Alink漫谈(二)：从源码看机器学习平台Alink设计和架构

目录

0x00 摘要

Alink 是阿里巴巴基于实时计算引擎 Flink 研发的新一代机器学习算法平台，是业界首个同时支持批式算法、流式算法的机器学习平台。本文是漫谈系列的第二篇，将从源码入手，带领大家具体剖析Alink设计思想和架构为何。

因为Alink的公开资料太少，所以均为自行揣测，肯定会有疏漏错误，希望大家指出，我会随时更新。

0x01 Alink设计原则

前文中 Alink漫谈(一)：从KMeans算法实现看Alink设计思想 我们推测总结出Alink部分设计原则

- 算法的归算法，Flink的归Flink，尽量屏蔽AI算法和Flink之间的联系。
- 采用最简单，最常见的开发语言和思维方式。
- 尽量借鉴市面上通用的机器学习设计思路和开发模式，让开发者无缝切换。
- 构建一套战术打法（middleware或adapter），即屏蔽了Flink，又可以利用好Flink，还能让用户快速开发算法。

下面我们就针对这些设计原则，从上至下看看Alink如何设计自己这套战术打法。

为了能让大家更好理解，先整理一个概要图。因为Alink系统主要可以分成三个层面(顶层流水线, 中间层算法组件, 底层迭代计算框架)，再加上一个Flink runtime，所以下图就是分别从这四个层面出发来看程序执行流程。

```
如何看待 pipeline.fit(data).transform(data).print();

// 从顶层流水线角度看
训练流水线  +-----> [VectorAssembler(Transformer)] -----> [KMeans(Estimator)]
           |       // KMeans.fit之后，会生成一个KMeansModel用来转换
           |
转换流水线  +-----> [VectorAssembler(Transformer)] -----> [KMeansModel(Transformer)]


// 从中间层算法组件角度看
训练算法组件 +-----> [MapBatchOp] -----> [KMeansTrainBatchOp]
            |        // VectorAssemblerMapper in MapBatchOp 是业务逻辑
            |
转换算法组件 +-----> [MapBatchOp] -----> [ModelMapBatchOp]
                    // VectorAssemblerMapper in MapBatchOp 是业务逻辑
                    // KMeansModelMapper in ModelMapBatchOp 是业务逻辑


// 从底层迭代计算框架角度看
训练by框架 +-----> [VectorAssemblerMapper] -----> [KMeansPreallocateCentroid / KMeansAssignClust
er / AllReduce / KMeansUpdateCentroids in IterativeComQueue]
           |        // 映射到Flink的各种算子进行训练
           |
转换(直接) +-----> [VectorAssemblerMapper] -----> [KMeansModelMapper]
                    // 映射到Flink的各种算子进行转换

// 从Flink runtime角度看
训练 +-----> map, mapPartiiton...
     |       // VectorAssemblerMapper.map等会被调用
     |
转换 +-----> map, mapPartiiton...
             // 比如调用 KMeansModelMapper.map 来转换
```

0x02 Alink实例代码

示例代码还是用之前的KMeans算法部分模块。

算法调用

```
public class KMeansExample {
            public static void main(String[] args) throws Exception {
        ......

        BatchOperator data = new CsvSourceBatchOp().setFilePath(URL).setSchemaStr(SCHEMA_STR);

        VectorAssembler va = new VectorAssembler()
            .setSelectedCols(new String[]{"sepal_length", "sepal_width", "petal_length", "petal
_width"})
            .setOutputCol("features");

        KMeans kMeans = new KMeans().setVectorCol("features").setK(3)
            .setPredictionCol("prediction_result")
            .setPredictionDetailCol("prediction_detail")
            .setReservedCols("category")
            .setMaxIter(100);

        Pipeline pipeline = new Pipeline().add(va).add(kMeans);
        pipeline.fit(data).transform(data).print();
    }
}
```

算法主函数

```
public final class KMeansTrainBatchOp extends BatchOperator <KMeansTrainBatchOp>
        implements KMeansTrainParams <KMeansTrainBatchOp> {

        static DataSet <Row> iterateICQ(...省略...) {

                return new IterativeComQueue()
                        .initWithPartitionedData(TRAIN_DATA, data)
                        .initWithBroadcastData(INIT_CENTROID, initCentroid)
                        .initWithBroadcastData(KMEANS_STATISTICS, statistics)
                        .add(new KMeansPreallocateCentroid())
                        .add(new KMeansAssignCluster(distance))
                        .add(new AllReduce(CENTROID_ALL_REDUCE))
                        .add(new KMeansUpdateCentroids(distance))
                        .setCompareCriterionOfNode0(new KMeansIterTermination(distance, tol))
                        .closeWith(new KMeansOutputModel(distanceType, vectorColName, latitudeC
olName, longitudeColName))
                        .setMaxIter(maxIter)
                        .exec();
        }
}
```

算法模块举例

基于点计数和坐标，计算新的聚类中心。

```
// Update the centroids based on the sum of points and point number belonging to the same clust
er.
public class KMeansUpdateCentroids extends ComputeFunction {
    @Override
    public void calc(ComContext context) {

        Integer vectorSize = context.getObj(KMeansTrainBatchOp.VECTOR_SIZE);
        Integer k = context.getObj(KMeansTrainBatchOp.K);
```

```
        double[] sumMatrixData = context.getObj(KMeansTrainBatchOp.CENTROID_ALL_REDUCE);

        Tuple2<Integer, FastDistanceMatrixData> stepNumCentroids;
        if (context.getStepNo() % 2 == 0) {
            stepNumCentroids = context.getObj(KMeansTrainBatchOp.CENTROID2);
        } else {
            stepNumCentroids = context.getObj(KMeansTrainBatchOp.CENTROID1);
        }

        stepNumCentroids.f0 = context.getStepNo();
        context.putObj(KMeansTrainBatchOp.K,
            updateCentroids(stepNumCentroids.f1, k, vectorSize, sumMatrixData, distance));
    }
}
```

0x03 顶层 -- 流水线

本部分实现的设计原则是 ： 尽量借鉴市面上通用的设计思路和开发模式，让开发者无缝切换。

1. 机器学习重要概念

一个典型的机器学习过程从数据收集开始，要经历多个步骤，才能得到需要的输出。这非常类似于流水线式工作，即通常会包含源数据ETL（抽取、转化、加载），数据预处理，指标提取，模型训练与交叉验证，新数据预测等步骤。

先来说一下几个重要的概念：

- Transformer：转换器，是一种可以将一个数据转换为另一个数据的算法。比如一个模型就是一个 Transformer。它可以把一个不包含转换标签的测试数据集 打上标签，转化成另一个包含转换标签的特征数据。Transformer可以理解为特征工程，即：特征标准化、特征正则化、特征离散化、特征平滑、onehot编码等。该类型有一个transform方法，用于fit数据之后，输入新的数据，进行特征变换。
- Estimator：评估器，它是学习算法或在训练数据上的训练方法的概念抽象。所有的机器学习算法模型，都被称为估计器。在 Pipeline 里通常是被用来操作 数据并生产一个 Transformer。从技术上讲，Estimator实现了一个方法fit()，它接受一个特征数据并产生一个转换器。比如一个随机森林算法就是一个 Estimator，它可以调用fit()，通过训练特征数据而得到一个随机森林模型。
- PipeLine：工作流或者管道。工作流将多个工作流阶段（转换器和估计器）连接在一起，形成机器学习的工作流，并获得结果输出。
- Parameter：Parameter 被用来设置 Transformer 或者 Estimator 的参数。

2. Alink中概念实现

从 Alink的目录结构中 ，我们可以看出，Alink提供了这些常见概念（其中有些代码借鉴了Flink ML）。

```
./java/com/alibaba/alink:
common          operator        params          pipeline

./java/com/alibaba/alink/params:
associationrule evaluation      nlp             regression      statistics
classification  feature         onlinelearning  shared          tuning
clustering      io              outlier         similarity      udf
dataproc        mapper          recommendation  sql             validators

./java/com/alibaba/alink/pipeline:
EstimatorBase.java      ModelBase.java          Trainer.java            feature
LocalPredictable.java   ModelExporterUtils.java TransformerBase.java    nlp
LocalPredictor.java     Pipeline.java           classification          recommendation
MapModel.java           PipelineModel.java      clustering              regression
MapTransformer.java     PipelineStageBase.java  dataproc                tuning
```

比较基础的是三个接口：PipelineStages，Transformer，Estimator，分别恰好对应了机器学习的两个通用概念 ： 转换器 ，评估器。PipelineStages是这两个的基础接口。

```
// Base class for a stage in a pipeline. The interface is only a concept, and does not have any
 actual functionality. Its subclasses must be either Estimator or Transformer. No other classes
 should inherit this interface directly.
public interface PipelineStage<T extends PipelineStage<T>> extends WithParams<T>, Serializable

// A transformer is a PipelineStage that transforms an input Table to a result Table.
public interface Transformer<T extends Transformer<T>> extends PipelineStage<T>

// Estimators are PipelineStages responsible for training and generating machine learning model
s.
public interface Estimator<E extends Estimator<E, M>, M extends Model<M>> extends PipelineStage
<E>
```

其次是三个抽象类定义：PipelineStageBase，EstimatorBase，TransformerBase，分别就对应了以上的三个接口。其中定义了一些基础操作，比如 fit，transform。

```
// The base class for a stage in a pipeline, either an EstimatorBase or a TransformerBase.
public abstract class PipelineStageBase<S extends PipelineStageBase<S>>
    implements WithParams<S>, HasMLEnvironmentId<S>, Cloneable

// The base class for estimator implementations.
public abstract class EstimatorBase<E extends EstimatorBase<E, M>, M extends ModelBase<M>>
    extends PipelineStageBase<E> implements Estimator<E, M>

// The base class for transformer implementations.
public abstract class TransformerBase<T extends TransformerBase<T>>
    extends PipelineStageBase<T> implements Transformer<T>
```

然后是Pipeline基础类，这个类就可以把Transformer，Estimator联系起来 。

```
// A pipeline is a linear workflow which chains EstimatorBases and TransformerBases to execute
an algorithm
public class Pipeline extends EstimatorBase<Pipeline, PipelineModel> {
        private ArrayList<PipelineStageBase> stages = new ArrayList<>();

        public Pipeline add(PipelineStageBase stage) {
                this.stages.add(stage);
                return this;
        }
}
```

最后是 Parameter 概念相关举例，比如实例中用到的 VectorAssemblerParams。

```
// Parameters for MISOMapper.
public interface MISOMapperParams<T> extends HasSelectedCols <T>,  HasOutputCol <T>,
        HasReservedCols <T> {}

// parameters of vector assembler.
public interface VectorAssemblerParams<T> extends MISOMapperParams<T> {
ParamInfo <String> HANDLE_INVALID = ParamInfoFactory
                .createParamInfo("handleInvalid", String.class)
                .setDescription("parameter for how to handle invalid data (NULL values)")
                .setHasDefaultValue("error")
                .build();
}
```

综合来说，因为模型和数据，在Alink运行时候，都统一转化为Table类型，所以可以整理如下：

- Transformer：将input table转换为output table。
- Estimator：将input table转换为模型。
- 模型：将input table转换为output table。

3. 结合实例看流水线

首先是一些基础抽象类，比如：

- MapTransformer是 flat map 的Transformer。
- ModelBase是模型定义，也是一个Transformer。
- Trainer是训练模型定义，是EstimatorBase。

```
// Abstract class for a flat map TransformerBase.
public abstract class MapTransformer<T extends MapTransformer <T>>
                extends TransformerBase<T> implements LocalPredictable {

// The base class for a machine learning model.
public abstract class ModelBase<M extends ModelBase<M>> extends TransformerBase<M>
    implements Model<M>

// Abstract class for a trainer that train a machine learning model.
public abstract class Trainer<T extends Trainer <T, M>, M extends ModelBase<M>>
        extends EstimatorBase<T, M>
```

然后就是我们实例用到的两个类型定义。

- KMeans 是一个Trainer，其实现了EstimatorBase；
- VectorAssembler 是一个TransformerBase。

```
// 这是一个 EstimatorBase 类型
public class KMeans extends Trainer <KMeans, KMeansModel> implements
        KMeansTrainParams <KMeans>, KMeansPredictParams <KMeans> {
        @Override
        protected BatchOperator train(BatchOperator in) {
                return new KMeansTrainBatchOp(this.getParams()).linkFrom(in);
        }
}

// 这是一个 TransformerBase 类型
public class VectorAssembler extends MapTransformer<VectorAssembler>
        implements VectorAssemblerParams <VectorAssembler> {
        public VectorAssembler(Params params) {
                super(VectorAssemblerMapper::new, params);
        }
}
```

实例中，分别构建了两个流水线阶段，然后这两个实例就被链接到流水线上。

```
VectorAssembler va = new VectorAssembler()
KMeans kMeans = new KMeans()
Pipeline pipeline = new Pipeline().add(va).add(kMeans);

// 能看出来，流水线上有两个阶段，分别是VectorAssembler和KMeans。

pipeline = {Pipeline@1201}
 stages = {ArrayList@2853}  size = 2
```

```
  0 = {VectorAssembler@1199}
   mapperBuilder = {VectorAssembler$lambda@2859}
   params = {Params@2860} "Params {outputCol="features", selectedCols=["sepal_length","sepal_wi
dth","petal_length","petal_width"]}"

  1 = {KMeans@1200}
   params = {Params@2857} "Params {vectorCol="features", maxIter=100, reservedCols=["category"]
, k=3, predictionCol="prediction_result", predictionDetailCol="prediction_detail"}"
```

0x04 中间层 -- 算法组件

算法组件是中间层的概念，可以认为是真正实现算法的模块/层次。主要作用是承上启下。

- 其上层是流水线各个阶段，**流水线的生成结果就是一个算法组件**。算法组件的作用是把流水线的Estimator或者Transformer翻译成具体算法。算法组件彼此是通过 linkFrom 串联在一起。
- 其下层是"迭代计算框架"，算法组件把具体算法逻辑中的计算/通信分成一个个小模块，映射到Mapper Function 或者具体"迭代计算框架"的计算/通信 Function 上，这样才能更好的利用Flink的各种优势。
- "迭代计算框架" 中，主要两个部分是 Mapper Function 和 计算/通信 Function，其在代码中分别对应 Mapper，ComQueueItem。
- Mapper Function 是映射Function（系统写好了部分Mapper，用户也可以根据算法来写自己的 Mapper）；
- 计算/通信 Function是专门为算法写的专用Function（也分成 系统内置的，算法自定义的）。
- 可以这么理解：各种Function是业务逻辑（组件）。算法组件只是提供运行规则，业务逻辑（组件）作为运行在算法组件上的插件。
- 也可以这么理解 ： 算法组件就是框架，其把部分业务逻辑委托给Mapper或者ComQueueItem。

比如

- KMeans 是 Estimator，其对应算法组件是 KMeansTrainBatchOp。其业务逻辑（组件）也在这个类中，是由IterativeComQueue为基础串联起来的一系列算法类(ComQueueItem)。
- VectorAssembler 是 Transformer，其对应算法组件是 MapBatchOp。其业务逻辑（组件）是 VectorAssemblerMapper（其 map 函数会做业务逻辑，把将多个数值列按顺序汇总成一个向量列）。

```
public final class KMeansTrainBatchOp extends BatchOperator <KMeansTrainBatchOp>   implements K
MeansTrainParams <KMeansTrainBatchOp>

// class for a flat map BatchOperator.
public class MapBatchOp<T extends MapBatchOp<T>> extends BatchOperator<T>
```

无论是调用Estimator.fit 还是 Transformer.transform，其本质都是通过linkFrom函数，把各个Operator联系起来，这样就把数据流串联起来。然后就可以逐步映射到Flink具体运行计划上。

1. Algorithm operators

AlgoOperator是算子组件的基类，其子类有BatchOperator和StreamOperator，分别对应了批处理和流处理。

```
// Base class for algorithm operators.
public abstract class AlgoOperator<T extends AlgoOperator<T>>
    implements WithParams<T>, HasMLEnvironmentId<T>, Serializable

// Base class of batch algorithm operators.
public abstract class BatchOperator<T extends BatchOperator <T>> extends AlgoOperator <T> {
    // Link this object to BatchOperator using the BatchOperators as its input.
        public abstract T linkFrom(BatchOperator <?>... inputs);

    public <B extends BatchOperator <?>> B linkTo(B next) {
```

```
      return link(next);
    }
    public BatchOperator print() throws Exception {
      return linkTo(new PrintBatchOp().setMLEnvironmentId(getMLEnvironmentId()));
    }
}

public abstract class StreamOperator<T extends StreamOperator <T>> extends AlgoOperator <T>
```

示例代码如下：

```
// 输入csv文件被转化为一个BatchOperator
BatchOperator data = new CsvSourceBatchOp().setFilePath(URL).setSchemaStr(SCHEMA_STR);

...

pipeline.fit(data).transform(data).print();
```

2. Mapper（提前说明）

Mapper是底层迭代计算框架的一部分，是业务逻辑（组件）。从目录结构能看出。这里提前说明，是因为在流水线讲解过程中大量涉及，所以就**提前放在这里说明**。

```
./java/com/alibaba/alink/common
linalg mapper model comqueue utils io
```

Mapper的几个主要类定义如下，其作用广泛，即可以映射输入到输出，也可以映射模型到具体数值。

```
// Abstract class for mappers.
public abstract class Mapper implements Serializable {}

// Abstract class for mappers with model.
public abstract class ModelMapper extends Mapper {}

// Find  the closest cluster center for every point.
public class KMeansModelMapper extends ModelMapper {}

// Mapper with Multi-Input columns and Single Output column(MISO).
public abstract class MISOMapper extends Mapper {}

// This mapper maps many columns to one vector. the columns should be vector or numerical colum
ns.
public class VectorAssemblerMapper extends MISOMapper {}
```

Mapper的业务逻辑依赖于算法组件来运作，比如 [VectorAssemblerMapper in MapBatchOp]，[KMeansModelMapper in ModelMapBatchOp]。

ModelMapper具体运行则需要依赖 ModelMapperAdapter 来和Flink runtime联系起来。ModelMapperAdapter继承了RichMapFunction，ModelMapper作为其成员变量，在map操作中执行业务逻辑，ModelSource则是数据来源。

对应本实例，KMeansModelMapper 就是最后转换的 BatchOperator，其map函数用来转换。

3. 系统内置算法组件

系统内置了一些常用的算法组件，比如：

- MapBatchOp 功能是基于输入来flat map，是 VectorAssembler 返回的算法组件。
- ModelMapBatchOp 功能是基于模型进行flat map，是 KMeans 返回的算法组件。

以 ModelMapBatchOp 为例给大家说明其作用，从下面代码注释中可以看出，linkFrom作用是：

- 把inputs"算法组件" 和 本身"算法组件" 联系起来，这就形成了一个算法逻辑链。
- 把业务逻辑映射成 "Flink算子"，这就形成了一个 "Flink算子链"。

```
public class ModelMapBatchOp<T extends ModelMapBatchOp<T>> extends BatchOperator<T> {
        @Override
        public T linkFrom(BatchOperator<?>... inputs) {
                checkOpSize(2, inputs);

                try {
                        BroadcastVariableModelSource modelSource = new BroadcastVariableModelSo
urce(BROADCAST_MODEL_TABLE_NAME);
      // mapper是映射函数
                        ModelMapper mapper = this.mapperBuilder.apply(
                                        inputs[0].getSchema(),
                                        inputs[1].getSchema(),
                                        this.getParams());
      // modelRows 是模型
                        DataSet<Row> modelRows = inputs[0].getDataSet().rebalance();
      // resultRows 是输入数据的映射变化
                        DataSet<Row> resultRows = inputs[1].getDataSet()
                                        .map(new ModelMapperAdapter(mapper, modelSource))
           // 把模型作为广播变量，后续会在 ModelMapperAdapter 中使用
                                        .withBroadcastSet(modelRows, BROADCAST_MODEL_TABLE_NAME
);

                        TableSchema outputSchema = mapper.getOutputSchema();
                        this.setOutput(resultRows, outputSchema);
                        return (T) this;
                } catch (Exception ex) {
                        throw new RuntimeException(ex);
                }
        }
}
```

ModelMapperAdapter

ModelMapperAdapter 是适配器的实现，用来在flink上运行业务逻辑Mapper。从代码可以看出，ModelMapperAdapter取出之前存储的mapper和模型数据，然后基于此来进行具体算法业务。

```
/**
 * Adapt a {@link ModelMapper} to run within flink.

 * This adapter class hold the target {@link ModelMapper} and it's {@link ModelSource}. Upon op
en(), it will load model rows from {@link ModelSource} into {@link ModelMapper}.
 */
public class ModelMapperAdapter extends RichMapFunction<Row, Row> implements Serializable {

    /**
     * The ModelMapper to adapt.
     */
    private final ModelMapper mapper;

    /**
     * Load model data from ModelSource when open().
     */
    private final ModelSource modelSource;

    public ModelMapperAdapter(ModelMapper mapper, ModelSource modelSource) {
```

```
        // mapper是业务逻辑，modelSource是模型Broadcast source
        this.mapper = mapper; // 在map操作中执行业务逻辑
        this.modelSource = modelSource; // 数据来源
    }

    @Override
    public void open(Configuration parameters) throws Exception {
        // 从广播变量中获取模型数据
        List<Row> modelRows = this.modelSource.getModelRows(getRuntimeContext());
        this.mapper.loadModel(modelRows);
    }

    @Override
    public Row map(Row row) throws Exception {
        // 执行业务逻辑，在数据来源上转换
        return this.mapper.map(row);
    }
}
```

4. 训练阶段fit

在 `pipeline.fit(data)` 之中，会沿着流水线依次执行。如果流水线下一个阶段遇到了是Transformer，就调用其transform；如果遇到的是EstimatorBase，就先调用其fit，把EstimatorBase转换为Transformer，然后再次调用这个转换出来的Transformer.transform。就这样一个一个阶段执行。

4.1 具体流水线处理

1. 如果流水线下一阶段遇到EstimatorBase，会处理EstimatorBase的fit，把流水线上的Estimator转换为TransformerBase。Estimator.fit 接受一个特征数据并产生一个转换器。

   （如果这个阶段 不是 流水线最后一个阶段）会对这个 TransformerBase继续处理。处理之后才能进入到流水线下一阶段。

   （如果这个阶段 是 流水线最后一个阶段）不会对这个 TransformerBase 做处理，直接结束流水线 fit 操作。

2. 如果流水线下一阶段遇到TransformerBase，就直接调用其transform函数。

3. 对于所有需要处理的TransformerBase，无论是从EstimatorBase转换出来的，还是Pipeline原有的 ，都调用其transform函数，转换其input。 `input = transformers[i].transform(input);` 。这样每次转换后的输出再次赋值给input，作为流水线下一阶段的输入。

4. 最后得到一个PipelineModel (其本身也是一个Transformer) ，这个属于下一阶段转换流水线。

4.2 结合本实例概述

本实例有两个stage。VectorAssembler是Transformer，KMeans是EstimatorBase。

这时候Pipeline其内部变量是：

```
this = {Pipeline@1195}
 stages = {ArrayList@2851}  size = 2

  0 = {VectorAssembler@1198}
   mapperBuilder = {VectorAssembler$lambda@2857}
   params = {Params@2858} "Params {outputCol="features", selectedCols=["sepal_length","sepal_wi
dth","petal_length","petal_width"]}"

  1 = {KMeans@2856}
   params = {Params@2860} "Params {vectorCol="features", maxIter=100, reservedCols=["category"]
```

```
, k=3, predictionCol="prediction_result", predictionDetailCol="prediction_detail"}"
    params = {HashMap@2862}  size = 6
```

- Pipeline 先调用Transformer类型的VectorAssembler，来处理其input (就是csv的BatchOperator)。这个处理csv是通过linkFrom(input)来构建的。处理之后再包装成一个MapBatchOp返回赋值给input。
- 其次调用EstimatorBase类型的Kmeans.fit函数，对input (就是 VectorAssembler 返回的MapBatchOp)进行fit。fit过程中调用了KMeansTrainBatchOp.linkFrom来设置，fit生成了一个KMeansModel (Transformer)。因为这时候已经是流水线最后一步，所以不做后续的KMeansModel.transform操作。KMeansModel 就是训练出来的判断模型。
- 在上述调用过程中，会在transformers数组中记录运算过的TransformerBase和EstimatorBase适配出来的Transformer。
- 最后以这个transformers数组为参数，生成一个 PipelineModel (其也是一个Transformer类型)。生成 PipelineModel 的目的是：PipelineModel是后续转换中的新流水线。

PipelineMode 的新流水线处理流程是：从 csv 读入/ 映射（VectorAssembler 处理），然后 KMeansModel 做转换（下一节会具体介绍）。

fit 具体代码是

```
public class Pipeline extends EstimatorBase<Pipeline, PipelineModel> {

    // Train the pipeline with batch data.
        public PipelineModel fit(BatchOperator input) {

      int lastEstimatorIdx = getIndexOfLastEstimator();
      TransformerBase[] transformers = new TransformerBase[stages.size()];
      for (int i = 0; i < stages.size(); i++) {
        PipelineStageBase stage = stages.get(i);
        if (i <= lastEstimatorIdx) {
          if (stage instanceof EstimatorBase) {
            // 这里会把流水线上的具体 Algorithm operators 通过 linkFrom 函数串联起来。
            transformers[i] = ((EstimatorBase) stage).fit(input);
          } else if (stage instanceof TransformerBase) {
            transformers[i] = (TransformerBase) stage;
          }
          // 注意，如果是流水线最后一个阶段，则不做transform处理。
          if (i < lastEstimatorIdx) {
            // 这里会调用到具体Transformer的transform函数，其会把流水线上的具体 Algorithm operators 通过 linkFrom 函数串联起来。
            input = transformers[i].transform(input);
          }
        } else {
          transformers[i] = (TransformerBase) stage;
        }
      }
      // 这里生成了一个PipelineModel，transformers会作为参数传给他
      return new PipelineModel(transformers).setMLEnvironmentId(input.getMLEnvironmentId());
    }
}

// MapTransformer是VectorAssembler的基类。transform会生成一个MapBatchOp，然后再调用MapBatchOp.linkFrom。
public abstract class MapTransformer<T extends MapTransformer <T>>
                extends TransformerBase<T> implements LocalPredictable {
        @Override
        public BatchOperator transform(BatchOperator input) {
                return new MapBatchOp(this.mapperBuilder, this.params).linkFrom(input);
        }
```

```
}

// Trainer是KMeans的基类。
public abstract class Trainer<T extends Trainer <T, M>, M extends ModelBase<M>>
        @Override
        public M fit(BatchOperator input) {
    // KMeans.train 会调用 KMeansTrainBatchOp(this.getParams()).linkFrom(in);
    // createModel会生成一个新的model，本示例中是 com.alibaba.alink.pipeline.clustering.KMeansModel
                return createModel(train(input).getOutputTable());
        }
}
```

下面会逐一论述这两个环节。

4.3 VectorAssembler.transform

这部分作用是把csv数据转化为KMeans训练所需要的数据类型。

VectorAssembler.transform会调用到MapBatchOp.linkFrom。linkFrom首先把 csv input 进行了转换，变成DataSet，然后以此为参数生成一个MapBatchOp返回，这个返回的 MapBatchOp。其业务逻辑是在 VectorAssemblerMapper 中实现的（将多个数值列按顺序汇总成一个向量列）。

```
public class MapBatchOp<T extends MapBatchOp<T>> extends BatchOperator<T> {
    public T linkFrom(BatchOperator<?>... inputs) {
        BatchOperator in = checkAndGetFirst(inputs);

        try {
            Mapper mapper = (Mapper)this.mapperBuilder.apply(in.getSchema(), this.getParams());
            // 这里对csv输入进行了map，这里只是生成逻辑执行计划，具体操作会在print之后才做的。
            DataSet<Row> resultRows = in.getDataSet().map(new MapperAdapter(mapper));
            TableSchema resultSchema = mapper.getOutputSchema();
            this.setOutput(resultRows, resultSchema);
            return this;
        } catch (Exception var6) {
            throw new RuntimeException(var6);
        }
    }
}

// MapBatchOp本身
this = {MapBatchOp@3748} "UnnamedTable$1"
 mapperBuilder = {VectorAssembler$lambda@3744}
 params = {Params@3754} "Params {outputCol="features", selectedCols=["sepal_length","sepal_widt
h","petal_length","petal_width"]}"
 output = {TableImpl@5862} "UnnamedTable$1"
 sideOutputs = null

// mapper就是业务逻辑模块
mapper = {VectorAssemblerMapper@5785}
 handleInvalid = {VectorAssemblerMapper$HandleType@5813} "ERROR"
 outputColsHelper = {OutputColsHelper@5814}
 colIndices = {int[4]@5815}
 dataFieldNames = {String[5]@5816}
 dataFieldTypes = {DataType[5]@5817}
 params = {Params@5818} "Params {outputCol="features", selectedCols=["sepal_length","sepal_widt
h","petal_length","petal_width"]}"

// 返回数值如下
resultRows = {MapOperator@5788}
 function = {MapperAdapter@5826}
```

```
  mapper = {VectorAssemblerMapper@5785}
 defaultName = "linkFrom(MapBatchOp.java:35)"

// 调用栈如下

linkFrom:31, MapBatchOp (com.alibaba.alink.operator.batch.utils)
transform:34, MapTransformer (com.alibaba.alink.pipeline)
fit:122, Pipeline (com.alibaba.alink.pipeline)
main:31, KMeansExample (com.alibaba.alink)
```

4.4 KMeans.fit

这部分就是训练模型。

KMeans是一个Trainer，其进而实现了EstimatorBase类型，所以流水线就调用到了其fit函数

KMeans.fit就是调用了Trainer.fit。

- Trainer.fit首先调用train函数，最终调用KMeansTrainBatchOp.linkFrom，这样就和VectorAssembler串联起来。KMeansTrainBatchOp 把VectorAssembler返回的 MapBatchOp进行处理。最后返回一个同样类型KMeansTrainBatchOp。
- Trainer.fit其次调用Trainer.createModel，该函数会根据this的类型决定应该生成什么Model。对于KMeans，就生成了KMeansModel。

因为KMeans是流水线最后一个阶段，这时候不调用 `input = transformers[i].transform(input);` 所以目前还是训练，生成一个模型 KMeansModel。

```
// 实际部分代码

Trainer.fit(BatchOperator input) {
                return createModel(train(input).getOutputTable());
}

public final class KMeansTrainBatchOp extends BatchOperator <KMeansTrainBatchOp>
        implements KMeansTrainParams <KMeansTrainBatchOp> {

        public KMeansTrainBatchOp linkFrom(BatchOperator <?>... inputs) {
            DataSet <Row> finalCentroid = iterateICQ(initCentroid, data,
                vectorSize, maxIter, tol, distance, distanceType, vectorColName, null, null);
            this.setOutput(finalCentroid, new KMeansModelDataConverter().getModelSchema());
            return this;
        }
}

// 变量内容

this = {KMeansTrainBatchOp@5887}
 params = {Params@5895} "Params {vectorCol="features", maxIter=100, reservedCols=["category"],
k=3, predictionCol="prediction_result", predictionDetailCol="prediction_detail"}"
 output = null
 sideOutputs = null
inputs = {BatchOperator[1]@5888}
 0 = {MapBatchOp@3748} "UnnamedTable$1"
  mapperBuilder = {VectorAssembler$lambda@3744}
  params = {Params@3754} "Params {outputCol="features", selectedCols=["sepal_length","sepal_wid
th","petal_length","petal_width"]}"
  output = {TableImpl@5862} "UnnamedTable$1"
  sideOutputs = null

// 调用栈如下
```

```
linkFrom:84, KMeansTrainBatchOp (com.alibaba.alink.operator.batch.clustering)
train:31, KMeans (com.alibaba.alink.pipeline.clustering)
fit:34, Trainer (com.alibaba.alink.pipeline)
fit:117, Pipeline (com.alibaba.alink.pipeline)
main:31, KMeansExample (com.alibaba.alink)
```

KMeansTrainBatchOp.linkFrom是算法重点。这里其实就是生成了算法所需要的一切前提，把各种Flink算子搭建好。后续会再提到。

fit函数生成了 KMeansModel，其transform函数在基类MapModel中实现，会在下一个transform阶段完成调用。这个就是训练出来的KMeans模型，其也是一个Transformer。

```
// Find  the closest cluster center for every point.
public class KMeansModel extends MapModel<KMeansModel>
        implements KMeansPredictParams <KMeansModel> {

        public KMeansModel(Params params) {
                super(KMeansModelMapper::new, params);
        }
}
```

4.5 生成新的转换流水线

前面说到了，Pipeline的fit函数，返回一个PipelineModel。这个PipelineModel在后续会继续调用transform，完成转换阶段。

```
return new PipelineModel(transformers).setMLEnvironmentId(input.getMLEnvironmentId());
```

5. 转换阶段transform

转换阶段的流水线，依然要从VectorAssembler入手来读取csv，进行map处理。然后调用 KMeansModel。

PipelineModel会继续调用transform函数。其作用是把Transformer转化为BatchOperator。这时候其内部变量如下，看出来已经从最初流水线各种类型参杂 转换为 统一transform实例。

```
this = {PipelineModel@5016}
 transformers = {TransformerBase[2]@5017}

  0 = {VectorAssembler@1198}
   mapperBuilder = {VectorAssembler$lambda@2855}
   params = {Params@2856} "Params {outputCol="features", selectedCols=["sepal_length","sepal_wi
dth","petal_length","petal_width"]}"

  1 = {KMeansModel@5009}
   mapperBuilder = {KMeansModel$lambda@5011}
   modelData = {TableImpl@4984} "UnnamedTable$2"
   params = {Params@5012} "Params {vectorCol="features", maxIter=100, reservedCols=["category"]
, k=3, predictionCol="prediction_result", predictionDetailCol="prediction_detail"}"
 modelData = null
 params = {Params@5018} "Params {MLEnvironmentId=0}"
```

- 第一次transform调用到了MapBatchOp.linkFrom，就是VectorAssembler.transform调用到的，其作用和 在 fit 流水线中起到的作用一样，下面注释中有解释。
- 第二次transform调用到了ModelMapBatchOp.linkFrom，就是KMeansModel.transform间接调用到的。下面注释中有解释。

这两次 transform 的调用生成了 BatchOperator 的串联。最终返回结果是 ModelMapBatchOp，即一个BatchOperator。转换将由ModelMapBatchOp来转换。

```
// The model fitted by Pipeline.
public class PipelineModel extends ModelBase<PipelineModel> implements LocalPredictable {
    @Override
    public BatchOperator<?> transform(BatchOperator input) {
        for (TransformerBase transformer : this.transformers) {
            input = transformer.transform(input);
        }
        return input;
    }
}

// 经过变化后，得到一个最终的转化结果 BatchOperator，以此来转换
// {KMeansModel$lambda@5050} 就是 KMeansModelMapper，转换逻辑。

input = {ModelMapBatchOp@5047} "UnnamedTable$3"
 mapperBuilder = {KMeansModel$lambda@5050}
 params = {Params@5051} "Params {vectorCol="features", maxIter=100, reservedCols=["category"],
k=3, predictionCol="prediction_result", predictionDetailCol="prediction_detail"}"
  params = {HashMap@5058}  size = 6
   "vectorCol" -> ""features""
   "maxIter" -> "100"
   "reservedCols" -> "["category"]"
   "k" -> "3"
   "predictionCol" -> ""prediction_result""
   "predictionDetailCol" -> ""prediction_detail""
 output = {TableImpl@5052} "UnnamedTable$3"
  tableEnvironment = {BatchTableEnvironmentImpl@5054}
  operationTree = {DataSetQueryOperation@5055}
  operationTreeBuilder = {OperationTreeBuilder@5056}
  lookupResolver = {LookupCallResolver@5057}
  tableName = "UnnamedTable$3"
 sideOutputs = null

// MapTransformer是VectorAssembler的基类。transform会生成一个MapBatchOp，然后再调用MapBatchOp.linkFr
om。
public abstract class MapTransformer<T extends MapTransformer <T>>
                extends TransformerBase<T> implements LocalPredictable {
        @Override
        public BatchOperator transform(BatchOperator input) {
                return new MapBatchOp(this.mapperBuilder, this.params).linkFrom(input);
        }
}

// MapModel是KMeansModel的基类，transform会生成一个ModelMapBatchOp，然后再调用ModelMapBatchOp.linkFr
om。
public abstract class MapModel<T extends MapModel<T>>
                extends ModelBase<T> implements LocalPredictable {
        @Override
        public BatchOperator transform(BatchOperator input) {
                return new ModelMapBatchOp(this.mapperBuilder, this.params)
                                .linkFrom(BatchOperator.fromTable(this.getModelData())
                                        .setMLEnvironmentId(input.getMLEnvironmentId()), input)
;
        }
}
```

在这两个linkFrom中，还是分别生成了两个MapOperator，然后拼接起来，构成了一个 BatchOperator 串。从上面代码中可以看出，KMeansModel对应的ModelMapBatchOp，其linkFrom会返回一个

ModelMapperAdapter。ModelMapperAdapter是一个RichMapFunction类型，它会把KMeansModelMapper作为RichMapFunction.function成员变量保存起来。然后会调用 `.map(new ModelMapperAdapter(mapper, modelSource))` ，map就是Flink算子，这样转换算法就和Flink联系起来了。

最后 Keans 算法的转换工作是通过 KMeansModelMapper.map 来完成的。

6. 运行

我们都知道，Flink程序中，为了让程序运行，需要

- 获取execution environment：调用类似 `getExecutionEnvironment()` 来获取environment；
- 触发程序执行：调用类似 `env.execute("KMeans Example");` 来真正执行。

Alink其实就是一个Flink应用，只不过要比普通Flink应用复杂太多。

但是从实例代码中，我们没有看到类似调用。这说明Alink封装的非常好，但是作为好奇的程序员，我们需要知道究竟这些调用隐藏在哪里。

获取执行环境

Alink是在Pipeline执行的时候，获取到运行环境。具体来说，因为csv文件是最初的输入，所以当transform调用其 `in.getSchema()` 时候，会获取运行环境。

```
public final class CsvSourceBatchOp extends BaseSourceBatchOp<CsvSourceBatchOp>
    implements CsvSourceParams<CsvSourceBatchOp> {
    @Override
    public Table initializeDataSource() {
      ExecutionEnvironment execEnv = MLEnvironmentFactory.get(getMLEnvironmentId()).getExecutio
nEnvironment();
    }
}

initializeDataSource:77, CsvSourceBatchOp (com.alibaba.alink.operator.batch.source)
getOutputTable:52, BaseSourceBatchOp (com.alibaba.alink.operator.batch.source)
getSchema:180, AlgoOperator (com.alibaba.alink.operator)
linkFrom:34, MapBatchOp (com.alibaba.alink.operator.batch.utils)
transform:34, MapTransformer (com.alibaba.alink.pipeline)
fit:122, Pipeline (com.alibaba.alink.pipeline)
main:31, KMeansExample (com.alibaba.alink)
```

触发程序运行

截止到现在，Alink已经做了很多东西，也映射到了 Flink算子上，那么究竟什么地方才真正和Flink联系起来呢?

print 调用的是BatchOperator.print，真正从这里开始，会一层一层调用下去，最后来到

```
package com.alibaba.alink.operator.batch.utils;

public class PrintBatchOp extends BaseSinkBatchOp<PrintBatchOp> {
        @Override
        protected PrintBatchOp sinkFrom(BatchOperator in) {
                this.setOutputTable(in.getOutputTable());
                if (null != this.getOutputTable()) {
                        try {
                // 在这个 collect 之后，会进入到 Flink 的runtime之中。
                                List <Row> rows = DataSetConversionUtil.fromTable(getMLEnvironm
entId(), this.getOutputTable()).collect();
                                batchPrintStream.println(TableUtil.formatTitle(this.getColNames
()));
```

```
                                for (Row row : rows) {
                                        batchPrintStream.println(TableUtil.formatRows(row));
                                }
                        } catch (Exception ex) {
                                throw new RuntimeException(ex);
                        }
                }
                return this;
        }
}
```

在 LocalEnvironment 这里把Alink和Flink的运行环境联系起来。

```
public class LocalEnvironment extends ExecutionEnvironment {
        @Override
        public String getExecutionPlan() throws Exception {
                Plan p = createProgramPlan(null, false);

        // 下面会真正的和Flink联系起来。
                if (executor != null) {
                        return executor.getOptimizerPlanAsJSON(p);
                }
                else {
                        PlanExecutor tempExecutor = PlanExecutor.createLocalExecutor(configurat
ion);
                        return tempExecutor.getOptimizerPlanAsJSON(p);
                }
        }
}

// 调用栈如下

execute:91, LocalEnvironment (org.apache.flink.api.java)
execute:820, ExecutionEnvironment (org.apache.flink.api.java)
collect:413, DataSet (org.apache.flink.api.java)
sinkFrom:40, PrintBatchOp (com.alibaba.alink.operator.batch.utils)
sinkFrom:18, PrintBatchOp (com.alibaba.alink.operator.batch.utils)
linkFrom:31, BaseSinkBatchOp (com.alibaba.alink.operator.batch.sink)
linkFrom:17, BaseSinkBatchOp (com.alibaba.alink.operator.batch.sink)
link:89, BatchOperator (com.alibaba.alink.operator.batch)
linkTo:239, BatchOperator (com.alibaba.alink.operator.batch)
print:337, BatchOperator (com.alibaba.alink.operator.batch)
main:31, KMeansExample (com.alibaba.alink)
```

0x05 底层--迭代计算框架

这里对应如下设计原则：

- 构建一套战术打法（middleware或者adapter），即屏蔽了Flink，又可以利用好Flink，还可以让用户基于此可以快速开发算法。
- 采用最简单，最常见的开发语言和开发模式。

让我们想想看，大概有哪些基础工作需要做：

- 如何初始化
- 如何通信
- 如何分割代码，如何广播代码
- 如何分割数据，如何广播数据
- 如何迭代算法

其中最重要的概念是IterativeComQueue，这是把通信或者计算抽象成ComQueueItem，然后把ComQueueItem串联起来形成队列。这样就形成了面向迭代计算场景的一套迭代通信计算框架。

再次把目录结构列在这里：

```
./java/com/alibaba/alink/common:
MLEnvironment.java              linalg MLEnvironmentFactory.java        mapper
VectorTypes.java                model comqueue                  utils io
```

里面大致有 ：

- Flink 封装模块 ： MLEnvironment.java， MLEnvironmentFactory.java。
- 线性代数模块：linalg。
- 计算/通讯队列模块：comqueue，其中ComputeFunction进行计算，比如训练算法。
- 映射模块：mapper，其中Mapper进行各种映射，比如 ModelMapper 把模型映射为数值（就是转换算法）。
- 模型 ： model，主要是用来读取model source。
- 基础模块：utils，io。

算法组件在其linkFrom函数中，会做如下操作：

- 先进行部分初始化，此时会调用部分Flink算子，比如groupBy等等。
- 再将算法逻辑剥离出来，委托给Mapper或者ComQueueItem。
- Mapper或者ComQueueItem会调用Flink map算子或者mapPartition算子等。
- 调用Flink算子过程就是把算法分割然后适配到Flink上的过程。

下面就一一阐述。

1. Flink上下文封装

MLEnvironment 是个重要的类。其封装了Flink开发所必须要的运行上下文。用户可以通过这个类来获取各种实际运行环境，可以建立table，可以运行SQL语句。

```
/**
 * The MLEnvironment stores the necessary context in Flink.
 * Each MLEnvironment will be associated with a unique ID.
 * The operations associated with the same MLEnvironment ID
 * will share the same Flink job context.
 */
public class MLEnvironment {
    private ExecutionEnvironment env;
    private StreamExecutionEnvironment streamEnv;
    private BatchTableEnvironment batchTableEnv;
    private StreamTableEnvironment streamTableEnv;
}
```

2. Function

Function是计算框架中，对于计算和通讯等业务逻辑的最小模块。具体定义如下。

- ComputeFunction 是计算模块。
- CommunicateFunction 是通讯模块。CommunicateFunction和ComputeFunction都是ComQueueItem子类，它们是业务逻辑实现者。
- CompareCriterionFunction 是判断模块，用来判断何时结束循环。这就允许用户指定迭代终止条件。
- CompleteResultFunction 用来在结束循环时候调用，作为循环结果。
- Mapper也是一种Funciton，即Mapper Function。

后续将统称为 Function。

```
/**
 * Basic build block in {@link BaseComQueue}, for either communication or computation.
 */
public interface ComQueueItem extends Serializable {}

/**
 * An BaseComQueue item for computation.
 */
public abstract class ComputeFunction implements ComQueueItem {

        /**
         * Perform the computation work.
         *
         * @param context to get input object and update output object.
         */
        public abstract void calc(ComContext context);
}

/**
 * An BaseComQueue item for communication.
 */
public abstract class CommunicateFunction implements ComQueueItem {

    /**
     * Perform communication work.
     *
     * @param input     output of previous queue item.
     * @param sessionId session id for shared objects.
     * @param <T>       Type of dataset.
     * @return result dataset.
     */
        public abstract <T> DataSet <T> communicateWith(DataSet <T> input, int sessionId);
}
```

结合我们代码来看，KMeansTrainBatchOp算法组件的部分作用是：KMeans算法被分割成若干CommunicateFunction。然后被添加到计算通讯队列上。

下面代码中，具体 Item 如下：

- **ComputeFunction** ： KMeansPreallocateCentroid，KMeansAssignCluster，KMeansUpdateCentroids
- **CommunicateFunction** ： AllReduce
- **CompareCriterionFunction** ： KMeansIterTermination
- **CompleteResultFunction** ： KMeansOutputModel

即算法实现的主要工作是：

- 构建了一个IterativeComQueue。
- 初始化数据，这里有两种办法：initWithPartitionedData将DataSet分片缓存至内存。initWithBroadcastData将DataSet整体缓存至每个worker的内存。
- 将计算分割为若干ComputeFunction，串联在IterativeComQueue
- 运用AllReduce通信模型完成了数据同步

```
        static DataSet <Row> iterateICQ(...省略...) {

                return new IterativeComQueue()
```

```
                        .initWithPartitionedData(TRAIN_DATA, data)
                        .initWithBroadcastData(INIT_CENTROID, initCentroid)
                        .initWithBroadcastData(KMEANS_STATISTICS, statistics)
                        .add(new KMeansPreallocateCentroid())
                        .add(new KMeansAssignCluster(distance))
                        .add(new AllReduce(CENTROID_ALL_REDUCE))
                        .add(new KMeansUpdateCentroids(distance))
                        .setCompareCriterionOfNode0(new KMeansIterTermination(distance, tol))
                        .closeWith(new KMeansOutputModel(distanceType, vectorColName, latitudeC
olName, longitudeColName))
                        .setMaxIter(maxIter)
                        .exec();
        }
```

3. 计算/通讯队列

BaseComQueue 就是这个迭代框架的基础。它维持了一个 `List<ComQueueItem> queue` 。用户在生成算法模块时候，会把各种 Function 添加到队列中。

IterativeComQueue 是 BaseComQueue 的缺省实现，具体实现了setMaxIter, setCompareCriterionOfNode0两个函数。

BaseComQueue两个重要函数是：

- optimize 函数：把队列上相邻的 ComputeFunction串联起来，形成一个 ChainedComputation。在框架中进行优化，就是Alink的一个优势所在。
- exec 函数：运行队列上的各个 Function，返回最终的 Dataset。实际上，这里才真正到了 Flink，比如把计算队列上的各个 ComputeFunction 映射到 Flink 的 RichMapPartitionFunction。然后在 mapPartition函数调用中，会调用真实算法逻辑片断 `computation.calc(context);` 。

可以认为，BaseComQueue 是个逻辑概念，让算法工程师可以更好的组织自己的业务语言。而通过其exec函数把算法逻辑映射到Flink算子上。这样在某种程度上起到了与Flink解耦合的作用。

具体定义（摘取函数内部分代码）如下：

```
// Base class for the com(Computation && Communicate) queue.
public class BaseComQueue<Q extends BaseComQueue<Q>> implements Serializable {

        /**
         * All computation or communication functions.
         */
        private final List<ComQueueItem> queue = new ArrayList<>();

        /**
         * The function executed to decide whether to break the loop.
         */
        private CompareCriterionFunction compareCriterion;

        /**
         * The function executed when closing the iteration
         */
        private CompleteResultFunction completeResult;

        private void optimize() {
                if (queue.isEmpty()) {
                        return;
                }

                int current = 0;
```

```
                for (int ahead = 1; ahead < queue.size(); ++ahead) {
                        ComQueueItem curItem = queue.get(current);
                        ComQueueItem aheadItem = queue.get(ahead);

                // 这里进行判断，是否是前后都是 ComputeFunction，然后合并成 ChainedComputation
                        if (aheadItem instanceof ComputeFunction && curItem instanceof ComputeF
unction) {
                                if (curItem instanceof ChainedComputation) {
                                        queue.set(current, ((ChainedComputation) curItem).add((
ComputeFunction) aheadItem));
                                } else {
                                        queue.set(current, new ChainedComputation()
                                                .add((ComputeFunction) curItem)
                                                .add((ComputeFunction) aheadItem)
                                        );
                                }
                        } else {
                                queue.set(++current, aheadItem);
                        }
                }

                queue.subList(current + 1, queue.size()).clear();
        }

        /**
         * Execute the BaseComQueue and get the result dataset.
         *
         * @return result dataset.
         */
        public DataSet<Row> exec() {

                optimize();

                IterativeDataSet<byte[]> loop
                        = loopStartDataSet(executionEnvironment)
                        .iterate(maxIter);

                DataSet<byte[]> input = loop
                        .mapPartition(new DistributeData(cacheDataObjNames, sessionId))
                        .withBroadcastSet(loop, "barrier")
                        .name("distribute data");

                for (ComQueueItem com : queue) {
                        if ((com instanceof CommunicateFunction)) {
                                CommunicateFunction communication = ((CommunicateFunction) com)
;
         // 这里会调用比如 AllReduce.communication，其会返回allReduce包装后赋值给input，当循环遇到了下一个ComputeFunction（KMeansUpdateCentroids）时候，会把input赋给它处理。比如input = {MapPartitionOperator@5248}, input.function = {AllReduce$AllReduceRecv@5260}, input调用mapPartition，去间接调用KMeansUpdateCentroids。
                                input = communication.communicateWith(input, sessionId);
                        } else if (com instanceof ComputeFunction) {
                                final ComputeFunction computation = (ComputeFunction) com;

        // 这里才到了 Flink，把计算队列上的各个 ComputeFunction 映射到 Flink 的RichMapPartitionFunction。
                                input = input
                                                .mapPartition(new RichMapPartitionFunction<byte
[], byte[]>() {
```

```
                                                @Override
                                                public void mapPartition(Iterable<byte[]> value
s, Collector<byte[]> out) {
                                                        ComContext context = new ComContext(
                                                                sessionId, getIterationRuntimeC
ontext()
                                                        );
                        // 在这里会被Flink调用具体计算函数，就是之前算法工程师拆分的算法片段。
                                                        computation.calc(context);
                                                }
                                        })
                                        .withBroadcastSet(input, "barrier")
                                        .name(com instanceof ChainedComputation ?
                                                ((ChainedComputation) com).name()
                                                : "computation@" + computation.getClass().getSi
mpleName());
                                } else {
                                        throw new RuntimeException("Unsupported op in iterative queue."
);
                                }
                        }

                        return serializeModel(clearObjs(loopEnd));
                }
        }
```

4. Mapper（Function）

Mapper是底层迭代计算框架的一部分，可以认为是 Mapper Function。因为涉及到业务逻辑，所以提前说明。

5. 初始化

初始化发生在 KMeansTrainBatchOp.linkFrom 中。我们可以看到在初始化时候，是可以调用 Flink 各种算子(比如.rebalance().map())，因为这时候还没有和框架相关联，这时候的计算是用户自行控制，不需要加到 IterativeComQueue 之上。

如果某一个计算既要加到 IterativeComQueue 之上，还要自己玩 Flink 算子，那框架就懵圈了，不知道该如何处理。所以用户自由操作只能发生在没有和框架联系之前。

```
        @Override
        public KMeansTrainBatchOp linkFrom(BatchOperator <?>... inputs) {
                DataSet <FastDistanceVectorData> data = statistics.f0.rebalance().map(
                        new MapFunction <Vector, FastDistanceVectorData>() {
                                @Override
                                public FastDistanceVectorData map(Vector value) {
                                        return distance.prepareVectorData(Row.of(value), 0);
                                }
                        });
                ......
    }
```

框架也提供了初始化功能，用于将DataSet缓存到内存中，缓存的形式包括Partition和Broadcast两种形式。前者将DataSet分片缓存至内存，后者将DataSet整体缓存至每个worker的内存。

```
                return new IterativeComQueue()
                        .initWithPartitionedData(TRAIN_DATA, data)
                        .initWithBroadcastData(INIT_CENTROID, initCentroid)
```

```
                .initWithBroadcastData(KMEANS_STATISTICS, statistics)
        ......
```

6. ComputeFunction

这是算法的具体计算模块，算法工程师应该把算法拆分成各个可以并行处理的模块，分别用 ComputeFunction 实现，这样可以利用 Flnk 的分布式计算效力。

下面举出一个例子如下，这段代码为每个点(point)计算最近的聚类中心，为每个聚类中心的点坐标的计数和求和：

```
/**
 * Find the closest cluster for every point and calculate the sums of the points belonging to t
he same cluster.
 */
public class KMeansAssignCluster extends ComputeFunction {
    private FastDistance fastDistance;
    private transient DenseMatrix distanceMatrix;

    @Override
    public void calc(ComContext context) {
        Integer vectorSize = context.getObj(KMeansTrainBatchOp.VECTOR_SIZE);
        Integer k = context.getObj(KMeansTrainBatchOp.K);
        // get iterative coefficient from static memory.
        Tuple2<Integer, FastDistanceMatrixData> stepNumCentroids;
        if (context.getStepNo() % 2 == 0) {
            stepNumCentroids = context.getObj(KMeansTrainBatchOp.CENTROID1);
        } else {
            stepNumCentroids = context.getObj(KMeansTrainBatchOp.CENTROID2);
        }

        if (null == distanceMatrix) {
            distanceMatrix = new DenseMatrix(k, 1);
        }

        double[] sumMatrixData = context.getObj(KMeansTrainBatchOp.CENTROID_ALL_REDUCE);
        if (sumMatrixData == null) {
            sumMatrixData = new double[k * (vectorSize + 1)];
            context.putObj(KMeansTrainBatchOp.CENTROID_ALL_REDUCE, sumMatrixData);
        }

        Iterable<FastDistanceVectorData> trainData = context.getObj(KMeansTrainBatchOp.TRAIN_DA
TA);
        if (trainData == null) {
            return;
        }

        Arrays.fill(sumMatrixData, 0.0);
        for (FastDistanceVectorData sample : trainData) {
            KMeansUtil.updateSumMatrix(sample, 1, stepNumCentroids.f1, vectorSize, sumMatrixDat
a, k, fastDistance,
                distanceMatrix);
        }
    }
}
```

这里能够看出，在 ComputeFunction 中，使用的是 命令式编程模式，这样能够最大的契合目前程序员现状，极高提升生产力。

7. CommunicateFunction

前面代码中有一个关键处 `.add(new AllReduce(CENTROID_ALL_REDUCE))` 。这部分代码起到了承前启后的作用。之前的 `KMeansPreallocateCentroid, KMeansAssignCluster` 和其后的 `KMeansUpdateCentroids` 通过它做了一个 reduce / broadcast 通讯。

具体从注解中可以看到，AllReduce 是 MPI 相关通讯原语的一个实现。这里主要是对 double[] object 进行 reduce / broadcast。

```
public class AllReduce extends CommunicateFunction {
        public static <T> DataSet <T> allReduce(
                DataSet <T> input,
                final String bufferName,
                final String lengthName,
                final SerializableBiConsumer <double[], double[]> op,
                final int sessionId) {
                final String transferBufferName = UUID.randomUUID().toString();

                return input
                        .mapPartition(new AllReduceSend <T>(bufferName, lengthName, transferBuf
ferName, sessionId))
                        .withBroadcastSet(input, "barrier")
                        .returns(
                                new TupleTypeInfo <>(Types.INT, Types.INT, PrimitiveArrayTypeIn
fo.DOUBLE_PRIMITIVE_ARRAY_TYPE_INFO))
                        .name("AllReduceSend")
                        .partitionCustom(new Partitioner <Integer>() {
                                @Override
                                public int partition(Integer key, int numPartitions) {
                                        return key;
                                }
                        }, 0)
                        .name("AllReduceBroadcastRaw")
                        .mapPartition(new AllReduceSum(bufferName, lengthName, sessionId, op))
                        .returns(
                                new TupleTypeInfo <>(Types.INT, Types.INT, PrimitiveArrayTypeIn
fo.DOUBLE_PRIMITIVE_ARRAY_TYPE_INFO))
                        .name("AllReduceSum")
                        .partitionCustom(new Partitioner <Integer>() {
                                @Override
                                public int partition(Integer key, int numPartitions) {
                                        return key;
                                }
                        }, 0)
                        .name("AllReduceBroadcastSum")
                        .mapPartition(new AllReduceRecv <T>(bufferName, lengthName, sessionId))
                        .returns(input.getType())
                        .name("AllReduceRecv");
        }
}
```

经过调试我们能看出来，AllReduceSum 是在自己mapPartition实现中，调用了 SUM。

```
        /**
         * The all-reduce operation which does elementwise sum operation.
         */
        public final static SerializableBiConsumer <double[], double[]> SUM
                = new SerializableBiConsumer <double[], double[]>() {
                @Override
```

```
                public void accept(double[] a, double[] b) {
                        for (int i = 0; i < a.length; ++i) {
                                a[i] += b[i];
                        }
                }
        };

private static class AllReduceSum extends RichMapPartitionFunction <Tuple3 <Integer, Integer, d
ouble[]>, Tuple3 <Integer, Integer, double[]>> {
                @Override
                public void mapPartition(Iterable <Tuple3 <Integer, Integer, double[]>> values,
                                                                        Collector <Tuple3 <Integer, In
teger, double[]>> out) {

      // 省略各种初始化操作，比如确定传输位置，传输目标等
      ......

                        do {
                                Tuple3 <Integer, Integer, double[]> val = it.next();
                                int localPos = val.f1 - startPos;
                                if (sum[localPos] == null) {
                                        sum[localPos] = val.f2;
                                        agg[localPos]++;
                                } else {
          // 这里会调用 SUM
                                        op.accept(sum[localPos], val.f2);
                                }
                        } while (it.hasNext());

                        for (int i = 0; i < numOfSubTasks; ++i) {
                                for (int j = 0; j < cnt; ++j) {
                                        out.collect(Tuple3.of(i, startPos + j, sum[j]));
                                }
                        }
                }
        }

accept:129, AllReduce$3 (com.alibaba.alink.common.comqueue.communication)
accept:126, AllReduce$3 (com.alibaba.alink.common.comqueue.communication)
mapPartition:314, AllReduce$AllReduceSum (com.alibaba.alink.common.comqueue.communication)
run:103, MapPartitionDriver (org.apache.flink.runtime.operators)
run:504, BatchTask (org.apache.flink.runtime.operators)
run:157, AbstractIterativeTask (org.apache.flink.runtime.iterative.task)
run:107, IterationIntermediateTask (org.apache.flink.runtime.iterative.task)
invoke:369, BatchTask (org.apache.flink.runtime.operators)
doRun:705, Task (org.apache.flink.runtime.taskmanager)
run:530, Task (org.apache.flink.runtime.taskmanager)
run:745, Thread (java.lang)
```

0x06 另一种打法

总结到现在，我们发现这个迭代计算框架设计的非常优秀。但是Alink并没有限定大家只能使用这个框架来实现算法。如果你是Flink高手，你完全可以随心所欲的实现。

Alink例子中本身就有一个这样的实现 ALSExample。其核心类 AlsTrainBatchOp 就是直接使用了 Flink 算子，IterativeDataSet 等。

这就好比是武松武都头，一双戒刀搠得倒贪官佞臣，赤手空拳也打得死吊睛白额大虫。

```
public final class AlsTrainBatchOp
    extends BatchOperator<AlsTrainBatchOp>
    implements AlsTrainParams<AlsTrainBatchOp> {

    @Override
    public AlsTrainBatchOp linkFrom(BatchOperator<?>... inputs) {
        BatchOperator<?> in = checkAndGetFirst(inputs);

                ......

        AlsTrain als = new AlsTrain(rank, numIter, lambda, implicitPrefs, alpha, numMiniBatches
, nonNegative);
        DataSet<Tuple3<Byte, Long, float[]>> factors = als.fit(alsInput);

        DataSet<Row> output = factors.mapPartition(new RichMapPartitionFunction<Tuple3<Byte, Lo
ng, float[]>, Row>() {
            @Override
            public void mapPartition(Iterable<Tuple3<Byte, Long, float[]>> values, Collector<Ro
w> out) {
                new AlsModelDataConverter(userColName, itemColName).save(values, out);
            }
        });

        return this;
    }
}
```

多提一点，Flink ML中也有ALS算法，是一个Scala实现。没有Scala经验的算法工程师看代码会咬碎钢牙。

0x07 总结

经过这两篇文章的推测和验证，现在我们总结如下。

Alink的部分设计原则

- 算法的归算法，Flink的归Flink，尽量屏蔽AI算法和Flink之间的联系。
- 采用最简单，最常见的开发语言和思维方式。
- 尽量借鉴市面上通用的机器学习设计思路和开发模式，让开发者无缝切换。
- 构建一套战术打法（middleware或者adapter），即屏蔽了Flink，又可以利用好Flink，还可以让用户基于此可以快速开发算法。

针对这些原则，Alink实现了

- 顶层流水线，Estimator, Transformer...
- 算法组件中间层
- 底层迭代计算框架

这样Alink即可以最大限度的享受Flink带来的各种优势，也能顺应目前形势，让算法工程师工作更方便。从而达到系统性能和生产力的双重提升。

下一篇文章争取介绍 AllReduce 的具体实现。

0x08 参考

k-means聚类算法原理简析

flink kmeans聚类算法实现

Spark ML简介之Pipeline，DataFrame，Estimator，Transformer

开源 | 全球首个批流一体机器学习平台

斩获GitHub 2000+ Star，阿里云开源的 Alink 机器学习平台如何跑赢双11数据“博弈”？ | AI 技术生态论

Flink DataSet API