# NNVM 的内存分配

## 一、 编译计算图

编译计算图的过程由 GraphCompile 执行，该 pass 位于“nnvm/src/compiler/graph_compile.cc”。

作为在 build 的最后阶段被调用的 pass，它的处理流程是：

1. 处理所有可融合的算子节点，然后将其转化为融合后的节点，包括将融合后的算子编译成为 LoweredFunc；
2. 建立原图和融合后的计算图的映射关系，并且据此构建出新的计算图。这一过程中会对 assign 算子进行特殊处理：标记所有的 assign 节点，并且检测可以简化的 assign，将其特殊标记并转为空操作；
3. 将处理完成的计算图进一步 build 为 module，得到目标代码；
4. 执行内存分配，并且对 Placeholder、assign 节点的内存分配情况作出额外处理。

在这篇文档将讨论 NNVM 的内存分配机制，所以我们主要关注上述第二步和第四步。

## 二、 处理 assign 节点

assign 算子是一个较为特殊的算子，在 NNVM 中，只有这个算子具备修改 Variable 内容的能力。所以在处理内存分配时，我们必须对其进行一些特殊处理，使得 assign 算子的输出被存储在输入 Variable 的内存中。否则，按照通常的内存分配方法，即输出总是使用与输入无关的内存空间，将不能实现上述效果。

处理 assign 节点的步骤主要分为两部分：首先，为了方便后续的内存分配处理过程，我们在之前的图构建阶段扫描到 assign 节点时，就可以直接对其进行标记。标记结果将会存储在数组 assign_flag 中（参考“graph_compile.cc:222”）。

阅读源代码我们还会发现，对 assign 节点的标记情况分为以下三种：

1. 当前节点不属于 assign；

2. 当前节点是一个普通的 assign；
3. 当前节点不仅是 assign，而且 rhs 还是一个只被引用过一次的节点，即：rhs 实际上是一个临时节点，它只为了当前的赋值操作而存在。在这种情况下我们就没必要让 assign 再占用一步操作了，直接让 rhs 的内存和 lhs 分配在同一空间即可，然后这个 assign 就可以被简化了（置为空）。

graph_compile.cc 中，第 236 到 245 行实际上做的就是如上的处理。这些代码完成了不同种 assign 情况的判别，对 assign 节点作出了标记，并且将符合第三种情况的 assign 节点处理为了空操作。

assign 节点的内存分配并不能在现在进行，因为计算图整体的内存分配尚未开始。后续的 DecorateMemoryPlan 函数会针对当前收集到的 assign 标记，对内存分配进行进一步的修改。

## 三、 进行内存分配

内存分配过程由 PlanMemory 执行，该 pass 位于“nnvm/src/pass/plan_memory.cc”。

PlanMemory 的处理流程如下：

1. 计算待分配的计算图中所有节点的引用计数信息；
2. 调用内存分配算法执行内存分配。当环境变量“NNVM_AUTO_SEARCH_MATCH_RANGE”被设置时，该 pass 会执行多次不同的内存分配，然后选出分配的总内存最小的一次，作为最终的分配方案；默认情况下，该 pass 只会执行一次内存分配。

PlanMemory 会首先计算计算图各个节点的引用计数信息，这个信息可以帮助内存分配算法复用一部分已分配的内存。因为在 NNVM 运行计算图时，通常情况下，计算图节点从输入到输出前向传播的流程是串行执行的，每个节点都只会被执行一次。当一个节点的所有输出都已经被执行过一次内存分配之后，当前节点所有的输入的引用计数都会被减少；如果某个输入节点的引用计数已经被减少到了 0，说明这个节点不会再被任何其他节点使用了，其占用的内存也已经可以被其他新分配的节点复用。

这里提到的通常情况指的是用户未指定环境变量“NNVM_EXEC_NUM_TEMP”的情况，根据 NNVM 中图分配器（GraphAllocator）部分的代码，该环境变量控制了 NNVM 对计算

图的划分。如果环境变量的值大于 1，分配器会尝试调用“ColorNodeGroup”对计算图执行图着色，着色算法会将计算图分为若干可并行执行的子图。如果分配内存时发现，待复用内存之前所属的子图和目前待分配的节点所属不同，则当前内存实际上是不会被复用的。

在计算完引用计数后，PlanMemory 会执行实际的内存分配流程。在目前的 NNVM 实现中，内存分配的流程由两个部分辅助实现：

1. AllocMemory：内部函数，执行一次内存分配，PlanMemory 可能会多次调用该函数，以找出最优的内存分配情况；
2. GraphAllocator：内部类，可以实例化一个内存分配器。内存分配器的作用是在分配过程中存储所有节点的内存分配信息，包括所有已分配和已释放的内存块、每个内存块的大小、着色后的计算图划分等等。所有内存分配的流程都由 GraphAllocator 执行，其内部会进行具体的复用内存块的操作。

之前已经提到过，当环境变量“NNVM_AUTO_SEARCH_MATCH_RANGE”被设置时，PlanMemory 会多次调用 AllocMemory 函数，自动找出最省空间的内存分配方案。此过程可以参考“nnvm/src/pass/plan_memory.cc”第 373 到 399 行。

## （一）AllocMemory

AllocMemory 函数具体执行了内存分配的过程。该函数会：

1. 遍历计算图待分配范围中的每一个节点；
2. 检查是否对当前节点执行就地（inplace）内存分配操作。如果当前节点的算子支持 inplace 分配，其“FinplaceOption”属性中会提供对应的处理函数，函数返回内存分配时所需的键值对信息。AllocMemory 则会根据这些信息，在条件允许的情况下，直接将节点输入的内存复用做节点输出；
3. 遍历当前节点的输出。对所有未分到内存的输出，按照所需内存大小的高低次序，为其分配对应大小的内存（调用 GraphAllocator 的 Request 方法）；
4. 减少节点所有输入的引用计数，如果某个输入的引用计数已经为 0，则释放该输入占用的内存（调用 GraphAllocator 的 Release 方法）。忽略“FignoreInputs”属性指定的输入节点；
5. 检查当前节点输出的引用计数，如果某输出已经不会被其他任何节点所引用，则直

接释放已分配的内存。

上述处理过程会跳过所有的 Placeholder（即 variable）。对 variable 的处理位于计算图编译部分的最后，DecorateMemoryPlan 函数中，下文将详细叙述。

## （二）GraphAllocator

GraphAllocator 中需要关注的方法只有两个：

1. Request：根据节点的 dtype 和 shape，给节点分配对应大小的内存，必要时进行内存复用；
2. Release：将指定的内存块标记为“已释放”，后续分配时可对该内存进行复用。

Release 方法较为简单：该方法被调用时，会将指定的内存块放入一个 multimap 中（类的成员变量“free_”）。该 multimap 以内存块的大小为索引，方便复用时快速找到符合大小的空闲内存。除此之外，该方法还会标记内存块之前所属的节点。之前提到过，如果待分配内存的节点和空闲内存块之前所属节点不属于同一个划分，则空闲内存不能被该节点复用。

Request 方法稍复杂：该方法会根据 shape 计算出待分配的内存大小，然后查找之前提到的 multimap，试图进行内存复用。算法优先查找比待分配内存大的空闲块，因为在这种情况下，空闲块可以在不改变大小的情况下直接放入当前节点所需的数据；否则，算法会复用比待分配内存小的空闲块，并且在找到尽可能大的空闲块之后，将其内存大小扩大为待分配大小；如果什么都找不到，算法会分配一个新块。算法执行查找的范围在分配器初始化时指定（“match_range”参数）。

以上几个部分的相互协作，完成了计算图节点内存分配的整个过程。

# 四、 对分配结果做进一步处理

在执行完 PlanMemory 后，GraphCompile 会接着调用 DecorateMemoryPlan 函数，对内存分配的结果做进一步的处理。

DecorateMemoryPlan 函数主要做三件事：

1. 给所有的 variable 指定新的内存块 id（storage_id）。PlanMemory 函数并不会对

variable 执行内存分配，所以它们的内存块 id 均为-1；但是在计算图运行时，variable 需要额外的内存存储图的输入以及所有参数；

2. 处理 assign 节点的内存分配。之前我们说过，assign 节点分三种情况：
   a) 对于第一种情况（节点不属于 assign），该步骤不会执行；
   b) 对于第二种情况（普通 assign），函数会将 assign 节点输出的内存设为 lhs 的内存；
   c) 对于第三种情况（简化后的 assign），函数会同时将节点输出和 rhs 的内存均设为 lhs 的内存；
3. 更新处理过后的 storage_id。

## 五、 运行时的内存分配

NNVM 会使用 GraphRuntime 运行编译后的计算图，GraphRuntime 类的实现位于“src/runtime/graph/graph_runtime.cc”。

计算图在编译完成后，只会存储我们之前提到的”storage_id”信息，即每个节点分配到的内存块的 id。除此之外的诸如内存块大小等信息并不会被导出到配置文件中。GraphRuntime 会在初始化时，读取计算图结束后立即执行运行时的内存分配操作，类的私有方法“SetupStorage”实现了这一过程，方法流程如下：

1. 遍历计算图节点，根据每个节点的 dtype 和 shape 算出占用内存的大小；
2. 根据节点的 storage_id，将节点占用内存大小和 device_id 更新入一个临时的列表；计算图节点遍历完成后，该列表中存储的信息就是每个内存块所占用的最终大小，以及每个内存块对应的 device_id 了；
3. 遍历之前得到的临时列表，在内存池（storage_pool）中实例化所有的 NDArray，其大小和 device_id 由之前的信息确定；
4. 遍历计算图节点，根据内存池中的内存块，为每个节点创建一个 NDArray 引用。这个 NDArray 就是计算图节点运行时的表示了。

以上就是 NNVM 内存分配部分的具体内容。