Flink 作业执行解析

岳猛 · 网易云音乐/ 实时计算开发

Apache Flink Community China

Apache Flink

CONTENT

目录 >>

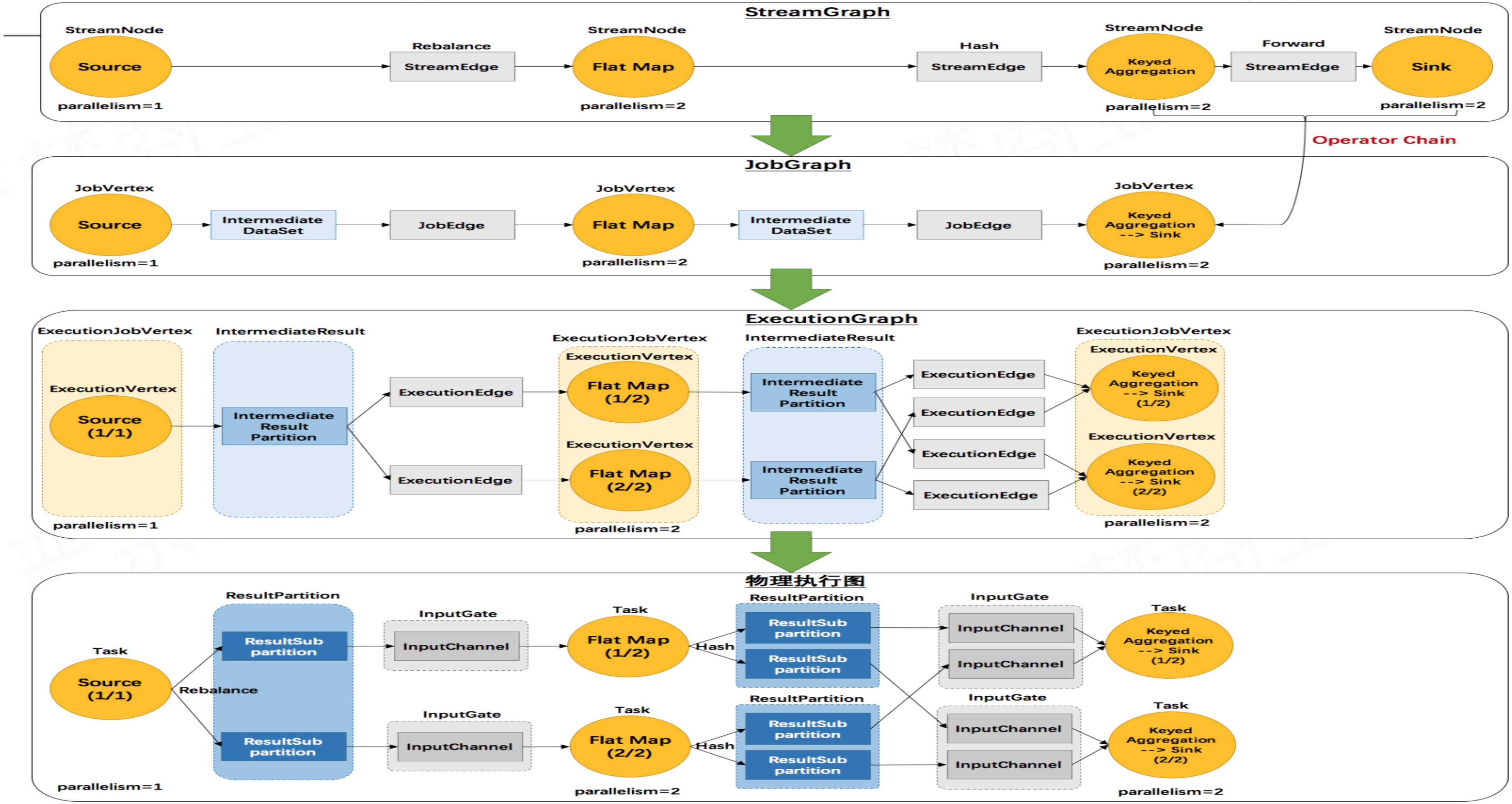
01

Flink 4层转化流程

Apache Flink

第一层:Program –> StreamGraph
第二层：StreamGraph –> JobGraph
第三层：JobGraph –> ExecutionGraph
第四层：Execution –> 物理执行计划

4层转化流程



StreamGraph
StreamNode
Source
parallelism=1
Rebalance
StreamEdge
StreamNode
Flat Map
parallelism=2
Hash
StreamEdge
StreamNode
Keyed Aggregation
parallelism=2
Forward
StreamEdge
StreamNode
Sink
parallelism=2
Operator Chain
JobGraph
JobVertex
Source
parallelism=1
Intermediate DataSet
JobEdge
JobVertex
Flat Map
parallelism=2
Intermediate DataSet
JobEdge
JobVertex
Keyed Aggregation --> Sink
parallelism=2
ExecutionGraph
ExecutionJobVertex
ExecutionVertex
Source (1/1)
parallelism=1
IntermediateResult
Intermediate Result Partition
ExecutionEdge
ExecutionEdge
ExecutionJobVertex
ExecutionVertex
Flat Map (1/2)
ExecutionVertex
Flat Map (2/2)
parallelism=2
IntermediateResult
Intermediate Result Partition
Intermediate Result Partition
ExecutionEdge
ExecutionEdge
ExecutionEdge
ExecutionEdge
ExecutionJobVertex
ExecutionVertex
Keyed Aggregation --> Sink (1/2)
ExecutionVertex
Keyed Aggregation --> Sink (2/2)
parallelism=2
物理执行图
Task
Source (1/1)
parallelism=1
Rebalance
ResultPartition
ResultSub partition
ResultSub partition
InputGate
InputChannel
InputGate
InputChannel
Task
Flat Map (1/2)
Task
Flat Map (2/2)
parallelism=2
Hash
Hash
ResultPartition
ResultSub partition
ResultSub partition
ResultPartition
ResultSub partition
ResultSub partition
InputGate
InputChannel
InputChannel
InputGate
InputChannel
InputChannel
Task
Keyed Aggregation --> Sink (1/2)
Task
Keyed Aggregation --> Sink (2/2)
parallelism=2

如何转换成StreamGraph

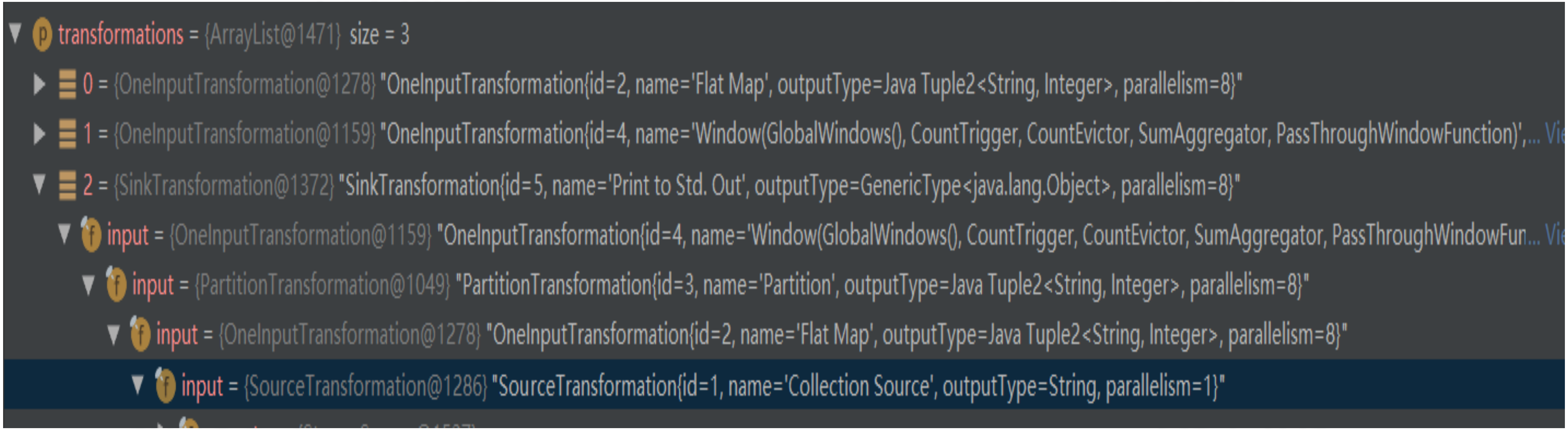
- 从StreamExecutionEnvironment.execute开始执行程序，将transform添加到StreamExecutionEnvironment的transformations
- 调用StreamGraphGenerator的generateInternal方法，遍历transformations构建StreamNode及StreamEage
- 通过streaEdge连接StreamNode

WindowWordCount 3层图转化

Apache Flink

```java
public class WindowWordCount {

    // *************************************************************************
    // PROGRAM
    // *************************************************************************

    public static void main(String[] args) throws Exception {

        final ParameterTool params = ParameterTool.fromArgs(args);

        // set up the execution environment
        final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutionEnvironment();

        // get input data
        DataStream<String> text = env.readTextFile(params.get("input")).setParallelism(2);

        // make parameters available in the web interface
        env.getConfig().setGlobalJobParameters(params);

        final int windowSize = params.getInt("window", 10);
        final int slideSize = params.getInt("slide", 5);

        DataStream<Tuple2<String, Integer>> counts =
        // split up the lines in pairs (2-tuples) containing: (word,1)
        text.flatMap(new WordCount.Tokenizer()).setParallelism(4).slotSharingGroup("flatMap_sg")
                // create windows of windowSize records slided every slideSize records
                .keyBy(0)
                .countWindow(windowSize, slideSize)
                // group by the tuple field "0" and sum up tuple field "1"
                .sum(1).setParallelism(3).slotSharingGroup("sum_sg");

        // emit result
        counts.print().setParallelism(3);

        // execute program
        env.execute("WindowWordCount");
    }
}
```

Apache Flink



Apache Flink

StreamNode及StreamEdge

StreamNode
描述operator的逻辑节点

- 关键成员变量
- slotSharingGroup
- jobVertexClass
- inEdges
- outEdges
- transformationUID

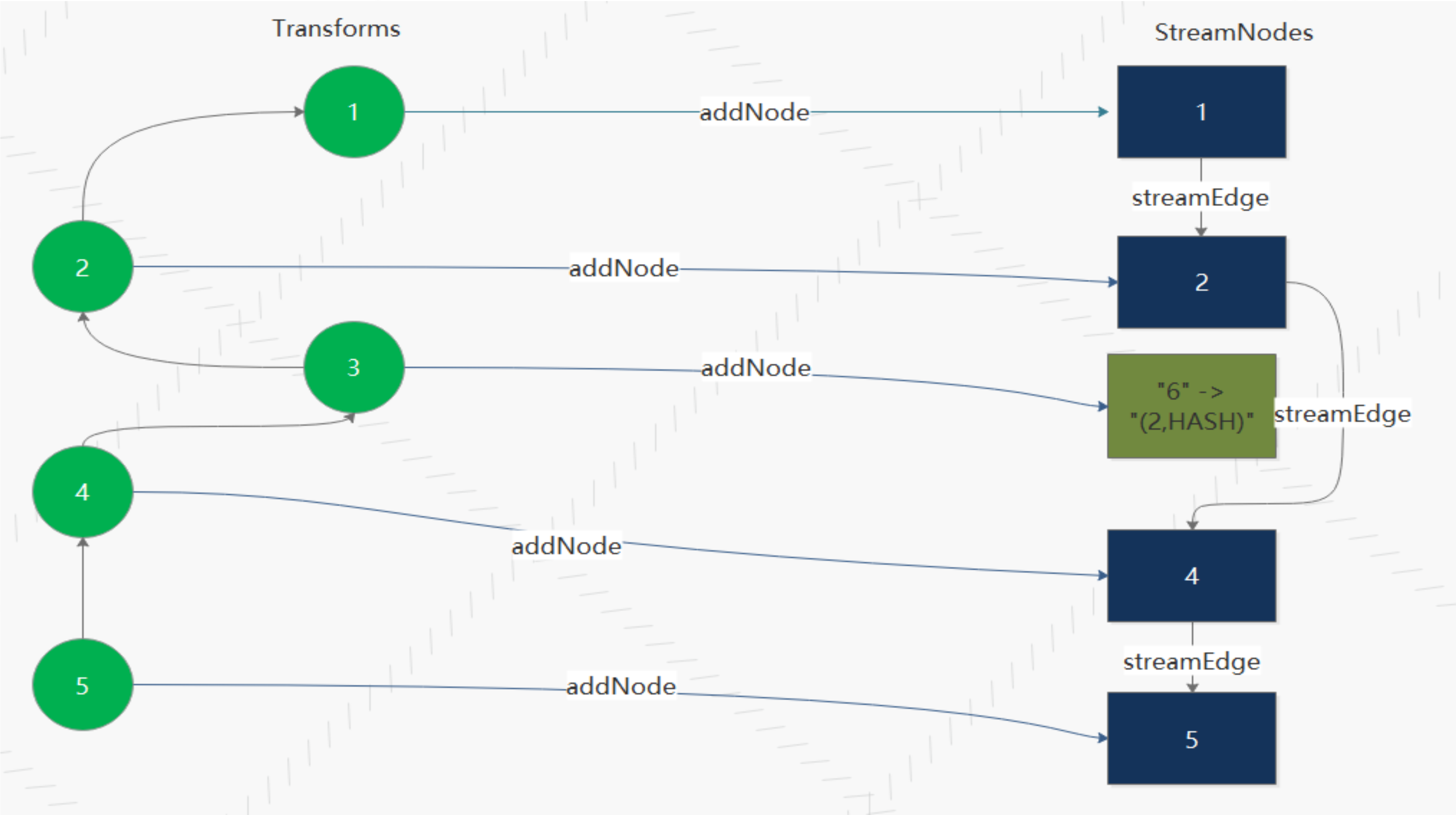
StreamEdge

描述两个operator逻辑的链接边
关键变量

- sourceVertex
- targetVertex

WindowWordCount transform 到StreamGraph转化

Apache Flink



Apache Flink

```
▼ this = {StreamGraph@1178}
  ▶ jobName = "WindowWordCount"
  ▶ environment = {LocalStreamEnvironment@1044}
  ▶ executionConfig = {ExecutionConfig@1181}
  ▶ checkpointConfig = {CheckpointConfig@1179}
    chaining = true
  ▼ streamNodes = {HashMap@1182} size = 4
    ▶ 0 = {HashMap$Node@1198} "1" -> "Source: Collection Source-1"
    ▼ 1 = {HashMap$Node@1199} "2" -> "Flat Map-2"
      ▶ key = {Integer@1204} 2
      ▶ value = {StreamNode@1205} "Flat Map-2"
    ▶ 2 = {HashMap$Node@1200} "4" -> "Window(GlobalWindows(), CountTrigger, CountEvictor, SumAggregator, PassThroughWindowFunction)-4"
    ▶ 3 = {HashMap$Node@1201} "5" -> "Sink: Print to Std. Out-5"
```

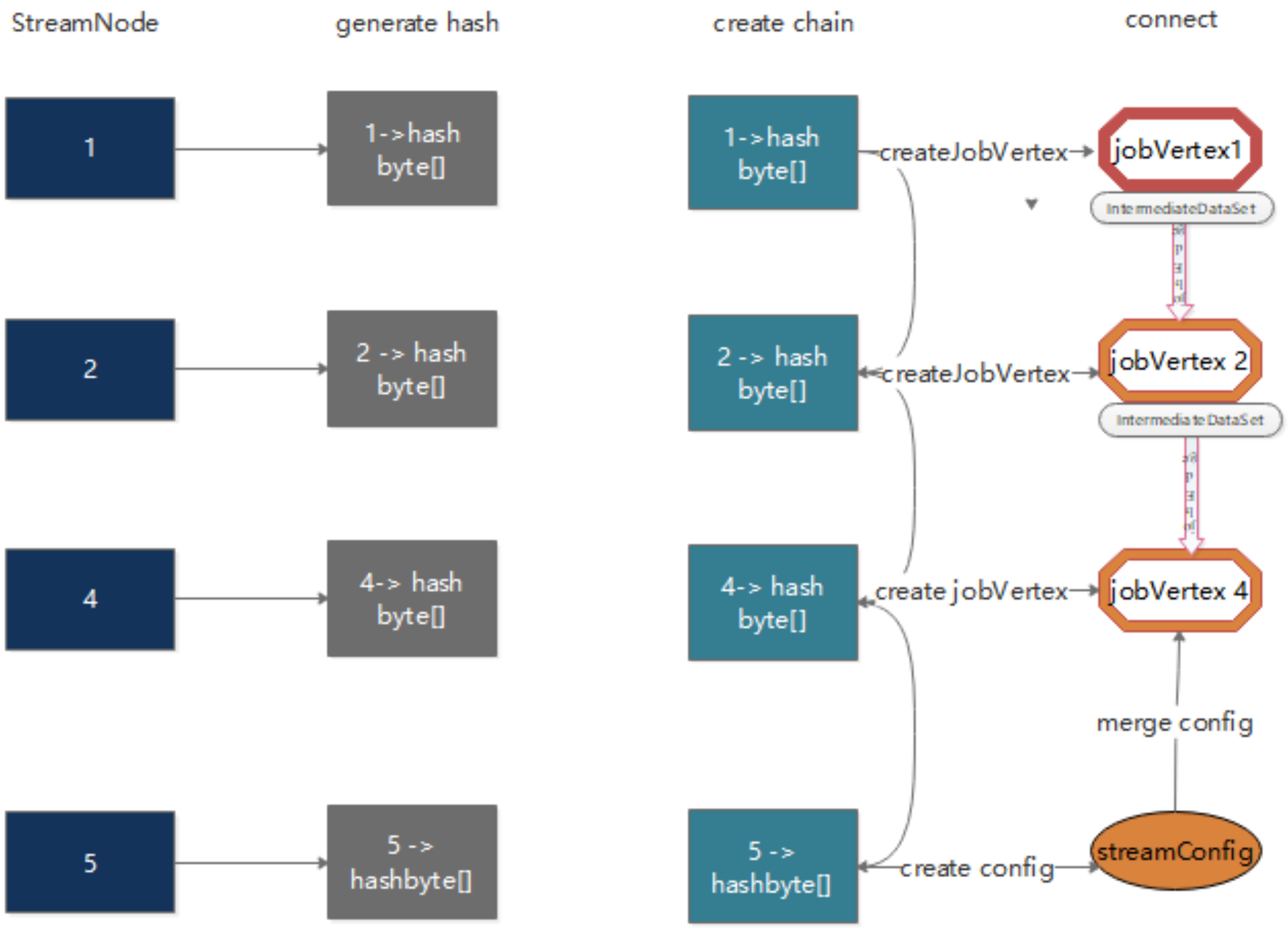
Apache Flink

```
▼ 1 = {HashMap$Node@1199} "2" -> "Flat Map-2"
  ▶ key = {Integer@1204} 2
  ▼ value = {StreamNode@1205} "Flat Map-2"
    ▶ env = {LocalStreamEnvironment@1044}
      id = 2
    ▶ parallelism = {Integer@1215} 8
      maxParallelism = -1
    ▶ minResources = {ResourceSpec@1216} "ResourceSpec{cpuCores=0.0, heapMemoryInMB=0, directMemoryInMB=0, nativeMemoryInMB=0, stateSizeInMB=0}"
    ▶ preferredResources = {ResourceSpec@1216} "ResourceSpec{cpuCores=0.0, heapMemoryInMB=0, directMemoryInMB=0, nativeMemoryInMB=0, stateSizeInMB=0}"
      bufferTimeout = null
    ▶ operatorName = "Flat Map"
    ▶ slotSharingGroup = "flatMap"
      coLocationGroup = null
      statePartitioner1 = null
      statePartitioner2 = null
      stateKeySerializer = null
    ▶ operator = {StreamFlatMap@1219}
      outputSelectors = {ArrayList@1220} size = 0
    ▶ typeSerializerIn1 = {StringSerializer@1221}
      typeSerializerIn2 = null
    ▶ typeSerializerOut = {TupleSerializer@1222}
    ▶ inEdges = {ArrayList@1223} size = 1
    ▶ outEdges = {ArrayList@1224} size = 1
    ▶ jobVertexClass = {Class@1225} "class org.apache.flink.streaming.runtime.tasks.OneInputStreamTask" ... Navigate
      inputFormat = null
```

StreamGraph到JobGraph的转化

具体步骤

- 设置调度模式，Eager所有节点立即启动
- 广度优先遍历StreamGraph，为每个streamNode生成byte数组类型的hash值
- 从source节点开始递归寻找chain到一起的operator，不能chain到一起的节点单独生成jobVertex，能够chaind到一起的,开始节点生成jobVertex,其他节点以序列化的形式写入到StreamConfig，然后merge到CHAINED_TASK_CONFIG ，然后通过JobEdge链接上下游JobVertex
- 将每个JobVertex的入边(StreamEdge)序列化到该StreamConfig
- 根据group name为每个JobVertext指定SlotSharingGroup
- 配置checkpoint
- 将缓存文件存文件的配置添加到configuration中
- 设置置ExecutionConfig


StreamNode
generate hash
create chain
connect
1
1->hash byte[]
1->hash byte[]
createJobVertex
jobVertex1
IntermediateDataSet
2
2 -> hash byte[]
2 -> hash byte[]
createJobVertex
jobVertex 2
IntermediateDataSet
4
4-> hash byte[]
4-> hash byte[]
create jobVertex
jobVertex 4
merge config
5
5 -> hashbyte[]
5 -> hashbyte[]
create config
streamConfig

Apache Flink

Chain的条件?

下游节点只有一个输入

下游节点的操作符不爲null

上游节点的操作符不爲null

上下游节点在一个槽位共享组内

下游节点的连接策略是 ALWAYS

上游节点的连接策略是 HEAD 或者 ALWAYS

edge 的分区函数是 ForwardPartitioner 的实例

上下游节点的并行度相等

可以进行节点连接操作

Apache Flink

```
▼ jobGraph = {JobGraph@1209} "JobGraph(jobId: 8616b72e01737666bf4512f53fb0000b)"
  ▼ taskVertices = {LinkedHashMap@1685} size = 3
    ▶ 0 = {LinkedHashMap$Entry@1704} "e70bbd798b564e0a50e10e343f1ac56b" -> "Window(GlobalWindows(), CountTrigger, CountEvictor, SumAggregator, PassThroughWindowFunction) -> Sink: Print to Std. Out...
    ▼ 1 = {LinkedHashMap$Entry@1705} "0a448493b4782967b150582570326227" -> "Flat Map (org.apache.flink.streaming.runtime.tasks.OneInputStreamTask)"
      ▶ key = {JobVertexID@1580} "0a448493b4782967b150582570326227"
      ▶ value = {JobVertex@1542} "Flat Map (org.apache.flink.streaming.runtime.tasks.OneInputStreamTask)"
    ▶ 2 = {LinkedHashMap$Entry@1706} "bc764cd8ddf7a0cff126f51c16239658" -> "Source: Collection Source (org.apache.flink.streaming.runtime.tasks.SourceStreamTask)"
```

为什么要为每个operator生成hash值?
Flink任务失败的时候，各个operator是能够从checkpoint中恢复到失败之前的状态的，恢复的时候是依据JobVertexID（hash值)进行状态恢复的。相同的任务在恢复的时候要求operator的hash值不变
如何生成?
如果用户对节点指定了一个散列值，则基于用户指定的值，产生一个长度爲16的字节数组；
如果用户没有指定，则根据当前节点所处的位置，产生一个散列值，考虑的因素有：
2.1 在当前StreamNode之前已经处理过的节点的个数，作爲当前StreamNode的id，添加到hasher中；
2.2 遍历当前StreamNode输出的每个StreamEdge，并判断当前StreamNode与这个StreamEdge的目标StreamNode是否可以进行链接，如果可以，则将目标StreamNode的id也放入hasher中，且这个目标StreamNode的id与当前StreamNode的id取相同的值；
2.3 将上述步骤后产生的字节数据，与当前StreamNode的所有输入StreamNode对应的字节数据，进行相应的位操作，最终得到的字节数据，就是当前StreamNode对应的长度爲16的字节数组。
为何不用StreamNode Id?
静态累加器，相同处理逻辑，可以产生不同的id组合

JobGraph到ExexcutionGraph以及物理执行计划

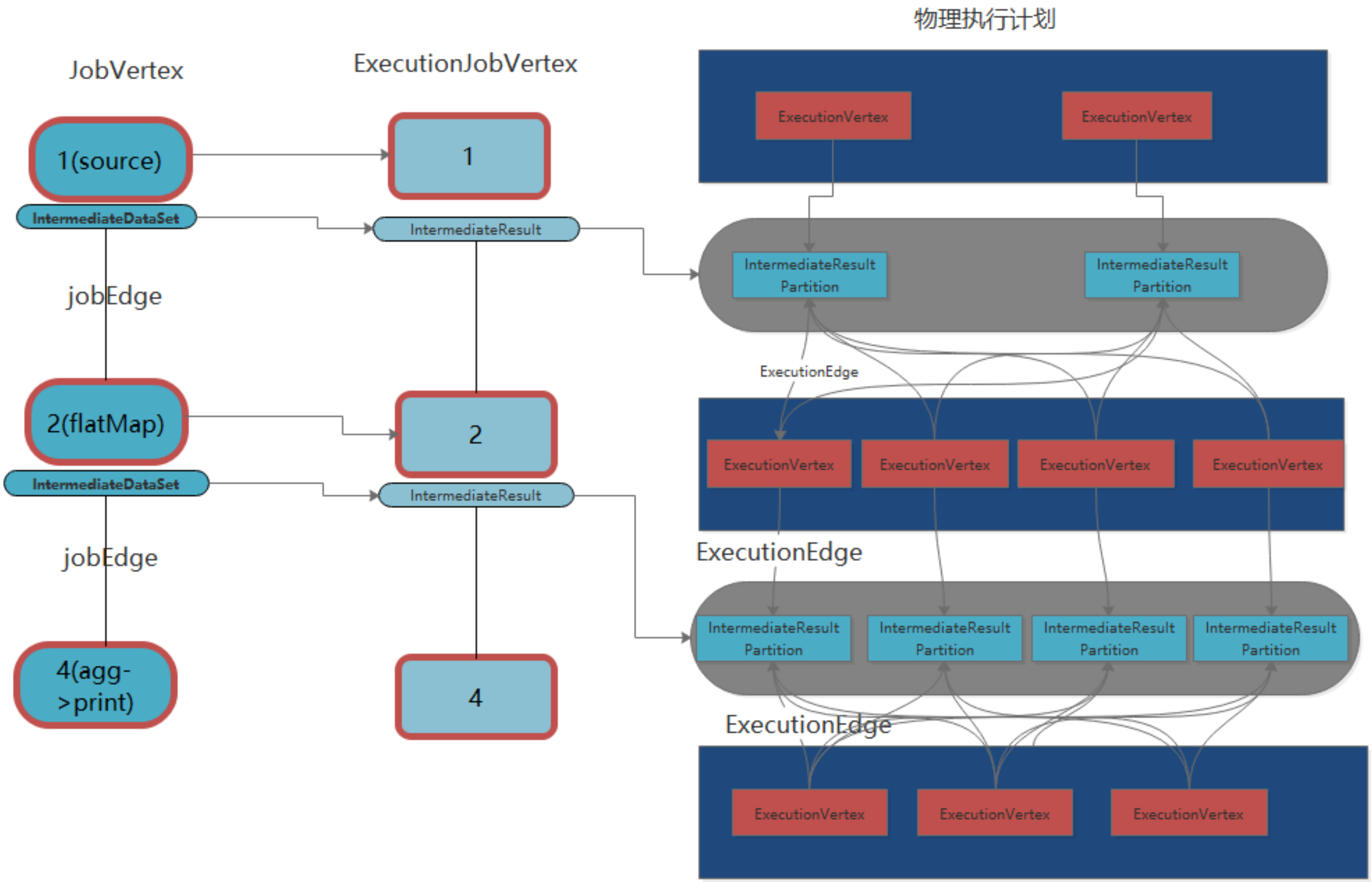
主要ExecutionGraphBuilder的buildGraph方法里面，关键流程

1）将JobGraph里面的jobVertex从source节点开始排序

2）executionGraph.attachJobGraph(sortedTopology)方法里面

- 根据JobVertex生成ExecutionJobVertex，在ExecutionJobVertex构造方法里面，根据jobVertex的IntermediateDataSet构建IntermediateResult，根据jobVertex并发构建ExecutionVertex，ExecutionVertex构建的时候，构建IntermediateResultPartition（每一个Execution构建IntermediateResult个数个IntermediateResultPartition ）
- 将创建的ExecutionJobVertex与前置的IntermediateResult连接起来

3）构建ExecutionEdge ，连接到前面的IntermediateResultPartition



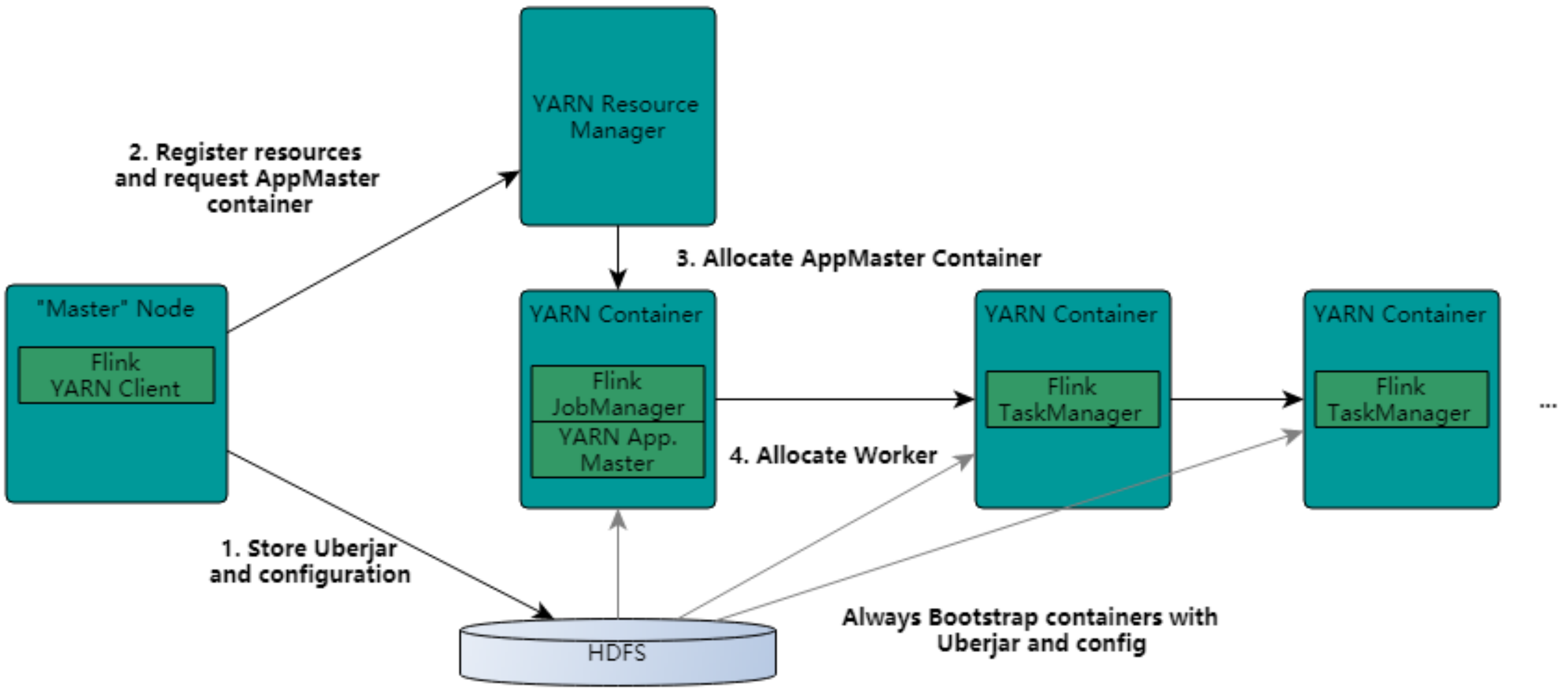
物理执行计划
JobVertex
ExecutionJobVertex
1(source)
1
IntermediateDataSet
IntermediateResult
jobEdge
2(flatMap)
2
IntermediateDataSet
IntermediateResult
jobEdge
4(agg->print)
4
ExecutionVertex
ExecutionVertex
IntermediateResult Partition
IntermediateResult Partition
ExecutionEdge
ExecutionVertex
ExecutionVertex
ExecutionVertex
ExecutionVertex
ExecutionEdge
IntermediateResult Partition
IntermediateResult Partition
IntermediateResult Partition
IntermediateResult Partition
ExecutionEdge
ExecutionVertex
ExecutionVertex
ExecutionVertex

02

Job调度和执行

Flink On Yarn 模式

Apache Flink



缺陷

资源分配是静态的

On-YARN 模式下，所有的 container 都是固定大小的，导致无法根据作业需求来调整 container 的结构

On-YARN 模式下，作业管理页面会在作业完成后消失不可访问

Dispatcher

Dispathcer 是在新设计里引入的一个新概念。Dispatcher 会从 client 端接受作业提交请求并代表它在集群管理器上启动作业。

引入 Dispatcher 的原因是:

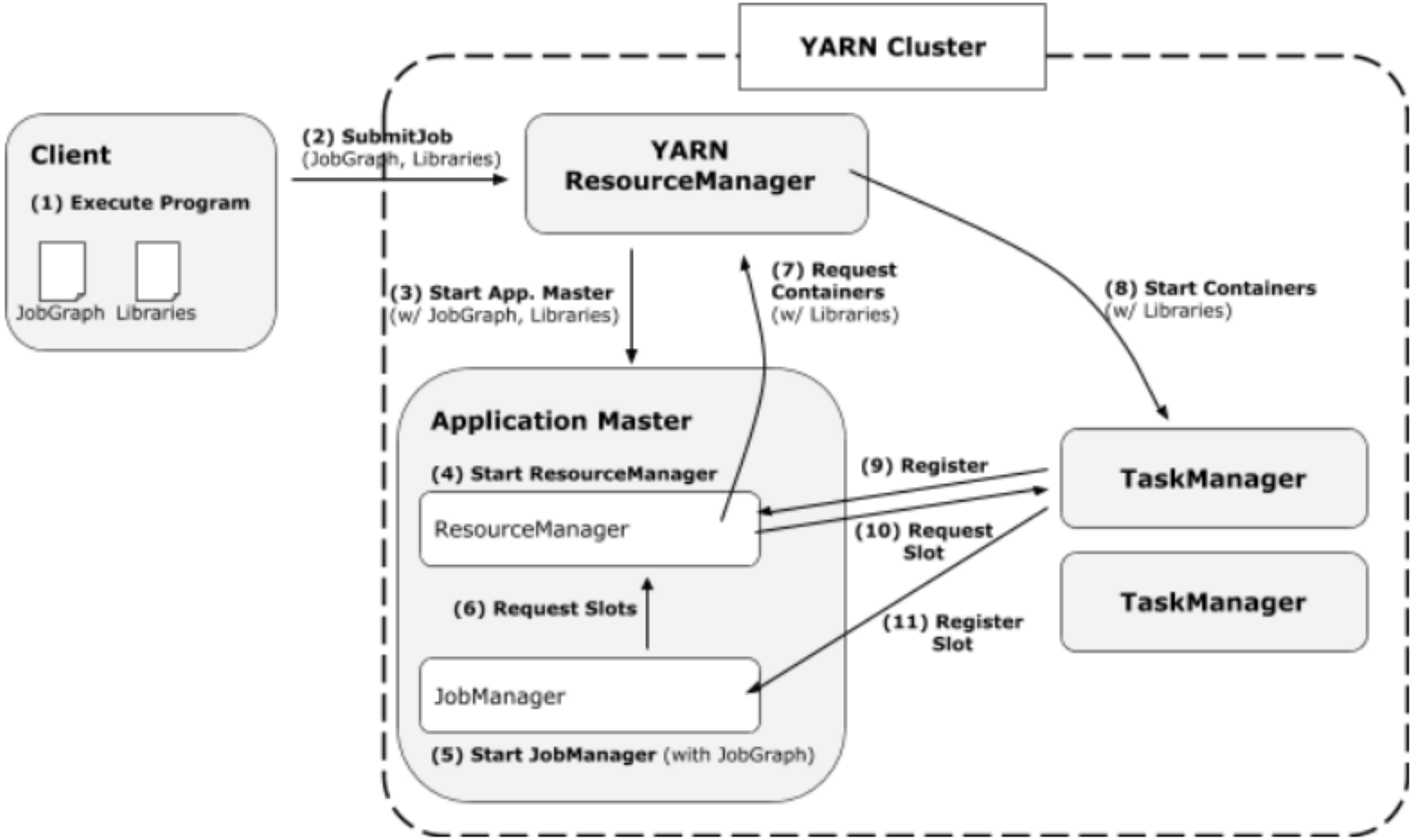
•一些集群管理器需要一个中心化的作业生成和监控实例。

•实现 Standalone 模式下 JobManager 的角色，等待作业提交。

在一些案例中，Dispathcer 是可选的(YARN)或者不兼容的(kubernetes)。

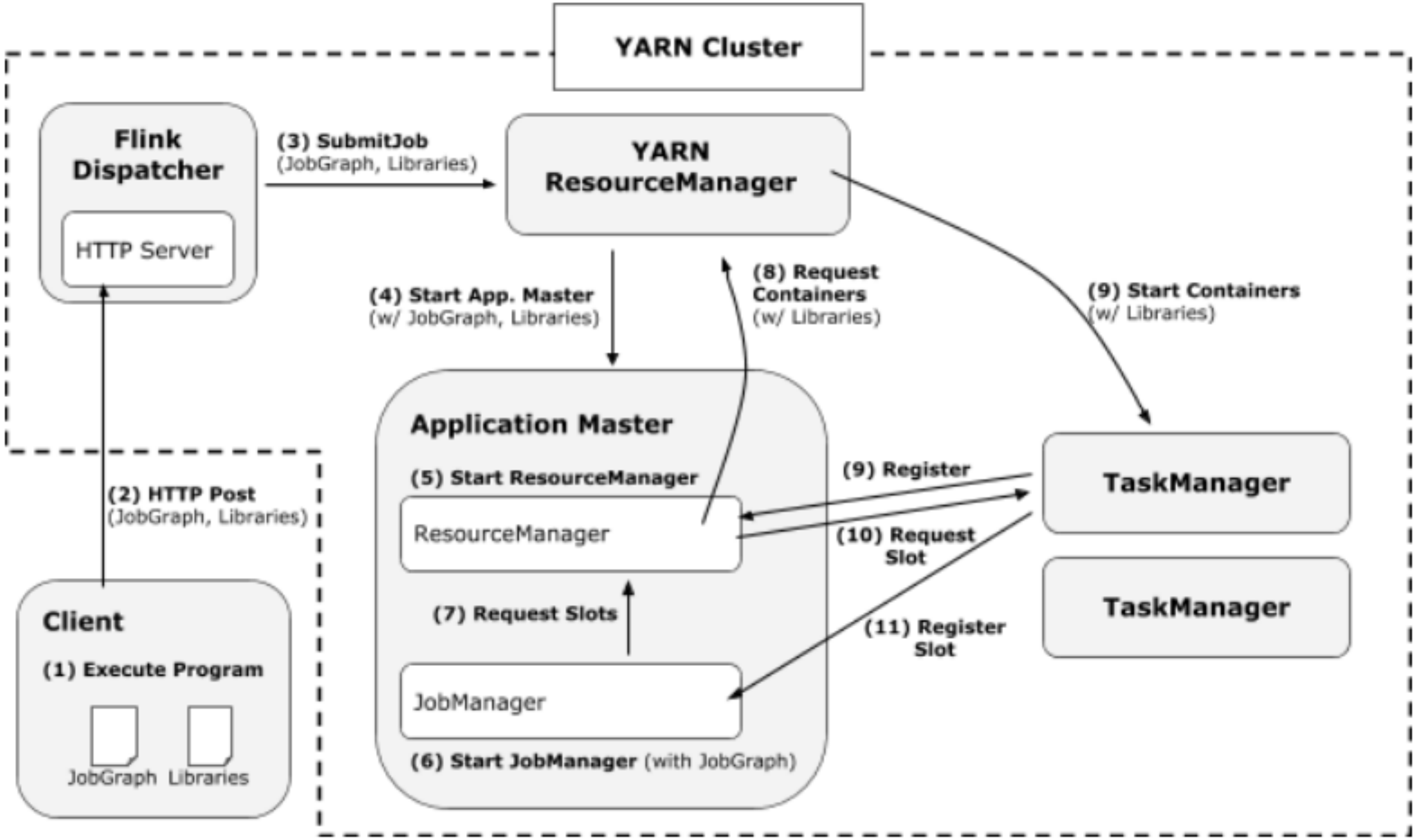
FLIP6: 资源调度模型重构下的Flink On Yarn模式

Without Dispatcher



Apache Flink

With Dispatcher



新框架优势

1.client 直接在 YARN 上启动作业，而不需要先启动一个集群然后再提交作业到集群。因此 client 再提交作业后可以马上返回。

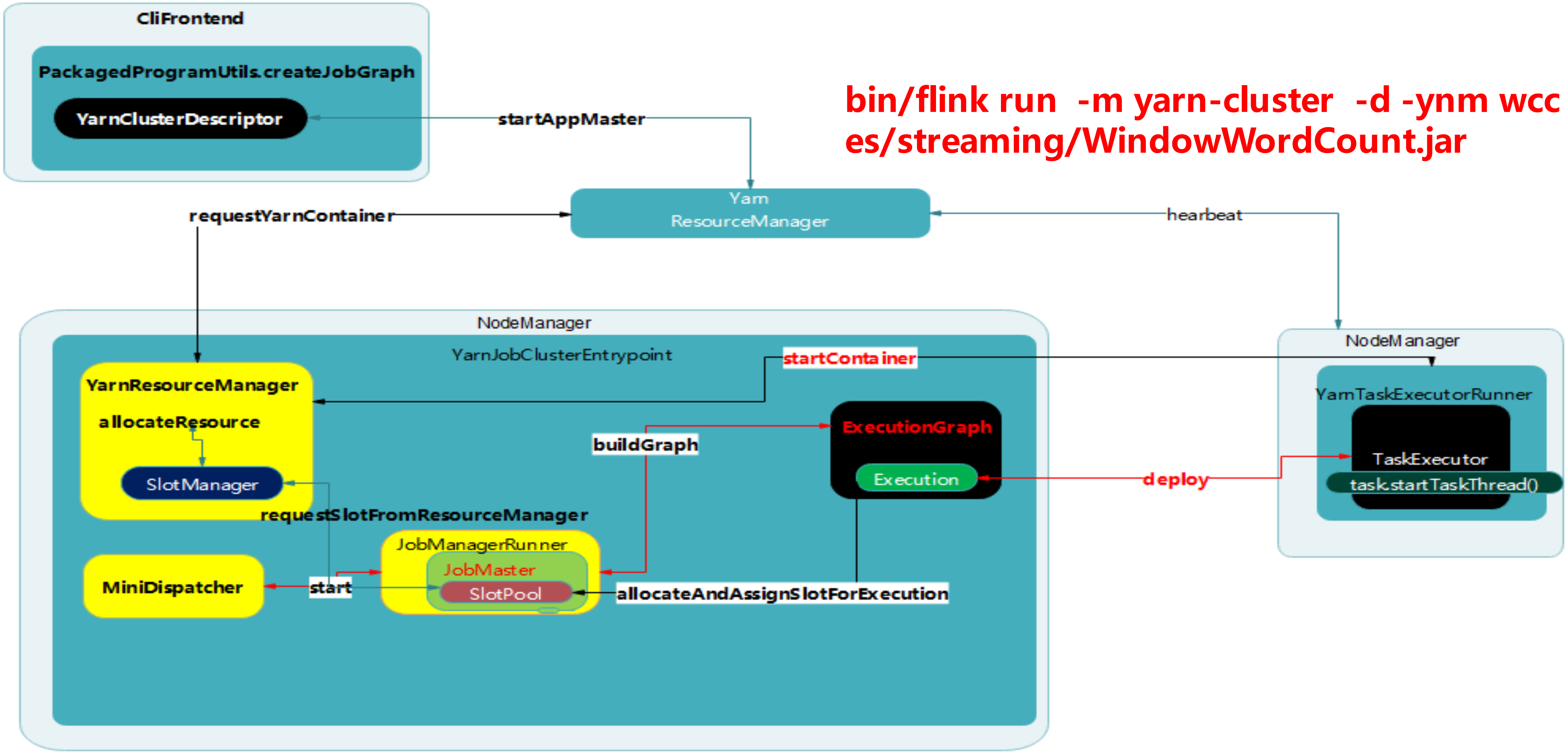
2.所有的用户依赖库和配置文件都被直接放在应用的 classpath，而不是用动态的用户代码 classloader 去加载。

3.container 在需要时才请求，不再使用时会被释放。

4.“需要时申请”的 container 分配方式允许不同算子使用不同 profile (CPU 和内存结构)的 container。

single cluster job on yarn

bin/flink run -m yarn-cluster -d -ynm wcc exampl es/streaming/WindowWordCount.jar



客户端

```
main->
    ->PackagedProgramUtils.createJobGraph
    ->YarnClusterDescriptor.deployJobCluster
        ->deployInternal
            ->startAppMaster
            ->setupSingleLocalResource
```

（AM）
YarnJobClusterEntrypoint

```
main->
runClusterEntrypoint->
startCluster->
runCluster->
   ->dispatcherResourceManagerComponentFactory.create
      -> resourceManagerFactory.createResourceManager
           -> new YarnResourceManager
      -> dispatcherFactory.createDispatcher
           -> jobGraphRetriever.retrieveJobGraph(configuration)
           -> new MiniDispatcher
      -> resourceManager.start()
           -> leaderElectionService.start(this) -> grantLeadership(final UUID newLeaderSessionID)-> slotManager.start
     -> dispatcher.start()
            leaderElectionService.start(this) -> grantLeadership-> tryAcceptLeadershipAndRunJobs -> runJobc
            -> createJobManagerRunner
                -> jobManagerRunnerFactory.createJobManagerRunner
                -> startJobManagerRunner

JobManager
jobManagerRunner.start()
->leaderElectionService.start(this)
->grantLeadership
->verifyJobSchedulingStatusAndStartJobManager
->jobMaster.start

JobMaster
start-> startJobExecution-> resetAndScheduleExecutionGraph
->createAndRestoreExecutionGraph->ExecutionGraphBuilder.buildGraph
                                ->scheduleExecutionGraph->executionGraph.scheduleForExecution

ExecutionGraph
scheduleForExecution->scheduleEager
                        -> ExecutionJobVertex.allocateResourcesForAll
                            -> Execution.allocateAndAssignSlotForExecution -> ProviderAndOwner.allocateSlot -> SlotPool.allocateSlot->SlotPool.allocateMultiTaskSlot
                        -> execution.deploy()->execution.createDeploymentDescriptor->slot.getTaskManagerGateway->RpcTaskManagerGateway.submitTask->taskExecutorGateway.submitTask

SlotPool
requestNewAllocatedSlot->requestSlotFromResourceManager->ResourceManager.requestSlot-SlotManager.registerSlotRequest

SlotManager
registerSlotRequest->internalRequestSlot->allocateResource->ResourceActionsImpl.allocateResource->YarnResourceManager.startNewWorker->YarnResourceManager.requestYarnContainer
```

(TM)
YarnTaskExexcutorRunner

```
YarnTaskExecutorRunner
main->TaskManagerRunner.run->TaskExecutor.start

TaskExecutor
submitTask->tdd.getSerializedTaskInformation->Task.run-》
loadAndInstantiateInvokable(userCodeClassLoader, nameOfInvokableClass, env)->
StreamTask.invoke->StreamInputProcessor.processInput(operator, lock)
```

用户代码处理逻辑

Apache Flink

```
public boolean processInput(OneInputStreamOperator<IN, ?> streamOperator, final Object lock) throws Exception {
    // ...

        while (true) {
            if (currentRecordDeserializer != null) {
            // ...

                if (result.isFullRecord()) {
                    StreamElement recordOrMark = deserializationDelegate.getInstance();

                // 处理watermark, 则框架处理
                    if (recordOrMark.isWatermark()) {
                        // watermark处理逻辑
                        // ...
                         continue;
                    } else if(recordOrMark.isLatencyMarker()) {
                         // 处理latency mark, 也是由框架处理
                         synchronized (lock) {
                             streamOperator.processLatencyMarker(recordOrMark.asLatencyMarker());
                         }
                         continue;
                    } else {
                         // ***** 这里是真正的用户逻辑代码 *****
                         StreamRecord<IN> record = recordOrMark.asRecord();
                         synchronized (lock) {
                             numRecordsIn.inc();
                             streamOperator.setKeyContextElement1(record);
                             streamOperator.processElement(record);
                         }
                         return true;
                    }
                }
            }

        // 其他处理逻辑
        // ...
        }
}
```

```
public void processElement(StreamRecord<IN> element) throws Exception {
    collector.setTimestamp(element);
    userFunction.flatMap(element.getValue(), collector);
}
```

03

QA

Apache Flink

THANKS

【2群】Apache Flink C...

扫一扫群二维码，立刻加入该群。