LLVM 语言参考手册（部分）

一．摘要

本文档是 LLVM 汇编语言的参考手册。LLVM 是一种基于静态单赋值（SSA）的表示方法，它提供了类型安全性、低级操作、灵活性以及清晰地表示“所有”高级语言的能力。它是在 LLVM 编译策略的所有阶段中使用的通用代码表示。

二．介绍

LLVM 代码表示被设计成三种不同的形式：作为内存中的编译器 IR、作为磁盘上的位代码表示（适合于由实时编译器快速加载）和作为人类可读的汇编语言表示。这允许 LLVM 为高效的编译器转换和分析提供强大的中间表示，同时提供调试和可视化转换的自然方法。LLVM 的三种不同形式都是等价的。本文档描述了人类可读的表示法和符号。

LLVM 表示的目标是轻量级和低级别，同时具有表现性、类型化和可扩展性。它的目标是成为一种“通用 IR”，其级别足够低，高级思想可以清晰地映射到它上面（类似于微处理器是“通用 IR”，允许许多源语言映射到它们）。通过提供类型信息，LLVM 可以作为优化的目标：例如，通过指针分析，可以证明在当前函数之外永远不会访问 C 自动变量，从而允许将其提升为简单的 SSA 值，而不是内存位置。

三．完构性

需要注意的是，本文档描述了“格式良好”的 LLVM 汇编语言。解析器接受什么和被认为是“格式良好”的区别。例如，以下指令在语法上是可以的，但格式不好：

%x = add i32 1，%x

因为%x 的定义并不支配它的所有用途。LLVM 基础结构提供了一个验证过程，该验证过程可用于验证 LLVM 模块是否已形成良好。解析输入程序集后，解析器自动运行此过程，优化器在输出比特代码之前自动运行。验证器过程指出的违规行为表明转换过程中的错误或解析器的输入错误。

四．标识符

LLVM 标识符有两种基本类型：全局标识符和本地标识符。全局标识符（函数、全局变量）以“@”字符开头。本地标识符（寄存器名、类型）以“%”字符开头。此外，标识符有三种不同的格式，用于不同的目的：

命名值表示为带有前缀的字符串。例如，%foo，@DivisionByZero，%a.really.long.identifier。实际使用的正则表达式是“[%@][-a-zA-Z$._][-a-zA-Z$._ 0-9]*”。名称中需要其他字符的标识符可以用引号括起来。特殊字符可以用“\xx”转义，其中 xx 是十六进制字符的 ASCII 码。这样，任何字符都可以在名称值中使用，甚至可以引用它们自己。可以在全局值上使用“\01”前缀来抑制损坏。

未命名的值表示为带有前缀的无符号数值。例如，%12、@2、%44。

常量，在下面的常量一节中描述。

LLVM 要求值以前缀开头有两个原因：编译器不需要担心名称与保留字的冲突，并且保留字集将来可以扩展而不受惩罚。此外，未命名标识符允许编译器快速生成临时变量，而无需避免符号表冲突。

LLVM 中的保留字与其他语言中的保留字非常相似。不同的操作码有关键字（'add'、'bitcast'、'ret'等）、基元类型名（'void'、'i32'等）等。这些保留字不能与变量名冲突，因为它们都不是以前缀字符（“%”或“@”）开头的。

以下是整数变量“%X”乘以 8 的 LLVM 代码示例：

简单方法：

```
%result = mul i32%X, 8
```

强度降低后：

```
%result = shl i32%X, 3
```

困难方法：

```
%0 = add i32%X, %X          ；产生 i32:%0
%1 = add i32%0, %0          ；产生 i32:%1
%result = add i32%1, %1
```

将%X 乘以 8 的最后一种方法说明了 LLVM 的几个重要词汇特性：

注释用“；”分隔，一直到行尾。

当计算结果未指定给命名值时，将创建未命名临时值。

未命名的临时变量按顺序编号（使用每个函数递增计数器，从 0 开始）。请注意，基本块和未命名的函数参数都包含在此编号中。例如，如果 entry basic 块没有给定标签名，并且所有函数参数都被命名，那么它将得到数字 0。

它还显示了我们在本文件中遵循的惯例。在演示指令时，我们将遵循一条指令，其中包含一条注释，用于定义生成的值的类型和名称。

五．高层结构

1. 模块结构

LLVM 程序由模块组成，每个模块都是输入程序的翻译单元。每个模块由函数、全局变量和符号表条目组成。模块可以与 LLVM 链接器组合在一起，LLVM 链接器合并函数（和全局变量）定义、解析前向声明和合并符号表条目。以下是“hello world”模块的示例：

```
; Declare the string constant as a global constant.
@.str = private unnamed_addr constant [13 x i8] c"hello world\0A\00"

; External declaration of the puts function
declare i32 @puts(i8* nocapture) nounwind

; Definition of main function
define i32 @main() {     ; i32()*
  ; Convert [13 x i8]* to i8*...
  %cast210 = getelementptr [13 x i8], [13 x i8]* @.str, i64 0, i64 0

  ; Call puts function to write out the string to stdout.
  call i32 @puts(i8* %cast210)
  ret i32 0
}

; Named metadata
!0 = !{i32 42, null, !"string"}
!foo = !{!0}
```

这个示例由一个名为“.str”的全局变量、“puts”函数的外部声明、“main”的函数定义和命名元数据“foo”组成。

通常，模块由全局值列表组成（其中函数和全局变量都是全局值）。全局值由指向内存位置的指针（在本例中是指向字符数组的指针和指向函数的指针）表示，并具有以下链接类型之一。

2. 链接类

所有全局变量和函数都有以下一种类型的链接：

私有的：具有“private”链接的全局值只能由当前模块中的对象直接访问。特别是，将代

码链接到具有私有全局值的模块中可能会导致根据需要重命名私有以避免冲突。因为符号是模块专用的，所以可以更新所有引用。这不会显示在对象文件的任何符号表中。

内部的：类似于 private，但该值在对象文件中显示为本地符号（在 ELF 的情况下为 STB\u local）。这与 C 中“static”关键字的概念相对应。

外部可用：带有“available_ external”链接的全局变量永远不会被发送到与 LLVM 模块对应的对象文件中。从链接器的角度来看，一个可用的外部全局性声明相当于一个外部声明。它们的存在是为了允许内联和其他优化在给定全局定义的知识的情况下进行，而全局定义已知在模块之外的某个地方。允许随意丢弃具有可用链接的全局变量，并允许进行内联和其他优化。此链接类型仅允许用于定义，不允许用于声明。

Linkonce: 当链接发生时，具有“linkonce”链接的全局变量与其他同名全局变量合并。这可以用于实现某些形式的内联函数、模板或其他代码，这些代码必须在使用它的每个翻译单元中生成，但稍后可能会用更明确的定义覆盖主体。允许丢弃未引用的 linkonce 全局变量。请注意，linkonce 链接实际上不允许优化器将函数体内联到调用程序中，因为优化器不知道函数的这个定义是否是程序中的最终定义，也不知道它是否会被更强大的定义覆盖。要启用内联和其他优化，请使用“linkonce_odr”链接。

弱链接: “弱”链接与 linkonce 链接具有相同的合并语义，只是具有弱链接的未引用全局变量不能被丢弃。这用于在 C 源代码中声明为“弱”的全局变量。

公共链接：“公共”链接与“弱”链接最为相似，但它们在 C 中用于暂定定义，例如全局范围内的“intx；”。具有“公共”链接的符号以与弱符号相同的方式合并，如果未引用，则不能删除它们。公共符号不能有显式节，必须有零初始值设定项，并且不能标记为“常量”。函数和别名可能没有公共链接。

追加链接：“追加”链接只能应用于指向数组类型的指针的全局变量。当两个带有附加链接的全局变量链接在一起时，两个全局数组被附加在一起。这是 LLVM，typesafe，相当于在链接.o 文件时让系统链接器用相同的名称将“节”附加在一起。

不幸的是，这与.o 文件中的任何特性都不对应，因此它只能用于 llvm.global 目录 llvm 专门解释的。

外部，弱链接：这个链接的语义遵循 ELF 对象文件模型：符号在链接之前是弱的，如果没有链接，符号将变为 null 而不是未定义的引用。

linkonce_ odr, weak_odr：有些语言允许合并不同的全局变量，例如两个具有不同语义的函数。其他语言，如 C++，确保只有统一的全局被合并（“一个定义规则”——“ODR”）。这类语言可以使用 linkonce\u odr 和 weak_ odr 链接类型来表示 global 只能与等价的 global 合并。这些链接类型在其他方面与其非 odr 版本相同。

外部的

如果没有使用上述标识符，则全局是外部可见的，这意味着它参与链接并可用于解析外部符号引用。

全局变量或函数声明具有外部或外部以外的任何链接类型是非法的。

六．类型系统

LLVM 类型系统是中间表示最重要的特性之一。类型化使许多优化可以直接在中间表示上执行，而不必在转换之前对另一方面进行额外的分析。强类型系统使读取生成的代码变得更容易，并支持对常规三地址代码表示不可行的新颖分析和转换。

1. void 类

概述：void 类型不表示任何值并且没有大小。

语法：void

2. Function 类

概述：函数类型可以看作是函数签名。它由一个返回类型和一系列形式参数类型组成。函数类型的返回类型是 void 类型或第一类类型（label 和 metadata 类型除外）。

语法：<returntype>（<parameter list>）

其中“<parameter list>”是以逗号分隔的类型说明符列表。（可选）参数列表可以包含类型…，该类型指示函数采用可变数量的参数。变量参数函数可以通过变量参数处理内在函数来访问它们的参数。“<returntype>”是除标签和元数据之外的任何类型。

示例：

i32（i32）函数获取 i32，返回 i32

float（i16，i32*）*指向一个函数的指针，该函数接受一个 i16 和一个指向 i32 的指针，返回 float。

i32（i8*，…）一个 vararg 函数，它至少有一个指向 i8 的指针（C 中的字符），该指针返回一个整数。这是 LLVM 中 printf 的签名。

{i32，i32}（i32）接受 i32 的函数，返回包含两个 i32 值的结构。

3. First Class 类

第一类类型可能是最重要的。这些类型的值是唯一可以由指令生成的值。

3.1 Single Value 类

从 CodeGen 的角度来看，这些类型在寄存器中是有效的。

3.2 Integer 类

概述：integer 类型是一种非常简单的类型，它只是为所需的 integer 类型指定一个任意的位宽度。可以指定从 1 位到 223-1（约 800 万）的任何位宽度。

语法：iN

整数将占用的位数由 N 值指定。

示例：

i1 是一个单位整数。

i32 是 32 位整数。

i1942652 是一个超过一百万位的大整数。

3.3 Floating-Point 类

Type	Description
half	16 位浮点值
bfloat	16 位“脑”浮点值（7 位有效）。提供与 float 相同数量的指数位，以便匹配其动态范围，但精度大大降低。用于 Intel 的 AVX-512 BF16 扩展和 Arm 的 ARMv8.6-A 扩展等。
float	32 位浮点值
double	64 位浮点值
fb128	128 位浮点值（113 位有效位）
x86_fp80	80 位浮点值（X87）
ppc_fp128	128 位浮点值（两个 64 位）

half、float、double 和 fp128 的二进制格式分别对应于 binary16、binary32、binary64 和 binary128 的 IEEE-754-2008 规范。

3.4 X86_mmx 类

概述：x86_mmx 类型表示保存在 x86 机器上的 mmx 寄存器中的值。它允许的操作非常有限：参数和返回值、加载和存储以及位广播。用户指定的 MMX 指令表示为具有此类型的参数和/或结果的内在调用或 asm 调用。没有这种类型的数组、向量或常量。

语法：X86_mmx

3.5 Pointer 类

概述：指针类型用于指定内存位置。指针通常用于引用内存中的对象。指针类型可能有一个可选的地址空间属性，定义指向对象所在的编号地址空间。默认地址空间是数字 0。非零地址空间的语义是特定于目标的。

注意，LLVM 不允许指针 void（void*），也不允许指针指向标签（label*）。改用 i8*。

语法：<type> *

样例：

[4 x i32]*	指向四个 i32 值的数组的指针。
i32 (i32*) *	指向函数的指针，该函数采用 i32*，返回 i32。
i32 addrspace(5)*	指向驻留在地址空间#5 中的 i32 值的指针。

3.6 Vector 类

概述：向量类型是表示元素向量的简单派生类型。当使用单个指令（SIMD）并行操作多个基本数据时，使用向量类型。一个向量类型需要一个大小（元素的数量）、一个基本的数据类型和一个可伸缩的属性来表示在编译时确切的硬件向量长度未知的向量。向量类型被认为是第一类。

内存布局：一般来说，向量元素在内存中的布局方式与数组类型相同。只要向量元素是字节大小的，这种异常就可以正常工作。然而，当向量的元素不是字节大小时，它会变得更复杂一些。描述布局的一种方法是描述当诸如<N x iM>的向量被比特转换为具有 N×M 比特的整数类型时发生的情况，然后遵循将这样的整数存储到存储器的规则。

从向量类型到标量整数类型的位广播将看到元素被打包在一起（没有填充）。在整数中插入元素的顺序取决于大端还是小端。对于小端元素，零放在整数的最低有效位，对于大端元素，零放在最高有效位。

以<i4 1，i4 2，i4 3，i4 5>这样的向量为例，再加上我们可以将向量存储替换为一个位广播后接一个整数存储的类比，我们将其用于大端。

3.7 Label 类

概述：标签类型表示代码标签。

语法：label

3.8 Token 类

概述：当值与指令关联时使用令牌类型，但该值的所有使用都不得试图反省或模糊它。因此，不适合使用 phi 或类型为 select 的令牌。

语法：token

3.9 Metadata 类

概述：元数据类型表示嵌入的元数据。除了函数参数外，不能从元数据创建派生类型。

语法：metadata

3.10 Aggregate Types 类

聚合类型是可以包含多个成员类型的派生类型的子集。数组和结构是聚合类型。向量不被认为是聚合类型。

3.10.1 Array 类

概述：数组类型是一个非常简单的派生类型，它在内存中按顺序排列元素。数组类型需

要大小（元素数）和基础数据类型。

语法：[<# elements> x <elementtype>]

元素的数目是一个常量整数值；elementtype 可以是任何具有大小的类型。

样例：

[40 x i32]	40 个 32 位整数值的数组。
[41 x i32]	41 个 32 位整数值的数组。
[4 x i8]	4 个 8 位整数值的数组。

以下是多维数组的一些示例：

[3 x [4 x i32]]	32 位整数值的 3x4 数组。
[12 x [10 x float]]	12x10 单精度浮点值数组。
[2 x [3 x [4 x i16]]]	2x3x4 16 位整数值数组。

对于超出静态类型所暗示的数组结尾的索引没有限制（尽管在某些情况下，对于超出已分配对象的边界的索引有限制）。这意味着可以在具有零长度数组类型的 LLVM 中实现一维“可变大小数组”寻址。例如，LLVM 中“pascal 样式数组”的实现可以使用类型“{i32，[0xfloat]}”。

七．常量

LLVM 有几种不同的基本类型的常量。本节将介绍它们及其语法。

1. 布尔常量

两个字符串'true'和'false'都是 i1 类型的有效常量。

2. 整型常量

标准整数（如“4”）是整数类型的常量。负数可以与整数类型一起使用。

3. 浮点常量

浮点常量使用标准十进制表示法（如 123.421）、指数表示法（如 1.23421e+2）或更精确的十六进制表示法（见下文）。汇编程序需要浮点常量的精确十进制值。例如，汇编程序接受 1.25，但拒绝 1.3，因为 1.3 是二进制的重复小数。浮点常量必须具有浮点类型。

4. 空指针常量

标识符“null”被识别为空指针常量，并且必须是指针类型。

八．指令参考

1. ‘ret’指令

语法： ret<type><value>；从非 void 函数返回值

ret void；从 void 函数返回

概述：“ret”指令用于将控制流（以及可选的值）从函数返回给调用者。“ret”指令有两种形式：一种是返回一个值然后引起控制流，另一种是只引起控制流。“ret”指令可以选择接受一个参数，即返回值。返回值的类型必须是“first class”类型。

如果函数具有非 void 返回类型并且包含没有返回值的'ret'指令，或者返回值的类型与其类型不匹配，或者如果函数具有 void 返回类型并且包含具有返回值的'ret'指令，则该函数的格式不正确。

样例：

ret i32 5；返回整数值 5

ret void；从 void 函数返回

ret{i32，i8}{i32 4，i8 2}；返回值为 4 和 2 的结构体

2. ‘br’指令

语法：

br i1 <cond>, label <iftrue>, label <iffalse>

br label <dest>　　;无条件跳转

概述：“br”指令用于使控制流转移到当前函数中的不同基本块。此指令有两种形式，分别对应于条件分支和无条件分支。

样例：

```
Test:
    %cond = icmp eq i32 %a, %b
     br i1 %cond, label %IfEqual, label %IfUnequal
IfEqual:
     ret i32 1
IfUnequal:
    ret i32 0
```

3. ‘add’指令

语法：

<result> = add <ty> <op1>, <op2>

<result> = add nuw <ty> <op1>, <op2>

<result> = add nsw <ty> <op1>, <op2>

<result> = add nuw nsw <ty> <op1>, <op2>

概述：“add”指令返回其两个操作数之和。

样例：<result> = add i32 4, %var　　；产生结果 i32:result = 4 + %var

4. ‘sub’指令

语法：

<result> = sub <ty> <op1>, <op2>

<result> = sub nuw <ty> <op1>, <op2>

<result> = sub nsw <ty> <op1>, <op2>

<result> = sub nuw nsw <ty> <op1>, <op2>

概述：“sub”指令返回其两个操作数之差。请注意，“sub”指令用于表示大多数其他中间表示形式中的“neg”指令。

样例：<result> = sub i32 4, %var　　；产生结果 i32:result = 4 - %var

<result> = sub i32 0, %val　　；产生结果 i32:result = -%var

5. ‘mul’指令

语法：

<result> = mul <ty> <op1>, <op2>

<result> = mul nuw <ty> <op1>, <op2>

<result> = mul nsw <ty> <op1>, <op2>

<result> = mul nuw nsw <ty> <op1>, <op2>

概述：“mul”指令返回其两个操作数的乘积。

样例：<result> = mul i32 4, %var　　；产生结果 i32:result = 4 * %var

6. ‘alloca’指令

语法：

<result>=alloca[inalloca]<type>[，<ty><NumElements>][，align<alignment>][，addrspace（<num>）]；产生类型 addrspace（num）*:result

概述：“alloca”指令在当前正在执行的函数的堆栈帧上分配内存，当该函数返回到调用方时自动释放内存。对象总是在 datalayout 中指定的 alloca 的地址空间中分配。

样例：%ptr = alloca i32 ; 产生 i32*:ptr

%ptr = alloca i32, i32 4 ; 产生 i32*:ptr

%ptr = alloca i32, i32 4, align 1024 ; 产生 i32*:ptr

%ptr = alloca i32, align 1024 ; 产生 i32*:ptr

7. ‘load’指令

概述：“load”指令用于从内存中读取数据。

样例：%ptr = alloca i32 ; 产生 i32*:ptr

store i32 3, i32* %ptr ; 产生 void

%val = load i32, i32* %ptr ; 产生 i32:val = i32 3

8. ‘icmp’指令

语法：<result>=icmp<cond><ty><op1>，<op2>；产生 i1 或<N x i1>：result

概述：“icmp”指令根据其两个整数、整数向量、指针或指针向量操作数的比较，返回布尔值或布尔值向量。

“icmp”指令采用三个操作数。第一个操作数是指示要执行的比较类型的条件代码。它不是一个值，只是一个关键字。可能的条件代码为：

eq：相等

ne:不相等

ugt:无符号大于

uge:无符号的大于或等于

ult:无符号小于

ule：无符号小于或等于

sgt：签名大于

sge：符号大于或等于

slt:签名小于

sle：符号小于或等于

样例：

<result> = icmp eq i32 4, 5 ; 产生: result=false

<result> = icmp ne float* %X, %X ; 产生: result=false

<result> = icmp ult i16 4, 5 ; 产生: result=true

<result> = icmp sgt i16 4, 5 ; 产生: result=false

<result> = icmp ule i16 -4, 5 ; 产生: result=false

<result> = icmp sge i16 4, 5 ; 产生: result=false

九．样例

1. 定义一个 add 函数，并在 main 函数中掉用。

```
define i32 @add(i32 %add.a, i32 %add.b) {      //int add(int a, int b)
entry:                                          //{
    %add.c = alloca i32                         //int c = 0
    %0 = add i32 0, 0
    store i32 %0, i32* %add.c
    %1 = add i32 0, %add.a                      //c = a + b
    %2 = add i32 0, %add.b
```

```
    %3 = add i32 %1, %2
    store i32 %3, i32* %add.c
    %4 = add i32 0, %3
    %5 = load i32, i32* %add.c                    //return c
    ret i32 %5                                    //}
}

define i32 @main() {                              //int main(){
entry:
    %main.x = alloca i32                          //int x = 1
    %0 = add i32 0, 1
    store i32 %0, i32* %main.x
    %main.y = alloca i32                          //int y = 2
    %1 = add i32 0, 2
    store i32 %1, i32* %main.y
    %2 = load i32, i32* %main.x                   //x = add(x,y)
    %3 = load i32, i32* %main.y
    %4 = call i32 @add(i32 %2, i32 %3)
    store i32 %4, i32* %main.x
    %5 = add i32 0, %4                            //return 0
    %6 = add i32 0, 0
    ret i32 %6                                    //}
}
```

2. 循环

```
int main(){
    int sum = 0;
    for(int i = 0; i<6; i++){
        sum = sum + i;
    }
    return 0;
}

define i32 @main() {
entry:
    %main.sum = alloca i32                //int sum = 0
    %0 = add i32 0, 0
    store i32 %0, i32* %main.sum
    %main.for_0.i = alloca i32            //int i = 0
    %1 = add i32 0, 0
    store i32 %1, i32* %main.for_0.i
    br label %iteration_0_cond
iteration_0_cond:
    %2 = load i32, i32* %main.for_0.i     //i<sum
```

```
    %3 = add i32 0, 6
    %4 = icmp slt i32 %2, %3
    br i1 %4, label %iteration_0_body, label %iteration_0_end
iteration_0_step:
    %5 = load i32, i32* %main.for_0.i      //i++;
    %6 = add i32 %5, 1
    store i32 %6, i32* %main.for_0.i
    br label %iteration_0_cond
iteration_0_body:
    %7 = load i32, i32* %main.sum           //sum = sum + i
    %8 = load i32, i32* %main.for_0.i
    %9 = add i32 %7, %8
    store i32 %9, i32* %main.sum
    %10 = add i32 0, %9
    br label %iteration_0_step
iteration_0_end:
    %11 = add i32 0, 0
    ret i32 %11
}
```