第5章

中 间 表 示

中间表示是程序编译过程中位于源语言和目标语言之间的程序表示，是编译器的核心数据结构之一。中间表示的两大核心任务是表达和优化，一方面，中间表示作为源程序和机器指令之间的桥梁，能够表达不同类型源程序的结构和语义信息，并且转化为不同类型的硬件设备机器指令；另一方面，中间表示作为编译器的内部表示，能够准确地表达源程序的更多细节，并且完成更多类型的优化。

当前深度学习应用发展迅速，中间表示也在与时俱进、不断发展。深度学习中间表示是一种面向深度学习应用设计和开发的中间表示，该中间表示需要借鉴传统编译器中间表示的设计理念，充分考虑深度学习模型的特点，能够针对模型的特点完成优化，提升模型的性能和效率，适用于与深度学习相关的应用场景和任务。本章从中间表示的概念、分类和深度学习中间表示的设计三个角度入手展开介绍，帮助开发人员深入理解深度学习编译器中间表示的设计方法和重要作用。

5.1 中间表示的概念

编译器通常包括前端、优化器和后端三部分。编译器工作过程中，前端主要负责词法分析和语法分析，并且能够将源程序转换为抽象语法树，优化器主要负责对中间表示进行优化，后端主要负责将优化后的中间表示转换为机器指令。

中间表示是一种在编译器工作过程中表达源程序的方法，可以看作一种单独的计算机语言，该语言通常比高级语言更接近机器指令，并且比机器指令更容易理解和分析。编译器工作过程中，中间表示能够面向多种类型的前端并表达多种类型的源程序，能够对接多种类型的后端并连接不同类型的硬件设备。使用中间表示可以降低编译器处理不同类型源程序的难度，简化编译优化算法，能够将源程序映射到目标程序的过程分为多个具有明确输入和输出的阶段，方便在不同阶段完成特定的优化，能够更好地支持跨平台编译，提高编译器的可移植性和优化能力。编译器工作流程如图 5.1 所示。

如图 5.1 所示，编译器工作流程中，前端对接多种类型的源语言，将 C、Python、Java 等不同类型的源程序转换为中间表示，优化器对中间表示进行转换和优化，后端将优化后的中间表示转换为 x86、PowerPC、ARM 等不同类型硬件设备使用的机器指令。

传统编译器中间表示的优点是能够满足编译器的基本功能需求，包括类型系统、控制

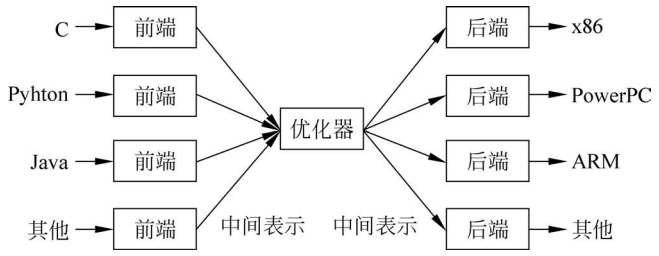

图 5.1　编译器工作流程

流和数据流分析等，能够在函数级别、模块级别和链接级别等不同层次上进行优化，优化过程主要通过各种分析和变换实现，能够解耦编译器的前端和后端，提高编译器的可移植性和可拓展性。缺点是难以适用于与深度学习相关的应用场景和任务。

深度学习任务具有结构化程度高、控制依赖少、处理数据以张量为主、有规律访存等特点。传统编译器处理深度学习任务时，中间表示难以表达和处理张量数据及算子，无法满足深度学习任务的需求。深度学习中间表示是一种面向深度学习应用设计和开发的中间表示，该中间表示在设计时需要充分考虑深度学习任务的特点，以弥补传统编译器中间表示的不足。

5.2　中间表示的分类

根据结构组织进行分类，中间表示可以分为线性中间表示、图中间表示和混合中间表示，不同类型的中间表示各有特点。图中间表示是一种基于图的表达方式，该中间表示能够将源程序转换为图，并且使用图结构表达源程序的操作、数据流和控制流。混合中间表示是一种将线性中间表示和图中间表示相结合的表达方式，该中间表示能够使用不同类型的中间表示表达不同类型的源程序信息。下面将通过一些具体的例子分别介绍线性中间表示、图中间表示和混合中间表示的基本概念与特点。

5.2.1　线性中间表示

线性中间表示是一种基于有序序列的表达方式，该中间表示可以将源程序表达为一系列有序指令。线性中间表示的优点是可以方便地生成目标代码，缺点是缺乏高层次的信息。

三地址代码也被称作四元组，是一种常见的线性中间表示。三地址代码中包含一个操作符和三个地址，其中两个地址表示运算数，一个地址表示运算结果。例如，在三地址代码中，(1+2)×3 可以表示为指令 Add 1,2,t 和 Mul t,3,r，编译器编译过程中，结果变量 t 和 r 会生成表达临时变量的中间表示。

静态单赋值机制是一种编译器中广泛使用的中间表示技术，能够保证程序中的每个变量只被赋值一次，主要应用在线性中间表示中。例如，基于静态单赋值的中间表示中存在许多变量，当一个变量被赋值时，系统不会修改原变量的值，而是生成新变量，新变量是对原变量进行引用的对象。静态单赋值的优点是能够清晰地表达变量之间的数据依赖关系，简化程序对变量的优化。

5.2.2 图中间表示

图中间表示是一种基于图的表达方式，该中间表示可以将编译器工作过程中的信息存储在图结构中，并且使用图结构中的节点和边表达程序中各种类型的关联关系。虽然所有类型的图中间表示都包含节点和边，但是不同类型的图中间表示在抽象层次、图结构、图与底层代码之间的关系等方面各有不同的特点。图中间表示的优点是能够方便地表达控制流和数据流结构，缺点是结构比较复杂和冗余。语法解析树、抽象语法树、有向无环图、控制流图、数据流图、静态单赋值图、程序依赖图等是典型的图中间表示。

语法解析树是一种能够对输入程序进行推导或者语法分析的图中间表示。语法解析树包含解析过程中使用的所有非终止符，能够体现推导或者语法分析的过程，主要用于语法分析。1.3.2 节中使用的 MSELoss 损失函数对应的语法解析树示例如图 5.2 所示，P 表示变量，E 表示语法规则。

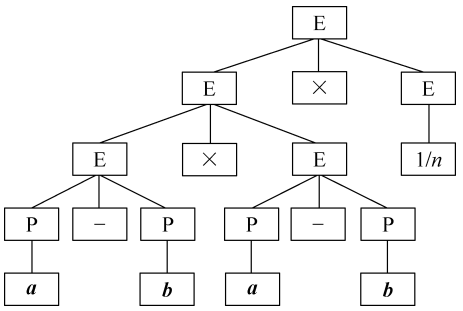

图 5.2 语法解析树示例

抽象语法树是一种更接近源程序的图中间表示，该中间表示在语法解析树的基础上删除了非必要的节点，保留了基本的结构。编译器工作过程中往往需要使用复杂的树遍历算法访问各个节点的数据，但是因为抽象语法树中存在冗余计算，所以抽象语法树不利于完成优化，主要用于类型检查和语义分析。1.3.2 节中使用的 MSELoss 损失函数对应的抽象语法树示例如图 5.3 所示。

有向无环图是一种在抽象语法树基础上改进的图中间表示。有向无环图中的一个节点可以有多个父节点，同一个表达式只存在一个副本，所以构造有向无环图时需要检查并且消除代码中潜在的冗余。与抽象语法树相比，有向无环图使用的内存更少，并且编译器对子树的求值次数更少，能够实现计算结果的重用。1.3.2 节中使用的 MSELoss 损失函数对应的有向无环图示例如图 5.4 所示。

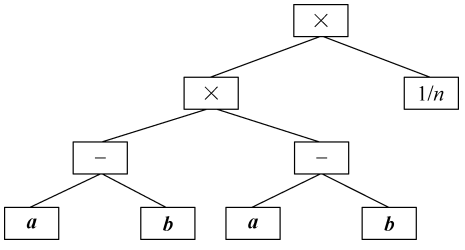

图 5.3 抽象语法树示例

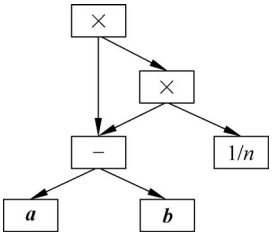

图 5.4 有向无环图示例

控制流图是一种表达程序中各个基本程序块之间控制流的图中间表示，该中间表示是由节点集合和边集合组成的有向图，节点表示基本程序块，基本程序块是指令的序列，边表示不同节点之间的跳转关系。在控制流图中，start 节点和 end 节点是控制流图的入口节点和出口节点，节点表达式中的? 后缀表示条件判断，标记为 true 的边表示条件判断为真时的控制流向，标记为 false 的边表示条件判断为假时的控制流向。控制流图通常用于表达单

个函数并且拥有程序中所有指令的执行顺序。调用图是表达程序中函数之间调用关系的有向图，节点表示函数，边表示调用关系。控制流图可以从线性指令集合或者抽象语法树中生成，能够执行多种优化和转换并且直接进行代码生成。控制流图示例如图 5.5 所示。

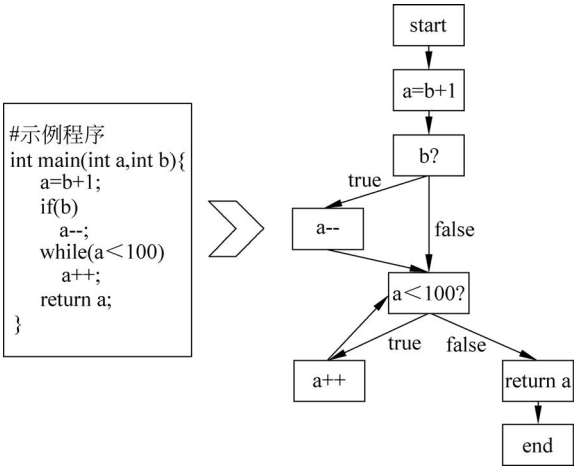

图 5.5　控制流图示例

数据流图是一种表达程序中数据流的图中间表示，该中间表示是由节点集合和边集合组成的有向图，节点表示变量和指令，边表示一个指令的输出数据作为另外一个指令的输入数据。数据流图中的指令获得所有输入数据时才会被执行，指令执行后会产生新的数据并将其传递给数据流指向的指令。与控制流图相比，数据流图没有限定指令集合的执行顺序，不包含整体程序信息，可以看作控制流图的伴随图。当同时访问控制流图和数据流图时，编译器可以有效地执行各种优化，但是该方法会增加执行的时间复杂度。1.3.2 节中使用的 MSELoss 损失函数对应的数据流图示例如图 5.6 所示，$\boldsymbol{r}$ 是指令 $\boldsymbol{r}=\boldsymbol{a}-\boldsymbol{b}$ 的输出和指令 $\boldsymbol{s}=\boldsymbol{r}\times 1/n$ 的输入，$\boldsymbol{s}$ 是指令 $\boldsymbol{s}=\boldsymbol{r}\times 1/n$ 的输出和指令 $\boldsymbol{t}=\boldsymbol{r}\times\boldsymbol{s}$ 的输入，$\boldsymbol{t}$ 是指令 $\boldsymbol{t}=\boldsymbol{r}\times\boldsymbol{s}$ 的输出，箭头指明了数据流向。

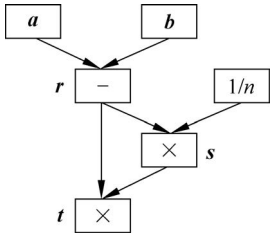

图 5.6　数据流图示例

静态单赋值图是定义使用链在图数据结构上的扩展，节点表示计算操作，有向边将值的使用和值的定义连接，到达节点的边表示计算操作需要的参数，从节点出发的边表示计算结果的传播方向。定义使用链和使用定义链是表达变量数据流信息的数据结构，该数据结构可以用于完成静态优化。变量的定义使用链将该变量的一次定义链接到所有对该变量的使用，变量的使用定义链将该变量的一次使用链接到所有对该变量的定义。

程序依赖图是一种表达程序中控制依赖和数据依赖的图中间表示，该中间表示是由节点集合和边集合组成的有向图，节点包括指令节点、判断节点或者区域节点，边表示节点之间的依赖关系。在程序依赖图中，Entry 节点和 Exit 节点是程序依赖图的入口节点和出口节点，矩形节点表示指令节点，菱形节点表示判断节点，圆形节点表示区域节点。判断节点能够进行判断，标记为 true 的边表示条件判断为真时的控制流向，标记为 false 的边表示条件判断为假时的控制流向。区域节点将具有相同控制依赖关系的节点分组在一起，并且排序到层次结构中。如果区域节点的控制依赖条件得到满足，对应的所有子节点都可以执

行。边有两个不同的子集，分别表示控制依赖和数据依赖，实线表示控制依赖，虚线表示数据依赖。程序依赖图示例如图 5.7 所示。

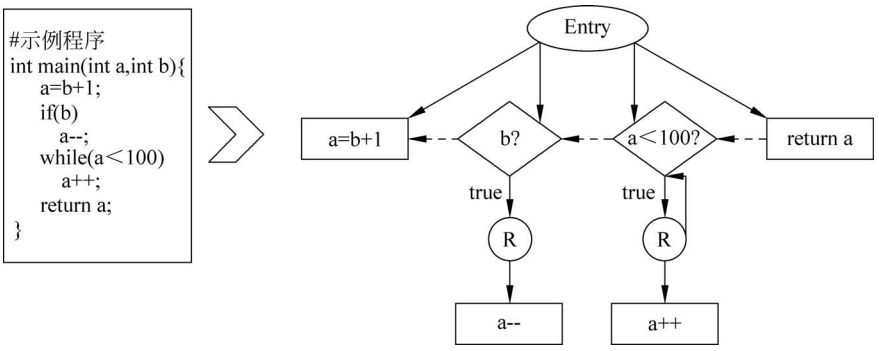

图 5.7　程序依赖图示例

5.2.3　混合中间表示

混合中间表示结合了线性中间表示和图中间表示的要素与特点，获得了两者的优点并且避免了两者的缺点。混合中间表示通常使用线性中间表示表达无循环代码块，使用图中间表示表达无循环代码块之间的控制流。随着技术的不断发展，单一类型的中间表示难以满足各种应用场景和任务的需求，当前主流编译器通常采用多种类型的中间表示。

LLVM IR 是一种典型的混合中间表示，底层是基本块，顶层是控制流图。基本块是一个采用静态单赋值技术设计指令的线性指令序列，控制流图中的每个节点代表一个基本块，基本块之间的边代表控制流从第一个基本块的最后一条指令流向第二个基本块的第一条指令。LLVM IR 结构示例如图 5.8 所示，在 LLVM IR 中，单个基本块是基于静态单赋值的中间表示，所有基本块是基于控制流图的中间表示，基本块之间的连线表示控制流。

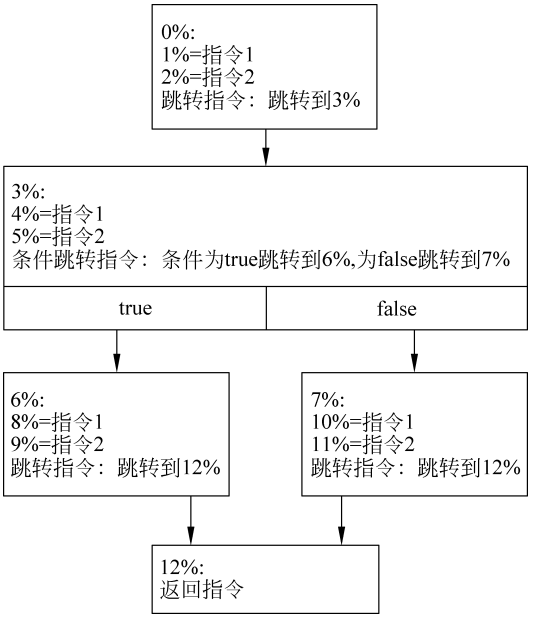

图 5.8　LLVM IR 结构示例

LLVM IR 具有通用、静态、类型安全、平台无关等特点。在硬件支持上，该中间表示能够支持无限数量和任意类型的寄存器。在表示形式上，该中间表示支持文本格式和二进制格式，文本格式方便阅读和编辑，二进制格式方便硬件设备处理和传输，两种格式可以相互转换。

5.3　中间表示的设计

当前，深度学习应用场景类型繁多并且发展迅速，开发人员需要不断结合新的需求和技术，设计和开发具有不同特点的深度学习中间表示。深度学习中间表示的设计需要考虑 Python 等语言描述的模型和目标机器指令间存在的巨大抽象层次差异，实现模型在图层、算子层等多种抽象层次上的建模和优化。不同类型深度学习中间表示的设计理念相似，但是使用的设计方法不同。根据层次数量进行分类，深度学习中间表示的设计方法可以分为单层设计、两层设计和多层设计。无论使用哪种设计方法，一种优秀的深度学习中间表示都应该充分考虑表达和优化、设计层次、类型系统、扩展性、具体实现等方面的问题。

在表达和优化方面，深度学习编译器的输入是不同类型的模型，输出是在不同类型硬件设备上运行的机器指令，其编译过程中涉及大量的硬件无关优化和硬件相关优化。深度学习中间表示需要能够对接不同类型的模型和硬件设备，需要保留张量、算子、数据流和控制流结构进而正确和高效地处理张量数据类型，需要支持各种优化的实现进而提升模型的训练、推理和部署效率。此外，深度学习中间表示需要通过即时编译或者静态编译在不同类型的硬件设备上执行。即时编译能够在运行时把程序转换为机器指令，静态编译能够在执行前把程序转换为机器指令。

在设计层次方面，不同的设计方法可以选择实现不同抽象层次的功能。单层设计可以只实现图层或者算子层的功能，也可以同时实现图层和算子层的功能，前者可能会产生功能不完整等问题，后者可以打破图层和算子层的边界，更好地实现图层和算子层的融合。两层设计可以在不同层级的中间表示中分别实现图层或者算子层的功能，使得不同层级的中间表示分别完成对应的优化，但是低层级中间表示会缺乏高层级中间表示的部分信息。多层设计的中间表示可以支持部分降级，即在高层级中间表示降级为低层级中间表示的过程中，允许部分高层级中间表示不进行降级，使得低层级中间表示保留高层级中间表示的部分信息。

不同设计方法各有优缺点，多层设计的优点是能够灵活地表达更多的源程序信息，开发人员能够根据需求选择在最合适层级的中间表示中实现对应的功能并且高效地执行优化算法。缺点是中间表示的层级越多，设计和开发的工作量越大，不同层级的中间表示之间转换和信息交换越困难，不同层级的中间表示定义的算子粒度不同可能给精度带来影响。高层级中间表示缺乏对低层级中间表示信息的感知能力，无法确定其他层级中间表示完成的优化，开发人员需要决策不同层级中间表示需要完成的优化。因为高层级中间表示降级为低层级中间表示的过程中会产生信息损失，所以低层级中间表示的优化设计有较多的限制。单层中间表示的优点是不需要处理不同层级中间表示之间的转换和信息交换问题，缺点是表达方式的灵活性较低，不同类型的优化集中在同一层级中间表示中，设计和开发优化的难度较大。

在类型系统方面，根据变量类型检查时间进行分类，类型系统可以分为静态类型和动态类型，当前常见的中间表示大部分具有静态类型的特点。按照类型系统的严格程度进行分类，类型系统可以分为强类型和弱类型两种，强类型倾向于不容忍隐式类型转换，弱类型倾向于容忍隐式类型转换。强类型有助于防止类型错误，使得程序更加安全和稳定。此外，为了解决深度学习中的动态形状和量化等重要问题，深度学习中间表示需要支持更多的类型，特别是未知类型和低精度类型。

在扩展性方面，深度学习的应用场景复杂多样，深度学习算子的种类也越来越多。深度学习中间表示需要考虑统一不同层级中间表示的开发方法，提高开发人员的自主性，使得开发人员在遵循基本开发规则的基础上，不需要受限于固定的层数，而是根据具体的需求选择合适的层数和确定各个层级需要完成的功能。为了顺应深度学习的发展趋势，深度学习中间表示需要考虑支持未来可能出现的算子或者提供简单易用的自定义算子接口，支持对类型系统进行扩展。

在具体实现方面，不同的深度学习编译器有不同的设计重点，其使用的深度学习中间表示需要考虑语言特性、生成方式、结构特性等多方面的问题。在深度学习中间表示的具体实现过程中，开发人员不需要解决所有的问题，只需要根据业务场景和任务需求进行选择和取舍。

理解了不同设计方法的基本概念和评价标准，下面将以一些主流的深度学习中间表示为例，介绍单层、两层和多层中间表示设计。

5.3.1 单层中间表示设计

单层中间表示设计是指在同一编译器中设计单层中间表示的设计方法，MindIR、TorchScript IR、HLO IR 等中间表示都采用了该设计方法。下面将以上述三种中间表示为例介绍单层中间表示设计。

1. MindIR

MindSpore 能够同时支持静态计算图模式和动态计算图模式，静态计算图模式是默认执行模式，其使用的中间表示是 MindSpore 中间表示，简称 MindIR。

MindIR 使用的核心数据结构主要包括 AnfNode、ANode、Cnode、ParameterNode、ValueNode 等。AnfNode 主要包括 ANode 和 Cnode，ANode 表示原子表达式，CNode 表示复合表达式。ANode 主要包括 ParameterNode 和 ValueNode，ParameterNode 是参数节点，表示函数的形参；ValueNode 是常数节点，表示常数值、原语函数、普通函数等。

MindIR 是一种函数式风格的图中间表示，该中间表示中的函数定义本身是一个值，能够存储为变量并且进行递归调用，也能够作为参数传到其他函数中或者从其他函数中返回。该中间表示是一种计算流，当前计算的结果是下一次计算的输入。除了完成基本的中间表示功能，MindIR 在表达和优化、设计层次、类型系统、具体实现等方面都有突出特点。

在表达和优化方面，MindIR 支持数据流和控制流结构，能够将控制流转换为高阶函数数据流，支持硬件无关优化、硬件相关优化和部署推理相关优化，能够显著提升 MindSpore 的编译执行能力，支持通过即时编译在不同类型的硬件设备上执行。

在设计层次方面，MindIR 采用单层设计的设计方法，该中间表示能够使用统一的形

式定义模型的逻辑结构和算子属性，消除不同类型模型的差异，对接不同类型的后端，同时实现图层和算子层的功能，核心目的是实现自动微分。

在类型系统方面，MindIR 具有强类型的特点，计算图中的每个节点都有一个具体的类型并且可以在计算过程中表现出最大的性能。深度学习应用算子的处理会耗费大量时间，尽早捕获错误能够减少大量的资源浪费。MindIR 支持函数调用和高阶函数，具有强大的类型推导和形状推导功能，能够尽早发现并且捕获程序中的错误。

在具体实现方面，MindIR 具有纯函数、支持闭包表示等特点，该中间表示能够解决因为使用副作用函数而造成的问题。函数具有副作用是指函数依赖或者影响外部的状态，如果在无法保证程序执行顺序的情况下使用具有副作用的函数，可能会得到错误的结果。纯函数是指返回值只依赖函数形参并且没有副作用的函数。MindIR 能够将具有副作用的中间表示转换为纯函数的中间表示，能够在保持函数式语义不变的同时确保程序执行顺序。闭包是一种编程语言特性，该特性是指代码块和作用域环境的结合可以实现一个函数访问另一个函数作用域中的变量。MindIR 中的代码块是以函数图的形式呈现的，作用域环境可以理解为函数被调用时的上下文环境，能够方便地表达程序中的闭包表示，进而实现不同函数之间变量的相互访问。

2．TorchScript IR

为了保存和加载模型，PyTorch 提供了多种将动态计算图转换为静态计算图的图捕获解决方案。TorchScript 机制是 PyTorch 提供的图捕获解决方案之一，TorchScript IR 是 TorchScript 机制保存模型的中间表示，该中间表示能够从 Python 进程中保存并且加载到没有 Python 依赖的进程中。

TorchScript IR 使用的核心数据结构主要包括 Graph、Block、Node、Value、Pass 等。Graph 表示函数，主要包括 Node、Block、Value 等数据结构。Block 是一组 Node 的集合，主要负责管理 Node。Node 表示算子，Value 表示算子的输入和输出，Pass 表示各种类型的优化。

TorchScript IR 是一种基于有向无环图的中间表示，该中间表示使得深度学习编译器能够方便地分析算子之间的依赖关系。除了完成基本的中间表示功能，TorchScript IR 在表达和优化、类型系统、具体实现等方面都有突出特点。

在表达和优化方面，TorchScript IR 支持数据流和控制流结构，支持硬件无关优化和硬件相关优化，能够提高计算图的执行效率并且减少内存占用。PyTorch 默认使用动态计算图模式，该模式下的动态计算图结构在提高开发效率和降低开发难度的同时也带来了许多问题。例如，动态计算图非固定的网络结构会给网络结构分析和优化带来困难，多数参数都能以张量形式传输使得资源分配变得复杂。因为动态计算图是由 Python 程序构造的，所以部署时需要依赖 Python 环境，该特性增加了部署难度并且降低了保密性。TorchScript IR 能够把动态计算图转换为静态计算图，优点是静态计算图具有相对固定的图结构，能够解决动态计算图结构上存在的问题。缺点是静态计算图结构复杂，修改和增加优化的难度较高，只能表示前向计算图而不能处理反向计算图和动态形状张量。此外，TorchScript IR 支持通过即时编译在不同类型的硬件设备上执行。

在类型系统方面，TorchScript IR 是 Python 语言的子集，在变量声明机制方面，TorchScript IR 具有静态类型的特点，其变量在使用前需要声明数据类型，运行时不能变

换数据类型，该特性使得 TorchScript IR 能够在部分高性能环境中运行。此外，TorchScript IR 只支持部分用于表达模型的特定数据类型，无法支持 typing 模块的所有功能和数据类型。typing 模块是 Python 的一个标准库，主要为类型注解提供运行时支持，类型注解是一种在程序中指定变量、函数参数和返回值类型的方法。

在具体实现方面，TorchScript IR 能够通过基于追踪方式或者基于分析方式生成。在通过基于追踪方式生成 TorchScript IR 的过程中，每读取一个张量或者执行一个算子，深度学习编译器会向计算图中增加一个对应节点。所有操作完成后，深度学习编译器能够构造一张包含所有读取和执行信息的静态计算图。在通过基于分析方式生成 TorchScript IR 的过程中，TorchScript 实现了一个完整的编译器，能够使用@torch.jit.script 装饰器解析模型程序并且构造对应的抽象语法树，然后经过一系列分析过程生成静态计算图。基于追踪方式和基于分析方式各有优缺点，开发人员可以将两种方式结合使用以同时获得两种方式的优点。

3. HLO IR

TensorFlow 能够同时支持静态计算图模式和动态计算图模式，其中静态计算图模式更为人所熟知。为了提高模型的执行效率，TensorFlow 使用了深度学习编译器 XLA，该编译器能够将模型转换为 HLO IR。

HLO IR 使用的核心数据结构主要包括 HLOModule、HLOComputation、HLOInstrution 等。HLOModule 表示整个模型，可以包含多个 HLOComputation。HLOComputation 表示模型中的函数，可以包含多个 HLOInstrution。HLOInstrution 表示函数中的指令。

HLO IR 是一种函数式风格的图中间表示，除了完成基本的中间表示功能，其在表达和优化、类型系统、扩展性、具体实现等方面都有突出特点。

在表达和优化方面，HLO IR 支持数据流和控制流结构，支持硬件无关优化和硬件相关优化。HLO IR 能够表达更加细粒度的图，使用多个基础算子节点表示计算图的一个复杂算子节点，减少对自定义算子的依赖。计算图中的许多算子都是逐元素的计算，HLO IR 能够将多个连续的基础算子节点融合成一个复杂算子节点，减少算子中间结果的内存占用和内核的启动次数，进而提高计算图的执行效率。此外，HLO IR 支持通过即时编译和静态编译在不同类型的硬件设备上执行。

在类型系统方面，HLO IR 是一种纯静态形状语义的中间表示，不支持动态形状张量。XLA 编译器工作过程中，HLO IR 中的形状表达会被静态化，所有的形状计算都会被固化为编译时常量并且保留在编译结果中。

在扩展性方面，HLO IR 具有可移植性强的特点，能够降低 TensorFlow 程序适配不同类型硬件设备的难度。应用 XLA 深度学习编译器之前，TensorFlow 程序在新硬件设备上运行时需要重新运行所有算子，该过程的工作量巨大。应用 XLA 深度学习编译器之后，XLA 深度学习编译器能够自动地将 TensorFlow 计算图中的复杂算子拆解为原语算子，该特性使得大部分 TensorFlow 程序都能够以未经修改的方式在新的硬件设备上运行。

在具体实现方面，HLO IR 使用分层嵌套的结构，能够方便地表达图与图之间的调用关系，并且有助于完成计算图融合。在 HLO IR 中，一个 HLOModule 只能有一个入口函数，

其他函数都需要通过入口函数进行调用。一个 HLOComputation 只能有一个根指令，根指令的输出就是函数的输出。所有指令都能够调用函数，其执行顺序取决于指令之间的依赖关系。当进行计算图融合时，编译器需要将多个需要融合的指令添加到一个融合 HLOComputation 中，使用一个融合 HLOInstrution 代替被融合的指令。当进行代码生成时，编译器需要根据融合 HLOInstrution 生成对应的代码。HLO IR 分层嵌套结构如图 5.9 所示。

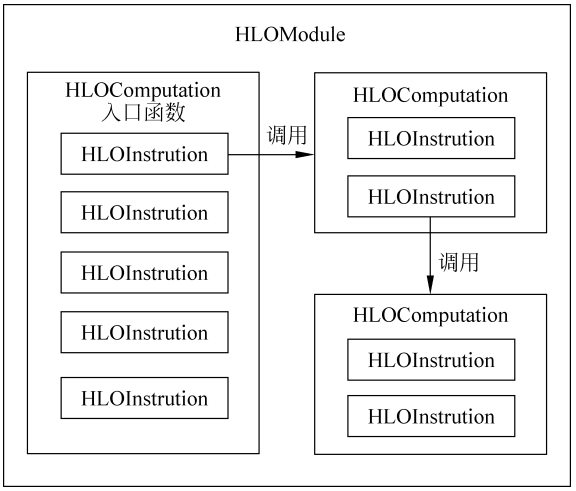

图 5.9　HLO IR 分层嵌套结构

5.3.2　两层中间表示设计

两层中间表示设计是指在同一编译器中设计两层中间表示的设计方法。TVM IR、oneDNN Graph IR 等中间表示都采用了该设计方法。下面将以上述两种中间表示为例介绍两层中间表示设计。

1. TVM IR

TVM 编译器是一种采用两层中间表示设计的端到端深度学习编译器，其发展早期受 Halide 编译器影响较大，低层级中间表示使用的是具有 Halide 特性的中间表示，高层级中间表示最初使用 NVVM IR。Halide 是一种使用 C++作为宿主语言进行图像处理的领域特定语言，其基本思想是分离计算与调度，调度能够决定计算的执行顺序、数据的存放和中间表示完成的优化。基于 Halide 特性设计开发的中间表示能够实现对嵌套循环的优化，能够针对不同类型的硬件设备选择不同的调度策略，能够方便地完成不同类型的优化并且具有较强的扩展性。随着技术的不断发展，当前 TVM 编译器的高层级中间表示使用 Relay IR，低层级中间表示使用 TIR。TVM 编译器工作过程中，TVM 编译器首先将模型转换为 Relay IR 并且完成硬件无关优化，然后将 Relay IR 降级为张量表达式并且提供自定义算子方法，之后将张量表达式降级为 TIR 并且完成硬件相关优化，最后将 TIR 转换为机器指令。

IRModule 是 Relay IR 和 TIR 的核心数据结构，也是模型构建和中间表示转换的基础，不同模型最终都会被封装到 IRModule 数据结构中进行编译。此外，Relay IR 和 TIR

共用一套中间表示基础设施，其中的元素主要由 Type 类和 Expr 类两个基类派生而来，所以 Type 类和 Expr 类是 TVM IR 设计的两大核心。

Type 类主要用于表达各种数据类型，包括 bool、int8、float32 等基础数据类型和张量、函数等高级数据类型。Relay IR 主要使用张量、函数等高级数据类型，TIR 主要使用指针、缓冲区等数据类型。Expr 类主要用于表达对各种数据类型的处理，包括控制结构、分支信息等。Expr 类包括 RelayExpr 和 PrimExpr 两个子类，Relay IR 主要使用 RelayExpr 子类表达数据流和控制流结构，包括 if 表达式、Let 表达式等。TIR 主要使用 PrimExpr 子类完成低层级的代码优化和整数类型分析检查，包括部分基本数据类型的常量表示、基本运算等。

在 Relay IR 降级为 TIR 的过程中，TVM 编译器采用了 Halide 的思想，通过计算和调度分离的方式对 Relay IR 中的算子进行生成和优化，张量表达式是该过程中的一个重要概念。张量表达式是一种使用纯函数语言描述张量计算的表达形式，需要和调度一起使用，每个张量表达式都能够接收输入张量并且生成输出张量。调度主要分为循环级调度和多线程并行优化，循环级调度主要包括循环切分、融合、重排、分块、展开等，多线程并行优化主要包括向量化、线程绑定、并行化、张量化等。针对不同硬件设备的体系架构特点，同一个算子在不同的硬件设备上具有不同的调度策略。算子策略是计算和调度策略的组合使用，能够实现某个算子在某种平台上的算子生成。

因为张量表达式并不能直接被编译为机器指令，所以张量表达式需要继续降级为一种能够描述计算过程的中间表示。TIR 是 TVM 编译器中一种接近硬件设备的中间表示，该中间表示能够表达变量声明、初始化、计算、函数调用、控制流等。TIR 采用抽象语法树数据结构，因此通过遍历抽象语法树完成对应的代码生成工作，TVM 编译器能够将 TIR 转换为不同类型硬件设备的机器指令，包括 LLVM IR、CUDA 程序等。TIR 代码生成流程示例如图 5.10 所示，stmt 表示指令语句，expr 表示对应的表达式。

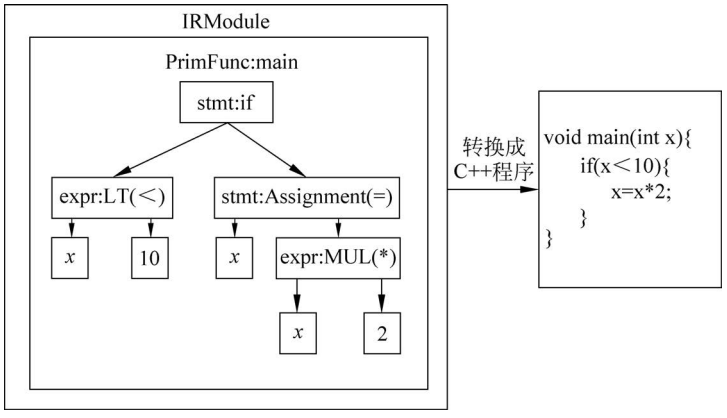

图 5.10 TIR 代码生成流程示例

自动张量化是深度学习领域中与硬件相关的一个热点问题。随着硬件设备的发展，越来越多的深度学习加速器具有矩阵乘等深度学习特定算子的快速计算指令，该类指令的使用往往需要开发人员手工编写，其在程序优化与拓展方面成本较高。自动张量化的实现原理包括自底向上和自顶向下，主流实现是在编译器内部中间表示对张量化程序进行建模，

以解决当前硬件后端提供的多种张量化指令在程序中的表达和优化问题。

自底向上实现自动张量化是指从嵌套的标量计算循环中找到张量模式以生成张量化程序，该方法与高级编程语言中编译的自动向量化优化类似，不同之处在于使用该方法完成深度学习应用的编译时，输入的张量数据被降级为更细粒度的嵌套标量循环计算。自底向上实现自动张量化的缺点是会产生从张量中间表示到标量中间表示，再到张量中间表示的转换，该转换过程会产生不必要的信息损失。以卷积算子为例，自底向上实现自动张量化示例如图 5.11 所示。该过程中，首先高层级中间表示会被分解为由标量运算构成的多层循环，然后编译器需要改变标量运算过程以进行调度优化，最后编译器需要生成后端机器指令。

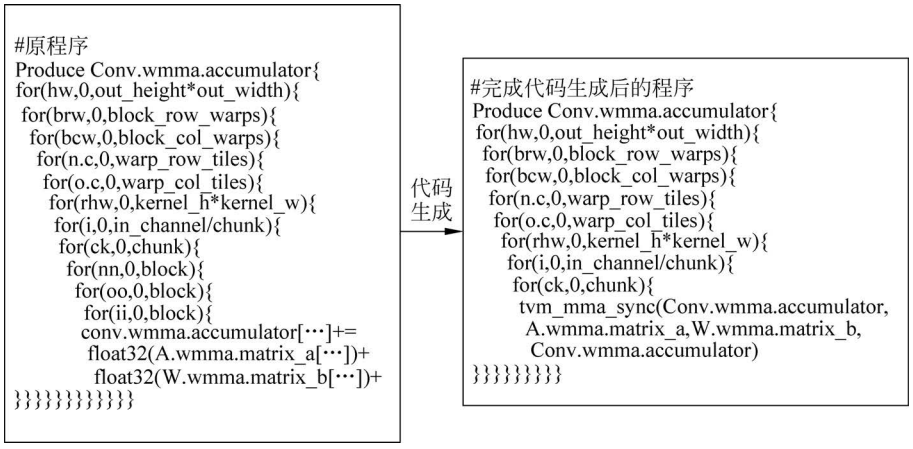

图 5.11　自底向上实现自动张量化示例

自顶向下实现自动张量化是指通过张量计算问题的逐步解耦，对任务进行划分以映射到张量化原语。TIR 使用自顶向下实现自动张量化，其核心设计理念是在嵌套循环内构建一个 Block 结构以分离张量计算负载和外层循环，进而独自进行转换和优化，Block 结构支持嵌套表达复杂的张量计算模式。以卷积算子为例，自顶向下实现自动张量化示例如图 5.12 所示。该过程中的中间表示具有算子的算法信息，能够直观地进行数据分块和优化，通过简单的循环分解实现高效的代码生成。

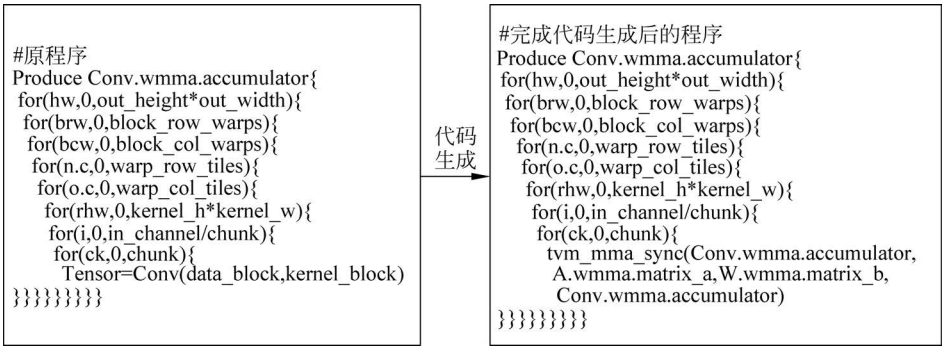

图 5.12　自顶向下实现自动张量化示例

TVM IR 主要包括 Relay IR 和 TIR，Relay IR 是一种同时基于有向无环图、函数式风格和 Let 绑定的图中间表示，TIR 采用抽象语法树的数据结构。Let 绑定是一种解决语义

歧义的方法，使用 Let 绑定的编译器能够构建变量与所有计算结果之间的映射，能够通过查找映射快速查找计算结果。除了完成基本的中间表示功能，TVM IR 在表达和优化、设计层次、类型系统、扩展性等方面都有突出特点。

在表达和优化方面，Relay IR 支持数据流和控制流结构，并且支持数据流和函数式风格的混合编程，能够兼容不同类型的模型，主要完成硬件无关优化。该中间表示具有明确规定的语义，能够表达通用程序和复杂模型，能够快速地通过查询获得结果，但是表达方式和优化方式比较复杂，其使用的函数式风格计算图不够直观。TIR 支持数据流和控制流结构，主要完成硬件相关优化，能够实现对张量化程序的表达和优化，支持通过即时编译和静态编译在不同类型的硬件设备上执行。

在设计层次方面，TVM IR 采用两层设计的设计方法，该设计方法的优点是 Relay IR 和 TIR 分别负责实现图层和算子层的功能，进而降低开发人员设计和开发优化的难度。缺点是 Relay IR 降级为 TIR 的过程中会产生信息损失，使得 TIR 无法获取完整的模型信息，编译器难以确定不同层级中间表示的边界。

在类型系统方面，Relay IR 具有静态类型的特点，其支持的数据类型与 Type 类相关。此外，Relay IR 支持使用代数数据类型和 Lambda 表达式，能够表达更加复杂的模型，但是不支持动态形状张量。代数数据类型是一种复合类型，该类型可以由其他类型组合而成。Lambda 表达式是匿名内部类的一种替代，该特性允许将函数作为函数参数传入，能够解决匿名内部类因为使用一次就销毁特性而造成的语法冗长问题。

在扩展性方面，Relay IR 在设计上是可扩展的并且容易开发新的大规模程序变换和优化。此外，TVM Unify 统一多层抽象是 TVM 编译器未来发展的重要方向，该技术的关键是统一计算图、张量程序、算子库和运行环境、硬件专用指令等四类抽象，能够打破不同抽象之间的隔阂，使得不同抽象之间能够相互交互和联合优化。TVM Unify 统一多层抽象主要包括 Relax IR、自动张量化机制、TVM FFI 机制和 Tensor IR，Relax IR 是 Relay IR 的迭代版本，该中间表示支持动态类型的张量；自动张量化机制能够实现硬件指令声明和张量程序对接；TVM FFI 机制能够灵活地引入任意的算子库和运行库函数，并且能够在各个编译模块和自定义模块里面相互调用；Tensor IR 负责张量级别程序和硬件张量指令的整合。

2. OneDNN Graph IR

oneDNN Graph 编译器是一种采用两层中间表示设计的深度学习计算图编译模块，该编译器特别适合完成与 CPU 相关的优化。oneDNN Graph 编译器的高层级中间表示使用 Graph IR，主要用于表达计算图；低层级中间表示使用 Tensor IR，主要用于表达算子内部的具体计算。Graph IR 和 Tensor IR 是 oneDNN Graph 编译器使用的中间表示名称，而不是中间表示的类型。oneDNN Graph 编译器工作过程中，oneDNN Graph 编译器首先将 oneDNN 计算子图转换为 Graph IR 并且完成硬件无关优化，然后将 Graph 中间表示降级为 Tensor IR 并且完成硬件相关优化，最后将 Tensor IR 转换为机器指令。

Graph IR 主要用于表达计算图算子之间的连接关系，该中间表示使用的核心数据结构主要包括 Graph、Op、Tensor 和 Pass。Graph 表示整张计算图，Op 表示计算图中的算子节点，Tensor 表示计算图中的张量节点，Pass 表示各种类型的优化。张量形状主要包括两部分，一部分是逻辑上的形状，即想象的张量形状，该形状可以在内存重新排布时提升性能；另一部分是在内存中的形状，即张量的实际形状。

Tensor IR 主要用于描述具体运算，该中间表示使用的核心数据结构主要包括 expr、stmt、IR function 和 IR module。expr 节点表示计算结果，主要包括 float16、bfloat16、signed8、unsigned8 等数据类型和加法、减法等表达式。expr 节点具有支持嵌套定义、可扩展、有返回值等特点，通过扩展 expr 基类可以实现各种类型运算表达式的定义，通过指针可以指向其他 expr 节点，但是无法指向 stmt 节点。stmt 节点表示指令语句，主要包括控制流、赋值语句，以及每一句完整的程序代码等。stmt 节点没有返回值但是能够引用其他的 expr 节点和 stmt 节点。IR function 表示函数，是由 expr 节点和 stmt 节点组成的集合。函数之间可以相互调用，也可以组成 IR module，用于表示完整的模型。

oneDNN Graph 编译器中间表示主要包括 Graph IR 和 Tensor IR，两者在设计理念、功能特点等方面分别与 TVM 编译器的 Relay IR 和 TIR 相似。除了完成基本的中间表示功能，TVM IR 在表达和优化方面有突出特点。

在表达和优化方面，Graph IR 支持数据流结构，主要完成硬件无关优化。除了常见的公共子表达式消除、死代码消除、常量折叠等优化，Graph IR 还支持低精度转换、常量权重预处理、布局转换、细粒度融合、粗粒度融合等深度学习领域相关优化。低精度转换能够将模型计算图和密集型计算中的计算过程转换为低精度的计算过程，减少计算过程中需要使用的计算资源和内存带宽。常量权重预处理能够识别并且预处理常量权重，确保常量权重在编译过程中只需要执行一次，减少常量权重的处理次数并且提升编译效率。布局转换能够为每个算子选择合适的输入张量布局，当输入张量布局与期望布局不同时，可以在内存中对算子进行重新排序，进而提升算子的执行效率。细粒度融合能够对多个可融合算子进行融合，粗粒度融合能对多个已经融合过的算子进行融合。

TIR 支持数据流和控制流结构，主要完成硬件相关优化，支持通过即时编译在不同类型的硬件设备上执行。TIR 支持完成张量形状优化和内存缓冲区优化等张量相关优化。张量形状优化能够尽可能地减小临时张量的形状，进而降低内存占用。内存缓冲区优化能够尽可能地重用临时张量的内存缓冲区，降低临时缓冲区大小并且提高临时缓冲区的使用局部性。

5.3.3 多层中间表示设计

多层中间表示设计是指在同一编译器中设计三层及三层以上中间表示的设计方法。MLIR(multi-level intermediate representation)是一种典型的多层中间表示设计方法，MLIR 编译器是基于 MLIR 框架设计、开发的编译器，该编译器使用的中间表示是 MLIR 中间表示。下面将以 MLIR 编译器为例介绍多层中间表示设计。

MLIR 编译器和 MLIR 中间表示并不是具体的编译器或者中间表示，而是两种类别。所有基于 MLIR 框架设计、开发的编译器都是 MLIR 编译器，MLIR 编译器中的所有中间表示都是 MLIR 中间表示。

MLIR 中间表示的核心概念主要包括 Dialect、Operation、Attribute、Type、Interface、Constraint、Trait、Region、Block 等。Dialect 在特定的命名空间下为抽象提供了分组机制，能够将所有的抽象放在同一个命名空间中，分别为每种抽象定义对应的 MLIR 表达式并且绑定对应的 Operation，进而生成 MLIR 中间表示。Operation 是 Dialect 中具有特定语意的抽象和计算的核心单元，能够表示指令、函数、模块等。Attribute 是属性信息，Type 是类型信息，Interface 是标准化的接口，Constraint 和 Trait 是 Operation 的限制信息和特征信息，

Region 和 Block 是 Operation 的作用域，两者支持嵌套使用。Dialect 的结构如图 5.13 所示，Operation 主要包括 Attribute、Type、Interface、Constraint、Trait、Region 和 Block，Dialect 主要包括 Operation、Attribute、Type 和 Interface。

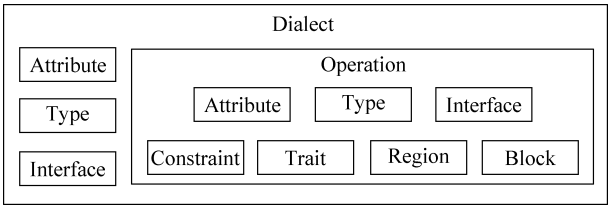

图 5.13 Dialect 的结构

MLIR 编译器的工作流程主要分为前端接入、转换和优化、后端代码生成三个阶段。在前端接入阶段，MLIR 编译器需要将高级语言抽象语法树、ONNX、TensorFlow 等格式的模型文件转换为 MLIR 中间表示。在转换和优化阶段，MLIR 编译器需要完成多层 MLIR 中间表示之间的转换和每层 MLIR 中间表示对应的优化，即 MLIR 编译器提供了多种内置的低层级中间表示，开发人员可以使用内置的 Dialect 构建自定义编译流程，也可以根据业务需求和应用场景构建不同层次和功能的 Dialect。在后端代码生成阶段，MLIR 编译器需要通过即时编译或者使用 LLVM 编译器的方式生成硬件设备机器指令。当前，大部分 MLIR 编译器都选择使用 LLVM 编译器生成机器指令。MLIR 中间表示部分 Dialect 构建编译流程如图 5.14 所示，负载和结构分别表示 Dialect 负责表达和优化的类型，张量和内存分别表示 Dialect 操作的对象，实线表示 Dialect 之间的转换，虚线表示 Dialect 之间的转换需要大

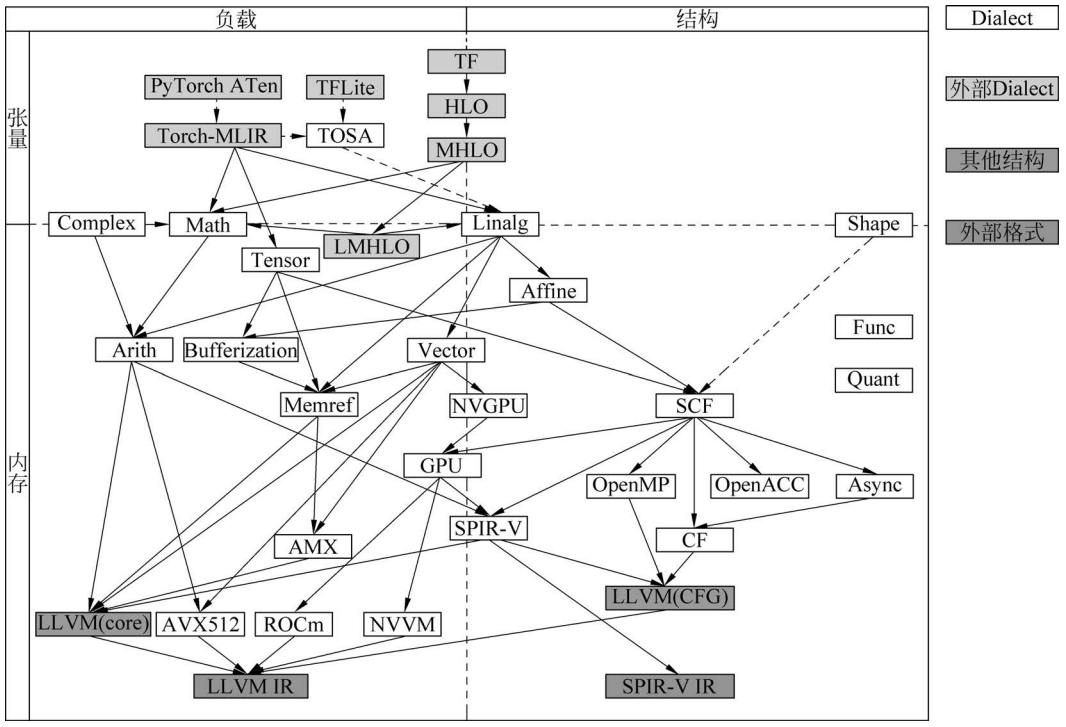

图 5.14 MLIR 中间表示部分 Dialect 构建编译流程

量的优化才能实现，部分同时处于多个区域的 Dialect 表示该 Dialect 可以表达和优化多种类型或者操作多种对象。

如图 5.14 所示，MLIR 中间表示提供了大量的 Dialect 供开发人员使用。PyTorch ATen Dialect、TFLite Dialect、TF Dialect 等的主要功能是导入模型程序。TOSA Dialect、MHLO Dialect 等是 MLIR 编译器的较高层级中间表示，主要功能是使用一种统一的 Dialect 完整地表达模型，便于作为后续降级过程的输入。Linalg Dialect 是 MLIR 编译器的中间层级中间表示，其本质是完美嵌套循环，主要功能是将较高层级中间表示的算子进一步细分，描述线性代数的相关结构，实现算子融合和分块优化等。Affine Dialect 借鉴了多面体编译中循环索引表达，常用于描述访存模式。Vector Dialect 是 Linalg Dialect 的主要降级对象，其结构设计与 Lianlg Dialect 相似，主要功能是实现结构化代码生成。Arith Dialect、Math Dialect 等通常作为 Linalg Dialect 和 Vector Dialect 等的负载操作，其主要功能是表达整数和浮点数计算，支持张量、向量、标量等各种抽象层级。SCF Dialect 和 CF Dialect 的主要功能是描述 if、for、while 等结构化控制流操作，其使用过程中需要标明边界，进而简化分析和变换过程。Func Dialect 的主要功能是对模型中的函数对象进行建模，Quant Dialect 的主要功能是对模型中的量化类型进行建模。GPU Dialect 的主要功能是对 GPU 的硬件特性进行建模，包括模型中 GPU 算子的计算和访存模式等。LLVM Dialect、SPIR-V Dialect 等是 MLIR 编译器的较低层级中间表示，其主要功能是将 MLIR 中间表示转换到 LLVM 编译器等外部工具链，以进一步完成代码生成。LLVM IR 和 SPIR-V IR 是常见的传统编译器中间表示。

MLIR 是一种典型的多层中间表示设计方法，基于该设计方法开发的中间表示除了完成基本的中间表示功能，其在表达和优化、设计层次、类型系统、扩展性、具体实现等方面都有突出特点。

在表达和优化方面，MLIR 中间表示支持数据流和控制流结构，支持硬件无关优化和硬件相关优化，支持通过即时编译或者使用 LLVM 编译器的方式在不同类型的硬件设备上执行。

在设计层次方面，MLIR 采用多层设计的设计方法，该设计方法的优点是 MLIR 编译器中各个层级的中间表示都使用了相同的设计规则，可以轻松地进行转换和实现各种类型的优化。因为各个层级的中间表示不再相互独立，所以单层中间表示的设计不需要考虑所有的优化，而是可以将不同类型的优化放在最合适的层级。当某一层级中间表示需要完成其他层级中间表示的优化时，可以先将当前层级中间表示转换为其他层级中间表示，再完成相关优化。缺点是 MLIR 中间表示的可定制性会带来内部碎片化的风险，需要更加灵活的设计来支持抽象级别的可扩展性，不同层级的中间表示设计可能存在重复问题并且造成较大的开销。

MLIR 中间表示采用渐进式降级的设计理念并且支持部分降级。许多编译器都采用多层级中间表示的设计理念并且引入了多个固定的抽象级别，但是固定降级的实现方式比较僵化。在 MLIR 中间表示中，源程序能够以较小的步幅依次经过多个抽象级别，从较高层级中间表示降低到较低层级中间表示。此外，在从较高层级中间表示降低到较低层级中间表示的过程中，MLIR 编译器允许只对部分中间表示降级，剩下的中间表示仍然可以保持较高层级。该特性使得 MLIR 编译器能够保留部分必要的高层级中间表示信息，提升编译器

工作的流畅度和效率。

MLIR 中间表示采用源位置跟踪和可追溯性的设计理念。在结构复杂的编译器中，开发人员往往很难获得最终层级中间表示构造的完整过程。MLIR 中间表示能够准确地将较高层级中间表示的信息传递给较低层级中间表示，实现安全且可回溯的编译过程，解决复杂编译器中常见的缺乏透明性的问题。

在类型系统方面，MLIR 中间表示采用复杂、全面的类型系统，该类型系统能够支持多种内置类型和量化类型，允许开发人员创建自定义类型并且规定语义，具有可定制的特点。此外，MLIR 中间表示不仅提供了预设的操作与抽象类型，而且允许开发人员自由地设计中间表示，有针对性地解决对应领域的问题。MLIR 类型系统如图 5.15 所示。

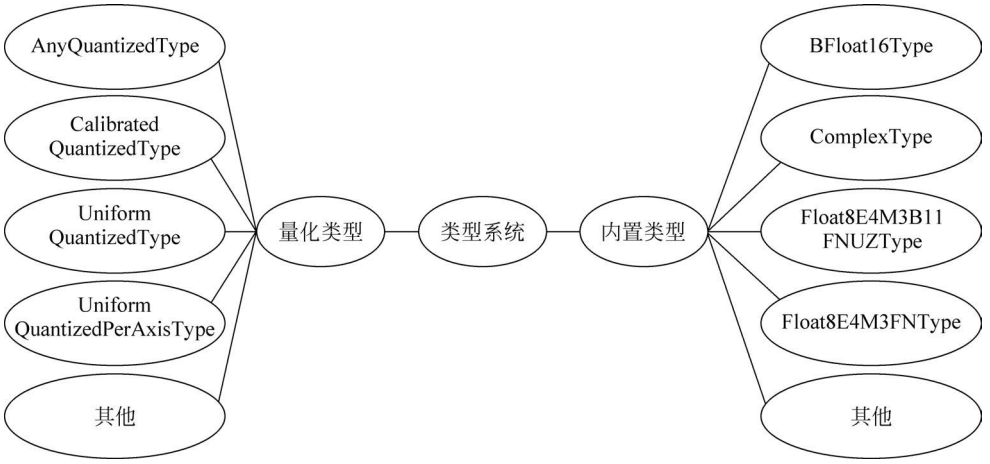

图 5.15　MLIR 类型系统

在扩展性方面，MLIR 中间表示支持嵌套结构，可扩展性极强。开发人员可以任意对 Dialect、Operation、Type、Interface、Trait 进行扩展，构建满足需求的 MLIR 中间表示。MLIR 中间表示的嵌套结构如图 5.16 所示，一个 Region 可以包含一系列 Block，一个 Block 可以包含一系列 Operation，一个 Operation 可以包含一系列 Block 和 Region。

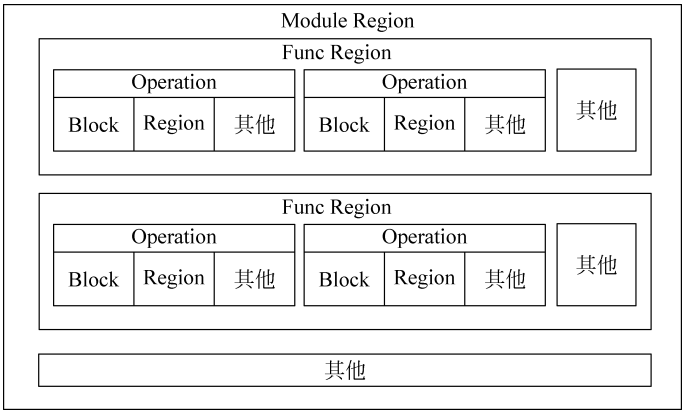

图 5.16　MLIR 中间表示的嵌套结构

在具体实现方面，MLIR 中间表示支持基于静态单赋值的控制流图。MLIR 中间表示

与 LLVM IR 有相似的结构，但是当控制流图中某一基本块中的变量来自多个基本块时，LLVM IR 需要使用 PHI 节点判断当前基本块之前执行的基本块并且确定变量值，而 MLIR 中间表示的终止符语义定义了当前基本块中的变量值，所以不需要使用 PHI 节点。

以上是部分常见的深度学习中间表示设计方法及其典型应用，从上述分析可以看出，深度学习中间表示是一种贯穿深度学习编译器整个工作流程的重要工具，不同类型的深度学习中间表示各有侧重点，不同类型的设计方法各有优缺点。开发人员在设计和开发深度学习中间表示的过程中，不需要拘泥于单一的设计理念和设计方法，而是需要根据业务场景和任务需求进行结合与取舍。