第5章

方舟编译器IR的设计与实现

方舟编译器设计了自己的IR体系，将其称为Maple IR，简称MIR。MIR是多层IR设计，其体现了目前编译器IR设计的发展方向及思路。本章将就Maple IR的设计、结构、实现等方面进行分析和介绍。

5.1 Maple IR设计的起源与思想

根据方舟编译器技术沙龙所披露的PPT内容，Maple IR的设计起源于Fred Chow大一统思想：A standard for universal IR that enables target-independent program binary distribution and is usable internally by all compilers may sound idealistic, but it is a good cause that holds promise for the entire computing industry。

其中，还提到了Fred Chow的一篇关键的论文 *The increasing significance of intermediate representations in compilers* (https://queue.acm.org/detail.cfm?id=2544374)。

Fred Chow 在该论文中提出了一个支持多语言和多目标平台的编译系统，这个系统支持多语言和多目标平台，其架构如图 5.1 所示。

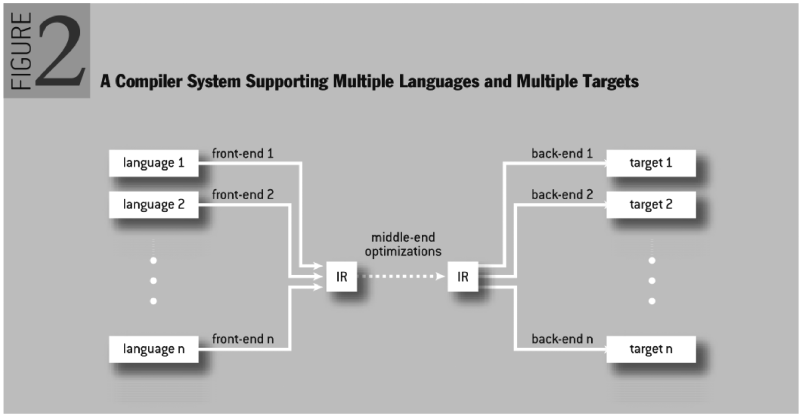

图 5.1　支持多种语言和多目标平台的编译系统

（图源：Fred Chow，*The increasing significance of intermediate representations in compilers*，P2）

现在越来越多的多语言和多目标平台采用类似的架构，比较著名的如 LLVM、Open64 等。所以，方舟编译器的 Maple IR 起源于 Fred Chow 的这个思想，也是和其支持多语言多目标平台的定位完全结合在一起的。

另外，Maple IR 的设计采用了多层 IR 设计。多层 IR 设计也是近年来编译器 IR 设计的一个重要发展方向。在方舟编译器之前，Open64 就采用了多层 IR 设计。不得不提，Fred Chow 在 Open64 设计中，也是核心人物。

多层 IR 设计有着诸多优点，将其简单归纳可以分为以下

几点：可以提供更多的源程序信息；IR 表达上更加灵活，更方便优化；使得优化算法更加高效；可以将优化算法的负面影响降到最低。但是，IR 设计也有缺点：底层 IR 的优化器将面临更多的可能，增加了特定语义的识别难度。总体而言，多层 IR 的设计还是利大于弊的，所以多层 IR 设计也逐渐成为一种趋势。

方舟编译器的多层 IR 设计，其思想可以简单地总结为 3 点。第一，高层 IR 更接近于源程序，包含了更多的程序信息；底层 IR 更接近于目标平台的机器指令，甚至有的时候和机器指令是一对一的关系。第二，高层 IR 保留了程序语言的层次结构，和目标机器平台无关；底层 IR 更加扁平化，依赖具体的目标平台。第三，越高层次的 IR，其所支持的 Opcodes 越多；最底层的 IR，其所支持的 Opcodes 和目标处理器的操作是一对一的。

但是，方舟编译器目前的多层 IR 设计还存在一个比较重要的缺陷，那就是并没有明确地提出多层 IR 之间的分层和衔接。Open64 的多层 IR 体系，称为 WHIRL IR，其有明确的分层和各层之间的衔接，甚至包含了每层要做的操作。WHIRL IR 用一张图表达了这些信息，如图 5.2 所示。

这种统一地按照层次去介绍 IR 的分层及体系的内容，在方舟编译器之中还是个空缺。从目前的代码中，也无法看出明确的分层。这一点也是方舟编译器在后续发展中必须要解决的问题，否则无法理清楚多层 IR 的设计体系，不利于学习和社区发展。

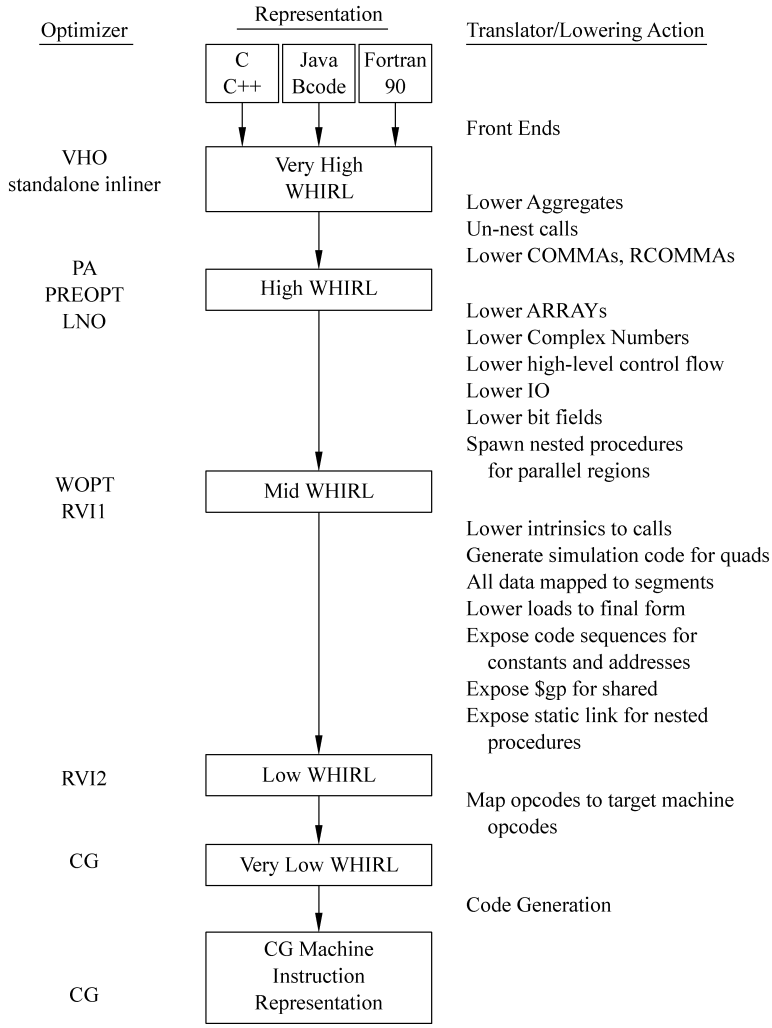

图 5.2　Open64 的 IR 分层

（图源：*Open64 Compiler WHIRL Intermediate Representation*，P7）

5.2 Maple IR 的结构

从 Maple IR 的设计起源与思想中可以找到 Maple IR 的设计精髓，在理解其设计精髓之后，对其更进一步地了解则需要深入分析其具体结构及其代码实现。本部分内容将对 Maple IR 的结构进行简要分析。

理解 Maple IR 的结构，需要从 Maple IR 在方舟编译器中的位置入手。Maple IR 在方舟编译器的架构图中位于核心位置，上接方舟 IR 转换器（也就是我们所讲的传统意义上的编译器前端），向下面向语言特性实现、优化及代码生成（即我们传统意义上讲的编译器的中端优化和后端）。其结构和过程如图 5.3 所示，图中用红色圆圈圈出了方舟 IR，这里的方舟 IR 和 Maple IR 指的是同样的内容，只是称呼不同，后续不再区分。

方舟编译器的 Maple IR 文档，并没有专门介绍 IR 结构的部分，在文档中有一个相近的部分叫 Program Representation。这部分描述了 Maple IR 的表达方式。按照文档的描述，Maple IR 采用类似 C 语言的形式（并不遵循 C 的语法），将 IR 分为声明语句（declaration statements）和执行语句（executable statements）两部分，前者表达符号表信息，后者表达要执行的具体程序代码。

在结构方面，结构的最顶层，每个 Maple IR 文件对应一个 CU（Compilation Unit，编译单元），每个 Maple IR 文件由全局的声明组成。这些声明内部是函数，或者叫 PUs（Program Units）。在

PUs 内部是局部范围的声明和紧随其后的函数执行代码。而 Maple 的 IR 中的可执行节点又分为 Leaf nodes(叶节点)、Expression nodes(表达式节点)和 Statement nodes(语句节点)。具体结构如图 5.4 所示。

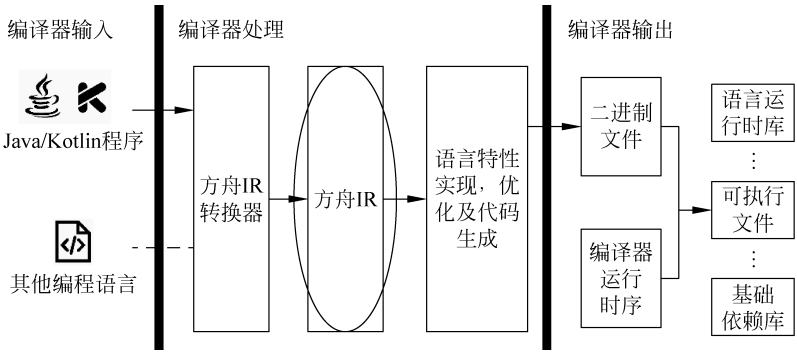

图 5.3　方舟编译器架构设计

(图源：https://www.openarkcompiler.cn/document/frameworkDesgin(有修改))

叶节点、表达式节点和语句节点一起构建了一个节点体系。叶节点通常也称为终端节点(terminal nodes),这些节点通常在运行时直接展示了一个具体的值,这个值可以是常量或者一个在存储单元里的值。表达式节点通常是表达一个对其操作数的操作,这个操作是为了计算一个结果,而它的操作数可以是一个叶节点或者是其他的表达式节点。表达式节点其实是表达式树中的内部节点,并且表达式节点的类型域里会给出表达式节点操作结果的类型。语句节点主要用来表示控制流。语句的执行是从函数的入口开始,顺序逐条执行语句,直到遇到控制流语句。语句不光可以用来修改控制流,还可以修改程序中的数据存储。语句节点的操作数可以是叶节点、表达式节点和语句节点。

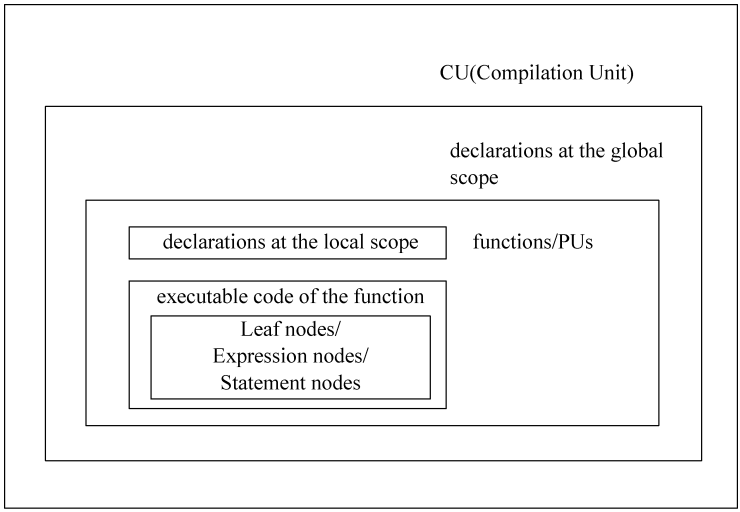

图 5.4　Maple IR 的文件结构

所以，将这三类节点的关系可以简单地理解为：语句节点可以包含自身、表达式节点和叶节点；表达式节点可以包含自身和叶节点；叶节点可以包含自身。具体如图 5.5 所示。

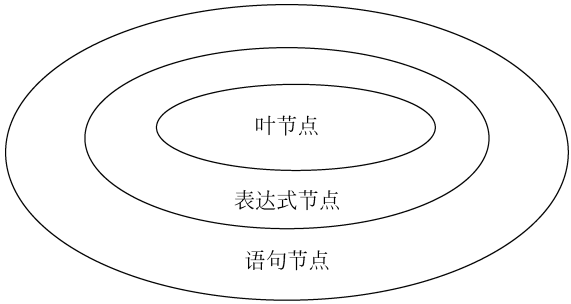

图 5.5　叶节点、表达式节点和语句节点的关系

至此，我们对 Maple IR 在方舟编译器体系结构中的位置，以及 Maple IR 本身的结构及其内部要素已经有了一个比较清

晰的认识，下一步将从代码实现的角度来认识 Maple IR 的结构。

5.3　Maple IR 结构表示代码

Maple IR 结构的表示代码，通常根据其层面的不同，涉及如下几个常用的类：MIRModule 类、MIRFunction 类、BaseNode 类等。

MIRModule 类是用来表示 Maple IR 的 module 的相关信息，对应着 Maple IR 结构中的编译单元(CU)，所有 module 相关的信息和操作都在该类中定义。该类的定义和实现在源码中位于 src/maple _ ir/include/mir _ module. h 和 src/maple _ ir/src/mir _ module. cpp 中。

MIRFunction 类用来表示 Maple IR 的 function 的相关信息，对应着 Maple IR 结构中的 function，是 module 的下一层结构，它包含了 function 相关的信息和操作。该类的定义和实现在源码中位于 src/maple_ir/include/mir_function. h 和 src/maple_ir/src/mir_function. cpp 中。

BaseNode 类是 Maple IR 中节点类的父类，所有的各个类型的节点类都继承自它或者它的子类。BaseNode 类及其子类通常对应一个表达式或者一个语句，是 Maple IR 中 function 下一层结构，属于 function 的一部分。BaseNode 与子类所对应的节点类构成的节点体系，正是结构中的节点体系的实现。BaseNode 类的定义和实现是在 src/maple_ir/include/mir_nodes. h 和 src/maple_

ir/src/mir_nodes.cpp 中。

MIRModule、MIRFunction 和 BaseNode 的众多子类，一起构成了 Maple IR 代码实现层面上的一个基本结构，对应上文所介绍的 Maple IR 的结构中的内容。

5.4 Maple IR 中的基本类型的设计与实现

基本类型系统是 IR 设计中需要重点考虑的内容，也是 IR 中的重要元素，它直接决定着整个 IR 的类型表达体系。本部分内容将对 Maple IR 基本类型的设计与实现进行简要介绍。

5.4.1 基本类型的设计

Maple IR 的官方文档 *Maple IR Design* 中，对基本类型进行了系统描述。具体如下：

- no type -void
- signed integers -i8，i16，i32，i64
- unsigned integers -u8，u16，u32，u64
- booleans-u1
- addresses -ptr，ref，a32，a64
- floating point numbers -f32，f64
- complex numbers -c64，c128
- JavaScript types：
  - dynany

  - dynu32
  - dyni32
  - dynundef
  - dynnull
  - dynhole
  - dynbool
  - dynptr
  - dynf64
  - dynf32
  - dynstr
  - dynobj
- SIMD types -(to be defined)
- unknown

Maple IR 将其基本类型分为 10 类，分别是：空类型（no type）、符号整型（signed integers）、无符号整型（unsigned integers）、布尔类型（booleans）、地址类型（addresses）、浮点数类型（floating point numbers）、复杂数（complex numbers）、JavaScript 类型（JavaScript types）、SIMD types 和 unknown 类型。

这里需要专门把 JavaScript 类型进行单独介绍。方舟编译器的设计初衷是要支持多语言和多目标平台，其多语言支持计划中就包含了对 JavaScript 的支持。而 Maple IR 中专门预留了一系列的 JavaScript 类型，想必是为了支持 JavaScript。但是，在支持多语言的编译器 IR 设计中，专门为某种语言设计一类专有的基本类型这种操作，并不常见。因为这种语言专用的基本类型，对于其他语言来讲都是冗余信息，而且随着语言的增多，语言专用的基本类型

可能会越来越多，那么发展到最后 IR 的基本体系就会变得繁复无比，失去了多语言 IR 的优势，从而变成了多个语言的 IR 的简单合并。目前，方舟编译器还未能支持 JavaScript 语言，所以不清楚是什么原因导致这种设计，只能对 Maple IR 的基本类型演进保持关注。

Maple IR 的基本类型，在代码中也有列表呈现，位于 src/maple_ir/include/prim_types.def 中，代码如下：

```
//第 5 章/prim_types.def
 PRIMTYPE(void)
  PRIMTYPE(i8)
  PRIMTYPE(i16)
  PRIMTYPE(i32)
  PRIMTYPE(i64)
  PRIMTYPE(u8)
  PRIMTYPE(u16)
  PRIMTYPE(u32)
  PRIMTYPE(u64)
  PRIMTYPE(u1)
  PRIMTYPE(ptr)
  PRIMTYPE(ref)
  PRIMTYPE(a32)
  PRIMTYPE(a64)
  PRIMTYPE(f32)
  PRIMTYPE(f64)
  PRIMTYPE(f128)
  PRIMTYPE(c64)
  PRIMTYPE(c128)
#ifdef DYNAMICLANG
  PRIMTYPE(simplestr)
  PRIMTYPE(simpleobj)
  PRIMTYPE(dynany)
  PRIMTYPE(dynundef)
```

```
    PRIMTYPE(dynnull)
    PRIMTYPE(dynbool)
    PRIMTYPE(dyni32)
    PRIMTYPE(dynstr)
    PRIMTYPE(dynobj)
    PRIMTYPE(dynf64)
    PRIMTYPE(dynf32)
    PRIMTYPE(dynnone)
  #endif
    PRIMTYPE(constStr)
    PRIMTYPE(gen)
    PRIMTYPE(agg)
    PRIMTYPE(unknown)
```

PRIMTYPE(agg)这个列表中的基本类型和文档 *Maple IR Design* 中的基本类型列表中的基本类型并不相同。prim_types.def 里定义的基本类型和文档中所描述的基本类型对比起来有几点问题：代码里定义了 f128，文档中并没有 f128；代码里定义了 simplestr、simpleobj、dynnone、constStr、gen 和 agg，但是文档中并没有定义这几个基本类型；文档中为 JavaScript 类型定义了 dynu32、dynhole、dynptr，但是代码中没有定义这 3 个基本类型。所以，代码和文档之中的基本类型定义，在主题内容相同的情况下，还存在着部分差异。如图 5.6 所示，文档和源码相同的部分，就是两个椭圆公共的交集部分，这部分内容进行了省略，剩余两个椭圆自有的部分则是两个部分的差异。其整体情况如图 5.7 所示。

这种文档和源码出现差异的情况，比较大的概率是因为方舟编译器刚刚开源，支持的程序语言和目标平台还比较单一，没有对 Maple IR 进行更多打磨，否则不会出现这种情况。在方舟编译器

未来的发展过程中，这两者对于基本类型的描述，必然会趋于统一，变成完全一致的内容。而现阶段，在文档和代码有冲突的情况下，我们只能以代码为准。

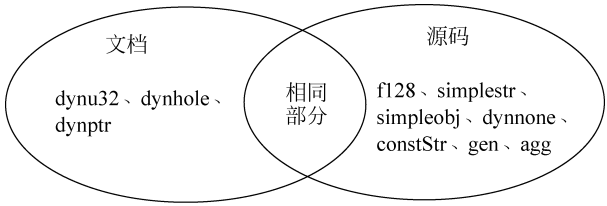

图 5.6　文档和源码中基本类型的差异

5.4.2　Maple IR 基本类型的实现

Maple IR 基本类型的代码实现，主要涉及基本类型的定义文件 prim_types.def、结构体 PrimitiveTypeProperty 和 PrimitiveType 类。

1. prim_types.def 分析

基本类型的定义文件 prim_types.def 位于 src/maple_ir/include/目录之下，其主要内容是通过宏 PRIMTYPE(P)列出的基本类型列表。具体内容在上文介绍代码的基本类型时已经有引用，代码如下：

```
PRIMTYPE(void)
PRIMTYPE(i8)
PRIMTYPE(i16)
PRIMTYPE(i32)
PRIMTYPE(i64)
```

同时，在该文件中，还为每个基本类型定义了一个结构体

PrimitiveTypeProperty 类型的静态常量，内容主要是类型的类别和属性，这和后面要介绍的 PrimitiveTypeProperty 结构体要结合起来看。以 i8 为例，其用 PTY_i8 表示其类型，true 表示的是其为整型，这些信息都可以从其对应的注释中看出。例如 PTY_i8 对应的注释是 type，true 对应的注释是 isInteger，其他的内容也是类型情况，代码如下：

```
//第 5 章/prim_types1.def
static const PrimitiveTypeProperty PTProperty_i8 = {
  /* type = */PTY_i8, /* isInteger = */true, /* isUnsigned =
*/false, /* isAddress = */false, /* isFloat = */false,
  /* isPointer = */false, /* isSimple = */false, /* isDynamic =
*/false, /* isDynamicAny = */false, /* isDynamicNone = */false
};
```

2. 结构体 PrimitiveTypeProperty

结构体 PrimitiveTypeProperty 定义位于 src/maple_ir/include/cfg_primitive_type.h 中。这个文件中除了定义该结构体之外，还定义了枚举 PrimType，以及 GetPrimitiveTypeProperty 函数的声明。

结构体 PrimitiveTypeProperty 的定义不复杂，除了一个 PrimType 类型的变量 type，就是一系列的 bool 值，用来标明 type 的一些基本属性，代码如下：

```
//第 5 章/cfg_primitive_type.h
struct PrimitiveTypeProperty {
  PrimType type;
```

```
    bool isInteger;
    bool isUnsigned;
    bool isAddress;
    bool isFloat;
    bool isPointer;
    bool isSimple;
    bool isDynamic;
    bool isDynamicAny;
    bool isDynamicNone;
};
```

其中枚举 PrimType 中包含了所有的基本类型，但是并没有直接在内部列出来，而是通过定义宏 PRIMTYPE(P)并且包含 prim_types.def 的形式来实现的，代码如下：

```
//第 5 章/cfg_primitive_type1.h
enum PrimType {
  PTY_begin,                    // PrimType begin
#define PRIMTYPE(P) PTY_##P,
#include "prim_types.def"
  PTY_end,                      // PrimType end
#undef PRIMTYPE
};
```

在文件 src/maple_ir/include/cfg_primitive_type.h 中，还声明了 GetPrimitiveTypeProperty 函数，但是 cfg_primitive_type.h 及 src/maple_ir/include/prim_types.h 没有专门对应的 cpp 文件。所以，该函数的具体实现在 src/maple_ir/src/mir_type.cpp 中，这个函数返回的就是基本类型所对应的 PTProperty_##P。而 PTProperty_##P 这个静态常量的实现，则以列表的形式和基本类型一起在 prim_types.def 文件中。所返回的静态常量，也是为

了表示基本类型和基本类型的属性，代码如下：

```
//第 5 章/cfg_primitive_type2.h
const PrimitiveTypeProperty &GetPrimitiveTypeProperty(PrimType
pType) {
  switch (pType) {
    case PTY_begin:
      return PTProperty_begin;
#define PRIMTYPE(P) \
    case PTY_##P: \
      return PTProperty_##P;
#include "prim_types.def"
#undef PRIMTYPE
    case PTY_end:
    default:
      return PTProperty_end;
  }
}
```

3. PrimitiveType 类

PrimitiveType 类的定义位于 src/maple_ir/include/prim_types.h 中，该头文件专属于 PrimitiveType 类，没有其他的内容。PrimitiveType 类中只有一个私有成员变量，是 PrimitiveTypeProperty 类型的变量。所有的成员函数，其功能是获取 PrimitiveTypeProperty 类型的成员变量内的相关数值，代码如下：

```
//第 5 章/prim_types.h
class PrimitiveType {
 public:
  // we need implicit conversion from PrimType to PrimitiveType, so
//there is no explicit keyword here.
```

```
PrimitiveType(PrimType type) : property(GetPrimitiveTypeProperty
(type)) {}
  ~PrimitiveType() = default;

  PrimType GetType() const {
    return property.type;
  }

  bool IsInteger() const {
    return property.isInteger;
  }
  bool IsUnsigned() const {
    return property.isUnsigned;
  }
  bool IsAddress() const {
    return property.isAddress;
  }
  bool IsFloat() const {
    return property.isFloat;
  }
  bool IsPointer() const {
    return property.isPointer;
  }
  bool IsDynamic() const {
    return property.isDynamic;
  }
  bool IsSimple() const {
    return property.isSimple;
  }
  bool IsDynamicAny() const {
    return property.isDynamicAny;
  }
  bool IsDynamicNone() const {
    return property.isDynamicNone;
```

```
    }

  private:
    const PrimitiveTypeProperty &property;
};
```

代码较为简单，等于将之前的基本类型相关的内容，都封装在这个类里，通过这个类可以表示一个基本类型的所有相关信息，并且可以进行读取操作。一个基本类型的所有信息也不多，主要是基本类型的具体类型和相关属性。

本节通过 prim_types.def、cfg_primitive_types.h 和 prim_types.h 三个文件的内容，以及部分 mir_types.cpp 的内容，介绍了 Maple IR 的基本类型的具体实现。而这些具体实现，最终都封装到了 PrimitiveType 类中，以 PrimitiveType 类去实现具体的基本类型的相关操作。所以，也可以简单地将 PrimitiveType 类直接视为是 Maple IR 基本类型的实现。

5.5　Maple IR 中的控制流语句的设计与实现

控制流语句也是 Maple IR 中的重要构成部分，它影响着 Maple IR 的程序的执行流程。程序的控制流是通过层次型语句或者平坦型语句列表来展现的，所以 Maple IR 的控制流语句分为两种：Hierarchical control flow statements 和 Flat control flow statements。前者更接近源程序，后者更加接近机器指令。所以在

多层的 IR 设计体系之中，前者多用在高层次的 IR 中，后者多用在低层次的 IR 中。

5.5.1 控制流语句的设计

Maple IR 的设计中，将控制流语句分为 Hierarchical control flow statements 和 Flat control flow statements 两种。按照文档 *Maple IR Design* 的描述，Hierarchical control flow statements 有：doloop、dowhile、foreachelem、if 和 while；而 Flat control flow statements 有：brfalse、brtrue、goto、multiway、return、switch、rangegoto 和 indexgoto。

然而，源码中关于控制流语句的分类，却和文档之中所介绍的有一些差异。文件 src/maple_ir/include/opcodes.def 中包含了 opcode 列表，其中控制流语句相关内容也在其中，代码如下：

```
//第 5 章/opcodes.def
// hierarchical control flow opcodes
OPCODE(block, BlockNode, (OPCODEISSTMT | OPCODENOTMMPL), 0)
OPCODE(doloop, DoloopNode, (OPCODEISSTMT | OPCODENOTMMPL), 0)
OPCODE(dowhile, WhileStmtNode, (OPCODEISSTMT | OPCODENOTMMPL), 0)
OPCODE(if, IfStmtNode, (OPCODEISSTMT | OPCODENOTMMPL), 0)
OPCODE(while, WhileStmtNode, (OPCODEISSTMT | OPCODENOTMMPL), 0)
OPCODE(switch, SwitchNode, (OPCODEISSTMT | OPCODENOTMMPL), 8)
OPCODE(multiway, MultiwayNode, (OPCODEISSTMT | OPCODENOTMMPL), 8)
OPCODE ( foreachelem,  ForeachelemNode,  ( OPCODEISSTMT  |
OPCODENOTMMPL), 0)

  // flat control flow opcodes
OPCODE(goto, GotoNode, OPCODEISSTMT, 8)
OPCODE(brfalse, CondGotoNode, OPCODEISSTMT, 8)
OPCODE(brtrue, CondGotoNode, OPCODEISSTMT, 8)
```

```
OPCODE(return, NaryStmtNode, (OPCODEISSTMT | OPCODEISVARSIZE |
OPCODEHASSSAUSE), 0)
OPCODE(rangegoto, RangeGotoNode, OPCODEISSTMT, 8)
```

根据上述代码，switch、multiway 不属于 flat control flow statements，而属于 hierarchical control flow statements。同时，indexgoto 这个控制流语句在代码之中根本没出现，目前开源的所有代码中都没有它的相关内容，疑似在文档中设计了此控制流语句之后并没有在实际之中使用。

5.5.2　控制流语句的实现

每一个控制流语句都有对应的节点类，这些节点类一起构成了控制流语句的实现体系。控制流语句的实现其实是语句实现中的一部分，所以控制流语句的实现体系，也是语句的实现体系的一部分。接下来则逐个介绍控制流语句的实现类，并且会用图展现这些类之间的继承关系。

根据文档的分类，hierarchical control flow statements 有 doloop、dowhile、foreachelem、if 和 while。其中 doloop 对应的节点类是 DoloopNode 类，它继承于 StmtNode 类。dowhile 和 while 对应着同一个节点类 WhileStmtNode，WhileStmtNode 继承自 UnaryStmtNode，UnaryStmtNode 继承自 StmtNode 类。foreachelem 对应的节点类为 ForeachelemNode，ForeachelemNode 继承自 StmtNode 类。if 语句对应的节点类为 IfStmtNode，IfStmtNode 继承自 UnaryStmtNode。这几个节点类及其父类，其继承关系如图 5.7 所示，这几个节点类都是 StmtNode 或者其子类 UnaryStmtNode 的子类，而 StmtNode 类是继承于 BaseNode 和 PtrListNodeBase。

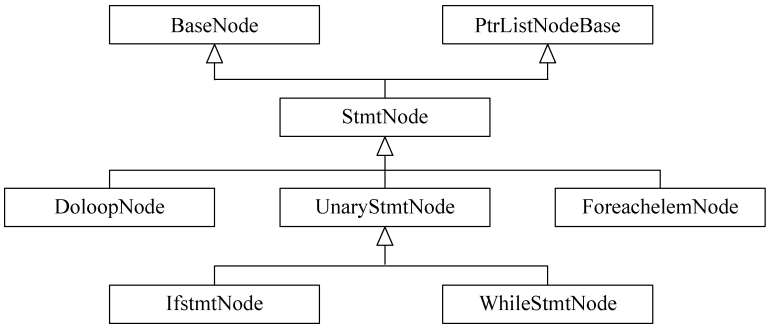

图 5.7　hierarchical control flow statements 实现类及继承关系

flat control flow statements 有 brfalse、brtrue、goto、multiway、return、switch、rangegoto 和 indexgoto。其中，brfalse 和 brtrue 对应的节点类是 CondGotoNode，CondGotoNode 继承于 UnaryStmtNode 类。goto 对应的节点类是 GotoNode，GotoNode 继承于 StmtNode 类。multiway 对应的节点类是 MultiwayNode，MultiwayNode 继承于 StmtNode 类。return 对应的节点类是 NaryStmtNode，它继承于 StmtNode 和 NaryOpnds。switch 对应的节点类是 SwitchNode，SwitchNode 继承自 StmtNode 类。rangegoto 对应的节点类是 RangegotoNode，RangegotoNode 继承自 UnaryStmtNode 类。indexgoto 并没有在源码之中使用过，所以也没有对应的节点类。这些节点的继承关系如图 5.8 所示。

总之，所有的控制流语句所对应的节点，都是 StmtNode 或者其子类 UnaryStmtNode 的子类。这些节点类的实现，都位于 src/maple_ir/include/mir_nodes.h 和 src/maple_ir/src/mir_nodes.cpp 文件中。

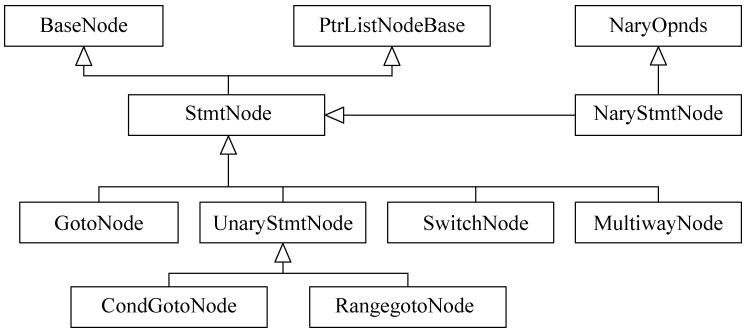

图 5.8　flat control flow statements 实现类及继承关系