24 中间代码：兼容不同的语言和硬件

前几节课，我带你尝试不通过IR，直接生成汇编代码，这是为了帮你快速破冰，建立直觉。在这个过程中，你也遇到了一些挑战，比如：

- 你要对生成的代码进行优化，才有可能更好地使用寄存器和内存，同时也能减少代码量；
- 另外，针对不同的CPU和操作系统，你需要调整生成汇编代码的逻辑。

这些实际体验，都进一步验证了20讲中，IR的作用：我们能基于IR对接不同语言的前端，也能对接不同的硬件架构，还能做很多的优化。

既然IR有这些作用，那你可能会问，**IR都是什么样子的呢？有什么特点？如何生成IR呢？**

本节课，我就带你了解IR的特点，认识常见的三地址代码，学会如何把高级语言的代码翻译成IR。然后，我还会特别介绍LLVM的IR，以便后面使用LLVM这个工具。

首先，来看看IR的特征。

介于中间的语言

IR的意思是中间表达方式，它在高级语言和汇编语言的中间，这意味着，它的特征也是处于二者之间的。

与高级语言相比，IR丢弃了大部分高级语言的语法特征和语义特征，比如循环语句、if语句、作用域、面向对象等等，它更像高层次的汇编语言；而相比真正的汇编语言，它又不会有那么多琐碎的、与具体硬件相关的细节。

相信你在学习汇编语言的时候，会发现汇编语言的细节特别多。比如，你要知道很多指令的名字和用法，还要记住很多不同的寄存器。在22讲，我提到，如果你想完整地掌握x86-64架构，还需要接触很多指令集，以及调用约定的细节、内存使用的细节等等（参见Intel的手册）。

仅仅拿指令的数量来说，据有人统计，Intel指令的助记符有981个之多！都记住怎么可能啊。**所以说，汇编语言并不难，而是麻烦。**

IR不会像x86-64汇编语言那么繁琐，但它却包含了足够的细节信息，能方便我们实现优化算法，以及生成针对目标机器的汇编代码。

另外，我在20讲提到，IR有很多种类（AST也是一种IR），每种IR都有不同的特点和用途，有的编译器，甚至要用到几种不同的IR。

我们在后端部分所讲的IR，目的是方便执行各种优化算法，并有利于生成汇编。**这种IR，可以看做是一种高层次的汇编语言，主要体现在：**

- 它可以使用寄存器，但寄存器的数量没有限制；
- 控制结构也跟汇编语言比较像，比如有跳转语句，分成多个程序块，用标签来标识程序块等；
- 使用相当于汇编指令的操作码。这些操作码可以一对一地翻译成汇编代码，但有时一个操作码会对应多个汇编指令。

下面来看看一个典型IR：三地址代码，简称TAC。

认识典型的IR：三地址代码（TAC）

下面是一种常见的IR的格式，它叫做三地址代码（Three Address Code, TAC），它的优点是很简洁，所以适合用来讨论算法：

```
x := y op z    //二元操作
x := op y      //一元操作
```

每条三地址代码最多有三个地址，其中两个是源地址（比如第一行代码的y和z），一个是目的地址（也就是x），每条代码最多有一个操作（op）。

我来举几个例子，带你熟悉一下三地址代码，**这样，你能掌握三地址代码的特点，从高级语言的代码转换生成三地址代码。**

1.基本的算术运算：

```
int a, b, c, d;
a = b + c * d;
```

TAC：

```
t1 := c * d
a  := b + t1
```

t1是新产生的临时变量。当源代码的表达式中包含一个以上的操作符时，就需要引入临时变量，并把原来的一条代码拆成多条代码。

2.布尔值的计算：

```
int a, b;
bool x, y;
x = a * 2 < b;
y = a + 3 == b;
```

TAC：

```
t1 := a * 2;
x  := t1 < b;
t2 := a + 3;
y  := t2 == b;
```

布尔值实际上是用整数表示的，0代表false，非0值代表true。

3.条件语句：

```
int a, b c;
if (a < b )
    c = b;
else
    c = a;
c = c * 2;
```

TAC：

```
  t1 := a < b;
  IfZ t1 Goto L1;
  c := a;
  Goto L2;
L1:
  c := b;
L2:
  c := c * 2;
```

IfZ是检查后面的操作数是否是0，“Z”就是“Zero”的意思。这里使用了标签和Goto语句来进行指令的跳转（Goto相当于x86-64的汇编指令jmp）。

4.循环语句：

```
int a, b;
while (a < b){
  a = a + 1;
}
a = a + b;
```

TAC：

```
L1:
  t1 := a < b;
  IfZ t1 Goto L2;
  a := a + 1;
  Goto L1;
L2:
  a := a + b;
```

三地址代码的规则相当简单，我们可以通过比较简单的转换规则，就能从AST生成TAC。

在课程中，三地址代码主要用来描述优化算法，因为它比较简洁易读，操作（指令）的类型很少，书写方式也符合我们的日常习惯。**不过，我并不用它来生成汇编代码，因为它含有的细节信息还是比较少，**比如，整数是16位的、32位的还是64位的？目标机器的架构和操作系统是什么？生成二进制文件的布局是怎样的等等？

我会用LLVM的IR来承担生成汇编的任务，因为它有能力描述与目标机器（CPU、操作系统）相关的更加具体的信息，准确地生成目标代码，从而真正能够用于生产环境。

在讲这个问题之前，我想先延伸一下，讲讲另外几种IR的格式，主要想帮你开拓思维，如果你的项目需求，恰好能用这种IR实现，到时不妨拿来用一下：

- 首先是四元式。它是与三地址代码等价的另一种表达方式，格式是：（OP，arg1，arg2，result）所以，“a := b + c” 就等价于（+，b，c，a）。
- 另一种常用的格式是逆波兰表达式。它把操作符放到后面，所以也叫做后缀表达式。“b + c”对应的逆波兰表达式是“b c +”；而“a = b + c”对应的逆波兰表达式是“a b c + =”。

逆波兰表达式特别适合用栈来做计算。比如计算“b c +”，先从栈里弹出加号，知道要做加法操作，然后从栈里弹出两个操作数，执行加法运算即可。这个计算过程，跟深度优先的遍历AST是等价的。所以，采用逆波兰表达式，有可能让你用一个很简单的方式就实现公式计算功能，**如果你编写带有公式功能的软件时可以考虑使用它。**而且，从AST生成逆波兰表达式也非常容易。

三地址代码主要是学习算法的工具，或者用于实现比较简单的后端，要实现工业级的后端，充分发挥硬件的性能，你还要学习LLVM的IR。

认识LLVM汇编码

LLVM汇编码（LLVM Assembly），是LLVM的IR。有的时候，我们就简单地称呼它为LLVM语言，因此我们可以把用LLVM汇编码书写的一个程序文件叫做LLVM程序。

我会在下一讲，详细讲解LLVM这个开源项目。本节课作为铺垫，告诉我们在使用LLVM之前，要先了解它的核心——IR。

首先，LLVM汇编码是采用静态单赋值代码形式的。

在三地址代码上再加一些限制，就能得到另一种重要的代码，即静态单赋值代码（Static Single Assignment, SSA），在静态单赋值代码中，一个变量只能被赋值一次，来看个例子。

“y = x1 + x2 + x3 + x4”的普通三地址代码如下：

```
y := x1 + x2;
y := y + x3;
y := y + x4;
```

其中，y被赋值了三次，如果写成SSA的形式，就只能写成下面的样子：

```
t1 := x1 + x2;
t2 := t1 + x3;
y  := t2 + x4;
```

为什么要费力写成这种形式呢，还要为此多添加t1和t2两个临时变量？原因是SSA的形式，体现了精确的“使用-定义”关系。

每个变量很确定地只会被定义一次，然后可以多次使用。这种特点使得基于SSA更容易做数据流分析，而数据流分析又是很多代码优化技术的基础，所以，几乎所有语言的编译器、解释器或虚拟机中都使用了SSA，因为有利于做代码优化。而LLVM的IR，也是采用SSA的形式，也是因为SSA方便做代码优化。

其次，LLVM IR比起三地址代码，有更多的细节信息。比如整型变量的字长、内存对齐方式等等，所以使用LLVM IR能够更准确地翻译成汇编码。

看看下面这段C语言代码：

```
int fun1(int a, int b){
  int c = 10;
  return a + b + c;
}
```

对应的LLLM汇编码如下（这是我在macOS上生成的）：

```
; ModuleID = 'fun1.c'
source_filename = "fun1.c"
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.14.0"
```

```
; Function Attrs: noinline nounwind optnone ssp uwtable
define i32 @fun1(i32, i32) #0 {
  %3 = alloca i32, align 4        //为3个变量申请空间
  %4 = alloca i32, align 4
  %5 = alloca i32, align 4
  store i32 %0, i32* %3, align 4  //参数1赋值给变量1
  store i32 %1, i32* %4, align 4  //参数2赋值给变量2
  store i32 10, i32* %5, align 4  //常量10赋值给变量3
  %6 = load i32, i32* %3, align 4 //
  %7 = load i32, i32* %4, align 4
  %8 = add nsw i32 %6, %7
  %9 = load i32, i32* %5, align 4
  %10 = add nsw i32 %8, %9
  ret i32 %10
}
attributes #0 = { noinline nounwind optnone ssp uwtable "correctly-rounded-divide-sqrt-

!llvm.module.flags = !{!0, !1, !2}
!llvm.ident = !{!3}

!0 = !{i32 2, !"SDK Version", [2 x i32] [i32 10, i32 14]}
!1 = !{i32 1, !"wchar_size", i32 4}
!2 = !{i32 7, !"PIC Level", i32 2}
!3 = !{!"Apple LLVM version 10.0.1 (clang-1001.0.46.4)"}
```

这些代码看上去确实比三地址代码复杂，但还是比汇编精简多了，比如LLVM IR的指令数量连x86-64汇编的十分之一都不到。

我们来熟悉一下里面的元素：

- 模块

LLVM程序是由模块构成的，这个文件就是一个模块。模块里可以包括函数、全局变量和符号表中的条目。链接的时候，会把各个模块拼接到一起，形成可执行文件或库文件。

在模块中，你可以定义目标数据布局（target datalayout）。例如，开头的小写“e”是低字节序（Little Endian）的意思，对于超过一个字节的数据来说，低位字节排放在内存的低地址端，高位字节排放在内存的高地址端。

```
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
```

“target triple”用来定义模块的目标主机，它包括架构、厂商、操作系统三个部分。

```
target triple = "x86_64-apple-macosx10.14.0"
```

- 函数

在示例代码中有一个以define开头的函数的声明，还带着花括号。这有点儿像C语言的写法，比汇编用采取标签来表示一个函数的可读性更好。

函数声明时可以带很多修饰成分，比如链接类型、调用约定等。如果不写，缺省的链接类型是external的，也就是可以像23讲中做链接练习的那样，暴露出来被其他模块链接。调用约定也有很多种选择，缺省是“ccc”，也就是C语言的调用约定（C Calling Convention），而“swiftcc”则是swift语言的调用约定。**这些信息都是生成汇编时所需要的。**

示例中函数fun1还带有“#0”的属性值，定义了许多属性。这些也是生成汇编时所需要的。

- 标识符

分为全局的（Glocal）和本地的（Local）：全局标识符以@开头，包括函数和全局变量，前面代码中的@fun1就是；本地标识符以%开头。

有的标识符是有名字的，比如@fun1或%a，有的是没有名字的，用数字表示就可以了，如%1。

- 操作码

alloca、store、load、add、ret这些，都是操作码。它们的含义是：

操作码	含义
alloca	栈上分配空间
store	写入内存
load	从内存中读取
add	加法运算
ret	从过程中返回

它们跟我们之前学到的汇编很相似。但是似乎函数体中的代码有点儿长。怎么一个简单的“a+b+c”就翻译成了10多行代码，还用到了那么多临时变量？不要担心，**这只是完全没经过优化的格式，**带上优化参数稍加优化以后，它就会被精简成下面的样子：

```
define i32 @fun1(i32, i32) local_unnamed_addr #0 {
  %3 = add i32 %0, 10
  %4 = add i32 %3, %1
  ret i32 %4
}
```

- 类型系统

汇编是无类型的。如果你用add指令，它就认为你操作的是整数。而用fadd（或addss）指令，就认为你操作的是浮点数。这样会有类型不安全的风险，把整型当浮点数用了，造成的后果是计算结果完全错误。

LLVM汇编则带有一个类型系统。它能避免不安全的数据操作，并且有助于优化算法。这个类型系统包括**基础数据类型、函数类型和void类型。**

LLVM的基础数据类型				
基础数据类型	举例			
用iN表示各种长度的整型	i1：是1个比特的整型		i32：32位的整型	
多种精度的浮点型	half： 16位浮点型	float： 32位浮点型	double： 64位浮点型	fp128： 128位浮点型
指针 （用*表示，用法很像C语言的指针）	[4 x i32]*： 一个指向4个i32整数的数组的指针	i32 (i32*) *： 一个函数指针，该函数有一个参数是i32指针，返回一个i32值		
向量	如<4 x float>代表4个浮点数的向量			
数组	如[4 x i32]代表4个i32整数的数组			
结构体 （有点像C语言的结构体，或java语言的对象）	普通结构体： {float, i32 (i32) *}，两个元素的结构体， 一个是浮点数，一个是函数指针。		紧凑结构体： <{i8, i32}>，比普通结构体多了尖括号， 它的元素在存储时是紧挨着的，不考虑内存对齐，因此这个结构体是占40个比特，也就是5个字节。	
其他	标签类型	Token类型	元数据类型	

函数类型是包括对返回值和参数的定义，比如：i32 (i32)；

void类型不代表任何值，也没有长度。

- 全局变量和常量

在LLVM汇编中可以声明全局变量。全局变量所定义的内存，是在编译时就分配好了的，而不是在运行时，例如下面这句定义了一个全局变量C：

```
@c = global i32 100, align 4
```

你也可以声明常量，它的值在运行时不会被修改：

```
@c = constant i32 100, align 4
```

- 元数据

在代码中你还看到以“!”开头的一些句子，这些是元数据。这些元数据定义了一些额外的信息，提供给优化器和代码生成器使用。

- 基本块

函数中的代码会分成一个个的基本块，可以用标签（Label）来标记一个基本块。下面这段代码有4个基本块，其中第一个块有一个缺省的名字“entry”，也就是作为入口的基本块，这个基本块你不给它标签也可以。

```
define i32 @bb(i32) #0 {
  %2 = alloca i32, align 4
  %3 = alloca i32, align 4
  store i32 %0, i32* %3, align 4
  %4 = load i32, i32* %3, align 4
  %5 = icmp sgt i32 %4, 0
  br i1 %5, label %6, label %9

; <label>:6:                                      ; preds = %1
  %7 = load i32, i32* %3, align 4
  %8 = mul nsw i32 %7, 2
  store i32 %8, i32* %2, align 4
  br label %12

; <label>:9:                                      ; preds = %1
  %10 = load i32, i32* %3, align 4
  %11 = add nsw i32 %10, 3
  store i32 %11, i32* %2, align 4
  br label %12

; <label>:12:                                     ; preds = %9, %6
  %13 = load i32, i32* %2, align 4
  ret i32 %13
}
```

这段代码实际上相当于下面这段C语言的代码：

```
int bb(int b){
    if (b > 0)
        return b * 2;
    else
        return b + 3;
}
```

每个基本块是一系列的指令。我们分析一下标签为9的基本块，**让你熟悉一下基本块和LLVM指令的特点：**

第一行（%10 = load i32, i32* %3, align 4）的含义是：把3号变量（32位整型）从内存加载到寄存器，叫做10号变量，其中，内存对齐是4字节。

我在这里延伸一下，我们在内存里存放数据的时候，有时会从2、4、8个字节的整数倍地址开始存。有些汇编指令要求必须从这样对齐的地址来取数据。另一些指令没做要求，但如果是不对齐的，比如是从0x03地址取数据，就要花费更多的时钟周期。但缺点是，内存对齐会浪费内存空间。

第一行是整个基本块的唯一入口，从其他基本块跳转过来的时候，只能跳转到这个入口行，不能跳转到基本块中的其他行。

第二行（%11 = add nsw i32 %10, 3）的含义是：把10号变量（32位整型）加上3，保存到11号变量，其中nsw是加法计算时没有符号环绕（No Signed Wrap）的意思。它的细节你可以查阅“LLVM语言参考手册”。

第三行（store i32 %11, i32* %2, align 4）的含义是：把11号变量（32位整型）存入内存中的2号变量，内存对齐4字节。

第四行（br label %12）的含义是：跳转到标签为12的代码块。其中，br指令是一条终结指令。终结指令要么是跳转到另一个基本块，要么是从函数中返回（ret指令），基本块的最后一行必须是一条终结指令。

最后我要强调，从其他基本块不可以跳转到入口基本块，也就是函数中的第一个基本块。这个规定也是有利于做数据优化。

以上就是对LLVM汇编码的概要介绍（更详细的信息了解可以参见“LLVM语言参考手册”）。

这样，你实际上就可以用LLVM汇编码来编写程序了，或者将AST翻译成LLVM汇编码。听上去有点让人犯怵，因为LLVM汇编码的细节也相当不少，好在，LLVM提供了一个IR生成的API（应用编程接口），可以让我们更高效、更准确地生成IR。

课程小结

IR是我们后续做代码优化、汇编代码生成的基础，在本节课中，我想让你明确的要点如下：

1.三地址代码是很常见的一种IR，包含一个目的地址、一个操作符和至多两个源地址。它等价于四元式。我们在27讲和28讲中的优化算法，会用三地址代码来讲解，这样比较易于阅读。

2.LLVM IR的第一个特点是静态单赋值（SSA），也就是每个变量（地址）最多被赋值一次，它这种特性有利于运行代码优化算法；第二个特点是带有比较多的细节，方便我们做优化和生成高质量的汇编代码。

通过本节课，你应该对于编译器后端中常常提到的IR建立了直观的认识，相信通过接下来的练习，你一定会消除对IR的陌生感，让它成为你得心应手的好工具！

一课一思

我们介绍了IR的特点和几种基本的IR，在你的领域，比如人工智能领域，你了解其他的IR吗？它带来了什么好处？欢迎分享你的经验和观点。

最后，感谢你的阅读，如果这篇文章让你有所收获，也欢迎你将它分享给更多的人。

上一页 下一页