**[作业管理器（JobManager）](https://ashiamd.github.io/docsify-notes/" \l "/study/BigData/Flink/%E5%B0%9A%E7%A1%85%E8%B0%B7Flink%E5%85%A5%E9%97%A8%E5%88%B0%E5%AE%9E%E6%88%98-%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B0?id=%e4%bd%9c%e4%b8%9a%e7%ae%a1%e7%90%86%e5%99%a8%ef%bc%88jobmanager%ef%bc%89)**

​ 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）
* 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**。

​ 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）**](https://ashiamd.github.io/docsify-notes/#/study/BigData/Flink/%E5%B0%9A%E7%A1%85%E8%B0%B7Flink%E5%85%A5%E9%97%A8%E5%88%B0%E5%AE%9E%E6%88%98-%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B0?id=%e8%b5%84%e6%ba%90%e7%ae%a1%e7%90%86%e5%99%a8%ef%bc%88resourcemanager%ef%bc%89)

​ 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）**](https://ashiamd.github.io/docsify-notes/#/study/BigData/Flink/%E5%B0%9A%E7%A1%85%E8%B0%B7Flink%E5%85%A5%E9%97%A8%E5%88%B0%E5%AE%9E%E6%88%98-%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B0?id=%e4%bb%bb%e5%8a%a1%e7%ae%a1%e7%90%86%e5%99%a8%ef%bc%88taskmanager%ef%bc%89)

​ 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）**](https://ashiamd.github.io/docsify-notes/#/study/BigData/Flink/%E5%B0%9A%E7%A1%85%E8%B0%B7Flink%E5%85%A5%E9%97%A8%E5%88%B0%E5%AE%9E%E6%88%98-%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B0?id=%e5%88%86%e5%8f%91%e5%99%a8%ef%bc%88dispatcher%ef%bc%89)

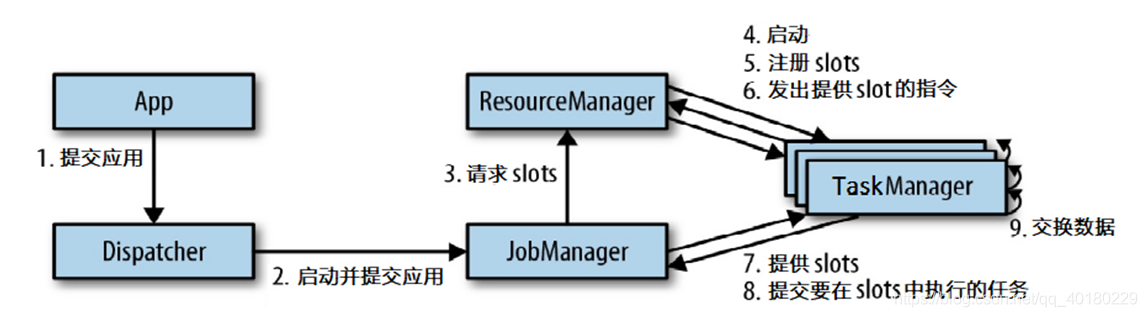
​ 可以跨作业运行，它为应用提交提供了REST接口。

​ 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。*

[**4.2 任务提交流程**](https://ashiamd.github.io/docsify-notes/#/study/BigData/Flink/%E5%B0%9A%E7%A1%85%E8%B0%B7Flink%E5%85%A5%E9%97%A8%E5%88%B0%E5%AE%9E%E6%88%98-%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B0?id=_42-%e4%bb%bb%e5%8a%a1%e6%8f%90%e4%ba%a4%e6%b5%81%e7%a8%8b)

​ 我们来看看当一个应用提交执行时，Flink的各个组件是如何交互协作的：

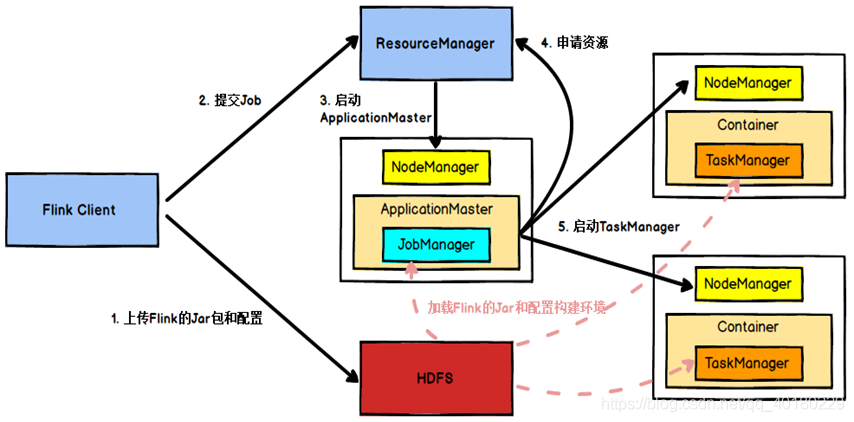


​ *ps：上图中7.指TaskManager为JobManager提供slots，8.表示JobManager提交要在slots中执行的任务给TaskManager。*

​ 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。

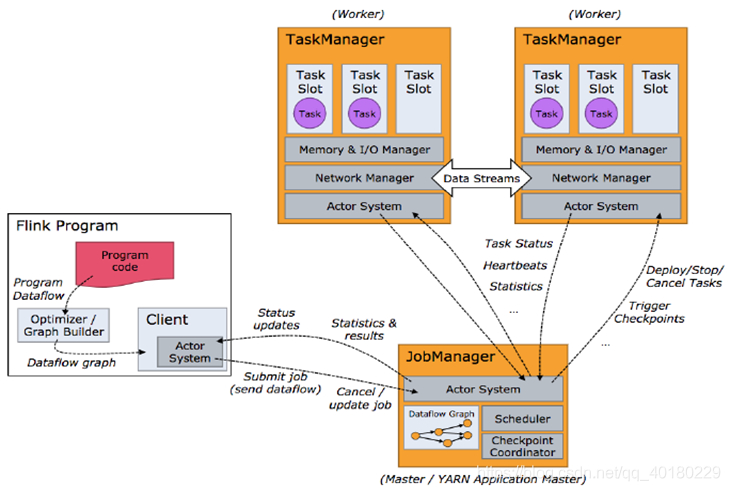
​ 如果部署的集群环境不同（例如YARN，Mesos，Kubernetes，standalone等），其中一些步骤可以被省略，或是有些组件会运行在同一个JVM进程中。

​ 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：



1. Flink任务提交后，Client向HDFS上传Flink的Jar包和配置
2. 之后客户端向Yarn ResourceManager提交任务，ResourceManager分配Container资源并通知对应的NodeManager启动ApplicationMaster
3. ApplicationMaster启动后加载Flink的Jar包和配置构建环境，去启动JobManager，之后**JobManager向Flink自身的RM进行申请资源，自身的RM向Yarn 的ResourceManager申请资源(因为是yarn模式，所有资源归yarn RM管理)启动TaskManager**
4. Yarn ResourceManager分配Container资源后，由ApplicationMaster通知资源所在节点的NodeManager启动TaskManager
5. 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。

[**4.3 任务调度原理**](https://ashiamd.github.io/docsify-notes/#/study/BigData/Flink/%E5%B0%9A%E7%A1%85%E8%B0%B7Flink%E5%85%A5%E9%97%A8%E5%88%B0%E5%AE%9E%E6%88%98-%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B0?id=_43-%e4%bb%bb%e5%8a%a1%e8%b0%83%e5%ba%a6%e5%8e%9f%e7%90%86)



1. 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。而Job Manager会产生一个执行图(Dataflow Graph)
2. 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。
3. 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。
4. 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。
5. 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。

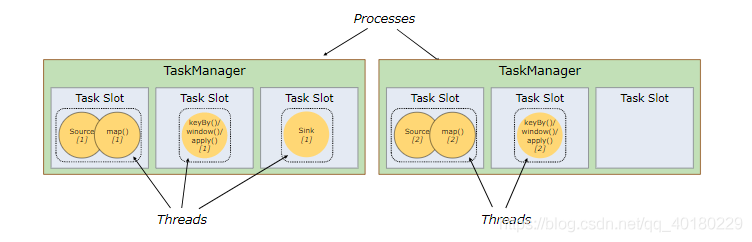
*注：如果一个Slot中启动多个线程，那么这几个线程类似CPU调度一样共用同一个slot*

[**4.3.1 TaskManger与Slots**](https://ashiamd.github.io/docsify-notes/#/study/BigData/Flink/%E5%B0%9A%E7%A1%85%E8%B0%B7Flink%E5%85%A5%E9%97%A8%E5%88%B0%E5%AE%9E%E6%88%98-%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B0?id=_431-taskmanger%e4%b8%8eslots)

要点：

* 考虑到Slot分组，所以实际运行Job时所需的Slot总数 = 每个Slot组中的最大并行度。

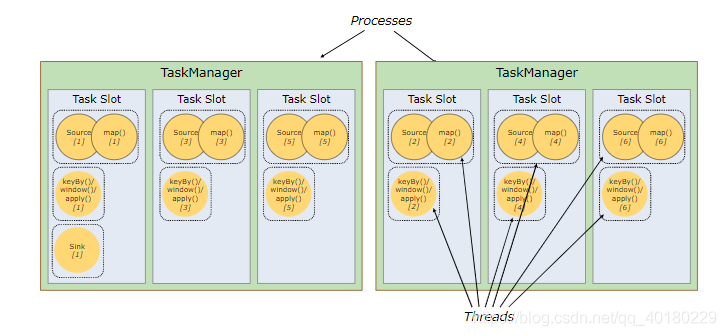
eg（1，1，2，1）,其中第一个归为组“red”、第二个归组“blue”、第三个和第四归组“green”，那么运行所需的slot即max（1）+max（1）+max（2，1） = 1+1+2 = 4



* 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）。

\*\*上图这个每个子任务各自占用一个slot，可以在代码中通过算子的.slotSharingGroup("组名")指定算子所在的Slot组名，默认每一个算子的SlotGroup和上一个算子相同，而默认的SlotGroup就是"default"\*\*。

**同一个SlotGroup的算子能共享同一个slot，不同组则必须另外分配独立的Slot。**



* 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（前提需要来自同一个Job）。这样结果是，**一个slot可以保存作业的整个管道pipeline**。
  + **不同任务共享同一个Slot的前提：这几个任务前后顺序不同，如上图中Source和keyBy是两个不同步骤顺序的任务，所以可以在同一个Slot执行**。
  + 一个slot可以保存作业的整个管道的好处：
    - 如果有某个slot执行完了整个任务流程，那么其他任务就可以不用继续了，这样也省去了跨slot、跨TaskManager的通信损耗（降低了并行度）
    - 同时slot能够保存整个管道，使得整个任务执行健壮性更高，因为某些slot执行出异常也能有其他slot补上。
    - 有些slot分配到的子任务非CPU密集型，有些则CPU密集型，如果每个slot只完成自己的子任务，将出现某些slot太闲，某些slot过忙的现象。
  + *假设拆分的多个Source子任务放到同一个Slot，那么任务不能并行执行了=>因为多个相同步骤的子任务需要抢占的具体资源相同，比如抢占某个锁，这样就不能并行。*
* 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。*

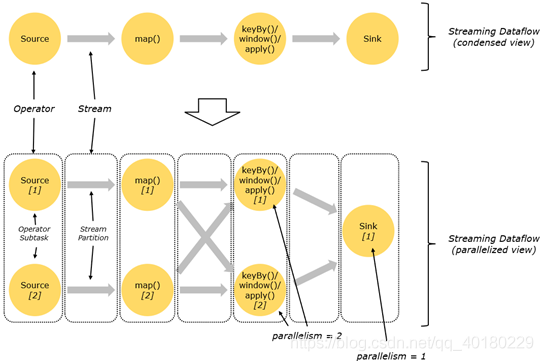
​ 每个task slot表示TaskManager拥有资源的一个固定大小的子集。假如一个TaskManager有三个slot，那么它会将其管理的内存分成三份给各个slot。资源slot化意味着一个subtask将不需要跟来自其他job的subtask竞争被管理的内存，取而代之的是它将拥有一定数量的内存储备。

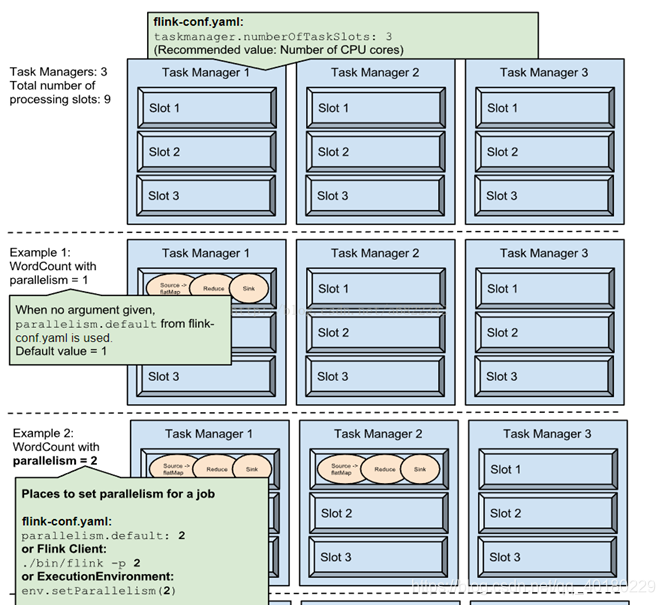
​ **需要注意的是，这里不会涉及到CPU的隔离，slot目前仅仅用来隔离task的受管理的内存**。

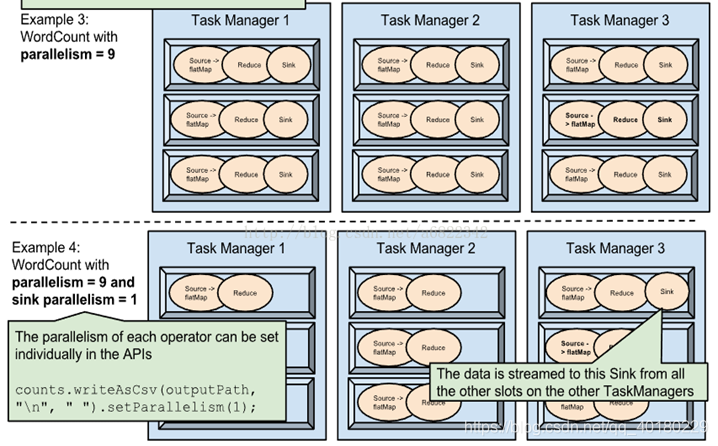
​ 通过调整task slot的数量，允许用户定义subtask之间如何互相隔离。如果一个TaskManager一个slot，那将意味着每个task group运行在独立的JVM中（该JVM可能是通过一个特定的容器启动的），而一个TaskManager多个slot意味着更多的subtask可以共享同一个JVM。而在同一个JVM进程中的task将共享TCP连接（基于多路复用）和心跳消息。它们也可能共享数据集和数据结构，因此这减少了每个task的负载。

[**4.3.2 Slot和并行度**](https://ashiamd.github.io/docsify-notes/#/study/BigData/Flink/%E5%B0%9A%E7%A1%85%E8%B0%B7Flink%E5%85%A5%E9%97%A8%E5%88%B0%E5%AE%9E%E6%88%98-%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B0?id=_432-slot%e5%92%8c%e5%b9%b6%e8%a1%8c%e5%ba%a6)

1. **一个特定算子的 子任务（subtask）的个数被称之为其并行度（parallelism）**，我们可以对单独的每个算子进行设置并行度，也可以直接用env设置全局的并行度，更可以在页面中去指定并行度。
2. 最后，由于并行度是实际Task Manager处理task 的能力，而一般情况下，**一个 stream 的并行度，可以认为就是其所有算子中最大的并行度**，则可以得出**在设置Slot时，在所有设置中的最大设置的并行度大小则就是所需要设置的Slot的数量。**（如果Slot分组，则需要为每组Slot并行度最大值的和）



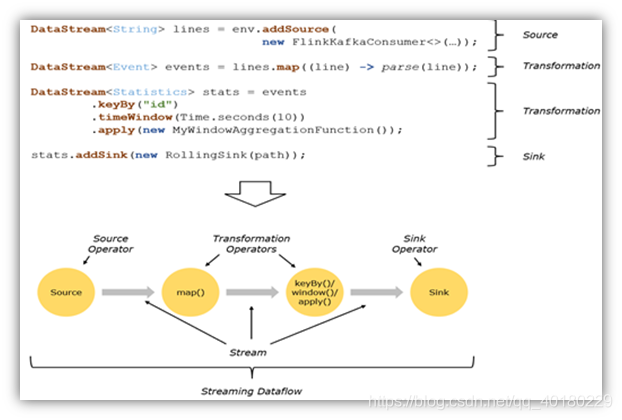




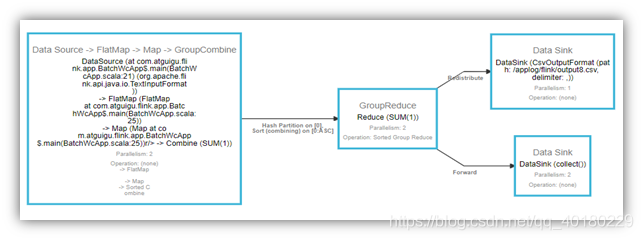
​ 假设一共有3个TaskManager，每一个TaskManager中的分配3个TaskSlot，也就是每个TaskManager可以接收3个task，一共9个TaskSlot，如果我们设置parallelism.default=1，即运行程序默认的并行度为1，9个TaskSlot只用了1个，有8个空闲，因此，设置合适的并行度才能提高效率。

​ *ps：上图最后一个因为是输出到文件，避免多个Slot（多线程）里的算子都输出到同一个文件互相覆盖等混乱问题，直接设置sink的并行度为1。*

[**4.3.3 程序和数据流（DataFlow）**](https://ashiamd.github.io/docsify-notes/#/study/BigData/Flink/%E5%B0%9A%E7%A1%85%E8%B0%B7Flink%E5%85%A5%E9%97%A8%E5%88%B0%E5%AE%9E%E6%88%98-%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B0?id=_433-%e7%a8%8b%e5%ba%8f%e5%92%8c%e6%95%b0%e6%8d%ae%e6%b5%81%ef%bc%88dataflow%ef%bc%89)



* **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

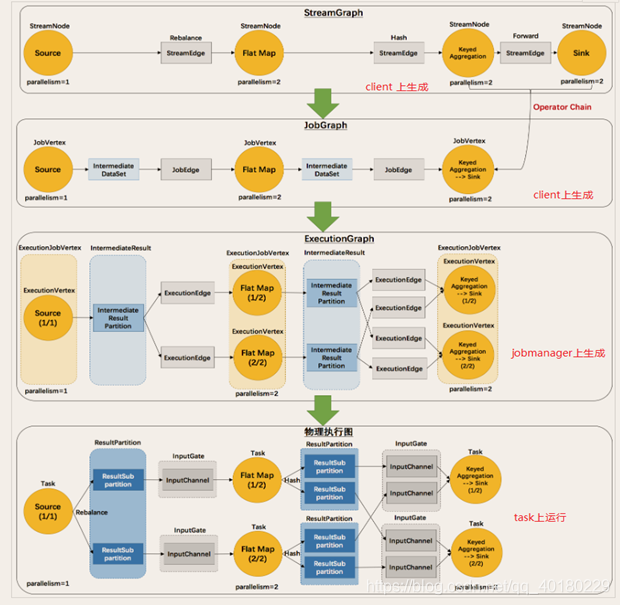


* 在运行时，Flink上运行的程序会被映射成“逻辑数据流”（dataflows），它包含了这三部分
* 每一个dataflow以一个或多个sources开始以一个或多个sinks结束。dataflow类似于任意的有向无环图（DAG）
* 在大部分情况下，程序中的转换运算（transformations）跟dataflow中的算子（operator）是一一对应的关系

[**4.3.4 执行图（ExecutionGraph）**](https://ashiamd.github.io/docsify-notes/#/study/BigData/Flink/%E5%B0%9A%E7%A1%85%E8%B0%B7Flink%E5%85%A5%E9%97%A8%E5%88%B0%E5%AE%9E%E6%88%98-%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B0?id=_434-%e6%89%a7%e8%a1%8c%e5%9b%be%ef%bc%88executiongraph%ef%bc%89)

​ 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。



[**4.3.5 数据传输形式**](https://ashiamd.github.io/docsify-notes/#/study/BigData/Flink/%E5%B0%9A%E7%A1%85%E8%B0%B7Flink%E5%85%A5%E9%97%A8%E5%88%B0%E5%AE%9E%E6%88%98-%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B0?id=_435-%e6%95%b0%e6%8d%ae%e4%bc%a0%e8%be%93%e5%bd%a2%e5%bc%8f)

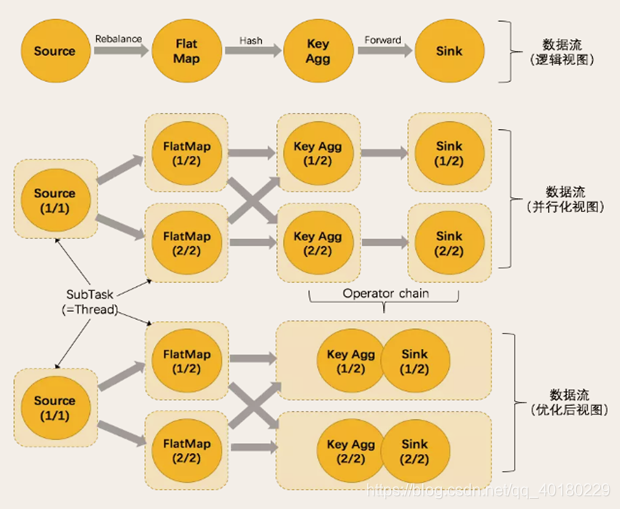
* 一个程序中，不同的算子可能具有不同的并行度
* 算子之间传输数据的形式可以是 one-to-one (forwarding) 的模式也可以是redistributing 的模式，具体是哪一种形式，取决于算子的种类
  + **One-to-one**：stream维护着分区以及元素的顺序（比如source和map之间）。这意味着map 算子的子任务看到的元素的个数以及顺序跟 source 算子的子任务生产的元素的个数、顺序相同。**map、fliter、flatMap等算子都是one-to-one的对应关系**。
  + **Redistributing**：stream的分区会发生改变。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、而 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。

[**4.3.6 任务链（OperatorChains）**](https://ashiamd.github.io/docsify-notes/#/study/BigData/Flink/%E5%B0%9A%E7%A1%85%E8%B0%B7Flink%E5%85%A5%E9%97%A8%E5%88%B0%E5%AE%9E%E6%88%98-%E5%AD%A6%E4%B9%A0%E7%AC%94%E8%AE%B0?id=_436-%e4%bb%bb%e5%8a%a1%e9%93%be%ef%bc%88operatorchains%ef%bc%89)

​ Flink 采用了一种称为任务链的优化技术，可以在特定条件下减少本地通信的开销。为了满足任务链的要求，必须将两个或多个算子设为**相同的并行度**，并通过本地转发（local forward）的方式进行连接

* **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 subtask
  + 并行度相同、并且是 one-to-one 操作，两个条件缺一不可

​ **为什么需要并行度相同，因为若flatMap并行度为1，到了之后的map并行度为2，从flatMap到map的数据涉及到数据由于并行度map为2会往两个slot处理，数据会分散，所产生的元素个数和顺序发生的改变所以有2个单独的task，不能成为任务链**



​ **如果前后任务逻辑上可以是OneToOne，且并行度一致，那么就能合并在一个Slot里**（并行度原本是多少就是多少，两者并行度一致）执行。

* keyBy需要根据Hash值分配给不同slot执行，所以只能Hash，不能OneToOne。
* 逻辑上可OneToOne但是并行度不同，那么就会Rebalance，轮询形式分配给下一个任务的多个slot。
* **代码中如果算子.disableChaining()，能够强制当前算子的子任务不参与任务链的合并，即不和其他Slot资源合并，但是仍然可以保留“Slot共享”的特性**。
* **如果StreamExecutionEnvironment env.disableOperatorChaining()则当前执行环境全局设置算子不参与"任务链的合并"。**
* **如果算子.startNewChain()表示不管前面任务链合并与否，从当前算子往后重新计算任务链的合并。通常用于前面强制不要任务链合并，而当前往后又需要任务链合并的特殊场景。**

*ps：如果算子.shuffle()，能够强制算子之后重分区到不同slot执行下一个算子操作，逻辑上也实现了任务不参与任务链合并=>但是仅为“不参与任务链的合并”，这个明显不是最优解操作*

[**Flink slotSharingGroup disableChain startNewChain 用法案例**](https://blog.csdn.net/qq_31866793/article/details/102786249)