Alink漫谈(五)：迭代计算和Superstep

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。迭代算法在很多数据分析领域会用到，比如机器学习或者图计算。本文将通过Superstep入手看看Alink是如何利用Flink迭代API来实现具体算法。

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

0x01 缘由

为什么提到 Superstep 这个概念，是因为在撸KMeans代码的时候，发现几个很奇怪的地方，比如以下三个步骤中，都用到了context.getStepNo()，而且会根据其数值的不同进行不同业务操作：

```
public class KMeansPreallocateCentroid extends ComputeFunction {
    public void calc(ComContext context) {
        LOG.info("liuhao  KMeansPreallocateCentroid ");
        if (context.getStepNo() == 1) {
          /** 具体业务逻辑代码
           * Allocate memory for pre-round centers and current centers.
           */
        }
    }
}

public class KMeansAssignCluster extends ComputeFunction {
    public void calc(ComContext context) {
        ......
        if (context.getStepNo() % 2 == 0) {
            stepNumCentroids = context.getObj(KMeansTrainBatchOp.CENTROID1);
        } else {
            stepNumCentroids = context.getObj(KMeansTrainBatchOp.CENTROID2);
        }
      /** 具体业务逻辑代码
       * Find the closest cluster for every point and calculate the sums of the points belongin
g to the same cluster.
       */
    }
}

public class KMeansUpdateCentroids extends ComputeFunction {
    public void calc(ComContext context) {
        if (context.getStepNo() % 2 == 0) {
            stepNumCentroids = context.getObj(KMeansTrainBatchOp.CENTROID2);
        } else {
            stepNumCentroids = context.getObj(KMeansTrainBatchOp.CENTROID1);
        }
      /** 具体业务逻辑代码
       * Update the centroids based on the sum of points and point number belonging to the same
 cluster.
       */
    }
```

查看ComContext的源码，发现stepNo的来源居然是 `runtimeContext.getSuperstepNumber()` 。

```
public class ComContext {
   private final int taskId;
   private final int numTask;
   private final int stepNo; // 对，就是这里
   private final int sessionId;
        public ComContext(int sessionId, IterationRuntimeContext runtimeContext) {
                this.sessionId = sessionId;
                this.numTask = runtimeContext.getNumberOfParallelSubtasks();
                this.taskId = runtimeContext.getIndexOfThisSubtask();
                this.stepNo = runtimeContext.getSuperstepNumber(); // 这里进行了变量初始化
        }
        /**
         * Get current iteration step number, the same as {@link IterationRuntimeContext#getSup
```

```
erstepNumber()}.
         * @return iteration step number.
         */
        public int getStepNo() {
                return stepNo; // 这里是使用
        }
}
```

看到这里有的兄弟可能会虎躯一震，*这不是BSP模型的概念嘛。我就是想写个KMeans算法，怎么除了MPI模型，还要考虑BSP模型*。下面就让我们一步一步挖掘究竟Alink都做了什么工作。

0x02 背景概念

2.1 四层执行图

在 Flink 中的执行图可以分为四层：StreamGraph -> JobGraph -> ExecutionGraph -> 物理执行图

- StreamGraph：Stream API 编写的代码生成的最初的图。用来表示程序的拓扑结构。
- JobGraph：StreamGraph 经过优化后生成了 JobGraph， JobGraph是提交给 JobManager 的数据结构。主要的优化为，将多个符合条件的节点 chain 在一起作为一个节点，这样可以减少数据在节点之间流动所需要的序列化/反序列化/传输消耗。JobGraph是唯一被Flink的数据流引擎所识别的表述作业的数据结构，也正是这一共同的抽象体现了流处理和批处理在运行时的统一。
- ExecutionGraph：JobManager 根据 JobGraph 生成 ExecutionGraph。ExecutionGraph 是 JobGraph 的并行化版本，是调度层最核心的数据结构。
- 物理执行图：JobManager 根据 ExecutionGraph 对 Job 进行调度后，在各个TaskManager 上部署 Task 后形成的"图"，并不是一个具体的数据结构。

2.2 Task和SubTask

因为某种原因，Flink内部对这两个概念的使用本身就有些混乱：在Task Manager里这个subtask的概念由一个叫Task的类来实现。Task Manager里谈论的Task对象实际上对应的是ExecutionGraph里的一个subtask。

所以这两个概念需要理清楚。

- Task(任务)：Task对应JobGraph的一个节点，是一个算子Operator。Task 是一个阶段多个功能相同 subTask 的集合，类似于 Spark 中的 TaskSet。
- subTask(子任务)：subTask 是 Flink 中任务最小执行单元，是一个 Java 类的实例，这个 Java 类中有属性和方法，完成具体的计算逻辑。在ExecutionGraph里Task被分解为多个并行执行的subtask 。每个subtask作为一个excution分配到Task Manager里执行。
- Operator Chains(算子链)：没有 shuffle 的多个算子合并在一个 subTask 中，就形成了 Operator Chains，类似于 Spark 中的 Pipeline。Operator subTask 的数量指的就是算子的并行度。同一程序的不同算子也可能具有不同的并行度（因为可以通过 setParallelism() 方法来修改并行度）。

Flink 中的程序本质上是并行的。在执行期间，每一个算子 Operator (Transformation)都有一个或多个算子 subTask（Operator SubTask），每个算子的 subTask 之间都是彼此独立，并在不同的线程中执行，并且可能在不同的机器或容器上执行。

Task（ SubTask） 是一个Runnable 对象， Task Manager接受到TDD 后会用它实例化成一个Task对象， 并启动一个线程执行Task的Run方法。

TaskDeploymentDescriptor(TDD)：是Task Manager在submitTask是提交给TM的数据结构。 他包含了关于Task的所有描述信息。比如：

- TaskInfo：包含该Task 执行的java 类，该类是某个 AbstractInvokable的实现类， 当然也是某个operator的实现类 （比如DataSourceTask, DataSinkTask, BatchTask,StreamTask 等）。
- IG描述： 通常包含一个或两个InputGateDeploymentDescriptor（IGD)。

- 目标RP的描述：ParitionId, PartitionType, RS个数等等。

2.3 如何划分 Task 的依据

在以下情况下会重新划分task

- 并行度发生变化时
- keyBy() /window()/apply() 等发生 Rebalance 重新分配;
- 调用 startNewChain() 方法，开启一个新的算子链;
- 调用 diableChaining()方法，即：告诉当前算子操作不使用 算子链 操作。

比如有如下操作

```
DataStream<String> text = env.socketTextStream(hostname, port);

DataStream counts = text
    .filter(new FilterClass())
    .map(new LineSplitter())
    .keyBy(0)
    .timeWindow(Time.seconds(10))
    .sum(2)
```

那么StreamGraph的转换流是：

```
 Source --> Filter --> Map --> Timestamps/Watermarks --> Window(SumAggregator) --> Sink
```

其task是四个：

- Source --> Filter --> Map
- keyBy
- timeWindow
- Sink

其中每个task又会被分成分若干subtask。在执行时，一个Task会被并行化成若干个subTask实例进行执行，一个subTask对应一个执行线程。

2.4 JobGraph

以上说了这么多，就是要说jobGraph和subtask，因为本文中我们在分析源码和调试时候，主要是从jobGraph这里开始入手来看subtask。

JobGraph是在StreamGraph的基础之上，对StreamNode进行了关联合并的操作，比如对于source -> flatMap -> reduce -> sink 这样一个数据处理链，当source和flatMap满足链接的条件时，可以可以将两个操作符的操作放到一个线程并行执行，这样可以减少网络中的数据传输，由于在source和flatMap之间的传输的数据也不用序列化和反序列化，所以也提高了程序的执行效率。

相比流图（StreamGraph）以及批处理优化计划（OptimizedPlan），JobGraph发生了一些变化，已经不完全是“静态”的数据结构了，因为它加入了中间结果集（IntermediateDataSet）这一“动态”概念。

作业顶点（JobVertex）、中间数据集（IntermediateDataSet）、作业边（JobEdge）是组成JobGraph的基本元素。这三个对象彼此之间互为依赖：

- 一个JobVertex关联着若干个JobEdge作为输入端以及若干个IntermediateDataSet作为其生产的结果集；每个JobVertex都有诸如并行度和执行代码等属性。
- 一个IntermediateDataSet关联着一个JobVertex作为生产者以及若干个JobEdge作为消费者；
- 一个JobEdge关联着一个IntermediateDataSet可认为是源以及一个JobVertex可认为是目标消费者；

那么JobGraph是怎么组织并存储这些元素的呢？其实JobGraph只以Map的形式存储了所有的JobVertex，键是JobVertexID：

```
private final Map<JobVertexID, JobVertex> taskVertices = new LinkedHashMap<JobVertexID,
JobVertex>();
```

至于其它的元素，通过JobVertex都可以根据关系找寻到。需要注意的是，用于迭代的反馈边（feedback edge）当前并不体现在JobGraph中，而是被内嵌在特殊的JobVertex中通过反馈信道（feedback channel）在它们之间建立关系。

2.5 BSP模型和Superstep

BSP模型

BSP模型是并行计算模型的一种。并行计算模型通常指从并行算法的设计和分析出发，将各种并行计算机（至少某一类并行计算机）的基本特征抽象出来，形成一个抽象的计算模型。

BSP模型是一种异步MIMD-DM模型（DM: distributed memory，SM: shared memory），BSP模型支持消息传递系统，块内异步并行，块间显式同步，该模型基于一个master协调，所有的worker同步(lock-step)执行，数据从输入的队列中读取。

BSP计算模型不仅是一种体系结构模型，也是设计并行程序的一种方法。BSP程序设计准则是整体同步(bulk synchrony)，其独特之处在于超步(superstep)概念的引入。一个BSP程序同时具有水平和垂直两个方面的结构。从垂直上看,一个BSP程序由一系列串行的超步(superstep)组成。

BSP模型的实现

BSP模型的实现大概举例如下：

- **Pregel** ： Google的大规模图计算框架，首次提出了将BSP模型应用于图计算，具体请看Pregel——大规模图处理系统，不过至今未开源。
- **Apache Giraph** ： ASF社区的Incubator项目，由Yahoo!贡献，是BSP的java实现，专注于迭代图计算（如pagerank，最短连接等），每一个job就是一个没有reducer过程的hadoop job。
- **Apache Hama** ： 也是ASF社区的Incubator项目，与Giraph不同的是它是一个纯粹的BSP模型的java实现，并且不单单是用于图计算，意在提供一个通用的BSP模型的应用框架。

Flink-Gelly

Flink-Gelly利用Flink的高效迭代算子来支持海量数据的迭代式图处理。目前，Flink Gelly提供了“Vertex-Centric”，“Scatter-Gather”以及“Gather-Sum-Apply”等计算模型的实现。

“Vertex-Centric”迭代模型也就是我们经常听到的“Pregel”，是一种从Vertex角度出发的图计算方式。其中，同步地迭代计算的步骤称之为“superstep”。在每个“superstep”中，每个顶点都执行一个用户自定义的函数，且顶点之间通过消息进行通信，当一个顶点知道图中其他任意顶点的唯一ID时，该顶点就可以向其发送一条消息。

但是实际上，KMeans不是图处理，Alink也没有基于Flink-Gelly来构建。也许只是借鉴了其概念。所以我们还需要再探寻。

0x03 Flink的迭代算法（superstep-based）

迭代算法在很多数据分析领域会用到，比如机器学习或者图计算。为了从大数据中抽取有用信息，这个时候往往会需要在处理的过程中用到迭代计算。

所谓迭代运算，就是给定一个初值，用所给的算法公式计算初值得到一个中间结果，然后将中间结果作为输入参数进行反复计算，在满足一定条件的时候得到计算结果。

大数据处理框架很多，比如spark，mr。实际上这些实现迭代计算都是很困难的。

Flink直接支持迭代计算。Flink实现迭代的思路也是很简单，就是实现一个step函数，然后将其嵌入到迭代算子中去。有两种迭代操作算子：Iterate和Delta Iterate。两个操作算子都是在未收到终止迭代信号之前一直调用step函数。

3.1 Bulk Iterate

这种迭代方式称为全量迭代，它会将整个数据输入，经过一定的迭代次数，最终得到你想要的结果。

迭代操作算子包括了简单的迭代形式：每次迭代，step函数会消费全量数据(本次输入和上次迭代的结果)，然后计算得到下轮迭代的输出(例如，map，reduce，join等)

迭代过程主要分为以下几步：

- Iteration Input（迭代输入）：是初始输入值或者上一次迭代计算的结果。
- Step Function（step函数）：每次迭代都会执行step函数。它迭代计算DataSet，由一系列的operator组成，比如map，flatMap，join等，取决于具体的业务逻辑。
- Next Partial Solution（中间结果）：每一次迭代计算的结果，被发送到下一次迭代计算中。
- Iteration Result（迭代结果）：最后一次迭代输出的结果，被输出到datasink或者发送到下游处理。

它迭代的结束条件是：

- 达到最大迭代次数
- 自定义收敛聚合函数

编程的时候，需要调用iterate(int),该函数返回的是一个IterativeDataSet，当然我们可以对它进行一些操作，比如map等。Iterate函数唯一的参数是代表最大迭代次数。

迭代是一个环。我们需要进行闭环操作，那么这时候就要用到closeWith(Dataset)操作了，参数就是需要循环迭代的dataset。也可以可选的指定一个终止标准，操作closeWith(DataSet, DataSet)，可以通过判断第二个dataset是否为空，来终止迭代。如果不指定终止迭代条件，迭代就会在迭代了最大迭代次数后终止。

3.2 迭代机制

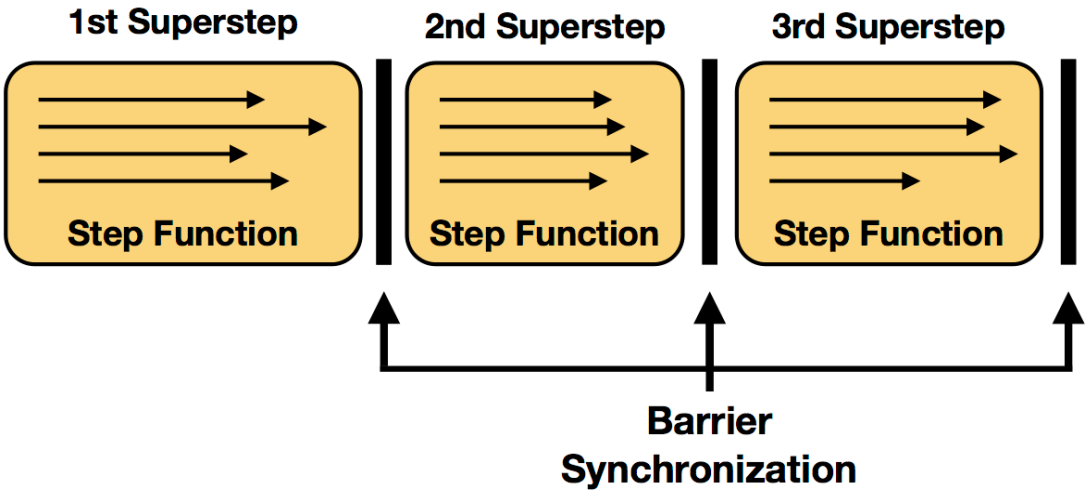
DataSet API引进了独特的同步迭代机制（superstep-based），仅限于用在有界的流。

我们将迭代操作算子的每个步骤函数的执行称为单个迭代。在并行设置中，在迭代状态的不同分区上并行计算step函数的多个实例。在许多设置中，对所有并行实例上的step函数的一次评估形成了所谓的superstep，这也是同步的粒度。因此，迭代的所有并行任务都需要在初始化下一个superstep之前完成superstep。终止准则也将被评估为superstep*同步*屏障。

下面是Apache原文

> We referred to each execution of the step function of an iteration operator as *a single iteration*. In parallel setups, **multiple instances of the step function are evaluated in parallel** on different partitions of the iteration state. In many settings, one evaluation of the step function on all parallel instances forms a so called **superstep**, which is also the granularity of synchronization. Therefore, *all* parallel tasks of an iteration need to complete the superstep, before a next superstep will be initialized. **Termination criteria** will also be evaluated at superstep barriers.

下面是apache原图



概括如下：

```
每次迭代都是一个superstep
    每次迭代中有若干subtask在不同的partition上分别执行step
        每个step有一个HeadTask，若干IntermediateTask，一个TailTask
    每个superstep有一个SynchronizationSinkTask 同步，因为迭代的所有并行任务需要在下一个迭代前完成
```

由此我们可以知道，superstep这是Flink DataSet API的概念，但是你从这里能够看到BSP模型的影子，比如：

- 在传统的BSP模型中，一个superstep被分为3步： 本地的计算， 消息的传递， 同步的barrier.
- Barrier Synchronization又叫障碍同步或栅栏同步。每一次同步也是一个超步的完成和下一个超步的开始；
- Superstep超步 是一次计算迭代，从起始每往前步进一层对应一个超步。
- 程序该什么时候结束是程序自己控制

0x04 Alink如何使用迭代

KMeansTrainBatchOp.iterateICQ函数中，生成了一个IterativeComQueue，而IterativeComQueue之中就用到了superstep-based迭代。

```
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)) // 终止条件
    .closeWith(new KMeansOutputModel(distanceType, vectorColName, latitudeColName, longitudeColN
ame))
    .setMaxIter(maxIter) // 迭代最大次数
    .exec();
```

而BaseComQueue.exec函数中则有：

```
public DataSet<Row> exec() {
    IterativeDataSet<byte[]> loop // Flink 迭代API
       = loopStartDataSet(executionEnvironment)
       .iterate(maxIter);
     // 后续操作能看出来，之前添加在queue上的比如KMeansPreallocateCentroid，都是在loop之上运行的。
                if (null == compareCriterion) {
        loopEnd = loop.closeWith...
```

```
        } else {
        // compare Criterion.
        DataSet<Boolean> criterion = input ... compareCriterion
        loopEnd = loop.closeWith( ... criterion ... )
      }
}
```

再仔细研究代码，我们可以看出：

superstep包括：

.add(new KMeansPreallocateCentroid())
.add(new KMeansAssignCluster(distance))
.add(new AllReduce(CENTROID_ALL_REDUCE))
.add(new KMeansUpdateCentroids(distance))

终止标准就是

利用KMeansIterTermination构建了一个RichMapPartitionFunction作为终止标准。最后结束时候调用KMeansOutputModel完成业务操作。

最大循环就是

.setMaxIter(maxIter)

于是我们可以得出结论，***superstep-based Bulk Iterate 迭代算子是用来实现整体KMeans算法，KMeans算法就是一个superstep进行迭代。但是在superstep内容如果需要通讯或者栅栏同步，则采用了MPI的allReduce。***

0x05 深入Flink源码和runtime来验证

我们需要深入到Flink内部去挖掘验证，如果大家有兴趣，可以参见下面调用栈，自己添加断点来研究。

```
execute:56, LocalExecutor (org.apache.flink.client.deployment.executors)
executeAsync:944, ExecutionEnvironment (org.apache.flink.api.java)
execute:860, ExecutionEnvironment (org.apache.flink.api.java)
execute:844, ExecutionEnvironment (org.apache.flink.api.java)
collect:413, DataSet (org.apache.flink.api.java)
sinkFrom:44, PrintBatchOp (com.alibaba.alink.operator.batch.utils)
sinkFrom:20, 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:35, KMeansExample (com.alibaba.alink)
```

5.1 向Flink提交Job

Alink和Flink构建联系，是在print调用中完成的。因为是本地调试，Flink会启动一个miniCluster，然后会做如下操作。

- 首先生成执行计划Plan。Plan以数据流形式来表示批处理程序，但它只是批处理程序最初的表示，然后计划会被优化以生成更高效的方案OptimizedPlan。
- 然后，计划被编译生成JobGraph。这个图是要交给flink去生成task的图。
- 生成一系列配置。
- 将JobGraph和配置交给flink集群去运行。如果不是本地运行的话，还会把jar文件通过网络发给其他节点。

- 以本地模式运行的话，可以看到启动过程，如启动性能度量、web模块、JobManager、ResourceManager、taskManager等等。

当我们看到了 `submitJob` 调用，就知道KMeans代码已经和Flink构建了联系。

```
@Internal
public class LocalExecutor implements PipelineExecutor {

   public static final String NAME = "local";

   @Override
   public CompletableFuture<JobClient> execute(Pipeline pipeline, Configuration configuration)
throws Exception {

      // we only support attached execution with the local executor.
      checkState(configuration.getBoolean(DeploymentOptions.ATTACHED));

      final JobGraph jobGraph = getJobGraph(pipeline, configuration);
      final MiniCluster miniCluster = startMiniCluster(jobGraph, configuration);
      final MiniClusterClient clusterClient = new MiniClusterClient(configuration, miniCluster)
;

      CompletableFuture<JobID> jobIdFuture = clusterClient.submitJob(jobGraph);

      jobIdFuture
            .thenCompose(clusterClient::requestJobResult)
            .thenAccept((jobResult) -> clusterClient.shutDownCluster());

      return jobIdFuture.thenApply(jobID ->
            new ClusterClientJobClientAdapter<>(() -> clusterClient, jobID));
   }
```

5.2 生成JobGraph

生成jobGraph的具体流程是：

- IterativeDataSet.closeWith会生成一个BulkIterationResultSet。
- PrintBatchOp.sinkFrom中会调用到ExecutionEnvironment.executeAsync
- 调用createProgramPlan构建一个Plan
- OperatorTranslation.translate函数发现 `if (dataSet instanceof BulkIterationResultSet)` ，则调用 `translateBulkIteration(bulkIterationResultSet);`
- 这时候生成了执行计划Plan
- ExecutionEnvironment.executeAsync调用LocalExecutor.execute
- 然后调用FlinkPipelineTranslationUtil.getJobGraph来生成jobGraph
- GraphCreatingVisitor.preVisit中会判断 `if (c instanceof BulkIterationBase)` ，以生成BulkIterationNode
- PlanTranslator.translateToJobGraph会调用到JobGraphGenerator.compileJobGraph，最终调用到createBulkIterationHead就生成了迭代处理的Head。
- 最后将jobGraph提交给Cluster ,jobGraph 变形为 ExceutionGraph在JM和TM上执行。

5.3 迭代对应的Task

前面代码中，getJobGraph函数作用是生成了job graph。

然后 JobManager 根据 JobGraph 生成 ExecutionGraph。ExecutionGraph 是 JobGraph 的并行化版本，是调度层最核心的数据结构。

最后 JobManager 根据 ExecutionGraph 对 Job 进行调度后，在各个TaskManager 上部署 Task。

所以我们需要看看最终运行时候，迭代API对应着哪些Task。

针对IterativeDataSet，即superstep-based Bulk Iterate，Flink生成了如下的task。

- IterationHeadTask
- IterationIntermediateTask
- IterationTailTask
- IterationSynchronizationSinkTask

5.3.1 IterationHeadTask

IterationHeadTask主要作用是协调一次迭代。

它会读取初始输入，和迭代Tail建立一个BlockingBackChannel。在成功处理输入之后，它会发送EndOfSuperstep事件给自己的输出。它在每次superstep之后会联系 synchronization task，等到自己收到一个用来同步的AllWorkersDoneEvent。AllWorkersDoneEvent表示所有其他的heads已经完成了自己的迭代。

下一次迭代时候，上一次迭代中tail的输出就经由backchannel传输，形成了head的输入。何时进入到下一个迭代，是由HeadTask完成的。一旦迭代完成，head将发送TerminationEvent给所有和它关联的task，告诉他们shutdown。

```
                                barrier.waitForOtherWorkers();

                                if (barrier.terminationSignaled()) {
                                        requestTermination();
                                        nextStepKickoff.signalTermination();
                                } else {
                                        incrementIterationCounter();
                                        String[] globalAggregateNames = barrier.getAggregatorNa
mes();
                                        Value[] globalAggregates = barrier.getAggregates();
                                        aggregatorRegistry.updateGlobalAggregatesAndReset(globa
lAggregateNames, globalAggregates);
          // 在这里发起下一次Superstep。
                                        nextStepKickoff.triggerNextSuperstep();
                                }
                        }
```

IterationHeadTask是在JobGraphGenerator.createBulkIterationHead中构建的。其例子如下：

```
"PartialSolution (Bulk Iteration) (org.apache.flink.runtime.iterative.task.IterationHeadTask)"
```

5.3.2 IterationIntermediateTask

IterationIntermediateTask是superstep中间段的task，其将传输EndOfSuperstepEvent和TerminationEvent给所有和它关联的tasks。此外，IterationIntermediateTask能更新the workset或者the solution set的迭代状态。

如果迭代状态被更新，本task的输出将传送回IterationHeadTask，在这种情况下，本task将作为head再次被安排。

IterationIntermediateTask的例子如下：

```
 "MapPartition (computation@KMeansUpdateCentroids) (org.apache.flink.runtime.iterative.task.Ite
rationIntermediateTask)"

 "Combine (SUM(0), at kMeansPlusPlusInit(KMeansInitCentroids.java:135) (org.apache.flink.runtim
e.iterative.task.IterationIntermediateTask)"
```

```
 "MapPartition (AllReduceSend) (org.apache.flink.runtime.iterative.task.IterationIntermediateTask)"

"Filter (Filter at kMeansPlusPlusInit(KMeansInitCentroids.java:130)) (org.apache.flink.runtime.iterative.task.IterationIntermediateTask)"
```

5.3.3 IterationTailTask

IterationTailTask是迭代的最末尾。如果迭代状态被更新，本task的输出将通过BlockingBackChannel传送回IterationHeadTask，反馈给迭代头就意味着一个迭代完整逻辑的完成，那么就可以关闭这个迭代闭合环了。这种情况下，本task将在head所在的实例上重新被调度。

这里有几个关键点需要注意：

如何和Head建立联系

Flink有一个BlockingQueueBroker类，这是一个阻塞式的队列代理，它的作用是对迭代并发进行控制。Broker是单例的，迭代头任务和尾任务会生成同样的broker ID，所以头尾在同一个JVM中会基于相同的dataChannel进行通信。dataChannel由迭代头创建。

IterationHeadTask中会生成BlockingBackChannel，这是一个容量为1的阻塞队列。

```
// 生成channel
BlockingBackChannel backChannel = new BlockingBackChannel(new SerializedUpdateBuffer(segments, segmentSize, this.getIOManager()));

// 然后block在这里，等待Tail
superstepResult = backChannel.getReadEndAfterSuperstepEnded();
```

IterationTailTask则是如下：

```
// 在基类得到channel，因为是单例，所以会得到同一个
worksetBackChannel = BlockingBackChannelBroker.instance().getAndRemove(brokerKey());

// notify iteration head if responsible for workset update 在这里通知Head
worksetBackChannel.notifyOfEndOfSuperstep();
```

而两者都是利用如下办法来建立联系，在同一个subtask中会使用同一个brokerKey，这样首尾就联系起来了。

```
public String brokerKey() {
    if (this.brokerKey == null) {
        int iterationId = this.config.getIterationId();
        this.brokerKey = this.getEnvironment().getJobID().toString() + '#' + iterationId + '#' + this.getEnvironment().getTaskInfo().getIndexOfThisSubtask();
    }

    return this.brokerKey;
}
```

如何把用户返回的数值传给Head

这是通过output.collect来完成的。

首先，在Tail初始化时候，会生成一个outputCollector，这个outputCollector会被设置为本task的输出outputCollector。这样就保证了用户函数的输出都会转流到outputCollector。

而outputCollector的输出就是worksetBackChannel的输出，这里设置为同一个instance。这样用户输出就输出到backChannel中。

```
        @Override
        protected void initialize() throws Exception {
                super.initialize();

                // set the last output collector of this task to reflect the iteration tail sta
te update:
                // a) workset update,
                // b) solution set update, or
                // c) merged workset and solution set update

                Collector<OT> outputCollector = null;
                if (isWorksetUpdate) {
      // 生成一个outputCollector
                        outputCollector = createWorksetUpdateOutputCollector();

                        // we need the WorksetUpdateOutputCollector separately to count the col
lected elements
                        if (isWorksetIteration) {
                                worksetUpdateOutputCollector = (WorksetUpdateOutputCollector<OT
>) outputCollector;
                        }
                }

    ......
    // 把outputCollector设置为本task的输出
                setLastOutputCollector(outputCollector);
        }
```

outputCollector的输出就是worksetBackChannel的输出buffer，这里设置为同一个instance。

```
        protected Collector<OT> createWorksetUpdateOutputCollector(Collector<OT> delegate) {
                DataOutputView outputView = worksetBackChannel.getWriteEnd();
                TypeSerializer<OT> serializer = getOutputSerializer();
                return new WorksetUpdateOutputCollector<OT>(outputView, serializer, delegate);
        }
```

运行时候如下：

```
        @Override
        public void run() throws Exception {

                SuperstepKickoffLatch nextSuperStepLatch = SuperstepKickoffLatchBroker.instance
().get(brokerKey());

                while (this.running && !terminationRequested()) {

      // 用户在这里输出，最后会输出到output.collect，也就是worksetBackChannel的输出buffer。
                        super.run();

      // 这时候以及输出到channel完毕，只是通知head进行读取。
                        if (isWorksetUpdate) {
                                // notify iteration head if responsible for workset update
                                worksetBackChannel.notifyOfEndOfSuperstep();
                        } else if (isSolutionSetUpdate) {
                                // notify iteration head if responsible for solution set update
                                solutionSetUpdateBarrier.notifySolutionSetUpdate();
                        }
```

```
    ...
      }
```

IterationTailTask例子如下：

```
"Pipe (org.apache.flink.runtime.iterative.task.IterationTailTask)"
```

5.3.4 IterationSynchronizationSinkTask

IterationSynchronizationSinkTask作用是同步所有的iteration heads，IterationSynchronizationSinkTask被是实现成一个 output task。其只是用来协调，不处理任何数据。

在每一次superstep，IterationSynchronizationSinkTask只是等待直到它从每一个head都收到一个WorkerDoneEvent。这表示下一次superstep可以开始了。

这里需要注意的是 SynchronizationSinkTask 如何等待各个并行度的headTask。比如Flink的并行度是5，那么SynchronizationSinkTask怎么做到等待这5个headTask。

在IterationSynchronizationSinkTask中，注册了SyncEventHandler来等待head的WorkerDoneEvent。

```
this.eventHandler = new SyncEventHandler(numEventsTillEndOfSuperstep, this.aggregators, this.ge
tEnvironment().getUserClassLoader());
this.headEventReader.registerTaskEventListener(this.eventHandler, WorkerDoneEvent.class);
```

在SyncEventHandler中，我们可以看到，在构建时候，numberOfEventsUntilEndOfSuperstep就被设置为并行度，每次收到一个WorkerDoneEvent，workerDoneEventCounter就递增，当等于numberOfEventsUntilEndOfSuperstep，即并行度时候，就说明本次superstep中，所有headtask都成功了。

```
    private void onWorkerDoneEvent(WorkerDoneEvent workerDoneEvent) {
        if (this.endOfSuperstep) {
            throw new RuntimeException("Encountered WorderDoneEvent when still in End-of-Supers
tep status.");
        } else {
          // 每次递增
            ++this.workerDoneEventCounter;
            String[] aggNames = workerDoneEvent.getAggregatorNames();
            Value[] aggregates = workerDoneEvent.getAggregates(this.userCodeClassLoader);
            if (aggNames.length != aggregates.length) {
                throw new RuntimeException("Inconsistent WorkerDoneEvent received!");
            } else {
                for(int i = 0; i < aggNames.length; ++i) {
                    Aggregator<Value> aggregator = (Aggregator)this.aggregators.get(aggNames[i]
);

                    aggregator.aggregate(aggregates[i]);
                }

              // numberOfEventsUntilEndOfSuperstep就是并行度，等于并行度时候就说明所有head都成功了。
                if (this.workerDoneEventCounter % this.numberOfEventsUntilEndOfSuperstep == 0)
{
                    this.endOfSuperstep = true;
                    Thread.currentThread().interrupt();
                }
            }
        }
    }
```

IterationSynchronizationSinkTask的例子如下：

```
"Sync (BulkIteration (Bulk Iteration)) (org.apache.flink.runtime.iterative.task.IterationSynchr
```

```
onizationSinkTask)"
```

5.4 superstep

综上所述，我们最终得到superstep如下：

```
***** 文字描述如下 *****

每次迭代都是一个superstep
  每次迭代中有若干subtask在不同的partition上分别执行step
     每个step有一个HeadTask，若干IntermediateTask，一个TailTask
  每个superstep有一个SynchronizationSinkTask

***** 伪代码大致如下 *****

for maxIter :
  begin superstep
      for maxSubTask :
         begin step
           IterationHeadTask
           IterationIntermediateTask
           IterationIntermediateTask
           ...
           IterationIntermediateTask
           IterationIntermediateTask
           IterationTailTask
         end step
    IterationSynchronizationSinkTask
  end superstep
```

0x06 结合KMeans代码看superset

6.1 K-means算法概要

K-means算法的过程，为了尽量不用数学符号，所以描述的不是很严谨，大概就是这个意思，“物以类聚、人以群分”：

1. 首先输入k的值，即我们希望将数据集经过聚类得到k个分组。
2. 从数据集中随机选择k个数据点作为初始大哥（质心，Centroid）
3. 对集合中每一个小弟，计算与每一个大哥的距离（距离的含义后面会讲），离哪个大哥距离近，就跟定哪个大哥。
4. 这时每一个大哥手下都聚集了一票小弟，这时候召开人民代表大会，每一群选出新的大哥（其实是通过算法选出新的质心）。
5. 如果新大哥和老大哥之间的距离小于某一个设置的阈值（表示重新计算的质心的位置变化不大，趋于稳定，或者说收敛），可以认为我们进行的聚类已经达到期望的结果，算法终止。
6. 如果新大哥和老大哥距离变化很大，需要迭代3~5步骤。

6.2 KMeansPreallocateCentroid

KMeansPreallocateCentroid也是superstep一员，但是只有 `context.getStepNo() == 1` 的时候，才会进入实际业务逻辑，预分配Centroid。当superstep为大于1的时候，本task会执行，但不会进入具体业务代码。

```
public class KMeansPreallocateCentroid extends ComputeFunction {
    private static final Logger LOG = LoggerFactory.getLogger(KMeansPreallocateCentroid.class);

    @Override
    public void calc(ComContext context) {
        // 每次superstep都会进到这里
```

```
        LOG.info("  KMeansPreallocateCentroid 我每次都会进的呀    ");
        if (context.getStepNo() == 1) {
          // 实际预分配业务只进入一次
        }
    }
}
```

6.3 KMeansAssignCluster 和 KMeansUpdateCentroids

KMeansAssignCluster 作用是为每个点(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。

KMeansUpdateCentroids 作用是基于计算出来的点计数和坐标，计算新的聚类中心。

Alink在整个计算过程中维护一个特殊节点来记住待求中心点当前的结果。

这就是为啥迭代时候需要区分奇数次和偶数次的原因了。奇数次就表示老大哥，偶数次就表示新大哥。每次superstep只会计算一批大哥，留下另外一批大哥做距离比对。

另外要注意的一点是：普通的迭代计算，是通过Tail给Head回传用户数据，但是KMeans这里的实现并没有采用这个办法，而是把计算出来的中心点都存在共享变量中，在各个intermediate之间互相交互。

```
public class KMeansAssignCluster extends ComputeFunction {
    public void calc(ComContext context) {
        ......
        if (context.getStepNo() % 2 == 0) {
            stepNumCentroids = context.getObj(KMeansTrainBatchOp.CENTROID1);
        } else {
            stepNumCentroids = context.getObj(KMeansTrainBatchOp.CENTROID2);
        }
      /** 具体业务逻辑代码
       * Find the closest cluster for every point and calculate the sums of the points belongin
g to the same cluster.
       */
    }
}

public class KMeansUpdateCentroids extends ComputeFunction {
    public void calc(ComContext context) {
        if (context.getStepNo() % 2 == 0) {
            stepNumCentroids = context.getObj(KMeansTrainBatchOp.CENTROID2);
        } else {
            stepNumCentroids = context.getObj(KMeansTrainBatchOp.CENTROID1);
        }
      /** 具体业务逻辑代码
       * Update the centroids based on the sum of points and point number belonging to the same
 cluster.
       */
    }
```

6.4 KMeansOutputModel

这里要特殊说明，因为KMeansOutputModel是最终输出模型，而KMeans算法的实现是：所有subtask都拥有所有中心点，就是说所有subtask都会有相同的模型，就没有必要全部输出，所以这里限定了第一个subtask才能输出，其他的都不输出。

```
        @Override
        public List <Row> calc(ComContext context) {
    // 只有第一个subtask才输出模型数据。
                if (context.getTaskId() != 0) {
                        return null;
```

```
            }

....

            modelData.params = new KMeansTrainModelData.ParamSummary();
            modelData.params.k = k;
            modelData.params.vectorColName = vectorColName;
            modelData.params.distanceType = distanceType;
            modelData.params.vectorSize = vectorSize;
            modelData.params.latitudeColName = latitudeColName;
            modelData.params.longtitudeColName = longtitudeColName;

            RowCollector collector = new RowCollector();
            new KMeansModelDataConverter().save(modelData, collector);
            return collector.getRows();
    }
```

0x07 参考

几种并行计算模型的区别(BSP LogP PRAM)

https://ci.apache.org/projects/flink/flink-docs-release-1.10/dev/batch/iterations.html
聚类、K-Means、例子、细节

Flink-Gelly：Iterative Graph Processing

从BSP模型到Apache Hama

Flink DataSet迭代运算

几种并行计算模型的区别(BSP LogP PRAM)

Flink架构，源码及debug

Flink 之 Dataflow、Task、subTask、Operator Chains、Slot 介绍

Flink 任务和调度

Flink运行时之生成作业图