> 认定的事情就去做吧，不要再犹豫了

Flink运行时架构

Flink运行时的组件

> Flink运行时架构主要包括四个不同的组件，它们会在运行流处理应用程序时协同工作：作业管理器（JobManager）、资源管理器（ResourceManager）、任务管理器（TaskManager），以及分发器（Dispatcher）。因为Flink是用Java和Scala实现的，所以所有组件都会**运行在Java虚拟机**上

作业管理器（JobManager）

> 控制一个应用程序执行的主进程，也就是说，**每个应用程序都会被一个不同的JobManager所控制执行。**JobManager会先接收到要执行的应用程序，这个应用程序会包括：作业图（JobGraph）、逻辑数据流图（logical dataflow graph）和打包了所有的类、库和其它资源的JAR包。JobManager会把JobGraph转换成一个物理层面的数据流图，这个图被叫做**“执行图”（ExecutionGraph）**，包含了所有可以并发执行的任务。JobManager会向资源管理器（ResourceManager）请求执行任务必要的资源，也就是任务管理器（TaskManager）上的**插槽（slot）**。一旦它获取到了足够的资源，就会将执行图分发到**真正运行它们的TaskManager上。**而在运行过程中，JobManager会负责所有需要**中央协调**的操作，比如说检查点（checkpoints）的协调。

资源管理器（ResourceManager）

> 主要负责管理任务管理器（TaskManager）的插槽（slot），**TaskManger插槽是Flink中定义的处理资源单元。**Flink为不同的环境和资源管理工具提供了不同资源管理器，比如YARN、Mesos、K8s，以及standalone部署。当JobManager申请插槽资源时，ResourceManager会将**有空闲插槽的TaskManager分配给JobManager。**如果ResourceManager**没有足够的插槽**来满足JobManager的请求，它还可以向资源提供平台**发起会话**，以提供启动TaskManager进程的容器。另外，ResourceManager还负责终止空闲的TaskManager，释放计算资源。

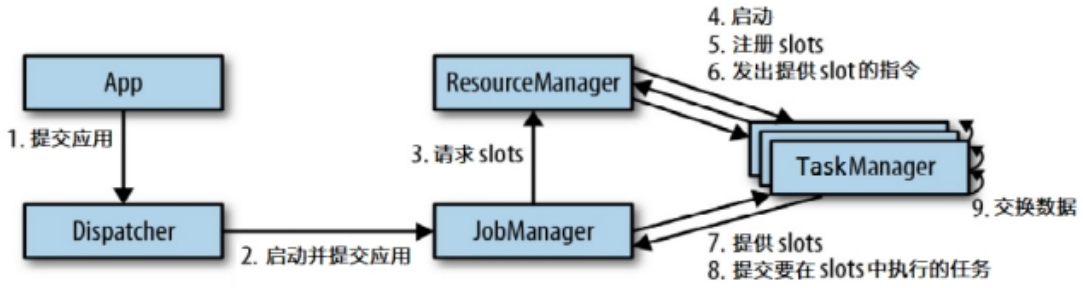
任务管理器（TaskManager）

> Flink中的工作进程。通常在Flink中会有多个TaskManager运行，每一个**TaskManager都包含了一定数量的插槽**（slots）。插槽的数量限制了TaskManager能够执行的任务数量。启动之后，TaskManager会向资源管理器注册它的插槽；收到资源管理器的指令后，TaskManager就会将一个或者多个插槽提供给JobManager**调用。**JobManager就可以向插槽分配任务（tasks）来执行了。在执行过程中，一个TaskManager可以跟其它运行同一应用程序的TaskManager交换数据。

分发器（Dispatcher）

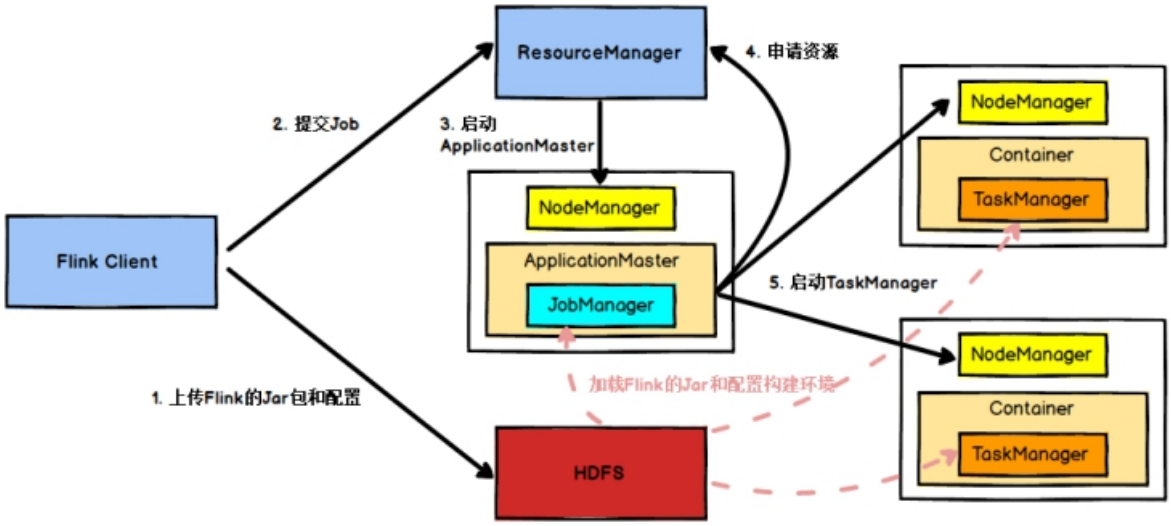
> 可以跨作业运行，它为应用提交提供了**REST接口。**当一个应用被提交执行时，分发器就会启动并将应用移交给一个JobManager。由于是REST接口，所以Dispatcher可以作为集群的一个**HTTP接入点**，这样就能够不受防火墙阻挡。Dispatcher也会**启动一个Web UI**，用来方便地展示和监控作业执行的信息。Dispatcher在架构中可能并不是必需的，这取决于应用提交运行的方式。

任务提交流程



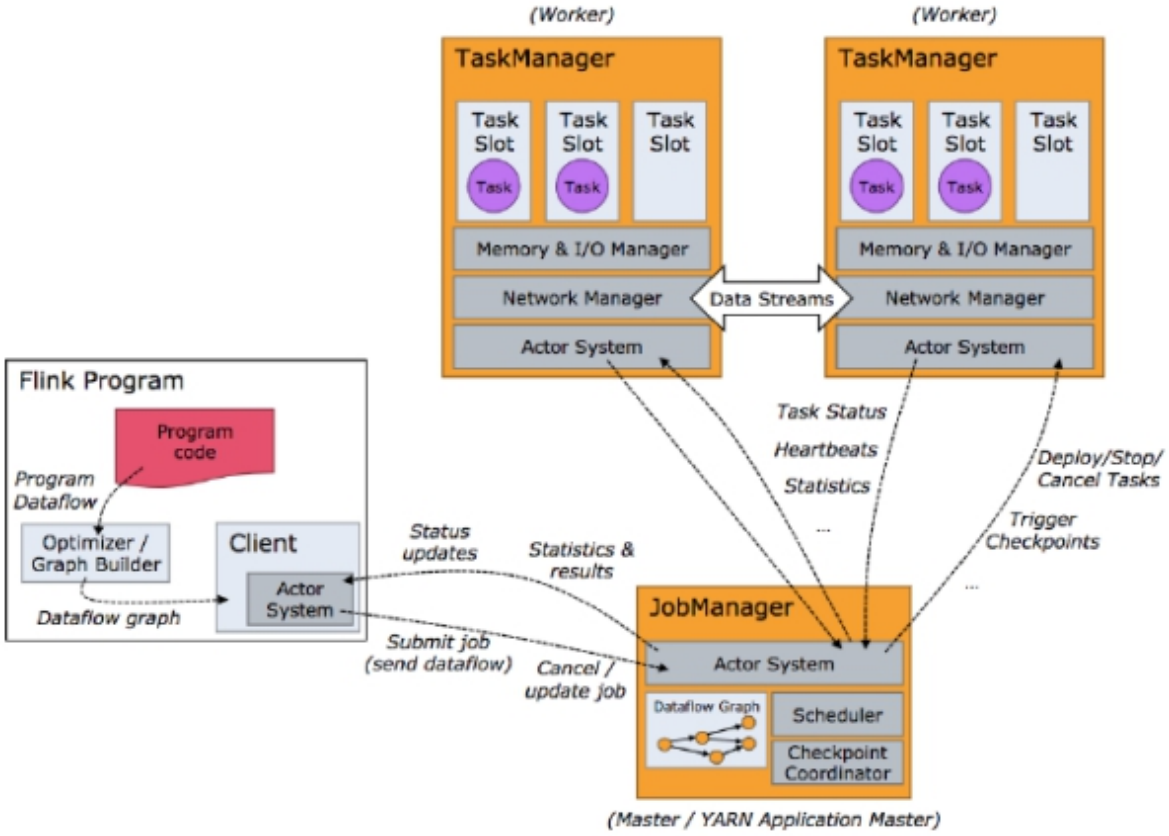
上图是从一个较为高层级的视角，来看应用中各组件的交互协作。如果部署的集群环境不同（例如YARN，Mesos，Kubernetes，standalone等），其中一些步骤可以被省略，或是有些组件会运行在同一个JVM进程中。

具体地，如果我们将Flink集群部署到YARN上，那么就会有如下的提交流程：



> Flink任务提交后，Client向HDFS上传Flink的Jar包和配置，之后向Yarn ResourceManager提交任务，ResourceManager分配Container资源并通知对应的NodeManager启动ApplicationMaster，ApplicationMaster启动后加载Flink的Jar包和配置构建环境，然后启动JobManager，之后ApplicationMaster向ResourceManager申请资源启动TaskManager，ResourceManager分配Container资源后，由ApplicationMaster通知资源所在节点的NodeManager启动TaskManager，NodeManager加载Flink的Jar包和配置构建环境并启动TaskManager，TaskManager启动后向JobManager发送心跳包，并等待JobManager向其分配任务。【上面这段都是重点】

任务调度原理



客户端不是运行时和程序执行的一部分，但它用于准备并发送dataflow(JobGraph)给Master(JobManager)，然后，客户端断开连接或者维持连接以等待接收计算结果。

> 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。

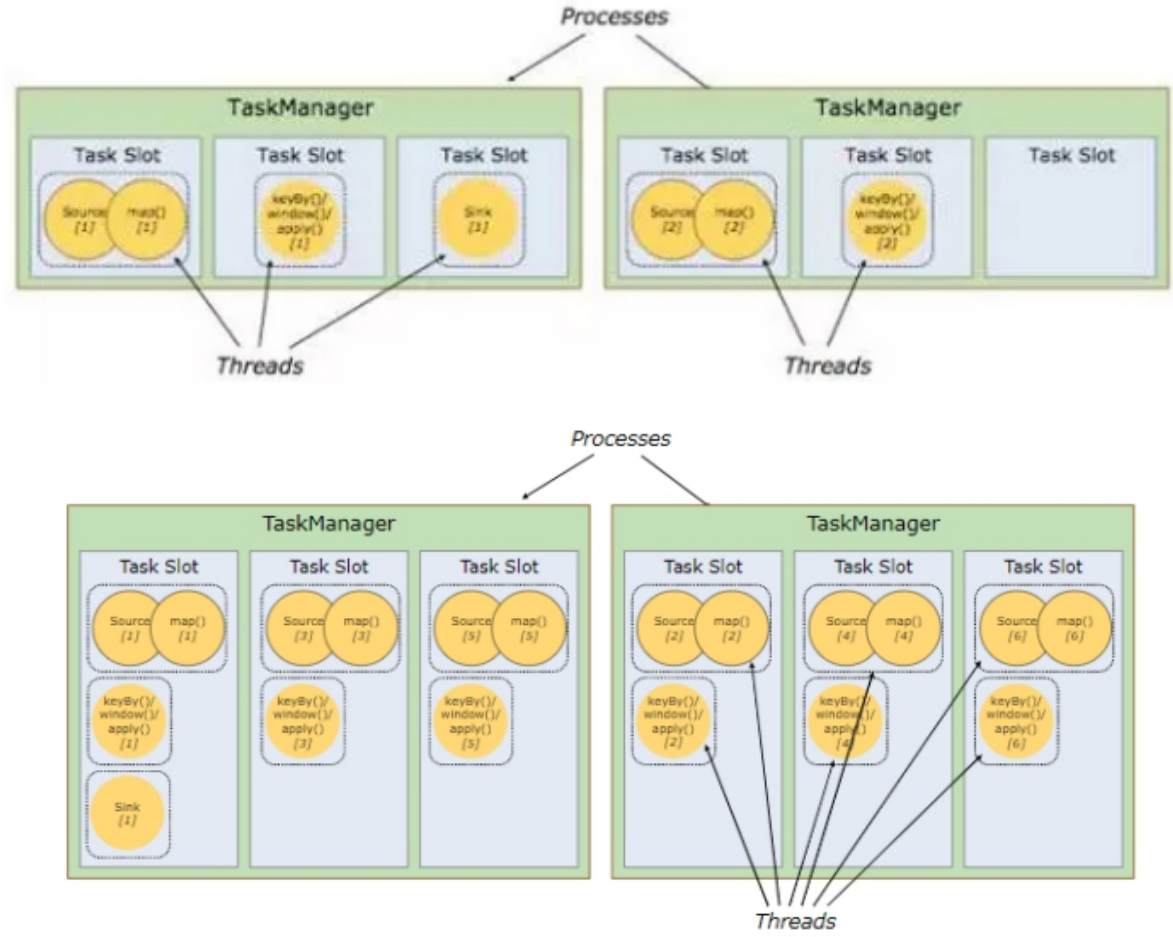
- **Client** 为提交 Job 的客户端，可以是运行在任何机器上（与 JobManager 环境连通即可）。提交 Job 后，Client 可以结束进程（Streaming的任务），也可以不结束并等待结果返回。
- **JobManager***主要负责调度 Job 并协调 Task 做 checkpoint，职责上很像 Storm 的 Nimbus。从 Client 处接收到 Job 和 JAR 包等资源后，会生成优化后的执行计划，并以 Task 的单元调度到各个 TaskManager 去执行。
- **TaskManager**在启动的时候就设置好了槽位数（Slot），每个 slot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

TaskManger与Slots

Flink中每一个worker(TaskManager)都是一个**JVM进程**，它可能会在**独立的线程**上执行一个或多个subtask。为了控制一个worker能接收多少个task，worker通过task slot来进行控制（一个worker至少有一个task slot）。

> 每个task slot表示TaskManager拥有资源的**一个固定大小的子集**。假如一个TaskManager有三个slot，那么它会将其管理的内存分成三份给各个slot。资源slot化意味着一个subtask将不需要跟来自其他job的subtask竞争被管理的内存，取而代之的是它将拥有一定数量的内存储备。需要注意的是，这里不会涉及到CPU的隔离，slot目前仅仅用来隔离task的受管理的内存。

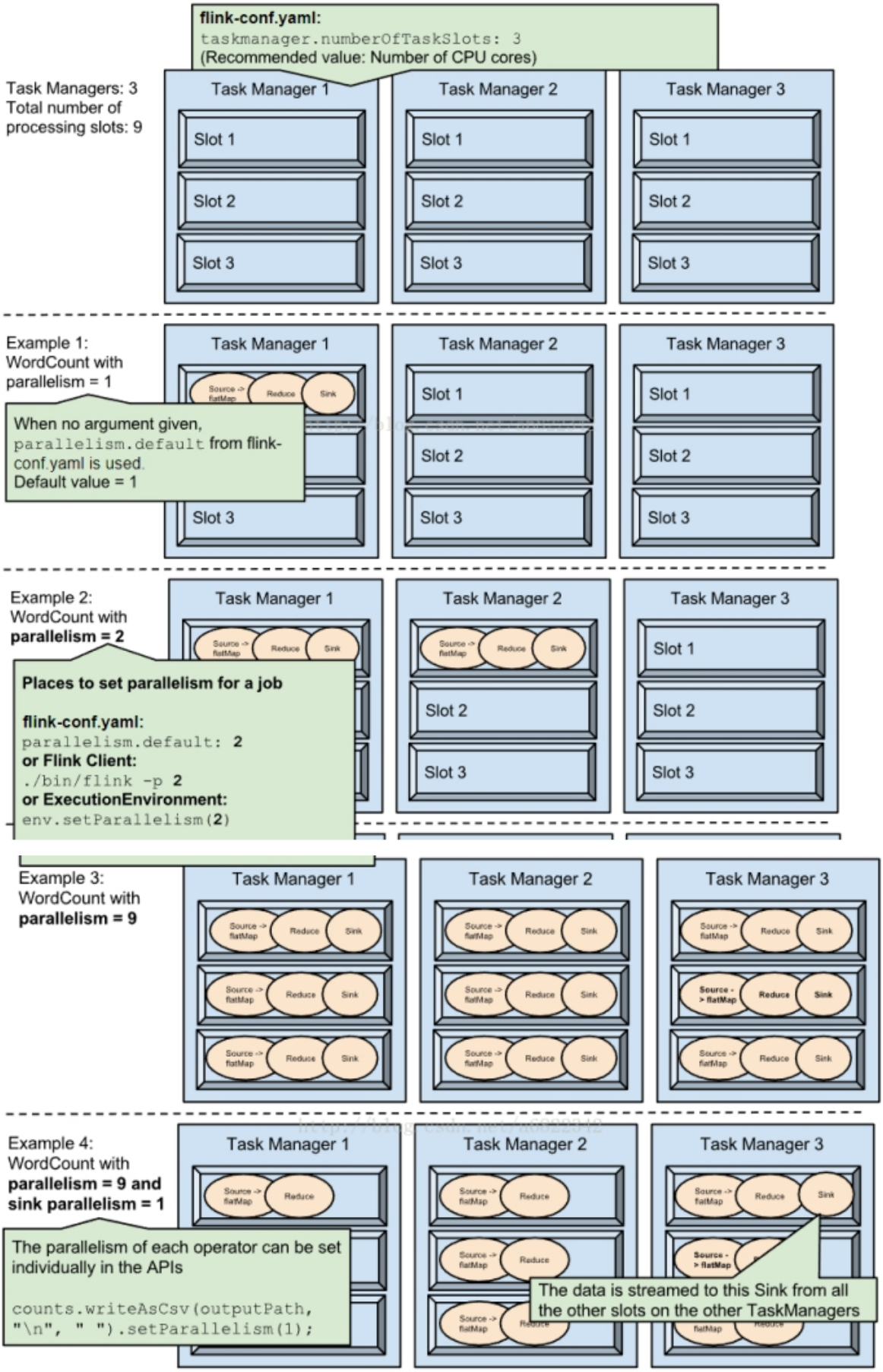
通过调整task slot的数量，允许用户定义subtask之间如何互相隔离。如果一个TaskManager一个slot，那将意味着每个task group运行在独立的JVM中（该JVM可能是通过一个特定的容器启动的），而一个TaskManager多个slot意味着更多的subtask可以共享同一个JVM。而在同一个JVM进程中的task将共享TCP连接（基于多路复用）和心跳消息。它们也可能**共享数据集和数据结构，因此这减少了每个task的负载。**



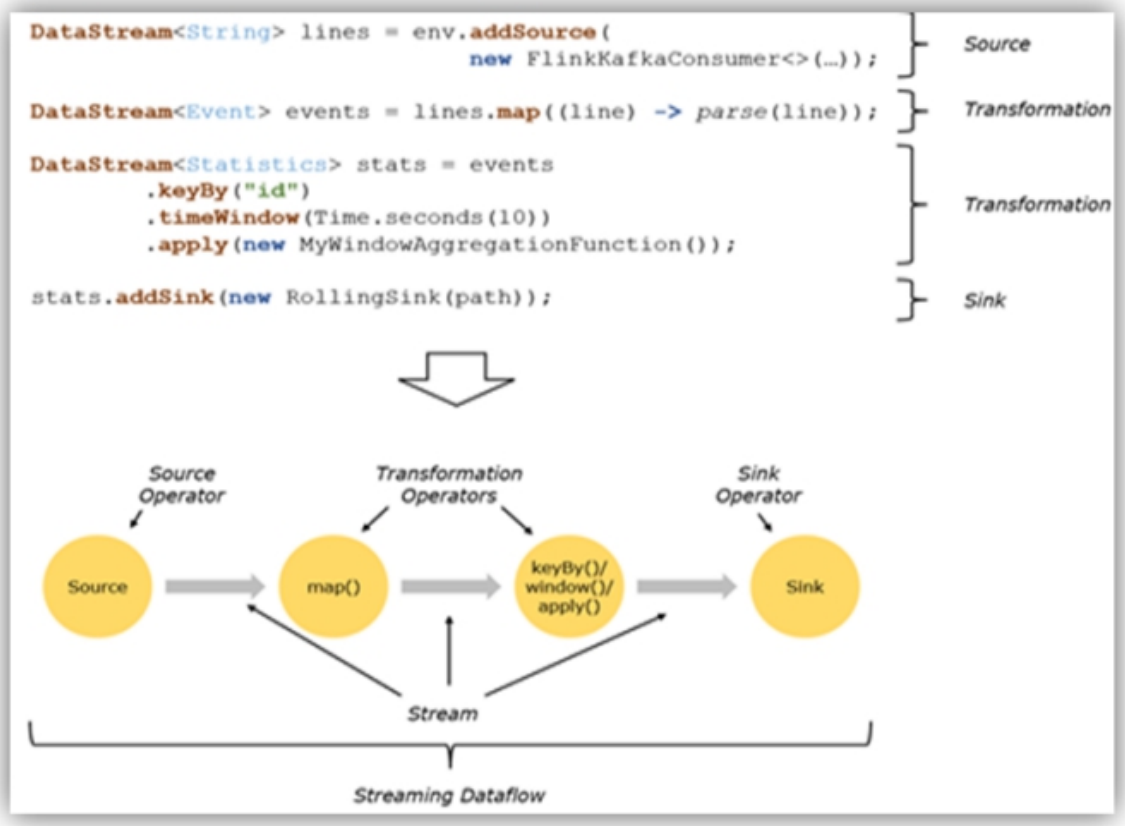
默认情况下，Flink允许子任务共享slot，即使它们是不同任务的子任务（前提是它们来自同一个job）。 这样的结果是，一个slot可以保存作业的整个管道。

> **Task Slot是静态的概念，是指TaskManager具有的并发执行能力**，可以通过参数taskmanager.numberOfTaskSlots进行配置；而**并行度parallelism是动态概念，即TaskManager运行程序时实际使用的并发能力**，可以通过参数parallelism.default进行配置。

也就是说，假设一共有3个TaskManager，每一个TaskManager中的分配3个TaskSlot，也就是每个TaskManager可以接收3个task，一共9个TaskSlot，如果我们设置parallelism.default=1，即运行程序默认的并行度为1，9个TaskSlot只用了1个，有8个空闲，因此，设置合适的并行度才能提高效率。

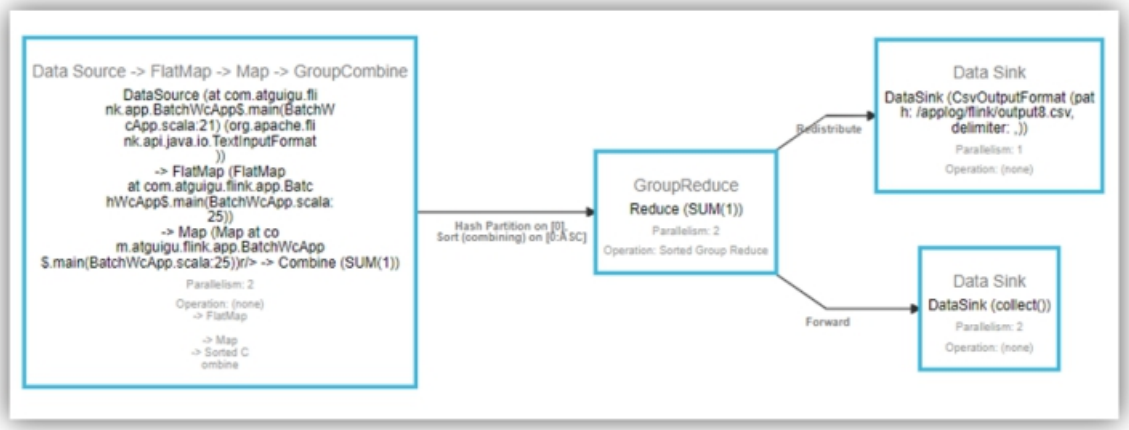


程序与数据流（DataFlow）



所有的Flink程序都是由三部分组成的： **Source 、Transformation和Sink。**

> Source负责读取数据源，Transformation利用各种算子进行处理加工，Sink负责输出。在运行时，Flink上运行的程序会被映射成“逻辑数据流”（dataflows），它包含了这三部分。**每一个dataflow以一个或多个sources开始以一个或多个sinks结束。**dataflow类似于任意的有向无环图（DAG）。在大部分情况下，程序中的转换运算（transformations）跟dataflow中的算子（operator）是一一对应的关系，但有时候，一个transformation可能对应多个operator。



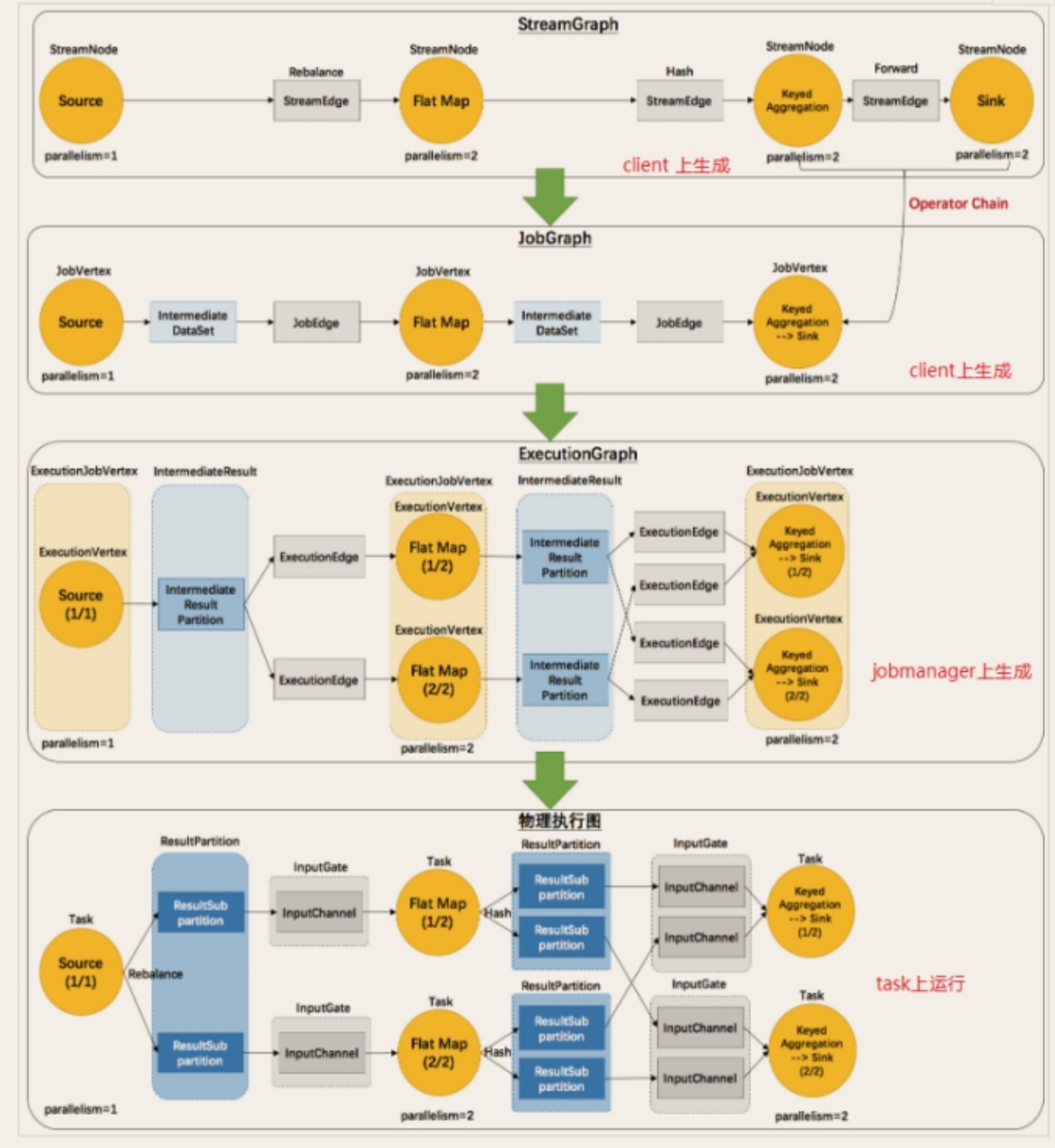
执行图（ExecutionGraph）【重点掌握】

> 由Flink程序直接映射成的数据流图是StreamGraph，也被称为**逻辑流图**，因为它们表示的是计算逻辑的高级视图。为了执行一个流处理程序，Flink需要将逻辑流图转换为物理数据流图（也叫执行图），详细说明程序的执行方式。

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。

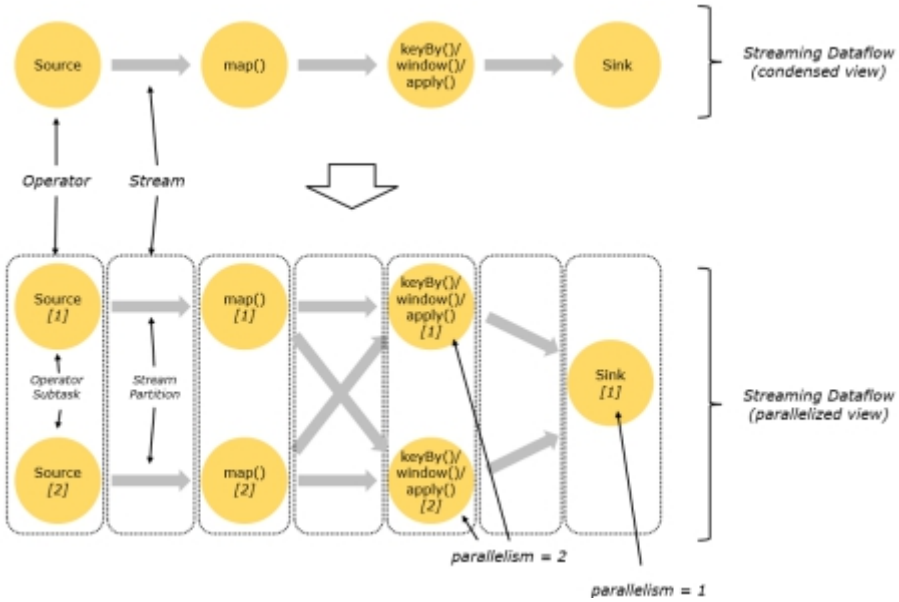


并行度（Parallelism）

Flink程序的执行具有**并行、分布式**的特性。

> 在执行过程中，一个流（stream）包含一个或多个分区（stream partition），而每一个算子（operator）可以包含一个或多个子任务（operator subtask），这些子任务在不同的线程、不同的物理机或不同的容器中彼此互不依赖地执行。

一个特定算子的子任务（subtask）的个数被称之为其并行度（parallelism）。一般情况下，一个流程序的并行度，可以认为就是其所有算子中最大的并行度。一个程序中，不同的算子可能具有不同的并行度。

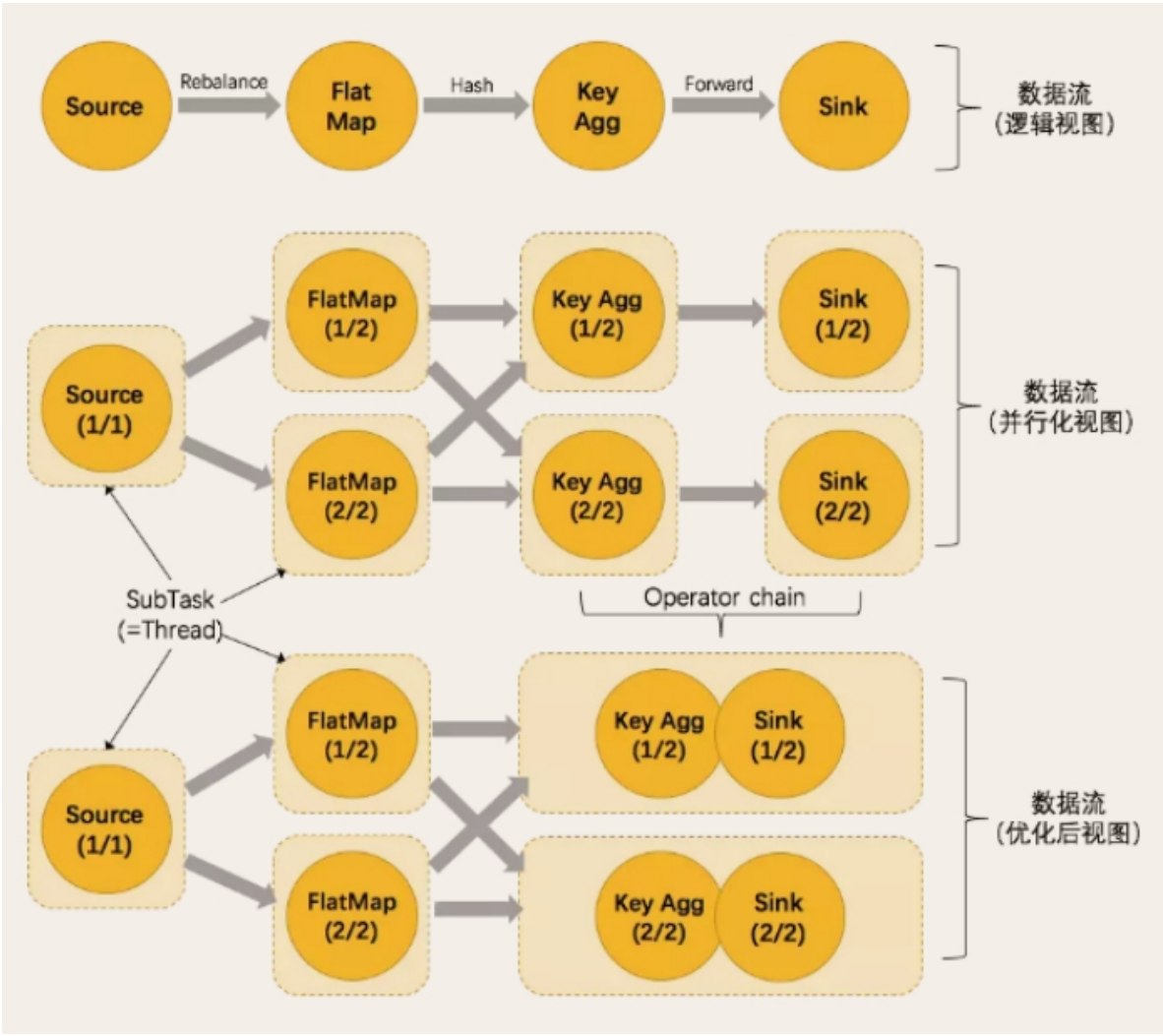


Stream在算子之间传输数据的形式可以是one-to-one(forwarding)的模式也可以是redistributing的模式，具体是哪一种形式，取决于算子的种类。

- **One-to-one**：stream(比如在source和map operator之间)维护着分区以及元素的顺序。那意味着map 算子的子任务看到的元素的个数以及顺序跟source 算子的子任务生产的元素的个数、顺序相同，map、fliter、flatMap等算子都是one-to-one的对应关系。类似于spark中的**窄依赖**
- **Redistributing**：stream(map()跟keyBy/window之间或者keyBy/window跟sink之间)的分区会发生改变。每一个算子的子任务依据所选择的transformation发送数据到不同的目标任务。例如，keyBy() 基于hashCode重分区、broadcast和rebalance会随机重新分区，这些算子都会引起redistribute过程，而redistribute过程就类似于Spark中的shuffle过程。类似于spark中的**宽依赖**

任务链（Operator Chains）【重点掌握】

> **相同并行度的one to one操作**，Flink这样**相连的算子链接在一起形成一个task**，原来的算子成为里面的一部分。将算子链接成task是非常有效的优化：它能减少线程之间的切换和基于缓存区的数据交换，在减少时延的同时提升吞吐量。链接的行为可以在编程API中进行指定。


Source
Rebalance
Flat Map
Hash
Key Agg
Forward
Sink
数据流
（逻辑视图）
Source (1/1)
FlatMap (1/2)
FlatMap (2/2)
Key Agg (1/2)
Key Agg (2/2)
Sink (1/2)
Sink (2/2)
数据流
（并行化视图）
SubTask
(=Thread)
Operator chain
Source (1/1)
FlatMap (1/2)
FlatMap (2/2)
Key Agg (1/2)
Sink (1/2)
Key Agg (2/2)
Sink (2/2)
数据流
（优化后视图）