

## 26 | 生成IR：实现静态编译的语言

2019-10-23 宫文学 来自北京

《编译原理之美》



目前来讲，你已经初步了解了 LLVM 和它的 IR，也能够使用它的命令行工具。**不过，我们还是要通过程序生成 LLVM 的 IR**，这样才能复用 LLVM 的功能，从而实现一门完整的语言。

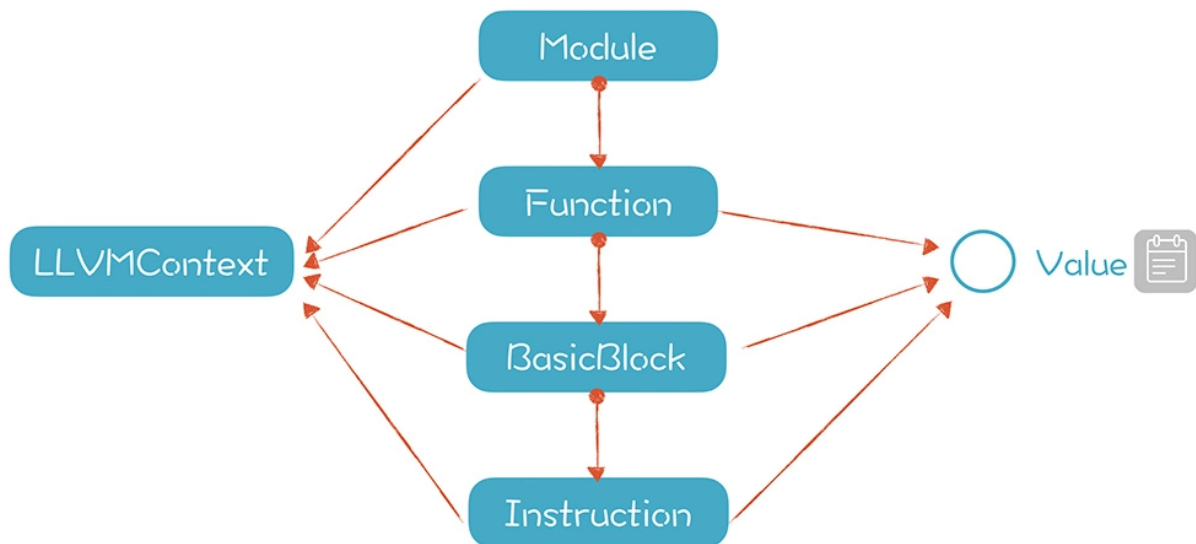
不过，如果我们要像前面生成汇编语言那样，通过字符串拼接来生成 LLVM 的 IR，除了要了解 LLVM IR 的很多细节之外，代码一定比较啰嗦和复杂，因为字符串拼接不是结构化的方法，所以，最好用一个定义良好的数据结构来表示 IR。

好在 LLVM 项目已经帮我们考虑到了这一点，它提供了代表 LLVM IR 的一组对象模型，我们只要生成这些对象，就相当于生成了 IR，这个难度就低多了。而且，LLVM 还提供了一个工具类，IRBuilder，我们可以利用它，进一步提升创建 LLVM IR 的对象模型的效率，让生成 IR 的过程变得更加简单！

接下来，就让我们先来了解 LLVM IR 的对象模型。

## LLVM IR 的对象模型

LLVM 在内部有用 C++ 实现的对象模型，能够完整表示 LLVM IR，当我们把字节码读入内存时，LLVM 就会在内存中构建出这个模型。只有基于这个对象模型，我们才可以做进一步的工作，包括代码优化，实现即时编译和运行，以及静态编译生成目标文件。**所以说，这个对象模型是 LLVM 运行时的核心。**



IR 对象模型的头文件在 [include/llvm/IR](#) 目录下，其中最重要的类包括：

### Module (模块)

Module 类聚合了一个模块中的所有数据，它可以包含多个函数。你可以通过 `Module::iterator` 来遍历模块中所有的函数。它也包含了一个模块的全局变量。

### Function (函数)

Function 包含了与函数定义 (definition) 或声明 (declaration) 有关的所有对象。函数定义包含了函数体，而函数声明，则仅仅包含了函数的原型，它是在其他模块中定义的，在本模块中使用。

你可以通过 `getArgumentList()` 方法来获得函数参数的列表，也可以遍历函数体中的所有基本块，这些基本块会形成一个 CFG (控制流图)。

```
1 //函数声明，没有函数体。这个函数是在其他模块中定义的，在本模块中使用
2 declare void @foo(i32)
3
4 //函数定义，包含函数体
5 define i32 @fun3(i32 %a) {
6     %calltmp1 = call void @foo(i32 %a) //调用外部函数
7     ret i32 10
8 }
```

## BasicBlock (基本块)

BasicBlock 封装了一系列的 LLVM 指令，你可以借助 `begin()/end()` 模式遍历这些指令，还可以通过 `getTerminator()` 方法获得最后一条指令（也就是终结指令）。你还可以用到几个辅助方法在 CFG 中导航，比如获得某个基本块的前序基本块。

## Instruction (指令)

Instruction 类代表了 LLVM IR 的原子操作（也就是一条指令），你可以通过 `getOpcode()` 来获得它代表的操作码，它是一个 `llvm::Instruction` 枚举值，你可以通过 `op_begin()` 和 `op_end()` 方法对获得这个指令的操作数。

## Value (值)

Value 类代表一个值。在 LLVM 的内存 IR 中，如果一个类是从 Value 继承的，意味着它定义了一个值，其他方可以去使用。函数、基本块和指令都继承了 Value。

## LLVMContext (上下文)

这个类代表了 LLVM 做编译工作时的一个上下文，包含了编译工作中的一些全局数据，比如各个模块用到的常量和类型。

这些内容是 LLVM IR 对象模型的主要部分，我们生成 IR 的过程，就是跟这些类打交道，其他一些次要的类，你可以在阅读和编写代码的过程中逐渐熟悉起来。

接下来，就让我们用程序来生成 LLVM 的 IR。


## 尝试生成 LLVM IR

我刚刚提到的每个 LLVM IR 类，都可以通过程序来构建。那么，为下面这个 fun1() 函数生成 IR，应该怎么办呢？

 复制代码

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

**第一步**，我们可以来生成一个 LLVM 模块，也就是顶层的 IR 对象。


 复制代码

```
1 Module *mod = new Module("fun1.ll", TheModule);
```

**第二步**，我们继续在模块中定义函数 fun1，因为模块最主要的构成要素就是各个函数。

不过在定义函数之前，要先定义函数的原型（或者叫函数的类型）。函数的类型，我们在前端讲过：如果两个函数的返回值相同，并且参数也相同，这两个函数的类型是相同的，这样就可以做函数指针或函数型变量的赋值。示例代码的函数原型是：返回值是 32 位整数，参数是两个 32 位整数。

有了函数原型以后，就可以使用这个函数原型定义一个函数。我们还可以为每个参数设置一个名称，便于后面引用这个参数。

 复制代码

```
1 //函数原型
2 vector<Type*> argTypes(2, Type::getInt32Ty(TheContext));
3 FunctionType *fun1Type = FunctionType::get(Type::getInt32Ty(TheContext), //返回值是
4     argTypes, //两个整型参数
5     false);    //不是变长参数
6
```

```


7 //函数对象
8 Function *fun = Function::Create(fun1Type,
9     Function::ExternalLinkage,    //链接类型
10    "fun2",                        //函数名称
11    TheModule.get());             //所在模块
12
13 //设置参数名称
14 string argNames[2] = {"a", "b"};
15 unsigned i = 0;
16 for (auto &arg : fun->args()){
17     arg.setName(argNames[i++]);
18 }

```

**这里你需要注意，代码中是如何使用变量类型的。**所有的基础类型都是提前定义好的，可以通过 Type 类的 getXXXTy() 方法获得（我们使用的是 Int32 类型，你还可以获得其他类型）。

### 第三步，创建一个基本块。

这个函数只有一个基本块，你可以把它命名为“entry”，也可以不给它命名。在创建了基本块之后，我们用了一个辅助类 IRBuilder，设置了一个插入点，后序生成的指令会插入到这个基本块中（IRBuilder 是 LLVM 为了简化 IR 生成过程所提供的的一个辅助类）。

 复制代码

```

1 //创建一个基本块
2 BasicBlock *BB = BasicBlock::Create(TheContext, //上下文
3     "",    //基本块名称
4     fun);  //所在函数
5 Builder.SetInsertPoint(BB); //设置指令的插入点

```

### 第四步，生成"a+b"表达式所对应的 IR，插入到基本块中。

a 和 b 都是函数 fun 的参数，我们把它取出来，分别赋值给 L 和 R（L 和 R 是 Value）。然后用 IRBuilder 的 CreateAdd() 方法，生成一条 add 指令。这个指令的计算结果存放在 addtemp 中。

 复制代码

```

1 //把参数变量存到NamedValues里面备用

```



```

2 NamedValues.clear();
3 for (auto &Arg : fun->args())
4     NamedValues[Arg.getName()] = &Arg;
5
6 //做加法
7 Value *L = NamedValues["a"];
8 Value *R = NamedValues["b"];
9 Value *addtmp = Builder.CreateAdd(L, R);

```

**第五步**，利用刚才获得的 addtmp 创建一个返回值。


 复制代码

```

1 //返回值
2 Builder.CreateRet(addtmp);

```

**最后一步**，检查这个函数的正确性。这相当于是做语义检查，比如，基本块的最后一个语句就必须是一个正确的返回指令。


 复制代码

```

1 //验证函数的正确性
2 verifyFunction(*fun);

```

完整的代码我也提供给你，放在 [codegen\\_fun1\(\)](#) 里了，你可以看一下。我们可以调用这个方法，然后打印输出生成的 IR：

 复制代码

```

1 Function *fun1 = codegen_fun1();    //在模块中生成Function对象
2 TheModule->print(errs(), nullptr);  //在终端输出IR

```

生成的 IR 如下：

 复制代码

```

1 ; ModuleID = 'llvmdemo'
2 source_filename = "llvmdemo"

```

```
3 define i32 @fun1(i32 %a, i32 %b) {
4   %1 = add i32 %a, %b
5   ret i32 %1
6 }
```

这个例子简单，过程直观，只有一个加法运算，而我建议你在这个过程中注意每个 IR 对象都是怎样被创建的，在大脑中想象出整个对象结构。

为了熟悉更多的 API，接下来，我再带你生成一个稍微复杂一点儿的，带有 if 语句的 IR。然后来看一看，函数中包含多个基本块的情况。

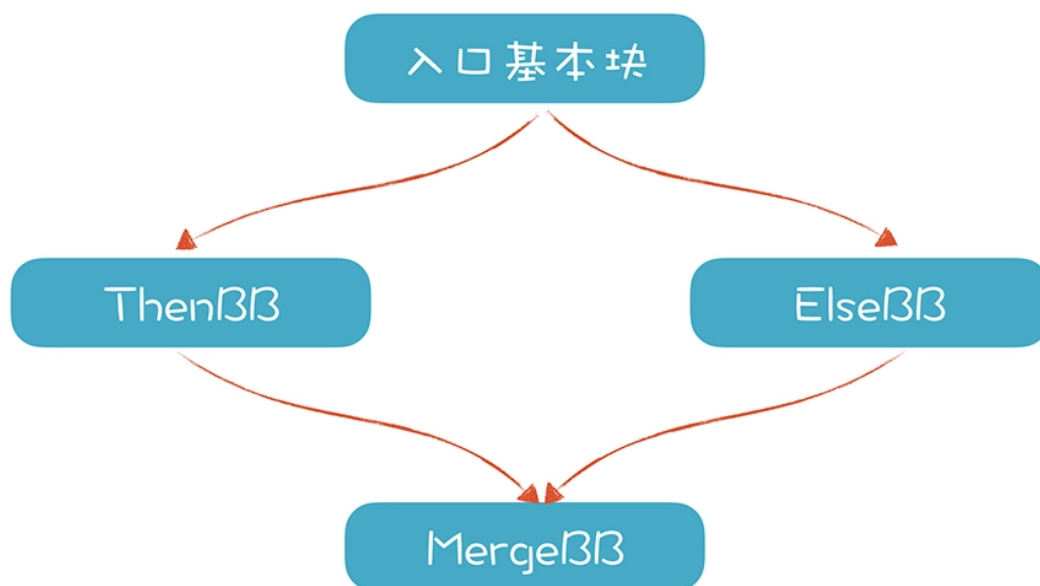
## 支持 if 语句

具体说，我们要为下面的一个函数生成 IR（函数有一个参数 a，当 a 大于 2 的时候，返回 2；否则返回 3）。


```
1 int fun_ifstmt(int a)
2   if (a > 2)
3     return 2;
4   else
5     return 3;
6 }
```

 复制代码

这样的函数，需要包含 4 个基本块：**入口基本块**、**Then 基本块**、**Else 基本块**和 **Merge 基本块**。控制流图（CFG）是先分开，再合并，像下面这样：




**在入口基本块中**，我们要计算 “a>2” 的值，并根据这个值，分别跳转到 ThenBB 和 ElseBB。这里，我们用到了 IRBuilder 的 CreateICmpUGE() 方法（UGE 的意思，是“不大于等于”，也就是小于）。这个指令的返回值是一个 1 位的整型，也就是 int1。

 复制代码

```
1 //计算a>2
2 Value * L = NamedValues["a"];
3 Value * R = ConstantInt::get(TheContext, APInt(32, 2, true));
4 Value * cond = Builder.CreateICmpUGE(L, R, "cmptmp");
```

接下来，我们创建另外 3 个基本块，并用 IRBuilder 的 CreateCondBr() 方法创建条件跳转指令：当 cond 是 1 的时候，跳转到 ThenBB，0 的时候跳转到 ElseBB。


 复制代码

```
1 BasicBlock *ThenBB = BasicBlock::Create(TheContext, "then", fun);
2 BasicBlock *ElseBB = BasicBlock::Create(TheContext, "else");
3 BasicBlock *MergeBB = BasicBlock::Create(TheContext, "ifcont");
4 Builder.CreateCondBr(cond, ThenBB, ElseBB);
```

**如果你细心的话**，可能会发现，在创建 ThenBB 的时候，指定了其所在函数是 fun，而其他两个基本块没有指定。这是因为，我们接下来就要为 ThenBB 生成指令，所以先加到 fun 中。




之后，再顺序添加 ElseBB 和 MergeBB 到 fun 中。

 复制代码

```
1 //ThenBB
2 Builder.SetInsertPoint(ThenBB);
3 Value *ThenV = ConstantInt::get(TheContext, APInt(32, 2, true));
4 Builder.CreateBr(MergeBB);
5
6 //ElseBB
7 fun->getBasicBlockList().push_back(ElseBB); //把基本块加入到函数中
8 Builder.SetInsertPoint(ElseBB);
9 Value *ElseV = ConstantInt::get(TheContext, APInt(32, 3, true));
10 Builder.CreateBr(MergeBB);
```

在 ThenBB 和 ElseBB 这两个基本块的代码中，我们分别计算出了两个值：ThenV 和 ElseV。它们都可能是最后的返回值，但具体采用哪个，还要看实际运行时，控制流走的是 ThenBB 还是 ElseBB。这就需要用到 phi 指令，它完成了根据控制流来选择合适的值的任务。

 复制代码

```
1 //MergeBB
2 fun->getBasicBlockList().push_back(MergeBB);
3 Builder.SetInsertPoint(MergeBB);
4 //PHI节点：整型，两个候选值
5 PHINode *PN = Builder.CreatePHI(Type::getInt32Ty(TheContext), 2);
6 PN->addIncoming(ThenV, ThenBB); //前序基本块是ThenBB时，采用ThenV
7 PN->addIncoming(ElseV, ElseBB); //前序基本块是ElseBB时，采用ElseV
8
9 //返回值
10 Builder.CreateRet(PN);
```

从上面这段代码中你能看出，在 if 语句中，phi 指令是关键。因为当程序的控制流经过多个基本块，每个基本块都可能改变某个值的时候，通过 phi 指令可以知道运行时实际走的是哪条路径，从而获得正确的值。

最后生成的 IR 如下，其中的 phi 指令指出，如果前序基本块是 then，取值为 2，是 else 的时候取值为 3。

[复制代码](#)

```
1 define i32 @fun_ifstmt(i32 %a) {
2   %cmptmp = icmp uge i32 %a, 2
3   br i1 %cmptmp, label %then, label %else
4
5   then:                                ; preds = %0
6   br label %ifcont
7
8   else:                                ; preds = %0
9   br label %ifcont
10
11  ifcont:                               ; preds = %else, %then
12   %1 = phi i32 [ 2, %then ], [ 3, %else ]
13   ret i32 %1
14 }
```

其实循环语句也跟 if 语句差不多，因为它们都是要涉及到多个基本块，要用到 phi 指令，所以一旦你会写 if 语句，肯定就会写循环语句的。

## 支持本地变量

在写程序的时候，本地变量是必不可少的一个元素，所以，我们趁热打铁，把刚才的示例程序变化一下，用本地变量 b 保存 ThenBB 和 ElseBB 中计算的值，借此学习一下 LLVM IR 是如何支持本地变量的。

改变后的示例程序如下：


[复制代码](#)

```
1 int fun_localvar(int a)
2   int b = 0;
3   if (a > 2)
4     b = 2;
5   else
6     b = 3;
7   return b;
8 }
```

其中，函数有一个参数 a，一个本地变量 b：如果 a 大于 2，那么给 b 赋值 2；否则，给 b 赋值 3。最后的返回值是 b。

**现在挑战来了**，在这段代码中，b 被声明了一次，赋值了 3 次。我们知道，LLVM IR 采用的是 SSA 形式，也就是每个变量只允许被赋值一次，那么对于多次赋值的情况，我们该如何生成 IR 呢？


其实，LLVM 规定了对寄存器只能做单次赋值，而对内存中的变量，是可以多次赋值的。对于“int b = 0;”，我们用下面几条语句生成 IR：

 复制代码

```
1 //本地变量b
2 AllocaInst *b = Builder.CreateAlloca(Type::getInt32Ty(TheContext), nullptr, "b");
3 Value* initValue = ConstantInt::get(TheContext, APInt(32, 0, true));
4
5 Builder.CreateStore(initValue, b);
```


上面这段代码的含义是：首先用 CreateAlloca() 方法，在栈中申请一块内存，用于保存一个 32 位的整型，接着，用 CreateStore() 方法生成一条 store 指令，给 b 赋予初始值。

上面几句生成的 IR 如下：

 复制代码

```
1 %b = alloca i32
2 store i32 0, i32* %b
```

接着，我们可以在 ThenBB 和 ElseBB 中，分别对内存中的 b 赋值：

 复制代码


```
1 //ThenBB
2 Builder.SetInsertPoint(ThenBB);
3 Value *ThenV = ConstantInt::get(TheContext, APInt(32, 2, true));
4 Builder.CreateStore(ThenV, b);
5 Builder.CreateBr(MergeBB);
6
```

```

7 //ElseBB
8 fun->getBasicBlockList().push_back(ElseBB);
9 Builder.SetInsertPoint(ElseBB);
10 Value *ElseV = ConstantInt::get(TheContext, APInt(32, 3, true));
11 Builder.CreateStore(ElseV, b);
12 Builder.CreateBr(MergeBB);

```

最后，在 MergeBB 中，我们只需要返回 b 就可以了：

 复制代码

```

1 //MergeBB
2 fun->getBasicBlockList().push_back(MergeBB);
3 Builder.SetInsertPoint(MergeBB);
4
5 //返回值
6 Builder.CreateRet(b);

```

最后生成的 IR 如下：

 复制代码

```

1 define i32 @fun_ifstmt.1(i32 %a) {
2     %b = alloca i32
3     store i32 0, i32* %b
4     %cmptmp = icmp uge i32 %a, 2
5     br i1 %cmptmp, label %then, label %else
6
7 then:                                     ; preds = %0
8     store i32 2, i32* %b
9     br label %ifcont
10
11 else:                                    ; preds = %0
12     store i32 3, i32* %b
13     br label %ifcont
14
15 ifcont:                                 ; preds = %else, %then
16     ret i32* %b
17 }

```

当然，使用内存保存临时变量的性能比较低，但我们可以很容易通过优化算法，把上述代码从使用内存的版本，优化成使用寄存器的版本。

通过上面几个示例，现在你已经学会了生成基本的 IR，包括能够支持本地变量、加法运算、if 语句。那么这样生成的 IR 能否正常工作呢？我们需要把这些 IR 编译和运行一下才知道。

## 编译并运行程序


现在已经能够在内存中建立 LLVM 的 IR 对象了，包括模块、函数、基本块和各种指令。LLVM 可以即时编译并执行这个 IR 模型。

我们先创建一个不带参数的 `__main()` 函数作为入口。同时，我会借这个例子延伸讲一下函数的调用。我们在前面声明了函数 `fun1`，现在在 `__main()` 函数中演示如何调用它。

 复制代码

```
1  Function * codegen_main(){
2      //创建main函数
3      FunctionType *mainType = FunctionType::get(Type::getInt32Ty(TheContext), false
4      Function *main = Function::Create(mainType, Function::ExternalLinkage, "__mai
5
6      //创建一个基本块
7      BasicBlock *BB = BasicBlock::Create(TheContext, "", main);
8      Builder.SetInsertPoint(BB);
9
10     //设置参数的值
11     int argValues[2] = {2, 3};
12     std::vector<Value *> ArgsV;
13     for (unsigned i = 0; i<2; ++i) {
14         Value * value = ConstantInt::get(TheContext, APInt(32,argValues[i],true))
15         ArgsV.push_back(value);
16         if (!ArgsV.back())
17             return nullptr;
18     }
19
20     //调用函数fun1
21     Function *callee = TheModule->getFunction("fun1");
22     Value * rtn = Builder.CreateCall(callee, ArgsV, "calltmp");
23
24     //返回值
25     Builder.CreateRet(rtn);
26     return main;
27 }
```


调用函数时，我们首先从模块中查找出名称为 fun1 的函数，准备好参数值，然后通过 IRBuilder 的 CreateCall() 方法来生成函数调用指令。最后生成的 IR 如下：

 复制代码

```
1 define i32 @__main() {
2   %calltmp = call i32 @fun1(i32 2, i32 3)
3   ret i32 %calltmp3
4 }
```


接下来，我们调用即时编译的引擎来运行 \_\_main 函数（与 JIT 引擎有关的代码，放到了 DemoJIT.h 中，你现在可以暂时不关心它的细节，留到以后再去了解）。使用这个 JIT 引擎，我们需要做几件事情：

### 1. 初始化与目标硬件平台有关的设置。

 复制代码

```
1 InitializeNativeTarget();
2 InitializeNativeTargetAsmPrinter();
3 InitializeNativeTargetAsmParser();
```

### 2. 把创建的模型加入到 JIT 引擎中，找到 \_\_main() 函数的地址（整个过程跟 C 语言中使用函数指针来执行一个函数没有太大区别）。

 复制代码


```
1 auto H = TheJIT->addModule(std::move(TheModule));
2
3 //查找__main函数
4 auto main = TheJIT->findSymbol("__main");
5
6 //获得函数指针
7 int32_t (*FP)() = (int32_t (*)(intptr_t))cantFail(main.getAddress());
8
9 //执行函数
10 int rtn = FP();
```



```
11 //打印执行结果
12 fprintf(stderr, "__main: %d\n", rtn);
13
```


3. 程序可以成功执行，并打印 `__main` 函数的返回值。

**既然已经演示了如何调用函数，在这里，我给你揭示 LLVM 的一个惊人的特性：**我们可以在 LLVM IR 里，调用本地编写的函数，比如编写一个 `foo()` 函数，用来打印输出一些信息：

 复制代码

```
1 void foo(int a){
2     printf("in foo: %d\n",a);
3 }
```

然后我们就可以在 `__main` 里直接调用这个 `foo` 函数，就像调用 `fun1` 函数一样：

 复制代码

```
1 //调用一个外部函数foo
2 vector<Type *> argTypes(1, Type::getInt32Ty(TheContext));
3 FunctionType *fooType = FunctionType::get(Type::getVoidTy(TheContext), argTypes,
4
5 Function *foo = Function::Create(fooType, Function::ExternalLinkage, "foo", TheMo
6
7 std::vector<Value *> ArgsV2;
8 ArgsV2.push_back(rtn);
9 if (!ArgsV2.back())
10     return nullptr;
11
12 Builder.CreateCall(foo, ArgsV2, "calltmp2");
```

注意，我们在这里只对 `foo` 函数做了声明，并没有定义它的函数体，这时 LLVM 会在外部寻找 `foo` 的定义，它会找到用 C++ 编写的 `foo` 函数，然后调用并执行；如果 `foo` 函数在另一个目标文件中，它也可以找到。

刚才讲的是即时编译和运行，你也可以生成目标文件，然后再去链接和执行。生成目标文件的代码参见 [emitObject\(\)](#) 方法，基本上就是打开一个文件，然后写入生成的二进制目标代码。

针对目标机器生成目标代码的大量工作，就用这么简单的几行代码就实现了，是不是帮了你的大忙了？

## 课程小结

本节课，我们完成了从生成 IR 到编译执行的完整过程，同时，也初步熟悉了 LLVM 的接口。当然了，完全熟悉 LLVM 的接口还需要多做练习，掌握更多的细节。就本节课而言，我希望你掌握的重点如下：

LLVM 用一套对象模型在内存中表示 IR，包括模块、函数、基本块和指令，你可以通过 API 来生成这些对象。这些对象一旦生成，就可以编译和执行。

对于 if 语句和循环语句，需要生成多个基本块，并通过跳转指令形成正确的控制流图（CFG）。当存在多个前序节点可能改变某个变量的值的时候，使用 phi 指令来确定正确的值。

存储在内存中的本地变量，可以多次赋值。

LLVM 能够把外部函数和 IR 模型中的函数等价对待。

另外，为了降低学习难度，本节课，我没有做从 AST 翻译成 IR 的工作，而是针对一个目标功能（比如一个 C 语言的函数），硬编码调用 API 来生成 IR。你理解各种功能是如何生成 IR 以后，再从 AST 来翻译，就更加容易了。

## 一课一思

既然我带你演示了 if 语句如何生成 IR，那么你能思考一下，对于 for 循环和 while 循环语句，它对应的 CFG 应该是什么样的？应该如何生成 IR？欢迎你在留言区分享你的看法。

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朋友。

## 精选留言 (12)



**ZYS**

2020-01-28

宫老师考虑把课程整理成一本书出版那，很好的内容

作者回复: 嗯，已经在准备中~~~

我们的教程，限于篇幅，很多细节还是不会延伸太多。写成书的话，就可以讲得更充分。



👍 5



**渔子长**

2019-12-11

请教一个问题宫老师，LLVM的Module，设计是什么粒度的？比如一堆脚本可能拆分了几百个文件，把这些文件编译成IR，是每个文件对应一个Module？还是整体只构建一个Module？

作者回复: 如果对应于C或C++语言，一个Module就相当于一个目标文件，里面有一些函数式对外公开的，另一些是私有的，然后多个Module可以链接到一起。

所以，基本上是一个文件对应一个Module。

共 2 条评论 >

👍 4



**xiaobang**

2019-11-12

请教一下老师，为什么if语句要生成merge块，明明then和else里已经return了？难道llvm只允许一个函数一个return

作者回复: 不是的，可以有多个return。

不过，一般CFG会要求有两个特殊的块：入口块（Entry Block）和退出块（Exit Block）。这样的话，CFG就是一个有根的图（Rooted Directed Graph），便于执行某些分析和优化算法。如果要深入研究，可以去看看图的算法。

在LLVM中，有一个优化算法可以把多个return语句合并，这样就只有一个Exit Block了。



👍 4



**沉淀的梦想**

2019-10-23

老师用的什么版本的Illum, 我使用Illum 7.0编译老师lab-26的代码, 发现LegacyRTDyldObjectLinkingLayer和AcknowledgeORCv1Deprecation都已经不存在了, 但是网上搜了一下, 也没找到什么可以替代的东西

作者回复: 用Illum 9.0吧。

并且最好下载源代码, 自己编译。这样会按照debug模式编译, 用opt等工具的时候, 可以输出更多信息, 有利于你做实验。

共 2 条评论 >



3



**Geek\_6c84aa**

2021-01-16

使用默认选项编译安装Illum9时默认是没有开启rtti的。在编译本节代码时遇到undefined reference 'typeinfo'问题, 需要在cmakelist中添加add\_compile\_options(-fno-rtti) 或者重新编译Illum (-DLLVM-ENABLE-RTTI)。希望对大家有帮助。

作者回复: 感谢分享经验:-)



1



**coconut**

2021-01-04

Python版本基于Illumlite库的实现。

<https://github.com/leveryd/PlayWithCompiler/blob/master/Illum/1.%E5%88%9D%E8%AF%86Illum/test.py>

作者回复: Great!

python + Illum。

感谢你分享了更丰富使用编译技术的场景!



**lion\_fly**

2020-12-22

如果是在linux下使用LLVM的JIT，在编译的时候需要注意如下问题：

If you are compiling this on Linux, make sure to add the "-rdynamic" option as well. This ma

作者回复：感谢分享！



**lion\_fly**

2020-12-16

declare void @foo(i32)

Failure value returned from cantFail wrapped call

Symbols not found: [ foo ]

UNREACHABLE executed at /usr/lib/llvm-11/include/llvm/Support/Error.h:749!

作者回复：llvm工具要用熟练，会遇到很多小坑。不过代价是值得的。



**lion\_fly**

2020-12-14

在新版中llvm\_map\_components\_to\_libnames(llvm\_libs all)已经过时，新版需要写成：set(llvm\_libs LLVM-11)

作者回复：感谢分享！



**overmind**

2020-05-08

请教一个问题，如何调用一个类中的方法，并生产ir



**overmind**

2019-12-21

如何通过优化算法，把上述代码从使用内存的版本，优化成使用寄存器的版本。谢谢

作者回复: 第29讲有介绍分配寄存器的算法。尽量多使用寄存器, 实在寄存器放不下的, 才“溢出”到内存。



**夜行观星**

2019-11-17

老师, 我把生成二进制为文件链接成可执行文件出现了错误, 运行JIT可以正常执行。  
错误日志打印在下面, 老师有空看下

→ cmake-build-debug git:(master) X clang -v bin/output.o -o a.out

clang version 9.0.0 (tags/RELEASE\_900/final)

Target: x86\_64-apple-darwin18.6.0

Thread model: posix

InstalledDir: /usr/local/opt/llvm/bin

"/usr/bin/ld" -demangle -lto\_library /usr/local/Cellar/llvm/9.0.0/lib/libLTO.dylib -dynamic -arch x86\_64 -macosx\_version\_min 10.14.0 -o a.out bin/output.o -lSystem /usr/local/Cellar/llvm/9.0.0/lib/clang/9.0.0/lib/darwin/libclang\_rt.osx.a

Undefined symbols for architecture x86\_64:

"\_main", referenced from:

implicit entry/start for main executable

(maybe you meant: \_\_main)

ld: symbol(s) not found for architecture x86\_64

clang-9: error: linker command failed with exit code 1 (use -v to see invocation)

