Flink Python UDF Environment and Dependency Management

Wei Zhong, Dian Fu

Motivation

The FLIP-58 which aimed for supporting Python UDF is already accepted. As a critical part of python UDF, the environment and dependency management of users' python code is not supported yet. That means users can not introduce third-party libraries in their UDFs currently, which is not acceptable in most scenarios. It will be a great benefit to support environment and dependency management.

The implementation plan of FLIP 58 includes environment and dependency management but lacks a detailed description. This design document describes it in detail.

Goals

- Support uploading python libraries to clusters and importing them in UDFs in flink python API
- Support uploading whole python environment(usually generated by virtualenv, conda, pipenv and so on) to clusters and running UDFs on it in flink python API
- Support uploading normal files to clusters in flink python API
- Only support process mode in the first version

Proposed Changes

Public Interfaces

4 new public method will be added to Python TableEnvironment and one new public method will be added to Python TableConfig:

```
class TableEnvironment(object):

    ...

    def add_python_file(self, file_path):
        pass
```

```
    def set_python_requirements(self, requirements_list_file, requirements_cached_dir=None):
        pass

    def add_python_archive(self, archive_path, extract_name):
        pass

    def set_environment_variable(self, key, value):
        pass

class TableConfig(object):

    …

    def set_python_executable(self, exec_path):
        pass
```

add_python_file(self, file_path)

This interface is used to upload the python libraries that can be imported directly, e.g. single .py files, egg packages and some zip packages. The parent directories of .py files and the packaging files will be append to the PYTHONPATH of python workers so that they can be imported in UDF.

set_python_requirements(self, requirements_list_file, requirements_cached_dir=None)

This interface is used to upload the python libraries which need to be installed before importing.

To use this interface you should first prepare a "requirements.txt" file which is often generated by executing "pip freeze > requirements.txt" in users' python environment. If you write it yourself please make sure that it contains all transitive dependencies.

Then you can prepare a directory and download all the packages listed in the requirements.txt to the directory. This step is optional. A recommended approach is executing the following command:

```
pip download -d {cached_dir} -r {requirements_txt_path} --no-binary :all:
```

After that, call this interface with the path of requirements.txt and the cached dir if exists. Before running python worker, the packages listed in requirements.txt will be installed to the environment of python worker one by one according to their order in requirements.txt. If users specified the cached dir, it will be uploaded to cluster and the installation program will search packages in this directory instead of default repository(usually pypi.org), which can work in the environments without Internet.

This is a more complete example of this interface, suppose we need to install numpy on the cluster:

```
# command executed in shell:
echo numpy==1.16.5 > requirements.txt
pip download -d cached_dir -r requirements.txt --no-binary :all:

# python code:
t_env.set_python_requirements("requirements.txt", "cached_dir")
```

add_python_archive(self, archive_path, extract_name)

This interface is used to upload python environment and normal files to cluster. The parameter "archive_path" is the path of the zip file which contains the python environment or other files users want to upload and the parameter "extract_name" specifies the directory name to store extracted content of the zip file. The extracted directory will be moved to working directory of python worker, users could access their files using the relative path like "{extract_name}/xxxx".

If the uploaded zip file is a python environment, please make sure that the python executable file can run on the platform which the cluster is running on, and the path of python executable file must be specified by TableEnvironment#get_config().set_python_executable(), e.g.:

```
# Suppose the python executable file is py37/bin/python
# command executed in shell:
zip -r venv.zip py37

# python code:
t_env.add_python_archive("venv.zip", "my_venv")
t_env.get_config().set_python_executable("my_venv/py37/bin/python")
```

~~set_environment_variable(self, key, value)~~

~~This interface is used to set the environment variable of python worker. Python itself and many python libraries like pip and virtualenv can be configured using specific environment variables, so we should allow users to set the environment variable of python workers to cover more use cases of environment management.~~

set_python_executable(self, exec_path)

This interface is user to set the path of python executable file on cluster to run python workers. The parameter of this interface is a job-wide configuration, so put this interface to TableConfig.

New Options of PythonDriver

It is necessary to support managing dependencies and environment through command line so that the python jobs with additional dependencies can be submitted via "flink run" and web UI or other approached in the future. The PythonDriver class will support several new options as follows:

Short Name	Full Name	Syntax	Description
-pyarch	--pyArchives	-pyarch <archiveFile1>#<extractName>,<archiveFile2>#<extractName>	The option is equivalent to "add_python_archive". "," can be used as the separator for multiple archives and "#" can be used as the separator if "extractName" exists.
-pyexec	--pyExecutable	-pyexec <pythonInterpreterPath>	This option is equivalent to `TableEnvironment#get_config().set_python_executable()`.
-pyfs	--pyFiles	-pyfs <filePaths>	This option already exists but it only appends the file to client side PYTHONPATH currently. Now it will upload the file to cluster and append it to python worker's PYTHONPATH, which is equivalent to "add_python_file".
-pyreq	--pyRequirements	-pyreq <requirementsFile>#<requirementsCachedDir>	This option is equivalent to "set_python_requirements". "#" can be used to as the separator if "requirementsCachedDir" exists.

Implementation

Implementation of SDK API

Flink has provided a distributed cache mechanism and allows users to upload their files using "registerCachedFile" method in ExecutionEnvironment/StreamExecutionEnvironment. The python files users specified through "add_python_file", "set_python_requirements" and "add_python_archive" are also uploaded through this method eventually.

Besides the python files, additional information, such as file names of python files(Flink distributed cache will wipe the origin file names of uploaded files)~~, environment variables~~ and the target directory names of python archives, is also need to upload to cluster. These metadata will be serialized into json strings and stored in GlobalJobParameters object in ExecutionConfig, which can be propagated to every running tasks.

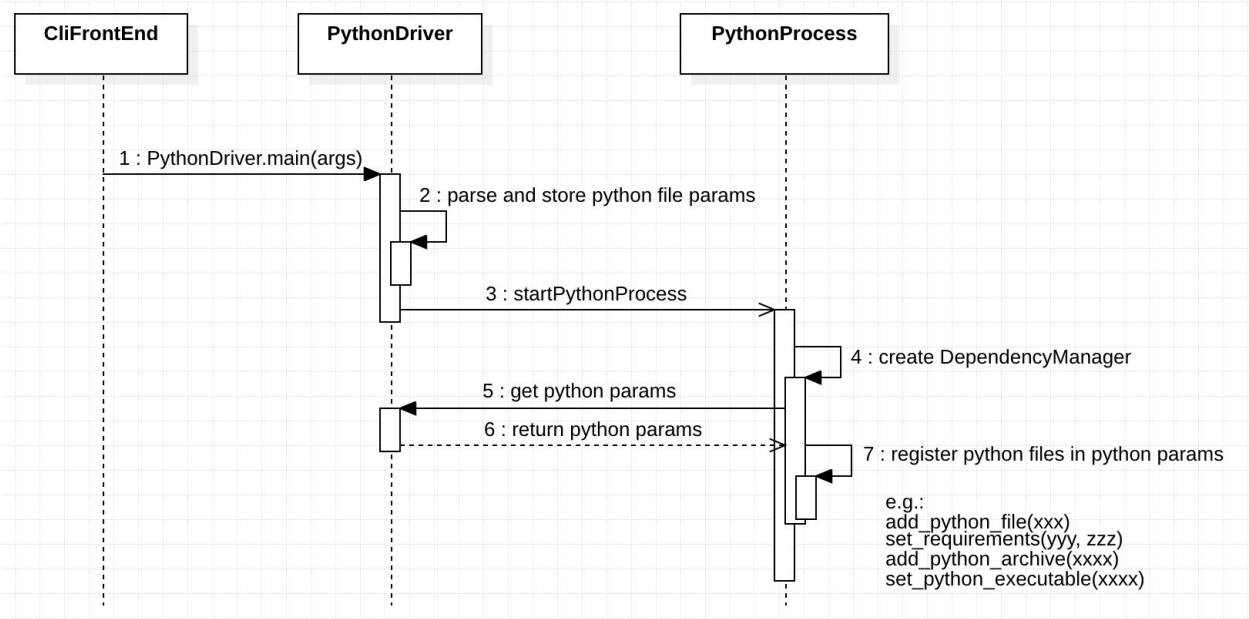
All logic of the environment and dependency management interfaces can be put in an independent class so that it can be easy to test. A simple architecture is as follows:

```
class DependencyManager(object):

    PYTHON_FILE_MAP = "PYTHON_FILE_MAP"
    PYTHON_REQUIREMENTS = "PYTHON_REQUIREMENTS"
    PYTHON_REQUIREMENTS_FILE = "PYTHON_REQUIREMENTS_FILE"
    PYTHON_ARCHIVES_MAP = "PYTHON_ARCHIVES_MAP"
    PYTHON_ENVIRONMENT_MAP = "PYTHON_ENVIRONMENT_MAP"

    def __init__(self):
        ...

    def add_python_file(self, file_path):
        ...

    def set_python_requirements(self, requirements_list_file, requirements_cached_dir):
        ...

    def add_python_archive(self, archive_path, extract_name):
        ...

    def set_environment_variable(self, key, value):
        ...

    def transmit_to_jvm(self, j_env, conf):
        conf.set_string(PYTHON_FILE_MAP, ….)
        conf.set_string(PYTHON_REQUIREMENTS, ….)
        …
        j_env.registerCachedFile(xxx, xxx)
        ...
```

```
class TableEnvironment(object):

    def __init__(self, ...):
        ....
        self._dependency_manager = DependencyManager()

    def add_python_file(self, file_path):
        self._dependency_manager.add_python_file(file_path)

    def set_python_requirements(self, requirements_list_file, requirements_cached_dir):
        self._dependency_manager.set_python_requirements(requirements_list_file, requirements_cached_dir)

    def add_python_archive(self, archive_path, extract_name):
        self._dependency_manager.add_python_archive(archive_path, extract_name)

    def set_environment_variable(self, key, value):
        self._dependency_manager.set_environment_variable(key, value)

    ...

    def execute(self, job_name):
        self._dependency_manager.transmit_to_jvm(self._j_tenv.execEnv(), self.get_config().get_configuration())
        self._j_tenv.execute(job_name)
```

Implementation of New PythonDriver Options

The PythonDriver will parse those new parameters and store them in a map. When the DependencyManager(see previous section) object creates, it will access the map and register the content of the map into itself. The sequence diagram is as follows:



Data Structures used in Operator

Two new roles will be introduced, named PythonDependencyManager and PythonEnvironmentManager separately.

PythonDependencyManager is used to parse the Python dependencies uploaded from client, and provide that information to PythonEnvironmentManager.
The structure of PythonDependencyManager is as follows:

```
public class PythonDependencyManager {

  // create PythonDependencyManager from ExecutionConfig.getGlobalJobParameters().toMap() and
  // distributedCaches.
  public static PythonDependencyManager create(
     Map<String, String> dependencyMetaData,
     DistributedCache distributedCache) {...}

  // key is the absolute path of the files to append to PYTHONPATH, value is the origin file name
  public Map<String, String> getPythonFiles() {...}

  // absolute path of requirements.txt
  public String getRequirementsFilePath() {...}

  // absolute path of the cached directory which contains user provided python packages
  public String getRequirementsDirPath() {...}

  //path of the python executable file
  public String getPythonExec() {...}

  // key is the name of the environment variable, value is the value of the environment variable
  public Map<String, String> getEnvironmentVariable() {...}

  // key is the absolute path of the zip file, value is the target directory name to be extracted to
  public Map<String, String> getArchives() {...}
}
```

PythonEnvironmentManager is used to manage the execution environment of python worker.
The structure of PythonEnvironmentManager is as follows:

```
public interface PythonEnvironmentManager {

  /**
   * Create Apache Beam Environment object of python worker.
   */
  RunnerApi.Environment createEnvironment();

  /**
   * Create the RetrievalToken file which records all the files that need to be transferred via Apache Beam's
   * ArtifactService.
   */
```

```
  String createRetrievalToken();

  /**
   * Delete generated files during above actions.
   */
  void cleanup();
}
```

Flink Python UDF is implemented based on Apache Beam Portability Framework which uses a RetrievalToken file to record the information of users' file. We will leverage the power of Apache Beam artifact staging for dependency management in docker mode.

PythonEnvironmentManager has two implementations, ProcessEnvironmentManager for process mode and DockerEnvironmentManager for docker mode.

Implementation of PythonEnvironmentManager

ProcessEnvironmentManager

The structure of ProcessEnvironmentManager is as follows:

```
public class ProcessEnvironmentManager implements PythonEnvironmentManager {

  public static ProcessEnvironmentManager create(
    PythonDependencyManager dependencyManager,
    String tmpDirectoryBase,
    Map<String, String> systemEnv) {

   }

  public ProcessEnvironmentManager(...) {
    prepareEnvironment();
  }

  @Override
  public void cleanup() {
    // perform the clean up work
    removeShutdownHook();
  }

  @Override
  public RunnerApi.Environment createEnvironment() {
    // command = path of udf runner
    return Environments.createProcessEnvironment("", "", command, generateEnvironmentVariable());
  }

  @Override
  public String createRetrievalToken() {
     // File transfer is unnecessary in process mode,
     // just create an empty RetrievalToken.
     return emptyRetrievalToken;
  }
```

```
  private Map<String, String> generateEnvironmentVariable() {
     // construct the environment variables such as PYTHONPATH, etc
  }

  private void prepareEnvironment() {
    registerShutdownHook();
    prepareWorkingDir();
  }

  private void prepareWorkingDir() {...}

  private Thread registerShutdownHook() {
    Thread thread = new Thread(new DeleteTemporaryFilesHook(pythonTmpDirectory));
    Runtime.getRuntime().addShutdownHook(thread);
    return thread;
  }
}
```

This class is used to prepare and cleanup the working directory and other temporary directories of python worker. It needs the information provided by PythonDependencyManager and a temporary directory as the root of the python working directory. The configured temporary directory of current task manager can be obtained using "getContainingTask().getEnvironment().getTaskManagerInfo().getTmpDirectories()". In current design, 3 kinds of directory are needed to prepare:

1. The directories store the files used to append to PYTHONPATH

Flink distributed cache will wipe the origin file name, including the file format suffix of the uploaded file. But different file formats have different logics to append files to PYTHONPATH:

- If the target file is .py file, we must restore its origin file name and append its parent directory to PYTHONPATH.
- If the target file is egg file or other packaging file which can be imported directly, just append itself to PYTHONPATH.

So it is necessary to restore the original file names of uploaded python files. To avoid naming conflict we should store them in separate directories. Symbolic links can be used here to save copy time and disk space.

2. The directory stores pip install results of the packages listed in the uploaded requirements.txt

Apparently we should not install the users' packages into system python environment. A feasible approach is using "--prefix" param of pip to redirect the install location to a temporary directory, and then append the "bin" directory under the location to PATH variable and append the "site-packages" directory to PYTHONPATH variable.

3. The directory stores the extracted results of the uploaded archives

This directory is used as the working directory of python workers. The contents of uploaded python archives, including users' python environment, will be extracted to the specified sub-directory and can be accessed using relative path in python worker and its launcher script.

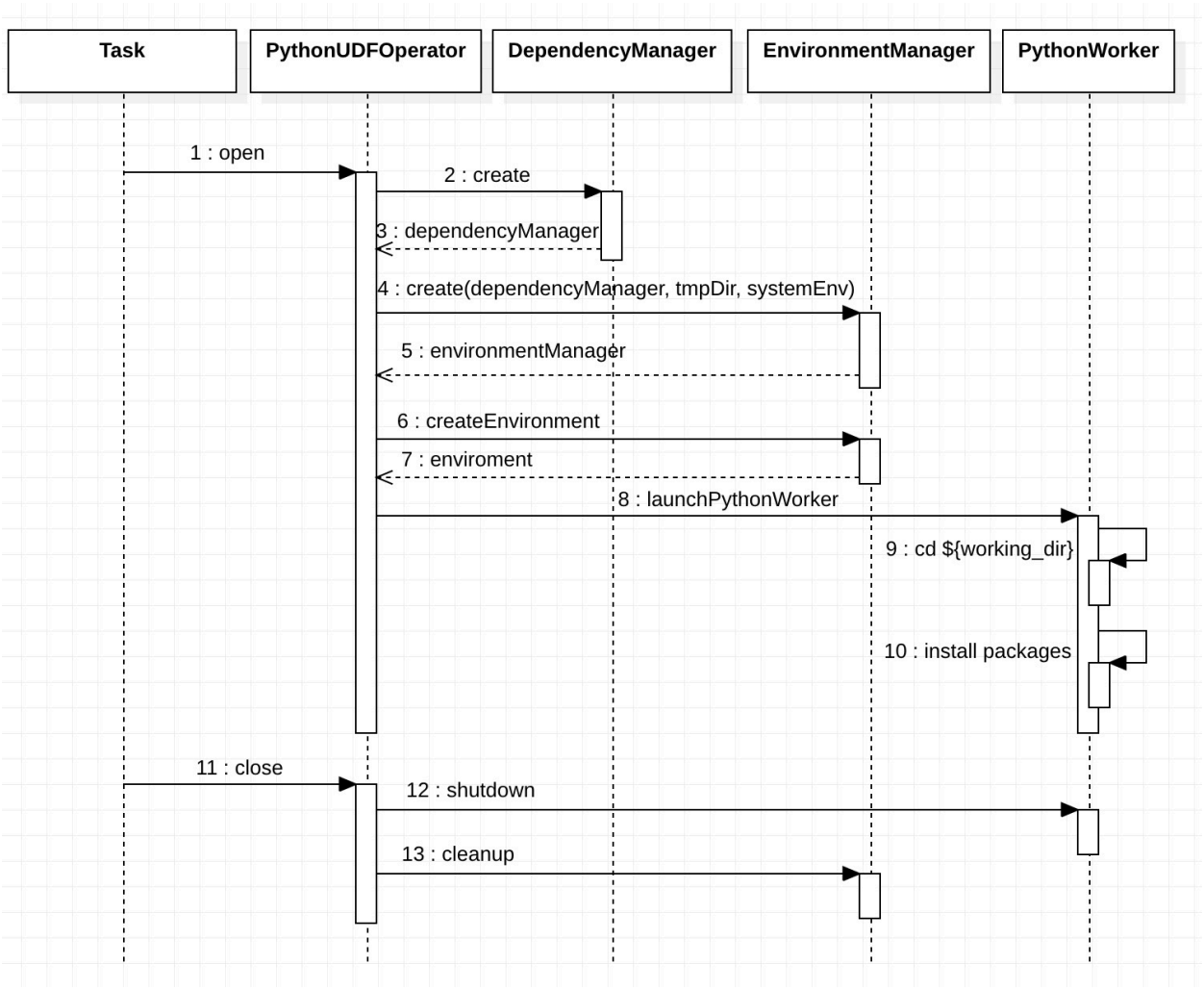
This class should create these directories, and remove them when the task is closing. It is also responsible for adding shutdown hook to ensure the created directories can be deleted once the jvm exits unexpectedly, and removing the shutdown hook when the task is closed to prevent memory leaks.

After the above directories are all ready, the shell script to launch python workers will be executed. The installation of required packages and the changing of working directory will be completed in this script. For each line of the requirements.txt file, the following command will be executed:

```
# just indicate the intention of appending the site-packages directory to PYTHONPATH
# actual code are more complicated
PYTHONPATH=${install_directory}/lib/pythonXY/site-packages:${PYTHONPATH}
export PYTHONPATH
PATH=${install_directory}/bin:${PATH}

${python} -m pip install ${every_line_content} --prefix ${install_directory} --ignore-installed
--no-index --find-links ${cached_dir}
```

If users did not specify the cached dir the param "--no-index --find-links ${cached_dir}" will not be added.

The sequence diagram of runtime environment and dependency management is as follows:



DockerEnvironmentManager

Apache Beam Portability Framework already supports artifact staging that works out of the box with the Docker environment. We can use the artifact staging service defined in Apache Beam to transfer the dependencies from the operator to Python SDK harness running in the docker container.

In general, to support running in docker mode, the following work will be done:

1. Build a docker image which integrates Apache Beam Python SDK harness and Flink Python, which uses boot.py in Flink Python as the entrypoint of container instead of boot.go in Apache Beam to plugin the operations and coders defined in Flink.
2. Build the RetrievalToken file according to user uploaded files. The RetrievalToken is constructed by creating a ProxyManifest object and serialize it into a json-format string. The definition of ProxyManifest can be found in beam_artifact_api.proto.
3. Improves the boot.py defined in Flink Python to download files using Beam's ArtifactService and deploy them inside the docker container.

The structure of DockerEnvironmentManager is as follows:

```
public class DockerEnvironmentManager implements PythonEnvironmentManager {

  public static DockerEnvironmentManager create(
     PythonDependencyManager dependencyManager,
     String tmpDirectoryBase,
     String dockerImageUrl) {

  }

  public DockerEnvironmentManager(...) {
    registerShutdownHook();
  }

  @Override
  public void cleanup() {
     // perform the clean up work
     removeShutdownHook();
  }

  @Override
  public RunnerApi.Environment createEnvironment() {
     return Environments.createDockerEnvironment(dockerImageUrl);
  }

  @Override
  public String createRetrievalToken() {
      // construct the RetrievalToken according to user uploaded files
  }

  private Thread registerShutdownHook() {
     Thread thread = new Thread(new DeleteTemporaryFilesHook(pythonTmpDirectory));
     Runtime.getRuntime().addShutdownHook(thread);
     return thread;
  }
}
```

Use Cases

1. UDF relies on numpy:

```
# command executed in shell:
echo numpy==1.16.5 > requirements.txt
pip download -d cached_dir -r requirements.txt --no-binary :all:

# python code:
t_env.set_python_requirements("requirements.txt", "cached_dir")
```

2. UDF relies on users' other libraries

```
t_env.add_python_file("/user/pyfile/1.py")
t_env.add_python_file("/user/pyfile/2.py")
# directory is also supported
t_env.add_python_file("/user/lib1")
t_env.add_python_file("/user/lib2.zip")
...
```

3. UDF relies on python3.7 but the python on flink cluster is 2.7

```
# command executed in shell:
virtualenv py37 --python=python3 --always-copy
zip -r venv.zip py37

# python code:
t_env.add_python_archives("venv.zip", "venv")
t_env.get_config().set_python_executable("venv/py37/bin/python")
```

4. UDF relies on a specific file, data.txt

```
# command executed in shell:
zip data.zip data.txt

# python code:
t_env.add_python_archives("data.zip", "data")

# in UDF:
with open("data/data.txt", "r") as f:
    ….
```

5. ~~Configure the behaviour of python process~~

```
t_env.set_environment_variable("PYTHONUNBUFFERED", "1")
```