# Flink 原理与实现：理解 Flink 中的计算资源

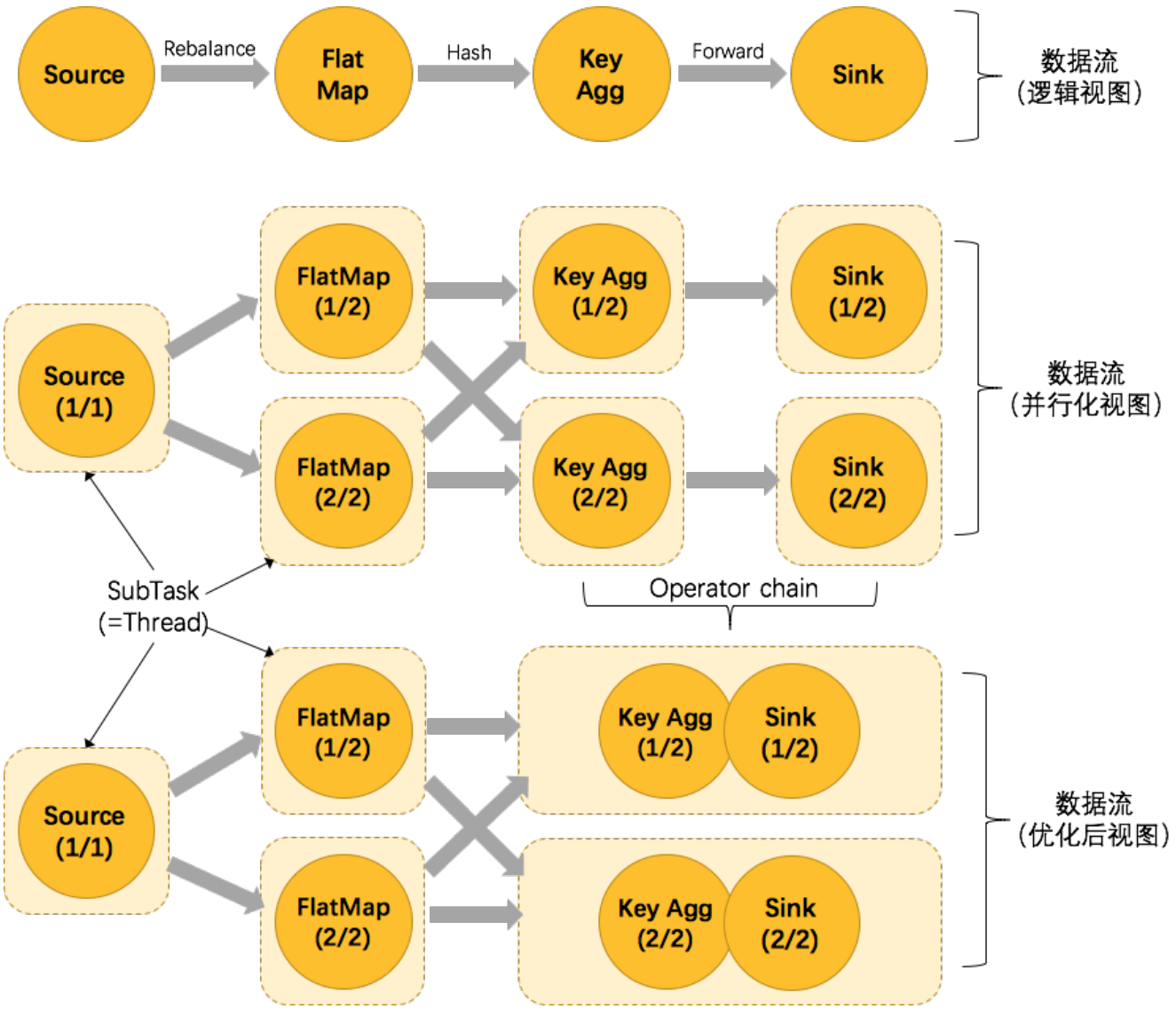
**摘要：** 本文所讨论的计算资源是指用来执行 Task 的资源，是一个逻辑概念。本文会介绍 Flink 计算资源相关的一些核心概念，如：Slot、SlotSharingGroup、CoLocationGroup、Chain等。并会着重讨论 Flink 如何对计算资源进行管理和隔离，如何将计算资源利用率最大化等等。理解 Flink 中的计算资源对于理解 Job 如何在集群中运行的有很大的帮助，也有利于我们更透彻

本文所讨论的计算资源是指用来执行 Task 的资源，是一个逻辑概念。本文会介绍 Flink 计算资源相关的一些核心概念，如：Slot、SlotSharingGroup、CoLocationGroup、Chain等。并会着重讨论 Flink 如何对计算资源进行管理和隔离，如何将计算资源利用率最大化等等。理解 Flink 中的计算资源对于理解 Job 如何在集群中运行的有很大的帮助，也有利于我们更透彻地理解 Flink 原理，更快速地定位问题。

## Operator Chains

为了更高效地分布式执行，Flink会尽可能地将operator的subtask链接（chain）在一起形成task。每个task在一个线程中执行。将operators链接成task是非常有效的优化：它能减少线程之间的切换，减少消息的序列化/反序列化，减少数据在缓冲区的交换，减少了延迟的同时提高整体的吞吐量。

我们仍以经典的 WordCount 为例（参考前文Job例子），下面这幅图，展示了Source并行度为1，FlatMap、KeyAggregation、Sink并行度均为2，最终以5个并行的线程来执行的优化过程。




上图中将KeyAggregation和Sink两个operator进行了合并，因为这两个合并后并不会改变整体的拓扑结构。但是，并不是任意两个 operator 就能 chain 一起的。其条件还是很苛刻的：

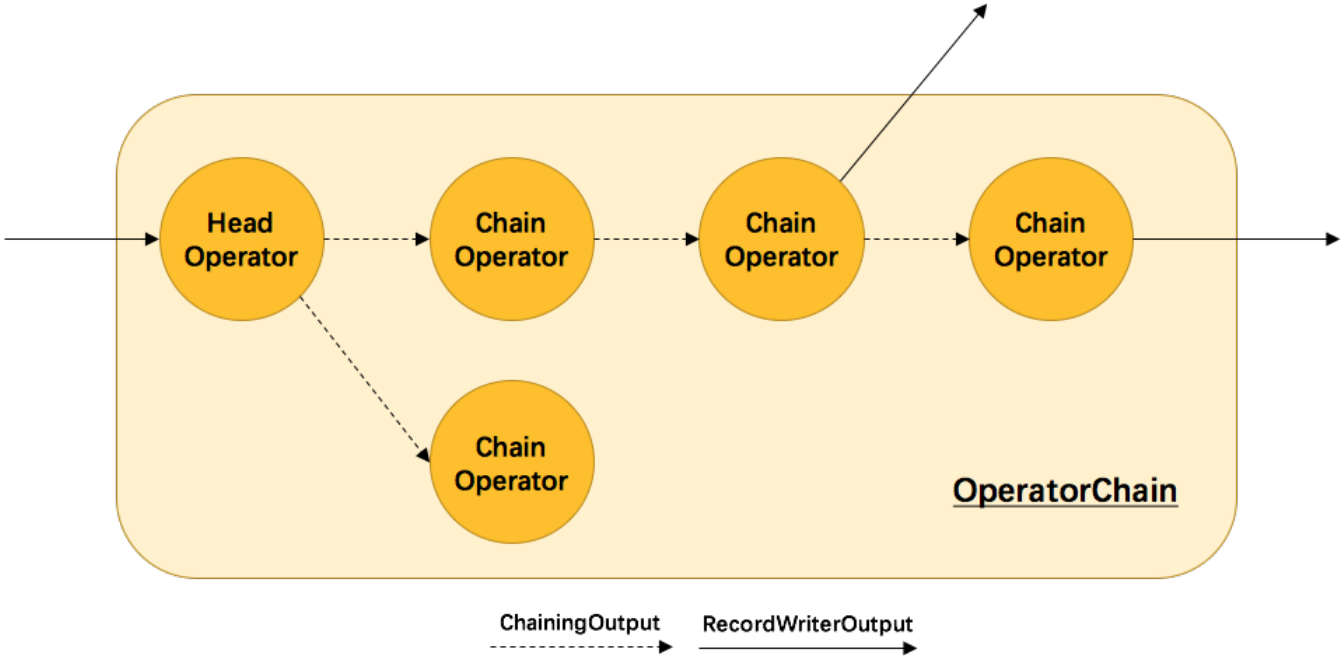
1. 上下游的并行度一致
2. 下游节点的入度为1 （也就是说下游节点没有来自其他节点的输入）
3. 上下游节点都在同一个 slot group 中（下面会解释 slot group）
4. 下游节点的 chain 策略为 ALWAYS（可以与上下游链接，map、flatmap、filter等默认是ALWAYS）
5. 上游节点的 chain 策略为 ALWAYS 或 HEAD（只能与下游链接，不能与上游链接，Source默认是HEAD）
6. 两个节点间数据分区方式是 forward（参考理解数据流的分区）
7. 用户没有禁用 chain

Operator chain的行为可以通过编程API中进行指定。可以通过在DataStream的operator后面（如`someStream.map(..)`)调用`startNewChain()`来指示从该operator开始一个新的chain（与前面截断，不会被chain到前面）。或者调用`disableChaining()`来指示该operator不参与chaining（不会与前后的operator chain一起）。在底层，这两个方法都是通过调整operator的 chain 策略

（HEAD、NEVER）来实现的。另外，也可以通过调用`StreamExecutionEnvironment.disableOperatorChaining()`来全局禁用chaining。

## 原理与实现

那么 Flink 是如何将多个 operators chain在一起的呢？chain在一起的operators是如何作为一个整体被执行的呢？它们之间的数据流又是如何避免了序列化/反序列化以及网络传输的呢？下图展示了operators chain的内部实现：




如上图所示，Flink内部是通过`OperatorChain`这个类来将多个operator链在一起形成一个新的operator。`OperatorChain`形成的框框就像一个黑盒，Flink 无需知道黑盒中有多少个ChainOperator、数据在chain内部是怎么流动的，只需要将input数据交给 HeadOperator 就可以了，这就使得`OperatorChain`在行为上与普通的operator无差别，上面的OperaotrChain就可以看做是一个入度为1，出度为2的operator。所以在实现中，对外可见的只有HeadOperator，以及与外部连通的实线输出，这些输出对应了JobGraph中的JobEdge，在底层通过`RecordWriterOutput`来实现。另外，框中的虚线是operator chain内部的数据流，这个流内的数据不会经过序列化/反序列化、网络传输，而是直接将消息对象传递给下游的 ChainOperator 处理，这是性能提升的关键点，在底层是通过 `ChainingOutput` 实现的，源码如下方所示，

*注：HeadOperator和ChainOperator并不是具体的数据结构，前者指代chain中的第一个operator，后者指代chain中其余的operator，它们实际上都是`StreamOperator`。*

```
private static class ChainingOutput<T> implements Output<StreamRecord<T>> {
    // 注册的下游operator
    protected final OneInputStreamOperator<T, ?> operator;
    public ChainingOutput(OneInputStreamOperator<T, ?> operator) {
        this.operator = operator;
    }
```

```
    @Override
    // 发送消息方法的实现，直接将消息对象传递给operator处理，不经过序列化/反序列化、网络传输
    public void collect(StreamRecord<T> record) {
        try {
            operator.setKeyContextElement1(record);
            // 下游operator直接处理消息对象
            operator.processElement(record);
        }
        catch (Exception e) {
            throw new ExceptionInChainedOperatorException(e);
        }
    }
    ...
}
```

## Task Slot

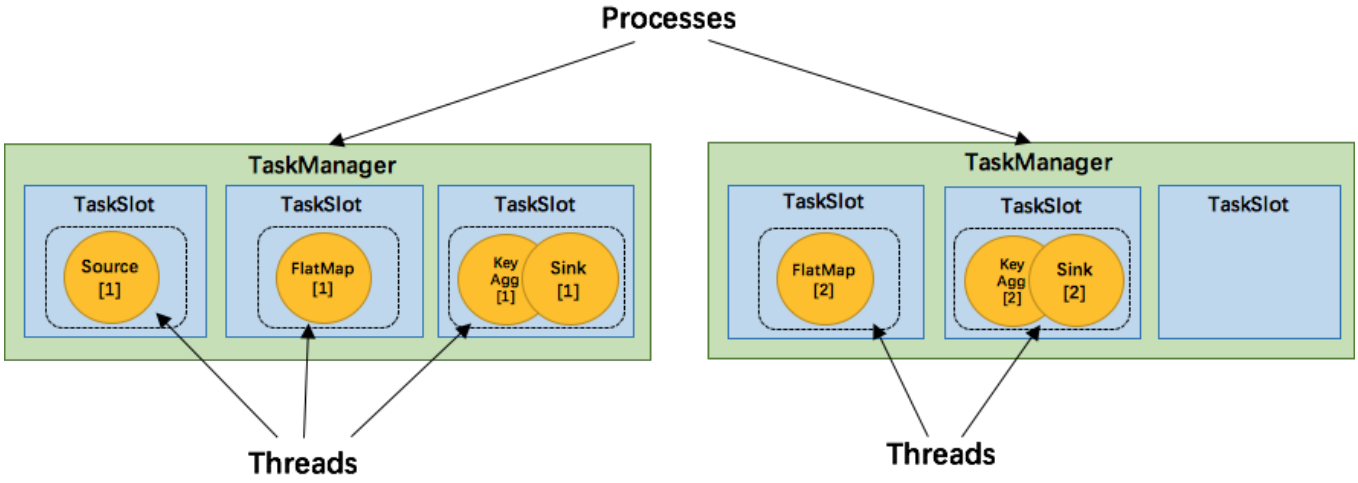
在架构概览中我们介绍了 TaskManager 是一个 JVM 进程，并会以独立的线程来执行一个task或多个subtask。为了控制一个 TaskManager 能接受多少个 task，Flink 提出了 *Task Slot* 的概念。

Flink 中的计算资源通过 *Task Slot* 来定义。每个 task slot 代表了 TaskManager 的一个固定大小的资源子集。例如，一个拥有3个slot的 TaskManager，会将其管理的内存平均分成三分分给各个 slot。将资源 slot 化意味着来自不同job的task不会为了内存而竞争，而是每个task都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的内存。

通过调整 task slot 的数量，用户可以定义task之间是如何相互隔离的。每个 TaskManager 有一个 slot，也就意味着每个task运行在独立的 JVM 中。每个 TaskManager 有多个slot的话，也就是说多个task运行在同一个JVM中。而在同一个JVM进程中的task，可以共享TCP连接（基于多路复用）和心跳消息，可以减少数据的网络传输。也能共享一些数据结构，一定程度上减少了每个task的消耗。

每一个 TaskManager 会拥有一个或多个的 task slot，每个 slot 都能跑由多个连续 task 组成的一个 pipeline，比如 MapFunction 的第n个并行实例和 ReduceFunction 的第n个并行实例可以组成一个 pipeline。

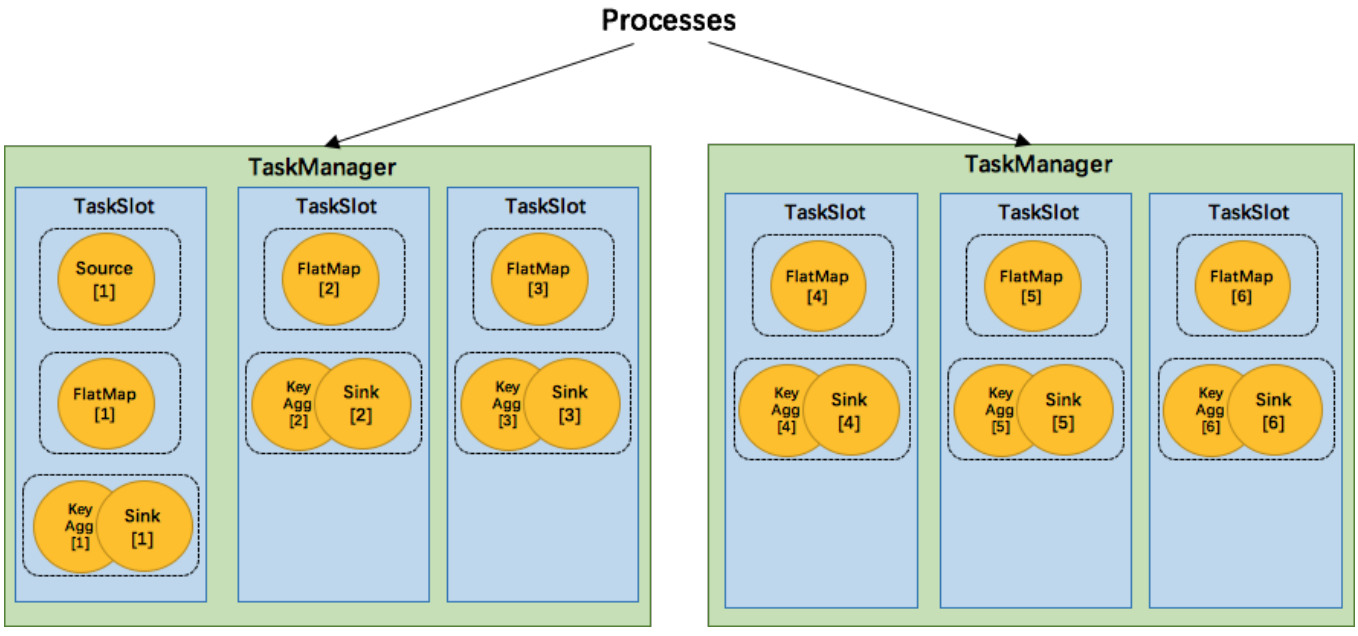
如上文所述的 WordCount 例子，5个Task可能会在TaskManager的slots中如下图分布，2个TaskManager，每个有3个slot：




## SlotSharingGroup 与 CoLocationGroup

默认情况下，Flink 允许subtasks共享slot，条件是它们都来自同一个Job的不同task的subtask。结果可能一个slot持有该job的整个pipeline。允许slot共享有以下两点好处：

1. Flink 集群所需的task slots数与job中最高的并行度一致。也就是说我们不需要再去计算一个程序总共会起多少个task了。
2. 更容易获得更充分的资源利用。如果没有slot共享，那么非密集型操作source/flatmap就会占用同密集型操作 keyAggregation/sink 一样多的资源。如果有slot共享，将基线的2个并行度增加到6个，能充分利用slot资源，同时保证每个TaskManager能平均分配到重的subtasks。




我们将 WordCount 的并行度从之前的2个增加到6个（Source并行度仍为1），并开启slot共享（所有operator都在default共享组），将得到如上图所示的slot分布图。首先，我们不用去计算这个job会其多少个task，总之该任务最终会占用6个slots（最高并行度为6）。其次，我们可以看到密集型操作 keyAggregation/sink 被平均地分配到各个 TaskManager。

`SlotSharingGroup`是Flink中用来实现slot共享的类，它尽可能地让subtasks共享一个slot。相应

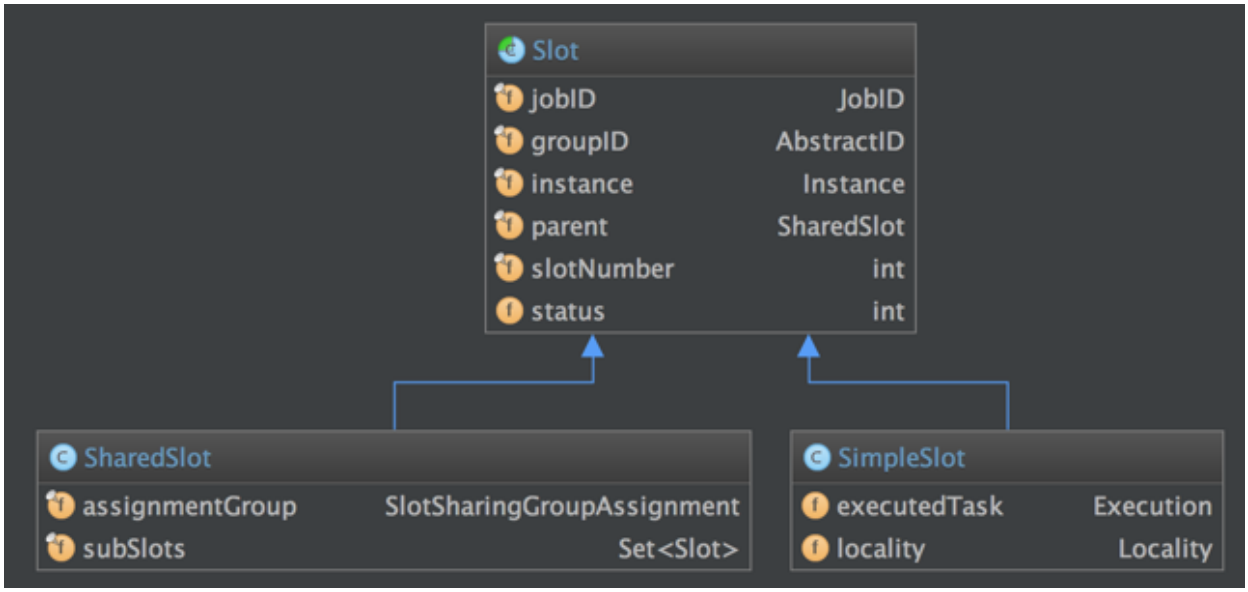
的，还有一个 `CoLocationGroup` 类用来强制将 subtasks 放到同一个 slot 中。`CoLocationGroup`主要用于迭代流中，用来保证迭代头与迭代尾的第i个subtask能被调度到同一个TaskManager上。这里我们不会详细讨论`CoLocationGroup`的实现细节。

怎么判断operator属于哪个 slot 共享组呢？默认情况下，所有的operator都属于默认的共享组`default`，也就是说默认情况下所有的operator都是可以共享一个slot的。而当所有input operators具有相同的slot共享组时，该operator会继承这个共享组。最后，为了防止不合理的共享，用户也能通过API来强制指定operator的共享组，比如：`someStream.filter(...).slotSharingGroup("group1");`就强制指定了filter的slot共享组为`group1`。

## 原理与实现

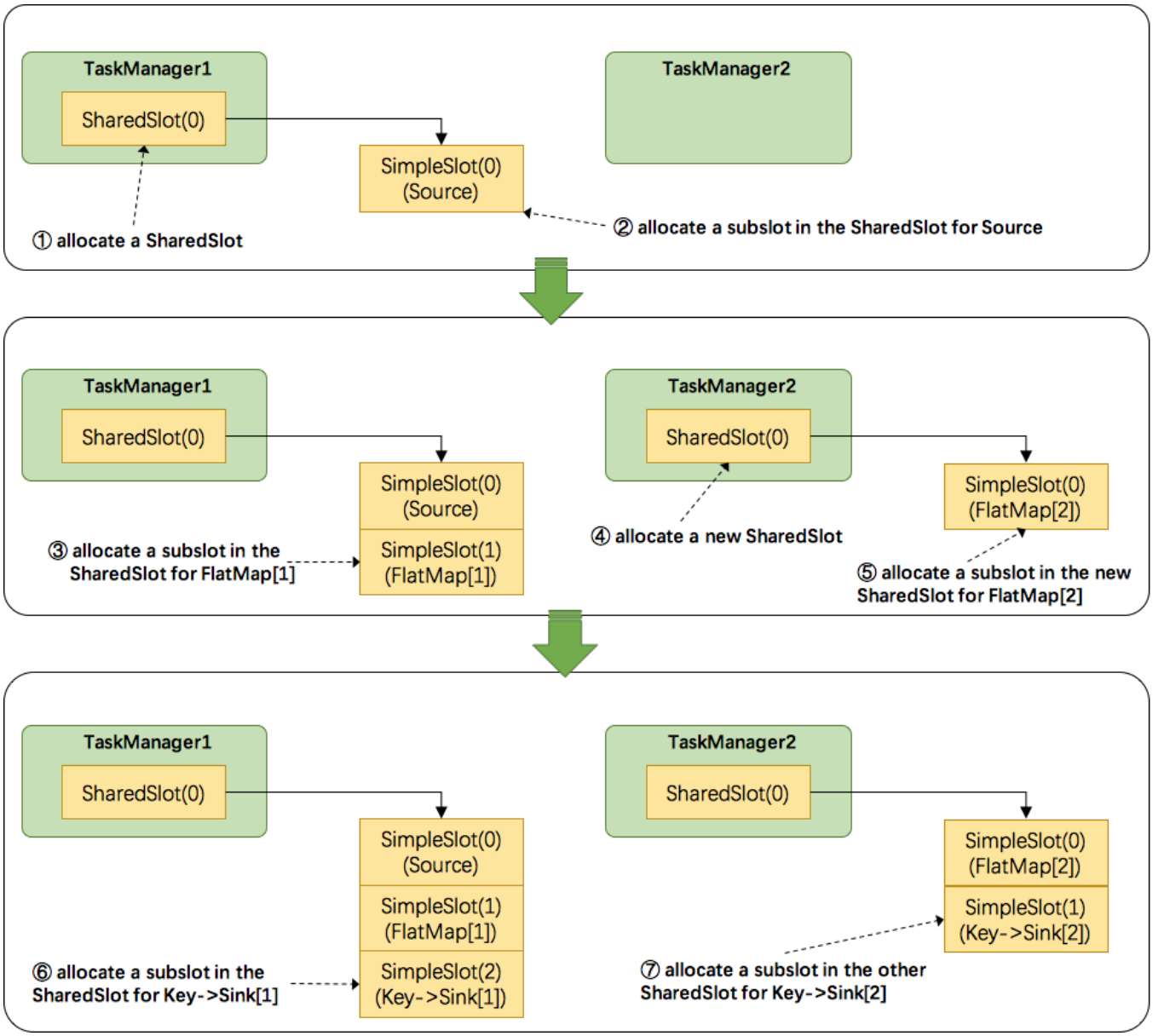
那么多个tasks（或者说operators）是如何共享slot的呢？

我们先来看一下用来定义计算资源的slot的类图：




抽象类`Slot`定义了该槽位属于哪个TaskManager（`instance`）的第几个槽位（`slotNumber`），属于哪个Job（`jobID`）等信息。最简单的情况下，一个slot只持有一个task，也就是`SimpleSlot`的实现。复杂点的情况，一个slot能共享给多个task使用，也就是`SharedSlot`的实现。SharedSlot能包含其他的SharedSlot，也能包含SimpleSlot。所以一个SharedSlot能定义出一棵slots树。

接下来我们来看看 Flink 为subtask分配slot的过程。关于Flink调度，有两个非常重要的原则我们必须知道：（1）同一个operator的各个subtask是不能呆在同一个SharedSlot中的，例如`FlatMap[1]`和`FlatMap[2]`是不能在同一个SharedSlot中的。（2）Flink是按照拓扑顺序从Source一个个调度到Sink的。例如WordCount（Source并行度为1，其他并行度为2），那么调度的顺序依次是：`Source` -> `FlatMap[1]` -> `FlatMap[2]` -> `KeyAgg->Sink[1]` -> `KeyAgg->Sink[2]`。假设现在有2个TaskManager，每个只有1个slot（为简化问题），那么分配slot的过程如图所示：




*注：图中 SharedSlot 与 SimpleSlot 后带的括号中的数字代表槽位号（slotNumber）*

1. 为`Source`分配slot。首先，我们从TaskManager1中分配出一个SharedSlot。并从SharedSlot中为`Source`分配出一个SimpleSlot。如上图中的①和②。
2. 为`FlatMap[1]`分配slot。目前已经有一个SharedSlot，则从该SharedSlot中分配出一个SimpleSlot用来部署`FlatMap[1]`。如上图中的③。
3. 为`FlatMap[2]`分配slot。由于TaskManager1的SharedSlot中已经有同operator的`FlatMap[1]`了，我们只能分配到其他SharedSlot中去。从TaskManager2中分配出一个SharedSlot，并从该SharedSlot中为`FlatMap[2]`分配出一个SimpleSlot。如上图的④和⑤。
4. 为`Key->Sink[1]`分配slot。目前两个SharedSlot都符合条件，从TaskManager1的SharedSlot中分配出一个SimpleSlot用来部署`Key->Sink[1]`。如上图中的⑥。
5. 为`Key->Sink[2]`分配slot。TaskManager1的SharedSlot中已经有同operator的`Key->Sink[1]`了，则只能选择另一个SharedSlot中分配出一个SimpleSlot用来部署`Key->Sink[2]`。如上图中的⑦。

最后`Source`、`FlatMap[1]`、`Key->Sink[1]`这些subtask都会部署到TaskManager1的唯一一个slot

中，并启动对应的线程。`FlatMap[2]`、`Key->Sink[2]`这些subtask都会被部署到TaskManager2的唯一一个slot中，并启动对应的线程。从而实现了slot共享。

## 总结

本文主要介绍了Flink中计算资源的相关概念以及原理实现。最核心的是 Task Slot，每个slot能运行一个或多个task。为了拓扑更高效地运行，Flink提出了Chaining，尽可能地将operators chain在一起作为一个task来处理。为了资源更充分的利用，Flink又提出了SlotSharingGroup，尽可能地让多个task共享一个slot。

## 参考资料


- Flink: Jobs and Scheduling
- Flink Concepts