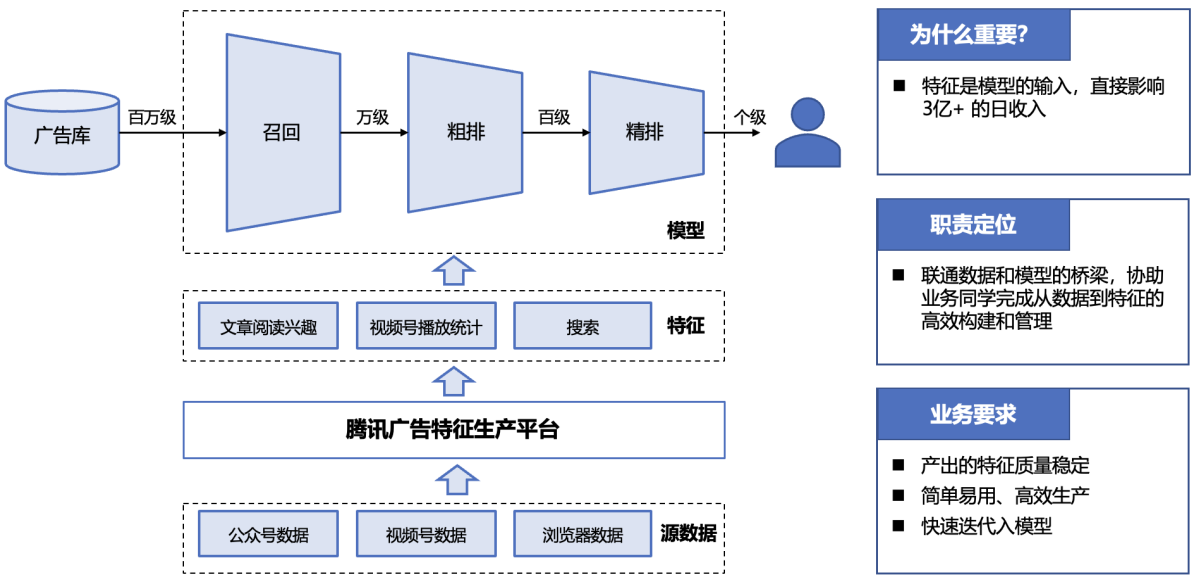
RFC代码串讲

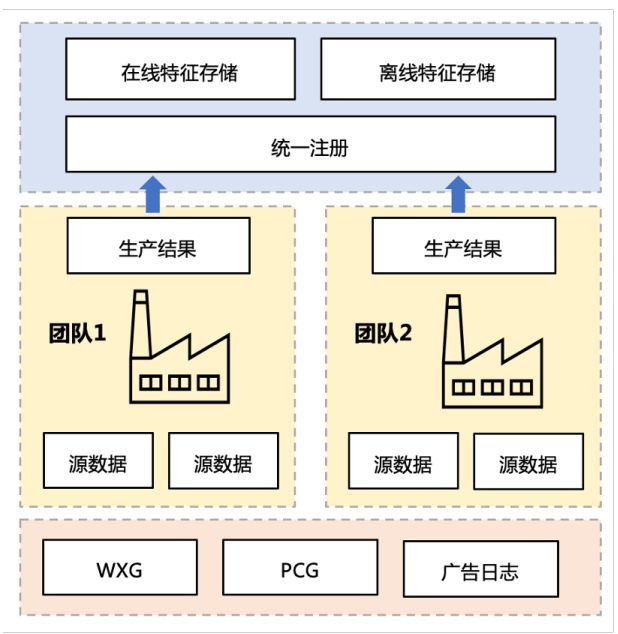
特征生产平台的架构

1. 特征是广告系统的重要组成部分



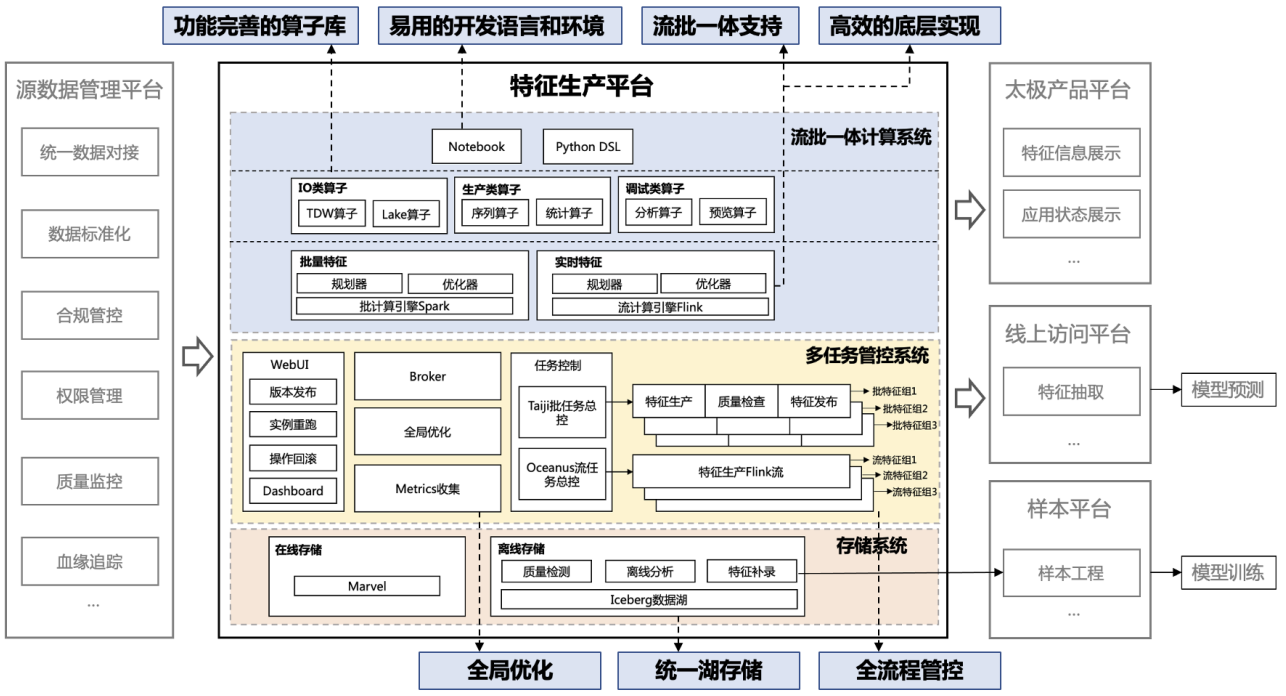
2. 特征生产时产生的问题

- 开发门槛高：业务同学需要学习底层多种大数据引擎，多种语言。
- 开发效率低：业务同学需要自己搭建任务流，需要在多平台切换，开发过程中调试困难。
- 任务性能差：业务同学的最终实现往往并非最优，性能较低。
- 缺乏特征生产的运维与监控工具。



3. 特征生产平台的介绍

特征生产平台通过屏蔽底层大数据组件，提供一站式的 Python NoteBook开发体验、流批一体的计算与存储、丰富高效的特征算子，统一管控特征计算、特征数据与特征应用的全生命流程，让业务同学专注于特征的探索，极大提升生产、调研和应用特征的效率。用户可以基于简单的语法开发特征，并一站式的完成调式、部署，降低了开发门槛，极大提升了开发效率。



上游是源数据，主要为商数接入的数据，包括：

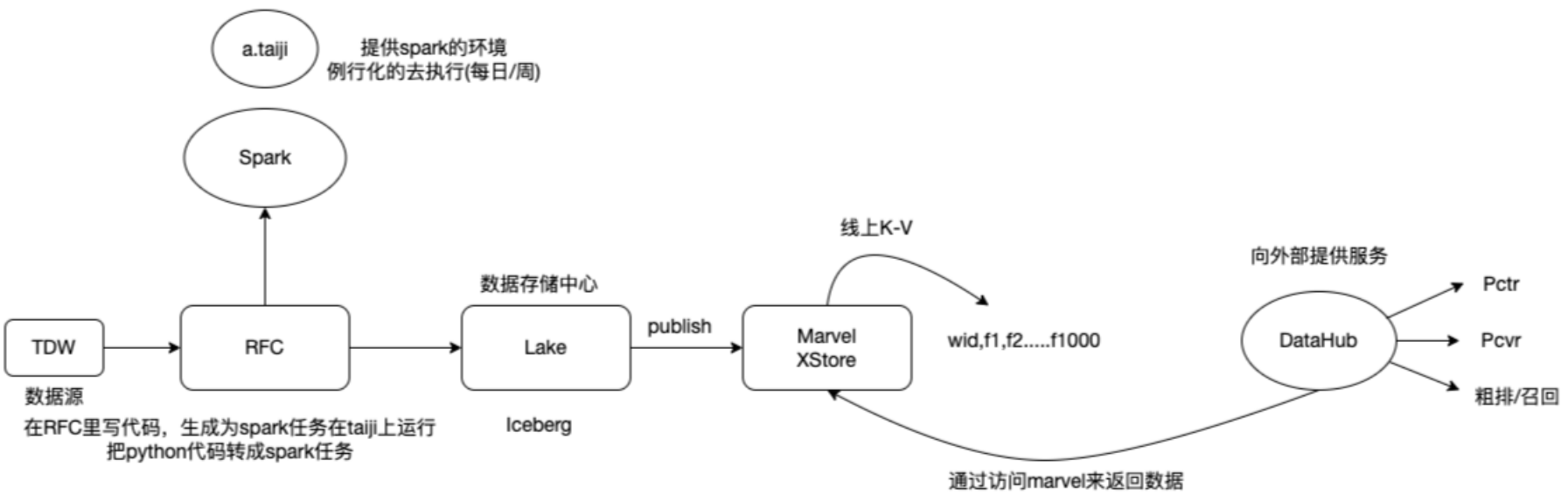
TDW（Tencent distributed Data Warehouse）是腾讯的分布式数据仓库，是一个用于海量数据存储和海量数据分析的分布式数据处理平台，数据通过hive进行存储。

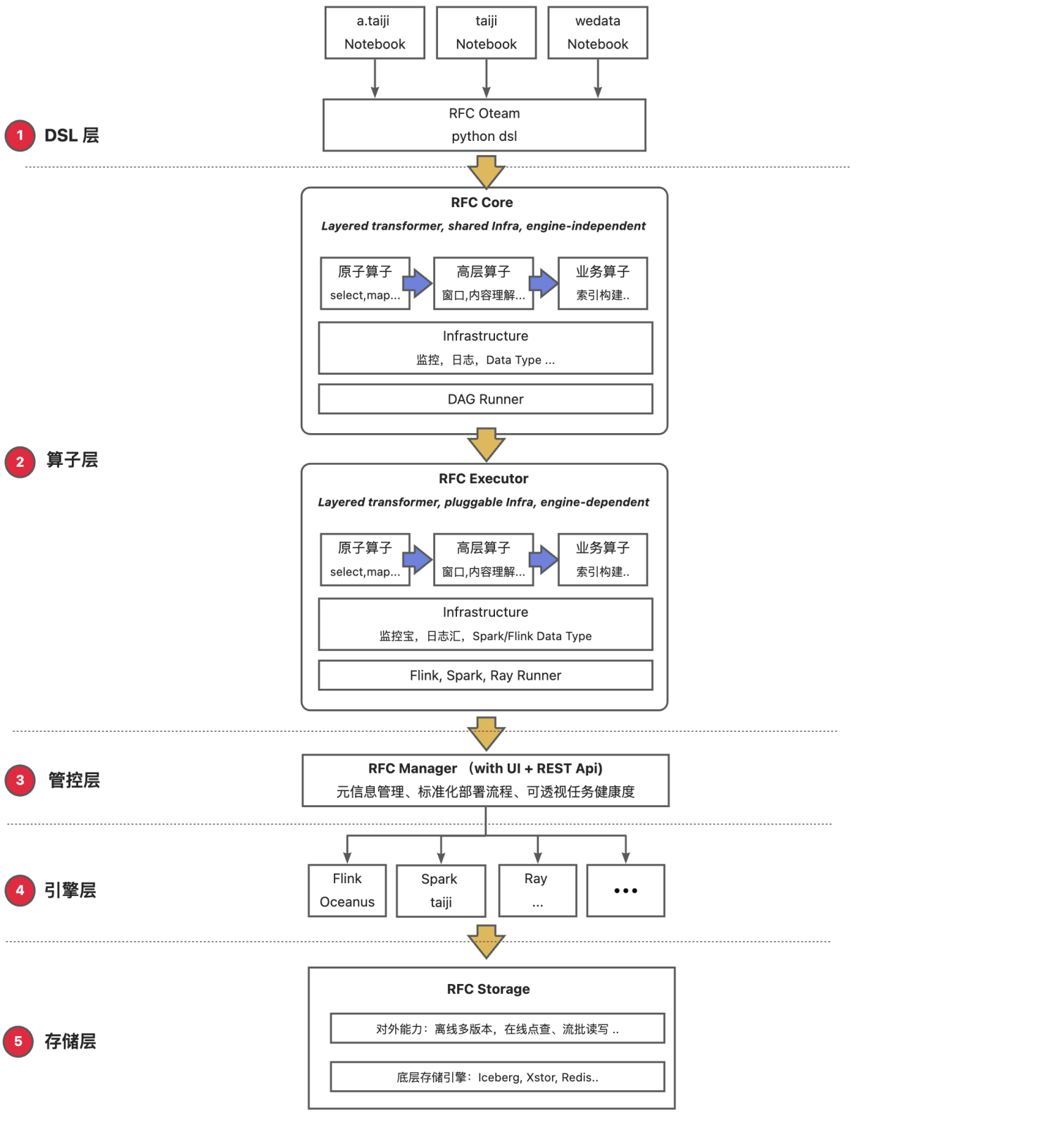
X-sotre等

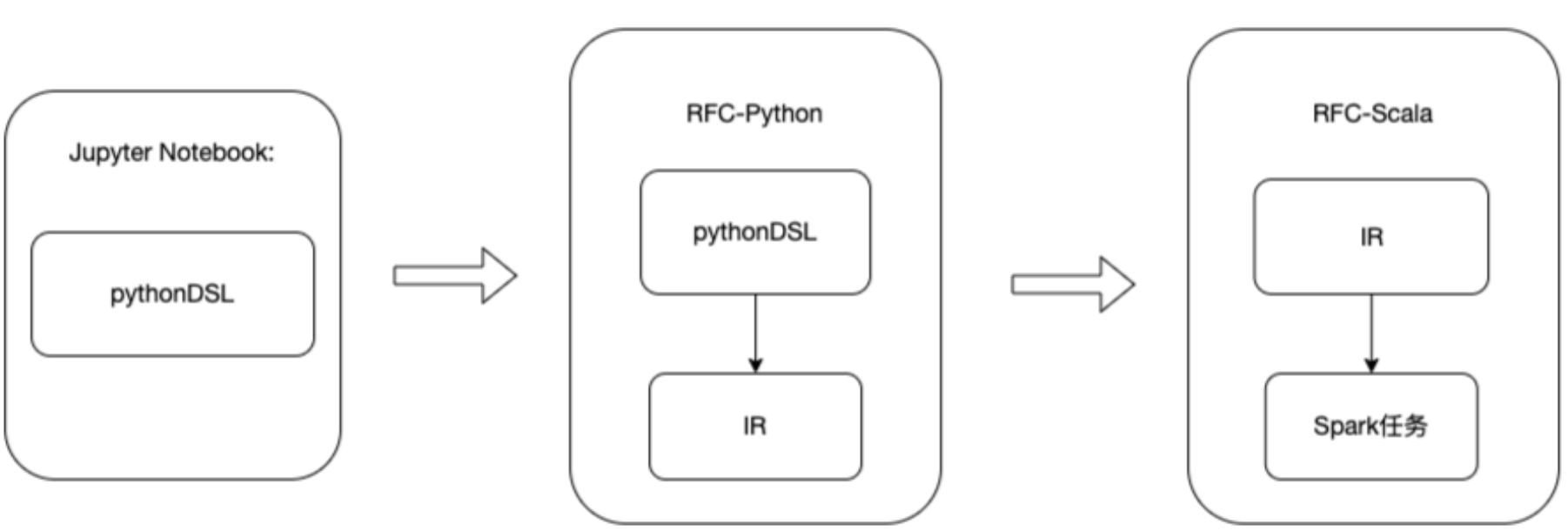
下游对接样本，太极，datahub。

特征生产以后写在marvel和xstore，datahub和样本那边会访问这些数据。线上datahub会落日志给到样本使用，太极给我们提供了spark的运行环境。特征发布以后，样本和datahub会通过线上k-v去访问

4. 整体架构

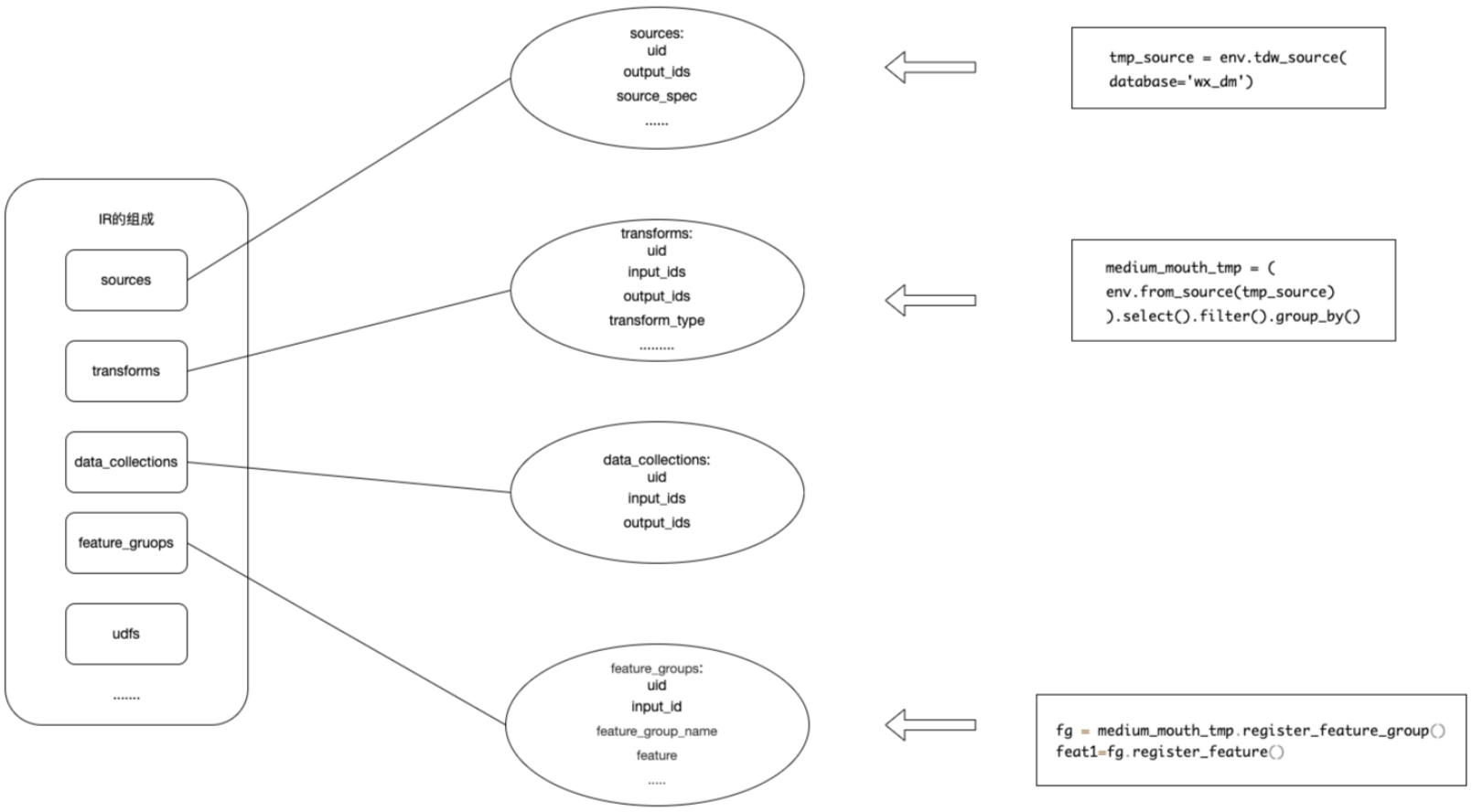






PythonDSL与IR的转换关系

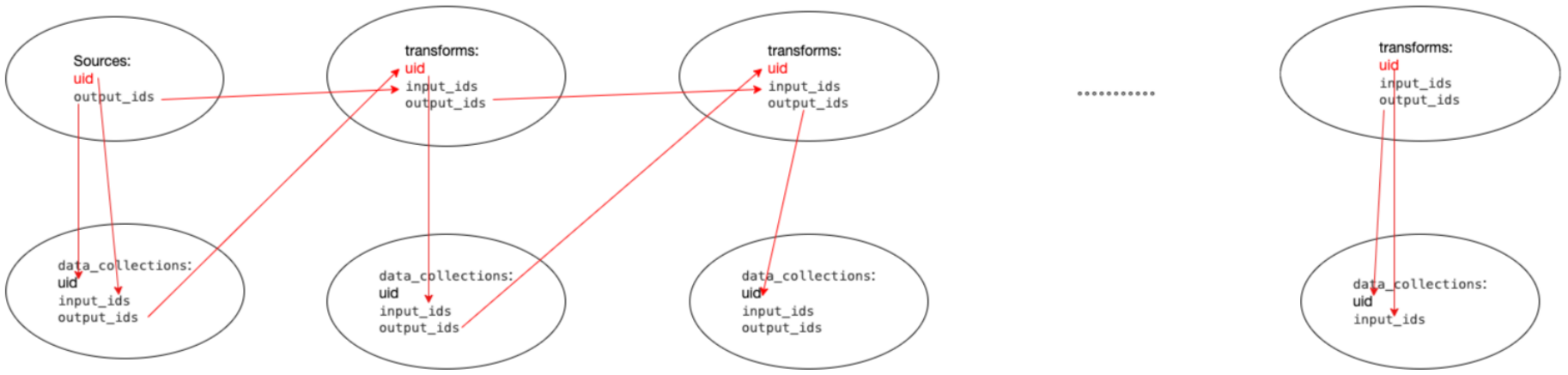
1. IR的组成部分



2. 节点之间的对应关系

每个sources/transforms都会产生一个data_collections，相当于是dataframe的抽象，记录了每次转换后的数据快照。

每个data_collections的uid都是它的对应source/transforms的output_ids，代表每次转换后的结果。input_ids记录当前对应节点的uid，output_ids记录下一个节点的uid。



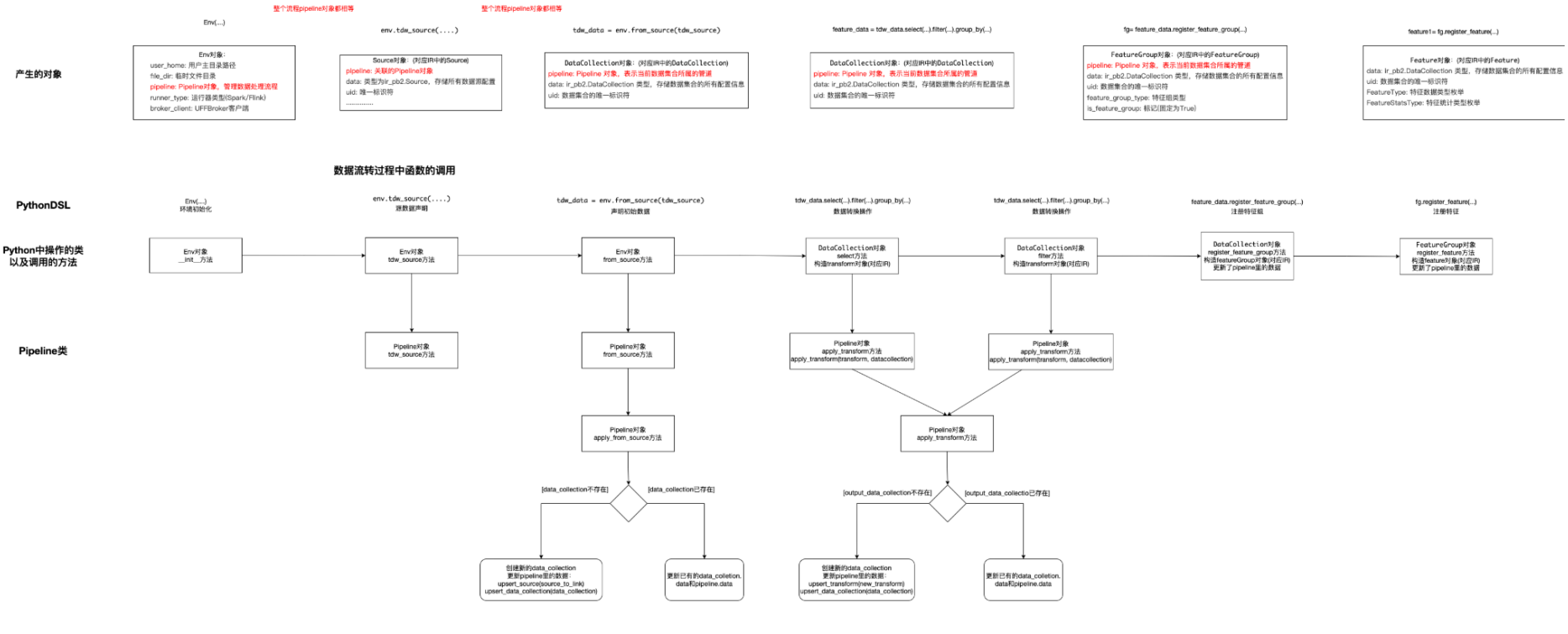
Python代码

1. 主要的类

类名	主要功能	核心成员变量	提供的核心方法：
Env	支持的DSL功能： ● 环境初始化 ● 源数据声明 ● 声明初始数据集	user_home: 用户主目录路径 file_dir: 临时文件目录 pipeline: Pipeline对象，管理数据处理流程 runner_type: 运行器类型(Spark/Flink)	**1. 初始化与配置方法** **2. 数据源操作方法** tdbank_source(): 创建TDBank数据源 from_source(): 从已有数据源创建数据集合 **3. 特征与表管理方法** register_feature_group(): 注册特征组 register_external_data(): 注册外部数据 **4. 部署相关方法** deploy(): 部署特征组到生产环境 **5. Git与版本控制方法** to_git(): 将任务代码提交到Git仓库
Pipeline	大部分DSL操作的底层实现类： ● 数据源操作 ● transform操作	name: 管道名称 data: protobuf格式 pipeline_type: 管道类型	**1. 构造与加载方法** **2. 属性检查方法** is_stream: 检查是否为流式管道 is_feature_group_pipeline(): 检查是否为特征组管道 **3. 管道操作方法** clear(): 清空管道内容 rename(): 重命名管道 prune_to(): 剪枝管道到指定节点 **4. 数据源操作方法** from_source(): 从数据源创建数据收集 tdbank_source(): 创建TDBank数据源 **5. 组件操作方法** upsert_source(): 更新数据源信息 upsert_transform(): 更新转换信息 upsert_feature_group(): 更新特征组信息 **6. 转换应用方法** apply_transform(): 应用转换到数据收集 apply_sink(): 应用sink到数据收集
DataCollection	支持的DSL功能： ● 单表数据转换 ● 多表数据转换 ● 落地中间数据	pipeline: Pipeline 对象，表示当前数据集合所属的管道 data: protobuf格式 uid: 唯一标识符	**1. 数据转换方法** 基础操作 select(columns): 选择指定列 with_column(column): 添加新列 窗口操作

	• 注册特征组 • 注册广播变量		windowed_agg(): 带时间窗口的聚合 集合操作 union(other_data_collection): 并集 连接操作 join(): 表连接 **2. 外部服务集成方法** with_http_request(): HTTP请求 **3. 数据输出方法** save_to_temp_table(): 保存为临时表 register_intermediate_table(): 注册中间表 register_feature_group(): 注册特征组(旧版) register_broadcast(): 注册广播变量
Interactive	支持的DSL功能： • 核心的交互方法：show（ ） • persist（ ）	runner: 运行器实例(SparkRunner或FlinkRunner) current_date source_sample_ratio: 数据采样比例	**1. UffClient 类** 功能：UFF框架的客户端入口，管理运行环境和配置 **2. InteractiveMixin 类** 功能：为UFF组件提供交互式操作的混合类： show(limit, truncate, options, vertical):显示数据内容 persist(options): 缓存数据 unpersist(options): 清除数据缓存
SparkRunner	Spark的主要执行	SparkSession	**1. 初始化方法** **2. 核心运行方法** execute_pipeline(): 执行整个管道 get_data_view(): 获取数据视图 get_raw_df(): 获取原始DataFrame **3. 数据操作方法** persist(): 缓存DataFrame unpersist(): 清除数据缓存 show(): 展示DataFrame内容 **4. UDF相关方法** register_udfs(): 注册所有UDF函数 has_udf(): 检查管道是否包含UDF
Feature	• 定义特征组和特征对象 • 支持特征注册	pipeline: Pipeline 对象，表示当前数据集合所属的管道 data: protobuf格式 uid: 唯一标识符	register_feature(): 注册新特征 analyse(): 执行特征分析
Source	定义了多种Source数据源的对象	pipeline: Pipeline 对象，表示当前数据集合所属的管道 data: protobuf格式 uid: 唯一标识符	UffSerializable └── Source ├── HdfsSource ├── TdwSource ├── KafkaSource ├── TDBankSource ├── AdDwdSource ├── PySparkSource ├── AdEventSource ├── XStorDbSource ├── RainbowSource └── PublishSource
Transformation	定义了特征转换相关的各种类	/	Col - 列表达式 Expr - SQL表达式封装 Agg - 基础聚合(SUM/AVG/COUNT等) WindowedAgg - 窗口聚合(带时间窗口的统计) Join - 表连接操作 FilterConfig - 数据过滤配置 TextSplitItem - 文本分割配置

2. 注册特征组的流程图

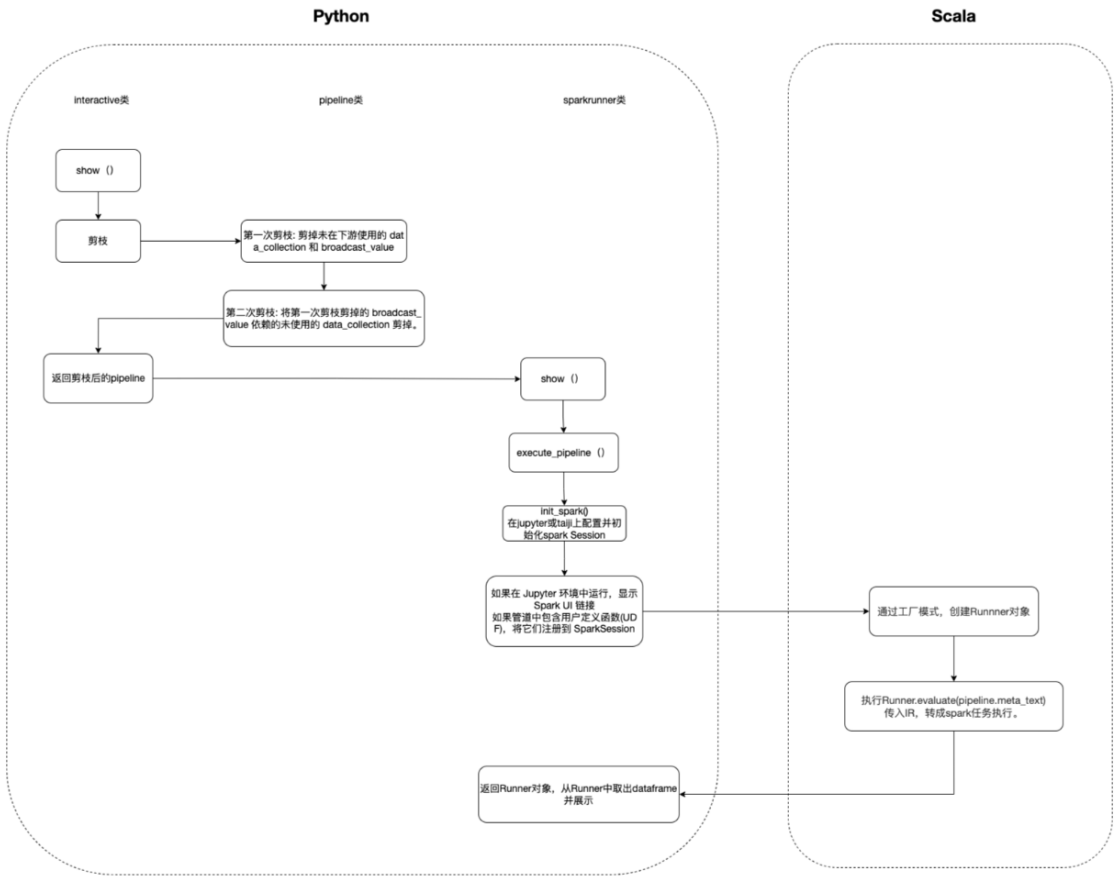


在整个注册特征组的流程中，会产生很多的对象，比如source，data_collection, transforms, frature_group，他们都包含pipeline对象，并且他们每个节点对应的数据都存在data变量里，也就是每个节点ir的信息作为protobuf的格式储存。

ir里的参数来自于用户在notebook里操作写的pythonDSL的方法和参数，然后在python的代码里会存到对应的节点的data里，再整合到pipeline.data里成为完整的ir文件。

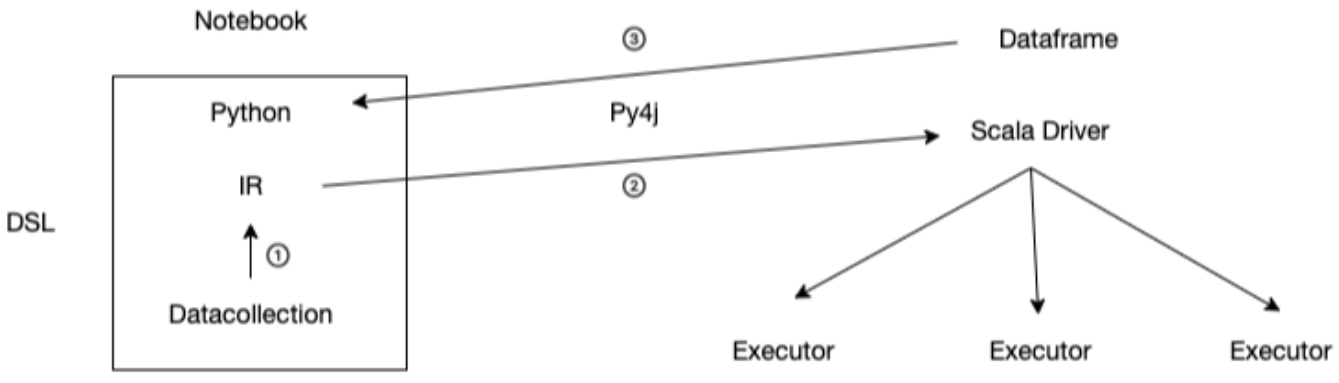
python这块的代码主要是完成了pythonDSL到IR到数据转换。

3. show（ ）方法的执行流程



Python与Scala的交互

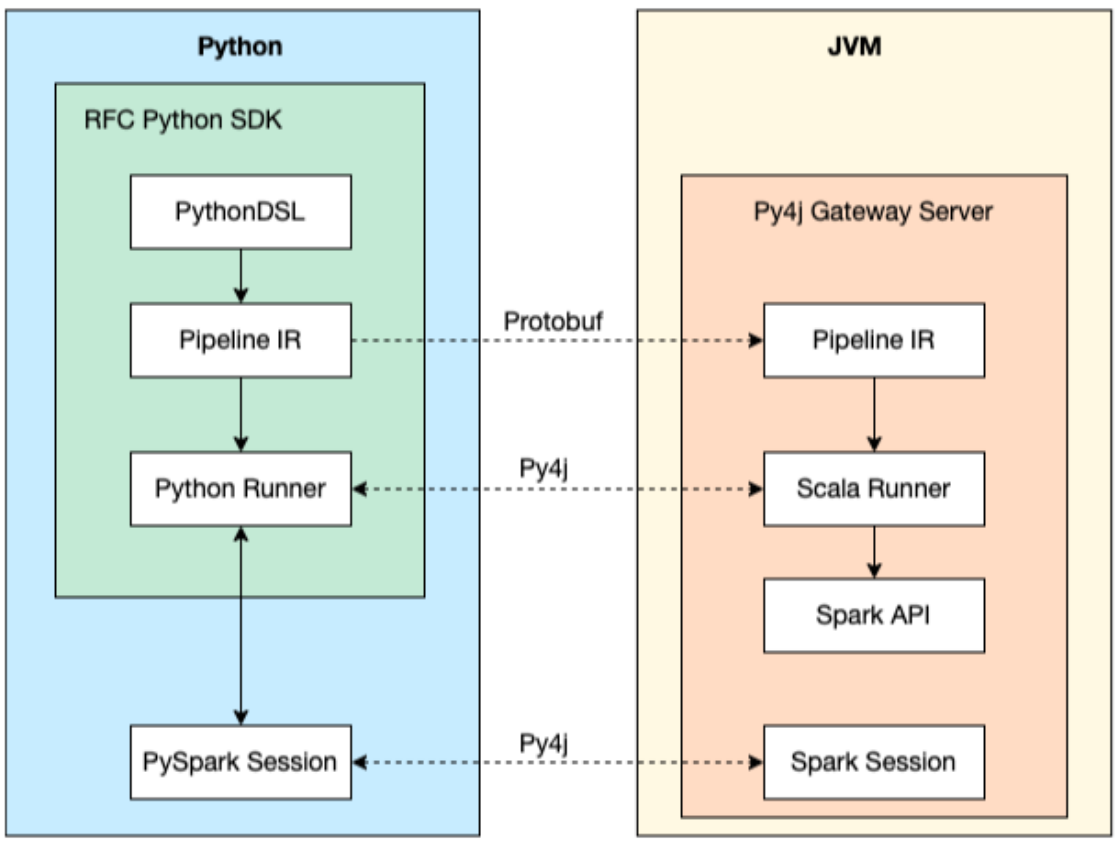
统一特征生产框架的核心组件是Python端的DSL SDK和Scala端的Runner。DSL SDK作为用户入口，预装在太极notebook的Python环境中，用户通过DSL提供的算子定义数据ETL操作，形成算子串联的Pipeline，Pipeline在框架内部通过Protobuf在Python端和Scala端传递。



1.Python DSL，datacollection数据的变换转成IR
2.把IR通过Py4j给到Scala，Scala根据ir执行spark任务，得到dataframe
3.通过Py4j把dataframe传给python，并执行show()

除Python UDF和PySpark Source之外，Python端不直接执行PySpark代码，只负责创建SparkSession，Pipeline解析和执行都在Scala端用DataFrame或RDD API实现，采用这种设计的好处是方便复杂的预置算子（如WndowedAgg）的实现，并且尽可能将执行放到JVM以提高性能。

也就是说提供给用户的界面在Python端，执行在Scala端，又要提供notebook环境交互式开发能力，这就需要解决用Python驱动Scala端执行、获取Scala端DataFrame的问题，所以我们用到了py4j。这样做的好处是，我们不需要自己再走一遍启动py4j GatewayServer、从Python端连接它的过程，启动了PySpark的SparkSessison之后，可以复用Spark的py4j JavaGateway。这样我们就能在Python端调用Scala实现的Runner执行DSL定义的Pipeline，并获取返回的DataFrame：



Scala代码

1. 核心的类

1.1 Runner类

这个类是数据处理流水线的执行框架核心，将逻辑计划(IR)转换为物理执行，同时提供执行过程中的状态管理和结果访问能力。
成员变量

runnerContext: RunnerContext类型
作用：存储运行时的上下文信息，包括当前组件、数据视图等
用途：在整个执行流程中传递和共享状态
logger: Logger类型
作用：用于记录日志信息
用途：记录执行过程中的调试和错误信息
options: Map[String, String]类型（通过构造函数传入）
作用：存储配置选项
用途：控制运行时的各种参数和行为

主要方法

evaluate方法（3个重载）
evaluate(ir: Array[Byte]): 从字节数组创建并执行pipeline
evaluate(irStr: String): 从字符串创建并执行pipeline
protected evaluate(pipeline: Pipeline): 核心执行逻辑
作用：执行整个数据处理流程
实现：遍历pipeline节点，更新payload并执行评估
数据视图获取方法
getDataView(id: String): 获取指定或当前输出数据视图
getSinkView(id: String): 获取指定或当前sink视图
getFeatureGroup(): 获取特征组视图

解决的问题

1. 统一执行框架
   - 为不同执行引擎(如Spark批处理、Flink流处理)提供统一的执行接口
   - 封装了从IR到实际执行的转换过程
2. 数据处理流水线管理
   - 提供创建、优化和执行数据处理流水线的基础设施
   - 管理数据处理过程中的节点遍历和执行顺序
3. 执行状态跟踪
   - 通过runnerContext维护执行状态
4. 数据视图访问
   - 提供统一的数据视图访问接口
   - 简化最终结果的获取方式

1.2 RunnerContext类

RunnerContext 类是特征处理流水线的执行上下文管理器。在保持低耦合的同时，为特征处理流水线提供了统一的状态管理和数据访问抽象。

成员变量

dataViewMap: Map[String, DataView]	存储所有数据视图（节点的UID -> DataView）
currentComponent: Node	当前正在执行的节点
featureGroup: DataView	特征组计算结果
sinkViewMap: Map[String, DataView]	存储输出视图（节点ID -> DataView）

解决的问题

1. 数据流状态维护
   - 数据视图存储：通过 dataViewMap 集中管理所有中间计算结果（DataView），以数据UID为键实现快速存取
   - 当前节点追踪：通过 currentComponent标记正在执行的节点，确保执行上下文一致性
2. 结果数据管理
   - FeatureGroup 专用于最终特征结果的存储和获取

1.3 Pipeline类

Pipeline是对IR的解析和封装，其将IR的信息组织为一个DAG的形式。由于IR本身已经描述了执行语义和依赖顺序，因此Pipeline也可看作是一种逻辑计划的表达。

成员变量

核心属性：
name: String - Pipeline名称
nodeMap: Map[String, Node] - 节点ID到Node对象的映射表
beginNodeIds: List[String] - DAG的起始节点ID列表
节点管理：
orderedNodeIds: mutable.ListBuffer[String] - 按拓扑排序的节点ID列表
excludedNodeIds: mutable.ListBuffer[String] - 需要排除的节点ID列表
功能组件：
udfs: List[Udf] - 用户定义函数列表

scalaUdfs: List[ScalaUdf] - Scala实现的UDF列表
externalDatas: List[ExternalData] - 外部数据源列表

主要方法

traverse(visitor: AbstractVisitor) - 使用访问者模式遍历所有节点
build() - 构建Pipeline的DAG结构
getOrderedNodeIds() - 获取拓扑排序后的节点ID列表

解决的问题

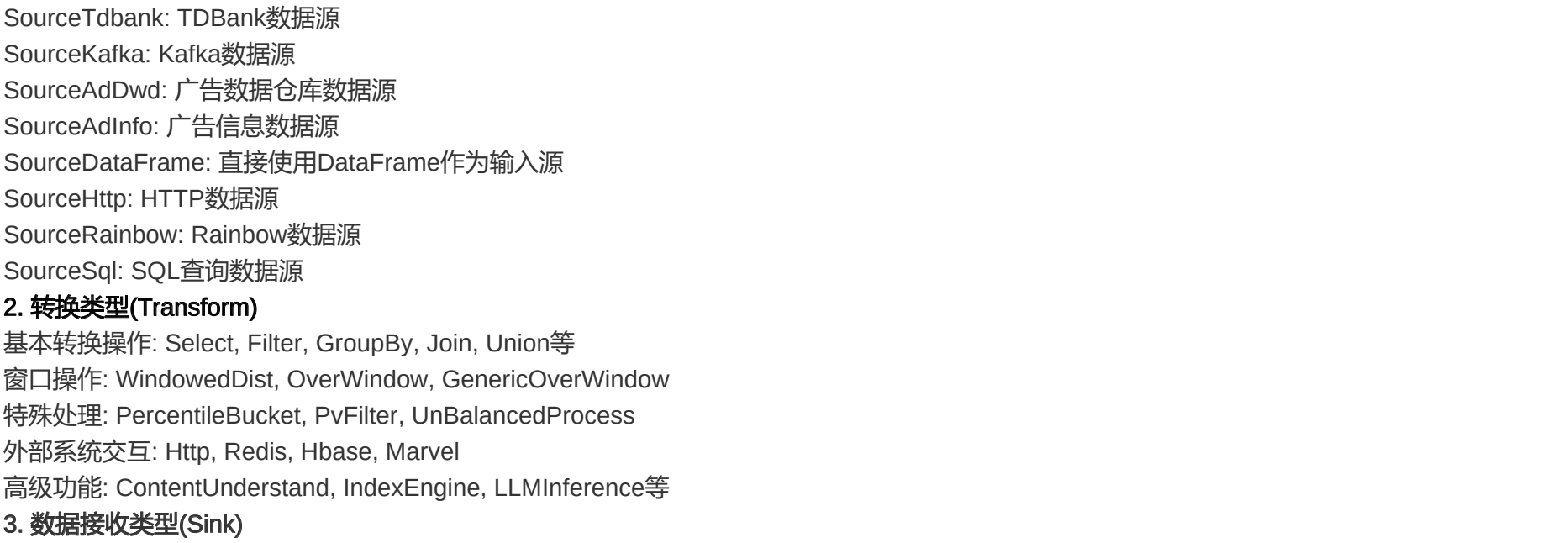
1. IR到DAG的转换
   - IR解析为可执行的DAG结构
   - 通过build()实现拓扑排序，确保依赖顺序正确
2. 提供标准化的traverse()接口，支持PayloadUpdator等不同访问者实现

1.4 Node类

用于表示数据处理流程中不同类型的节点。主要分为三类：数据源(Source)、转换(Transform)和数据接收(Sink)。与ir的对应，source和transform和sink节点，和node是一一对应的。

1. 数据源类型(Source)
SourceHdfs: HDFS数据源
SourceXStorDb: XStor数据库数据源
SourceTdw: TDW数据源
SourceTdbank: TDBank数据源
SourceKafka: Kafka数据源
SourceAdDwd: 广告数据仓库数据源
SourceAdInfo: 广告信息数据源
SourceDataFrame: 直接使用DataFrame作为输入源
SourceHttp: HTTP数据源
SourceRainbow: Rainbow数据源
SourceSql: SQL查询数据源
2. 转换类型(Transform)
基本转换操作: Select, Filter, GroupBy, Join, Union等
窗口操作: WindowedDist, OverWindow, GenericOverWindow
特殊处理: PercentileBucket, PvFilter, UnBalancedProcess
外部系统交互: Http, Redis, Hbase, Marvel
高级功能: ContentUnderstand, IndexEngine, LLMInference等
3. 数据接收类型(Sink)
存储到不同系统: Lake, MQ, Tdbank, Hdfs
特征视图: FeatureGroup, UserFeatureView, AdFeatureView
特殊视图: LevelTagView, UserStrategyView
虚拟接收器: Empty, DummyView

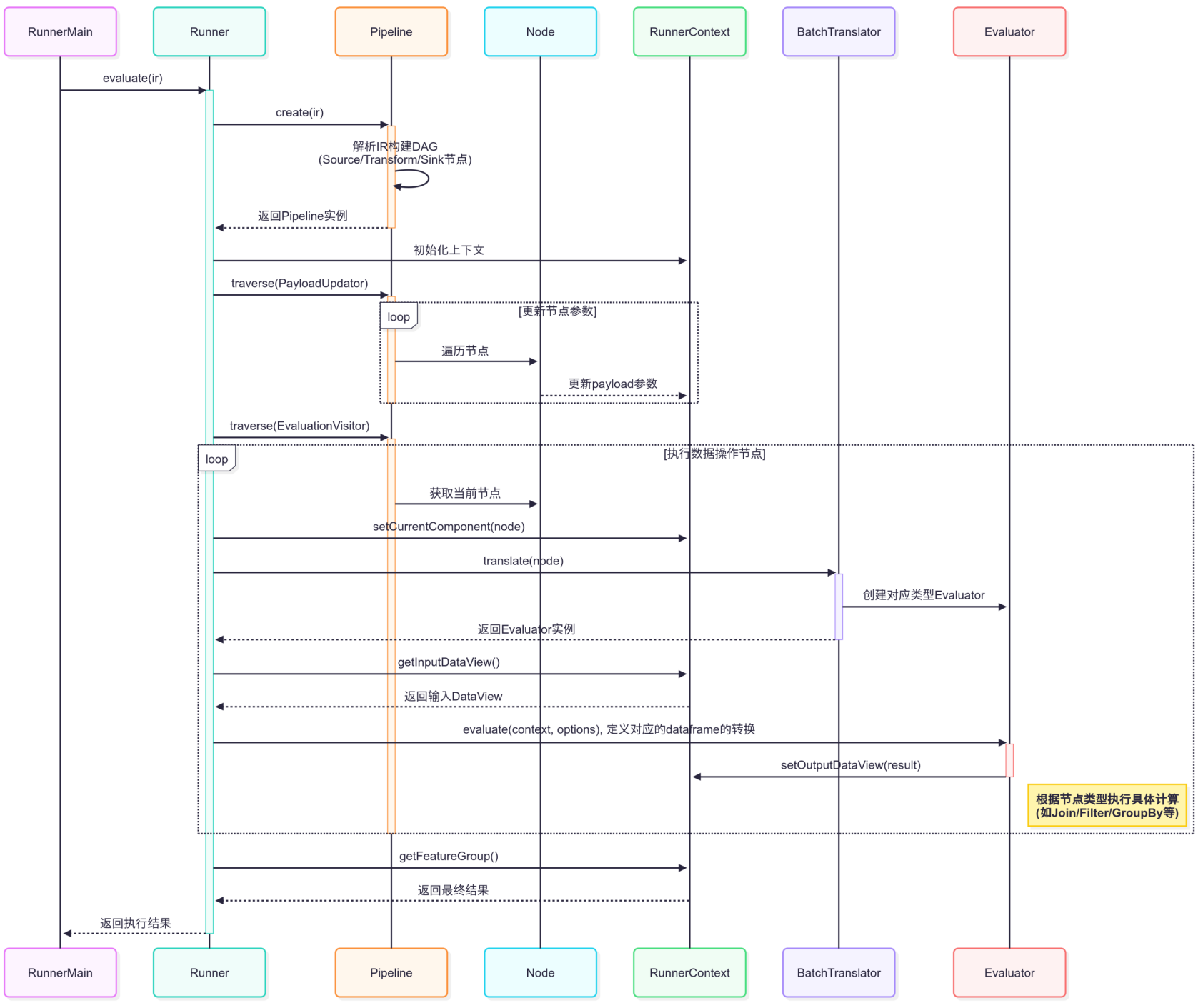
2. 类之间的关系图



Runner 是执行入口，负责协调整个流程
Runner 通过 Pipeline.create() 创建 Pipeline 实例
Pipeline 由多个 Node 组成DAG结构
Runner 通过 RunnerContext 管理执行状态和数据

Runner (执行器)：
抽象基类，提供批处理/流式等不同实现
核心方法evaluate()执行流程：
1. 创建Pipeline
2. 用PayloadUpdator更新节点参数
3. 用EvaluationVisitor遍历执行DAG
Pipeline (执行管道)：
从IR协议数据构建DAG结构
包含三类节点：
SourceNode -> TransformNode -> SinkNode
通过traverse()方法支持访问者模式遍历
RunnerContext (执行上下文)：
维护关键运行时数据：
dataViewMap: 存储中间计算结果
currentComponent: 当前执行的Node
featureGroup: 特征组输出
提供数据视图的存取接口
Node (执行节点)：
基础抽象表示DAG中的节点
通过NodeType区分为：
47种Source类型 | 32种Transform类型 | 12种Sink类型

3. Scala执行流程梳理



1. Runner创建Pipeline
2. Pipeline构建Node的DAG结构
3. Runner遍历Pipeline时：
   a) 通过RunnerContext获取输入DataView
   b) 执行当前Node计算
   c) 将结果存回RunnerContext

初始化阶段：
RunnerMain通过evaluate(ir)启动流程
Pipeline.create()解析IR协议，构建包含Source/Transform/Sink节点的DAG
参数准备阶段：
PayloadUpdator遍历所有节点，更新运行时参数（如时间参数替换）
执行阶段：
EvaluationVisitor按拓扑顺序遍历节点
每个节点通过BatchTranslator转换为对应的Evaluator实现
执行时从RunnerContext获取输入数据，计算结果存回上下文
结果输出：
最终通过getFeatureGroup()获取特征组计算结果
执行状态和指标通过InfoReporter上报

数据流关键点：
节点执行顺序：由Pipeline的orderedNodeIds保证拓扑顺序
数据传递：通过RunnerContext的dataViewMap维护中间结果
类型转换：BatchTranslator将91种NodeType映射到具体Evaluator实现

项目还可以优化的地方：

代码层面：

- python代码比较杂乱，比如show（ ），有很大一块逻辑处理pyspark source，但是现在已经没有用了。详细了解后发现，这块之前设计的时候是想暴露一些pyspark的接口出去，但是现在没有人用。
- 完善单元测试

功能层面：

- 优化算子的实现：join算子，select算子，窗口算子
- 监控指标太多，业务指标和系统指标混在一起，可以对监控指标进行分组，构建关键视图
- 文档建设
- 数据发布的拓展性

最开始设计RFC是只处理特征数据，后来策略数据和索引也加过来了。sinkView类的evaluator有很大的提升空间，这块有很多业务交织在一起了。只考虑特征生产的场景时，在用户侧在DSL的封装比较固定，比如register_feature_group包含了上游dataframe的信息，写到哪个表或者k-v这些都写的比较死，如果以后要支持更多数据发布的类型，需要考虑把格式转换，和数据存储，拆出来建立独立的接口。