如何在 Apache Flink 1.10 中使用 Python UDF?

作者：孙金城（金竹）

在 Apache Flink 1.9 版中，我们引入了 PyFlink 模块，支持了 Python Table API。Python 用户可以完成数据转换和数据分析的作业。但是，您可能会发现在 PyFlink 1.9 中还不支持定义 Python UDFs，对于想要扩展系统内置功能的 Python 用户来说，这可能有诸多不便。

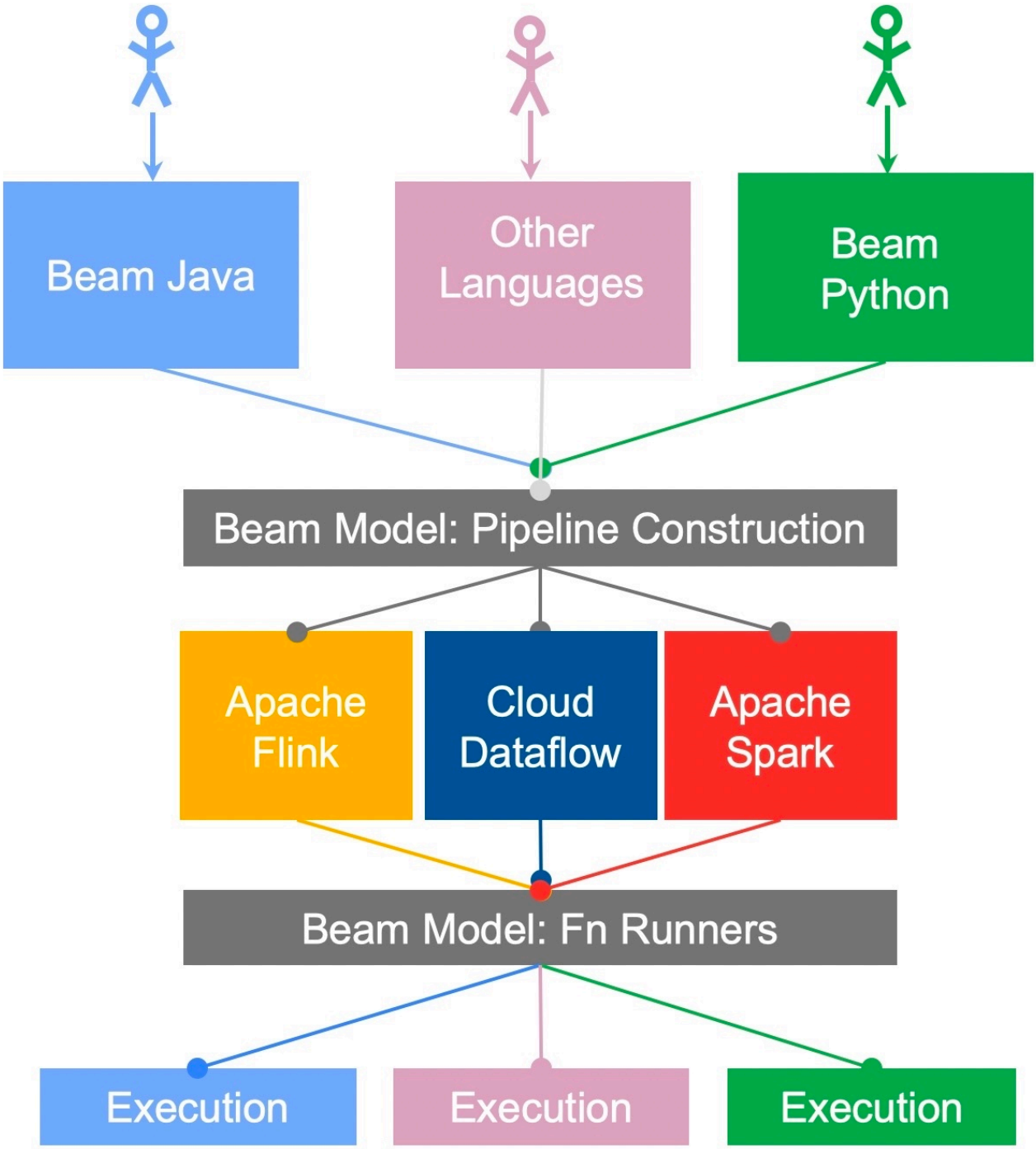
在刚刚发布的 ApacheFlink 1.10 中，PyFlink 添加了对 Python UDFs 的支持。这意味着您可以从现在开始用 Python 编写 UDF 并扩展系统的功能。此外，本版本还支持 Python UDF 环境和依赖管理，因此您可以在 UDF 中使用第三方库，从而利用 Python 生态丰富的第三方库资源。

PyFlink 支持 Python UDFs 的架构

在深入了解如何定义和使用 Python UDFs 之前，我们将解释 UDFs 在 PyFlink 中工作的架构和背景，并提供一些有关我们底层实现的细节介绍。

Beam on Flink

Apache Beam 是一个统一编程模型框架，实现了可使用任何语言开发可以运行在任何执行引擎上的批处理和流处理作业，这得益于 Beam 的 Portability Framework，如下图所示：

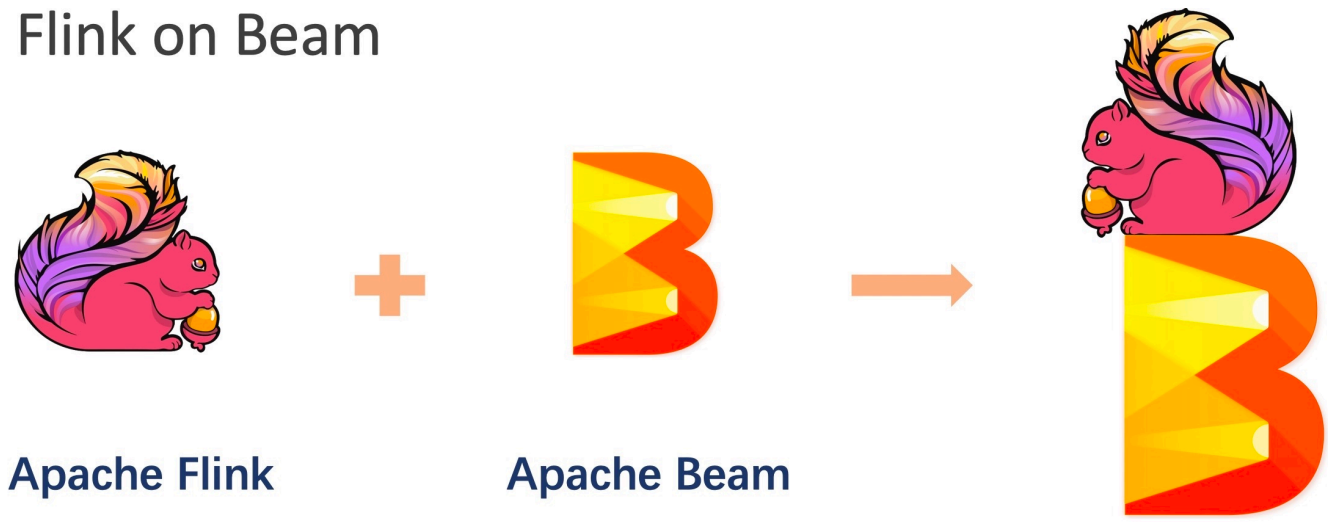


Portability Framework

上图是 Beam 的 Portability Framework 的体系结构。它描述了 Beam 如何支持多种语言和多种引擎的方式。关于 Flink Runner 部分，我们可以说是 Beam on Flink。那么，这与 PyFlink 支持 Python UDF 有什么关系呢？这将接下来“Flink on Beam”中介绍。

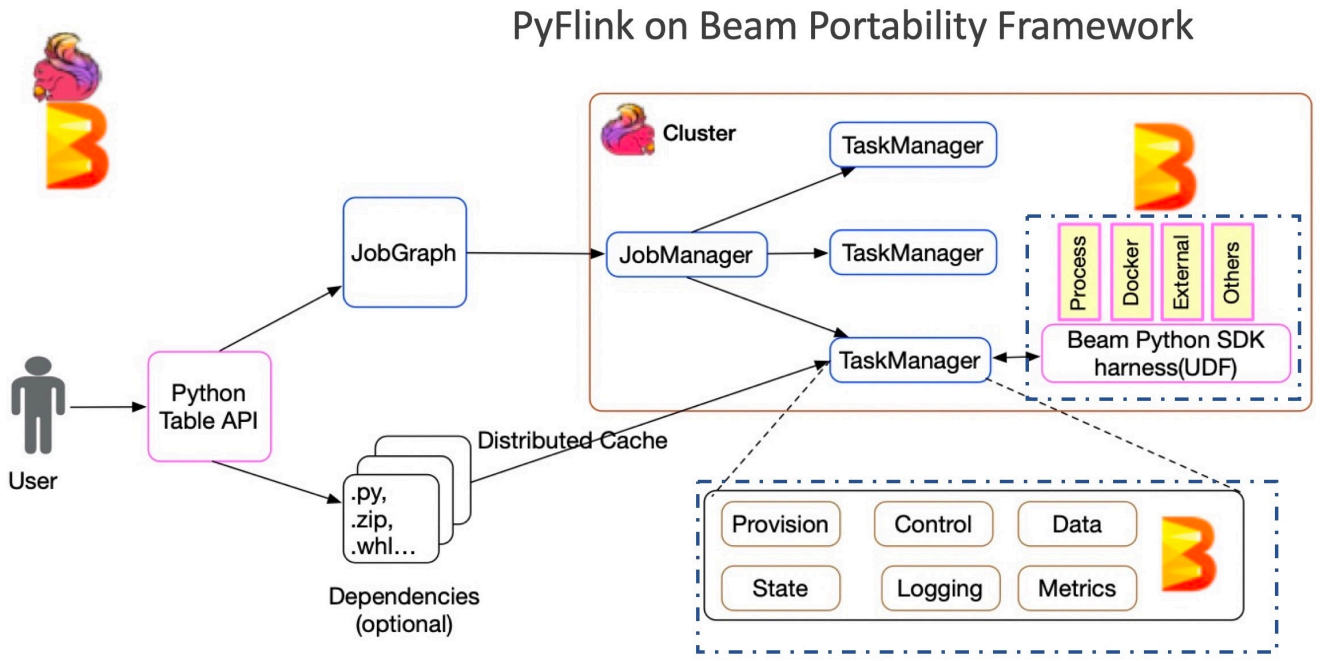
Flink on Beam

Apache Flink 是一个开源项目，因此，它的社区也更多地使用开源。例如，PyFlink 中对 Python UDF 的支持选择了基于 Apache Beam 这辆豪华跑车之上进行构建。:)



Flink on Beam

PyFlink 对 Python UDFs 的支持上，Python 的运行环境管理以及 Python 运行环境 Python VM 和 Java 运行环境 JVM 的通讯至关重要。幸运的是，Apache Beam 的 Portability Framework 完美解决了这个问题。所以才有了如下 PyFlink on Beam Portability Framework 的架构如下：



PyFlink on Beam Portability Framework

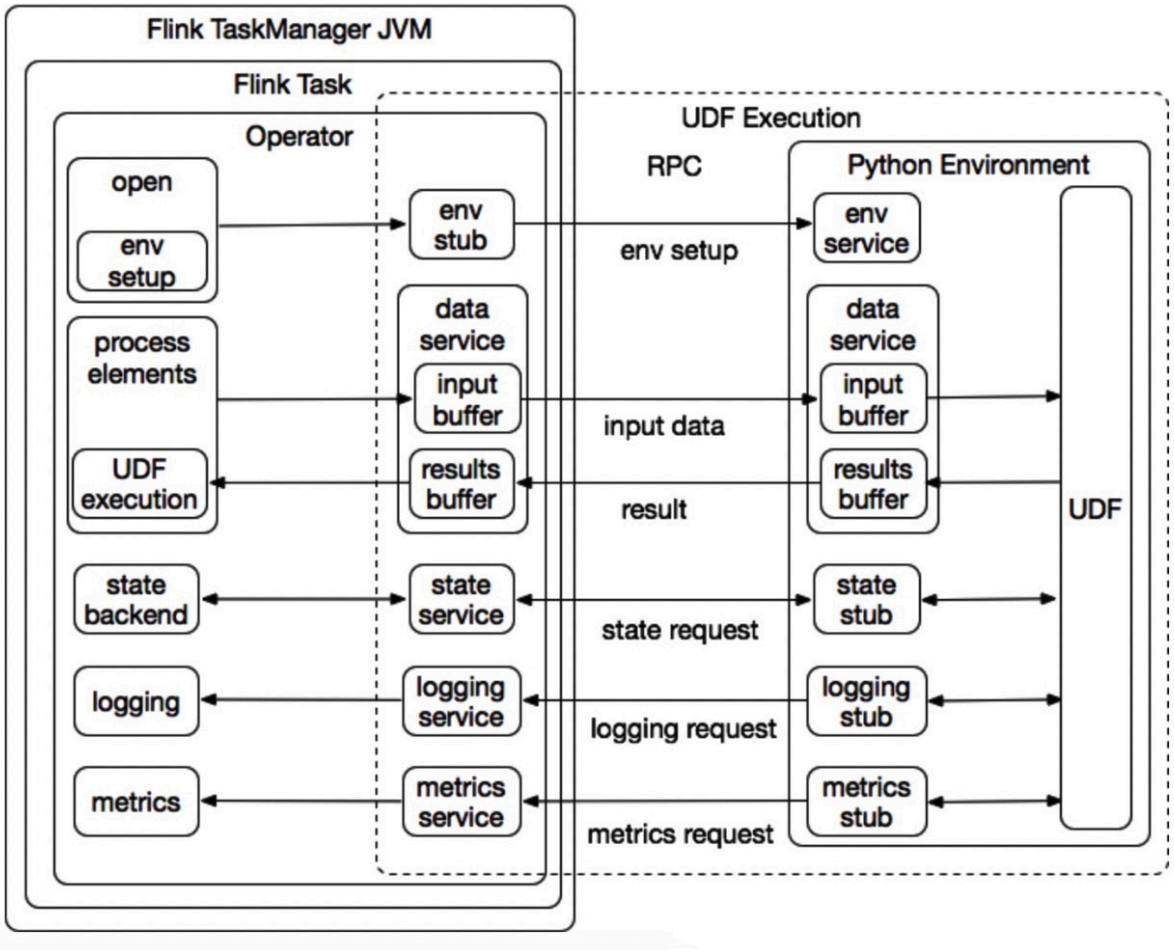
Beam Portability Framework 是一个成熟的多语言支持框架，框架高度抽象了语言之间的通信协议(gRPC),定义了数据的传输格式(Protobuf)，并且根据通用流计算框架所需要的组件，抽象个各种服务，比如, DataService, StateService，MetricsService 等。

在这样一个成熟的框架下，PyFlink 可以快速的构建自己的 Python 算子，同时重用 Apache Beam Portability Framework 中现有 SDK harness 组件，可以支持多种 Python 运行模式，如：Process，Docker，etc.，这使得 PyFlink 对 Python UDF 的支持变得非常容易，在 Apache Flink 1.10 中的功能也非常的稳定和完整。那么为啥说

是 Apache Flink 和 Apache Beam 共同打造呢，是因为我发现目前 Apache Beam Portability Framework 的框架也存在很多优化的空间，所以我在 Beam 社区进行了优化讨论，并且在 Beam 社区也贡献了 30+ 的优化补丁。

JVM 和 Python VM 的通讯

由于 Python UDF 无法直接在 JVM 中运行，因此需要由 Apache Flink 算子在初始化时启动的 Python 进程来准备 Python 执行环境。Python ENV 服务负责启动，管理和终止 Python 进程。如下图 4 所示，Apache Flink 算子和 Python 执行环境之间的通信和涉及多个组件：

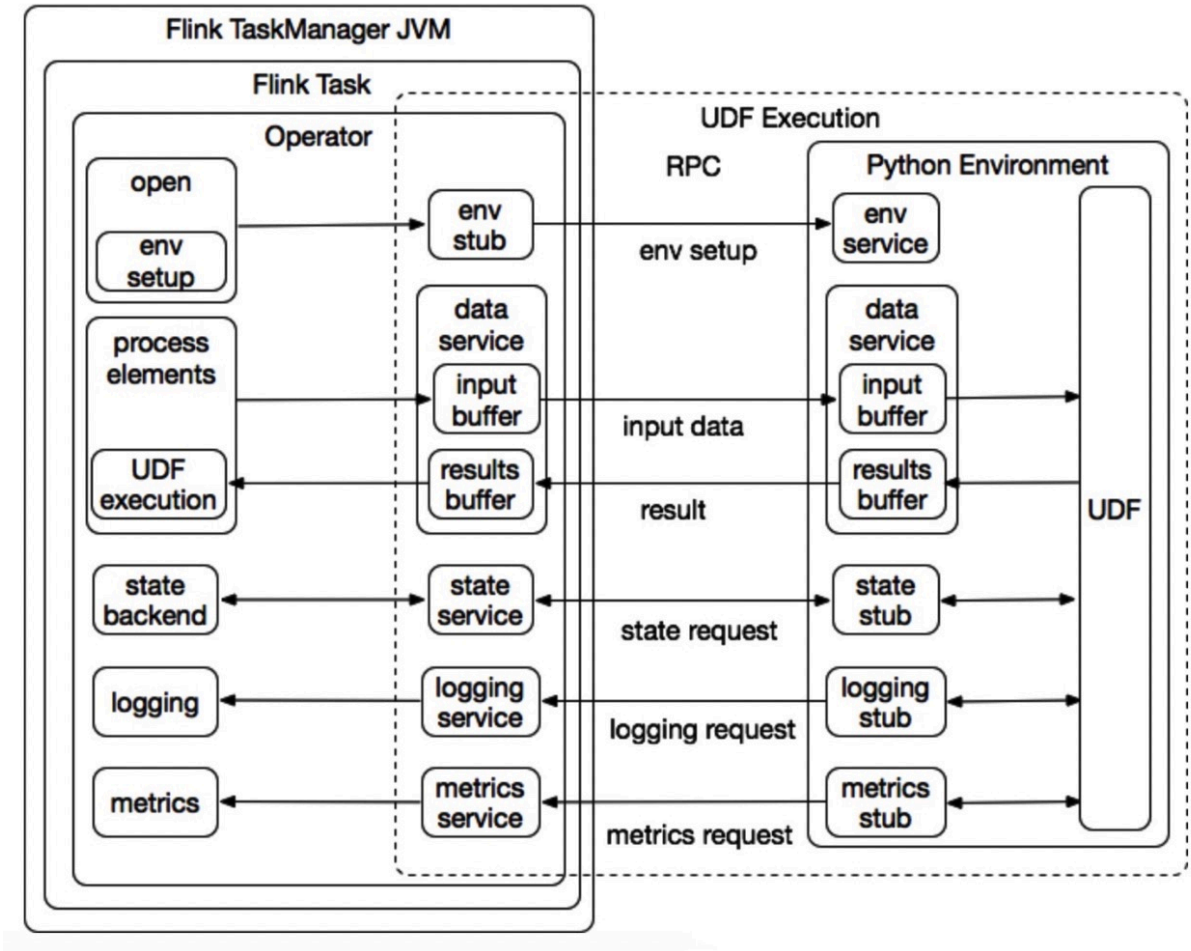


Communication between JVM and Python VM

- **环境管理服务**: 负责启动和终止 Python 执行环境。
- **数据服务**: 负责在 Apache Flink 算子和 Python 执行环境之间传输输入数据和接收用户 UDF 的执行结果。
- **日志服务**: 是记录对用户 UDF 日志输出支持的机制。它可以将用户 UDF 产生的日志传输到 Apache Flink 算子，并与 Apache Flink 的日志系统集成。

说明: 其中 metrics 服务计划在 Apache Flink 1.11 进行支持。

下图描述了从 Java 算子到 Python 进程之间初始化和执行 UDF 的概要流程。



High-level flow between Python VM and JVM

整体流程可以概括为如下两部分：

- 初始化 Python 执行环境。
  - Python UDF Runner 启动所需的 gRPC 服务，如数据服务、日志服务等。
  - Python UDF Runner 另起进程并启动 Python 执行环境。
  - Python worker 向 PythonUserDefinedFunctionRunner 进行注册。
  - Python UDF Runner 向 Python worker 发送需要在 Python 进程中执行的用户定义函数。
  - Python worker 将用户定义的函数转换为 Beam 执行算子（注意：目前，PyFlink 利用 Beam 的可移植性框架[1]来执行 Python UDF）。
  - Python worker 和 Flink Operator 之间建立 gRPC 连接，如数据连接、日志连接等。
- 处理输入元素。
  - Python UDF Runner 通过 gRPC 数据服务将输入元素发送给 Python worker 执行。
  - Python 用户定义函数还可以在执行期间通过 gRPC 日志服务和 metrics 服务将日志和 metrics 收集到 Python UDF Runner。
  - 执行结果可以通过 gRPC 数据服务发送到 Python UDF Runner。

如何在 Apache Flink 1.10 的 PyFlink 中使用 UDFs

本节将介绍用户如何定义 UDF，并完整展示了如何安装 PyFlink，如何在 PyFlink 中定义/注册/调用 UDF，以及如何执行作业。

安装 PyFlink

我们需要先安装 PyFlink，可以通过 PyPI 获得，并且可以使用 pip install 进行便捷安装。

注意: 安装和运行 PyFlink 需要 Python 3.5 或更高版本。

```
$ python -m pip install apache-Apache Flink
```

定义一个 UDF

除了扩展基类 ScalarFunction 之外，定义 Python UDF 的方法有很多。下面的示例显示了定义 Python UDF 的不同方法，该函数以 BIGINT 类型的两列作为输入参数，并返回它们的和作为结果。

- Option 1: extending the base class ScalarFunction

```
class Add(ScalarFunction):
  def eval(self, i, j):
    return i + j

add = udf(Add(), [DataTypes.BIGINT(), DataTypes.BIGINT()], DataTypes.BIGINT())
```

- Option 2: Python function

```
@udf(input_types=[DataTypes.BIGINT(), DataTypes.BIGINT()], result_type=DataTypes.
def add(i, j):
  return i + j
```

- Option 3: lambda function

```
add = udf(lambda i, j: i + j, [DataTypes.BIGINT(), DataTypes.BIGINT()], DataTypes
```

- Option 4: callable function

```
class CallableAdd(object):
  def __call__(self, i, j):
    return i + j

add = udf(CallableAdd(), [DataTypes.BIGINT(), DataTypes.BIGINT()], DataTypes.BIGI
```

- Option 5: partial function

```
return i + j + k

add = udf(functools.partial(partial_add, k=1), [DataTypes.BIGINT(), DataTypes.BIG
          DataTypes.BIGINT())
```

注册一个UDF

- register the Python function

```
table_env.register_function("add", add)
```

- Invoke a Python UDF

```
my_table.select(```js
"add(a, b)")
```

- Example Code

下面是一个使用 Python UDF 的完整示例。

```
from PyFlink.table import StreamTableEnvironment, DataTypes
from PyFlink.table.descriptors import Schema, OldCsv, FileSystem
from PyFlink.table.udf import udf

env = StreamExecutionEnvironment.get_execution_environment()
env.set_parallelism(1)
t_env = StreamTableEnvironment.create(env)

t_env.register_function("add", udf(lambda i, j: i + j, [DataTypes.BIGINT(), DataT

t_env.connect(FileSystem().path('/tmp/input')) \
    .with_format(OldCsv()
                 .field('a', DataTypes.BIGINT())
                 .field('b', DataTypes.BIGINT())) \
    .with_schema(Schema()
                 .field('a', DataTypes.BIGINT())
                 .field('b', DataTypes.BIGINT())) \
    .create_temporary_table('mySource')

t_env.connect(FileSystem().path('/tmp/output')) \
    .with_format(OldCsv()
                 .field('sum', DataTypes.BIGINT())) \
    .with_schema(Schema()
                 .field('sum', DataTypes.BIGINT())) \
    .create_temporary_table('mySink')

t_env.from_path('mySource')\
    .select("add(a, b)") \
```

```
    .insert_into('mySink')

t_env.execute("tutorial_job")
```

- 提交作业

首先，您需要在“ / tmp / input”文件中准备输入数据。例如，

```
$ echo "1,2" > /tmp/input
```

接下来，您可以在命令行上运行此示例：

```
$ python python_udf_sum.py
```

通过该命令可在本地小集群中构建并运行 Python Table API 程序。您还可以使用不同的命令行将 Python Table API 程序提交到远程集群。

最后，您可以在命令行上查看执行结果：

```
$ cat /tmp/output
3
```

Python UDF 的依赖管理

在许多情况下，您可能希望在 Python UDF 中导入第三方依赖。下面的示例将指导您如何管理依赖项。

假设您想使用 mpmath 来执行上述示例中两数的和。Python UDF 逻辑可能如下：

```
@udf(input_types=[DataTypes.BIGINT(), DataTypes.BIGINT()], result_type=DataTypes.
def add(i, j):
    from mpmath import fadd # add third-party dependency
    return int(fadd(1, 2))
```

要使其在不包含依赖项的工作节点上运行，可以使用以下 API 指定依赖项：

```
# echo mpmath==1.1.0 > requirements.txt
# pip download -d cached_dir -r requirements.txt --no-binary :all:
t_env.set_python_requirements("/path/of/requirements.txt", "/path/of/cached_dir")
```

用户需要提供一个 requirements.txt 文件，并且在里面申明使用的第三方依赖。如果无法在群集中安装依赖项（网络问题），则可以使用参数“requirements_cached_dir”，指定包含这些依赖项的安装包的目录，如上面的示例所示。依赖项将上传到群集并脱机安装。

下面是一个使用依赖管理的完整示例：

```
from PyFlink.datastream import StreamExecutionEnvironment
from PyFlink.table import StreamTableEnvironment, DataTypes
from PyFlink.table.descriptors import Schema, OldCsv, FileSystem
from PyFlink.table.udf import udf

env = StreamExecutionEnvironment.get_execution_environment()
env.set_parallelism(1)
t_env = StreamTableEnvironment.create(env)

@udf(input_types=[DataTypes.BIGINT(), DataTypes.BIGINT()], result_type=DataTypes.
def add(i, j):
    from mpmath import fadd
    return int(fadd(1, 2))

t_env.set_python_requirements("/tmp/requirements.txt", "/tmp/cached_dir")
t_env.register_function("add", add)

t_env.connect(FileSystem().path('/tmp/input')) \
    .with_format(OldCsv()
                 .field('a', DataTypes.BIGINT())
                 .field('b', DataTypes.BIGINT())) \
    .with_schema(Schema()
                 .field('a', DataTypes.BIGINT())
                 .field('b', DataTypes.BIGINT())) \
    .create_temporary_table('mySource')

t_env.connect(FileSystem().path('/tmp/output')) \
    .with_format(OldCsv()
                 .field('sum', DataTypes.BIGINT())) \
    .with_schema(Schema()
                 .field('sum', DataTypes.BIGINT())) \
    .create_temporary_table('mySink')

t_env.from_path('mySource')\
    .select("add(a, b)") \
    .insert_into('mySink')

t_env.execute("tutorial_job")
```

- 提交作业

首先，您需要在“/ tmp / input”文件中准备输入数据。例如，

```
echo "1,2" > /tmp/input
1
2
```

其次，您可以准备依赖项需求文件和缓存目录：

```
$ echo "mpmath==1.1.0" > /tmp/requirements.txt
$ pip download -d /tmp/cached_dir -r /tmp/requirements.txt --no-binary :all:
```

接下来，您可以在命令行上运行此示例：

```
$ python python_udf_sum.py
```

最后，您可以在命令行上查看执行结果：

```
$ cat /tmp/output
3
```

快速上手

PyFlink 为大家提供了一种非常方便的开发体验方式 - PyFlink Shell。当成功执行 python -m pip install apache-flink 之后，你可以直接以 pyflink-shell.sh local 来启动一个 PyFlink Shell 进行开发体验，如下所示：

更多场景

不仅仅是简单的 ETL 场景支持，PyFlink 可以完成很多复杂场的业务场景需求，比如我们最熟悉的双 11 大屏的场景，如下：

关于上面示例的更多详细请查阅：

https://enjoyment.cool/2019/12/05/Apache-Flink-说道系列-如何在PyFlink-1-10中自定义Python-UDF/

总结和未来规划

在本博客中，我们介绍了 PyFlink 中 Python UDF 的架构，并给出了如何定义、注册、调用和运行 UDF 的示例。随着 1.10 的发布，它将为 Python 用户提供更多的可能来编写 Python 作业逻辑。同时，我们一直积极与社区合作，不断改进 PyFlink 的功能和性能。今后，我们计划在标量和聚合函数中引入对 Pandas 的支持；通过 SQL 客户端增加对 Python UDF 使用的支持，以扩展 Python UDF 的使用范围；并做更多的性能改进。近期，邮件列表上有一个关于新功能支持的讨论，您可以查看并找到更多详细信息。

在社区贡献者的不断努力之下，PyFlink 的功能可以如上图一样可以迅速从幼苗变成大树:

PyFlink 需要你的加入

PyFlink 是一个新组件，仍然需要做很多工作。因此，热诚欢迎每个人加入对 PyFlink 的贡献，包括提出问题，提交错误报告，提出新功能，加入讨论，贡献代码或文档。期望在 PyFlink 见到你！