



# Flink 运行架构

讲师：武晟然

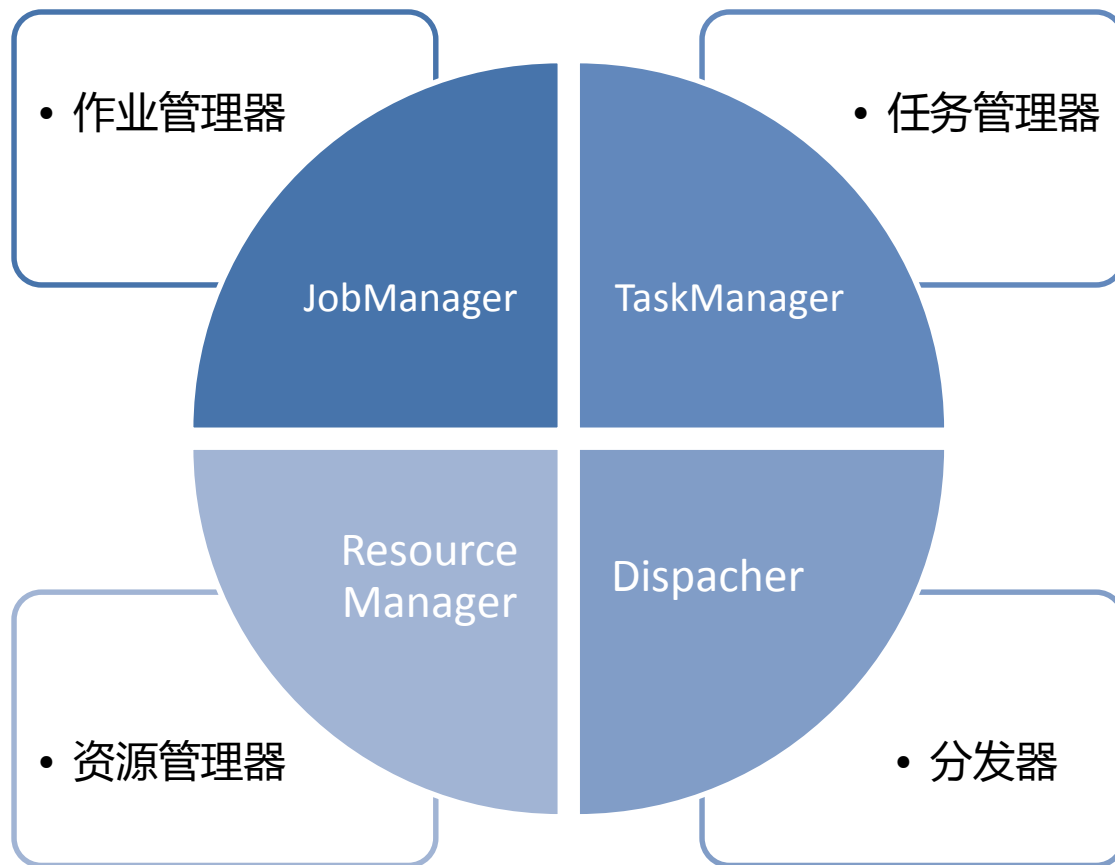


# 主要内容

- Flink 运行时的组件
- 任务提交流程
- 任务调度原理



# Flink 运行时的组件





# 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。



# 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。



# 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。

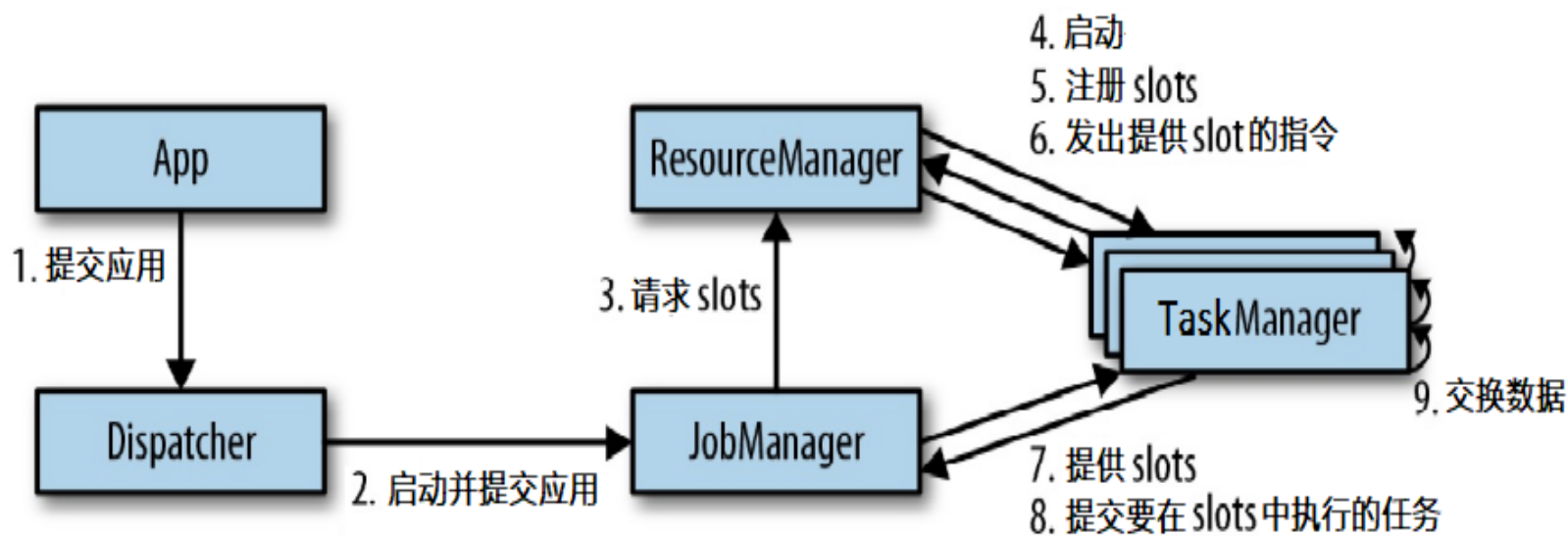


# 分发器 ( Dispatcher )

- 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。
- 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



这里的jobmanager是针对一个job提供的进程，不是一个节点。



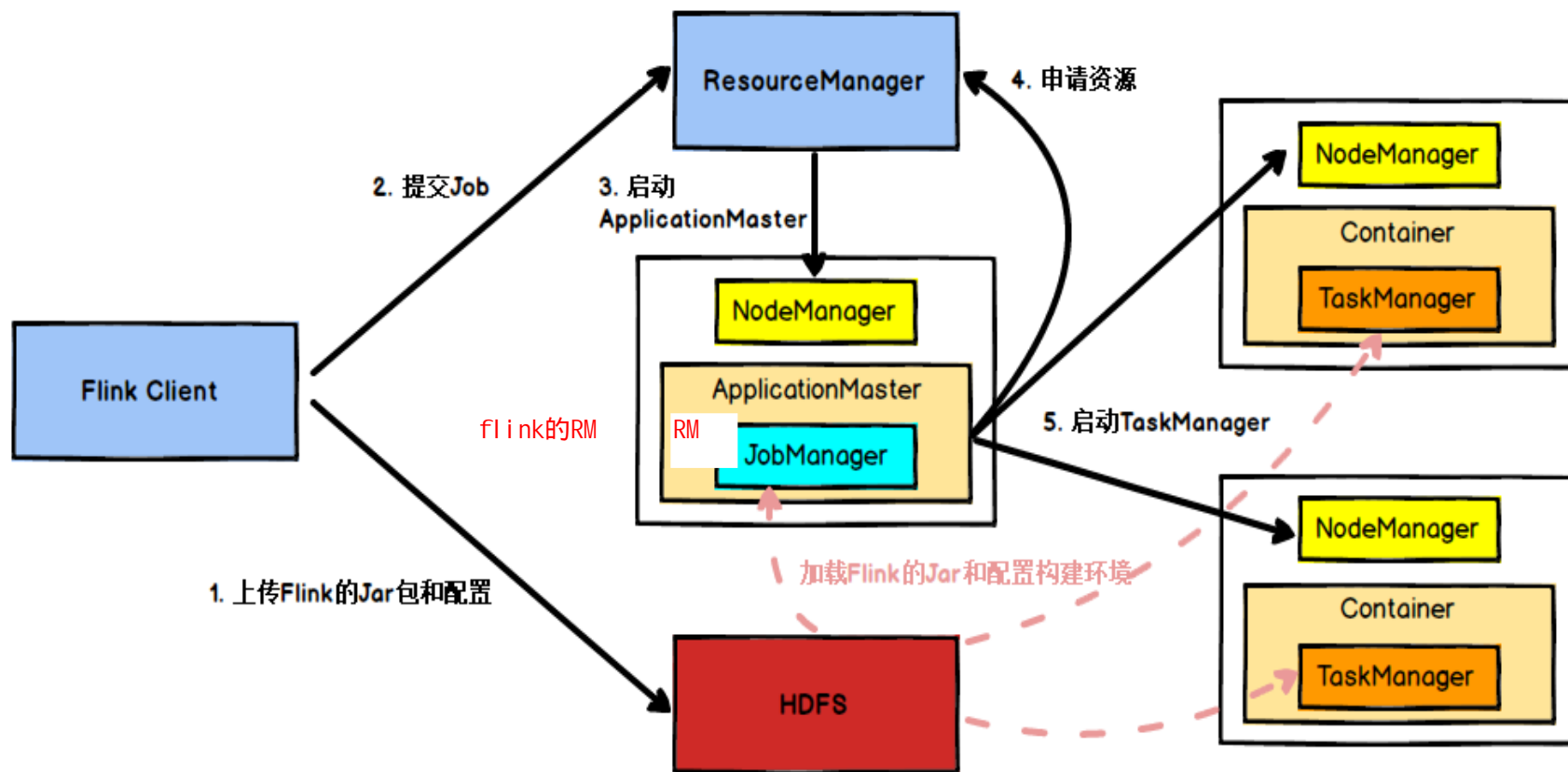


# 任务提交流程 ( YARN )

结合上图理解

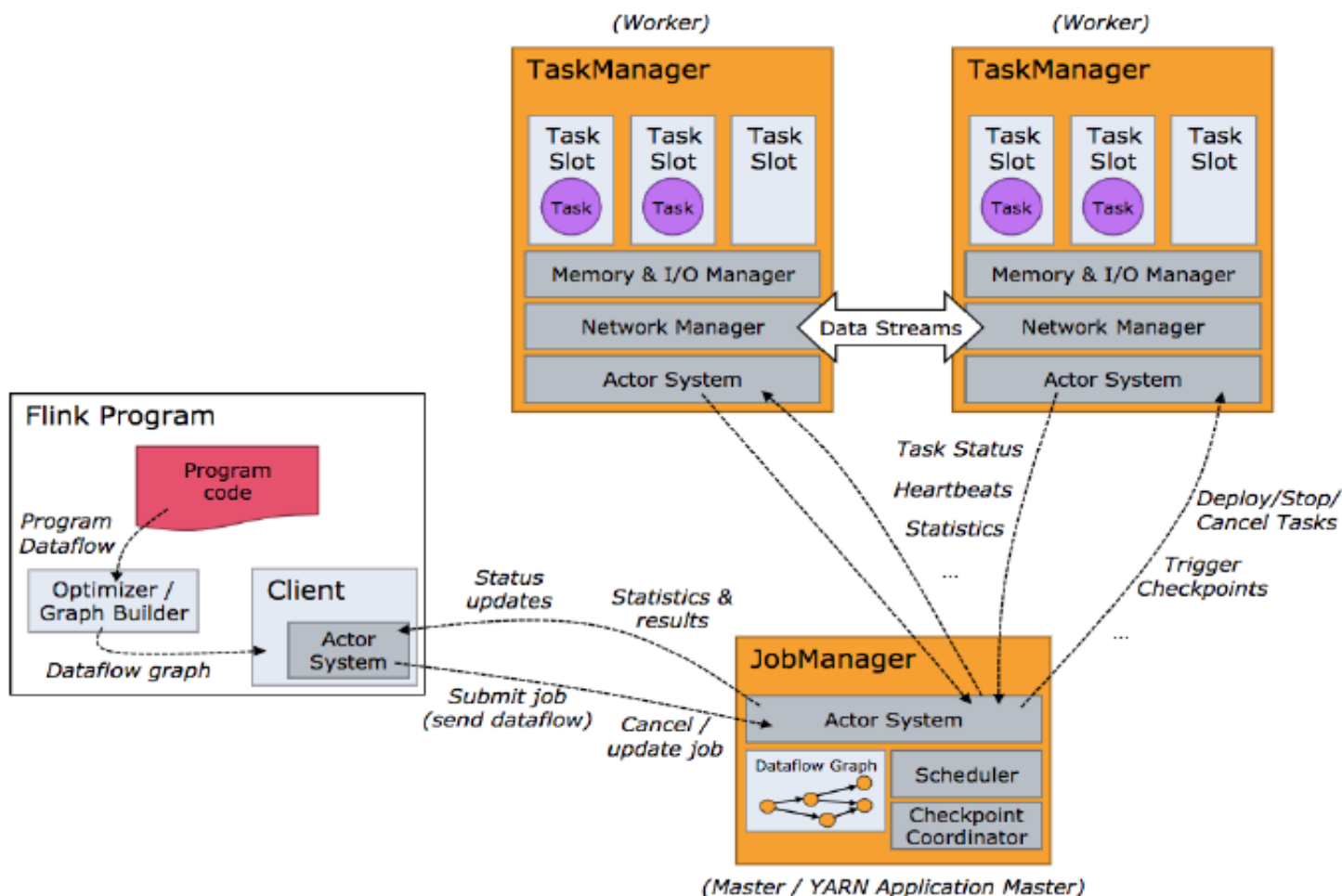
Per-Job-Cluster 模式

这是yarn的RM





# 任务调度原理





# 思考

- 怎样实现并行计算？

通过设置并行任务（并行度），从而将任务分配到不同的slot上，而每个slot 就可以运行一个单独的线程，所以多个slot就运行多线程的任务。



- 并行的任务，需要占用多少slot？

提交job后，需要的slot和每步任务的最大并行度有关。  
如果最大并行度是3，即使有7个任务，也只需要3个slot【不考虑重新设置了slot共享组】

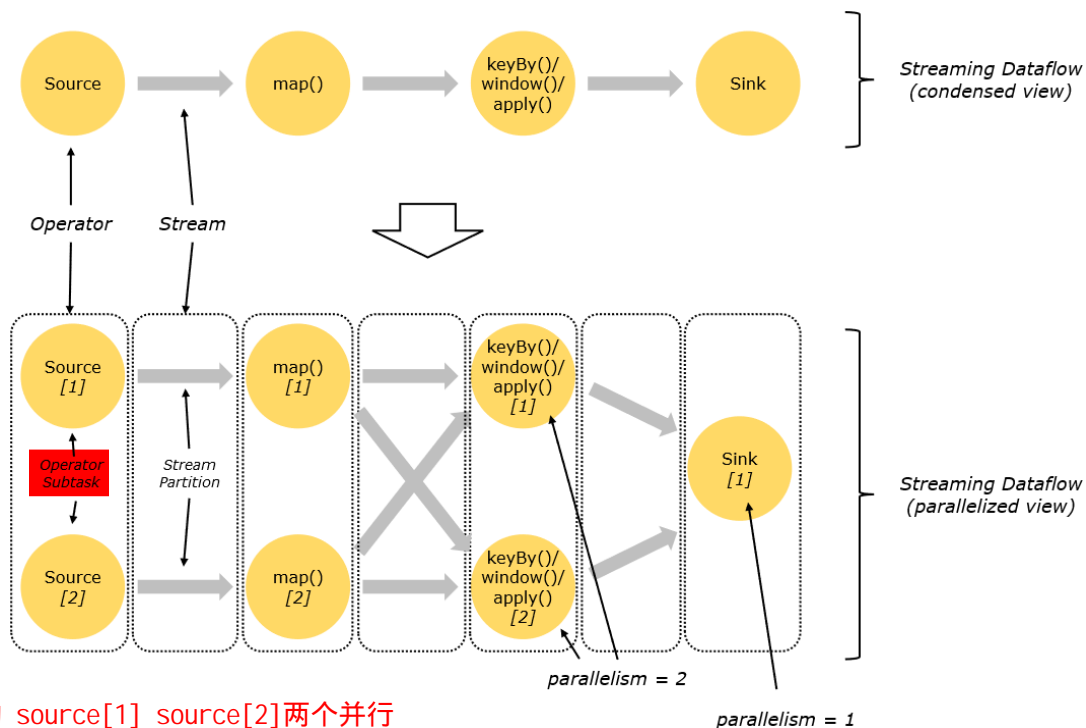
- 一个流处理程序，到底包含多少个任务？

参考下面的任务链。

具体看各算子的并行度、算子间的数据传输模式和是否是一个共享组三个条件



# 并行度 ( Parallelism )



如上图中的 source[1] source[2]两个并行子任务

- 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 stream 的并行度，可以认为就是其所有算子中最大的并行度。

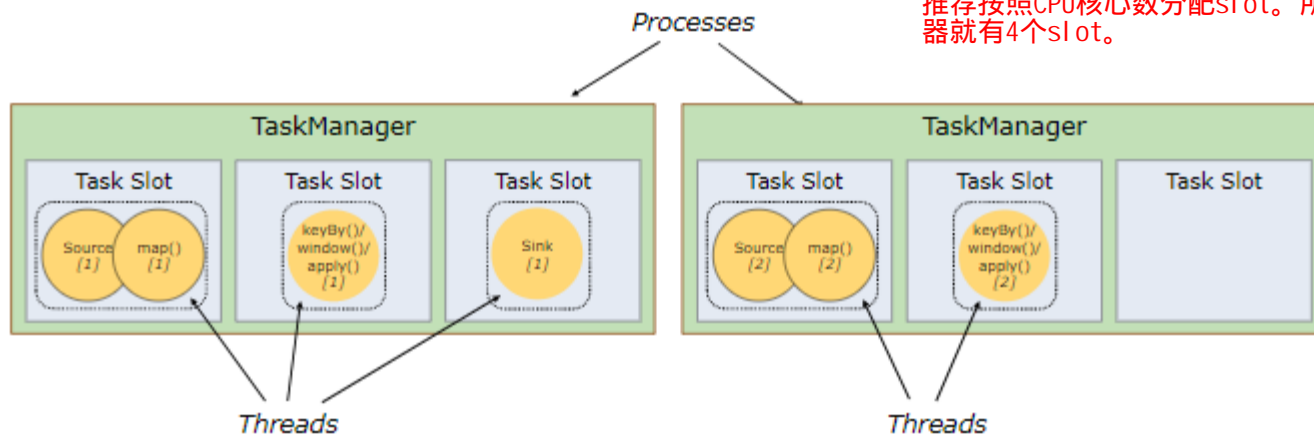
注意区分算子的并行度和整个流的并行度。



# TaskManager 和 Slots

一个slot就是计算一个子任务（一个线程）所需资源的最小计算单元。

推荐按照CPU核心数分配slot。所以，4核的机器就有4个slot。



- Flink 中每一个 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子任务
- 为了控制一个 TaskManager 能接收多少个 task，TaskManager 通过 task slot 来进行控制（一个 TaskManager 至少有一个 slot）



# TaskManager 和 Slots

如何分配slot:

比如下图:

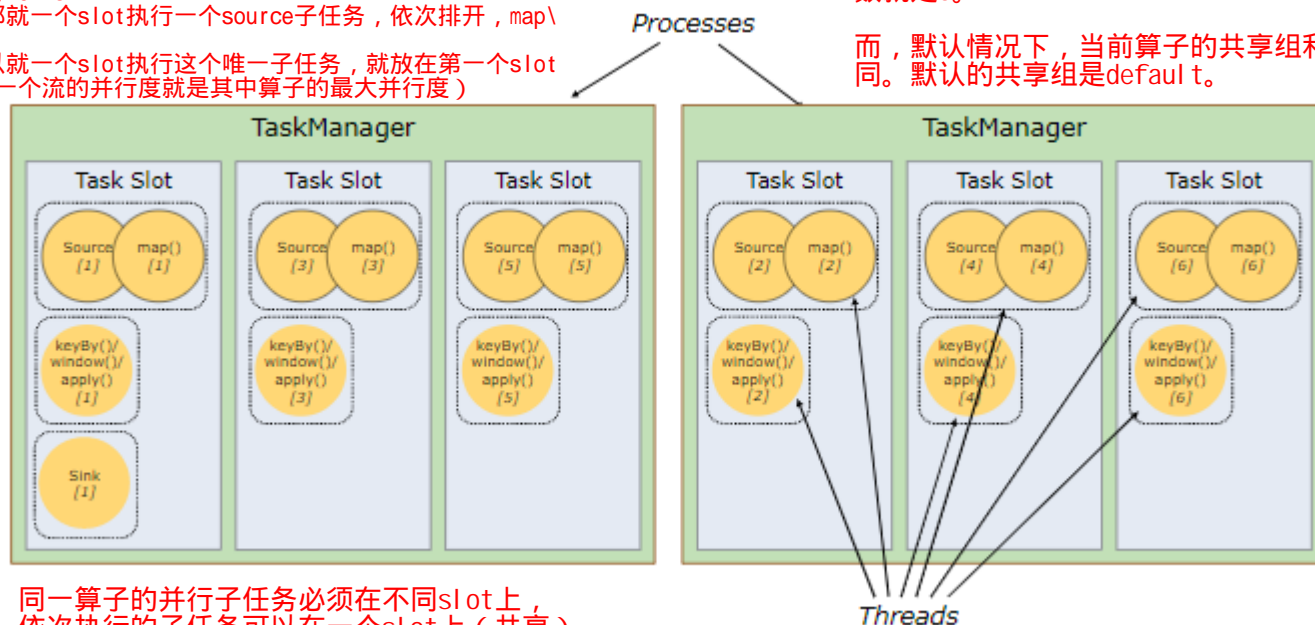
source-6 map-6 keyby-6 sink-1

source的并行度是6, 那就一个slot执行一个source子任务, 依次排开, map\keyby同理, sink的并行度是1, 所以就一个slot执行这个唯一子任务, 就放在第一个slot上了。(这就对应上了一个流的并行度就是其中算子的最大并行度)

针对每个算子设置slot共享组, 可以重新配置使用的slot数量, 其中组内的任务共享slot, 组间任务使用不同的slot。

此时, 程序使用的slot数就是各组的最大并行度的和, 比如, 组1的最大并行度1, 组2的最大并行度2, 那么程序使用的slot数就是3。

而, 默认情况下, 当前算子的共享组和前一个算子的共享组相同。默认的共享组是default。

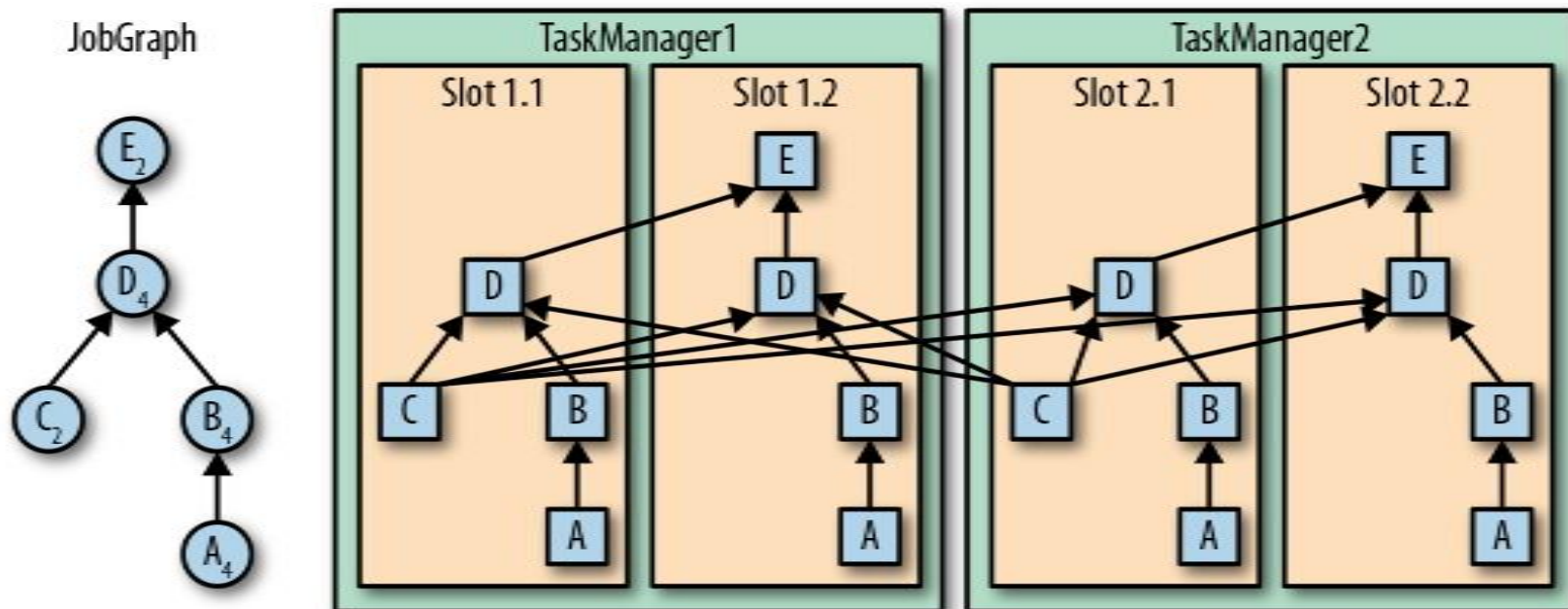


同一算子的并行子任务必须不同slot上, 依次执行的子任务可以在一个slot上(共享)。

- 默认情况下, Flink 允许子任务共享 slot, 即使它们是不同的任务的子任务。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



# 并行子任务的分配

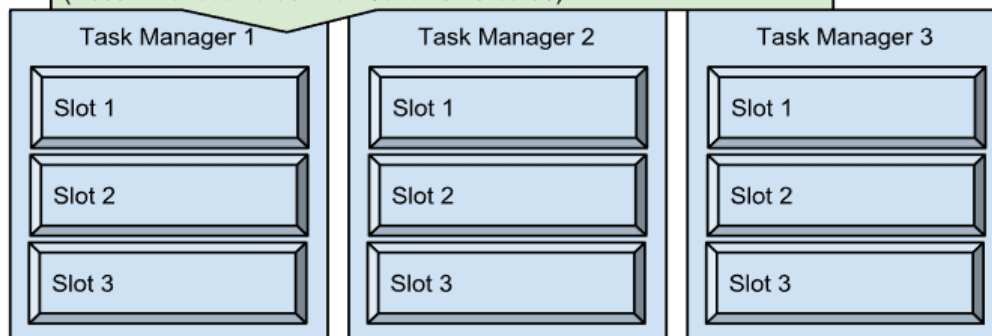




**flink-conf.yaml:**

```
taskmanager.numberOfTaskSlots: 3  
(Recommended value: Number of CPU cores)
```

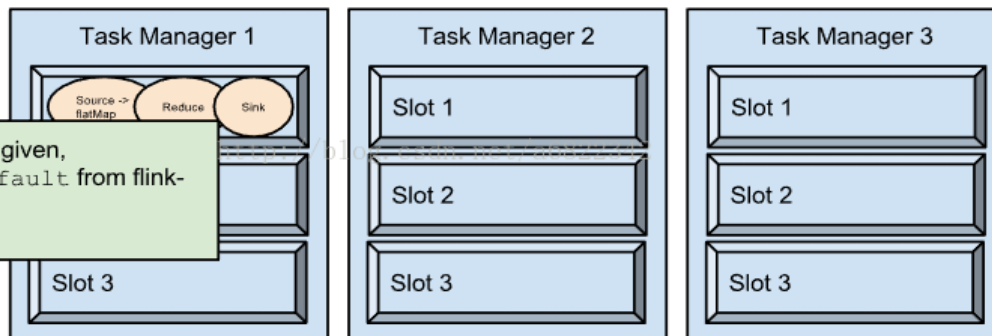
Task Managers: 3  
Total number of  
processing slots: 9



如何在实际程序中  
设置并行度、分配  
slot

Example 1:  
WordCount with  
parallelism = 1

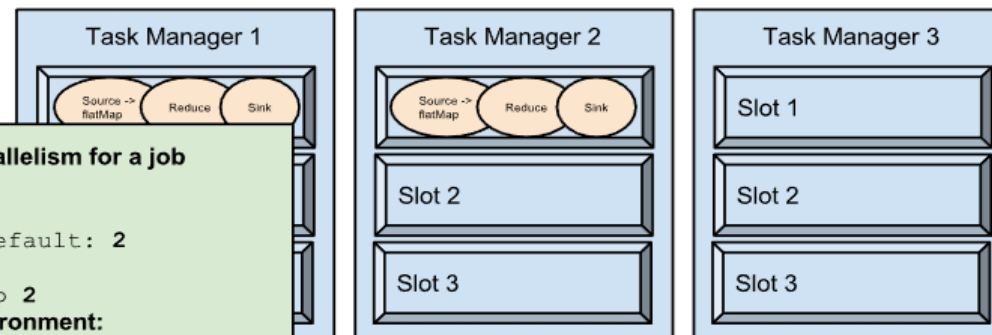
When no argument given,  
parallelism.default from flink-  
conf.yaml is used.  
Default value = 1



Example 2:  
WordCount with  
parallelism = 2

**Places to set parallelism for a job**

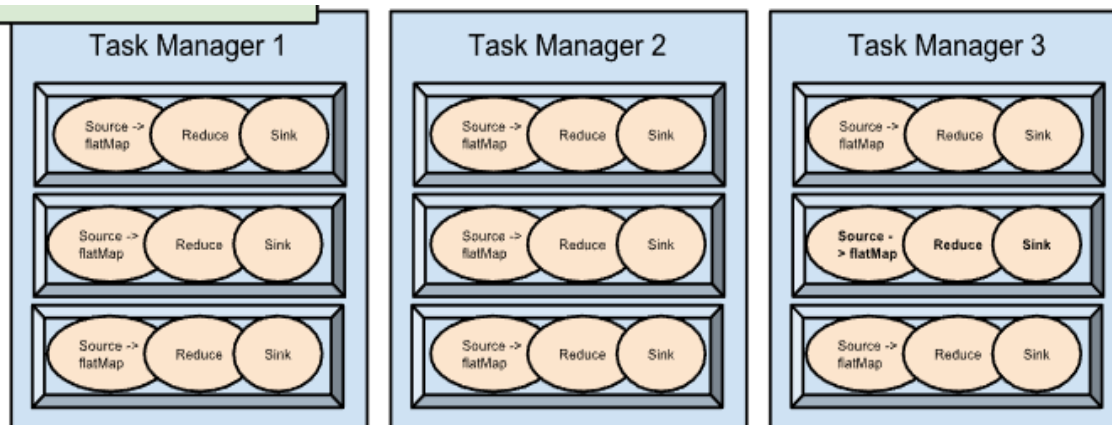
```
flink-conf.yaml:  
parallelism.default: 2  
or Flink Client:  
./bin/flink -p 2  
or ExecutionEnvironment:  
env.setParallelism(2)
```



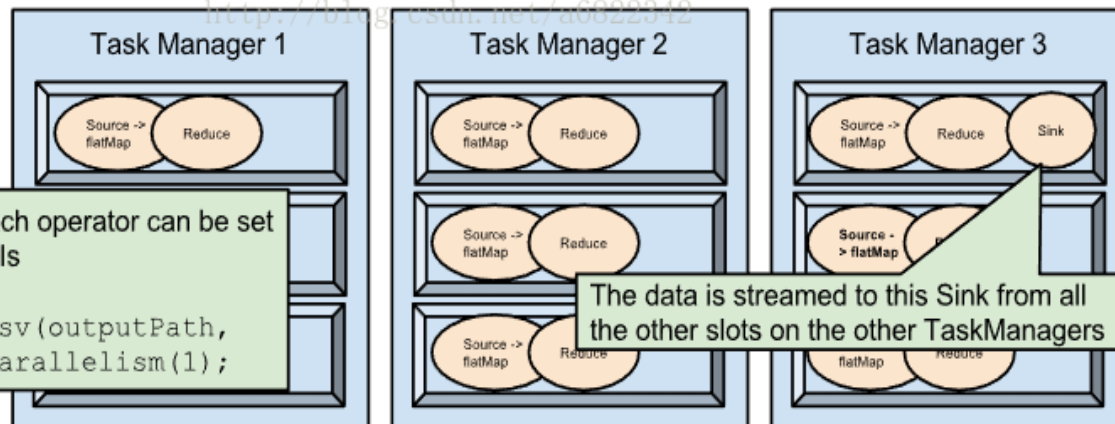




Example 3:  
WordCount with  
**parallelism = 9**



Example 4:  
WordCount with  
**parallelism = 9** and  
**sink parallelism = 1**



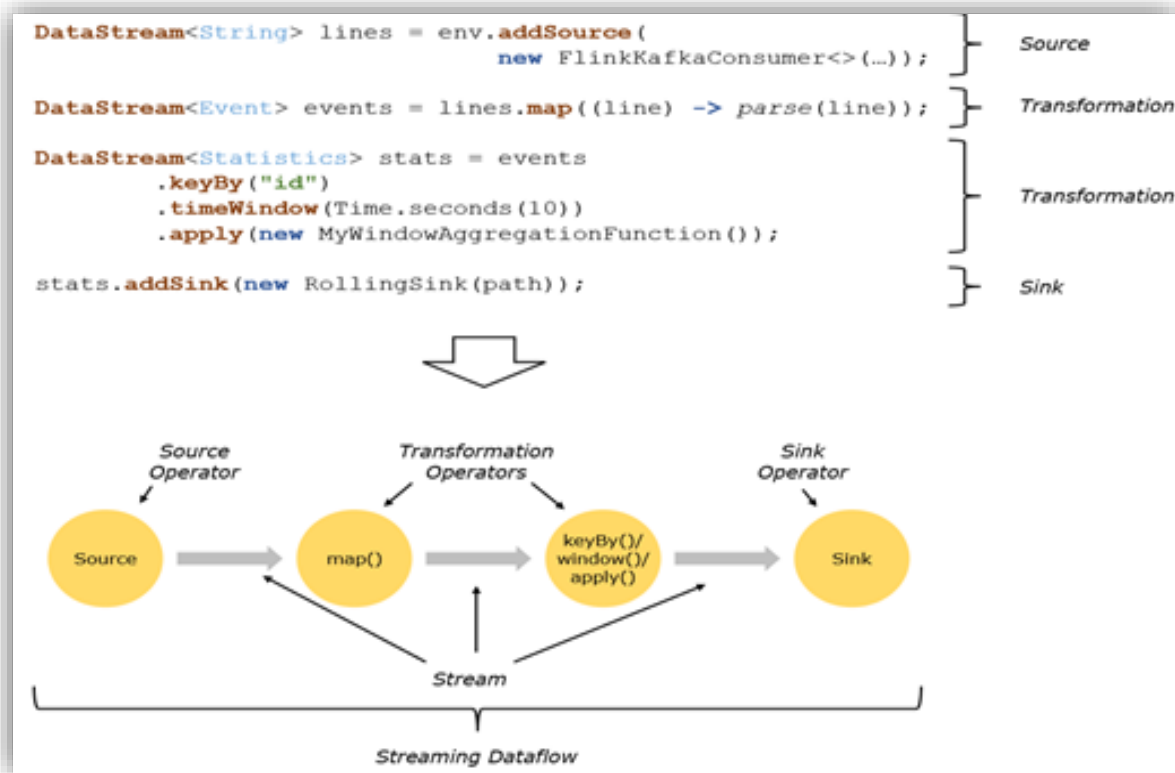
The parallelism of each operator can be set individually in the APIs

```
counts.writeAsCsv(outputPath,  
"\n", " ").setParallelism(1);
```

The data is streamed to this Sink from all the other slots on the other TaskManagers



# 程序与数据流 ( DataFlow )

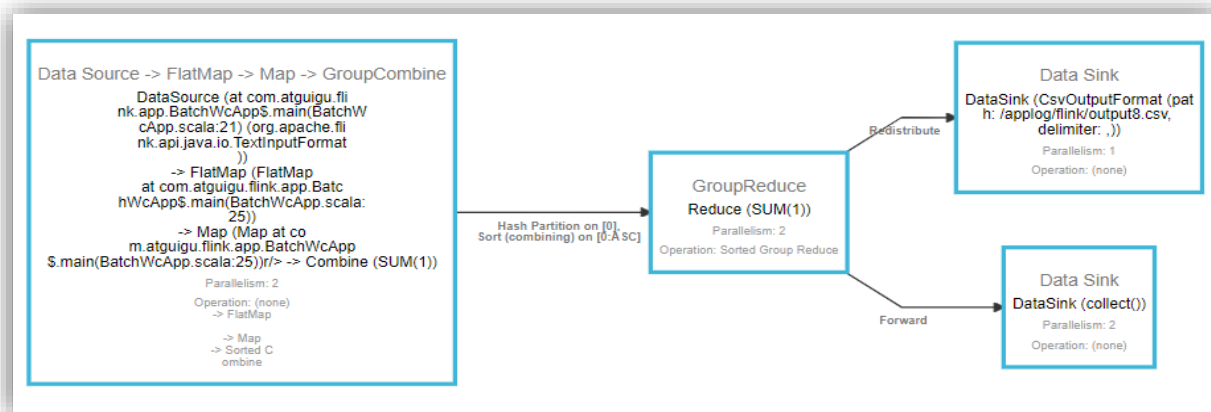


- 所有的Flink程序都是由三部分组成的： Source 、 Transformation 和 Sink。
- Source 负责读取数据源， Transformation 利用各种算子进行处理加工， Sink 负责输出



# 程序与数据流 ( DataFlow )

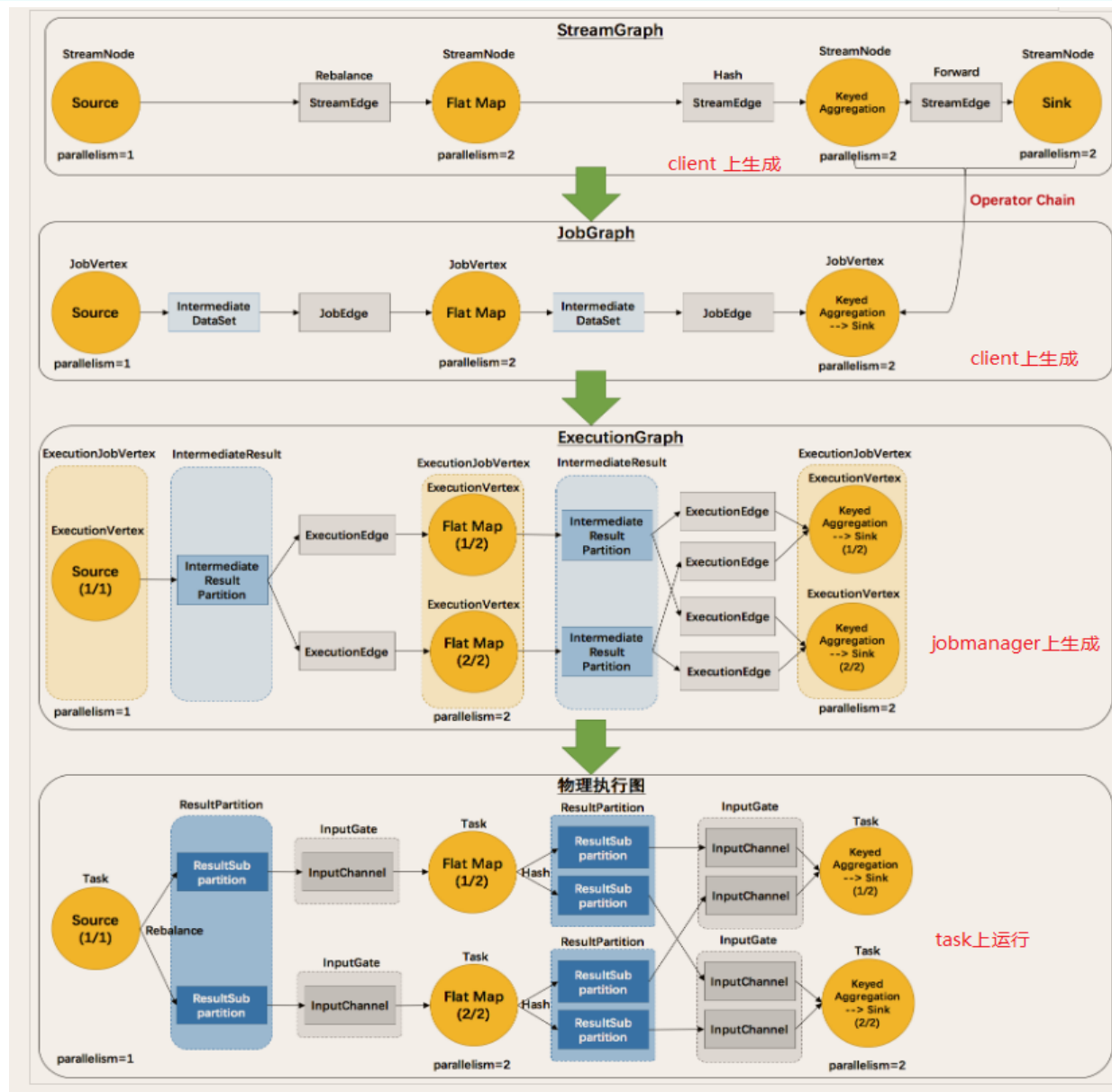
- 在运行时，Flink上运行的程序会被映射成“逻辑数据流”（dataflows），它包含了这三部分
- 每一个dataflow以一个或多个sources开始以一个或多个sinks结束。dataflow类似于任意的有向无环图（DAG）
- 在大部分情况下，程序中的转换运算（transformations）跟dataflow中的算子（operator）是一一对应的关系





# 执行图 ( ExecutionGraph )

- 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。
- 客户端上 ➤ 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
- 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。





# 数据传输形式

- 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、filter、flatMap等算子都是one-to-one 的对应关系。在一个线程中执行
- 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。  
rebalance: 轮询，如，先发下游的分区1，再发下游的分区2，再发分区1...

flink因为是流处理，它的shuffle类似是发牌，来条数据就发给下游。而spark是批处理，它的shuffle是洗牌，等数据到齐了，分发给下游。





# 任务链 ( Operator Chains )

在满足一定条件下，将前后两个算子合并成一个大的算子。

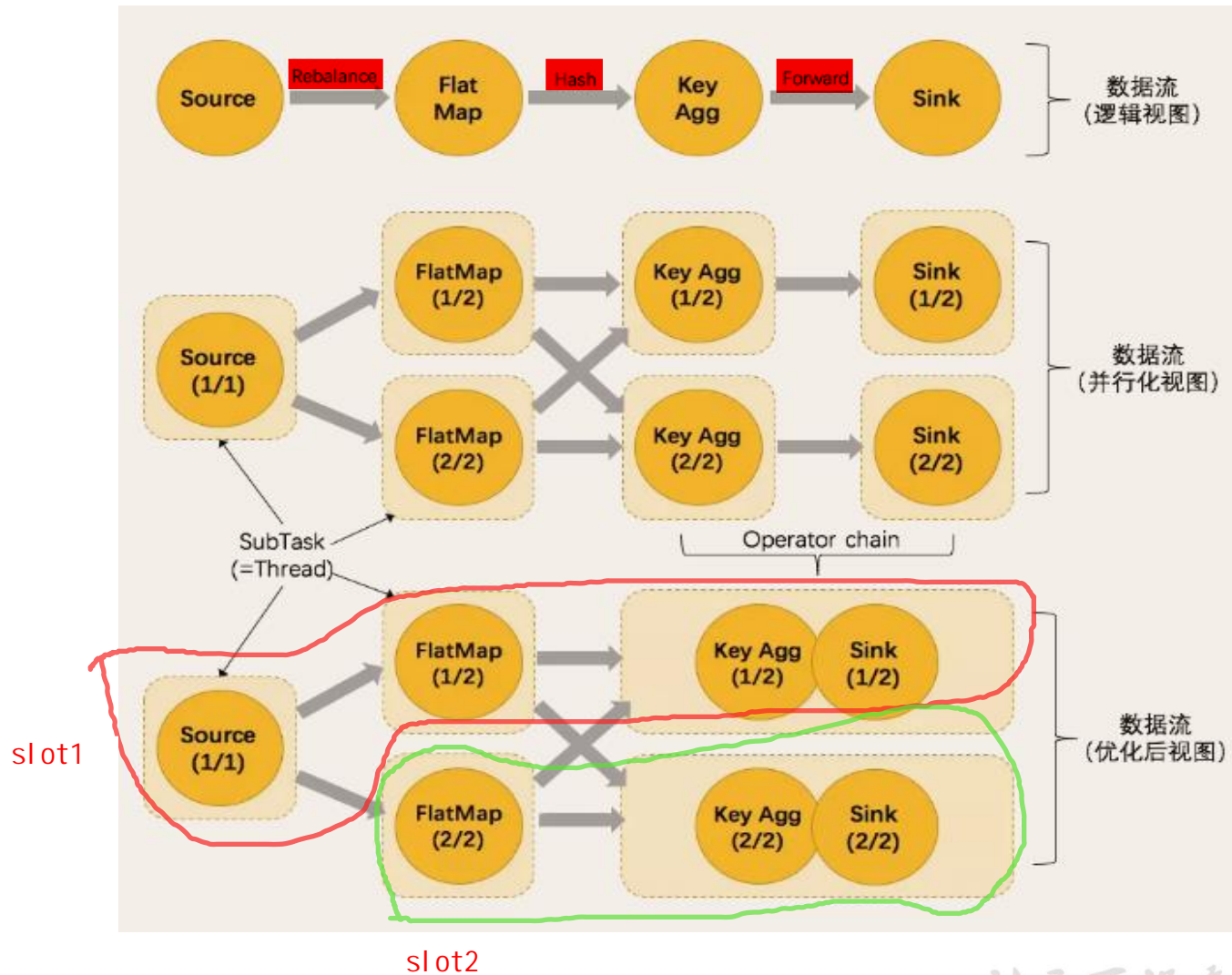
算子后，执行 `disableChain()` 断开链。或执行 `env.disableOperatorChaining()` 全局断开

- 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  
且是同一slot共享组





# 任务链 ( Operator Chains )







# Q & A