

24 | 中间代码：兼容不同的语言和硬件

2019-10-18 宫文学 来自北京

《编译原理之美》



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），它的优点是简洁，所以适合用来讨论算法：

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


 复制代码

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：

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

 复制代码


TAC:

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

 复制代码


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：

 复制代码

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

TAC:

 复制代码

```
1 t1 := a * 2;  
2 x  := t1 < b;  
3 t2 := a + 3;  
4 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：

 复制代码

```
1 int a, b c;  
2 if (a < b )  
3     c = b;  
4 else  
5     c = a;  
6 c = c * 2;
```

TAC:


 复制代码

```
1  t1 := a < b;
2  IfZ t1 Goto L1;
3  c := a;
4  Goto L2;
5  L1:
6  c := b;
7  L2:
8  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:

```
1  int a, b;
2  while (a < b){
3    a = a + 1;
4  }
5  a = a + b;
```

 复制代码

TAC:

```
1  L1:
2  t1 := a < b;
3  IfZ t1 Goto L2;
4  a := a + 1;
5  Goto L1;
6  L2:
7  a := a + b;
```

 复制代码

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：

```
1 y := x1 + x2;  
2 y := y + x3;  
3 y := y + x4;
```

 复制代码

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：

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

 复制代码

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：

[复制代码](#)

```
1 int fun1(int a, int b){
2     int c = 10;
3     return a + b + c;
4 }
```

对应的 LLVM 汇编码如下（这是我在 macOS 上生成的）：

[复制代码](#)

```
1 ; ModuleID = 'fun1.c'
2 source_filename = "fun1.c"
3 target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
4 target triple = "x86_64-apple-macosx10.14.0"
5 ; Function Attrs: noinline nounwind optnone ssp uwtable
6 define i32 @fun1(i32, i32) #0 {
7     %3 = alloca i32, align 4          //为3个变量申请空间
8     %4 = alloca i32, align 4
9     %5 = alloca i32, align 4
10    store i32 %0, i32* %3, align 4    //参数1赋值给变量1
11    store i32 %1, i32* %4, align 4    //参数2赋值给变量2
12    store i32 10, i32* %5, align 4    //常量10赋值给变量3
13    %6 = load i32, i32* %3, align 4 //
14    %7 = load i32, i32* %4, align 4
15    %8 = add nsw i32 %6, %7
16    %9 = load i32, i32* %5, align 4
17    %10 = add nsw i32 %8, %9
18    ret i32 %10
19 }
20 attributes #0 = { noinline nounwind optnone ssp uwtable "correctly-rounded-divide
21
22 !llvm.module.flags = !{!0, !1, !2}
23 !llvm.ident = !{!3}
24
25 !0 = !{i32 2, !"SDK Version", [2 x i32] [i32 10, i32 14]}
26 !1 = !{i32 1, !"wchar_size", i32 4}
27 !2 = !{i32 7, !"PIC Level", i32 2}
28 !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。

 复制代码

```
1 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。

 复制代码

```
1 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（Global）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：

 复制代码

```
1 define i32 @fun1(i32, i32) local_unnamed_addr #0 {
2   %3 = add i32 %0, 10
3   %4 = add i32 %3, %1
4   ret i32 %4
5 }
```

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：

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

[复制代码](#)

你也可以声明常量，它的值在运行时不会被修改：

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

[复制代码](#)

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。

```
1 define i32 @bb(i32) #0 {
2     %2 = alloca i32, align 4
3     %3 = alloca i32, align 4
4     store i32 %0, i32* %3, align 4
5     %4 = load i32, i32* %3, align 4
6     %5 = icmp sgt i32 %4, 0
7     br i1 %5, label %6, label %9
8
9     ; <label>:6:                                ; preds = %1
10    %7 = load i32, i32* %3, align 4
11    %8 = mul nsw i32 %7, 2
12    store i32 %8, i32* %2, align 4
```

[复制代码](#)

```

13    br label %12
14
15 ; <label>:9:                                ; preds = %1
16    %10 = load i32, i32* %3, align 4
17    %11 = add nsw i32 %10, 3
18    store i32 %11, i32* %2, align 4
19    br label %12
20
21 ; <label>:12:                                ; preds = %9, %6
22    %13 = load i32, i32* %2, align 4
23    ret i32 %13
24 }

```

这段代码实际上相当于下面这段 C 语言的代码：

 复制代码

```

1 int bb(int b){
2     if (b > 0)
3         return b * 2;
4     else
5         return b + 3;
6 }

```

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。

© 版权归极客邦科技所有，未经许可不得传播售卖。页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

精选留言 (8)



minghu6

2021-05-30

AST也是一种IR说得不严谨, AST显然是语义相关的, 要把AST转成类似LISP形式的树, 在这个过程中结构化地抽象掉语义, 这才算是一种IR 作者可能混淆了这两个概念.

作者回复: 你专注了一个有意思的问题。对概念的分析，往往非常有用。概念是一切的基础。值得我认真回答一下。

对于IR的定义，有狭义和窄义两种不同的理解。

从狭义的角度，IR是在AST之后，用于做优化的数据结构，这是我们大部分场景里提到IR时候的含义。我们用狭义的的定义的时候，通常字节码也不算IR，因为它已经算是目标代码了，也主要不是用于优化的目的。

从广义的角度，IR是处于源代码和目标代码之间的各种能够表示代码的数据结构。

在wikipedia中，对AST有这么一段描述：

Abstract syntax trees are data structures widely used in compilers to represent the structure of program code. An AST is usually the result of the syntax analysis phase of a compiler. It often serves as an intermediate representation of the program through several stages that the compiler requires, and has a strong impact on the final output of the compiler.

这里面，就指出AST是一种“intermediate representation”。而如果你去查IR的词条，里面也会有对AST的引用，跟字节码并列。

再加深一下理解：

1.基于AST能否做优化？

是可以的。在看JDK的编译器代码的时候，就能发现很多基于AST的优化。优化是不嫌早的。而因为JDK的前端编译器只是生成字节码而已，所以很多优化都是在AST上去做。

2.IR是否可以树状结构（甚至直接就基于AST）？

我们把IR根据抽象层次的高低，分为HIR、MIR和LIR。对于HIR来说，不少是采用树状结构的，也保留了像循环、分枝判断这样的高层次结构。

根据方舟编译器早期释放出来的代码和文档，其Maple IR就是一个树状结构的。随着优化的进程，树的节点逐步变成低级别的操作，树的高度也会降低。不过，我不知道方舟编译器最新的版本是否还会保持这个设计。我最近会问问内部相关的人员。

3.字节码算不算IR？

我的观点：广义上算，因为它毕竟是中间格式，基于字节码还可以进一步做优化，比如Web Assembly就是这个设计。运行时读入Web Assembly以后，会启动JIT的过程，在这个过程中可以做优化。

狭义上不算，因为它已经是某个虚拟机的目标代码。

4.AST能否作为目标代码？

这是另一个极端。AST通常并不用于执行程序。但是，对于特别简单的脚本语言，或者一个公式解释器，基于AST运行也无不可。

5.中间代码是否包含语义？

不同层次的IR在语义上是等价的。虽然从AST到HIR一直到LIR使用的操作是不同的，语义的抽象程度是从高到低的，但它们是等价的。不同层次的IR适合做不同的优化。

共 4 条评论 >



沉淀的梦想

2019-10-23

对与LLVM的SSA有点不太理解，`=`和`store`指令的区别是什么呢？为什么`store`就不算赋值呢，看llvm字节码里经常`store`到同一个寄存器多次，这是不是有点违背SSA？

作者回复：store是把寄存器的内容，写到内存中。与之对应的指令是load。

是的，涉及内存访问的时候，是可以违背SSA的。这点你看得很细。非常值得表扬！我在第26讲提到了这个知识点。你提前就发现了。



独钓寒江雪

2020-06-02

老师好，文中说，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。那么，如果出现递归调用，是怎么处理的呢？

作者回复：（我对另一个同学的回答，这里拷贝一份）

递归相当于调用另一个函数。所有的参数、本地变量，要再来一份，也就是要创建一个新的活动记录（栈帧）。

不过，对于递归调用中的尾递归，编译器是可以做优化的，也就是不用再进行一次新的函数调用，从而节省了系统资源的开销。



👍 2



coconut

2021-07-15

IfZ t1 Goto L1;

这个是不是不能算作tac，因为包含了ifZ、Goto两个操作



Geek_e8d55e

2021-05-26

nsw 无符号环绕怎么理解呢？



lion_fly

2020-12-11

Static Single Assignment: SSA（静态单赋值）

Three Address Code, TAC（三地址代码）



ztxc

2020-07-03

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。

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。

老师好，这两段不太理解。第一段是说跳转的入口基本块。第二段说不可以跳转。第二段的意思是说从本基本块内部不能跳转到本基本块的入口基本块吗？

作者回复: 不是。

第一段的意思: 每个基本块中的代码都是从第一行执行到最后一行, 不可能从别的地方跳转到这个基本块中间的一行, 然后再继续执行, 也不可能执行到中间就跳转出去。

第二段的意思: 入口基本块是CFG中的一个特殊基本块, 函数开始运行时就进入这个基本块。



超越杨超越

2020-05-06

老师的demo, if条件语句的ir是不是写错了? 看起来有些奇怪。

共 2 条评论 >

