LLVM架构简介

LLVM是什么

随着计算机技术的不断发展以及各种领域需求的增多，近几年来，许多编程语言如雨后春笋般出现，大多为了解决某一些特定领域的需求，比如说为JavaScript增加静态类型检查的TypeScript，为解决服务器端高并发的Golang，为解决内存安全和线程安全的Rust。随着编程语言的增多，编程语言的开发者往往都会遇到一些相似的问题：

- 怎样让我的编程语言能在尽可能多的平台上运行
- 怎样让我的编程语言充分利用各个平台自身的优势，做到最大程度的优化
- 怎样让我的编程语言在汇编层面实现「定制」，能够控制如符号表中的函数名、函数调用时参数的传递方法等汇编层面的概念

有的编程语言选择了使用C语言来解决这种问题，如早期的Haskell等。它们将使用自己语言的源代码编译成C代码，然后再在各个平台调用C编译器来生成可执行程序。为什么要选择C作为目标代码的语言呢？有几个原因：

第一，绝大部分的操作系统都是由C和汇编语言写成，因此平台大多会提供一个C编译器可以使用，这样就解决了第一个问题。

第二，绝大部分的操作系统都会提供C语言的接口，以及C库。我们的编程语言因此可以很方便地调用相应的接口来实现更广泛的功能。

第三，C语言本身并没有笨重的运行时，代码很贴近底层，可以使用一定程度的定制。

以上三个理由让许多的编程语言开发者选择将自己的语言编译成C代码。

然而，我们知道，一个平台最终运行的二进制可执行文件，实际上就是在运行与之等价的汇编代码。与汇编代码比起来，C语言还是太抽象了，我们希望能更灵活地操作一些更底层的部分。同时，我们也希望相应代码在各个平台能有和C语言一致，甚至比其更好的优化程度。

因此，LLVM出现后，成了一个更好的选择。我们可以从LLVM官网中看到：

> The LLVM Core libraries provide a modern source- and target-independent optimizer, along with code generation support for many popular CPUs (as well as some less common ones!) These libraries are built around a well specified code representation known as the LLVM intermediate representation ("LLVM IR"). The LLVM Core libraries are well documented, and it is particularly easy to invent your own language (or port an existing compiler) to use LLVM as an optimizer and code generator.

简单地说，LLVM代替了C语言在现代语言编译器实现中的地位。我们可以将自己语言的源代码编译成LLVM中间代码（LLVM IR），然后由LLVM自己的后端对这个中间代码进行优化，并且编译到相应的平台的二进制程序。

LLVM的优点正好对应我们之前讲的三个问题：

- LLVM后端支持的平台很多，我们不需要担心CPU、操作系统的问题（运行库除外）
- LLVM后端的优化水平较高，我们只需要将代码编译成LLVM IR，就可以由LLVM后端作相应的优化

- LLVM IR本身比较贴近汇编语言，同时也提供了许多ABI层面的定制化功能

因为LLVM的优越性，除了LLVM自己研发的C编译器Clang，许多新的工程都选择了使用LLVM，我们可以在其官网看到使用LLVM的项目的列表，其中，最著名的就是Rust、Swift等语言了。

LLVM架构

要解释使用LLVM后端的编译器整体架构，我们就拿最著名的C语言编译器Clang为例。

在一台x86_64指令集的macOS系统上，我有一个最简单的C程序`test.c`：

```
int main() {
    return 0;
}
```

我们使用

```
clang test.c -o test
```

究竟经历了哪几个步骤呢?

前端的语法分析

首先，Clang的前端编译器会将这个C语言的代码进行预处理、语法分析、语义分析，也就是我们常说的parse the source code。这里不同语言会有不同的做法。总之，我们是将「源代码」这一字符串转化为内存中有意义的数据，表示我们这个代码究竟想表达什么。

我们可以使用

```
clang -Xclang -ast-dump -fsyntax-only test.c
```

输出我们`test.c`经过编译器前端的预处理、语法分析、语义分析之后，生成的抽象语法树（AST）：

```
TranslationUnitDecl 0x7fc02681ea08 <<invalid sloc>> <invalid sloc>
|-TypedefDecl 0x7fc02681f2a0 <<invalid sloc>> <invalid sloc> implicit __int128_t '__int128'
| `-BuiltinType 0x7fc02681efa0 '__int128'
|-TypedefDecl 0x7fc02681f310 <<invalid sloc>> <invalid sloc> implicit __uint128_t 'unsigned __int128'
| `-BuiltinType 0x7fc02681efc0 'unsigned __int128'
|-TypedefDecl 0x7fc02681f5f8 <<invalid sloc>> <invalid sloc> implicit __NSConstantString 'struct __NSConstantString_tag'
| `-RecordType 0x7fc02681f3f0 'struct __NSConstantString_tag'
|   `-Record 0x7fc02681f368 '__NSConstantString_tag'
|-TypedefDecl 0x7fc02681f690 <<invalid sloc>> <invalid sloc> implicit __builtin_ms_va_list 'char *'
| `-PointerType 0x7fc02681f650 'char *'
|   `-BuiltinType 0x7fc02681eaa0 'char'
```

```
|-TypedefDecl 0x7fc02681f968 <<invalid sloc>> <invalid sloc> implicit __builtin_va_list 'struct __va_list_tag [1]'
| `-ConstantArrayType 0x7fc02681f910 'struct __va_list_tag [1]' 1
|   `-RecordType 0x7fc02681f770 'struct __va_list_tag'
|     `-Record 0x7fc02681f6e8 '__va_list_tag'
`-FunctionDecl 0x7fc02585a228 <test.c:1:1, line:3:1> line:1:5 main 'int ()'
  `-CompoundStmt 0x7fc02585a340 <col:12, line:3:1>
    `-ReturnStmt 0x7fc02585a330 <line:2:5, col:12>
      `-IntegerLiteral 0x7fc02585a310 <col:12> 'int' 0
```

这一长串输出看上去就让人眼花缭乱，然而，我们只需要关注最后四行：

```
`-FunctionDecl 0x7fc02585a228 <test.c:1:1, line:3:1> line:1:5 main 'int ()'
  `-CompoundStmt 0x7fc02585a340 <col:12, line:3:1>
    `-ReturnStmt 0x7fc02585a330 <line:2:5, col:12>
      `-IntegerLiteral 0x7fc02585a310 <col:12> 'int' 0
```

这才是我们源代码的AST。可以很方便地看出，经过Clang前端的预处理、语法分析、语义分析，我们的代码被分析成一个函数，其函数体是一个复合语句，这个复合语句包含一个返回语句，返回语句中使用了一个整型字面量`0`。

因此，总结而言，我们基于LLVM的编译器的第一步，就是将源代码转化为内存中的抽象语法树AST。

前端生成中间代码

第二个步骤，就是根据内存中的抽象语法树AST生成LLVM IR中间代码（有的比较新的编译器还会先将AST转化为MLIR再转化为IR）。

我们知道，我们写编译器的最终目的，是将源代码交给LLVM后端处理，让LLVM后端帮我们优化，并编译到相应的平台。而LLVM后端为我们提供的中介，就是LLVM IR。我们只需要将内存中的AST转化为LLVM IR就可以放手不管了，接下来的所有事都是LLVM后端帮我们实现。

关于LLVM IR，我在下面会详细解释。我们现在先看看将AST转化之后，会产生什么样的LLVM IR。我们使用

```
clang -S -emit-llvm test.c
```

这时，会生成一个`test.ll`文件：

```
; ModuleID = 'test.c'
source_filename = "test.c"
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.15.0"

; Function Attrs: noinline nounwind optnone ssp uwtable
define i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
```

```
  ret i32 0
}

attributes #0 = { noinline nounwind optnone ssp uwtable "correctly-rounded-divide-sqrt-fp-math"="false" "darwin-stkchk-strong-link" "disable-tail-calls"="false" "frame-pointer"="all" "less-precise-fpmad"="false" "min-legal-vector-width"="0" "no-infs-fp-math"="false" "no-jump-tables"="false" "no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false" "no-trapping-math"="false" "probe-stack"="___chkstk_darwin" "stack-protector-buffer-size"="8" "target-cpu"="penryn" "target-features"="+cx16,+cx8,+fxsr,+mmx,+sahf,+sse,+sse2,+sse3,+sse4.1,+ssse3,+x87" "unsafe-fp-math"="false" "use-soft-float"="false" }

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

!0 = !{i32 2, !"SDK Version", [3 x i32] [i32 10, i32 15, i32 4]}
!1 = !{i32 1, !"wchar_size", i32 4}
!2 = !{i32 7, !"PIC Level", i32 2}
!3 = !{!"Apple clang version 11.0.3 (clang-1103.0.32.62)"}
```

这看上去更加让人迷惑。然而，我们同样地只需要关注五行内容：

```
define i32 @main() #0 {
  %1 = alloca i32, align 4
  store i32 0, i32* %1, align 4
  ret i32 0
}
```

这是我们AST转化为LLVM IR中最核心的部分，可以隐约感受到这个代码所表达的意思。

LLVM后端优化IR

LLVM后端在读取了IR之后，就会对这个IR进行优化。这在LLVM后端中是由`opt`这个组件完成的，它会根据我们输入的LLVM IR和相应的优化等级，进行相应的优化，并输出对应的LLVM IR。

我们可以用

```
opt test.ll -S -O3
```

对相应的代码进行优化，也可以直接用

```
clang -S -emit-llvm -O3 test.c
```

优化，并输出相应的优化结果：

```
; ModuleID = 'test.c'
source_filename = "test.c"
```

```
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.15.0"

; Function Attrs: norecurse nounwind readnone ssp uwtable
define i32 @main() local_unnamed_addr #0 {
  ret i32 0
}

attributes #0 = { norecurse nounwind readnone ssp uwtable "correctly-rounded-divide-sqrt-fp-math"="false" "darwin-stkchk-strong-link" "disable-tail-calls"="false" "frame-pointer"="all" "less-precise-fpmad"="false" "min-legal-vector-width"="0" "no-infs-fp-math"="false" "no-jump-tables"="false" "no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false" "no-trapping-math"="false" "probe-stack"="___chkstk_darwin" "stack-protector-buffer-size"="8" "target-cpu"="penryn" "target-features"="+cx16,+cx8,+fxsr,+mmx,+sahf,+sse,+sse2,+sse3,+sse4.1,+ssse3,+x87" "unsafe-fp-math"="false" "use-soft-float"="false" }

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

!0 = !{i32 2, !"SDK Version", [3 x i32] [i32 10, i32 15, i32 4]}
!1 = !{i32 1, !"wchar_size", i32 4}
!2 = !{i32 7, !"PIC Level", i32 2}
!3 = !{!"Apple clang version 11.0.3 (clang-1103.0.32.62)"}
```

我们观察`@main`函数，可以发现其函数体确实减少了不少。

LLVM后端生成汇编代码

LLVM后端帮我们做的最后一步，就是由LLVM IR生成汇编代码，这是由`llc`这个组件完成的。

我们可以用

```
llc test.ll
```

生成`test.s`：

```
    .section    __TEXT,__text,regular,pure_instructions
    .build_version macos, 10, 15    sdk_version 10, 15, 4
    .globl  _main                   ## -- Begin function main
    .p2align    4, 0x90
_main:                                  ## @main
    .cfi_startproc
### %bb.0:
    pushq   %rbp
    .cfi_def_cfa_offset 16
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
```

```
    .cfi_def_cfa_register %rbp
    xorl    %eax, %eax
    movl    $0, -4(%rbp)
    popq    %rbp
    retq
    .cfi_endproc
                                        ## -- End function

.subsections_via_symbols
```

这就回到了我们熟悉的汇编代码中。

有了汇编代码之后，我们就需要调用操作系统自带的汇编器、链接器，最终生成可执行程序。

LLVM IR

根据我们上面讲的，一个基于LLVM后端的编译器的整体过程是

```
.c --frontend--> AST --frontend--> LLVM IR --LLVM opt--> LLVM IR --LLVM llc--> .s Assembly --OS Assembler--> .o --OS Linker--> executable
```

这样一个过程。由此我们可以见，LLVM IR是连接编译器前端与LLVM后端的一个桥梁。同时，整个LLVM后端也是围绕着LLVM IR来进行的。所以，我的这个系列就打算介绍的是LLVM IR的入门级教程。

那么，LLVM IR究竟是什么呢？它的全称是LLVM Intermediate Representation，也就是LLVM的中间表示，我们可以在这篇LangRef中查看其所有信息。这看起来模糊不清的名字，也容易让人产生疑问。事实上，在我理解中，LLVM IR同时表示了三种东西：

- 内存中的LLVM IR
- 比特码形式的LLVM IR
- 可读形式的LLVM IR

内存中的LLVM IR是编译器作者最常接触的一个形式，也是其最本质的形式。当我们在内存中处理抽象语法树AST时，需要根据当前的项，生成对应的LLVM IR，这也就是编译器前端所做的事。我们的编译器前端可以用许多语言写，LLVM也为许多语言提供了Binding，但其本身还是用C++写的，所以这里就拿C++为例。

LLVM的C++接口在`llvm/IR`目录下提供了许多的头文件，如`llvm/IR/Instructions.h`等，我们可以使用其中的`Value`，`Function`，`ReturnInst`等等成千上万的类来完成我们的工作。也就是说，我们并不需要把AST变成一个个字符串，如`ret i32 0`等，而是需要将AST变成LLVM提供的IR类的实例，然后在内存中交给LLVM后端处理。

而比特码形式和可读形式则是将内存中的LLVM IR持久化的方法。比特码是采用特定格式的二进制序列，而可读形式的LLVM IR则是采用特定格式的human readable的代码。我们可以用

```
clang -S -emit-llvm test.c
```

生成可读形式的LLVM IR文件`test.ll`，采用

```
clang -c -emit-llvm test.c
```

生成比特码形式的LLVM IR文件`test.bc`，采用

```
llvm-as test.ll
```

将可读形式的`test.ll`转化为比特码`test.bc`，采用

```
llvm-dis test.bc
```

将比特码`test.bc`转化为可读形式的`test.ll`。

我这个系列，将主要介绍的是可读形式的LLVM IR的语法。

LLVM的下载与安装

macOS的Xcode会自带`clang`、`clang++`、`swiftc`等基于LLVM的编译器，但并不会带全部的LLVM的套件，如`llc`，`opt`等。类似的，Windows的Visual Studio同样也可以下载Clang编译器，但依然没有带全部的套件。而Linux下则并没有自带编译器或套件。

我们可以直接在LLVM的官网上下载LLVM全部套件，但也可以去相应的系统包管理器中下载：

在macOS中，可以直接使用

```
brew install llvm
```

下载，而在Ubuntu下，也可以使用

```
apt-get install llvm
```

进行下载。使用系统包管理器下载的好处在于，我们可以使用国内的镜像，能够更快地实现下载。

在哪可以看到我的文章

我的LLVM IR入门指南系列可以在我的个人博客、GitHub：Evian-Zhang/llvm-ir-tutorial、知乎、CSDN中查看，本教程中涉及的大部分代码也都在同一GitHub仓库中。

本人水平有限，写此文章仅希望与大家分享学习经验，文章中必有缺漏、错误之处，望方家不吝斧正，与大家共同学习，共同进步，谢谢大家！

Hello world

在系统学习LLVM IR语法之前，我们应当首先掌握的是使用LLVM IR写的最简单的程序，也就是大家常说的Hello world版程序。这是因为，编程语言的学习，往往需要伴随着练习。但是一个独立的程序需要许多的前置语法基础，那么我们不可能在了解了所有前置语法基础之后才完成第一个独立程序，否则在学习前置语法基础的时候，就没有办法在实际的程序中练习了。因此，正确的学习方式应该是，首先掌握这门语言独立程序的基础框架，然后每学习一个新的语法知识，就在框架中练习，并编译看结果是否是自己期望的结果。

综上所述，学习一门语言的第一步，就是掌握其最简单的程序的基本框架是如何写的。

最基本的程序

以macOS 10.15为例，我们最基本的程序为：

```
; main.ll
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.15.0"

define i32 @main() {
    ret i32 0
}
```

这个程序可以看作最简单的C语言代码：

```
int main() {
    return 0;
}
```

在macOS 10.15上编译而成的结果。

我们可以直接测试这个代码的正确性：

```
clang main.ll -o main
./main
```

使用`clang`可以直接将`main.ll`编译成可执行文件`main`。运行这个程序后，程序自动退出，并返回`0`。这正符合我们的预期。

基本概念

下面，我们对`main.ll`逐行解释一些比较基本的概念。

注释

首先，第一行`; main.ll`。这是一个注释。在LLVM IR中，注释以`;`开头，并一直延伸到行尾。所以在LLVM IR中，并没有像C语言中的`/* comment block */`这样的注释块，而全都类似于`// comment line`这样的注释行。

目标数据分布和平台

第二行和第三行的`target datalayout`和`target triple`，则是注明了目标汇编代码的数据分布和平台。我们之前提到过，LLVM是一个面向多平台的深度定制化编译器后端，而我们LLVM IR的目的，则是让LLVM后端根据IR代码生成相应平台的汇编代码。所以，我们需要在IR代码中指明我们需要生成哪一个平台的代码，也就是`target triple`字段。类似地，我们还需要定制数据的大小端序、对齐形式等需求，所以我们也需要指明`target datalayout`字段。关于这两个字段的值的详细情况，我们可以参考Data Layout和Target Triple这两个官方文档。我们可以对照官方文档，解释我们在macOS上得到的结果：

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

表示：

- `e`：小端序
- `m:o`：符号表中使用Mach-O格式的name mangling（这玩意儿我一直不知道中文是啥，就是把程序中的标识符经过处理得到可执行文件中的符号表中的符号）
- `i64:64`：将`i64`类型的变量采用64比特的ABI对齐
- `f80:128`：将`long double`类型的变量采用128比特的ABI对齐
- `n8:16:32:64`：目标CPU的原生整型包含8比特、16比特、32比特和64比特
- `S128`：栈以128比特自然对齐

```
target triple = "x86_64-apple-macosx10.15.0"
```

表示：

- `x86_64`：目标架构为x86_64架构
- `apple`：供应商为Apple
- `macosx10.15.0`：目标操作系统为macOS 10.15

在一般情况下，我们都是想生成当前平台的代码，也就是说不太会改动这两个值。因此，我们可以直接写一个简单的`test.c`程序，然后使用

```
clang -S -emit-llvm test.c
```

生成LLVM IR代码`test.ll`，在`test.ll`中找到`target datalayout`和`target triple`这两个字段，然后拷贝到我们的代码中即可。

比方说，我在x86_64指令集的Ubuntu 20.04的机器上得到的就是：

```
target datalayout = "e-m:e-p270:32:32-p271:32:32-p272:64:64-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"
```

和我们在macOS上生成的代码就不太一样。

主程序

我们知道，主程序是可执行程序的入口点，所以任何可执行程序都需要`main`函数才能运行。所以，

```
define i32 @main() {
    ret i32 0
}
```

就是这段代码的主程序。关于正式的函数、指令的定义，我会在之后的文章中提及。这里我们只需要知道，在`@main()`之后的，就是这个函数的函数体，`ret i32 0`就代表C语言中的`return 0;`。因此，如果我们要增加代码，就只需要在大括号内，`ret i32 0`前增加代码即可。

在哪可以看到我的文章

我的LLVM IR入门指南系列可以在我的个人博客、GitHub：Evian-Zhang/llvm-ir-tutorial、知乎、CSDN中查看，本教程中涉及的大部分代码也都在同一GitHub仓库中。

本人水平有限，写此文章仅希望与大家分享学习经验，文章中必有缺漏、错误之处，望方家不吝斧正，与大家共同学习，共同进步，谢谢大家！

数据表示

LLVM IR和其它的汇编语言类似，其核心就是对数据的操作。这涉及到了两个问题：什么数据和怎么操作。具体到这篇文章中，我就将介绍的是，在LLVM IR中，是如何表示一个数据的。

汇编层次的数据表示

LLVM IR是最接近汇编语言的一层抽象，所以我们首先需要了解在计算机底层，汇编语言的层次中，数据是怎样表示的。

谈到汇编层次的数据表示，一个老生常谈的程序就是

```c
##include <stdlib.h>

int global_data = 0;

int main() {
    int stack_data = 0;
    int *heap_pointer = (int *)malloc(16 * sizeof(int));
    return 0;
}
```

我们知道，一个C语言从代码到执行的过程是代码-->硬盘上的二进制程序-->内存中的进程。在代码被编译到二进制程序的时候，`global_data`本身就写在了二进制程序中。在操作系统将二进制程序载入内存时，就会在特定的区域（数据区）初始化这些值。而`stack_data`代表的局部变量，则是在程序执行其所在的函数时，在栈上初始化，类似地，`heap_pointer`这个指针也是在栈上，而其指向的内容，则是操作系统分配在堆上的。

用一个图可以简单地表示：

```
+-------------------------------+
|           stack_data          |
|         heap_pointer          |  <-------------- stack
+-------------------------------+
|                               |
|                               |  <-------------- available memory space
|                               |
+-------------------------------+
| data pointed by heap_pointer  |  <-------------- heap
+-------------------------------|
|           global_data         |  <-------------- .DATA section
+-------------------------------+
```

这就是一个简化后的进程的内存模型。也就是说，一共有三种数据：

- 栈上的数据
- 堆中的数据

- 数据区里的数据

但是，我们仔细考虑一下，在堆中的数据，能否独立存在。操作系统提供的在堆上创建数据的接口如`malloc`等，都是返回一个指针，那么这个指针会存在哪里呢？寄存器里，栈上，数据区里，或者是另一个被分配在堆上的指针。也就是说，可能会是：

```
##include <stdlib.h>

int *global_pointer = (int *)malloc(16 * sizeof(int));

int main() {
    int *stack_pointer = (int *)malloc(16 * sizeof(int));
    int **heap_pointer = (int **)malloc(sizeof(int *));
    *heap_pointer = (int *)malloc(16 * sizeof(int));
    return 0;
}
```

但不管怎样，堆中的数据都不可能独立存在，一定会有一个位于其他位置的引用。所以，在内存中的数据按其表示来说，一共分为两类：

- 栈上的数据
- 数据区里的数据

除了内存之外，还有一个存储数据的地方，那就是寄存器。因此，我们在程序中可以用来表示的数据，一共分为三类：

- 寄存器中的数据
- 栈上的数据
- 数据区里的数据

LLVM IR中的数据表示

LLVM IR中，我们需要表示的数据也是以上三种。那么，这三种数据各有什么特点，又需要根据LLVM的特性做出什么样的调整呢？

数据区里的数据

我们知道，数据区里的数据，其最大的特点就是，能够给整个程序的任何一个地方使用。同时，数据区里的数据也是占静态的二进制可执行程序的体积的。所以，我们应该只将需要全程序使用的变量放在数据区中。而现代编程语言的经验告诉我们，这类全局静态变量应该越少越好。

同时，由于LLVM是面向多平台的，所以我们还需要考虑的是该怎么处理这些数据。一般来说，大多数平台的可执行程序格式中都会包含`.DATA`分区，用来存储这类的数据。但除此之外，每个平台还有专门的更加细致的分区，比如说，Linux的ELF格式中就有`.rodata`来存储只读的数据。因此，LLVM的策略是，让我们尽可能细致地定义一个全局变量，比如说注明其是否只读等，然后依据各个平台，如果平台的可执行程序格式支持相应的特性，就可以进行优化。

一般来说，在LLVM IR中定义一个存储在数据区中的全局变量，其格式为：

```
@global_variable = global i32 0
```

这个语句定义了一个`i32`类型的全局变量`@global_variable`，并且将其初始化为`0`。

如果是只读的全局变量，也就是常量，我们可以用`constant`来代替`global`：

```
@global_constant = constant i32 0
```

这个语句定义了一个`i32`类型的全局常量`@global_constant`，并将其初始化为`0`。

符号表

关于在数据区的数据，有一个特别需要注意的，就是数据的名称与二进制文件中的符号表。在LLVM IR中，所有的全局变量的名称都需要用`@`开头。我们有一个这样的LLVM IR：

```
; global_variable_test.ll
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.15.0"

@global_variable = global i32 0

define i32 @main() {
    ret i32 0
}
```

也就是说，在之前最基本的程序的基础上，新增了一个全局变量`@global_variable`。我们将其直接编译成可执行文件：

```
clang global_variable_test.ll -o global_variable_test
```

然后，我们使用`nm`命令查看其符号表：

```
nm global_variable_test
```

结果为：

```
0000000100000000 T __mh_execute_header
0000000100001000 S _global_variable
0000000100000f70 T _main
                 U dyld_stub_binder
```

我们注意到，出现了`_global_variable`这个字段。这里开头的`_`可以不用关注，这是Mach-O的name mangling策略导致的，我们在Ubuntu下可以用同样的步骤，查看到出现的是`global_variable`字段。

这表明，直接定义的全局变量，其名称会出现在符号表之中。那么，怎么控制这个行为呢？首先，我们需要简单地了解一下符号表。

简单来说，ELF文件中的符号表会有两个区域：`.symtab`和`.dynsym`。在最初只有静态链接的时期，符号表的作用主要有两个：Debug和静态链接。我们在Debug的时候，往往会需要某些数据的符号，而这就是放在`.symtab`里的；同样地，当我们用链接器将两个目标文件链接的时候，也需要解决其中的符号交叉引用，这时的信息也是放在`.symtab`里。然而，这些信息有一个特点：不需要在运行时载入内存。我们在运行时根本不关心某些数据的符号，也不需要链接，所以`.symtab`在运行时不会载入内存。然而，在出现了动态链接之后，就产生了变化。动态链接允许可执行文件在载入内存、运行这两个阶段再链接动态链接库，那么这时就需要解决符号的交叉引用。因此，有些符号就需要在运行时载入内存。将整个`.symtab`全部载入内存是不现实的，所以大家就把一部分需要载入内存的符号拷贝到`.dynsym`这个分区，也就是动态符号表中。

在LLVM IR中，控制符号表与两个概念密切相关：链接与可见性，LLVM IR也提供了Linkage Type和Visibility Styles这两个修饰符来控制相应的行为。

链接类型

对于链接类型，我们常用的主要有什么都不加（默认为`external`）、`private`和`internal`。

什么都不加的话，就像我们刚刚那样，直接把全局变量的名字放在了符号表中，用`nm`查看出来，在`_global_variable`之前是`S`，表示除了几个主流分区之外的其它分区，如果我们用`llc`将代码输出成汇编的话，可以看到`global_varaible`在macOS下是在`__DATA`段的`__common`节。

用`private`，则代表这个变量的名字不会出现在符号表中。我们将原来的代码改写成

```
@global_variable = private global i32 0
```

那么，用`nm`查看其编译出的可执行文件：

```
0000000100000000 T __mh_execute_header
0000000100000f70 T _main
                 U dyld_stub_binder
```

这个变量的名字就消失了。

用`internal`则表示这个变量是以局部符号的身份出现（全局变量的局部符号，可以理解成C中的`static`关键词）。我们将原来的代码改写成

```
@global_variable = internal global i32 0
```

那么，再次将其编译成可执行程序，并用`nm`查看：

```
0000000100000000 T __mh_execute_header
0000000100001000 b _global_variable
0000000100000f70 T _main
                 U dyld_stub_binder
```

`_global_variable`前面的符号变成了小写的`b`，这代表这个变量是位于`__bss`节的局部符号。

LLVM IR层次的链接类型也就控制了实际目标文件的链接策略，什么符号是导出的，什么符号是本地的，什么符号是消失的。但是，这个变量放在可执行程序中的哪个区、哪个节并不是统一的，是与平台相关的，如在macOS上什么都不加的`global_variable`是放在`__DATA`段的`__common`节，而`internal`的`global_variable`则是处于`__DATA`段的`__bss`节。而在Ubuntu上，什么都不加的`global_variable`则是位于`.bss`节，`internal`的`global_variable`也是处于`.bss`的局部符号。

可见性

可见性在实际使用中则比较少，主要分为三种`default`，`hidden`和`protected`，这里主要的区别在于符号能否被重载。`default`的符合可以被重载，而`protected`的符号则不可以；此外，`hidden`则不将变量放在动态符号表中，因此其它的模块不可以直接引用这个符号。

寄存器内的数据和栈上的数据

这两种数据我选择放在一起讲。我们知道，除了DMA等奇技淫巧之外，大多数对数据的操作，如加减乘除、比大小等，都需要操作的是寄存器内的数据。那么，我们为什么需要把数据放在栈上呢？主要有两个原因：

- 寄存器数量不够
- 需要操作内存地址

如果我们一个函数内有三四十个局部变量，但是家用型CPU最多也就十几个通用寄存器，所以我们不可能把所有变量都放在寄存器中，因此我们需要把一部分数据放在内存中，栈就是一个很好的存储数据的地方；此外，有时候我们需要直接操作内存地址，但是寄存器并没有通用的地址表示，所以只能把数据放在栈上来完成对地址的操作。

因此，在不操作内存地址的前提下，栈只是寄存器的一个替代品。有一个很简单的例子可以解释这个概念。我们有一个很简单的C程序：

```c
// max.c
int max(int a, int b) {
    if (a > b) {
        return a;
    } else {
        return b;
    }
}

int main() {
    int a = max(1, 2);
    return 0;
}
```

在x86_64架构macOS上编译的话，我们首先来看`max(1, 2)`是如何调用的：

```
movl    $1, %edi
movl    $2, %esi
callq   _max
```

将参数`1`和`2`分别放到了寄存器`edi`和`esi`里。那么，`max`函数又是如何操作的呢?

```
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -8(%rbp)      # move data stored in %edi to stack at
-8(%rbp)
    movl    %esi, -12(%rbp)     # move data stored in %esi to stack at
-12(%rbp)
    movl    -8(%rbp), %eax      # move data stored in stack at -8(%rbp) to
register %eax
    cmpl    -12(%rbp), %eax     # compare data stored in stack at
-12(%rbp) with data stored in %eax
    jle LBB0_2                  # if compare result is less than or equal
to, then go to label LBB0_2
### %bb.1:
    movl    -8(%rbp), %eax      # move data stored in stack at -8(%rbp) to
register %eax
    movl    %eax, -4(%rbp)      # move data stored in %eax to stack at
-4(%rbp)
    jmp LBB0_3                  # go to label LBB0_3
LBB0_2:
    movl    -12(%rbp), %eax     # move data stored in stack at -12(%rbp)
to register %eax
    movl    %eax, -4(%rbp)      # move data stored in %eax to stack at
-4(%rbp)
LBB0_3:
    movl    -4(%rbp), %eax      # move data stored in stack at -4(%rbp) to
register %eax
    popq    %rbp
    retq
```

考虑到篇幅，我将这个汇编每一个重要步骤所做的事都以注释形式写在了代码里面。这个看上去很复杂，但实际上做的是这样的事：

1. 把`int a`和`int b`看作局部变量，分别存储在栈上的`-8(%rbp)`和`-12(%rbp)`上
2. 为了比较这两个局部变量，将一个由栈上导入寄存器`eax`中
3. 比较`eax`寄存器中的值和另一个局部变量
4. 将两者中比较大的那个局部变量存储在栈上的`-4(%rbp)`上（由于x86_64架构不允许直接将内存中的一个值拷贝到另一个内存区域中，所以得先把内存区域中的值拷贝到`eax`寄存器里，再从`eax`寄存器里拷贝到目标内存中）
5. 将栈上`-4(%rbp)`这个用来存储返回值的区域的值拷贝到`eax`中，并返回

这看上去真是太费事了。但是，这也是无可奈何之举。这是因为，在不开优化的情况下，一个C的函数中的局部变量（包括传入参数）和返回值都应该存储在函数本身的栈帧中，所以，我们得把这简单的两个值在不同的内存区域和寄存器里来回拷贝。

那么，如果我们优化一下会怎样呢？我们使用

```
clang -O1 -S max.c
```

之后，我们的`_max`函数的汇编代码是：

```
pushq   %rbp
movq    %rsp, %rbp
movl    %esi, %eax
cmpl    %esi, %edi
cmovgel %edi, %eax
popq    %rbp
retq
```

那么长的一串代码竟然变的如此简洁了。这个代码翻译成伪代码就是

```
function max(register a, register b) {
    register c = register b
    if (register a >= register c) {
        register c = register a
    }
    return register c
}
```

很简单的事，并且把所有的操作都从对内存的操作变成了对寄存器的操作。

因此，由这个简单的例子我们可以看出来，如果寄存器的数量足够，并且代码中没有需要操作内存地址的时候，寄存器是足够胜任的，并且更加高效的。

寄存器

正因为如此，LLVM IR引入了虚拟寄存器的概念。在LLVM IR中，一个函数的局部变量可以是寄存器或者栈上的变量。对于寄存器而言，我们只需要像普通的赋值语句一样操作，但需要注意名字必须以`%`开头：

```
%local_variable = add i32 1, 2
```

此时，`%local_variable`这个变量就代表一个寄存器，它此时的值就是`1`和`2`相加的结果。我们可以写一个简单的程序验证这一点：

```
; register_test.ll
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.15.0"

define i32 @main() {
    %local_variable = add i32 1, 2
    ret i32 %local_variable
}
```

我们在x86_64的macOS系统上查看其编译出的汇编代码，其主函数为：

```
_main:
    movl    $2, %eax
    addl    $1, %eax
    retq
```

确实这个局部变量`%local_variable`变成了寄存器`eax`。

关于寄存器，我们还需了解一点。在不同的ABI下，会有一些called-saved register和calling-saved register。简单来说，就是在函数内部，某些寄存器的值不能改变。或者说，在函数返回时，某些寄存器的值要和进入函数前相同。比如，在System V的ABI下，`rbp`, `rbx`, `r12`, `r13`, `r14`, `r15`都需要满足这一条件。由于LLVM IR是面向多平台的，所以我们需要一份代码适用于多种ABI。因此，LLVM IR内部自动帮我们做了这些事。如果我们把所有没有被保留的寄存器都用光了，那么LLVM IR会帮我们把这些被保留的寄存器放在栈上，然后继续使用这些被保留寄存器。当函数退出时，会帮我们自动从栈上获取到相应的值放回寄存器内。

那么，如果所有通用寄存器都用光了，该怎么办？LLVM IR会帮我们把剩余的值放在栈上，但是对我们用户而言，实际上都是虚拟寄存器，用户是感觉不到差别的。

因此，我们可以粗略地理解LLVM IR对寄存器的使用：

- 当所需寄存器数量较少时，直接使用called-saved register，即不需要保留的寄存器
- 当called-saved register不够时，将calling-saved register原本的值压栈，然后使用calling-saved register
- 当寄存器用光以后，就把多的虚拟寄存器的值压栈

我们可以写一个简单的程序验证。对于x86_64架构下，我们只需要使用15个虚拟寄存器就可以验证这件事。鉴于篇幅，我就不把代码放在文章中了，如果想看详细代码可以去我的GitHub仓库中查看`many_registers_test.ll`。我们将其编译成汇编语言之后，可以看到在函数开头就有

```
pushq   %r15
pushq   %r14
pushq   %r13
pushq   %r12
pushq   %rbx
```

也就是把那些需要保留的寄存器压栈。然后随着寄存器用光，第15个虚拟寄存器就会使用栈：

```
movl    %ecx, -4(%rsp)
addl    $1, %ecx
```

栈

我们之前说过，当不需要操作地址并且寄存器数量足够时，我们可以直接使用寄存器。而LLVM IR的策略保证了我们可以使用无数的虚拟寄存器。那么，在需要操作地址以及需要可变变量（之后会提到为什么）时，我们就需要使用栈。

LLVM IR对栈的使用十分简单，直接使用`alloca`指令即可。如：

```
%local_variable = alloca i32
```

就可以声明一个在栈上的变量了。关于栈上变量的操作，我会在下面提到，目前我们对栈上变量的了解只需这么多。

全局变量和栈上变量皆指针

下面，我们就需要讲怎样操作全局变量和栈上的变量。这两种变量实际上是类似的，LLVM IR把它们都看作指针。也就是说，对于全局变量：

```
@global_variable = global i32 0
```

和栈上变量

```
%local_variable = alloca i32
```

这两个变量实际上都是`i32*`类型的指针，指向它们所处的内存区域。所以，我们不能这样：

```
%1 = add i32 1, @global_variable ; wrong!
```

因为`@global_variable`只是一个指针。

如果要操作这些值，必须使用`load`和`store`这两个命令。如果我们要获取`@global_variable`的值，就需要

```
%1 = load i32, i32* @global_variable
```

这个指令的意思是，把一个`i32*`类型的指针`@global_variable`的`i32`类型的值赋给虚拟寄存器`%1`，然后我们就能愉快地

```
%2 = add i32 1, %1
```

这样了。

类似地，如果我们要将值存储到全局变量或栈上变量里，会需要`store`命令：

```
store i32 1, i32* @global_variable
```

这个代表将`i32`类型的值`1`赋给`i32*`类型的全局变量`@global_variable`所指的内存区域中。

SSA

LLVM IR是一个严格遵守SSA(Static Single Assignment)策略的语言。SSA的要求很简单：每个变量只被赋值一次。也就是说，你不能

```
%1 = add i32 1, 2
%1 = add i32 3, 4
```

对`%1`同时赋值两次是不被允许的。

那么，我们应该怎样实现可变变量呢？很简单，把可变变量放到全局变量或者栈内变量里，虚拟寄存器只存储不可变的变量。比如说，我想实现上面的功能，把两次运算结果储存到同一个变量内：

```
%stack_variable = alloca i32
%1 = add i32 1, 2
store i32 %1, i32* %stack_variable
%2 = add i32 3, 4
store i32 %2, i32* %stack_variable
```

我们同样遵守了SSA，而且也满足了可变变量的需求。此外，虽然LLVM IR上看上去很复杂，LLVM后端也会帮我们优化到比较简单的形式，不会因为SSA而降低性能。

在哪可以看到我的文章

我的LLVM IR入门指南系列可以在我的个人博客、GitHub：Evian-Zhang/llvm-ir-tutorial、知乎、CSDN中查看，本教程中涉及的大部分代码也都在同一GitHub仓库中。

本人水平有限，写此文章仅希望与大家分享学习经验，文章中必有缺漏、错误之处，望方家不吝斧正，与大家共同学习，共同进步，谢谢大家！

类型系统

我们知道，汇编语言是弱类型的，我们操作汇编语言的时候，实际上考虑的是一些二进制串。但是，LLVM IR却是强类型的，在LLVM IR中所有变量都必须有类型。这是因为，我们在使用高级语言编程的时候，往往都会使用强类型的语言，弱类型的语言无必要性，也不利于维护。因此，使用强类型语言，LLVM IR可以更好地进行优化。

基本的数据类型

LLVM IR中比较基本的数据类型包括：

- 空类型（`void`）
- 整型（`iN`）
- 浮点型（`float`、`double`等）

空类型一般是作为不返回值的函数的返回类型，没有特别的含义，就代表「什么都没有」。

整型是指`i1`，`i8`，`i16`，`i32`，`i64`这类的数据类型。这里`iN`的`N`可以是任意正整数，可以是`i3`，`i1942652`。但最常用，最符合常理的就是`i1`以及8的整数倍。`i1`有两个值：`true`和`false`。也就是说，下面的代码可以正确编译：

```
%boolean_variable = alloca i1
store i1 true, i1* %boolean_variable
```

对于大于1位的整型，也就是如`i8`，`i16`等类型，我们可以直接用数字字面量赋值：

```
%integer_variable = alloca i32
store i32 128, i32* %integer_variable
store i32 -128, i32* %integer_variable
```

符号

有一点需要注意的是，在LLVM IR中，整型默认是有符号整型，也就是说我们可以直接将`-128`以补码形式赋值给`i32`类型的变量。在LLVM IR中，整型的有无符号是体现在操作指令而非类型上的，比方说，对于两个整型变量的除法，LLVM IR分别提供了`udiv`和`sdiv`指令分别适用于无符号整型除法和有符号整型除法：

```
%1 = udiv i8 -6, 2  ; get (256 - 6) / 2 = 125
%2 = sdiv i8 -6, 2  ; get (-6) / 2 = -3
```

我们可以用这样一个简单的程序验证：

```
; div_test.ll
target datalayout = "e-m:o-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-apple-macosx10.15.0"

define i8 @main() {
    %1 = udiv i8 -6, 2
    %2 = sdiv i8 -6, 2

    ret i8 %1
}
```

分别将`ret`语句的参数换成`%1`和`%2`以后，将代码编译成可执行文件，在终端下运行并查看返回值即可。

总结一下就是，LLVM IR中的整型默认按有符号补码存储，但一个变量究竟是否要被看作有无符号数需要看其参与的指令。

转换指令

与整型密切相关的就是转换指令，比如说，将`i8`类型的数`-127`转换成`i32`类型的数，将`i32`类型的数`257`转换成`i8`类型的数等。总的来说，LLVM IR中提供三种指令：`trunc` .. `to`指令，`zext` .. `to`指令和`sext` .. `to`指令。

将长的整型转换成短的整型很简单，直接把多余的高位去掉就行，LLVM IR提供的是`trunc` .. `to`指令：

```
%trunc_integer = trunc i32 257 to i8 ; trunc 32 bit 100000001 to 8 bit, get 1
```

将短的整型变成长的整型则相对比较复杂。这是因为，在补码中最高位是符号位，并不表示实际的数值。因此，如果单纯地在更高位补`0`，那么`i8`类型的`-1`（补码为`11111111`）就会变成`i32`的`255`。这虽然符合道理，但有时候我们需要`i8`类型的`-1`扩展到`i32`时仍然是`-1`。LLVM IR为我们提供了两种指令：零扩展的`zext` .. `to`指令和符号扩展的`sext` .. `to`指令。

零扩展就是最简单的，直接在高位补`0`，而符号扩展则是用原数的符号位来填充。也就是说我们如下的代码：

```
%zext_integer = zext i8 -1 to i32 ; extend 8 bit 0xFF to 32 bit 0x000000FF, get 255
%sext_integer = sext i8 -1 to i32 ; extend 8 bit 0xFF to 32 bit 0xFFFFFFFF, get -1
```

类似地，浮点型的数和整型的数也可以相互转换，使用`fptoui` .. `to`, `fptosi` .. `to`, `uitofp` .. `to`, `sitofp` .. `to`可以分别将浮点数转换为无符号、有符号整型，将无符号、有符号整型转换为浮点数。不过有一点要注意的是，如果将大数转换为小的数，那么并不保证截断，如将浮点型的`257.1`转换成`i8`（上限为`128`），那么就会产生未定义行为。所以，在浮点型和整型相互转换的时候，需要在高级语言层面做一些调整，如使用饱和转换等，具体方案可以看Rust最近1.45.0的更新Announcing Rust

1.45.0和GitHub上的PR：Out of range float to int conversions using `as` has been defined as a saturating conversion.。

指针类型

将基本的数据类型后加上一个`*`就变成了指针类型`i8*`，`i16*`，`float*`等。我们之前提到，LLVM IR中的全局变量和栈上分配的变量都是指针，所以其类型都是指针类型。

在高级语言中，直接操作裸指针的机会都比较少，除非在性能极其敏感的场景下，由最厉害的大佬才能操作裸指针。这是因为，裸指针极其危险，稍有不慎就会出现段错误等致命错误，所以我们使用指针时应该慎之又慎。

LLVM IR为大佬们提供了操作裸指针的一些指令。在C语言中，我们会遇到这种场景：

```
int x, y;
size_t address_of_x = (size_t)&x;
size_t address_of_y = address_of_x - sizeof(int);
int also_y = *(int *)address_of_y;
```

这种场景比较无脑，但确实是合理的，需要将指针看作一个具体的数值进行加减。到x86_64的汇编语言层次，取地址就变成了`lea`命令，解引用倒是比较正常，就是一个简单的`mov`。

在LLVM IR层次，为了使指针能像整型一样加减，提供了`ptrtoint` .. `to`指令和`inttoptr` .. `to`指令，分别解决将指针转换为整型，和将整型转换为指针的功能。也就是说，我们可以粗略地将上面的程序转写为

```
%x = alloca i32 ; %x is of type i32*, which is the address of variable x
%y = alloca i32 ; %y is of type i32*, which is the address of variable y
%address_of_x = ptrtoint i32* %x to i64
%address_of_y = sub i64 %address_of_x, 4
%also_y = inttoptr i64 %address_of_y to i32* ; %also_y is of type i32*,
which is the address of variable y
```

聚合类型

比起指针类型而言，更重要的是聚合类型。我们在C语言中常见的聚合类型有数组和结构体，LLVM IR也为我们提供了相应的支持。

数组类型很简单，我们要声明一个类似C语言中的`int a[4]`，只需要

```
%a = alloca [4 x i32]
```

也就是说，C语言中的`int[4]`类型在LLVM IR中可以写成`[4 x i32]`。注意，这里面是个`x`不是`*`。

我们也可以使用类似地语法进行初始化：

```
@global_array = global [4 x i32] [i32 0, i32 1, i32 2, i32 3]
```

特别地，我们知道，字符串在底层可以看作字符组成的数组，所以LLVM IR为我们提供了语法糖：

```
@global_string = global [12 x i8] c"Hello world\00"
```

在字符串中，转义字符必须以`\xy`的形式出现，其中`xy`是这个转义字符的ASCII码。比如说，字符串的结尾，C语言中的`\0`，在LLVM IR中就表现为`\00`。

结构体的类型也相对比较简单，在C语言中的结构体

```
struct MyStruct {
    int x;
    char y;
};
```

在LLVM IR中就成了

```
%MyStruct = type {
    i32,
    i8
}
```

我们初始化一个结构体也很简单：

```
@global_structure = global %MyStruct { i32 1, i8 0 }
; or
@global_structure = global { i32, i8 } { i32 1, i8 0 }
```

值得注意的是，无论是数组还是结构体，其作为全局变量或栈上变量，依然是指针，也就是说，`@global_array`的类型是`[4 x i32]*`，`@global_structure`的类型是`%MyStruct*`也就是`{ i32, i8 }*`。接下来的问题就是，我们如何对聚合类型进行操作呢？

getelementptr

首先，我们要讲的是对聚合类型的指针进行操作。一个最全面的例子，用C语言来说，就是

```
struct MyStruct {
    int x;
    int y;
};

struct MyStruct my_structs[4];
```

我们有一个长度为4的`MyStruct`类型的数组`my_structs`，我们需要的是`my_structs[2].y`这个数。

我们先直接看结论，用LLVM IR来表示为

```
%MyStruct = type {
    i32,
    i32
}
%my_structs = alloca [4 x %MyStruct]

%1 = getelementptr [4 x %MyStruct], [4 x %MyStruct]* %my_structs, i64 2,
i32 1 ; %1 is pointer to my_structs[2].y
%2 = load i32, i32* %1 ; %2 is value of my_structs[2].y
```

核心就在于`getelementptr`这个指令。这个指令的前两个参数很显然，第一个是这个聚合类型的类型，第二个则是这个聚合类型对象的指针，也就是我们的`my_structs`。第三个参数，则是指明在数组中的第几个元素，第四个，则是指明在结构体中的第几个字段（LLVM IR中结构体的字段不是按名称，而是按下标索引来区分）。用人话来说，`%1`就是`my_structs`数组第2个元素的第1个字段的地址。

这看上去似乎很好理解，但是，下面的例子就似乎有些特殊了：

```
%MyStruct = type {
    i32,
    i32
}
%my_struct = alloca %MyStruct

%1 = getelementptr %MyStruct, %MyStruct* %my_struct, i64 0, i32 1 ; %1 is
pointer to my_struct.y
```

没想到吧，如果想根据结构体的指针获取结构体的字段，`getelementptr`的第三个参数居然还需要一个`i64 0`。这是做什么用的呢？这里就是指数组的第一个元素，想象一下我们有一个C语言代码：

```
struct MyStruct {
    int x;
    int y;
};

struct MyStruct my_struct;
struct MyStruct* my_struct_ptr = &my_struct;
int *y_ptr = my_struct_ptr[0].y;
```

这里的`my_struct_ptr[0]`就代表了我们`getelementptr`的第三个参数，这万万不可省略。

此外，`getelementptr`还可以接多个参数，类似于级联调用。我们有C程序：

```
struct MyStruct {
    int x;
    int y[5];
};

struct MyStruct my_structs[4];
```

那么如果我们想获得`my_structs[2].y[3]`的地址，只需要

```
%MyStruct = type {
    i32,
    [5 x i32]
}
%my_structs = alloca [4 x %MyStruct]

%1 = getelementptr [4 x %MyStruct], [4 x %MyStruct]* %my_structs, i64 2,
i32 1, i64 3
```

我们可以查看官方提供的The Often Misunderstood GEP Instruction指南更多地了解`getelementptr`的机理。

`extractvalue`和`insertvalue`

除了我们上面讲的这种情况，也就是把结构体分配在栈或者全局变量，然后操作其指针以外，还有什么情况呢？我们考虑这种情况：

```
; extract_insert_value.ll
%MyStruct = type {
    i32,
    i32
}
@my_struct = global %MyStruct { i32 1, i32 2 }

define i32 @main() {
    %1 = load %MyStruct, %MyStruct* @my_struct

    ret i32 0
}
```

这时，我们的结构体是直接放在虚拟寄存器`%1`里，`%1`并不是存储`@my_struct`的指针，而是直接存储这个结构体的值。这时，我们并不能用`getelementptr`来操作`%1`，因为这个指令需要的是一个指针。因此，LLVM IR提供了`extractvalue`和`insertvalue`指令。

因此，如果要获得`@my_struct`第二个字段的值，我们需要

```
%2 = extractvalue %MyStruct %1, 1
```

这里的`1`就代表第二个字段（从`0`开始）。

类似地，如果要将`%1`的第二个字段赋值为`233`，只需要

```
%3 = insertvalue %MyStruct %1, i32 233, 1
```

然后`%3`就会是`%1`将第二个字段赋值为`233`后的值。

`extractvalue`和`insertvalue`并不只适用于结构体，也同样适用于存储在虚拟寄存器中的数组，这里不再赘述。

标签类型

在汇编语言中，一切的控制语句、函数调用都是由标签来控制的，在LLVM IR中，控制语句也是需要标签来完成。其具体的内容我会在之后专门有一篇控制语句的文章来解释。

元数据类型

在我们使用Clang将C语言程序输出成LLVM IR时，会发现代码的最后几行有

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

!0 = !{i32 2, !"SDK Version", [3 x i32] [i32 10, i32 15, i32 4]}
!1 = !{i32 1, !"wchar_size", i32 4}
!2 = !{i32 7, !"PIC Level", i32 2}
!3 = !{!"Apple clang version 11.0.3 (clang-1103.0.32.62)"}
```

类似于这样的东西。

在LLVM IR中，以`!`开头的标识符为元数据。元数据是为了将额外的信息附加在程序中传递给LLVM后端，使后端能够好地优化或生成代码。用于Debug的信息就是通过元数据形式传递的。我们可以使用`-g`选项：

```
clang -S -emit-llvm -g test.c
```

来在LLVM IR中附加额外的Debug信息。

LLVM IR的语法指南中有专门的一大章Metadata来解释各种元数据，这里与我们核心内容联系不太密切，我就不再赘述了。

属性

最后，还有一种叫做属性的概念。属性并不是类型，其一般用于函数。比如说，告诉编译器这个函数不会抛出错误，不需要某些优化等等。我们可以看到

```
define void @foo() nounwind {
    ; ...
}
```

这里`nounwind`就是一个属性。

有时候，一个函数的属性会特别特别多，并且有多个函数都有相同的属性。那么，就会有大量重复的篇幅用来给每一个函数说明属性。因此，LLVM IR引入了属性组的概念，我们在将一个简单的C程序编译成LLVM IR时，会发现代码中有

```
attributes #0 = { noinline nounwind optnone ssp uwtable "correctly-rounded-divide-sqrt-fp-math"="false" "darwin-stkchk-strong-link" "disable-tail-calls"="false" "frame-pointer"="all" "less-precise-fpmad"="false" "min-legal-vector-width"="0" "no-infs-fp-math"="false" "no-jump-tables"="false" "no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false" "no-trapping-math"="false" "probe-stack"="___chkstk_darwin" "stack-protector-buffer-size"="8" "target-cpu"="penryn" "target-features"="+cx16,+cx8,+fxsr,+mmx,+sahf,+sse,+sse2,+sse3,+sse4.1,+ssse3,+x87" "unsafe-fp-math"="false" "use-soft-float"="false" }
```

这种一大长串的，就是属性组。属性组总是以`#`开头。当我们函数需要它的时候，只需要

```
define void @foo #0 {
    ; ...
}
```

直接使用`#0`即可。

在哪可以看到我的文章

我的LLVM IR入门指南系列可以在我的个人博客、GitHub：Evian-Zhang/llvm-ir-tutorial、知乎、CSDN中查看，本教程中涉及的大部分代码也都在同一GitHub仓库中。

本人水平有限，写此文章仅希望与大家分享学习经验，文章中必有缺漏、错误之处，望方家不吝斧正，与大家共同学习，共同进步，谢谢大家！

控制语句

我在之前汇编语言的教程中，是将跳转与函数放在一起讲的，因为在汇编语言中这两个概念几乎没有太大的区别。然而，到了LLVM IR中，这两者就有了比较大的区别。因此，在这篇文章中，我主要讲的是LLVM IR中控制语句的构造方法。

汇编层面的控制语句

在大多数语言中，常见的控制语句主要有四种：

- `if .. else`
- `for`
- `while`
- `switch`

在汇编语言层面，控制语句则被分解为两种核心的指令：条件跳转与无条件跳转（`switch`其实还有一些工作，之后会提到）。我们下面分别来看看在汇编层面是怎样实现控制语句的。

`if .. else`

我们有以下C代码：

```
if (a > b) {
    // do something A
} else {
    // do something B
}
// do something C
```

为了将这个指令改写成汇编指令，我们同时需要条件跳转与无条件跳转。我们用伪代码表示其汇编指令为：

```
    compare a and b
    jump to label B if comparison is a is not greater than b // conditional jump
label A:
    do something A
    jump to label C // unconditional jump
label B:
    do something B
label C:
    do something C
```

汇编语言通过条件跳转、无条件跳转和三个标签（`label A`标签实际上没有作用，只不过让代码更加清晰）实现了高级语言层面的`if .. else`语句。

`for`

我们有以下C代码：

```c
for (int i = 0; i < 4; i++) {
    // do something A
}
// do something B
```

为了将这个指令改写为汇编指令，我们同样地需要条件跳转与无条件跳转：

```
    int i = 0
label start:
    compare i and 4
    jump to label B if comparison is i is not less than 4 // conditional jump
label A:
    do something A
    i++
    jump to label start // unconditional jump
label B:
    do something B
```

而`while`与`for`则极其类似，只不过少了初始化与自增的操作，这里不再赘述。

根据我们在汇编语言中积累的经验，我们得出，要实现大多数高级语言的控制语句，我们需要四个东西：

- 标签
- 无条件跳转
- 比较大小的指令
- 条件跳转

LLVM IR层面的控制语句

下面就以我们上面的`for`循环的C语言版本为例，解释如何写其对应的LLVM IR语句。

首先，我们对应的LLVM IR的基本框架为

```
%i = alloca i32 ; int i = ...
store i32 0, i32* %i ; ... = 0
%i_value = load i32, i32* %i
; do something A
%1 = add i32 %i_value, 1 ; ... = i + 1
store i32 %1, i32* %i ; i = ...
; do something B
```

这个程序缺少了一些必要的步骤，而我们之后会将其慢慢补上。

标签

在LLVM IR中，标签与汇编语言的标签一致，也是以`:`结尾作标记。我们依照之前写的汇编语言的伪代码，给这个程序加上标签：

```
    %i = alloca i32 ; int i = ...
    store i32 0, i32* %i ; ... = 0
start:
    %i_value = load i32, i32* %i
A:
    ; do something A
    %1 = add i32 %i_value, 1 ; ... = i + 1
    store i32 %1, i32* %i ; i = ...
B:
    ; do something B
```

比较指令

LLVM IR提供的比较指令为`icmp`。其接受三个参数：比较方案以及两个比较参数。这样讲比较抽象，我们就来看一下一个最简单的比较指令的例子：

```
%comparison_result = icmp uge i32 %a, %b
```

这个例子转化为C++语言就是

```
bool comparison_result = ((unsigned int)a >= (unsigned int)b);
```

这里，`uge`是比较方案，`%a`和`%b`就是用来比较的两个数，而`icmp`则返回一个`i1`类型的值，也就是C++中的`bool`值，用来表示结果是否为真。

`icmp`支持的比较方案很广泛：

- 首先，最简单的是`eq`与`ne`，分别代表相等或不相等。
- 然后，是无符号的比较`ugt`，`uge`，`ult`，`ule`，分别代表大于、大于等于、小于、小于等于。我们之前在数的表示中提到，LLVM IR中一个整型变量本身的符号是没有意义的，而是需要看在其参与的指令中被看作是什么符号。这里每个方案的`u`就代表以无符号的形式进行比较。
- 最后，是有符号的比较`sgt`，`sge`，`slt`，`sle`，分别是其无符号版本的有符号对应。

我们来看加上比较指令之后，我们的例子就变成了：

```
    %i = alloca i32 ; int i = ...
    store i32 0, i32* %i ; ... = 0
start:
    %i_value = load i32, i32* %i
    %comparison_result = icmp slt i32 %i_value, 4 ; test if i < 4
A:
    ; do something A
    %1 = add i32 %i_value, 1 ; ... = i + 1
    store i32 %1, i32* %i ; i = ...
B:
    ; do something B
```

条件跳转

在比较完之后，我们需要条件跳转。我们来看一下我们此刻的目的：若`%comparison_result`是`true`，那么跳转到`A`，否则跳转到`B`。

LLVM IR为我们提供的条件跳转指令是`br`，其接受三个参数，第一个参数是`i1`类型的值，用于作判断；第二和第三个参数分别是值为`true`和`false`时需要跳转到的标签。比方说，在我们的例子中，就应该是

```
br i1 %comparison_result, label %A, label %B
```

我们把它加入我们的例子：

```
    %i = alloca i32 ; int i = ...
    store i32 0, i32* %i ; ... = 0
start:
    %i_value = load i32, i32* %i
    %comparison_result = icmp slt i32 %i_value, 4 ; test if i < 4
    br i1 %comparison_result, label %A, label %B
A:
    ; do something A
    %1 = add i32 %i_value, 1 ; ... = i + 1
    store i32 %1, i32* %i ; i = ...
B:
    ; do something B
```

无条件跳转

无条件跳转更好理解，直接跳转到某一标签处。在LLVM IR中，我们同样可以使用`br`进行条件跳转。如，如果要直接跳转到`start`标签处，则可以

```
br label %start
```

我们也把这加入我们的例子：

```
    %i = alloca i32 ; int i = ...
    store i32 0, i32* %i ; ... = 0
start:
    %i_value = load i32, i32* %i
    %comparison_result = icmp slt i32 %i_value, 4 ; test if i < 4
    br i1 %comparison_result, label %A, label %B
A:
    ; do something A
    %1 = add i32 %i_value, 1 ; ... = i + 1
    store i32 %1, i32* %i ; i = ...
    br label %start
B:
    ; do something B
```

这样看上去就结束了，然而如果大家把这个代码交给`llc`的话，并不能编译通过，这是为什么呢?

Basic block

首先，我们来摘录一下LLVM IR的参考指南中Functions节的一段话：

> A function definition contains a list of basic blocks, forming the CFG (Control Flow Graph) for the function. Each basic block may optionally start with a label (giving the basic block a symbol table entry), contains a list of instructions, and ends with a terminator instruction (such as a branch or function return). If an explicit label name is not provided, a block is assigned an implicit numbered label, using the next value from the same counter as used for unnamed temporaries (see above).

这段话的大意有几个：

- 一个函数由许多基本块(Basic block)组成
- 每个基本块包含：
  - 开头的标签（可省略）
  - 一系列指令
  - 结尾是终结指令
- 一个基本块没有标签时，会自动赋给它一个标签

所谓终结指令，就是指改变执行顺序的指令，如跳转、返回等。

我们来看看我们之前写好的程序是不是符合这个规定。`start`开头的基本块，在一系列指令后，以

```
br i1 %comparison_result, label %A, label %B
```

结尾，是一个终结指令。`A`开头的基本块，在一系列指令后，以

```
br label %start
```

结尾，也是一个终结指令。`B`开头的基本块，在最后总归是需要函数返回的（这里为了篇幅省略了），所以也一定会带有一个终结指令。

看上去都很符合呀，那为什么编译不通过呢？我们来仔细想一下，我们考虑了所有基本块了吗？要注意到，一个基本块是可以没有名字的，所以，实际上还有一个基本块没有考虑到，就是函数开头的：

```
%i = alloca i32 ; int i = ...
store i32 0, i32* %i ; ... = 0
```

这个基本块。它并没有以终结指令结尾！

所以，我们把一个终结指令补充在这个基本块的结尾：

```
    %i = alloca i32 ; int i = ...
    store i32 0, i32* %i ; ... = 0
    br label %start
start:
    %i_value = load i32, i32* %i
    %comparison_result = icmp slt i32 %i_value, 4 ; test if i < 4
    br i1 %comparison_result, label %A, label %B
A:
    ; do something A
    %1 = add i32 %i_value, 1 ; ... = i + 1
    store i32 %1, i32* %i ; i = ...
    br label %start
B:
    ; do something B
```

这样就完成了我们的例子，大家可以在本系列的GitHub的仓库中查看对应的代码`for.ll`。

可视化

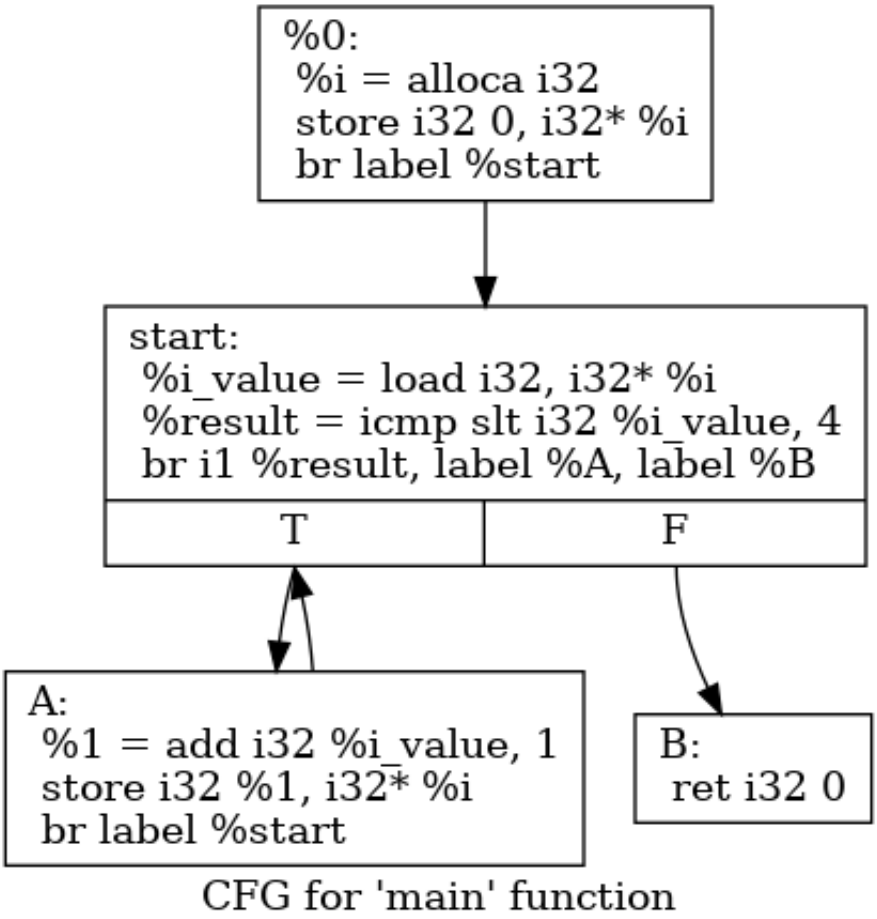
LLVM的工具链甚至为我们提供了可视化控制语句的方法。我们使用之前提到的LLVM工具链中用于优化的`opt`工具：

```
opt -dot-cfg for.ll
```

然后会生成一个`.main.dot`的文件。如果我们在计算机上装有Graphviz，那么就可以用

```
dot .main.dot -Tpng -o for.png
```

生成其可视化的控制流图（CFG）：



CFG for 'main' function

switch

下面我们来讲讲`switch`语句。我们有以下C语言程序：

```c
int x;
switch (x) {
    case 0:
        // do something A
        break;
    case 1:
        // do something B
        break;
    default:
        // do something C
        break;
}
// do something else
```

我们先直接来看其转换成LLVM IR是什么样子的：

```
switch i32 %x, label %C [
    i32 0, label %A
    i32 1, label %B
]
```

```
A:
    ; do something A
    br label %end
B:
    ; do something B
    br label %end
C:
    ; do something C
    br label %end
end:
    ; do something else
```

其核心就是第一行的`switch`指令。其第一个参数`i32 %x`是用来判断的，也就是我们C语言中的`x`。第二个参数`label %C`是C语言中的`default`分支，这是必须要有的参数。也就是说，我们的`switch`必须要有`default`来处理。接下来是一个数组，其意义已经很显然了，如果`%x`值是`0`，就去`label %A`，如果值是`1`，就去`label %B`。

LLVM后端对`switch`语句具体到汇编层面的实现则通常有两种方案：用一系列条件语句或跳转表。

一系列条件语句的实现方式最简单，用伪代码来表示的话就是

```
if (x == 0) {
    jump to label %A
} else if (x == 1) {
    jump to label %B
} else {
    jump to label %C
}
```

这是十分符合常理的。然而，我们需要注意到，如果这个`switch`语句一共有n个分支，那么其查找效率实际上是O(n)。那么，这种实现方案下的`switch`语句仅仅是`if .. else`的语法糖，除了增加可维护性，并不会优化什么性能。

跳转表则是一个可以优化性能的`switch`语句实现方案，其伪代码为：

```
labels = [label %A, label %B]
if (x < 0 || x > 1) {
    jump to label %C
} else {
    jump to labels[x]
}
```

这只是一个极其粗糙的近似的实现，我们需要的是理解其基本思想。跳转表的思想就是利用内存中数组的索引是O(1)复杂度的，所以我们可以根据目前的`x`值去查找应该跳转到哪一个地址，这就是跳转表的基本思想。

根据目标平台和`switch`语句的分支数，LLVM后端会自动选择不同的实现方式去实现`switch`语句。

`select`

我们经常会遇到一种情况，某一变量的值需要根据条件进行赋值，比如说以下C语言的函数：

```c
void foo(int x) {
    int y;
    if (x > 0) {
        y = 1;
    } else {
        y = 2;
    }
    // do something with y
}
```

如果`x`大于`0`，则`y`为`1`，否则`y`为`2`。这一情况很常见，然而在C语言中，如果要实现这种功能，`y`需要被实现为可变变量，但实际上无论`x`如何取值，`y`只会被赋值一次，并不应该是可变的。

我们知道，LLVM IR中，由于SSA的限制，局部可变变量都必须分配在栈上，虽然LLVM后端最终会进行一定的优化，但写起代码来还需要冗长的`alloca`，`load`，`store`等语句。如果我们按照C语言的思路来写LLVM IR，那么就会是：

```
define void @foo(i32 %x) {
    %y = alloca i32
    %1 = icmp sgt i32 %x, 0
    br i1 %1, label %btrue, label %bfalse
btrue:
    store i32 1, i32* %y
    br label %end
bfalse:
    store i32 2, i32* %y
    br label %end
end:
    ; do something with %y
    ret void
}
```

我们来看看其编译出的汇编语言是怎样的：

```
_foo:
    cmpl    $0, %edi
    jle LBB0_2
### %bb.1:                                 ## %btrue
    movl    $1, -4(%rsp)
    jmp LBB0_3
LBB0_2:                                   ## %bfalse
    movl    $2, -4(%rsp)
LBB0_3:                                   ## %end
    # do something with -4(%rsp)
    retq
```

C语言代码9行，汇编语言代码11行，LLVM IR代码14行。这LLVM IR同时比低层次和高层次的代码都长，这显然是不可以接受的。究其原因，就是这里把`y`看成了可变变量。那么，有没有什么办法让`y`不可变但仍然能实现这个功能呢?

首先，我们来看看同样区分可变变量与不可变变量的Rust是怎么做的：

```
fn foo(x: i32) {
    let y = if x > 0 { 1 } else { 2 };
    // do something with y
}
```

让代码简短的方式很简单，把`y`看作不可变变量，但同时需要语言支持把`if`语句视作表达式，当`x`大于`0`时，这个表达式返回`1`，否则返回`2`。这样，就很简单地实现了我们的需求。

LLVM IR中同样也有这样的指令，那就是`select`，我们来把上面的例子用`select`改写：

```
define void @foo(i32 %x) {
    %result = icmp sgt i32 %x, 0
    %y = select i1 %result, i32 1, i32 2
    ; do something with %y
}
```

`select`指令接受三个参数。第一个参数是用来判断的布尔值，也就是`i1`类型的`icmp`判断的结果，如果其为`true`，则返回第二个参数，否则返回第三个参数。极其合理。

phi

`select`只能支持两个选择，`true`选择一个分支，`false`选择另一个分支，我们是不是可以有支持多种选择的类似`switch`的版本呢? 同时，我们也可以换个角度思考，`select`是根据`i1`的值来进行判断，我们其实可以根据控制流进行判断。这就是传说中的`phi`指令。

为了方便起见，我们首先先来看用`phi`指令实现的我们上面这个代码：

```
define void @foo(i32 %x) {
    %result = icmp sgt i32 %x, 0
    br i1 %result, label %btrue, label %bfalse
btrue:
    br label %end
bfalse:
    br label %end
end:
    %y = phi i32 [1, %btrue], [2, %bfalse]
    ; do something with %y
    ret void
}
```

我们看到，`phi`的第一个参数是一个类型，这个类型表示其返回类型为`i32`。接下来则是两个数组，其表示，如果当前的basic block执行的时候，前一个basic block是`%btrue`，那么返回`1`，如果前一个basic block是`%bfalse`，那么返回`2`。也就是说，`select`是根据其第一个参数`i1`类型的变量的值来决定返回哪个值，而`phi`则是根据其之前是哪个basic block来决定其返回值。此外，`phi`之后可以跟无数的分支，如`phi i32 [1, %a], [2, %b], [3, %c]`等，从而可以支持多分支的赋值。

在哪可以看到我的文章

我的LLVM IR入门指南系列可以在我的个人博客、GitHub：Evian-Zhang/llvm-ir-tutorial、知乎、CSDN中查看，本教程中涉及的大部分代码也都在同一GitHub仓库中。

本人水平有限，写此文章仅希望与大家分享学习经验，文章中必有缺漏、错误之处，望方家不吝斧正，与大家共同学习，共同进步，谢谢大家！

函数

这篇文章我们来讲函数。有过汇编基础的同学都知道，在汇编层面，一个函数与一个控制语句极其相似，都是由标签组成，只不过在跳转时增加了一些附加的操作。而在LLVM IR层面，函数则得到了更高一层的抽象。

定义与声明

函数定义

在LLVM中，一个最基本的函数定义的样子我们之前已经遇到过多次，就是`@main`函数的样子：

```
define i32 @main() {
    ret i32 0
}
```

在函数名之后可以加上参数列表，如：

```
define i32 @foo(i32 %a, i64 %b) {
    ret i32 0
}
```

一个函数定义最基本的框架，就是返回值（`i32`）+函数名（`@foo`）+参数列表（`(i32 %a, i64 %b)`）+函数体（`{ ret i32 0 }`）。

我们可以看到，函数的名称和全局变量一样，都是以`@`开头的。并且，如果我们查看符号表的话，也会发现其和全局变量一样进入了符号表。因此，函数也有和全局变量完全一致的Linkage Types和Visibility Style，来控制函数名在符号表中的出现情况，因此，可以出现如

```
define private i32 @foo() {
    ; ...
}
```

这样的修饰符。

此外，我们还可以在参数列表之后加上之前说的属性，也就是控制优化器和代码生成器的指令。在之前讲属性组的时候，我就提过，如果我们单纯编译一个简单的C代码：

```
void foo() { }
int main() {
    return 0;
}
```

可以看到`@foo`函数之后会跟上一个属性组`#0`，在macOS下其内容为

```
attributes #0 = { noinline nounwind optnone ssp uwtable "correctly-rounded-divide-sqrt-fp-math"="false" "darwin-stkchk-strong-link" "disable-tail-calls"="false" "frame-pointer"="all" "less-precise-fpmad"="false" "min-legal-vector-width"="0" "no-infs-fp-math"="false" "no-jump-tables"="false" "no-nans-fp-math"="false" "no-signed-zeros-fp-math"="false" "no-trapping-math"="false" "probe-stack"="___chkstk_darwin" "stack-protector-buffer-size"="8" "target-cpu"="penryn" "target-features"="+cx16,+cx8,+fxsr,+mmx,+sahf,+sse,+sse2,+sse3,+sse4.1,+ssse3,+x87" "unsafe-fp-math"="false" "use-soft-float"="false" }
```

我们通过对一个函数附加许多属性，来控制最终的优化和代码生成。大部分的属性可以在Function Attributes一节看到，把这些属性放在一起一个一个来讲显然是不明智的举措，我会在之后提到某些与某个属性相关的概念时再阐述某个属性的含义。这里只需要知道，在函数的参数列表之后可以加上属性或属性组，如：

```
define void @foo() nounwind { ret void }
; or
define void @foo() #0 { ret void }
attributes #0 {
    ; ...
}
```

函数声明

除了函数定义之外，还有一种情况十分常见，那就是函数声明。我们在一个编译单元（模块）下，可以使用别的模块的函数，这时候就需要在本模块先声明这个函数，才能保证编译时不出错，从而在链接时正确将声明的函数与别的模块下其定义进行链接。

函数声明也相对比较简单，就是使用`declare`关键词替换`define`：

```
declare i32 @printf(i8*, ...) #1
```

这个就是在C代码中调用`stdio.h`库的`printf`函数时，在LLVM IR代码中可以看到的函数声明，其中`#1`就是又一大串属性组成的属性组。

函数的调用

在LLVM IR中，函数的调用与高级语言几乎没有什么区别：

```
define i32 @foo(i32 %a) {
    ; ...
}

define void @bar() {
    %1 = call i32 @foo(i32 1)
}
```

使用`call`指令可以像高级语言那样直接调用函数。我们来仔细分析一下这里做了哪几件事：

- 传递参数
- 执行函数
- 获得返回值

居然能干这么多事，这是汇编语言所羡慕不已的。

执行函数

我们知道，如果一个函数没有任何参数，返回值也是`void`类型，也就是说在C语言下这个函数是

```
void foo() {
    // ...
}
```

那么调用这个函数就没有了传递参数和获得返回值这两件事，只剩下执行函数，而这是一个最简单的事：

1. 把函数返回地址压栈
2. 跳转到相应函数的地址

函数返回也是一个最简单的事：

1. 弹栈获得函数返回地址
2. 跳转到相应的返回地址

这个在我们的汇编语言基础中已经反复遇到过多次，相信大家都会十分熟练。

传递参数与获得返回值

谈到这两点，就不得不说调用约定了。我们知道，在汇编语言中，是没有参数传递和返回值的概念的，有的仅仅是让当前的控制流跳转到指定函数执行。所以，一切的参数传递和返回值都需要我们人为约定。也就是说，我们需要约定两件事：

- 被调用的函数希望知道参数是放在哪里的
- 调用者希望知道调用函数的返回值是放在哪里的

这就是调用约定。不同的调用约定会产生不同的特效，也就产生了许多高级语言的feature。

C调用约定

最广泛使用的调用约定是C调用约定，也就是各个操作系统的标准库使用的调用约定。在x86_64架构下，C调用约定是System V版本的，所有参数按顺序放入指定寄存器，如果寄存器不够，剩余的则从右往左顺序压栈。而返回值则是按先后顺序放入寄存器或者放入调用者分配的空间中，如果只有一个返回值，那么就会放在`rax`里。

在LLVM IR中，函数的调用默认使用C调用约定。为了验证，我们可以写一个简单的程序：

```
; calling_convention_test.ll
%ReturnType = type { i32, i32 }
define %ReturnType @foo(i32 %a1, i32 %a2, i32 %a3, i32 %a4, i32 %a5, i32 %a6, i32 %a7, i32 %a8) {
    ret %ReturnType { i32 1, i32 2 }
}

define i32 @main() {
    %1 = call %ReturnType @foo(i32 1, i32 2, i32 3, i32 4, i32 5, i32 6, i32 7, i32 8)
    ret i32 0
}
```

我们在x86_64架构的macOS上查看其编译出来的汇编代码。在`main`函数中，参数传递是：

```
movl    $1, %edi
movl    $2, %esi
movl    $3, %edx
movl    $4, %ecx
movl    $5, %r8d
movl    $6, %r9d
movl    $7, (%rsp)
movl    $8, 8(%rsp)
callq   _foo
```

而在`foo`函数内部，返回值传递是：

```
movl    $1, %eax
movl    $2, %edx
retq
```

如果大家去查阅System V的指南的话，会发现完全符合。

这种System V的调用约定有什么好处呢？其最大的特点在于，当寄存器数量不够时，剩余的参数是按**从右向左**的顺序压栈。这就让基于这种调用约定的高级语言可以更轻松地实现可变参数的feature。所谓可变参数，最典型的例子就是C语言中的`printf`：

```
printf("%d %d %d %d", a, b, c, d);
```

`printf`可以接受任意数量的参数，其参数的数量是由第一个参数`"%d %d %d %d"`决定的。有多少个需要格式化的变量，接下来就还有多少个参数。

那么，System V的调用约定又是为什么能满足这样的需求呢？假设我们不考虑之前传入寄存器内的参数，只考虑压入栈内的参数。那么，如果是从右往左的顺序压栈，栈顶就是`"%d %d %d %d"`的地址，接着依次是`a`，`b`，`c`，`d`。那么，我们的程序就可以先读栈顶，获得字符串，然后确定有多少个参数，接着就继续在栈上读多少个参数。相反，如果是从左往右顺序压栈，那么程序第一个读到的是`d`，程序也不知道该读多少个参数。

fastcc

各种语言的调用约定还有许多，可以参考语言指南的Calling Conventions一节。把所有的调用约定都讲一遍显然是不可能且枯燥的。所以，我在这里除了C调用约定之外，只再讲一个调用约定fastcc，以体现不同的调用约定能实现不同的高级语言的feature。

fastcc方案是将变量全都传入寄存器中的方案。这种方案使尾调用优化能更方便地实现。

尾调用会出现在很多场景下，用一个比较平凡的例子：

```
int foo(int a) {
    if (a == 1) {
        return 1;
    } else {
        return foo(a - 1);
    }
}
```

我们注意到，这个函数在返回时有可能会调用自身，这就叫尾调用。为什么尾调用需要优化呢？我们知道，在正常情况下，调用一个函数会产生函数的栈帧，也就是把函数的参数传入栈，把函数的返回地址传入栈。那么如果`a`很大，那么调用的函数会越来越多，并且直到最后一个被调用的函数返回之前，所有调用的函数的栈都不会回收，也就是说，我们此时栈上充斥着一层一层被调用函数返回的地址。

然而，由于这个函数是在调用者的返回语句里调用，我们实际上可以复用调用者的栈，这就是尾调用优化的基础思想。我们希望，把这样的尾调用变成循环，从而减少栈的使用。通过将参数都传入寄存器，我们可以避免再将参数传入栈，这就是fastcc为尾调用优化提供的帮助。然后，就可以直接将函数调用变成汇编中的`jmp`。

我们来看如果用fastcc调用约定，LLVM IR该怎么写：

```
; tail_call_test.ll
define fastcc i32 @foo(i32 %a) {
    %res = icmp eq i32 %a, 1
    br i1 %res, label %btrue, label %bfalse
btrue:
    ret i32 1
bfalse:
    %sub = sub i32 %a, 1
    %tail_call = tail call fastcc i32 @foo(i32 %sub)
    ret i32 %tail_call
}
```

我们使用`llc`对其编译，并加上`-tailcallopt`的指令（实际上不加也没关系，LLVM后端会自动进行Sibling call optimization）：

```
llc tail_call_test.ll -tailcallopt
```

其编译而成的汇编代码中，其主体为：

```
_foo:
    pushq   %rax
    cmpl    $1, %edi
    jne .LBB0_2
    movl    %1, %eax
    popq    %rcx
    retq    $8
.LBB0_2:
    decl    %edi
    popq    %rax
    jmp _foo
```

我们可以发现，在结尾，使用的是`jmp`而不是`call`，所以从高级语言的角度，就可以看作其将尾部的调用变成了循环。并且，有三个动作：`pushq %ra`，`popq %rcx`和`popq %rax`。这三个操作只是为了栈对齐，具体可以参考stack overflow上的回答[Why does this function push RAX to the stack as the first operation?]。

帧指针清除优化

最后，再讲一个函数调用中的优化，就是帧指针清除优化（Frame Pointer Elimination）。

在讲这个之前，先讲一个比较小的优化。我们将一个非常简单的C程序

```
void foo(int a, int b) { }
int main() {
    foo(1, 2);
    return 0;
}
```

编译为汇编程序，可以发现，`foo`函数的汇编代码为：

```
_foo:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -4(%rbp)
    movl    %esi, -8(%rbp)
    popq    %rbp
    retq
```

与我们常识有些违背。为啥这里栈不先增加（也就是对`rsp`寄存器进行`sub`），就直接把`rdi`，`esi`的值移入栈内了呢？`-4(%rbp)`和`-8(%rbp)`的内存空间此刻似乎并不属于栈。这是因为，我们对栈指针进行操作，一个很重要的原因就是为了进一步函数调用的时候，使用`call`指令会自动将被调用函数的返回地址压栈，那么就需要在调用`call`指令之前，保证栈顶指针确实指向栈顶，否则压栈就会覆盖一些数据。

但此时，我们的`foo`函数并没有调用别的函数，也就不会产生压栈行为，从而编译器自动为我们省去了这样的操作。为了验证这一点，我们做一个小的修改：

```
void bar() { }
void foo(int a, int b) { bar(); }
int main() {
    foo(1, 2);
    return 0;
}
```

这时，我们再看编译出的`foo`函数的汇编代码：

```
_foo:
    pushq   %rbp
    movq    %rsp, %rbp
    subq    $16, %rsp
    movl    %edi, -4(%rbp)
    movl    %esi, -8(%rbp)
    callq   _bar
    addq    $16, %rsp
    popq    %rbp
    retq
```

确实增加了对`rbp`的`sub`和`add`操作。而此时的`bar`函数，也没有对`rsp`的操作。

接下来，就要讲帧指针清除优化了。经过我们上述的讨论，一个函数在进入时会有一些固定动作：

1. 把`rbp`压栈
2. 把`rsp`放入`rbp`
3. 减`rsp`，预留栈空间

在函数返回之前，也有其相应的操作：

1. 加`rsp`，回收栈空间
2. 把`rbp`最初的值弹栈回到`rbp`

我们刚刚讲的优化，使得没有调用别的函数的函数，可以省略掉进入时的第3步和返回前的第1步。那么，是否还可以继续省略呢?

那么，我们就要考虑为什么需要这些步骤。这些步骤都是围绕`rbp`进行的，而正是因为`rbp`经常进行这种操作，所以我们把`rbp`称为帧指针。之所以要进行这些操作，是因为我们在函数执行的过程中，栈顶指针随着不断调用别的函数，会不断移动，导致我们根据栈顶指针的位置，不太方便确定局部变量的位置。而如果我们在一开始就把`rsp`的值放在`rbp`中，那么局部变量的位置相对`rbp`是固定的，就更好确认了。注意到我们这里说根据`rsp`的值确认局部变量的位置只是不方便，但并不是不能做到。所以，我们可以增加一些编译器的负担，而把帧指针清除。

帧指针清除在LLVM IR层面其实十分方便，就是什么都不写。我们可以观察

```
define void @foo(i32 %a, i32 %b) {
    %1 = alloca i32
    %2 = alloca i32
    store i32 %a, i32* %1
    store i32 %b, i32* %2
    ret void
}
```

这个函数在编译成汇编语言之后，是：

```
_foo:
    movl    %edi, -4(%rsp)
    movl    %esi, -8(%rsp)
    retq
```

不仅没有了栈的增加减少（之前提过的优化），也没有了对`rbp`的操作（帧指针清除）。

要想恢复这一操作也十分简单，在函数参数列表后加上一个属性`"frame-pointer"="all"`：

```
define void @foo(i32 %a, i32 %b) "frame-pointer"="all" {
    %1 = alloca i32
    %2 = alloca i32
    store i32 %a, i32* %1
    store i32 %b, i32* %2
    ret void
}
```

其编译后的汇编程序就是：

```
_foo:
    pushq   %rbp
    movq    %rsp, %rbp
    movl    %edi, -4(%rbp)
    movl    %esi, -8(%rbp)
    popq    %rbp
    retq
```

恢复了往日的雄风。

在哪可以看到我的文章

我的LLVM IR入门指南系列可以在我的个人博客、GitHub：Evian-Zhang/llvm-ir-tutorial、知乎、CSDN中查看，本教程中涉及的大部分代码也都在同一GitHub仓库中。

本人水平有限，写此文章仅希望与大家分享学习经验，文章中必有缺漏、错误之处，望方家不吝斧正，与大家共同学习，共同进步，谢谢大家！

异常处理

在这篇文章中，我主要介绍的是LLVM IR中的异常处理的方法。主要的参考文献是Exception Handling in LLVM。

异常处理的要求

异常处理在许多高级语言中都是很常见的，在诸多语言的异常处理的方法中，`try` .. `catch`块的方法是最多的。对于用返回值来做异常处理的语言（如C、Rust、Go等）来说，可以直接在高级语言层面完成所有的事情，但是，如果使用`try` .. `catch`，就必须在语言的底层也做一些处理，而LLVM的异常处理则就是针对这种情况来做的。

首先，我们来看一看一个典型的使用`try` .. `catch`来做的异常处理应该满足怎样的要求。C++就是一个典型的使用`try` .. `catch`来做异常处理的语言，我们就来看看它的异常处理的语法：

```
// try_catch_test.cpp
struct SomeOtherStruct { };
struct AnotherError { };

struct MyError { /* ... */ };
void foo() {
    SomeOtherStruct other_struct;
    throw MyError();
    return;
}

void bar() {
    try {
        foo();
    } catch (MyError err) {
        // do something with err
    } catch (AnotherError err) {
        // do something with err
    } catch (...) {
        // do something
    }
}

int main() {
    return 0;
}
```

这是一串典型的异常处理的代码。我们来看看C++中的异常处理是怎样一个过程（可以参考throw expression和try-block）：

当遇到`throw`语句的时候，控制流会沿着函数调用栈一直向上寻找，直到找到一个`try`块。然后将抛出的异常与`catch`相比较，看看是否被捕获。如果异常没有被捕获，则继续沿着栈向上寻找，直到最终能被捕获，或者整个程序调用`std::terminate`结束。

按照我们上面的例子，控制流在执行`bar`的时候，首先执行`foo`，然后分配了一个局部变量`other_struct`，接着遇到了一个`throw`语句，便向上寻找，在`foo`函数内部没有找到`try`块，就去调用`foo`的`bar`函数里面寻找，发现有`try`块，然后通过对比进入了第一个`catch`块，顺利处理了异常。

这一过程叫stack unwinding，其中有许多细节需要我们注意。

第一，是在控制流沿着函数调用栈向上寻找的时候，会调用所有遇到的自动变量（大部分时候就是函数的局部变量）的析构函数。也就是说，在我们上面的例子里，当控制流找完了`foo`函数，去`bar`函数找之前，就会调用`other_struct`的析构函数。

第二，是如何匹配`catch`块。C++的标准中给出了一长串的匹配原则，在大多数情况下，我们只需要了解，只要`catch`所匹配的类型与抛出的异常的类型相同，或者是引用，或者是抛出异常类型的基类，就算成功。

所以，我们总结出，使用`try` .. `catch`来处理异常，需要考虑以下要求：

- 能够改变控制流
- 能够正确处理栈
- 能够保证抛出的异常结构体不会因为stack unwinding而释放
- 能够在运行时进行类型匹配

LLVM IR的异常处理

下面，我们就看看在LLVM IR层面，是怎么进行异常处理的。

我们要指出，异常处理实际上有许多种形式。我这篇文章主要以Clang对C++的异常处理为例来说的。而这主要是基于Itanium提出的零开销的一种错误处理ABI标准，关于这个的详细的信息，可以参考Itanium C++ ABI: Exception Handling。

首先，我们把上面的`try_catch_test.cpp`代码编译成LLVM IR：

```
clang++ -S -emit-llvm try_catch_test.cpp
```

然后，我们仔细研究一下错误处理。

关于上面的异常处理的需求，我们发现，可以分为两类：怎么抛，怎么接。

怎么抛

所谓怎么抛，就是如何抛出异常，主要需要保证抛出的异常结构体不会因为stack unwinding而释放，并且能够正确处理栈。

对于第一点，也就是让异常结构体存活，我们就需要不在栈上分配它。同时，我们也不能直接裸调用`malloc`等在堆上分配的方法，因为这个结构体也不需要我们手动释放。C++中采用的方法是运行时提供一个API：`__cxa_allocate_exception`。我们可以在`foo`函数编译而成的`@_Z3foov`中看到：

```
define void @_Z3foov() #0 {
    %1 = call i8* @__cxa_allocate_exception(i64 1) #3
    %2 = bitcast i8* %1 to %struct.MyError*
    call void @__cxa_throw(i8* %1, i8* bitcast ({ i8*, i8* }* @_ZTI7MyError
to i8*), i8* null) #4
    unreachable
}
```

第一步就使用了`@__cxa_allocate_exception`这个函数，为我们异常结构体开辟了内存。

然后就是要处理第二点，也就是正确地处理栈。这里的方法是使用另一个C++运行时提供的API：`__cxa_throw`，这个API同时也兼具了改变控制流的作用。这个API开启了我们的stack unwinding。我们可以在libc++abi Specification中看到这个函数的签名：

```
void __cxa_throw(void* thrown_exception, struct std::type_info * tinfo,
void (*dest)(void*));
```

其第一个参数，是指向我们需要抛出的异常结构体的指针，在LLVM IR的代码中就是我们的`%1`。第二个参数，`std::type_info`如果了解C++底层的人就会知道，这是C++的一个RTTI的结构体。简单来讲，就是存储了异常结构体的类型信息，以便在后面`catch`的时候能够在运行时对比类型信息。第三个参数，则是用于销毁这个异常结构体时的函数指针。

这个函数是如何处理栈并改变控制流的呢？粗略来说，它依次做了以下几件事：

1. 把一部分信息进一步储存在异常结构体中
2. 调用`_Unwind_RaiseException`进行stack unwinding

也就是说，用来处理栈并改变控制流的核心，就是`_Unwind_RaiseException`这个函数。这个函数也可以在我上面提供的Itanium的ABI指南中找到。

怎么接

所谓怎么接，就是当stack unwinding遇到`try`块之后，如何处理相应的异常。根据我们上面提出的要求，怎么接应该处理的是如何改变控制流并且在运行时进行类型匹配。

首先，我们来看如果`bar`单纯地调用`foo`，而非在`try`块内调用，也就是

```
void bar() {
    foo();
}
```

编译出的LLVM IR是：

```
define void @_Z3barv() #0 {
    call void @_Z3foov()
    ret void
}
```

和我们正常的不会抛出异常的函数的调用形式一样，使用的是`call`指令。

那么，如果我们代码改成之前的例子，也就是

```
void bar() {
    try {
        foo();
    } catch (MyError err) {
        // do something with err
    } catch (AnotherError err) {
        // do something with err
    } catch (...) {
        // do something
    }
}
```

其编译出的LLVM IR是一个很长很长的函数。其开头是：

```
define void @_Z3barv() #0 personality i8* bitcast (i32 (...)*
@__gxx_personality_v0 to i8*) {
    %1 = alloca i8*
    %2 = alloca i32
    %3 = alloca %struct.AnotherError, align 1
    %4 = alloca %struct.MyError, align 1
    invoke void @_Z3foov()
        to label %5 unwind label %6
    ; ...
}
```

我们发现，传统的`call`调用变成了一个复杂的`invoke` .. `to` .. `unwind`的指令，这个指令是什么意思呢?

`invoke`指令就是我们改变控制流的另一个关键，我们可以在`invoke` instruction中看到其详细的解释。粗略来说，在我们上面编译出的LLVM IR代码中

```
invoke void @_Z3foov() to label %5 unwind label %6
```

这个代码的意思是：

1. 调用`@_Z3foov`函数
2. 判断函数返回的方式：
    - 如果是以`ret`指令正常返回，则跳转至标签`%5`
    - 如果是以`resume`指令或者其他异常处理机制返回（如我们上面所说的`__cxa_throw`函数），则跳转至标签`%6`

所以这个`invoke`指令其实和我们之前在跳转中讲到的`phi`指令很类似，都是根据之前的控制流来进行之后的跳转的。

我们的`%5`的标签很简单，因为原来C++代码中，在`try` .. `catch`块之后啥也没做，就直接返回了，所以其就是简单的

```
5:
    br label %18
18:
    ret void
```

而我们的`catch`的方法，也就是在运行时进行类型匹配的关键，就隐藏在`%6`标签内。

我们通常称在调用函数之后，用来处理异常的代码块为landing pad，而`%6`标签，就是一个landing pad。我们来看看`%6`标签内是怎样的代码：

```
6:
    %7 = landingpad { i8*, i32 }
        catch i8* bitcast ({ i8*, i8* }* @_ZTI7MyError to i8*) ; is MyError or its derived class
        catch i8* bitcast ({ i8*, i8* }* @_ZTI12AnotherError to i8*) ; is AnotherError or its derived class
        catch i8* null ; is other type
    %8 = extractvalue { i8*, i32 } %7, 0
    store i8* %8, i8** %1, align 8
    %9 = extractvalue { i8*, i32 } %7, 1
    store i32 %9, i32* %2, align 4
    br label %10
10:
    %11 = load i32, i32* %2, align 4
    %12 = call i32 @llvm.eh.typeid.for(i8* bitcast ({ i8*, i8* }* @_ZTI7MyError to i8*)) #3
    %13 = icmp eq i32 %11, %12 ; compare if is MyError by typeid
    br i1 %13, label %14, label %19
19:
    %20 = call i32 @llvm.eh.typeid.for(i8* bitcast ({ i8*, i8* }* @_ZTI12AnotherError to i8*)) #3
    %21 = icmp eq i32 %11, %20 ; compare if is Another Error by typeid
    br i1 %21, label %22, label %26
```

说人话的话，是这样一个步骤：

1. `landingpad`将捕获的异常进行类型对比，并返回一个结构体。这个结构体的第一个字段是`i8*`类型，指向异常结构体。第二个字段表示其捕获的类型：
    - 如果是`MyError`类型或其子类，第二个字段为`MyError`的TypeID
    - 如果是`AnotherError`类型或其子类，第二个字段为`AnotherError`的TypeID
    - 如果都不是（体现在`catch i8* null`），第二个字段为`null`的TypeID
2. 根据获得的TypeID来判断应该进哪个`catch`块

我将上面代码中一些重要的步骤之后都写上了注释。

我们之前一直纠结的如何在运行时比较类型信息，`landingpad`帮我们做好了，其本质还是根据C++的RTTI结构。

在判断出了类型信息之后，我们会根据TypeID进入不同的标签：

- 如果是`MyError`类型或其子类，进入`%14`标签
- 如果是`AnotherError`类型或其子类，进入`%22`标签
- 如果都不是，进入`%26`标签

这些标签内错误处理的框架都很类似，我们来看`%14`标签：

```
14:
    %15 = load i8*, i8** %1, align 8
    %16 = call i8* @__cxa_begin_catch(i8* %15) #3
    %17 = bitcast i8* %16 to %struct.MyError*
    call void @__cxa_end_catch()
    br label %18
```

都是以`@__cxa_begin_catch`开始，以`@__cxa_end_catch`结束。简单来说，这里就是：

1. 从异常结构体中获得抛出的异常对象本身（异常结构体可能还包含其他信息）
2. 进行异常处理
3. 结束异常处理，回收、释放相应的结构体

在哪可以看到我的文章

我的LLVM IR入门指南系列可以在我的个人博客、GitHub：Evian-Zhang/llvm-ir-tutorial、知乎、CSDN中查看，本教程中涉及的大部分代码也都在同一GitHub仓库中。

本人水平有限，写此文章仅希望与大家分享学习经验，文章中必有缺漏、错误之处，望方家不吝斧正，与大家共同学习，共同进步，谢谢大家！