Flink 生态：一个案例快速上手 PyFlink

> 云栖号：https://yqh.aliyun.com第一手的上云资讯，不同行业精选的上云企业案例库，基于众多成功案例萃取而成的最佳实践，助力您上云决策！

Flink 从 1.9.0 版本开始增加了对 Python 的支持（PyFlink），在刚刚发布的 Flink 1.10 中，PyFlink 添加了对 Python UDFs 的支持，现在可以在 Table API/SQL 中注册并使用自定义函数。PyFlink 的架构如何，适用于哪些场景？本文将详细解析并进行 CDN 日志分析的案例演示。

PyFlink 的必要性

Flink on Python and Python on Flink

PyFlink 是什么？这个问题也许会让人感觉问题的答案太明显了，那就是 Flink + Python，也就是 Flink on Python。那么到底 Flink on Python 意味着这什么呢？那么一个非常容易想到的方面就是能够让 Python 用享受到 Flink 的所有功能。其实不仅如此，PyFlink 的存在还有另外一个非常重要的意义就是，Python on Flink，我们可以将 Python 丰富的生态计算能力运行在 Flink 框架之上，这将极大的推动 Python 生态的发展。其实，如果你再仔细深究一下，你会发现这个结合并非偶然。

Python 生态和大数据生态

Pythoh 生态与大数据生态有密不可分的关系，我们先看看大家都在用 Python 解决什么实际问题？通过一份用户调查我们发现，大多数 Python 用户正在解决 ”数据分析“，”机器学习“的问题，那么这些问题场景在大数据领域也有很好的解决方案。那么 Python 生态和大数据生态结合，抛开扩大大数据产品的受众用户之外，对 Python 生态一个特别重要到意义就是单机到分布式的能力增强，我想，这也是大数据时代海量数据分析对 Python 生态的强需求。

Why Flink and Why Python

好了， Python 生态和大数据的结合是时代的要求，那么 Flink 为啥选择 Python 生态作为多语言支持的切入点，而不是 Go 或者 R 呢？作为用户的你，为啥选择 PyFlink 而不是 PySpark 或者 PyHive 呢？

首先我们说说选择 Flink 的理由：

- 第一，最主要的是架构优势， Flink 是纯流架构的流批统一的计算引擎；
- 第二，从 ASF 的客观统计看， Flink 是 2019 年度最活跃的开源项目，这意味着 Flink 鲜活的生命力；
- 第三， Flink 不仅仅是开源项目而且也经历过无数次，各个大数据公司的生产环境的历练，值得信赖。

那么我们再来看看 Flink 在选择多语言支持时候，为啥选择了 Python 而不是其他语言呢？我们还是看一下数据统计，如下： Python 语言流行程度仅次于 Java 和 C，其实我们发现自18年开始 Python 的发展非常迅速，并且还在持续。那么 Java/Scala 是 Flink 的默认语言，所以选择 Python 来进行 Flink 多语言的支持似乎很合理。这些权威的统计信息，大家可以在我提供的链接进行查看更详细的信息。

目前看 PyFlink 的产生是时代的必然，但仅仅想清楚 PyFlink 存在的意义还远远不够，因为我们最终的目标是让 Flink 和 Python 用户受益，真真正正的解决实际的现实问题。所以，我们还需要继续深入，一起探究 PyFlink 该如

何落地?

PyFlink 架构

任何事情在想清楚之后，还要做明白，要将 PyFlink 落地，首要解决的是分析清楚要达成的核心目标和要达成目标解决的核心问题。那么 PyFlink 的核心目标到底是什么呢?

PyFlink 的核心目标

我们在前面的分析过程中已经提到过，这里我们再具体化一下，PyFlink 的核心目标：

1. 将 Flink 能力输出到 Python 用户，进而可以让 Python 用户使用所有的 Flink 能力。
2. 将 Python 生态现有的分析计算功能运行到 Flink 上，进而增强 Python 生态对大数据问题的解决能力。

围绕这 2 个核心的目标，我们再来分析，要达成这样的目标，需要解决的核心问题是什么?

Flink 功能 Python 化

为了 PyFlink 落地，我们需要在 Flink 上开发一套和现有 Java 一样的 Python 的引擎吗？答案是 NO，这在 Flink 1.8 之前已经尝试过。我们做设计有一个很好的原则就是追求以最小的代价完成既定的目标，所以最好的方式是仅仅提供一层 Python API 复用现有的计算引擎。

那么对于 Flink 而言我们要提供怎样的 Python API 呢？那就是我们熟知的： High-level 的 TableAPI/SQL 和有状态的 DataStream API。好，我们现在的思考越来越切近 Flink 内部了，接踵而来的问题就是，我们如何提供一套 Python 的 Table API 和 DataStream API 呢？核心要解决的问题是什么呢?

■ Flink 功能 Python 化的核心问题

核心问题显而易见是 Python VM 和 Java VM 的握手，他们之间要建立通讯，这是 Flink 多语言支持的核心问题。好，面对核心问题我们要进行技术选型. Here we go…

■ Flink 功能 Python 化的 VM 通讯技术选型

就当前的 Java VM 和 Python VM 通讯的问题而言，目前比较显著的解决方案有 Apache Beam，一个著名的多语言多引擎支持项目，另外一个专门解决 Java VM 和 Python VM 通讯问题的 Py4J。我们从不同视角进行分析对比， 首先， Py4J 和 Beam 对比，就好像有穿山功能的穿山甲和一个力量强大的大象，要穿越一道墙，我们可以打个洞，也可以推到整面墙。所以在当前 VM 通讯的场景， Beam 显得有些复杂。因为 Beam 在通用性上做了很多的努力，在极端情况会丧失一定程度的灵活性。

从另一个视角来看， Flink 本身有交互式编程的需求，比如 FLIP-36 ，同时还要在多语言支持的同时，确保各种语言的接口设计语义一致性，这些在 Beam 现有的架构下很难满足。所以在这样一种思考下，我们选择 Py4J 作为 Java VM 和 Python VM 之间通讯的桥梁。

■ Flink 功能 Python 化的技术架构

其实如果我们解决了 Python VM 和 Java VM 通讯的问题，本质上是在努力达成我们第一个目标，就是将现有 Flink 功能输出给 Python 用户，也就是我们 Flink 1.9 所完成的工作，接下来我们看看 Flink 1.9 PyFlink API 的架构，如下：

我们利用Py4J解决通讯问题，在 PythonVM 启动一个 Gateway，并且 Java VM 启动一个 Gateway Server 用于接受 Python 的请求，同时在 Python API 里面提供和 Java API 一样的对象，比如 TableENV， Table，等等。这样 Python 在写 Python API 的时候本质是在调用 Java API。当然，在 Flink 1.9 中还解决了作业部署问题，我们可以用 Python 命令，Python shell 和 CLI 等多种方式进行作业提交。

那么基于这样的架构有怎样的优势呢？第一个就是简单，并确保 Python API 语义和 Java API 的一致性，第二点， Python 作业可以达到和 Java 一样的极致性能，那么 Java 的性能怎样呢？我想大家已经熟知，在去年双 11 Flink Java API 已经具备了每秒25.51亿次的数据处理的能力。

Python 生态分布化

OK，在完成了现有 Flink 功能向 Python 用户的输出之后，接下来我们继续探讨，如何将 Python 生态功能引入 Flink 中，进而将 Python 功能分布式化。如何达成？通常我们可以有如下2种做法：

1. 选择有代表性的 Python 类库，将其 API 增加到 PyFlink 中，这种方式是一个漫长的过程，因为 Python 的生态库太多了，但无论如何，我们在引入这些 APIs 之前，首要解决的问题是，解决 Python 的执行问题。
2. 我们结合现有 Flink Table API 的现状和现有 Python 类库的特点，我们可以对现有所有的 Python 类库功能视为 用户自定义函数（UDF），集成到 Flink 中。这样我们就找到了集成 Python 生态到 Flink 中的手段是将其视为 UDF，也就是我们 Flink 1.10 中的工作。那么集成的核心问题是什么？没错，刚才说过，是 Python UDF 的执行问题。

好，我们针对这个核心问题进行技术选型吧，Here we go…

■ Python 生态分布化的 UDF 执行技术选型

解决 Python UDF 执行问题可不仅仅是 VM 之间通讯的问题了，它涉及到 Python 执行环境的管理，业务数据在 Java 和 Python 之间的解析， Flink State Backend 能力向 Python 的输出， Python UDF 执行的监控等等，是一个非常复杂的问题。面对这样复杂的问题，前面我们介绍过 Apache Beam ，支持多引擎多语言，无所不能的大象可以出场了，我们来看一下 Beam 是怎么解决 Python UDF 执行问题的：）

Beam 为了解决多语言和多引擎支持问题高度抽象了一个叫 Portability Framework 的架构，如下图，Beam 目前可以支持 Java/Go/Python 等多种语言，其中图下方 Beam Fu Runners 和 Execution 之间就解决了 引擎和 UDF 执行环境的问题。其核心是对利用 Protobuf 进行数据结构抽象，利用 gRPC 协议进行通讯，同时封装了核心的 gRPC 服务。所以这时候 Beam 更像是一只萤火虫，照亮了 PyFlink 解决 UDF 执行问题之路。：）（多说一嘴，萤火虫已经成为了 Aapche Beam 的吉祥物）。

我们接下来看看 Beam 到底提供了哪些 gRPC 服务。

如图 Runner部分是 Java 的算子执行， SDK Worker部分是 Python 的执行环境， Beam已经抽象 Control/Data/State/Logging 等服务。并这些服务已经在 Beam 的 Flink runner 上稳定高效的运行了很久了。所以在 PyFlink UDF 执行上面我们可以站在巨人的肩膀上了：），这里我们发现 Apache Beam 在 API 层面和在 UDF 的执行层面都有解决方案，而 PyFlink 在 API 层面采用了 Py4J 解决 VM 通讯问题，在 UDF 执行需求上采用了 Beam 的 Protability Framework 解决 UDF 执行环境问题。

这也表明了 PyFlink 在技术选型上严格遵循以最小的代价达成既定目标的原则，在技术选型上永远会选择最合适的，最符合 PyFlink 长期发展的技术架构。（BTW，与 Beam 的合作过程中，我也向 Beam 社区提交了20+的优化 patch）。

■ Python 生态分布化的 UDF 技术架构

在 UDF 的架构中我们我既要考虑 Java VM 和 Python VM 的通讯问题，又要考虑在编译阶段和在运行阶段的不同需求。图中我们以绿色表示 Java VM 的行为，蓝色表示 Python VM 的行为。首先我们看看编译阶段，也就是local 的设计，在local的设计是纯 API 的 mapping 调用，我们仍然要过 Py4J 来解决通讯问题，也就是如图 Python 每执行一个 API 就会同步的调用 Java 所对应的 API 。

对 UDF 的支持上，需要添加 UDF 注册的 API， register_function，但仅仅是注册还不够，用户在自定义 Python UDF 的时候往往会依赖一些三方库，所以我们还需要增加添加依赖的方法，那就是一系列的 add 方法，比如

add_Python_file()。在编写 Python 作业的同时， Java API 也会同时被调用在提交作业之前，Java 端会构建 JobGraph。然后通过 CLI 等多种方式将作业提交到集群进行运行。

我们再来看看运行时 Python 和 Java 的不同分工情况，首先在 Java 端与普通 Java 作业一样， JobMaster 将作业分配给 TaskManger ， TaskManager 会执行一个个 Task ，task里面就涉及到了Java和Python的算子执行。在Python UDF的算子中我们会设计各种 gRPC 服务来完成 Java VM 和 Python VM 的各种通讯，比如 DataService 完成业务数据通讯， StateService 完成 Python UDF 对 Java Statebackend 的调用，当然还有 Logging 和 Metrics 等其他服务。

这些服务都是基于 Beam 的 Fn API 来构建的，最终在Python的 Worker 里面运行用户的 UDF，运行结束之后再利用对应的 gRPC 服务将结果返回给 Java 端的 PythonUDF 算子。当然 Python 的 worker 不仅仅是 Process 模式，可以是 Docker 模式甚至是 External 的服务集群。这种扩展机制，为后面 PyFlink 与 Python 生态的其他框架集成打下了坚实的基础，在后面我们介绍 PyFlink 大图的时候，我们会介绍这一点：）。好，这就是 PyFlink 在 1.10 中引入 Python UDF 支持的架构。那么这样的架构有怎样的优势呢?

首先，Beam 是一个成熟的多语言支持框架，基于 Beam 进行架构我们后面可以很容易进行其他语言的支持扩展。 同时 Beam 对 State 的服务抽象也方便 PyFlink 增加对 Stateful UDF 的支持。还有一个方面是方便维护，同一套框架由 Apache Beam 和 Apache Flink 两个非常活跃的社区共同维护和优化 …

PyFlink 场景，怎么用?

好了解了这么多关于 PyFlink 的架构和架构背后的思考，我们还是以一个具体场景案例，来增加一些对 PyFlink 的体感吧!

PyFlink 适用的场景

在具体的案例之前我们先简单分享一些 PyFlink 所能适用的业务场景。首先 PyFlink 既然是 Python+Flink，那其适用场景也可以从 java 和 Python 两方面去分析，第一个 Java 所适用的场景 PyFlink 都适用。

- 第一个，事件驱动型，比如：刷单，监控等；
- 第二个，数据分析型的，比如：库存，双11大屏等；
- 第三个适用的场景是数据管道，也就是ETL场景，比如一些日志的解析等；
- 第四个场景，机器学习，比如个性推荐等。

这些都可以尝试使用 PyFlink 。除此之外还有 Python 生态特有的场景，比如科学计算等。那么这么多的应用场景，PyFlink 到底有哪些可用的 API 呢?

PyFlink 的安装

使用具体的 API 开发之前，首先要安装 PyFlink，目前 PyFlink 支持 pip install 进行安装，这里特别提醒一下具体命令是：pip install apache-Flink。

PyFlink 的 APIs

目前 PyFlink API 完全与 Java Table API 对齐，各种关系操作都支持，同时对 window 也有很好的支持，并且这里稍微提一下就是 PyFlink 里面有些易用性 API 比 SQL 还要强大，比如：这些对 columns 进行操作的 APIs。除了这些 APIs，PyFlink还提供多种定义 Python UDF 的方式。

PyFlink 的 UDF 定义

首先，可以扩展 ScalarFunction，这种方式可以提供更多的辅助功能，比如添加 Metrics 。除此之外 Python 语言所支持的任何方式的方法定义，在 PyFlink UDF 中都是支持的，比如：Lambda Function，Named Function 和

CallableFunction等。

当定义完方法后，我们用 PyFlink 所提供的 Decorators 进行打标，并描述 input 和 output 的数据类型就可以了。当然后面版本我们也可以根据 Python 语言的 type hint 特性再进一步简化，进行类型推导。为了直观，我们看一个具体的 UDF 定义的例子：

Python UDF 定义示例

我们定义两个数相加的例子，首先导入必须的类，然后是刚才我们提到的几种定义方式。这个简单直接，我们闲话少叙，开始看看实际的案例吧：）

PyFlink 案例-阿里云 CDN 实时日志分析

我们这里以一个阿里云 CDN 实时日志分析的例子来介绍如何用 PyFlink 解决实际的业务问题。CDN 我们都很熟悉，为了进行资源的下载加速。那么 CDN 日志的解析一般有一个通用的架构模式，就是首先要将各个边缘节点的日志数据进行采集，一般会采集到消息队列，然后将消息队列和实时计算集群进行集成进行实时的日志分析，最后将分析的结果写到存储系统里面。那么我今天的案例将架构实例化，消息队列采用 Kafka，实时计算采用Flink，最终将数据存储到 MySQL 中。

■ 阿里云 CDN 实时日志分析需求说明

我们在来看看业务统计的需求，为了介绍方便，我们将实际的统计需求进行简化，示例中只进行按地区分组，进行资源访问量，下载量和下载速度的统计。数据格式我们只选取核心的字段，比如：uuid，表示唯一的日志标示，client_ip 表示访问来源，request_time 表示资源下载耗时， response_size 表示资源数据大小。其中我们发现我们需求是按地区分组，但是原始日志里面并没有地区的字段信息，所以我们需要定义一个 Python UDF 根据 client_ip 来查询对应的地区。好，我们首先看如何定义这个 UDF。

■ 阿里云 CDN 实时日志分析 UDF 定义

这里我们用了刚才提到的 named function 的方式定义一个 ip_to_province() 的UDF，输入是 ip 地址，输出是地区名字字符串。我们这里描述了输入类型是一个字符串，输出类型也是一个字符串。当然这里面的查询服务仅供演示，大家在自己的生产环境要替换为可靠的地域查询服务。

```
import re
import json
from pyFlink.table import DataTypes
from pyFlink.table.udf import udf
from urllib.parse import quote_plus
from urllib.request import urlopen

@udf(input_types=[DataTypes.STRING()], result_type=DataTypes.STRING())
def ip_to_province(ip):
    """
    format:
        {
        'ip': '27.184.139.25',
        'pro': '河北省',
        'proCode': '130000',
        'city': '石家庄市',
        'cityCode': '130100',
        'region': '灵寿县',
        'regionCode': '130126',
```

```
        'addr': '河北省石家庄市灵寿县 电信',
        'regionNames': '',
        'err': ''
        }
    """
    try:
        urlobj = urlopen( \
         'http://whois.pconline.com.cn/ipJson.jsp?ip=%s' % quote_plus(ip))
        data = str(urlobj.read(), "gbk")
        pos = re.search("{[^{}]+\}", data).span()
        geo_data = json.loads(data[pos[0]:pos[1]])
        if geo_data['pro']:
            return geo_data['pro']
        else:
            return geo_data['err']
    except:
        return "UnKnow"
```

■ 阿里云 CDN 实时日志分析 Connector 定义

我们完成了需求分析和 UDF 的定义，我们开始进行作业的开发了，按照通用的作业结构，需要定义 Source connector 来读取 Kafka 数据，定义 Sink connector 来将计算结果存储到 MySQL。最后是编写统计逻辑。

在这特别说明一下，在 PyFlink 中也支持 SQL DDL 的编写，我们用一个简单的 DDL 描述，就完成了 Source Connector的开发。其中 connector.type 填写 kafka。SinkConnector 也一样，用一行DDL描述即可，其中 connector.type 填写 jdbc。描述 connector 的逻辑非常简单，我们再看看核心统计逻辑是否也一样简单：）

```
kafka_source_ddl = """
CREATE TABLE cdn_access_log (
 uuid VARCHAR,
 client_ip VARCHAR,
 request_time BIGINT,
 response_size BIGINT,
 uri VARCHAR
) WITH (
 'connector.type' = 'kafka',
 'connector.version' = 'universal',
 'connector.topic' = 'access_log',
 'connector.properties.zookeeper.connect' = 'localhost:2181',
 'connector.properties.bootstrap.servers' = 'localhost:9092',
 'format.type' = 'csv',
 'format.ignore-parse-errors' = 'true'
)
"""

mysql_sink_ddl = """
CREATE TABLE cdn_access_statistic (
 province VARCHAR,
 access_count BIGINT,
 total_download BIGINT,
 download_speed DOUBLE
```

```
) WITH (
 'connector.type' = 'jdbc',
 'connector.url' = 'jdbc:mysql://localhost:3306/Flink',
 'connector.table' = 'access_statistic',
 'connector.username' = 'root',
 'connector.password' = 'root',
 'connector.write.flush.interval' = '1s'
)
"""
```

■ 阿里云 CDN 实时日志分析核心统计逻辑

首先从数据源读取数据，然后需要先将 clien_ip 利用我们刚才定义的 ip_to_province(ip) 转换为具体的地区。之后，在进行按地区分组，统计访问量，下载量和资源下载速度。最后将统计结果存储到结果表中。这个统计逻辑中，我们不仅使用了Python UDF，而且还使用了 Flink 内置的 Java AGG 函数，sum 和 count。

```
# 核心的统计逻辑
t_env.from_path("cdn_access_log")\
   .select("uuid, "
           "ip_to_province(client_ip) as province, " # IP 转换为地区名称
           "response_size, request_time")\
   .group_by("province")\
   .select( # 计算访问量
           "province, count(uuid) as access_count, "
           # 计算下载总量
           "sum(response_size) as total_download,  "
           # 计算下载速度
           "sum(response_size) * 1.0 / sum(request_time) as download_speed"
) \
   .insert_into("cdn_access_statistic")
```

■ 阿里云 CDN 实时日志分析完整代码

我们在整体看一遍完整代码，首先是核心依赖的导入，然后是我们需要创建一个ENV，并设置采用的 planner（目前Flink支持Flink和blink两套 planner）建议大家采用 blink planner。

接下来将我们刚才描述的 kafka 和 mysql 的 ddl 进行表的注册。再将 Python UDF 进行注册，这里特别提醒一点，UDF所依赖的其他文件也可以在API里面进行制定，这样在job提交时候会一起提交到集群。然后是核心的统计逻辑，最后调用 executre 提交作业。这样一个实际的CDN日志实时分析的作业就开发完成了。我们再看一下实际的统计效果。

```
import os

from pyFlink.datastream import StreamExecutionEnvironment
from pyFlink.table import StreamTableEnvironment, EnvironmentSettings
from enjoyment.cdn.cdn_udf import ip_to_province
from enjoyment.cdn.cdn_connector_ddl import kafka_source_ddl, mysql_sink_dd
l

# 创建Table Environment,  并选择使用的Planner
env = StreamExecutionEnvironment.get_execution_environment()
```

```
t_env = StreamTableEnvironment.create(
   env,
   environment_settings=EnvironmentSettings.new_instance().use_blink_planne
r().build())

# 创建Kafka数据源表
t_env.sql_update(kafka_source_ddl)
# 创建MySql结果表
t_env.sql_update(mysql_sink_ddl)

# 注册IP转换地区名称的UDF
t_env.register_function("ip_to_province", ip_to_province)

# 添加依赖的Python文件
t_env.add_Python_file(
    os.path.dirname(os.path.abspath(__file__)) + "/enjoyment/cdn/cdn_udf.py
")
t_env.add_Python_file(os.path.dirname(
    os.path.abspath(__file__)) + "/enjoyment/cdn/cdn_connector_ddl.py")

# 核心的统计逻辑
t_env.from_path("cdn_access_log")\
   .select("uuid, "
           "ip_to_province(client_ip) as province, " # IP 转换为地区名称
           "response_size, request_time")\
   .group_by("province")\
   .select( # 计算访问量
           "province, count(uuid) as access_count, "
           # 计算下载总量
           "sum(response_size) as total_download,  "
           # 计算下载速度
           "sum(response_size) * 1.0 / sum(request_time) as download_speed"
) \
   .insert_into("cdn_access_statistic")

# 执行作业
t_env.execute("pyFlink_parse_cdn_log")
```

■ 阿里云 CDN 实时日志分析运行效果

我们采用 mock 的数据向 kafka 发送 CDN 日志数据，右边实时的按地区统计资源的访问量，下载量和下载速度。这个示例的 mock 数据工具，源代码和操作过程，在今天的直播后，会更新到我的博客当中。方便大家在自己的环境中进行体验。

PyFlink 未来，会怎样?

总体来看 PyFlink 的业务开发还是非常简洁的，不用关心底层的实现细节，只需要按照SQL或者Table API的方式描述业务逻辑就行。那么，我们再整体看看PyFlink的未来会怎样呢?

PyFlink 本心驱动 Roadmap

PyFlink 的发展始终要以本心驱动，我们要围绕将现有 Flink 功能输出到 Python 用户，将 Python 生态功能集成到 Flink当中为目标。PyFlink的 Roadmap 如图所示：首先解决 Python VM 和 Java VM 的通讯问题，然后将现有的 Table API 功能暴露给 Python 用户，提供 Python Table API，这也就是 Flink 1.9 中所进行的工作，接下来我们要为将Python功能集成到Flink做准备就是集成 Apache Beam，提供 Python UDF 的执行环境，并增加Python 对其他类库依赖的管理功能，为用户提供 User-defined-Funciton 的接口定义，支持 Python UDF，这就是 Flink 1.10 所做的工作。

为了进一步扩大Python生态的分布式功能，PyFlink 将提供 Pandas 的 Series 和 DataFram 的支持，也就是用户可以在 PyFlink 中直接使用 Pandas 的UDF。同时为增强用户的易用性，让用户有更多的方式使用 PyFlink，后续增加在 Sql Client 中使用 Python UDF。面对 Python 用户的机器学习问题，增加 Python 的 ML pipeline API。监控 Python UDF 的执行情况对，对实际的生产业务非常关键，所以 PyFlink 会增加 Python UDF 的 Metric 管理。这些点将在 Flink 1.11 中将与用户见面。

但这些功能只是 PyFlink 规划的冰山一角，后续我们还要进行性能优化，图计算API，Pandas on Flink 的 Pandas 原生 API 等等。。。进而完成不断将 Flink 现有功能推向 Python 生态，将 Python 生态的强大功能不断集成到 Flink 当中，进而完成 Python 生态分布化的初衷。

PyFlink 1.11 预览

我们快速的预览一下即将与大家见面的 Flink 1.11 中的 PyFlink 的重点内容。

■ 功能

我们将视角由远方拉近到 Flink 1.11 版本 PyFlink 的核心功能，PyFlink 会围绕着 功能，性能和易用性不断努力，在 1.11 在功能上会增加 Pandas UDF 的支持，这样Pandas 生态的实用类库功能可以在 PyFlink 中直接使用，比如累积分布函数， CDF 等。

还会增加 ML Pipeline API 的支持，这样大家可以利用 PyFlink 完成一些机器学习场景的业务需求，我这里是一个使用 pyFlink 完成 KMeans 的示例。

■ 性能

在性能上 PyFlink 也会有更多的投入，我们利用 Codegen，CPython，优化序列化和反序列化的方式提高 PythonUDF 的执行性能，目前我们初步对 1.10 和 1.11 进行性能对比来看，1.11 将比 1.10 有近 15 倍的性能提升。

■ 易用性

在用户的易用性上 PyFlink 会在 SQL DDL 和 SQL Client 中增加对 Python UDF 的支持。让用户有更多的方式选择来使用 PyFlink。

PyFlink 大图（使命愿景）

今天已经介绍了很多，比如什么是 PyFlink，PyFlink 的存在的意义，PyFlink API 架构，UDF 架构，以及架构背后的取舍和现有架构的优势，并介绍了 CDN 的案例，介绍了 PyFlink 的 Roadmap，预览了 Flink 1.11 版本中 PyFlink 的重点，那么接下来还有什么呢?

那么最后我们再来看看 PyFlink 的未来会怎样？在以 “Flink 功能 Python 化，Python 生态分布化” 的使命驱动下，PyFlink 会有怎样的布局？我们快速分享一下：PyFlink 是 Apache Flink 的一部分，涉及到 Runtime 层面和 API 层面。

在这两个层面 PyFlink 会有怎样的发展？ Runtime 层面，PyFlink 会构建解决 Java VM 和 Python VM 的通讯问题的 gRPC 通用服务，比如（Control/Data/State等）在这套框架之上会抽象出 Java 的 Python UDF 算子，Python

的执行容器构建，支持多种 Python 的 Execution，比如 Process，Docker 和 External，尤其值得强调的是 External 以Socket 的方式提供了无限的扩展能力，在后续的 Python 生态集成上至关重要。

API 层面，我们会使命驱动，将 Flink 上所以的 API 进行 Python 化，当然这也依托于引入 Py4J 的 VM 通讯框架之上，PyFlink 会逐渐增加各种 API 的支持，Python Table API，UDX 的接口 API，ML Pipeline，DataStream，CEP，Gelly，State，等Flink所具备的 Java APIs 和 Python 生态用户的最爱 Pandas APIs 等。在这些 API 的基础之上，PyFlink 还会不断的进行生态系统的集成，比如 方便用户开发的 Notebook 的集成，Zeppelin，Jupyter，并与阿里开源的 Alink 进行集成，目前 PyAlink 已经完全应用了 PyFlink 所提供的功能，后面还会和现有的 AI 系统平台进行集成，比如大家熟知的 TensorFlow 等等。

所以此时我会发现使命驱动的力量会让 PyFlink 的生命线不断延续…当然这种生命的延续更需要更多的血液融入。这里再次强调一下 PyFlink 的使命：“Flink 能力 Python 化，Python 生态分布化”。目前 PyFlink 的核心贡献者们正以这样的使命而持续活跃在社区。

PyFlink 核心贡献者及问题支持

在分享的最后，我想介绍一下目前 PyFlink 的核心贡献者。

首先是付典，目前付典是 Flink 以及另外两个 Apache 顶级项目的 Committer，在PyFlink 模块做了巨大的贡献。

第二位是黄兴勃，目前专注 PyFlink 的 UDF 性能优化，曾经是阿里与安全算法挑战赛的冠军，在 AI 和中间件性能比赛中也有很好的成绩。

第三位就是大家熟知的程鹤群，为大家做过多次分享，相信大家还记得他为大家带来的《Flink 知识图谱》分享。

第四位是钟葳，关注 PyFlink 的 UDF 依赖管理和易用性工作，目前已经有很多的代码贡献。最后一个是我自己。大家后续在使用 PyFlink 的时候，如果有什么问题都可以联系我们中的任何一位寻求支持。

当然遇到通用性问题还是建议大家邮件到 Flink 的用户列表和中文用户列表，这样能问题共享。当然如果你遇到特别急的个别问题，也非常欢迎您邮件到刚才介绍的小伙伴邮箱，同时，为了问题的积累和有效的分享，更期望大家遇到问题可以在 Stackoverflow 进行提问题。首先搜索你遇到问题是否已经被解答过，如果没有，请描述清楚，最后提醒大家要为问题打上 PyFlink 的 tags。这样我们及时订阅回复您问题。

总结

今天深入剖析了 PyFlink 深层含义；介绍了 PyFlink API 架构是核心采用 Py4J 框架进行 VM 之间的通讯，API 的设计上 Python API 和 Java API 保持语义的一致；还介绍了 Python UDF 架构，以集成 Apache Beam 的 Portability Framework 的方式获取高效稳定的 Python UDF 的支持，并且细致分析了架构背后思考，对技术选型的取舍和现有架构的优势；

然后介绍了 PyFlink 所适用的业务场景，并以阿里云 CDN 日志实时分析的案例让大家对 PyFlink 的使用有一定的体感；

最后介绍了 PyFlink 的 Roadmap 和预览了 Flink 1.11 版本中 PyFlink 的重点，预期 PyFlink 1.11 相对于1.10会有15倍以上的性能提升，最后和大家一起分享了 PyFlink 的使命，PyFlink 的使命是 ”Flink能力Python化，Python生态分布化”。

留在最后的是提供给大家一种更有效的问题求助的方式，大家有什么问题可以随时抛给刚才向大家介绍的 PyFlink 小伙伴，那么这些小伙伴已经在直播群里了，接下来有什么问题，我们可以一起探讨。：）

作者介绍：

孙金城（金竹），2011 年加入阿里，9 年的阿里工作中，主导过很多内部核心系统，如，阿里集团行为日志，阿里郎，云转码，文档转换等。在 2016 年初开始了解 Apache Flink 社区，由初期的参与社区开发到后来逐渐主导具体模块的开发，到负责 Apache Flink Python API(PyFlink) 的建设。 目前是 PMC member of Apache Flink and ALC(Beijing)， 以及 Committer for Apache Flink, Apache Beam and Apache IoTDB。