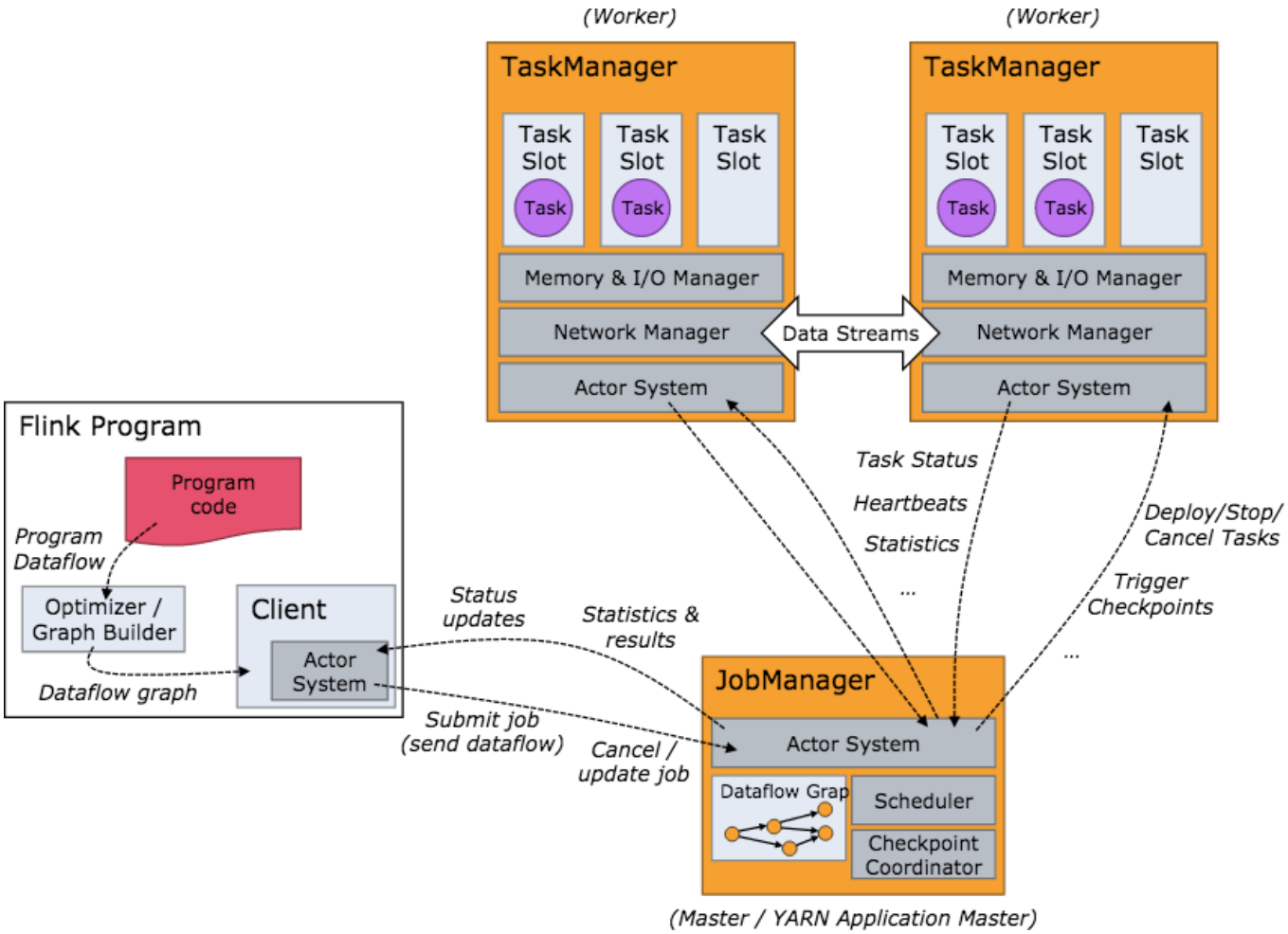
Flink 原理与实现：架构和拓扑概览

摘要： ## 架构 要了解一个系统，一般都是从架构开始。我们关心的问题是：系统部署成功后各个节点都启动了哪些服务，各个服务之间又是怎么交互和协调的。下方是 Flink 集群启动后架构图。 ![](http://img3.tbcdn.cn/5476e8b07b923/TB1ObBnJFXXXXXt

架构

要了解一个系统，一般都是从架构开始。我们关心的问题是：系统部署成功后各个节点都启动了哪些服务，各个服务之间又是怎么交互和协调的。下方是 Flink 集群启动后架构图。



当 Flink 集群启动后，首先会启动一个 JobManger 和一个或多个的 TaskManager。由 Client 提交任务给 JobManager，JobManager 再调度任务到各个 TaskManager 去执行，然后 TaskManager

将心跳和统计信息汇报给 JobManager。TaskManager 之间以流的形式进行数据的传输。上述三者均为独立的 JVM 进程。

- **Client** 为提交 Job 的客户端，可以是运行在任何机器上（与 JobManager 环境连通即可）。提交 Job 后，Client 可以结束进程（Streaming的任务），也可以不结束并等待结果返回。
- **JobManager** 主要负责调度 Job 并协调 Task 做 checkpoint，职责上很像 Storm 的 Nimbus。从 Client 处接收到 Job 和 JAR 包等资源后，会生成优化后的执行计划，并以 Task 的单元调度到各个 TaskManager 去执行。
- **TaskManager** 在启动的时候就设置好了槽位数（Slot），每个 slot 能启动一个 Task，Task 为线程。从 JobManager 处接收需要部署的 Task，部署启动后，与自己的上游建立 Netty 连接，接收数据并处理。

可以看到 Flink 的任务调度是多线程模型，并且不同Job/Task混合在一个 TaskManager 进程中。虽然这种方式可以有效提高 CPU 利用率，但是个人不太喜欢这种设计，因为不仅缺乏资源隔离机制，同时也不方便调试。类似 Storm 的进程模型，一个JVM 中只跑该 Job 的 Tasks 实际应用中更为合理。

Job 例子

> 本文所示例子为 flink-1.0.x 版本

我们使用 Flink 自带的 examples 包中的 `SocketTextStreamWordCount`，这是一个从 socket 流中统计单词出现次数的例子。

- 首先，使用 **netcat** 启动本地服务器：

```
$ nc -l 9000
```

- 然后提交 Flink 程序

```
$ bin/flink run examples/streaming/SocketTextStreamWordCount.jar \
  --hostname 10.218.130.9 \
  --port 9000
```

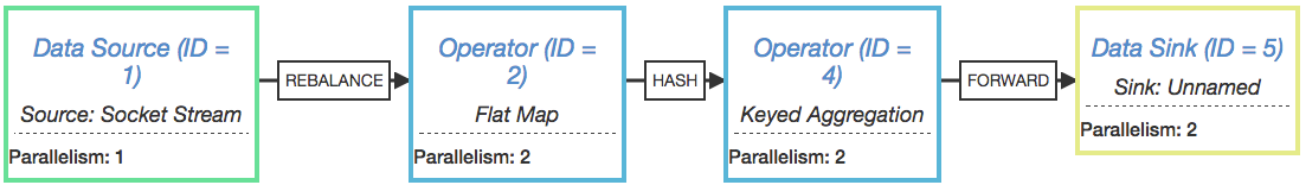
在netcat端输入单词并监控 taskmanager 的输出可以看到单词统计的结果。

`SocketTextStreamWordCount` 的具体代码如下：

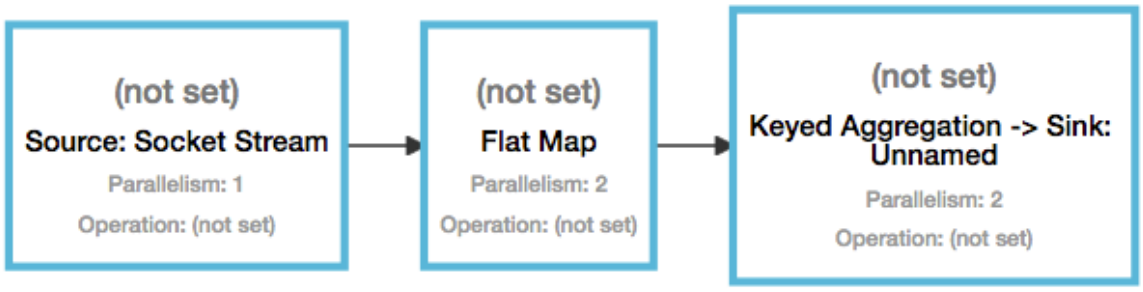
```
public static void main(String[] args) throws Exception {
    // 检查输入
    final ParameterTool params = ParameterTool.fromArgs(args);
    ...
```

```
    // set up the execution environment
    final StreamExecutionEnvironment env = StreamExecutionEnvironment.getExecutio

    // get input data
    DataStream<String> text =
            env.socketTextStream(params.get("hostname"), params.getInt("port"), '

    DataStream<Tuple2<String, Integer>> counts =
            // split up the lines in pairs (2-tuples) containing: (word,1)
            text.flatMap(new Tokenizer())
                    // group by the tuple field "0" and sum up tuple field "1"
                    .keyBy(0)
                    .sum(1);
    counts.print();

    // execute program
    env.execute("WordCount from SocketTextStream Example");
}
```

我们将最后一行代码 `env.execute` 替换成 `System.out.println(env.getExecutionPlan());` 并在本地运行该代码（并发度设为2），可以得到该拓扑的逻辑执行计划图的 JSON 串，将该 JSON 串粘贴到 http://flink.apache.org/visualizer/ 中，能可视化该执行图。



但这并不是最终在 Flink 中运行的执行图，只是一个表示拓扑节点关系的计划图，在 Flink 中对应了 SteramGraph。另外，提交拓扑后（并发度设为2）还能在 UI 中看到另一张执行计划图，如下所示，该图对应了 Flink 中的 JobGraph。



Graph

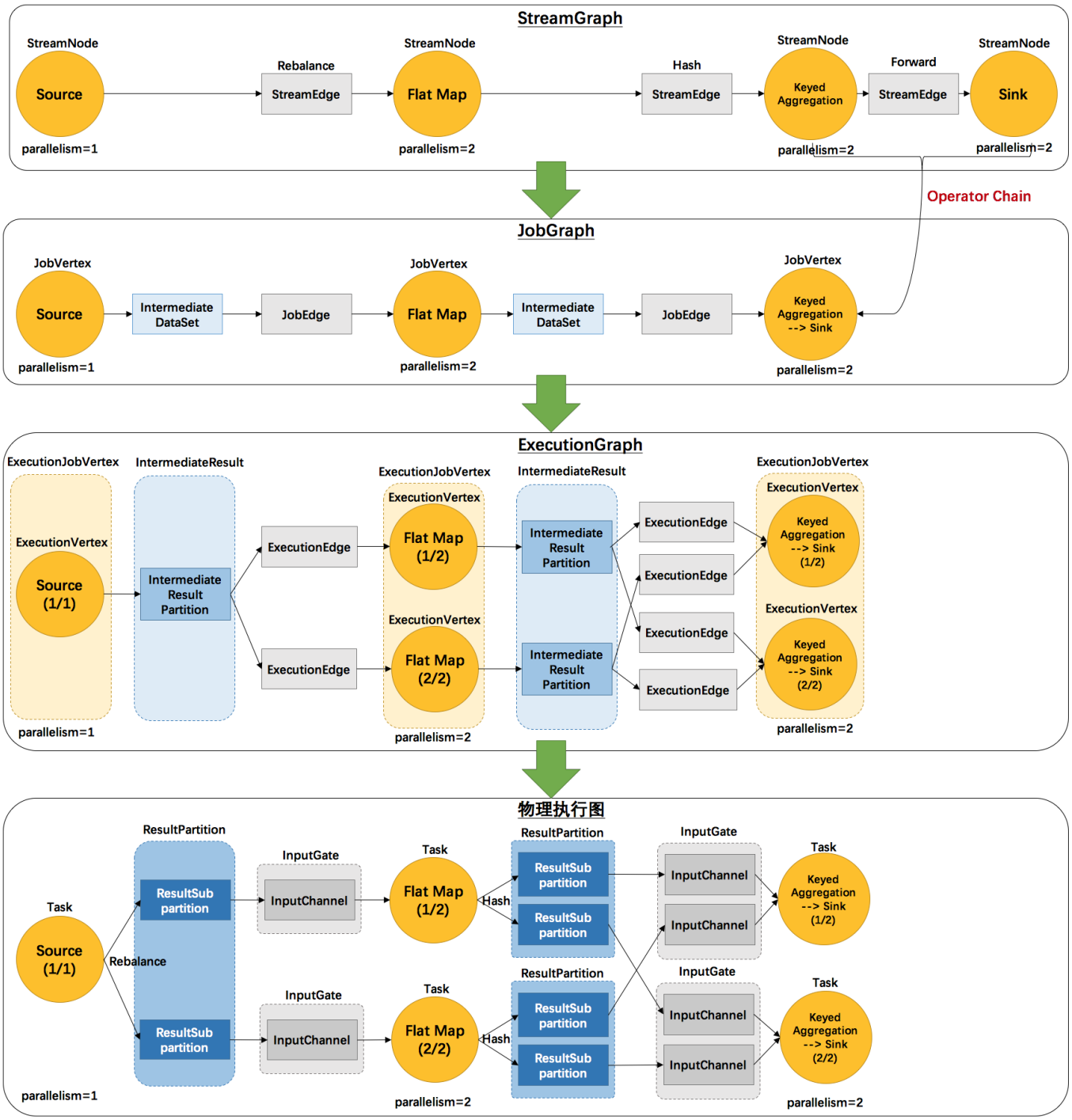
看起来有点乱，怎么有这么多不一样的图。实际上，还有更多的图。Flink 中的执行图可以分成四层：StreamGraph -> JobGraph -> ExecutionGraph -> 物理执行图。

- **StreamGraph**：是根据用户通过 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，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。
- **ExecutionGraph**：JobManager 根据 JobGraph 生成的分布式执行图，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。

例如上文中的2个并发度（Source为1个并发度）的 `SocketTextStreamWordCount` 四层执行图的演变过程如下图所示（点击查看大图）：



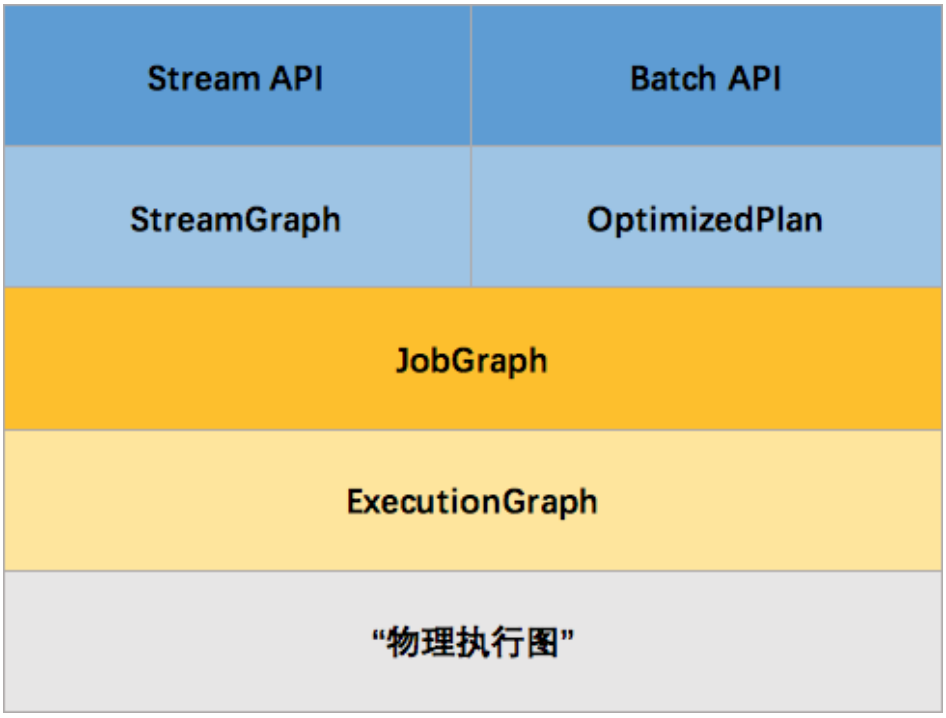
这里对一些名词进行简单的解释。

- **StreamGraph**：根据用户通过 Stream API 编写的代码生成的最初的图。
    - StreamNode：用来代表 operator 的类，并具有所有相关的属性，如并发度、入边和出边等。
    - StreamEdge：表示连接两个StreamNode的边。
- **JobGraph**：StreamGraph经过优化后生成了 JobGraph，提交给 JobManager 的数据结构。
    - JobVertex：经过优化后符合条件的多个StreamNode可能会chain在一起生成一个JobVertex，即一个JobVertex包含一个或多个operator，JobVertex的输入是JobEdge，输出是IntermediateDataSet。
    - IntermediateDataSet：表示JobVertex的输出，即经过operator处理产生的数据集。producer是JobVertex，consumer是JobEdge。
    - JobEdge：代表了job graph中的一条数据传输通道。source 是 IntermediateDataSet，target 是 JobVertex。即数据通过JobEdge由IntermediateDataSet传递给目标JobVertex。
- **ExecutionGraph**：JobManager 根据 JobGraph 生成的分布式执行图，是调度层最核心的数据结构。
    - ExecutionJobVertex：和JobGraph中的JobVertex一一对应。每一个ExecutionJobVertex都有和并发度一样多的 ExecutionVertex。
    - ExecutionVertex：表示ExecutionJobVertex的其中一个并发子任务，输入是ExecutionEdge，输出是IntermediateResultPartition。
    - IntermediateResult：和JobGraph中的IntermediateDataSet一一对应。每一个IntermediateResult有与下游ExecutionJobVertex相同并发数的IntermediateResultPartition。
    - IntermediateResultPartition：表示ExecutionVertex的一个输出分区，producer是ExecutionVertex，consumer是若干个ExecutionEdge。
    - ExecutionEdge：表示ExecutionVertex的输入，source是IntermediateResultPartition，target是ExecutionVertex。source和target都只能是一个。
    - Execution：是执行一个 ExecutionVertex 的一次尝试。当发生故障或者数据需要重算的情况下 ExecutionVertex 可能会有多个 ExecutionAttemptID。一个 Execution 通过 ExecutionAttemptID 来唯一标识。JM和TM之间关于 task 的部署和 task status 的更新都是通过 ExecutionAttemptID 来确定消息接受者。
- **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。
    - Task：Execution被调度后在分配的 TaskManager 中启动对应的 Task。Task 包裹了具

有用户执行逻辑的 operator。

- ResultPartition：代表由一个Task的生成的数据，和ExecutionGraph中的IntermediateResultPartition一一对应。
- ResultSubpartition：是ResultPartition的一个子分区。每个ResultPartition包含多个ResultSubpartition，其数目要由下游消费 Task 数和 DistributionPattern 来决定。
- InputGate：代表Task的输入封装，和JobGraph中JobEdge一一对应。每个InputGate消费了一个或多个的ResultPartition。
- InputChannel：每个InputGate会包含一个以上的InputChannel，和ExecutionGraph中的ExecutionEdge一一对应，也和ResultSubpartition一对一地相连，即一个InputChannel接收一个ResultSubpartition的输出。

那么 Flink 为什么要设计这4张图呢，其目的是什么呢？Spark 中也有多张图，数据依赖图以及物理执行的DAG。其目的都是一样的，就是解耦，每张图各司其职，每张图对应了 Job 不同的阶段，更方便做该阶段的事情。我们给出更完整的 Flink Graph 的层次图。



首先我们看到，JobGraph 之上除了 StreamGraph 还有 OptimizedPlan。OptimizedPlan 是由 Batch API 转换而来的。StreamGraph 是由 Stream API 转换而来的。为什么 API 不直接转换成 JobGraph？因为，Batch 和 Stream 的图结构和优化方法有很大的区别，比如 Batch 有很多执行前的预分析用来优化图的执行，而这种优化并不普适于 Stream，所以通过 OptimizedPlan 来做 Batch 的优化会更方便和清晰，也不会影响 Stream。JobGraph 的责任就是统一 Batch 和 Stream 的图，用来描述清楚一个拓扑图的结构，并且做了 chaining 的优化，chaining 是普适于 Batch 和 Stream 的，所以在这一层做掉。ExecutionGraph 的责任是方便调度和各个 tasks 状态的监控和跟踪，所以 ExecutionGraph 是并行化的 JobGraph。而“物理执行图”就是最终分布式在各个机器上运行着的tasks了。所以可以看到，这种解耦方式极大地方便了我们在各个层所做的工作，各个层之间是相互隔离的。

后续的文章，将会详细介绍 Flink 是如何生成这些执行图的。由于我目前关注 Flink 的流处理功能，所以主要有以下内容：

1. 如何生成 StreamGraph
2. 如何生成 JobGraph
3. 如何生成 ExecutionGraph
4. 如何进行调度（如何生成物理执行图）