StreamExecutionEnvironment.get_execution_environment()

# Config the Program run in Streaming Mode
env.set_runtime_mode(RuntimeExecutionMode.STREAMING)
env
```

[1]:
```
<pyflink.datastream.stream_execution_environment.StreamExecutionEnvironment at␣
↪0x7fd9e8fc6e48>
```

DataStream Creation

DataStream is a core component of the Python DataStream API. A DataStream object describes a pipeline of data transformations. It does not contain the data itself in any way. Instead, it describes how to read data from a source, how to add some compute on data and how to eventually write data to a sink. The declared pipeline can be printed, optimized, and eventually executed in a cluster. The pipeline can work with bounded or unbounded streams which enables both streaming and batch scenarios.

A DataStream can be created by a specific StreamExecutionEnvironment.

Firstly, you can create a DataStream from a Python List Object

[2]:
```
from pyflink.common.typeinfo import Types
ds = env.from_collection([(1, 'aaa|bb'), (2, 'bb|a'), (3, 'aaa|a')])
# if you don't specify the `type_info`, the default `type_info` is␣
↪`PickledByteArrayTypeInfo`
ds.get_type()
```

[2]:
```
PickledByteArrayTypeInfo
```

Create a DataStream with an explicit type_info.

[3]:
```
ds = env.from_collection(
    collection=[(1, 'aaa|bb'), (2, 'bb|a'), (3, 'aaa|a')],
    type_info=Types.ROW([Types.INT(), Types.STRING()]))
ds.get_type()
```

[3]:
```
RowTypeInfo(f0: Integer, f1: String)
```

Create a DataStream from DataStream Connectors

[4]:
```
from pyflink.common.watermark_strategy import WatermarkStrategy
from pyflink.datastream.connectors.number_seq import NumberSequenceSource

env = StreamExecutionEnvironment.get_execution_environment()
seq_num_source = NumberSequenceSource(1, 1000)
ds = env.from_source(
    source=seq_num_source,
    watermark_strategy=WatermarkStrategy.for_monotonous_timestamps(),
    source_name='seq_num_source',
    type_info=Types.LONG())
ds.get_type()
```

[4]:
```
Long
```

Create a DataStream from a Table

[5]:
```
from pyflink.table import DataTypes
from pyflink.table import StreamTableEnvironment

# create a `TableEnvironment` which is the entry point of `Table` & `SQL` program.
t_env = StreamTableEnvironment.create(env)
table = t_env.from_elements([(1, 'aaa|bb'), (2, 'bb|a'), (3, 'aaa|a')],
                            DataTypes.ROW([DataTypes.FIELD("id", DataTypes.INT()),
                                           DataTypes.FIELD("data", DataTypes.STRING())]))
ds = t_env.to_data_stream(table)
ds.get_type()
```

```
[5]: ExternalTypeInfo<RowTypeInfo(id: Integer, data: String)>
```

Viewing Data on DataStream

`DataStream.execute_and_collect()` collects the contents of the current `DataStream` to local client.

```
[6]: list(ds.execute_and_collect())
```

```
[6]: [<Row(1, 'aaa|bb')>, <Row(2, 'bb|a')>, <Row(3, 'aaa|a')>]
```

Print the data of `DataStream` to the console

```
[7]: ds.print()
env.execute()
```

```
[7]: <pyflink.common.job_execution_result.JobExecutionResult at 0x7fd5dcc73550>
```

Applying a Function on DataStream

`DataStream` programs in Flink are regular programs that implement transformations on data streams (e.g., mapping, filtering, reducing). Please see operators for an overview of the available transformations in Python DataStream API.

```
[7]: from pyflink.common import Row
from pyflink.datastream import FlatMapFunction

class MyFlatMapFunction(FlatMapFunction):
    def flat_map(self, value):
        for s in str(value.data).split('|'):
            yield Row(value.id, s)

list(ds.flat_map(MyFlatMapFunction(), output_type=Types.ROW([Types.INT(), Types.
↪STRING()])).execute_and_collect())
```

```
[7]: [<Row(1, 'aaa')>,
 <Row(1, 'bb')>,
 <Row(2, 'bb')>,
 <Row(2, 'a')>,
 <Row(3, 'aaa')>,
 <Row(3, 'a')>]
```

Emits Results of DataStream

There are many connectors and formats available in Flink. See also the latest DataStream Connectors.

```
[8]: from pyflink.common import Encoder
from pyflink.datastream.connectors.file_system import FileSink, RollingPolicy

def split(s):
    splits = s[1].split('|')
    for sp in splits:
```

(continues on next page)

(continued from previous page)

```
        yield s[0], sp

sink = (FileSink
    .for_row_format('/tmp/sink', Encoder.simple_string_encoder("UTF-8"))
    .with_rolling_policy(RollingPolicy.default_rolling_policy(
        part_size=1024 ** 3, rollover_interval=15 * 60 * 1000, inactivity_interval=5 *␣
↪60 * 1000))
    .build())

ds.map(lambda i: (i[0] + 1, i[1]), Types.TUPLE([Types.INT(), Types.STRING()])).sink_
↪to(sink)
# the result will be stored in the directory of /tmp/sink.
env.execute()
```

[8]:
```
<pyflink.common.job_execution_result.JobExecutionResult at 0x7fd9ececa080>
```

1.2 User Guide

1.2.1 RealTime Feature

1.2.1.1 Coming Soon.

1.2.2 PyFlink + Flink ML

1.2.2.1 Coming Soon.

1.3 Frequently Asked Questions (FAQ)

1.3.1 Installation issues

1.3.1.1 O1: Scala Dependency

PyFlink only provides official installation packages which contain JAR packages for Scala 2.11 before Flink 1.15 , Scala 2.12 in Flink 1.15 and without Scala dependency since Flink 1.16. If you want to use Scala 2.12, you can download the binary distribution of Scala 2.12, unzip it and then set the environment variable *FLINK_HOME* to point to the unzipped directory. This makes it use the JAR packages specified by FLINK_HOME instead of the JAR packages under PyFlink installation package. You can refer to PyFlink documentation for more details.

1.3.1.2 O2: Java gateway process exited before sending its port number

The exception stack is as following:

```
Traceback (most recent call last):
  File "/Users/dianfu/code/src/github/pyflink-faq/testing/test_utils.py", line 122, in␣
↪setUp
    self.t_env = TableEnvironment.create(EnvironmentSettings.in_streaming_mode())
  File "/Users/dianfu/code/src/github/pyflink-faq/testing/.venv/lib/python3.8/site-
```

(continues on next page)

(continued from previous page)

```
↪packages/apache_flink-1.14.4-py3.8-macosx-10.9-x86_64.egg/pyflink/table/environment_
↪settings.py", line 267, in in_streaming_mode
    get_gateway().jvm.EnvironmentSettings.inStreamingMode())
  File "/Users/dianfu/code/src/github/pyflink-faq/testing/.venv/lib/python3.8/site-
↪packages/apache_flink-1.14.4-py3.8-macosx-10.9-x86_64.egg/pyflink/java_gateway.py",␣
↪line 62, in get_gateway
    _gateway = launch_gateway()
  File "/Users/dianfu/code/src/github/pyflink-faq/testing/.venv/lib/python3.8/site-
↪packages/apache_flink-1.14.4-py3.8-macosx-10.9-x86_64.egg/pyflink/java_gateway.py",␣
↪line 112, in launch_gateway
    raise Exception("Java gateway process exited before sending its port number")
Exception: Java gateway process exited before sending its port number
```

This issue is usually caused by the reason that PyFlink isn't installed correctly. You can verify whether PyFlink is installed correctly as following:

Execute the following command:

```
python -c "import pyflink;import os;print(os.path.dirname(os.path.abspath(pyflink.__file_
↪_)))"
```

It will print something like the following:

```
/Users/duanchen/miniconda3/lib/python3.7/site-packages/pyflink
```

Execute the following command:

```
ls -lh /Users/duanchen/miniconda3/lib/python3.7/site-packages/pyflink
```

The structure would be as following:

```
total 144
-rw-r--r--   1 duanchen  staff   1.3K Oct 19 16:01 README.txt
-rw-r--r--   1 duanchen  staff   1.9K Oct 19 16:01 __init__.py
drwxr-xr-x  11 duanchen  staff   352B Oct 19 16:03 __pycache__
drwxr-xr-x  25 duanchen  staff   800B Oct 19 16:03 bin
drwxr-xr-x  22 duanchen  staff   704B Oct 19 16:03 common
drwxr-xr-x  13 duanchen  staff   416B Oct 19 16:03 conf
drwxr-xr-x  20 duanchen  staff   640B Oct 19 16:03 datastream
drwxr-xr-x   4 duanchen  staff   128B Oct 19 16:03 examples
-rw-r--r--   1 duanchen  staff   3.2K Oct 19 16:01 find_flink_home.py
drwxr-xr-x  25 duanchen  staff   800B Oct 19 16:03 fn_execution
-rw-r--r--   1 duanchen  staff   9.1K Oct 19 16:01 gen_protos.py
-rw-r--r--   1 duanchen  staff   7.7K Oct 19 16:01 java_gateway.py
drwxr-xr-x  16 duanchen  staff   512B Oct 19 16:03 lib
drwxr-xr-x  26 duanchen  staff   832B Oct 19 16:03 licenses
drwxr-xr-x   4 duanchen  staff   128B Oct 19 16:04 log
drwxr-xr-x   5 duanchen  staff   160B Oct 19 16:03 metrics
drwxr-xr-x   4 duanchen  staff   128B Oct 19 16:03 opt
drwxr-xr-x  11 duanchen  staff   352B Oct 19 16:03 plugins
-rw-r--r--   1 duanchen  staff   1.3K Oct 19 16:01 pyflink_callback_server.py
-rw-r--r--   1 duanchen  staff    13K Oct 19 16:01 pyflink_gateway_server.py
-rw-r--r--   1 duanchen  staff   5.3K Oct 19 16:01 serializers.py
```

(continues on next page)

(continued from previous page)

```
-rw-r--r--   1 duanchen  staff   7.9K Oct 19 16:01 shell.py
drwxr-xr-x  31 duanchen  staff   992B Oct 19 16:03 table
drwxr-xr-x   6 duanchen  staff   192B Oct 19 16:03 util
-rw-r--r--   1 duanchen  staff   1.1K Oct 19 16:01 version.py
```

Please check whether the directories *lib*, *opt* are available. Besides, the jar packages under these directories should be as following:

```
(base)   pyflink ls -lh /Users/duanchen/miniconda3/lib/python3.7/site-packages/pyflink/
↪lib
total 401216
-rw-r--r--  1 duanchen  staff   190K Oct 19 16:02 flink-cep-1.15.2.jar
-rw-r--r--  1 duanchen  staff   475K Oct 19 16:02 flink-connector-files-1.15.2.jar
-rw-r--r--  1 duanchen  staff    93K Oct 19 16:02 flink-csv-1.15.2.jar
-rw-r--r--  1 duanchen  staff   110M Oct 19 16:02 flink-dist-1.15.2.jar
-rw-r--r--  1 duanchen  staff   171K Oct 19 16:02 flink-json-1.15.2.jar
-rw-r--r--  1 duanchen  staff    20M Oct 19 16:02 flink-scala_2.12-1.15.2.jar
-rw-r--r--  1 duanchen  staff    10M Oct 19 16:02 flink-shaded-zookeeper-3.5.9.jar
-rw-r--r--  1 duanchen  staff    15M Oct 19 16:02 flink-table-api-java-uber-1.15.2.jar
-rw-r--r--  1 duanchen  staff    35M Oct 19 16:02 flink-table-planner-loader-1.15.2.jar
-rw-r--r--  1 duanchen  staff   2.9M Oct 19 16:02 flink-table-runtime-1.15.2.jar
-rw-r--r--  1 duanchen  staff   203K Oct 19 16:02 log4j-1.2-api-2.17.1.jar
-rw-r--r--  1 duanchen  staff   295K Oct 19 16:02 log4j-api-2.17.1.jar
-rw-r--r--  1 duanchen  staff   1.7M Oct 19 16:02 log4j-core-2.17.1.jar
-rw-r--r--  1 duanchen  staff    24K Oct 19 16:02 log4j-slf4j-impl-2.17.1.jar

(base)   pyflink ls -lh /Users/duanchen/miniconda3/lib/python3.7/site-packages/pyflink/
↪opt
total 76736
-rw-r--r--  1 duanchen  staff    37M Oct 19 16:02 flink-python_2.12-1.15.2.jar
-rw-r--r--  1 duanchen  staff   472K Oct 19 16:02 flink-sql-client-1.15.2.jar
```

1.3.2 Usage issues

1.3.2.1 Q1: How to prepare Python Virtual Environment

You can execute the following script to prepare a Python virtual env zip which can be used on Mac OS and most Linux distributions.

```
set -e
# download miniconda.sh
if [[ `uname -s` == "Darwin" ]]; then
    wget "https://repo.continuum.io/miniconda/Miniconda3-4.7.10-MacOSX-x86_64.sh" -O
↪"miniconda.sh"
else
    wget "https://repo.continuum.io/miniconda/Miniconda3-4.7.10-Linux-x86_64.sh" -O
↪"miniconda.sh"
fi

# add the execution permission
chmod +x miniconda.sh
```

(continues on next page)

(continued from previous page)

```

# create python virtual environment
./miniconda.sh -b -p venv

# activate the conda python virtual environment
source venv/bin/activate ""

# specify your apache-flink version and you can add other dependencies
pip install "apache-flink==$1"

# deactivate the conda python virtual environment
conda deactivate

# remove the cached packages
rm -rf venv/pkgs

# package the prepared conda python virtual environment
zip -r venv.zip venv
```

1.3.2.2 O2: How to add Python Files

You can use the command-line arguments pyfs

```
$ ./bin/flink run --python examples/python/table/word_count.py --pyFiles file:///user.
↪txt,hdfs:///$namenode_address/username.txt
```

For example, if you have a directory named *myDir* which has the following hierarchy:

```
myDir
├──utils
    ├──__init__.py
    ├──my_util.py
```

You can add the Python files of directory *myDir* as following:

```
table_env.add_python_file('myDir')

def my_udf():
    from utils import my_util
```

1.3.3 JDK issues

1.3.3.1 O1: InaccessibleObjectException: Unable to make field private final byte[] java.lang.String.value accessible: module java.base does not "opens java.lang" to unnamed module @4e4aea35

```
: java.lang.reflect.InaccessibleObjectException: Unable to make field private final␣
↪byte[] java.lang.String.value accessible: module java.base does not "opens java.lang"␣
↪to unnamed module @4e4aea35
```

(continues on next page)

(continued from previous page)

```
    at java.base/java.lang.reflect.AccessibleObject.
↪checkCanSetAccessible(AccessibleObject.java:354)
    at java.base/java.lang.reflect.AccessibleObject.
↪checkCanSetAccessible(AccessibleObject.java:297)
    at java.base/java.lang.reflect.Field.checkCanSetAccessible(Field.java:178)
    at java.base/java.lang.reflect.Field.setAccessible(Field.java:172)
    at org.apache.flink.api.java.ClosureCleaner.clean(ClosureCleaner.java:106)
    at org.apache.flink.api.java.ClosureCleaner.clean(ClosureCleaner.java:132)
    at org.apache.flink.api.java.ClosureCleaner.clean(ClosureCleaner.java:132)
    at org.apache.flink.api.java.ClosureCleaner.clean(ClosureCleaner.java:132)
    at org.apache.flink.api.java.ClosureCleaner.clean(ClosureCleaner.java:132)
    at org.apache.flink.api.java.ClosureCleaner.clean(ClosureCleaner.java:69)
    at org.apache.flink.streaming.api.environment.StreamExecutionEnvironment.
↪clean(StreamExecutionEnvironment.java:2138)
    at org.apache.flink.table.planner.plan.nodes.exec.common.CommonExecSink.
↪createSinkFunctionTransformation(CommonExecSink.java:331)
    at org.apache.flink.table.planner.plan.nodes.exec.common.CommonExecSink.
↪applySinkProvider(CommonExecSink.java:306)
    at org.apache.flink.table.planner.plan.nodes.exec.common.CommonExecSink.
↪createSinkTransformation(CommonExecSink.java:146)
    at org.apache.flink.table.planner.plan.nodes.exec.stream.StreamExecSink.
↪translateToPlanInternal(StreamExecSink.java:140)
    at org.apache.flink.table.planner.plan.nodes.exec.ExecNodeBase.
↪translateToPlan(ExecNodeBase.java:134)
```

This is an issue around Java 17. It still doesn't support Java 17 in Flink. You can refer to FLINK-15736 for more details. To solve this issue, you need to use JDK 1.8 or JDK 11.

1.3.4 Connector issues

1.3.4.1 Q1: Could not find any factory for identifier 'xxx' that implements 'org.apache.flink.table.factories.DynamicTableFactory' in the classpath

```
Exception Stack:

py4j.protocol.Py4JJavaError: An error occurred while calling o13.execute.
: org.apache.flink.table.api.ValidationException: Unable to create a source for reading␣
↪table 'default_catalog.default_database.sourceKafka'.

Table options are:

'connector'='kafka'
'format'='json'
'properties.bootstrap.servers'='192.168.101.109:9092'
'scan.startup.mode'='earliest-offset'
'topic'='pyflink_test'
    at org.apache.flink.table.factories.FactoryUtil.createTableSource(FactoryUtil.java:
↪150)
    at org.apache.flink.table.planner.plan.schema.CatalogSourceTable.
↪createDynamicTableSource(CatalogSourceTable.java:116)
    at org.apache.flink.table.planner.plan.schema.CatalogSourceTable.
```

(continues on next page)

(continued from previous page)

```
↪toRel(CatalogSourceTable.java:82)
    at org.apache.calcite.rel.core.RelFactories$TableScanFactoryImpl.
↪createScan(RelFactories.java:495)
    at org.apache.calcite.tools.RelBuilder.scan(RelBuilder.java:1099)
    at org.apache.calcite.tools.RelBuilder.scan(RelBuilder.java:1123)
    at org.apache.flink.table.planner.plan.QueryOperationConverter$SingleRelVisitor.
↪visit(QueryOperationConverter.java:351)
    at org.apache.flink.table.planner.plan.QueryOperationConverter$SingleRelVisitor.
↪visit(QueryOperationConverter.java:154)
    at org.apache.flink.table.operations.CatalogQueryOperation.
↪accept(CatalogQueryOperation.java:68)
    at org.apache.flink.table.planner.plan.QueryOperationConverter.
↪defaultMethod(QueryOperationConverter.java:151)
    at org.apache.flink.table.planner.plan.QueryOperationConverter.
↪defaultMethod(QueryOperationConverter.java:133)
    at org.apache.flink.table.operations.utils.QueryOperationDefaultVisitor.
↪visit(QueryOperationDefaultVisitor.java:92)
    at org.apache.flink.table.operations.CatalogQueryOperation.
↪accept(CatalogQueryOperation.java:68)
    at org.apache.flink.table.planner.plan.QueryOperationConverter.lambda$defaultMethod
↪$0(QueryOperationConverter.java:150)
    at java.util.Collections$SingletonList.forEach(Collections.java:4824)
    at org.apache.flink.table.planner.plan.QueryOperationConverter.
↪defaultMethod(QueryOperationConverter.java:150)
    at org.apache.flink.table.planner.plan.QueryOperationConverter.
↪defaultMethod(QueryOperationConverter.java:133)
    at org.apache.flink.table.operations.utils.QueryOperationDefaultVisitor.
↪visit(QueryOperationDefaultVisitor.java:47)
    at org.apache.flink.table.operations.ProjectQueryOperation.
↪accept(ProjectQueryOperation.java:76)
    at org.apache.flink.table.planner.calcite.FlinkRelBuilder.
↪queryOperation(FlinkRelBuilder.scala:184)
    at org.apache.flink.table.planner.delegation.PlannerBase.translateToRel(PlannerBase.
↪scala:219)
    at org.apache.flink.table.planner.delegation.PlannerBase$$anonfun$1.
↪apply(PlannerBase.scala:182)
    at org.apache.flink.table.planner.delegation.PlannerBase$$anonfun$1.
↪apply(PlannerBase.scala:182)
    at scala.collection.TraversableLike$$anonfun$map$1.apply(TraversableLike.scala:234)
    at scala.collection.TraversableLike$$anonfun$map$1.apply(TraversableLike.scala:234)
    at scala.collection.Iterator$class.foreach(Iterator.scala:891)
    at scala.collection.AbstractIterator.foreach(Iterator.scala:1334)
    at scala.collection.IterableLike$class.foreach(IterableLike.scala:72)
    at scala.collection.AbstractIterable.foreach(Iterable.scala:54)
    at scala.collection.TraversableLike$class.map(TraversableLike.scala:234)
    at scala.collection.AbstractTraversable.map(Traversable.scala:104)
    at org.apache.flink.table.planner.delegation.PlannerBase.translate(PlannerBase.scala:
↪182)
    at org.apache.flink.table.api.internal.TableEnvironmentImpl.
↪translate(TableEnvironmentImpl.java:1665)
    at org.apache.flink.table.api.internal.TableEnvironmentImpl.
↪translateAndClearBuffer(TableEnvironmentImpl.java:1657)
```

(continues on next page)

(continued from previous page)

```
    at org.apache.flink.table.api.internal.TableEnvironmentImpl.
↪execute(TableEnvironmentImpl.java:1607)
    at sun.reflect.NativeMethodAccessorImpl.invoke0(Native Method)
    at sun.reflect.NativeMethodAccessorImpl.invoke(NativeMethodAccessorImpl.java:62)
    at sun.reflect.DelegatingMethodAccessorImpl.invoke(DelegatingMethodAccessorImpl.java:
↪43)
    at java.lang.reflect.Method.invoke(Method.java:498)
    at org.apache.flink.api.python.shaded.py4j.reflection.MethodInvoker.
↪invoke(MethodInvoker.java:244)
    at org.apache.flink.api.python.shaded.py4j.reflection.ReflectionEngine.
↪invoke(ReflectionEngine.java:357)
    at org.apache.flink.api.python.shaded.py4j.Gateway.invoke(Gateway.java:282)
    at org.apache.flink.api.python.shaded.py4j.commands.AbstractCommand.
↪invokeMethod(AbstractCommand.java:132)
    at org.apache.flink.api.python.shaded.py4j.commands.CallCommand.execute(CallCommand.
↪java:79)
    at org.apache.flink.api.python.shaded.py4j.GatewayConnection.run(GatewayConnection.
↪java:238)
    at java.lang.Thread.run(Thread.java:748)
Caused by: org.apache.flink.table.api.ValidationException: Cannot discover a connector␣
↪using option: 'connector'='kafka'
    at org.apache.flink.table.factories.FactoryUtil.
↪enrichNoMatchingConnectorError(FactoryUtil.java:587)
    at org.apache.flink.table.factories.FactoryUtil.getDynamicTableFactory(FactoryUtil.
↪java:561)
    at org.apache.flink.table.factories.FactoryUtil.createTableSource(FactoryUtil.java:
↪146)
    ... 45 more
Caused by: org.apache.flink.table.api.ValidationException: Could not find any factory␣
↪for identifier 'kafka' that implements 'org.apache.flink.table.factories.
↪DynamicTableFactory' in the classpath.

Available factory identifiers are:

blackhole
datagen
filesystem
print
    at org.apache.flink.table.factories.FactoryUtil.discoverFactory(FactoryUtil.java:399)
    at org.apache.flink.table.factories.FactoryUtil.
↪enrichNoMatchingConnectorError(FactoryUtil.java:583)
    ... 47 more
```

It reuses the Java connectors implementations in PyFlink and most connectors are not bundled in the official PyFlink (and also Flink) distribution except the following connectors: blackhole, datagen, filesystem and print. So you need to specify the connector JAR package explicitly when executing PyFlink jobs:

- The connector JAR package could be found in the corresponding connector page in the official Flink documentation. For example, you can open the Kafka connector page and search keyword "SQL Client JAR" which is a fat JAR of Kafka connector.
- It should be noted that you should use the fat JAR which contains all the dependencies. Besides, the version of the connector JAR should be consistent with PyFlink version.

- For how to specify the connector JAR in PyFlink jobs, you can refer to the dependency management page of official PyFlink documentation.

1.3.4.2 O2: ClassNotFoundException: com.mysql.cj.jdbc.Driver

```
py4j.protocol.Py4JJavaError: An error occurred while calling o13.execute.
: org.apache.flink.runtime.client.JobExecutionException: Job execution failed.
...
Caused by: java.io.IOException: unable to open JDBC writer
    at org.apache.flink.connector.jdbc.internal.JdbcOutputFormat.open(JdbcOutputFormat.
↪java:145)
    at org.apache.flink.connector.jdbc.internal.GenericJdbcSinkFunction.
↪open(GenericJdbcSinkFunction.java:52)
    at org.apache.flink.api.common.functions.util.FunctionUtils.
↪openFunction(FunctionUtils.java:34)
    at org.apache.flink.streaming.api.operators.AbstractUdfStreamOperator.
↪open(AbstractUdfStreamOperator.java:100)
    at org.apache.flink.streaming.api.operators.StreamSink.open(StreamSink.java:46)
    at org.apache.flink.streaming.runtime.tasks.RegularOperatorChain.
↪initializeStateAndOpenOperators(RegularOperatorChain.java:110)
    at org.apache.flink.streaming.runtime.tasks.StreamTask.restoreGates(StreamTask.java:
↪711)
    at org.apache.flink.streaming.runtime.tasks.StreamTaskActionExecutor$1.
↪call(StreamTaskActionExecutor.java:55)
    at org.apache.flink.streaming.runtime.tasks.StreamTask.restoreInternal(StreamTask.
↪java:687)
    at org.apache.flink.streaming.runtime.tasks.StreamTask.restore(StreamTask.java:654)
    at org.apache.flink.runtime.taskmanager.Task.runWithSystemExitMonitoring(Task.java:
↪958)
    at org.apache.flink.runtime.taskmanager.Task.restoreAndInvoke(Task.java:927)
    at org.apache.flink.runtime.taskmanager.Task.doRun(Task.java:766)
    at org.apache.flink.runtime.taskmanager.Task.run(Task.java:575)
    at java.lang.Thread.run(Thread.java:748)
Caused by: java.lang.ClassNotFoundException: com.mysql.cj.jdbc.Driver
    at java.net.URLClassLoader.findClass(URLClassLoader.java:382)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:418)
    at org.apache.flink.util.FlinkUserCodeClassLoader.
↪loadClassWithoutExceptionHandling(FlinkUserCodeClassLoader.java:64)
    at org.apache.flink.util.ChildFirstClassLoader.
↪loadClassWithoutExceptionHandling(ChildFirstClassLoader.java:74)
    at org.apache.flink.util.FlinkUserCodeClassLoader.loadClass(FlinkUserCodeClassLoader.
↪java:48)
    at java.lang.ClassLoader.loadClass(ClassLoader.java:351)
    at org.apache.flink.runtime.execution.librarycache.FlinkUserCodeClassLoaders
↪$SafetyNetWrapperClassLoader.loadClass(FlinkUserCodeClassLoaders.java:172)
    at java.lang.Class.forName0(Native Method)
    at java.lang.Class.forName(Class.java:348)
    at org.apache.flink.connector.jdbc.internal.connection.SimpleJdbcConnectionProvider.
↪loadDriver(SimpleJdbcConnectionProvider.java:90)
    at org.apache.flink.connector.jdbc.internal.connection.SimpleJdbcConnectionProvider.
↪getLoadedDriver(SimpleJdbcConnectionProvider.java:100)
    at org.apache.flink.connector.jdbc.internal.connection.SimpleJdbcConnectionProvider.
```

(continues on next page)

(continued from previous page)

```
↪getOrEstablishConnection(SimpleJdbcConnectionProvider.java:117)
    at org.apache.flink.connector.jdbc.internal.JdbcOutputFormat.open(JdbcOutputFormat.
↪java:143)
```

This indicates that it the JDBC driver JAR package is missing. It should be noted that the JDBC driver is also required when using JDBC connector. The JAR packages of the JDBC drivers could be found in the JDBC connector page.

1.3.4.3 O3: NoSuchMethodError: org.apache.flink.table.factories.DynamicTableFactory$Context.getCatalogTable()

```
java.lang.NoSuchMethodError: org.apache.flink.table.factories.DynamicTableFactory
↪$Context.getCatalogTable()Lorg/apache/flink/table/catalog/CatalogTable;
    at org.apache.flink.streaming.connectors.kafka.table.KafkaDynamicTableFactory.
↪createDynamicTableSource(KafkaDynamicTableFactory.java:145)
    at org.apache.flink.table.factories.FactoryUtil.createTableSource(FactoryUtil.java:
↪147)
    ... 39 more
```

1.3.5 Runtime issues

1.3.5.1 Q1: OverflowError: timeout value is too large

```
File "D:\Anaconda3\envs\py37\lib\threading.py", line 926, in _bootstrap_inner
    self.run()
File "D:\Anaconda3\envs\py37\lib\site-packages\apache_beam\runners\worker\data_plane.py",
↪ line 218, in run
    while not self._finished.wait(next_call - time.time()):
File "D:\Anaconda3\envs\py37\lib\threading.py", line 552, in wait
    signaled = self._cond.wait(timeout)
File "D:\Anaconda3\envs\py37\lib\threading.py", line 300, in wait
    gotit = waiter.acquire(True, timeout)
OverflowError: timeout value is too large
```

This exception only occurs on Windows. It doesn't affect the execution of PyFlink jobs and so you could ignore it usually. Besides, you could also upgrade PyFlink versions to 1.12.8, 1.13.7, 1.14.6, 1.15.2 or 1.16.0 where this issue has been fixed. You could refer to FLINK-25883 for more details.

1.3.5.2 Q2: An error occurred while calling z:org.apache.flink.client.python.PythonEnvUtils.resetCallbackClient

```
py4j.protocol.Py4jError: An error occurred while calling z:org.apache.flink.client.
↪python.PythonEnvUtils.resetCallbackClient. Trace:
org.apache.flink.api.python.shaded.py4j.Py4jException: Method resetCallbackClient([class␣
↪java.lang.String, class java.lang.Integer]) does not exist
    at org.apache.flink.api.python.shaded.py4j.reflection.ReflectionEngine.
↪getMethod(ReflectionEngine.java:318)
    ...
```

This exception only occurs when the version of the flink-python jar (located in site-packages/pyflink/opt) isn't consistent with PyFlink version. It usually happens when you have tried to install multiple PyFlink versions and something

wrong happens which make multiple versions mixed in your environment. You can try to reinstall PyFlink in a clean environment.

1.3.6 Data type issues

1.3.6.1 Q1: 'tuple' object has no attribute '_values'

```
Caused by: java.util.concurrent.ExecutionException: java.lang.RuntimeException: Error␣
↪received from SDK harness for instruction 4:
Traceback (most recent call last):
File "/usr/local/lib/python3.7/site-packages/apache_beam/runners/worker/sdk_worker.py",␣
↪line 289, in _execute    response = task()
File "/usr/local/lib/python3.7/site-packages/apache_beam/runners/worker/sdk_worker.py",␣
↪line 362, in <lambda>    lambda:
    self.create_worker().do_instruction(request), request)
File "/usr/local/lib/python3.7/site-packages/apache_beam/runners/worker/sdk_worker.py",␣
↪line 607, in do_instruction    getattr(request, request_type),
    request.instruction_id)
File "/usr/local/lib/python3.7/site-packages/apache_beam/runners/worker/sdk_worker.py",␣
↪line 644, in process_bundle
    bundle_processor.process_bundle(instruction_id))
File "/usr/local/lib/python3.7/site-packages/apache_beam/runners/worker/bundle_processor.
↪py", line 1000, in process_bundle    element.data)
File "/usr/local/lib/python3.7/site-packages/apache_beam/runners/worker/bundle_processor.
↪py", line 228, in process_encoded    self.output(decoded_value)  File "apache_beam/
↪runners/worker/operations.py", line 357, in apache_beam.runners.worker.operations.
↪Operation.output
File "apache_beam/runners/worker/operations.py", line 359, in apache_beam.runners.worker.
↪operations.Operation.output
File "apache_beam/runners/worker/operations.py", line 221, in apache_beam.runners.worker.
↪operations.SingletonConsumerSet.receive
File "pyflink/fn_execution/beam/beam_operations_fast.pyx", line 158, in pyflink.fn_
↪execution.beam.beam_operations_fast.FunctionOperation.process
File "pyflink/fn_execution/beam/beam_operations_fast.pyx", line 174, in pyflink.fn_
↪execution.beam.beam_operations_fast.FunctionOperation.process
File "pyflink/fn_execution/beam/beam_operations_fast.pyx", line 104, in
    pyflink.fn_execution.beam.beam_operations_fast.IntermediateOutputProcessor.process_
↪outputs
File "pyflink/fn_execution/beam/beam_operations_fast.pyx", line 158, in pyflink.fn_
↪execution.beam.beam_operations_fast.FunctionOperation.process
File "pyflink/fn_execution/beam/beam_operations_fast.pyx", line 174, in pyflink.fn_
↪execution.beam.beam_operations_fast.FunctionOperation.process
File "pyflink/fn_execution/beam/beam_operations_fast.pyx", line 92, in
    pyflink.fn_execution.beam.beam_operations_fast.NetworkOutputProcessor.process_outputs
File "pyflink/fn_execution/beam/beam_coder_impl_fast.pyx", line 101, in
    pyflink.fn_execution.beam.beam_coder_impl_fast.FlinkLengthPrefixCoderBeamWrapper.
↪encode_to_stream
File "pyflink/fn_execution/coder_impl_fast.pyx", line 271, in pyflink.fn_execution.coder_
↪impl_fast.IterableCoderImpl.encode_to_stream
File "pyflink/fn_execution/coder_impl_fast.pyx", line 399, in pyflink.fn_execution.coder_
↪impl_fast.RowCoderImpl.encode_to_stream
File "pyflink/fn_execution/coder_impl_fast.pyx", line 389, in pyflink.fn_execution.coder_
```

(continues on next page)

(continued from previous page)

```
↪impl_fast.RowCoderImpl.encode_to_streamAttributeError: 'tuple'
    object has no attribute '_values'
```

This issue is usually caused by the reason that it returns an object other than Row type in a Python user-defined function, however, the return type of the function is declared as Row. Please double check the return value of the Python user-defined function to make sure that the type of the returned value is consitent with the declartion.

1.3.6.2 Q2: AttributeError: 'int' object has no attribute 'encode'

```
File "pyflink/fn_execution/beam/beam_operations_fast.pyx", line 71, in pyflink.fn_
↪execution.beam.beam_operations_fast.FunctionOperation.process
File "pyflink/fn_execution/beam/beam_operations_fast.pyx", line 85, in pyflink.fn_
↪execution.beam.beam_operations_fast.FunctionOperation.process
File "pyflink/fn_execution/coder_impl_fast.pyx", line 83, in pyflink.fn_execution.coder_
↪impl_fast.TableFunctionRowCoderImpl.encode_to_stream
File "pyflink/fn_execution/coder_impl_fast.pyx", line 256, in pyflink.fn_execution.coder_
↪impl_fast.FlattenRowCoderImpl._encode_one_row
File "pyflink/fn_execution/coder_impl_fast.pyx", line 260, in pyflink.fn_execution.coder_
↪impl_fast.FlattenRowCoderImpl._encode_one_row_with_row_kind
File "pyflink/fn_execution/coder_impl_fast.pyx", line 244, in pyflink.fn_execution.coder_
↪impl_fast.FlattenRowCoderImpl._encode_one_row_to_buffer
File "pyflink/fn_execution/coder_impl_fast.pyx", line 550, in pyflink.fn_execution.coder_
↪impl_fast.FlattenRowCoderImpl._encode_field_simple
AttributeError: 'int' object has no attribute 'encode'
```

This reason to this issue is usually that the actual result value of a Python user-defined function isn't consistent with the declared result type of the Python user-defined function.

1.3.6.3 Q3: Types.BIG_INT() VS Types.LONG()

It should be noted that Types.BIG_INT() represents type information for the Java BigInteger, while Types.LONG() represents type information for long integer. There are several users are using Types.BIG_INT() for long integer by mistake.

1.4 API reference