LLVM Essentials 中文版

Suyog Sarda, Mayur Pandey 著

Lifeng Pan 译

第 1 章 LLVM 概述

2000 年，LLVM 编译器基础设施项目发起于伊利诺斯大学。最初作为一个研究项目，它为任意静态的和动态的编程语言提供现代的基于 SSA 的编译技术。如今它已经成长为一个项目集合，包含多个子项目，提供一整套接口设计优良的程序库。

LLVM 用 C++写成，核心程序库是重点。这些程序库为我们提供 opt 工具、目标无关的优化程序、目标相关的代码生成器。代码生成器支持多种目标架构。本书着重关注以上三类程序，尽管还有很多其它利用核心程序库的工具。LLVM 中间表示（LLVM IR）是它们的构建基础，它几乎能够映射所有的高级语言。因此简单地说，想要利用 LLVM 的优化算法和代码生成技术编译一种编程语言，我们需要做的仅仅是设计一个前端（frontend）编译器，将高级语言翻译成 LLVM IR。人们已经为多种高级语言开发了这样的前端，如 C, C++, Go, Python，等。在这个章节我们将讨论以下内容：

- 模块化设计和程序库集合
- LLVM IR 概述
- LLVM 工具及其命令行使用方式

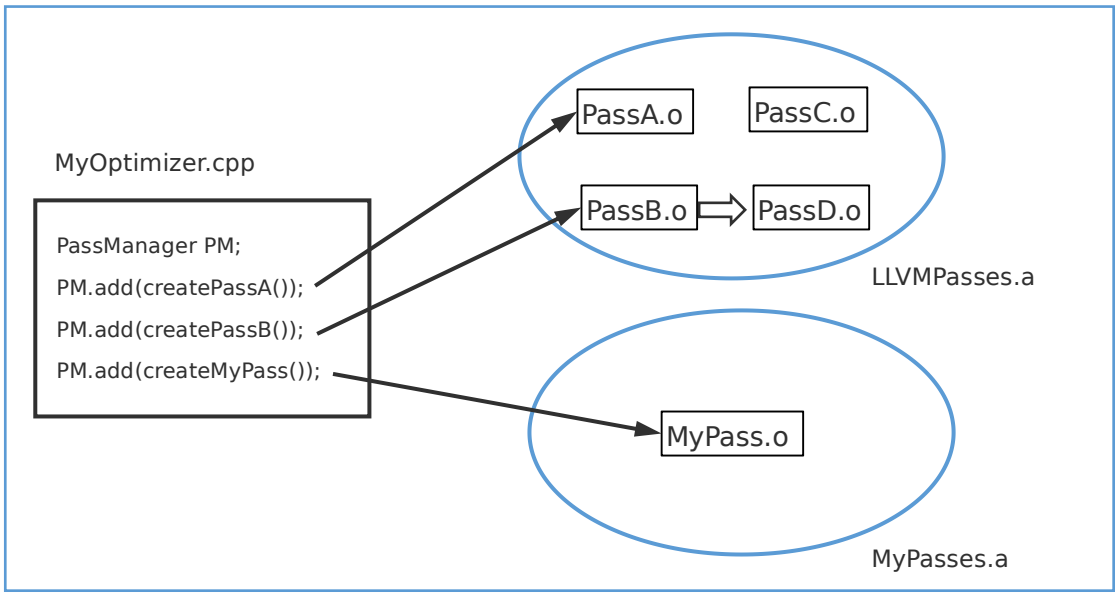
模块化设计和程序库集合

LLVM 最重要的一件事情是它设计成一个程序库集合。让我们以 LLVM 优化工具 opt 来说明其重要性。这个工具包含许多不同的优化 Pass。每一个优化 Pass 是一个 C++类，派生于 LLVM Pass 基类。它们都可以编译成.o 文件，继而存档到（同）一个.a 库文件。这个库文件包含所有优化 Pass，为 opt 工具所用。所有优化 Pass 都是松散耦合的，具体地说，它们显式地标明对其它 Pass 的依赖。

当优化器运行时，LLVM PassManager 利用显式标明的依赖信息，以最佳的方式运行

这些优化 Pass。这种模块化的设计方式允许程序员决定优化 Pass 的执行次序，根据需要决定执行哪些优化 Pass。只有需要的优化 Pass 才会被链接到最终的应用程序中，而不是所有的优化 Pass。

下图说明了每个优化 Pass 是如何被链接到一个属于特定库文件的特定目标文件（object file）中的。在图中，PassA 引用 LLVMPasses.a 中的 PassA.o，而定制的 MyPass 引用 MyPasses.a 中的 MyPass.o。



代码生成器也是如此利用这种模块化设计的。它划分成几个单独的 Pass，即指令选择，寄存器分配，指令调度，代码布局优化，汇编输出。

在上述各个 Pass 中，有些方法适用于几乎所有的目标架构，例如一种为虚拟寄存器分配物理寄存器的算法，即使不同目标的寄存器集是不同的。因此，编译器开发者可以修改上述每个 Pass，创建定制的适用特定目标的 Pass。借助 tablegen 工具不难实现这种需求。它用.td 文件描述特定的目标。我们将在后续章节讨论它是怎么工作的。

由此带来的另一个好处是，易于将 bug 定位到优化器的一个特定的 Pass。Bugpoint 工具能够自动地缩小测试案例并精确地找到导致 bug 的那个 Pass。

LLVM IR 概述

LLVM 中间表示，简称 LLVM IR（Intermediate Representation），是 LLVM 项目的核心。通常来说，所有编译器都生成中间表示，并在此之上进行大多数优化。对于一个能处理多种源语言并以不同架构为目标的编译器来说，选择一种 IR 是非常关键的。它既不能是非常高级的，即非常接近于源语言，也不能是非常低级的，即非常接近于目标机器指令。LLVM IR 被设计为一种通用 IR，它必须足够低级使得高级语言的思想可以被清楚地映射表达。理想的 LLVM IR 应该是目标机器无关的，但是实际上不是，因为有些编程语言自身就是目标机器相关的。例如，在 Linux 系统上使用标准 C 头文件，它们自身是目标相关的，它们可能为一个变量实体指定特殊的类型，以使它与特定目标架构的系统调用函数相匹配。

大多数 LLVM 工具都处理 LLVM IR。不同语言的前端编译器把高级源语言翻译成 LLVM IR。LLVM 优化器运行在此 IR 之上，优化代码以提高性能。代码生成器将此 IR 转换成特定机器代码。LLVM IR 有三种等价的形式：

- 驻留内存的编译器中间表示
- 贮存硬盘的位码（bitcode）表示
- 人类可读的文本形式（LLVM 汇编）

下面我们通过一个例子看看 LLVM IR 的样子。我们给出小段 C 代码，用 Clang 程序将它转化为 LLVM IR，然后试着将它映射回源语言，以此理解 LLVM IR 的细节。

```
$ cat add.c
int globvar = 12;
int add(int a) {
    return globvar + a;
}
```

用 Clang 程序将它转化为 LLVM IR，命令行参数如下：

```
$ clang -emit-llvm -c -S add.c
$ cat add.ll
; ModuleID = 'add.c'
```

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

@globvar = global i32 12, align 4

; Function Attrs: nounwind uwtable
define i32 @add(i32 %a) #0 {
  %1 = alloca i32, align 4
  store i32 %a, i32* %1, align 4
  %2 = load i32, i32* @globvar, align 4
  %3 = load i32, i32* %1, align 4
  %4 = add nsw i32 %2, %3
  ret i32 %4
}

attributes #0 = { nounwind uwtable "disable-tail-calls"="false" "less-precise-fpmad"="false" "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf" "no-infs-fp-math"="false" "no-nans-fp-math"="false" "stack-protector-buffer-size"="8" "target-cpu"="x86-64" "target-features"="+fxsr,+mmx,+sse,+sse2" "unsafe-fp-math"="false" "use-soft-float"="false" }

!llvm.ident = !{!0}

!0 = !{!"clang version 3.8.1-15 (tags/RELEASE_381/final)"}
```

下面我们来看生成的 IR 在做什么。你能看到第一行给出 ModuleID，它为 add.c 文件定义 LLVM 模块。一个 LLVM 模块是顶层数据结构，包含输入的 LLVM IR 文件的全部内容。它包含函数，全局变量，外部函数原型，符号表项。

接下来两行表示目标 data layout（数据布局）和目标 triple（三元）。目标 triple 告诉我们，目标是 x86_64 处理器，运行在 Linux 系统上。字符串 datalayout 告诉我们使用什么机器字节顺序（‘e’表示小端字节顺序），什么名字粉碎（name mangling）方式（m：e 表示 elf 类型）。每项设置以’-’分隔。后面的设置指定类型信息和类型的位长度。例如，i64:64 表示 64 位整数，长度是 64 位。

然后，我们看到一个全局变量 globvar。在 LLVM IR 中，全局变量都以’@’开头，局部变量都以’%’开头。变量名使用这些符号为前缀的理由有两个。其一，编译器不必担忧一个

变量名与保留字相冲突，其二，编译器不必担忧一个临时变量名和符号表相冲突。以静态单赋值（SSA: static single assignment）形式表达 IR 时，上面的第 2 条特性是很有用的。静态单赋值形式要求每个变量只赋值一次，并且变量的使用必须在它的定义之后。在转换普通程序为 SSA 形式时，我们为每个重定义变量都生成一个新的临时变量，并将对它的引用替换成这个新的临时变量，而先前的定义到此生命期终结。

LLVM 把全局变量当作指针，因此它用 load 指令加载数据显式解引用全局变量。类似地，它用显式 store 指令为全局变量存储数据。

局部变量有两种类别：

- 寄存器分配的局部变量：它们是临时存储单元和分配的虚拟寄存器。虚拟寄存器在代码生成阶段被分配为物理寄存器，这会在后续章节讲解。可以用新的符号生成这类局部变量，例如
  %1 = some value
- 堆栈分配的局部变量：它们是用 alloca 指令在当前执行函数的堆栈帧上分配的变量。alloca 指令产生指针数据类型，必须显式使用 load 指令对它读取数据，使用 store 指令对它写入数据。
  %2 = alloca i32

接下来我们来看 LLVM IR 是怎么表示 add 函数的。define i32 @add(i32 %a)，这和 C 语言声明函数非常类似。它指定函数返回整数类型 i32，有一个整数参数。函数名字也以’@’开头，说明它是全局可见的。

函数内部是函数的运算处理。这里值得特别注意的事情是 LLVM 使用三地址指令，这种数据处理指令有两个源操作数和一个单独保存结果的目标操作数（%4 = add i32 %2, %3）。这段代码是 SSA 形式，即每个值只有单一赋值定义它。这个特性在很多优化 Pass 中是有用的。

属性(attribute)字符串跟在所生成的 IR 之后，它们指定函数属性，这和 C++属性非常类似。这些属性是属于之前定义的函数的。LLVM IR 为每个定义的函数指定一套属性。

属性后面的代码是 ident 指示符，用于识别模块和编译器版本。

LLVM 工具及其命令行使用方式

至此，我们已经理解了 LLVM IR（人类可读形式）是什么，如何用它表示一种高级语言。接下来，我们将学习一些 LLVM 提供的工具，用它们处理 LLVM IR，转换到其它格式，又转换回初始格式。我们通过例子逐个认识这些工具。

- llvm-as：这是 LLVM 汇编器，它把汇编格式的 LLVM IR（人类可读形式）转换成位码（bitcode）格式。用之前的 add.ll 为例子，把它转换为位码。了解更多 LLVM IR 位码文件的格式，参考 http://llvm.org/docs/BitCodeFormat.html

  $ llvm-as add.ll -o add.bc

  可以用 hexdump 工具查看这个位码文件的内容。

  $ hexdump -c add.bc
- llvm-dis：这是 LLVM 反汇编器。它以 llvm 位码为输入，输出 llvm 汇编。

  $ llvm-dis add.bc -o add.ll

  如果你检查 add.ll 并比较它和之前的版本，你会发现它们是相同的。
- llvm-link：它链接两个或者多个 llvm 位码文件，生成一个新的 llvm 位码文件。举个例子。写一个 main.c 文件，它调用 add.c 文件中的函数。

  ```
  $ cat main.c
  #include <stdio.h>

  extern int add(int);

  int main() {
      int a = add(2);
      printf("%d\n", a);
      return 0;
  }
  ```

用下面的命令把这段 C 源代码转换为 llvm 位码格式。

$ clang -emit-llvm -c main.c

然后链接 main.bc 和 add.bc 以生成 output.bc。

$ llvm-link main.bc add.bc -o output.bc

- lli：lli 直接用即时（just-in-time）编译器执行 llvm 位码程序，如果有可用的当前架构的解释器（interpreter），就用解释器执行。lli 不像虚拟机，不能执行不同架构的 IR，只能为主机（host）架构解释执行。把之前生成的 llvm 位码文件输入给 lli，它会在标准输出上显示输出结果。

  $ lli output.bc
  14

- llc：llc 是静态编译器。它编译输入的 LLVM IR（汇编形式或位码形式）为指定架构的汇编语言。下面的例子把由 llvm-link 生成的 output.bc 文件编译成汇编文件 output.s。

  $ llc output.bc -o output.s

  我们看一看 output.s 汇编的内容，特别是所生成代码中的两个函数，和本地汇编器所能生成的非常类似。

```
    .text
    .file "output.bc"
    .globl    main
    .p2align      4, 0x90
    .type     main,@function
main:                                   # @main
    .cfi_startproc
# BB#0:
    pushq   %rbp
.Ltmp0:
    .cfi_def_cfa_offset 16
.Ltmp1:
    .cfi_offset %rbp, -16
    movq    %rsp, %rbp
.Ltmp2:
    .cfi_def_cfa_register %rbp
    subq    $16, %rsp
```

```
	movl	$0, -8(%rbp)
	movl	$2, %edi
	callq	add
	movl	%eax, %ecx
	movl	%ecx, -4(%rbp)
	movl	$.L.str, %edi
	xorl	%eax, %eax
	movl	%ecx, %esi
	callq	printf
	xorl	%eax, %eax
	addq	$16, %rsp
	popq	%rbp
	retq
.Lfunc_end0:
	.size	main, .Lfunc_end0-main
	.cfi_endproc

	.globl	add
	.p2align	4, 0x90
	.type	add,@function
add:                                    # @add
	.cfi_startproc
# BB#0:
	pushq	%rbp
.Ltmp3:
	.cfi_def_cfa_offset 16
.Ltmp4:
	.cfi_offset %rbp, -16
	movq	%rsp, %rbp
.Ltmp5:
	.cfi_def_cfa_register %rbp
	movl	%edi, -4(%rbp)
	addl	globvar(%rip), %edi
	movl	%edi, %eax
	popq	%rbp
	retq
.Lfunc_end1:
	.size	add, .Lfunc_end1-add
	.cfi_endproc

	.type	.L.str,@object          # @.str
	.section	.rodata.str1.1,"aMS",@progbits,1
.L.str:
	.asciz	"%d\n"
```

```
    .size   .L.str, 4

    .type   globvar,@object          # @globvar
    .data
    .globl  globvar
    .p2align    2
globvar:
    .long   12                       # 0xc
    .size   globvar, 4

    .ident  "clang version 3.8.1-15 (tags/RELEASE_381/final)"
    .ident  "clang version 3.8.1-15 (tags/RELEASE_381/final)"
    .section".note.GNU-stack","",@progbits
```

- opt：这是模块化的 LLVM 分析器和优化器。它对输入文件执行命令行指定的优化或者分析。执行优化还是分析取决于命令行选项。

  opt [option] [input file name]

  当给定了-analyze 选项时，它对输入执行各种分析。有一系列分析选项，可以通过命令行指定。此外，你也可以写一个自己的分析 Pass，以库文件的形式给出。以下列出一些有用的分析 Pass 的命令行参数：

  basicaa: basic alias analysis（基础别名分析）

  da: dependence analysis（依赖分析）

  instcount: count the various instruction types（计数各种指令类型）

  loops: information about loops（循环信息）

  scalar evolution: analysis of scalar evolution（标量进化分析）

  当未给定-analyze 选项时，opt 工具做真正的优化工作，根据给出的命令行选项尝试着优化代码。与前面类似，你可以使用已有的优化 Pass，或者写一个自己的优化 Pass。以下列出一些有用的优化 Pass 的命令行参数：

  constprop: simple constant propagation（简单常数传播）

  dce: dead code elimination（死亡代码消除）

globalopt: global variable optimization（全局变量优化）

inline: function inlining（函数内联）

instcombine: combining redundant instructions（合并冗余指令）

licm: loop invariant code motion（循环不变量代码移动）

tailcallelim: tail call elimination（尾调用消除）

注解

以上介绍的所有的工具都是为编译器开发者提供的。普通用户可以直接用 Clang 编译 C 源代码，而不需要先把它转换成中间表示（IR）形式。

Tip

下载示例代码

你可以通过你的账号从网址 http://www.packtpub.com 下载示例代码，为你所购买的所有 Packt Publishing 的书籍。如果你在别处买了这本书，你可以访问 http://www.packtpub.com/support，注册之后，所有的文件将通过 e-mail 直接发送给你。

总结

本章介绍了 LLVM 的模块化设计：opt 工具是如何利用它的，LLVM 核心程序库是如何交叉关联的。我们了解了 LLVM 中间表示，如何把各种语言元素（变量，函数等）映射为 LLVM 中间表示。我们还介绍了一些重要的 LLVM 工具，如何用它们把 LLVM IR 从一种形式转换为另一种形式。

下一章我们将学习如何为语言编写一个前端（frontend）编译器，它利用 LLVM 内在机制输出 LLVM IR。

第 2 章 构建 LLVM IR

人们利用高级编程语言与计算机交互非常便利。如今大多数流行的高级语言有若干基本的元素，例如变量，循环，if-else 条件语句，块（block），函数等。变量保存数据类型的数值。块表达变量域（scope）的概念。if-else 条件语句选择代码路径分支。函数让一段代码可以重复调用。高级语言可能在多个方面表现不同，例如类型检查，类型转换，变量声明，复杂数据类型等。然而，几乎所有的语言都有前面提到的基本元素。

一种语言可能有自己的解析器（parser），它把语句变成记号，提取语义信息，例如标识符的数据类型，函数的名称、声明、定义和调用，循环条件等。这些语义信息存储于数据结构中，便于获取语句序列。抽象语法树（AST: abstract syntax tree）是一种流行的树表达方式，用于表达源代码。AST 可用于进一步的转换和分析。

语言解析器可用不同方法和工具写成，例如 lex, yacc 等，甚至可以手写。写出一个高效的解析器本身是一种艺术。但是这不属于本章的范围。我们想要讨论的是，如何利用 LLVM 程序库解析高级语言，将之转换为 LLVM IR。

本章将介绍如何构建基本的 LLVM IR 程序，包括以下内容：

- 创建 LLVM 模块
- 为模块生成函数
- 为函数添加基本块
- 生成全局变量
- 生成返回语句
- 生成函数参数
- 在基本块中生成简单的算术语句
- 生成 if-else 条件语句

- 生成循环语句

创建 LLVM 模块

在前面章节中，我们已经初步认识 LLVM IR。在 LLVM 中，一个模块代表一份可以被集中处理的代码。LLVM 模块类是顶层容器数据结构，容纳所有其它的 LLVM IR 对象。一个 LLVM 模块包含全局变量，函数，数据布局（layout），主机三元（host triple）等。让我们创建一个简单的 LLVM 模块。

LLVM 提供模块类构造函数 Module()创建模块。它的第 1 个参数是模块名字。第 2 个参数是 LLVMContext。下面的例子演示了如何获取 LLVMContext，并创建一个模块：

```
static LLVMContext &Context = getGlobalContext();
static Module *ModuleOb = new Module("my compiler", Context);
```

为了能够正确引用的这两个函数，需要包含它们的头文件：

```
#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/Module.h"
using namespace llvm;

static LLVMContext Context;
static Module *ModuleOb = new Module("my compiler", Context);

int main(int argc, char *argv[]) {
    ModuleOb->dump();
    return 0;
}
```

保存以上代码为一个文件，命名 toy.cpp，然后编译它：

```
$ clang++ -O3 toy.cpp `llvm-config --cxxflags --ldflags --system-libs --libs core` -o toy
```

执行时输出如下：

```
$ ./toy
; ModuleID = 'my compiler'
```

为模块生成函数

我们已经创建了一个模块，下一步是生成函数。LLVM 用 IRBuilder 类生成 LLVM IR，用模块对象的 dump() 函数打印它。LLVM 用 llvm:Function 类创建函数，用 llvm::FunctionType 定义函数返回类型。假如我们想生成一个返回整数类型的函数 foo()。

```
Function *createFunc(IRBuilder<> &Builder, std::string Name) {
    FunctionType *funcType = llvm::FunctionType::get(Builder.getInt32Ty(), false);
    Function *fooFunc = llvm::Function::Create(funcType, llvm::Function::ExternalLinkage, Name, ModuleOb);
    return fooFunc;
}
```

函数生成以后，可以为它调用 verifyFunction()。这个函数对生成的 LLVM 代码执行多种一致性检查，确定我们的编译器是否正确工作。

```
int main(int argc, char *argv[]) {
    static IRBuilder<> Builder(Context);
    Function *fooFunc = createFunc(Builder, "foo");
    verifyFunction(*fooFunc);
    ModuleOb->dump();
    return 0;
}
```

为了让以上代码正确工作，需要包含头文件 IR/IRBuilder.h, IR/Verifier.h。整体代码如下：

```
#include "llvm/IR/IRBuilder.h"
#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/Module.h"
#include "llvm/IR/Verifier.h"
#include <vector>
using namespace llvm;

static LLVMContext Context;
static Module *ModuleOb = new Module("my compiler", Context);

Function *createFunc(IRBuilder<> &Builder, std::string Name) {
    FunctionType *funcType = llvm::FunctionType::get(Builder.getInt32Ty(), false);
    Function *fooFunc = llvm::Function::Create(funcType,
```

```
llvm::Function::ExternalLinkage, Name, ModuleOb);
    return fooFunc;
}

int main(int argc, char *argv[]) {
    static IRBuilder<> Builder(Context);
    Function *fooFunc = createFunc(Builder, "foo");
    verifyFunction(*fooFunc);
    ModuleOb->dump();
    return 0;
}
```

以同样的参数编译新的 toy.cpp：

```
$ clang++ -O3 toy.cpp `llvm-config --cxxflags --ldflags --system-libs --libs core` -o toy
```

执行时输出如下：

```
$ ./toy
; ModuleID = 'my compiler'

declare i32 @foo()
```

为函数添加基本块

一个函数由若干基本块组成。一个基本块有一个入口（entry point）。一个基本块由若干 IR 指令 (instruction) 组成，最后的指令是一条终结指令。它有单个出口 (exit point)。

LLVM 用 BasicBlock 类创建和处理基本块。

```
BasicBlock *createBB(Function *fooFunc, std::string Name) {
    return BasicBlock::Create(Context, Name, fooFunc);
}
```

一个基本块可能以它的入口为标签（label），指示下一条指令的插入位置。我们可以用 IRBuilder 对象持有一个当前的基本块，指示新指令的插入位置。整体代码如下：

```
#include "llvm/IR/IRBuilder.h"
#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/Module.h"
#include "llvm/IR/Verifier.h"
#include <vector>
using namespace llvm;
```

```
static LLVMContext Context;
static Module *ModuleOb = new Module("my compiler", Context);

Function *createFunc(IRBuilder<> &Builder, std::string Name) {
    FunctionType *funcType = llvm::FunctionType::get(Builder.getInt32Ty(), false);
    Function *fooFunc = llvm::Function::Create(funcType, llvm::Function::ExternalLinkage, Name, ModuleOb);
    return fooFunc;
}

BasicBlock *createBB(Function *fooFunc, std::string Name) {
    return BasicBlock::Create(Context, Name, fooFunc);
}

int main(int argc, char *argv[]) {
    static IRBuilder<> Builder(Context);
    Function *fooFunc = createFunc(Builder, "foo");
    BasicBlock *entry = createBB(fooFunc, "entry");
    Builder.SetInsertPoint(entry);
    verifyFunction(*fooFunc);
    ModuleOb->dump();
    return 0;
}
```

编译新的 toy.cpp：

```
$ clang++ -O3 toy.cpp `llvm-config --cxxflags --ldflags --system-libs --libs core` -o toy
```

执行时输出如下：

```
$ ./toy
; ModuleID = 'my compiler'

define i32 @foo() {
entry:
}
```

生成全局变量

全局变量对于一个模块中的所有函数都可见。LLVM 用 GlobalVariable 类创建全局变量，并设置它的属性，例如链接类型，对齐方式等。Module 类有方法 getOrInsertGlobal()

用于创建全局变量。它有两个参数，变量名字和变量数据类型。

全局变量是模块的一部分，因此我们在创建模块后创建全局变量。在 toy.cpp 文件中创建模块之后插入以下代码：

```
GlobalVariable *createGlob(IRBuilder<> &Builder, std::string Name) {
    ModuleOb->getOrInsertGlobal(Name, Builder.getInt32Ty());
    GlobalVariable *gVar = ModuleOb->getNamedGlobal(Name);
    gVar->setLinkage(GlobalValue::CommonLinkage);
    gVar->setAlignment(4);
    return gVar;
}
```

链接类型用于决定相同对象的多次声明是指向同一对象，还是指向各自的对象。LLVM 参考手册列出如下的链接类型：

```
ExternalLinkage: External visible function.
AvailableExternalLinkage: Available for inspection, not emission.
LinkOnceAnyLinkage: Keep one copy of function when linking (inline).
LinkOnceODRLinkage: Same, but only replaced by something equivalent.
WeakAnyLinkage: Keep one copy of named function when linking (weak).
WeakODRLinkage: Same, but only replaced by something equivalent.
AppendingLinkage: Special purpose, only applies to global arrays.
InternalLinkage: Rename collisions when linking (static functions).
PrivateLinkage: Like internal, but omit from symbol table.
ExternalWeakLinkage: ExternalWeak linkage description.
CommonLinkage: Tentative definitions.
```

对齐方式给出地址对齐信息。对齐方式必须是 2 的幂。如果没有显式指定，它由目标机器设定。最大对齐方式是 1<<29。

增加全局变量创建后，整体代码如下：

```
#include "llvm/IR/IRBuilder.h"
#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/Module.h"
#include "llvm/IR/Verifier.h"
#include <vector>
using namespace llvm;

static LLVMContext Context;
static Module *ModuleOb = new Module("my compiler", Context);
```

```
Function *createFunc(IRBuilder<> &Builder, std::string Name) {
    FunctionType *funcType = llvm::FunctionType::get(Builder.getInt32Ty(),
false);
    Function *fooFunc = llvm::Function::Create(funcType,
llvm::Function::ExternalLinkage, Name, ModuleOb);
    return fooFunc;
}

BasicBlock *createBB(Function *fooFunc, std::string Name) {
    return BasicBlock::Create(Context, Name, fooFunc);
}

GlobalVariable *createGlob(IRBuilder<> &Builder, std::string Name) {
    ModuleOb->getOrInsertGlobal(Name, Builder.getInt32Ty());
    GlobalVariable *gVar = ModuleOb->getNamedGlobal(Name);
    gVar->setLinkage(GlobalValue::CommonLinkage);
    gVar->setAlignment(4);
    return gVar;
}

int main(int argc, char *argv[]) {
    static IRBuilder<> Builder(Context);
    GlobalVariable *gVar = createGlob(Builder, "x");
    Function *fooFunc = createFunc(Builder, "foo");
    BasicBlock *entry = createBB(fooFunc, "entry");
    Builder.SetInsertPoint(entry);
    verifyFunction(*fooFunc);
    ModuleOb->dump();
    return 0;
}
```

编译新的 toy.cpp：

```
$ clang++ -O3 toy.cpp `llvm-config --cxxflags --ldflags --system-libs --libs core`
-o toy
```

执行时输出如下：

```
$ ./toy
; ModuleID = 'my compiler'

@x = common global i32, align 4

define i32 @foo() {
entry:
}
```

生成返回语句

一个函数可以返回一个数值，或者返回空类型（void）。在我们的例子中，我们已经定义了一个返回整数的函数。我们假设这个函数返回 0。第 1 步是获得一个 0 值，这可以用 Constant 类。第 2 步才是创建返回语句。

```
Builder.CreateRet(Builder.getInt32(0));
```

整体代码如下：

```
#include "llvm/IR/IRBuilder.h"
#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/Module.h"
#include "llvm/IR/Verifier.h"
#include <vector>
using namespace llvm;

static LLVMContext Context;
static Module *ModuleOb = new Module("my compiler", Context);

Function *createFunc(IRBuilder<> &Builder, std::string Name) {
    FunctionType *funcType = llvm::FunctionType::get(Builder.getInt32Ty(), false);
    Function *fooFunc = llvm::Function::Create(funcType, llvm::Function::ExternalLinkage, Name, ModuleOb);
    return fooFunc;
}

BasicBlock *createBB(Function *fooFunc, std::string Name) {
    return BasicBlock::Create(Context, Name, fooFunc);
}

GlobalVariable *createGlob(IRBuilder<> &Builder, std::string Name) {
    ModuleOb->getOrInsertGlobal(Name, Builder.getInt32Ty());
    GlobalVariable *gVar = ModuleOb->getNamedGlobal(Name);
    gVar->setLinkage(GlobalValue::CommonLinkage);
    gVar->setAlignment(4);
    return gVar;
}

int main(int argc, char *argv[]) {
    static IRBuilder<> Builder(Context);
    GlobalVariable *gVar = createGlob(Builder, "x");
```

```
    Function *fooFunc = createFunc(Builder, "foo");
    BasicBlock *entry = createBB(fooFunc, "entry");
    Builder.SetInsertPoint(entry);
    Builder.CreateRet(Builder.getInt32(0));

    verifyFunction(*fooFunc);
    ModuleOb->dump();
    return 0;
}
```

编译新的 toy.cpp：

```
$ clang++ -O3 toy.cpp `llvm-config --cxxflags --ldflags --system-libs --libs core` -o toy
```

执行时输出如下：

```
$ ./toy
; ModuleID = 'my compiler'

@x = common global i32, align 4

define i32 @foo() {
entry:
  ret i32 0
}
```

生成函数参数

函数接收参数，参数有数据类型。简单起见，假设我们的函数的所有参数都是 i32 类型（32 位整数）。举例来说，有两个参数，a 和 b，传递给函数。我们把它们存在一个 vector 中：

```
static std::vector <std::string> FunArgs;
FunArgs.push_back("a");
FunArgs.push_back("b");
```

下一步就是指定函数有两个参数。这可以通过给 functionType 传递整数参数做到。

```
Function *createFunc(IRBuilder<> &Builder, std::string Name) {
    std::vector<Type *> Integers(FunArgs.size(), Type::getInt32Ty(Context));
    FunctionType *funcType = llvm::FunctionType::get(Builder.getInt32Ty(), Integers, false);
    Function *fooFunc = llvm::Function::Create(funcType, llvm::Function::ExternalLinkage, Name, ModuleOb);
```

```
    return fooFunc;
}
```

最后一步是设置函数参数的名字。这可以通过在循环中迭代访问 Function 参数做到，

示例如下：

```
void setFuncArgs(Function *fooFunc, std::vector<std::string> FunArgs) {
    unsigned Idx = 0;
    Function::arg_iterator AI, AE;
    for (AI = fooFunc->arg_begin(), AE = fooFunc->arg_end(); AI != AE; ++AI,
++Idx) {
        AI->setName(FunArgs[Idx]);
    }
}
```

整体代码如下：

```
#include "llvm/IR/IRBuilder.h"
#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/Module.h"
#include "llvm/IR/Verifier.h"
#include <vector>
using namespace llvm;

static LLVMContext Context;
static Module *ModuleOb = new Module("my compiler", Context);
static std::vector <std::string> FunArgs;

Function *createFunc(IRBuilder<> &Builder, std::string Name) {
    std::vector<Type *> Integers(FunArgs.size(), Type::getInt32Ty(Context));
    FunctionType *funcType = llvm::FunctionType::get(Builder.getInt32Ty(),
Integers, false);
    Function *fooFunc = llvm::Function::Create(funcType,
llvm::Function::ExternalLinkage, Name, ModuleOb);
    return fooFunc;
}

void setFuncArgs(Function *fooFunc, std::vector<std::string> FunArgs) {
    unsigned Idx = 0;
    Function::arg_iterator AI, AE;
    for (AI = fooFunc->arg_begin(), AE = fooFunc->arg_end(); AI != AE; ++AI,
++Idx) {
        AI->setName(FunArgs[Idx]);
    }
}
```

```
BasicBlock *createBB(Function *fooFunc, std::string Name) {
    return BasicBlock::Create(Context, Name, fooFunc);
}

GlobalVariable *createGlob(IRBuilder<> &Builder, std::string Name) {
    ModuleOb->getOrInsertGlobal(Name, Builder.getInt32Ty());
    GlobalVariable *gVar = ModuleOb->getNamedGlobal(Name);
    gVar->setLinkage(GlobalValue::CommonLinkage);
    gVar->setAlignment(4);
    return gVar;
}

int main(int argc, char *argv[]) {
    FunArgs.push_back("a");
    FunArgs.push_back("b");
    static IRBuilder<> Builder(Context);
    GlobalVariable *gVar = createGlob(Builder, "x");
    Function *fooFunc = createFunc(Builder, "foo");
    setFuncArgs(fooFunc, FunArgs);
    BasicBlock *entry = createBB(fooFunc, "entry");
    Builder.SetInsertPoint(entry);
    Builder.CreateRet(Builder.getInt32(0));

    verifyFunction(*fooFunc);
    ModuleOb->dump();
    return 0;
}
```

编译新的 toy.cpp：

```
$ clang++ -O3 toy.cpp `llvm-config --cxxflags --ldflags --system-libs --libs core`
-o toy
```

执行时输出如下：

```
$ ./toy
; ModuleID = 'my compiler'

@x = common global i32, align 4

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

在基本块中生成简单的算术语句

基本块由一列指令组成。举例来说，一条指令可以是一条简单语句，执行简单的算术运算。我们将介绍如何用 LLVM API 生成算术指令。

举例来说，假设我们想让第 1 个参数乘以整数值 16，我们先得创建一个整数常量 16，如下：

```
Value *constant = Builder.getInt32(16);
```

我们从函数参数列表中得到参数 a：

```
Value *Arg1 = fooFunc->arg_begin();
```

LLVM 提供丰富的 API 用于创建二元运算（binary operation）。想更多地了解这些 API，你可以浏览 include/llvm/IR/IRBuilder.h 文件。

```
Value *createArith(IRBuilder<> &Builder, Value *L, Value *R) {
    return Builder.CreateMul(L, R, "multmp");
}
```

注解

为了演示的目的，上面的函数只返回乘法。它可以变得更灵活以返回任意二元运算。我们把这留给读者尝试。你能在 include/llvm/IR/IRBuilder.h 文件中看到更多的二元运算。

到此，整体代码如下：

```
#include "llvm/IR/IRBuilder.h"
#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/Module.h"
#include "llvm/IR/Verifier.h"
#include <vector>
using namespace llvm;

static LLVMContext Context;
static Module *ModuleOb = new Module("my compiler", Context);
static std::vector <std::string> FunArgs;

Function *createFunc(IRBuilder<> &Builder, std::string Name) {
    std::vector<Type *> Integers(FunArgs.size(), Type::getInt32Ty(Context));
    FunctionType *funcType = llvm::FunctionType::get(Builder.getInt32Ty(), Integers, false);
```

```
    Function *fooFunc = llvm::Function::Create(funcType,
llvm::Function::ExternalLinkage, Name, ModuleOb);
    return fooFunc;
}

void setFuncArgs(Function *fooFunc, std::vector<std::string> FunArgs) {
    unsigned Idx = 0;
    Function::arg_iterator AI, AE;
    for (AI = fooFunc->arg_begin(), AE = fooFunc->arg_end(); AI != AE; ++AI,
++Idx) {
        AI->setName(FunArgs[Idx]);
    }
}

BasicBlock *createBB(Function *fooFunc, std::string Name) {
    return BasicBlock::Create(Context, Name, fooFunc);
}

GlobalVariable *createGlob(IRBuilder<> &Builder, std::string Name) {
    ModuleOb->getOrInsertGlobal(Name, Builder.getInt32Ty());
    GlobalVariable *gVar = ModuleOb->getNamedGlobal(Name);
    gVar->setLinkage(GlobalValue::CommonLinkage);
    gVar->setAlignment(4);
    return gVar;
}

Value *createArith(IRBuilder<> &Builder, Value *L, Value *R) {
    return Builder.CreateMul(L, R, "multmp");
}

int main(int argc, char *argv[]) {
    FunArgs.push_back("a");
    FunArgs.push_back("b");
    static IRBuilder<> Builder(Context);
    GlobalVariable *gVar = createGlob(Builder, "x");
    Function *fooFunc = createFunc(Builder, "foo");
    setFuncArgs(fooFunc, FunArgs);
    BasicBlock *entry = createBB(fooFunc, "entry");
    Builder.SetInsertPoint(entry);
    Value *Arg1 = dyn_cast<Value>(fooFunc->arg_begin());
    Value *constant = Builder.getInt32(16);
    Value *val = createArith(Builder, Arg1, constant);
    Builder.CreateRet(val);
```

```
    verifyFunction(*fooFunc);
    ModuleOb->dump();
    return 0;
}
```

编译新的 toy.cpp：

```
$ clang++ -O3 toy.cpp `llvm-config --cxxflags --ldflags --system-libs --libs core` -o toy
```

执行时输出如下：

```
$ ./toy
; ModuleID = 'my compiler'

@x = common global i32, align 4

define i32 @foo(i32 %a, i32 %b) {
entry:
  %multmp = mul i32 %a, 16
  ret i32 %multmp
}
```

你注意到返回值了吗？我们用乘积代替常数 0 作为返回值。

生成 if-else 条件语句

if-else 语句有一个条件表达式和两条代码路径，根据条件表达式的值为真（true）或者假（false）执行其中一条路径。条件表达式通常是一个比较语句。我们将在基本块的开始处生成一个条件语句。举例来说，我们的条件是 a < 100。

```
Value *val2 = Builder.getInt32(100);
Value *Compare = Builder.CreateICmpULT(val, val2, "cmptmp");
```

编译执行后，得到如下输出：

```
; ModuleID = ‘my compiler’

@x = common global i32, align 4

define i32 @foo(i32 %a, i32 %b) {
entry:
  %multmp = mul i32 %a, 16
  %cmptmp = icmp ult i32 %multmp, 100
  ret i32 %multmp
```

}

下一步是定义 then 和 else 基本块，而条件表达式的结果是布尔（boolean）值。如果它是真，then 基本块的语句序列将被执行，否则 else 基本块的语句序列将被执行。这里，出现一个重要的概念，**PHI** 指令。Phi 指令接收多个来自不同基本块的值，根据代码执行的路径选择相应基本块的值。当然路径选择取决于条件表达式。

两个单独的基本块”ThenBB”和”ElseBB”将会被创建。假设 then 和 else 基本块的表达式分别是‘a + 1’和’a + 2‘。

第 3 个基本块是汇合基本块，它表示 then 和 else 基本块在此汇合，从而执行它所包含的指令。以上 3 个基本块都将被添加到函数 foo()。

为了重复使用，我们创建如下 BasicBlock 和 Value 容器：

```
typedef SmallVector<BasicBlock *, 16> BBList;
typedef SamllVector<Value *, 16> ValList;
```

注解

为简化编程，LLVM 提供了派生的 vector 容器 SmallVector<>。

我们生成一些 Value，添加到一个 Value*列表中，而后将在 if-else 基本块中处理它们：

```
Value *Condtn = Builder.CreateICmpNE(Compare, Builder.getInt32(0), “ifcond”);
ValList VL;
VL.push_back(Condtn);
VL.push_back(Arg1);
```

我们创建 3 个基本块，添加到一个 BasicBlock*列表中：

```
BasicBlock *ThenBB = createBB(fooFunc, "then");
BasicBlock *ElseBB = createBB(fooFunc, "else");
BasicBlock *MergeBB = createBB(fooFunc, "ifcont");
BBList List;
List.push_back(ThenBB);
List.push_back(ElseBB);
List.push_back(MergeBB);
```

最后我们将这些基本块添加到函数中：

```
Value *createIfElse(IRBuilder<> &Builder, BBList List, ValList VL) {
```

```
    Value *Condtn = VL[0];
    Value *Arg1 = VL[1];
    BasicBlock *ThenBB = List[0];
    BasicBlock *ElseBB = List[1];
    BasicBlock *MergeBB = List[2];
    Builder.CreateCondBr(Condtn, ThenBB, ElseBB);

    Builder.SetInsertPoint(ThenBB);
    Value *ThenVal = Builder.CreateAdd(Arg1, Builder.getInt32(1),
"thenaddtmp");
    Builder.CreateBr(MergeBB);

    Builder.SetInsertPoint(ElseBB);
    Value *ElseVal = Builder.CreateAdd(Arg1, Builder.getInt32(2),
"elseaddtmp");
    Builder.CreateBr(MergeBB);

    unsigned PhiBBSize = List.size() - 1;
    Builder.SetInsertPoint(MergeBB);
    PHINode *Phi = Builder.CreatePHI(Type::getInt32Ty(getGlobalContext()),
PhiBBSize, "iftemp");
    Phi->addIncomming(ThenVal, ThenBB);
    Phi->addIncomming(ElseVal, ElseBB);

    return Phi;
}
```

整体代码如下：

```
#include "llvm/IR/IRBuilder.h"
#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/Module.h"
#include "llvm/IR/Verifier.h"
#include <vector>
using namespace llvm;

static LLVMContext Context;
static Module *ModuleOb = new Module("my compiler", Context);
static std::vector <std::string> FunArgs;
typedef SmallVector<BasicBlock *, 16> BBList;
typedef SmallVector<Value *, 16> ValList;

Function *createFunc(IRBuilder<> &Builder, std::string Name) {
    std::vector<Type *> Integers(FunArgs.size(), Type::getInt32Ty(Context));
    FunctionType *funcType = llvm::FunctionType::get(Builder.getInt32Ty(),
```

```
Integers, false);
    Function *fooFunc = llvm::Function::Create(funcType,
llvm::Function::ExternalLinkage, Name, ModuleOb);
    return fooFunc;
}

void setFuncArgs(Function *fooFunc, std::vector<std::string> FunArgs) {
    unsigned Idx = 0;
    Function::arg_iterator AI, AE;
    for (AI = fooFunc->arg_begin(), AE = fooFunc->arg_end(); AI != AE; ++AI,
++Idx) {
        AI->setName(FunArgs[Idx]);
    }
}

BasicBlock *createBB(Function *fooFunc, std::string Name) {
    return BasicBlock::Create(Context, Name, fooFunc);
}

GlobalVariable *createGlob(IRBuilder<> &Builder, std::string Name) {
    ModuleOb->getOrInsertGlobal(Name, Builder.getInt32Ty());
    GlobalVariable *gVar = ModuleOb->getNamedGlobal(Name);
    gVar->setLinkage(GlobalValue::CommonLinkage);
    gVar->setAlignment(4);
    return gVar;
}

Value *createArith(IRBuilder<> &Builder, Value *L, Value *R) {
    return Builder.CreateMul(L, R, "multmp");
}

Value *createIfElse(IRBuilder<> &Builder, BBList List, ValList VL) {
    Value *Condtn = VL[0];
    Value *Arg1 = VL[1];
    BasicBlock *ThenBB = List[0];
    BasicBlock *ElseBB = List[1];
    BasicBlock *MergeBB = List[2];
    Builder.CreateCondBr(Condtn, ThenBB, ElseBB);

    Builder.SetInsertPoint(ThenBB);
    Value *ThenVal = Builder.CreateAdd(Arg1, Builder.getInt32(1),
"thenaddtmp");
    Builder.CreateBr(MergeBB);
```

```
    Builder.SetInsertPoint(ElseBB);
    Value *ElseVal = Builder.CreateAdd(Arg1, Builder.getInt32(2),
"elseaddtmp");
    Builder.CreateBr(MergeBB);

    unsigned PhiBBSize = List.size() - 1;
    Builder.SetInsertPoint(MergeBB);
    PHINode *Phi = Builder.CreatePHI(Type::getInt32Ty(Context), PhiBBSize,
"iftemp");
    Phi->addIncoming(ThenVal, ThenBB);
    Phi->addIncoming(ElseVal, ElseBB);

    return Phi;
}

int main(int argc, char *argv[]) {
    FunArgs.push_back("a");
    FunArgs.push_back("b");
    static IRBuilder<> Builder(Context);
    GlobalVariable *gVar = createGlob(Builder, "x");
    Function *fooFunc = createFunc(Builder, "foo");
    setFuncArgs(fooFunc, FunArgs);
    BasicBlock *entry = createBB(fooFunc, "entry");
    Builder.SetInsertPoint(entry);
    Value *Arg1 = dyn_cast<Value>(fooFunc->arg_begin());
    Value *constant = Builder.getInt32(16);
    Value *val = createArith(Builder, Arg1, constant);

    Value *val2 = Builder.getInt32(100);
    Value *Compare = Builder.CreateICmpULT(val, val2, "cmptmp");
    Value *Condtn = Builder.CreateICmpNE(Compare, Builder.getInt1(0),
"ifcond");

    ValList VL;
    VL.push_back(Condtn);
    VL.push_back(Arg1);

    BasicBlock *ThenBB = createBB(fooFunc, "then");
    BasicBlock *ElseBB = createBB(fooFunc, "else");
    BasicBlock *MergeBB = createBB(fooFunc, "ifcont");
    BBList List;
    List.push_back(ThenBB);
    List.push_back(ElseBB);
    List.push_back(MergeBB);
```

```
    Value *v = createIfElse(Builder, List, VL);

    Builder.CreateRet(v);

    verifyFunction(*fooFunc);
    ModuleOb->dump();
    return 0;
}
```

编译后执行，输出如下：

```
$ ./toy
; ModuleID = 'my compiler'

@x = common global i32, align 4

define i32 @foo(i32 %a, i32 %b) {
entry:
  %multmp = mul i32 %a, 16
  %cmptmp = icmp ult i32 %multmp, 100
  %ifcond = icmp ne i1 %cmptmp, false
  br i1 %ifcond, label %then, label %else

then:                                             ; preds = %entry
  %thenaddtmp = add i32 %a, 1
  br label %ifcont

else:                                             ; preds = %entry
  %elseaddtmp = add i32 %a, 2
  br label %ifcont

ifcont:                                           ; preds = %else, %then
  %iftemp = phi i32 [ %thenaddtmp, %then ], [ %elseaddtmp, %else ]
  ret i32 %iftemp
}
```

生成循环语句

跟 if-else 语句类似，循环也可以用 LLVM API 生成，稍微修改代码即可。举例来说，

我们想生成下面的循环：

```
for (i = 1; i < b; i++) {body}
```

这个循环有一个诱导变量 i，它有初始值，每次迭代都会更新。在上面的例子中，诱导变量在每次迭代后增 1。它有循环终止条件。'i = 1'指定初始值，'i < b'是循环终止条件，'i++'递增诱导变量，增量是 1。

在编写创建循环的函数之前，得先生成若干 Value 和 BasicBlock，如下：

```
Function::arg_iterator AI = fooFunc->arg_begin();
Value *Arg1 = AI++;
Value *Arg2 = AI;
Value *constant = Builder.getInt32(16);
Value *val = createArith(Builder, Arg1, constant);
ValList VL;
VL.push_back(Arg1);

BBList List;
BasicBlock *LoopBB = createBB(fooFunc, "loop");
BasicBlock *AfterBB = createBB(fooFunc, "afterloop");
List.push_back(LoopBB);
List.push_back(AfterBB);

Value *StartVal = Builder.getInt32(1);
```

下面编写一个创建循环的函数：

```
PHINode *createLoop(IRBuilder<> &Builder, BBList List, ValList VL, Value *StartVal, Value *EndVal) {
    BasicBlock *PreheaderBB = Builder.GetInsertBlock();
    Value *val = VL[0];
    BasicBlock *LoopBB = List[0];
    Builder.CreateBr(LoopBB);
    Builder.SetInsertPoint(LoopBB);
    PHINode *IndVar = Builder.CreatePHI(Type::getInt32Ty(Context), 2, "i");
    IndVar->addIncoming(StartVal, PreheaderBB);
    Value *Add = Builder.CreateAdd(val, Builder.getInt32(5), "addtmp");
    Value *StepVal = Builder.getInt32(1);
    Value *NextVal = Builder.CreateAdd(IndVar, StepVal, "nextval");
    Value *EndCond = Builder.CreateICmpULT(IndVar, EndVal, "endcond");
    EndCond = Builder.CreateICmpNE(EndCond, Builder.getInt32(0), "loopcond");
    BasicBlock *LoopEndBB = Builder.GetInsertBlock();
    BasicBlock *AfterBB = List[1];
    Builder.CreateCondBr(EndCond, LoopBB, AfterBB);
    Builder.SetInsertPoint(AfterBB);
```

```
    IndVar->addIncoming(NextVal, LoopEndBB);
    return Add;
}
```

考虑下面的代码：

```
IndVar->addIncoming(StartVal, PreheaderBB);...
IndVar->addIncoming(NextVal, LoopEndBB);
```

IndVar 是一条 PHI 指令，它有两个输入，分别来自两个基本块——StartVal（i = 1）来自

Preheader 基本块，NextVal 来自 LoopEnd 基本块。

整体代码如下：

```
#include "llvm/IR/IRBuilder.h"
#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/Module.h"
#include "llvm/IR/Verifier.h"
#include <vector>
using namespace llvm;

typedef SmallVector<BasicBlock *, 16> BBList;
typedef SmallVector<Value *, 16> ValList;

static LLVMContext Context;
static Module *ModuleOb = new Module("my compiler", Context);
static std::vector <std::string> FunArgs;

Function *createFunc(IRBuilder<> &Builder, std::string Name) {
    std::vector<Type *> Integers(FunArgs.size(), Type::getInt32Ty(Context));
    FunctionType *funcType = llvm::FunctionType::get(Builder.getInt32Ty(),
Integers, false);
    Function *fooFunc = llvm::Function::Create(funcType,
llvm::Function::ExternalLinkage, Name, ModuleOb);
    return fooFunc;
}

void setFuncArgs(Function *fooFunc, std::vector<std::string> FunArgs) {
    unsigned Idx = 0;
    Function::arg_iterator AI, AE;
    for (AI = fooFunc->arg_begin(), AE = fooFunc->arg_end(); AI != AE; ++AI,
++Idx) {
        AI->setName(FunArgs[Idx]);
    }
}
```

```
BasicBlock *createBB(Function *fooFunc, std::string Name) {
    return BasicBlock::Create(Context, Name, fooFunc);
}

GlobalVariable *createGlob(IRBuilder<> &Builder, std::string Name) {
    ModuleOb->getOrInsertGlobal(Name, Builder.getInt32Ty());
    GlobalVariable *gVar = ModuleOb->getNamedGlobal(Name);
    gVar->setLinkage(GlobalValue::CommonLinkage);
    gVar->setAlignment(4);
    return gVar;
}

Value *createArith(IRBuilder<> &Builder, Value *L, Value *R) {
    return Builder.CreateMul(L, R, "multmp");
}

Value *createLoop(IRBuilder<> &Builder, BBList List, ValList VL, Value *StartVal,
Value *EndVal) {
    BasicBlock *PreheaderBB = Builder.GetInsertBlock();
    Value *val = VL[0];
    BasicBlock *LoopBB = List[0];
    Builder.CreateBr(LoopBB);
    Builder.SetInsertPoint(LoopBB);
    PHINode *IndVar = Builder.CreatePHI(Type::getInt32Ty(Context), 2, "i");
    IndVar->addIncoming(StartVal, PreheaderBB);
    Value *Add = Builder.CreateAdd(val, Builder.getInt32(5), "addtmp");
    Value *StepVal = Builder.getInt32(1);
    Value *NextVal = Builder.CreateAdd(IndVar, StepVal, "nextval");
    Value *EndCond = Builder.CreateICmpULT(IndVar, EndVal, "endcond");
    EndCond = Builder.CreateICmpNE(EndCond, Builder.getInt1(0),
"loopcond");
    BasicBlock *LoopEndBB = Builder.GetInsertBlock();
    BasicBlock *AfterBB = List[1];
    Builder.CreateCondBr(EndCond, LoopBB, AfterBB);
    Builder.SetInsertPoint(AfterBB);
    IndVar->addIncoming(NextVal, LoopEndBB);
    return Add;
}

int main(int argc, char *argv[]) {
    FunArgs.push_back("a");
    FunArgs.push_back("b");
    static IRBuilder<> Builder(Context);
```

```
    GlobalVariable *gVar = createGlob(Builder, "x");
    Function *fooFunc = createFunc(Builder, "foo");
    setFuncArgs(fooFunc, FunArgs);
    BasicBlock *entry = createBB(fooFunc, "entry");
    Builder.SetInsertPoint(entry);

    Function::arg_iterator AI = fooFunc->arg_begin();
    Value *Arg1 = dyn_cast<Value>(AI++);
    Value *Arg2 = dyn_cast<Value>(AI);
    Value *constant = Builder.getInt32(16);
    Value *val = createArith(Builder, Arg1, constant);
    ValList VL;
    VL.push_back(Arg1);

    BBList List;
    BasicBlock *LoopBB = createBB(fooFunc, "loop");
    BasicBlock *AfterBB = createBB(fooFunc, "afterloop");
    List.push_back(LoopBB);
    List.push_back(AfterBB);

    Value *StartVal = Builder.getInt32(1);
    Value *Res = createLoop(Builder, List, VL, StartVal, Arg2);

    Builder.CreateRet(Res);

    verifyFunction(*fooFunc);
    ModuleOb->dump();
    return 0;
}
```

编译后执行，输出如下：

```
$ ./toy
; ModuleID = 'my compiler'

@x = common global i32, align 4

define i32 @foo(i32 %a, i32 %b) {
entry:
  %multmp = mul i32 %a, 16
  br label %loop

loop:                                             ; preds = %loop, %entry
  %i = phi i32 [ 1, %entry ], [ %nextval, %loop ]
  %addtmp = add i32 %a, 5
```

```
  %nextval = add i32 %i, 1
  %endcond = icmp ult i32 %i, %b
  %loopcond = icmp ne i1 %endcond, false
  br i1 %loopcond, label %loop, label %afterloop

afterloop:                                        ; preds = %loop
  ret i32 %addtmp
}
```

总结

在本章中我们学习了如何利用 LLVM 丰富的程序库创建简单的 LLVM IR。我们知道，LLVM 是一种中间表示。定制的解析器把高级语言转换成 LLVM IR，代码被分解为许多原子的实体，例如变量，函数，函数返回类型，函数参数，if-else 条件，循环，指针，数组等。这些原子元素可以存储在定制的数据结构中，正如本章中生成 LLVM IR 的示例所演示的那样。

编译器在扫描文本时作词法分析，在解析阶段作语法分析，在解析之后生成 IR 之前的中间阶段，作类型检查。

在实际应用中，很少有人像本章演示的那样以 hard code (原始)的方式生成 LLVM IR。相反，高级语言解析之后被表示为一棵语法树。而后遍历语法树并利用 LLVM 程序库生成 LLVM IR。LLVM 社区已经提供了优秀的教程，教导如何编写解析器以生成 LLVM IR。你可以访问 http://llvm.org/docs/tutorial 查看详细内容。

下一章我们将介绍如何生成复杂的数据结构，例如数组，指针等。我们将演示若干来自 C/C++前端编译器 Clang 的例子，理解语义分析是怎么做的。

第 3 章 高级 LLVM IR

LLVM 提供了强大的中间表示，使得编译器能够作高效的转换和分析，同时提供了原生的方法用于调试代码和可视化转换。LLVM IR 的设计使得它易于映射到高级语言。LLVM IR 提供了类型信息，能够用于各种优化处理。

上一章你学习了如何在函数和模块中创建一些简单的 LLVM 指令。从简单的例子开始，例如二元运算，我们为模块构造了函数，创建了复杂的程序范式，例如分支和循环。LLVM 提供了一套丰富的指令和本质函数（intrinsic）用以生成 IR。

本章我们将演示更多的 LLVM IR 例子，涉及到内存操作。也会涵盖一些高级的主题，例如聚合数据类型及其操作。本章的内容如下：

- 计算地址
- 读内存
- 写内存
- 向向量插入标量
- 从向量提取标量

内存访问操作

内存几乎是所有计算系统的重要组成部分。内存存储数据。计算系统从内存读取数据，然后执行运算。运算结果再存储回内存。

第 1 步是得到目标元素在内存中的地址，在这个位置可以找到目标元素。我们将学习如何计算地址，执行加载（load）存储（store）操作。

计算地址

LLVM 用 getelementptr 指令获取一个元素在聚合数据结构中的地址。它仅仅计算地址，不访问内存。

getelementptr 的第 1 个参数是类型，作为地址计算的基础。第 2 个参数是指针或者指针向量，作为基地址（base address）。后面的参数是若干索引，用于访问元素。

LLVM 语言参考页面对 getelementptr 指令提到如下重要注解：

（http://llvm.org/docs/LangRef.html#getelementptr-instruction）

第 1 个索引值总是索引第 1 个参数给出的指针，第 2 个索引值索引这个指针指向的类型的实体（不必是直接指向的实体，因为第 1 个索引值可能非 0）。第 1 个被索引的类型必须是指针，后续的类型可以是数组（array），向量（vector），结构（struct）。注意，后续的类型不能是指针，因为那样的话需要加载指针才能继续计算地址。

这隐含了两个要点：

1.指针总是有一个索引，这个索引即第 1 个索引，是数组索引。假如指针指向一个结构，想要访问结构的元素，那么第 1 个索引必须为 0（指示第 1 个结构），后面才是访问元素的索引。

2.getelementptr 根据第 1 个参数即类型，确定结构的长度和元素的长度，如此易于计算地址。结果的类型（指针类型）可以变化。

以下页面提供了更详细的解释：http://llvm.org/docs/GetElementPtr.html

假如我们有一个指针指向包含两个 32 位整数的向量，<2 x i32>* %a，我们想要访问向量中的第 2 个整数，可以这样获取地址：

%a1 = getelementptr <2 x i32>, <2 x i32>* %a, i32 0, i32 1

下面介绍如何用 LLVM API 生成这条指令。首先创建一个数组类型，它作为参数传递给

一个函数。

```
Function *createFunc(IRBuilder<> &Builder, std::string Name) {
    Type *u32Ty = Type::getInt32Ty(Context);
    Type *vecTy = VectorType::get(u32Ty, 2);
    Type *ptrTy = vecTy->getPointerTo(0);
    FunctionType *funcType = FunctionType::get(Builder.getInt32Ty(), ptrTy,
false);
    Function *fooFunc = Function::Create(funcType, Function::ExternalLinkage,
Name, ModuleOb);
    return fooFunc;
}

Value *getGEP(IRBuilder<> &Builder, Value *Base, Value *Offset) {
    Type *u32Ty = Type::getInt32Ty(Context);
    Type *vecTy = VectorType::get(u32Ty, 2);
    SmallVector<Value *, 4> ValList;
    ValList.push_back(Builder.getInt32(0));
    ValList.push_back(Offset);
    ArrayRef<Value *> Indices(ValList.begin(), ValList.end());
    return Builder.CreateGEP(vecTy, Base, Indices, "a1");
}
```

整体代码如下：

```
#include "llvm/IR/IRBuilder.h"
#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/Module.h"
#include "llvm/IR/Verifier.h"
#include <vector>
using namespace llvm;

static LLVMContext Context;
static Module *ModuleOb = new Module("my compiler", Context);
static std::vector <std::string> FunArgs;

Function *createFunc(IRBuilder<> &Builder, std::string Name) {
    Type *u32Ty = Type::getInt32Ty(Context);
    Type *vecTy = VectorType::get(u32Ty, 2);
    Type *ptrTy = vecTy->getPointerTo(0);
    FunctionType *funcType = FunctionType::get(Builder.getInt32Ty(), ptrTy,
false);
    Function *fooFunc = Function::Create(funcType, Function::ExternalLinkage,
Name, ModuleOb);
    return fooFunc;
```

```
}

void setFuncArgs(Function *fooFunc, std::vector<std::string> FunArgs) {
    unsigned Idx = 0;
    Function::arg_iterator AI, AE;
    for (AI = fooFunc->arg_begin(), AE = fooFunc->arg_end(); AI != AE; ++AI,
++Idx) {
        AI->setName(FunArgs[Idx]);
    }
}

BasicBlock *createBB(Function *fooFunc, std::string Name) {
    return BasicBlock::Create(Context, Name, fooFunc);
}

Value *getGEP(IRBuilder<> &Builder, Value *Base, Value *Offset) {
    Type *u32Ty = Type::getInt32Ty(Context);
    Type *vecTy = VectorType::get(u32Ty, 2);
    SmallVector<Value *, 4> ValList;
    ValList.push_back(Builder.getInt32(0));
    ValList.push_back(Offset);
    ArrayRef<Value *> Indices(ValList.begin(), ValList.end());
    return Builder.CreateGEP(vecTy, Base, Indices, "a1");
}

int main(int argc, char *argv[]) {
    FunArgs.push_back("a");
    static IRBuilder<> Builder(Context);
    Function *fooFunc = createFunc(Builder, "foo");
    setFuncArgs(fooFunc, FunArgs);
    Function::arg_iterator AI = fooFunc->arg_begin();
    Value *Base = dyn_cast<Value>(AI);
    BasicBlock *entry = createBB(fooFunc, "entry");
    Builder.SetInsertPoint(entry);
    Value *gep = getGEP(Builder, Base, Builder.getInt32(1));
    verifyFunction(*fooFunc);
    ModuleOb->dump();
    return 0;
}
```

编译代码并执行：

```
$ clang++ toy.cpp `llvm-config --cxxflags --ldflags --system-libs --libs core`
-fno-rtti -o toy
$ ./toy
```

输出如下：

```
; ModuleID = 'my compiler'

define i32 @foo(<2 x i32>* %a) {
entry:
  %a1 = getelementptr <2 x i32>, <2 x i32>* %a, i32 0, i32 1
  ret i32 0
}
```

读内存

至此，我们已经获得了地址，接着我们将从地址读取数据，把读取的值赋给变量。

LLVM 用 load 指令从某个内存位置读取数据。这条简单指令，或者类似指令的组合，将被映射为复杂的低级汇编内存读取指令。

load 指令接收一个参数，即内存地址，指示读取数据的位置。如前面介绍的那样，我们用 getelementptr 指令获取地址。

load 指令的样子如下：

%val = load i32, i32* a1

这表示 load 将读取 a1 指向的数据，保存到变量%val。

我们可以用 LLVM 提供的 API 生成 load 指令，如下所示：

```
Value *getLoad(IRBuilder<> &Builder, Value *Address) {
    return Builder.CreateLoad(Address, "load");
}
```

让它返回加载的值：

```
Value *load = getLoad(Builder, gep);
Builder.CreateRet(load);
```

整体代码如下：

```
#include "llvm/IR/IRBuilder.h"
#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/Module.h"
#include "llvm/IR/Verifier.h"
#include <vector>
using namespace llvm;
```

```
static LLVMContext Context;
static Module *ModuleOb = new Module("my compiler", Context);
static std::vector <std::string> FunArgs;

Function *createFunc(IRBuilder<> &Builder, std::string Name) {
    Type *u32Ty = Type::getInt32Ty(Context);
    Type *vecTy = VectorType::get(u32Ty, 2);
    Type *ptrTy = vecTy->getPointerTo(0);
    FunctionType *funcType = FunctionType::get(Builder.getInt32Ty(), ptrTy,
false);
    Function *fooFunc = Function::Create(funcType, Function::ExternalLinkage,
Name, ModuleOb);
    return fooFunc;
}

void setFuncArgs(Function *fooFunc, std::vector<std::string> FunArgs) {
    unsigned Idx = 0;
    Function::arg_iterator AI, AE;
    for (AI = fooFunc->arg_begin(), AE = fooFunc->arg_end(); AI != AE; ++AI,
++Idx) {
        AI->setName(FunArgs[Idx]);
    }
}

BasicBlock *createBB(Function *fooFunc, std::string Name) {
    return BasicBlock::Create(Context, Name, fooFunc);
}

Value *getGEP(IRBuilder<> &Builder, Value *Base, Value *Offset) {
    Type *u32Ty = Type::getInt32Ty(Context);
    Type *vecTy = VectorType::get(u32Ty, 2);
    SmallVector<Value *, 4> ValList;
    ValList.push_back(Builder.getInt32(0));
    ValList.push_back(Offset);
    ArrayRef<Value *> Indices(ValList.begin(), ValList.end());
    return Builder.CreateGEP(vecTy, Base, Indices, "a1");
}

Value *getLoad(IRBuilder<> &Builder, Value *Address) {
    return Builder.CreateLoad(Address, "load");
}

int main(int argc, char *argv[]) {
```

```
    FunArgs.push_back("a");
    static IRBuilder<> Builder(Context);
    Function *fooFunc = createFunc(Builder, "foo");
    setFuncArgs(fooFunc, FunArgs);
    Function::arg_iterator AI = fooFunc->arg_begin();
    Value *Base = dyn_cast<Value>(AI);
    BasicBlock *entry = createBB(fooFunc, "entry");
    Builder.SetInsertPoint(entry);
    Value *gep = getGEP(Builder, Base, Builder.getInt32(1));
    Value *load = getLoad(Builder, gep);
    Builder.CreateRet(load);
    verifyFunction(*fooFunc);
    ModuleOb->dump();
    return 0;
}
```

编译代码并执行：

```
$ clang++ toy.cpp `llvm-config --cxxflags --ldflags --system-libs --libs core` -fno-rtti -o toy
$ ./toy
```

输出如下：

```
; ModuleID = 'my compiler'

define i32 @foo(<2 x i32>* %a) {
entry:
  %a1 = getelementptr <2 x i32>, <2 x i32>* %a, i32 0, i32 1
  %load = load i32, i32* %a1
  ret i32 %load
}
```

写内存

LLVM 用 store 指令写内存位置。store 指令有两个参数：存储的数值，和存储的地址。它没有返回值。假设我们想要在一个<2 x i32>向量的第 2 个元素处写一个值，举例来说

```
store i32 3, i32* %a1
```

我们可以 LLVM 提供的 API 生成 store 指令：

```
void getStore(IRBuilder<> &Builder, Value *Address, Value *V) {
    Builder.CreateStore(V, Address);
}
```

在下面的例子中，我们读取<2 x i32>向量的第 2 个元素，将它乘以 16，然后存回相同的位置。代码如下：

```
#include "llvm/IR/IRBuilder.h"
#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/Module.h"
#include "llvm/IR/Verifier.h"
#include <vector>
using namespace llvm;

static LLVMContext Context;
static Module *ModuleOb = new Module("my compiler", Context);
static std::vector <std::string> FunArgs;

Function *createFunc(IRBuilder<> &Builder, std::string Name) {
    Type *u32Ty = Type::getInt32Ty(Context);
    Type *vecTy = VectorType::get(u32Ty, 2);
    Type *ptrTy = vecTy->getPointerTo(0);
    FunctionType *funcType = FunctionType::get(Builder.getInt32Ty(), ptrTy, false);
    Function *fooFunc = Function::Create(funcType, Function::ExternalLinkage, Name, ModuleOb);
    return fooFunc;
}

void setFuncArgs(Function *fooFunc, std::vector<std::string> FunArgs) {
    unsigned Idx = 0;
    Function::arg_iterator AI, AE;
    for (AI = fooFunc->arg_begin(), AE = fooFunc->arg_end(); AI != AE; ++AI, ++Idx) {
        AI->setName(FunArgs[Idx]);
    }
}

BasicBlock *createBB(Function *fooFunc, std::string Name) {
    return BasicBlock::Create(Context, Name, fooFunc);
}

Value *createArith(IRBuilder<> &Builder, Value *L, Value *R) {
    return Builder.CreateMul(L, R, "multmp");
}

Value *getGEP(IRBuilder<> &Builder, Value *Base, Value *Offset) {
```

```
    Type *u32Ty = Type::getInt32Ty(Context);
    Type *vecTy = VectorType::get(u32Ty, 2);
    SmallVector<Value *, 4> ValList;
    ValList.push_back(Builder.getInt32(0));
    ValList.push_back(Offset);
    ArrayRef<Value *> Indices(ValList.begin(), ValList.end());
    return Builder.CreateGEP(vecTy, Base, Indices, "a1");
}

Value *getLoad(IRBuilder<> &Builder, Value *Address) {
    return Builder.CreateLoad(Address, "load");
}

void getStore(IRBuilder<> &Builder, Value *Address, Value *V) {
    Builder.CreateStore(V, Address);
}

int main(int argc, char *argv[]) {
    FunArgs.push_back("a");
    static IRBuilder<> Builder(Context);
    Function *fooFunc = createFunc(Builder, "foo");
    setFuncArgs(fooFunc, FunArgs);
    Function::arg_iterator AI = fooFunc->arg_begin();
    Value *Base = dyn_cast<Value>(AI);
    BasicBlock *entry = createBB(fooFunc, "entry");
    Builder.SetInsertPoint(entry);
    Value *gep = getGEP(Builder, Base, Builder.getInt32(1));
    Value *load = getLoad(Builder, gep);
    Value *constant = Builder.getInt32(16);
    Value *val = createArith(Builder, load, constant);
    getStore(Builder, gep, val);
    Builder.CreateRet(val);
    verifyFunction(*fooFunc);
    ModuleOb->dump();
    return 0;
}
```

编译并执行：

```
$ clang++ toy.cpp `llvm-config --cxxflags --ldflags --system-libs --libs core` -fno-rtti -o toy
$ ./toy
```

输出如下：

```
; ModuleID = 'my compiler'
```

```
define i32 @foo(<2 x i32>* %a) {
entry:
  %a1 = getelementptr <2 x i32>, <2 x i32>* %a, i32 0, i32 1
  %load = load i32, i32* %a1
  %multmp = mul i32 %load, 16
  store i32 %multmp, i32* %a1
  ret i32 %multmp
}
```

向向量插入标量

LLVM IR 定义了向一个向量（vector）插入一个标量（scalar）的指令。LLVM 也提供了生成这样的指令的 API。注意，向量不同于数组。向量类型是一种简单的派生类型，表示一组元素。在单指令多数据（SIMD: single instruction multiple data）运行方式下并行操作多路数据时，向量类型派上用处。定义向量类型需要指定长度（元素的数量）和内在元素的类型。例如，一个<4 x i32>向量 Vec 包含 4 个 i32 类型的整数。我们想在向量的 0,1,2,3 索引位置分别插入 10,20,30 和 40。

LLVM 提供了一条 insertelement 指令，它有三个参数。第 1 个参数是一个向量值。第 2 个参数是一个标量值，它的类型必须与第 1 个参数给出的向量的元素类型相同。第 3 个参数是一个索引值，指示在向量的什么位置插入这个标量值。结果是一个相同类型的向量值。

举例来说，有一条这样的 insertelement 指令：

```
%vec0 = insertelement <4 x i32> Vec, %val0, %idx
```

如上所述，我们这样理解它：

- Vec 的类型是向量类型<4 x i32>
- val0 是待插入的标量值
- idx 是一个索引值，指示在向量的什么位置插入标量值

考虑下面的代码：

```
#include "llvm/IR/IRBuilder.h"
#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/Module.h"
#include "llvm/IR/Verifier.h"
#include <vector>
using namespace llvm;

static LLVMContext Context;
static Module *ModuleOb = new Module("my compiler", Context);
static std::vector <std::string> FunArgs;

Function *createFunc(IRBuilder<> &Builder, std::string Name) {
    Type *u32Ty = Type::getInt32Ty(Context);
    Type *vecTy = VectorType::get(u32Ty, 4);
    FunctionType *funcType = FunctionType::get(Builder.getInt32Ty(), vecTy,
false);
    Function *fooFunc = Function::Create(funcType, Function::ExternalLinkage,
Name, ModuleOb);
    return fooFunc;
}

void setFuncArgs(Function *fooFunc, std::vector<std::string> FunArgs) {
    unsigned Idx = 0;
    Function::arg_iterator AI, AE;
    for (AI = fooFunc->arg_begin(), AE = fooFunc->arg_end(); AI != AE; ++AI,
++Idx) {
        AI->setName(FunArgs[Idx]);
    }
}

BasicBlock *createBB(Function *fooFunc, std::string Name) {
    return BasicBlock::Create(Context, Name, fooFunc);
}

Value *getInsertElement(IRBuilder<> &Builder, Value *Vec, Value *Val, Value
*Index) {
    return Builder.CreateInsertElement(Vec, Val, Index);
}

int main(int argc, char *argv[]) {
    FunArgs.push_back("a");
    static IRBuilder<> Builder(Context);
```

```
    Function *fooFunc = createFunc(Builder, "foo");
    setFuncArgs(fooFunc, FunArgs);

    BasicBlock *entry = createBB(fooFunc, "entry");
    Builder.SetInsertPoint(entry);

    Value *Vec = dyn_cast<Value>(fooFunc->arg_begin());
    for (unsigned int i = 0; i < 4; i++) {
        Value *V = getInsertElement(Builder, Vec, Builder.getInt32((i+1)*10),
Builder.getInt32(i));
    }

    Builder.CreateRet(Builder.getInt32(0));
    verifyFunction(*fooFunc);
    ModuleOb->dump();
    return 0;
}
```

编译代码并执行：

```
$ clang++ toy.cpp `llvm-config --cxxflags --ldflags --system-libs --libs core`
-fno-rtti -o toy
$ ./toy
```

输出如下：

```
; ModuleID = 'my compiler'

define i32 @foo(<4 x i32> %a) {
entry:
  %0 = insertelement <4 x i32> %a, i32 10, i32 0
  %1 = insertelement <4 x i32> %a, i32 20, i32 1
  %2 = insertelement <4 x i32> %a, i32 30, i32 2
  %3 = insertelement <4 x i32> %a, i32 40, i32 3
  ret i32 0
}
```

向量 Vec 的内容将会是<10, 20, 30, 40>。

从向量提取标量

LLVM 提供了 extractelement 指令，从一个向量中提取一个单独的标量元素。这个指令有两个参数。第 1 个参数是一个向量值。第 2 个参数是一个索引值，指示在向量的什么

位置提取元素。

举例来说，这是一条 extractelement 指令：

```
%result = extractelement <4 x i32> %vec, i32 %idx
```

如上所述，我们这样理解它：

- vec 是一个向量值
- idx 是一个索引值，指示在向量的什么位置提取元素
- result 是一个标量值，这里它为 i32 类型

考虑下面的代码：

```
#include "llvm/IR/IRBuilder.h"
#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/Module.h"
#include "llvm/IR/Verifier.h"
#include <vector>
using namespace llvm;

static LLVMContext Context;
static Module *ModuleOb = new Module("my compiler", Context);
static std::vector <std::string> FunArgs;

Function *createFunc(IRBuilder<> &Builder, std::string Name) {
    Type *u32Ty = Type::getInt32Ty(Context);
    Type *vecTy = VectorType::get(u32Ty, 4);
    FunctionType *funcType = FunctionType::get(Builder.getInt32Ty(), vecTy,
false);
    Function *fooFunc = Function::Create(funcType, Function::ExternalLinkage,
Name, ModuleOb);
    return fooFunc;
}

void setFuncArgs(Function *fooFunc, std::vector<std::string> FunArgs) {
    unsigned Idx = 0;
    Function::arg_iterator AI, AE;
    for (AI = fooFunc->arg_begin(), AE = fooFunc->arg_end(); AI != AE; ++AI,
++Idx) {
        AI->setName(FunArgs[Idx]);
    }
}
```

```
BasicBlock *createBB(Function *fooFunc, std::string Name) {
    return BasicBlock::Create(Context, Name, fooFunc);
}

Value *createArith(IRBuilder<> &Builder, Value *L, Value *R) {
    return Builder.CreateAdd(L, R, "add");
}

Value *getExtractElement(IRBuilder<> &Builder, Value *Vec, Value *Index) {
    return Builder.CreateExtractElement(Vec, Index);
}

int main(int argc, char *argv[]) {
    FunArgs.push_back("a");
    static IRBuilder<> Builder(Context);
    Function *fooFunc = createFunc(Builder, "foo");
    setFuncArgs(fooFunc, FunArgs);

    BasicBlock *entry = createBB(fooFunc, "entry");
    Builder.SetInsertPoint(entry);

    Value *Vec = dyn_cast<Value>(fooFunc->arg_begin());
    SmallVector<Value *, 4> V;
    for (unsigned int i = 0; i < 4; i++) {
        Value *Val = getExtractElement(Builder, Vec, Builder.getInt32(i));
        V.push_back(Val);
    }
    Value *add1 = createArith(Builder, V[0], V[1]);
    Value *add2 = createArith(Builder, add1, V[2]);
    Value *add = createArith(Builder, add2, V[3]);

    Builder.CreateRet(add);
    verifyFunction(*fooFunc);
    ModuleOb->dump();
    return 0;
}
```

编译并执行：

```
$ clang++ toy.cpp `llvm-config --cxxflags --ldflags --system-libs --libs core` -fno-rtti -o toy
$ ./toy
```

输出如下：

```
; ModuleID = 'my compiler'

define i32 @foo(<4 x i32> %a) {
entry:
  %0 = extractelement <4 x i32> %a, i32 0
  %1 = extractelement <4 x i32> %a, i32 1
  %2 = extractelement <4 x i32> %a, i32 2
  %3 = extractelement <4 x i32> %a, i32 3
  %add = add i32 %0, %1
  %add1 = add i32 %add, %2
  %add2 = add i32 %add1, %3
  ret i32 %add2
}
```

总结

对于大多数目标架构来说，内存操作都是重要的。一些架构为内存读写操作提供了复杂的指令。有的甚至直接在内存单元上执行二元运算，而有的在执行运算之前加载内存数据到寄存器（CISC vs RISC）。LLVM 中很多 load-store 操作由本质函数（intrinsic）给出。关于本质函数，请参考 http://llvm.org/docs/LangRef.html#masked-vector-load-and-store-intrinsics

LLVM IR 为所有机器架构提供了共同的表达方式。它为内存数据操作和聚合类型数据操作提供了基本的指令。当 LLVM IR 转换为一种架构的指令时，可能结合多条 IR 指令生成特定的机器指令。在本章中，我们学习了一些高级的 IR 指令和它们的例子。想获得详细的信息，请访问 http://llvm.org/docs/LangRef.html，它为 LLVM IR 提供权威的学习资料。

下一章我们将学习 LLVM IR 是如何被优化从而减少指令的数量生成简洁的代码。

第 4 章 基础转换

至此，我们已经知道，LLVM IR 不依赖于目标机器，它产生的代码适用于任意特定的后端编译器(backend)。为了给后端生成高效的代码，我们优化由前端编译器(frontend) 产生的 IR，利用 LLVM Pass 管理器运行一系列分析和转换 Pass。值得注意的是，一个编译器的大多数优化发生在 IR 之上，理由之一是，IR 是目标可重定向的，同样的优化适用于很多目标。这减少了为多个目标设计同一优化算法的工作量。当然，有些优化是目标相关的，它们发生在 DAG (有向无环图) 选择阶段，这在后面会介绍。优化以 IR 为对象的理由之二是，LLVM IR 是 SSA 形式，这意味着每个变量仅仅赋值一次，每次新的赋值产生新的变量。SSA 表示的一个非常明显的好处是，我们不必分析多个变量之间的相关性，即一个变量被赋以另一个变量的值。很多优化算法受益于 SSA 表示，例如常量传播，死亡代码消除等。此后，我们将介绍 LLVM 中一些重要的优化，LLVM Pass 基础设施的功能，以及如何用 opt 工具执行不同的优化。

本章我们将学习以下内容：

- opt 工具
- Pass 和 Pass 管理器
- 使用其它 Pass 的信息
- 简化 IR 的例子
- 结合 IR 的例子

opt 工具

opt 是 LLVM 的优化和分析工具，运行在 LLVM IR 之上，优化代码或者分析代码。在第 1 章中我们已经见识过 opt 工具，简要介绍了如何用它分析和转换 LLVM IR。接下来，

让我们看看它还能做些什么。我们必须了解，opt 是一个开发工具，它所提供的所有优化程序，也能被前端编译器调用。

使用 opt 工具时，我们可以按需指定优化级别 (level)，也就是说，可以指定从 O0, O1, O2 到 O3 的优化级别（O0 优化强度最小，O3 优化强度最大）。除此之外，还有两个优化级别，Os 和 Oz，它们优化空间，即减小代码长度。通过 opt 工具运行优化程序的命令行格式如下：

```
$ opt -Ox -S input.ll
```

这里，x 代表优化级别，它可以是 0 到 3 的一个数值，或者 s，或者 z。这些优化级别类似于 Clang 所用的优化级别。-O0 表示不作优化，-O1 表示启用少量优化，-O2 表示启用适度优化，而-O3 表示启用最强优化。-O3 与-O2 类似，只是它相对允许优化运行得更久，可能生成更长的代码（-O3 不保证生成的代码是最优最高效的，它只意味着编译器尽其所能优化代码，甚至可能以失败告终）。-Os 表示优化代码长度，一般地意味着不运行那些增加代码长度的优化（例如，不作 slp-vectorizer 优化），运行减小代码长度的优化（例如，指令结合优化）。

我们可以让 opt 工具按照我们的需要运行一个的指定的 Pass。可以运行一个 LLVM 定义的 Pass（参考 http://llvm.org/docs/Passes.html 中的列表），或者一个自己编写的 Pass。上述链接中列出的 Pass 运行于-O1, -O2, -O3 优化级别之下。想要查看哪些 Pass 运行于某个优化级别之下，用命令行选项-debug-pass=Structure 运行 opt 即可。

让我们用一个例子演示 O1 和 O2 优化级别之间的差别。一般来说，O3 优化级别包含来自 O2 的一两个 Pass（而比较 O1 和 O2 更易说明问题）。我们给出一个例子，看看 O2 级别优化 Pass 对代码优化到什么程度。编写如下测试代码文件 test.ll：

```
define internal i32 @test(i32* %X, i32* %Y)
{
    %A = load i32, i32* %X
```

```
    %B = load i32, i32* %Y
    %C = add i32 %A, %B
    ret i32 %C
}
define internal i32 @caller(i32* %B)
{
    %A = alloca i32
    store i32 1, i32* %A
    %C = call i32 @test(i32* %A, i32* %B)
    ret i32 %C
}
define i32 @callercaller()
{
    %B = alloca i32
    store i32 2, i32* %B
    %X = call i32 @caller(i32* %B)
    ret i32 %X
}
```

在以上测试代码中，函数 callercaller 调用函数 caller，后者接着调用函数 test，它相加两个数字，返回结果给调用者 caller，caller 又转而返回结果给 callercaller。

下面分别运行 O1 和 O2 级别优化：

```
$ opt -O1 -S test.ll > 1.ll
$ opt -O2 -S test.ll > 2.ll
```

下面的截屏图片对比两者优化后的代码：

1.ll (O1)　　2.ll (O2)

我们看到，O2 已经优化了代码中的函数调用和加法运算，直接从 callercaller 函数返

回加法结果。之所以获得如此效果，是因为 O2 优化级别会运行 always-inline Pass，内联所有函数调用，将代码做成一个大的函数。它也运行 globaldce Pass，消除代码中不可达的全局变量。之后是 constmerge Pass，合并重复的全局常量为单个常量。它还运行 gvn Pass (全局值编号)，消除部分或者全部冗余的指令，这里消除的是冗余的 load 指令。

Pass 和 Pass 管理器

Pass 基础设施是 LLVM 系统的重要功能之一。通过 Pass 基础设施，我们可以调用大量分析和优化 Pass。学习 LLVM Pass 的起点是 Pass 类（C++ class），它是所有 Pass 的基类。当我们定义新的 Pass 时，需要根据它的用途继承预定义的子类。

- **ModulePass**：这是最常用的子类。一个 Pass 继承这个子类之后，它将一次分析整个模块。它引用模块中的所有函数，引用的次序可能不是固定的。具体做法是，编写一个继承 ModulePass 的 Pass 类，重载 runOnModule 函数。

注解

在继续介绍其它 Pass 类之前，让我们看看 Pass 类需要重写的 3 个虚函数：

- **doInitialization**：用于初始化与当前被处理函数不相关的内容。
- **runOn{Passtype}**：这个方法是我们实现 Pass 子类的功能的地方。对于 FunctionPass 它将是 runOnFunction，对于 LoopPass 它将是 runOnLoop，等。
- **doFinalization**：当 runOn{Passtype}处理完毕模块中的所有函数之后，它将被调用。

- **FunctionPass**：这类 Pass 处理模块中的每个函数，各个函数的处理过程各不相关。函数处理的先后次序是不确定的。不允许修改当前正在处理的函数之外的函数，也不允

许增加或者删除当前模块中的函数。继承 FunctionPass 的子类需要重写上述三个虚函数，实现 runOnFunction 方法。

- **BasicBlockPass**：这类 Pass 处理模块中的每个基本块，各个基本块的处理过程各不相关。不允许增加或者删除任何基本块，也不允许改变 CFG（control flow graph）。FunctionPass 不允许做的事情，它也不允许做。为了实现它们，可以重写 FunctionPass 的 doInitialization 和 doFinalization 方法，或者重写它们自己的同名的两个虚函数和 runOnBasicBlock 方法。
- **LoopPass**：这类 Pass 处理函数中的循环，各个循环的处理各不相关。首先处理最内层循环，依次向外，直到最外层循环。为了实现它们，需要重写 doInitialization，doFinalization 和 runOnLoop 方法。

下面，我们学习如何开始编写一个定制的 Pass。让我们编写这样一个 Pass，它将打印所有函数的名字。

在开始编程之前，我们需要修改几处代码，使得这个 Pass 可以被识别和运行。

我们需要在 LLVM 目录树中创建一个目录，lib/Transforms/FnNamePrint。在这个目录中，我们需要创建一个 Makefile，包含以下内容，使得这个 Pass 可以被编译：

```
LEVEL = ../../..

LIBRARYNAME = FnNamePrint

LOADABLE_MODULE = 1

include $(LEVEL)/Makefile.common
```

这表示所有的.cpp 文件将编译和链接为一个共享目标（shared object），它将出现在 lib 目录中（builder-foler/lib/FnNamePrint）。

下面我们真正开始编写这个 Pass。在 lib/Transforms/FnNamePrint 目录中创建一个

源文件，命名为 FnNamePrint.cpp。第一步是选择一个正确的子类。在这个例子中，我们想要打印每个函数的名字，则 FunctionPass 类符合我们的意图，它一次处理一个函数。还有，由于我们只要打印函数名字而不修改函数内容，FunctionPass 是简单之选。我们也可以选择 ModulePass，因为它是不可变 Pass（Immutable Pass）（可访问模块中的所有函数）。

我们编写如下代码：

```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"

using namespace llvm;

namespace {
    struct FnNamePrint: public FunctionPass {
        static char ID;
        FnNamePrint() : FunctionPass(ID) {}
        bool runOnFunction(Function &F) override {
            errs() << "Function " << F.getName() << '\n';
            return false;
        }
    };
}

char FnNamePrint::ID = 0;
static RegisterPass< FnNamePrint > X("funcnameprint", "Function Name Print",
false, false);
```

上面的代码首先包含了必需的头文件，使用了 llvm 名字空间。

```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"

using namespace llvm;
```

我们声明 Pass FnNamePrint 为一个结构，是 FunctionPass 的子类。在函数 runOnFunction 中，我们实现了打印名字的逻辑。在末尾返回的 bool 值指示我们是否修

改了函数的任何内容。如果函数被修改了，就返回 true，否则返回 false。在这个例子中，我们没作任何修改，故返回 false。

```
struct FnNamePrint: public FunctionPass {
    static char ID;
    FnNamePrint() : FunctionPass(ID) {}
    bool runOnFunction(Function &F) override {
        errs() << "Function " << F.getName() << '\n';
        return false;
    }
};
```

然后，我们声明了 Pass 的 ID，用于识别这个 Pass：

```
char FnNamePrint::ID = 0;
```

最后，我们需要用 Pass 管理器注册这个 Pass。第 1 个参数是 Pass 的名字，opt 工具用它辨别这个 Pass。第 2 个参数实际的 Pass 名字。第 3 个参数指定它是否修改 CFG，第 4 个参数指定它是不是一个分析 Pass。

```
static RegisterPass< FnNamePrint > X("funcnameprint", "Function Name Print", false, false);
```

注解

这就是 Pass 的实现。在使用它之前，必须用 make 命令编译 LLVM，它将编译得到.so 文件（build-folder/lib/FnNamePrint.so）。

译注

如果 LLVM 用 cmake 命令产生 Makefile，我们需要为 FnNamePrint 目录创建一个 CMakeLists.txt 文件，内容如下：

```
if(WIN32 OR CYGWIN)
  set(LLVM_LINK_COMPONENTS Core Support)
endif()

add_llvm_loadable_module( FnNamePrint
  FnNamePrint.cpp

  DEPENDS
  intrinsics_gen
```

```
  PLUGIN_TOOL
  opt
  )
```

然后，在 lib/Transforms/CMakeLists.txt 文件中增加一行代码：

```
add_subdirectory(FnNamePrint)
```

这样，就可以顺利执行 cmake 命令了（例如，cmake -G "Unix Makefiles" ..）。

至此，我们可以通过 opt 工具运行这个 Pass，方法如下：

```
$ opt -load path-to-llvm/build/lib/FnNamePrint.so -funcnameprint test.ll
```

命令行选项-load 指定 Pass 的.so 文件的路径，选项-funcnameprint 告诉 opt 运行我们所写的这个 Pass。test.ll 是测试用例。

这个 Pass 打印用例程序中所有函数的名字。对于第 1 节中的例子，它将打印出：

```
Function test
Function caller
Function callercaller
```

我们已经初步介绍了编写 Pass 的方法。接下来，我们将看看 PassManager 类在 LLVM 中扮演的重要角色。

PassManager 类安排各个 Pass 高效地运行。所有需要运行 Pass 的 LLVM 工具都通过 PassManager 运行 Pass。PassManager 负责保证 Pass 之间正确地交互。为了让各个 Pass 以良好的方式运行，PassManager 必须知道 Pass 之间如何交互，Pass 之间的依赖关系有何不同。

一个 Pass 自己可以指定它对其它 Pass 的依赖，就是说，运行当前 Pass 之前，必须运行哪些 Pass。它还可以指明，当前 Pass 运行之后哪些 Pass 将失效。PassManager 分析依赖关系，然后运行 Pass。后面我们将看到一个 Pass 如何指定依赖关系。

PassManager 的重要工作是恰当地分析依赖关系，避免重复计算。为此，它追踪哪些分析结果是存在的，哪些是无效的，哪些是需要的。它还追踪分析结果的生命周期，不再需要之后释放所用的内存，优化内存使用。

为了优化编译器的内存使用，提升高速缓存（cache）的效率，PassManager 管线式组合所有 Pass。当有一组连续的 FunctionPass 待运行时，它将首先对第 1 个函数运行这些 Pass，然后对第 2 个函数，依次类推。这提升了高速缓存的效率，因为它仅仅处理单个函数的 LLVM IR，而不是整个程序。

当 opt 以-debug-pass=Argument 选项运行时，PassManager 让我们看到 Pass 之间是如何交互的，每个 Pass 都做了什么。用-debug-pass=Structure 选项，我们可以看到 Pass 是如何运行的。它也将给出运行过的 Pass 的名字。让我们以第 1 节中的测试代码为例子。

```
$ opt -O2 -S test.ll -debug-pass=Structure

$ opt -load path-to-llvm/build/lib/FnNamePrint.so test.ll -funcnameprint
-debug-pass=Structure

Pass Arguments:  -targetlibinfo -tti -funcnameprint -verify
Target Library Information
Target Transform Information
  ModulePass Manager
    FunctionPass Manager
      Function Name Print
      Module Verifier
Function test
Function caller
Function callercaller
```

在以上输出中，Pass Arguments 显示所运行的 Pass，后面的列表是运行每个 Pass 所用到的结构。在 ModulePass Manager 之后，显示对每个模块运行的 Pass（这里是空的）。在 FunctionPass Manager 层次，显示对每个函数运行的 Pass（Function Name Print 和 Module Verifier），正如我们期待的那样。

PassManager（opt）还提供了其它有用的选项，其中一些如下：

-time-passes：列出运行的 Pass 和时间信息。

-stats：打印每个 Pass 的统计信息。

-instcount：计数每种指令的数量。必须同时使用-instcount 和-stats，opt 才会输出指令数量信息。

使用其它 Pass 的信息

为了更好地工作，PassManager 需要知道 Pass 之间的依赖关系。每个 Pass 可以自己声明依赖关系：当前 Pass 运行之前需要运行的分析 Pass，当前 Pass 运行之后变为无效的 Pass。为了指定这些依赖关系，一个 Pass 需要实现 getAnalysisUsage 方法。

```
virtual void
getAnalysisUsage(AnalysisUsage &Info) const;
```

借助这个方法，当前 Pass 可以指定 Pass 依赖关系，具体信息填充在 AnalysisUsage 对象中。为了填充依赖关系信息，Pass 需要调用如下方法：

AnalysisUsage::addRequired<>方法

这个方法指定当前 Pass 的依赖 Pass，将它们排列在前面使之优先执行。例如，对于内存复制优化来说，它需要别名分析（alias analysis）的结果：

```
void getAnalysisUsage(AnalysisUsage &AU) const override {
    AU.addRequired<AliasAnalysis>();
}
```

这保证别名分析 Pass 在内存复制优化（MemCpyOpt）之前执行，而且当别名分析结果被其它 Pass 作废之后，它将在内存复制优化之前再次执行。

AnalysisUsage::addRequiredTransitive<>方法

当一个分析 Pass 串联其它分析 Pass 时，我们用这个方法指定依赖关系，而不是用 addRequired 方法。就是说，我们用这个方法，指定若干分析 Pass 按照规定的次序运行。例如：

```
void
```

```
DependenceAnalysis::getAnalysisUsage(AnalysisUsage &AU) const {
    ...
    AU.addRequiredTransitive<AliasAnalysis>();
    AU.addRequiredTransitive<ScalarEvolution>();
    AU.addRequiredTransitive<LoopInfo>();
}
```

这里，DependenceAlalysis（向上）串联 AliasAnalysis，ScalarEvolution 和 LoopInfo Pass，并且依赖它们的结果。

AnalysisUsage::addPreserved<>方法

借助这个方法，一个 Pass 可以标明它对哪些 Pass 不产生影响。这就是说，这个 Pass 运行之后，它们的分析结果仍然存在并且有效，如果已经存在的话。这意味着，依赖它们的后续 Pass 可以使用它们的结果，而不需要再次运行它们。

例如，前面说到的内存复制优化（MemCpyOpt）Pass，它需要 AliasAnalysis Pass，同时也保持它的分析结果。

```
void getAnalysisUsage(AnalysisUsage &AU) const override {
    ...
    AU.addPreserved<AliasAnalysis>();
    ...
}
```

为了更细致地理解多个 Pass 是如何连接并且一起工作的,你可以挑选任意的转换 Pass（Transformation Pass），浏览它们的源代码，你将明白它们是如何获得并使用来自其它 Pass 的信息的。

指令简化示例

在这个章节，我们将了解 LLVM 如何合并指令使之简化。这里，它并不会创建新的指令。简化指令的一种方法是常量合并：

```
sub i32 2, 1 --> 1
```

这里，sub 指令简化成一个常数 1。

它也能处理非常量操作数，例如：

```
or i32 %x, 0 --> %x
and i32 %x, %x --> %x
```

这里，它们都简化成变量%x。这是一个已经存在的变量。

指令简化的代码实现在这个文件中：lib/Analysis/InstructionSimplify.cpp。

实现指令简化的一些重要方法是：

SimplifyBinOp 方法：用于简化二元操作，例如加法，减法，乘法等。它的函数签名形式如下：

```
static Value
*SimplifyBinOp(unsigned Opcode,
               Value *LHS,
               Value *RHS,
               const Query &Q,
               unsigned MaxRecurse)
```

这里，Opcode 表示需要简化的指令的操作符。LHS 和 RHS 表示操作符两边的操作数。MaxRecurse 是我们设定的最大递归深度，达到这个深度后，指令简化动作必须停止。

这个方法根据 Opcode 选择多路分支（switch case）：

```
switch (Opcode {
...
}
```

根据 Opcode，这个方法决定调用哪个函数来简化指令。其中一些函数如下：

- **SimplifyAddInst**：当操作数已知时，它尝试合并加法操作。合并方式的例子如下：

  ```
  X + undef --> undef
  X + 0 --> X
  X + (Y - X) --> Y 或者 (Y - X) + X --> Y
  ```

  在函数 static Value *SimplifyAddInst(Value *Op0, Value *Op1, bool isNSW, bool isNUW, const Query &Q, unsigned MaxRecurse)中，实现上面第 3 种合并方式的代码是这样的：

```
if (match(Op1, m_Sub(m_Value(Y), m_Specific(Op0))) ||
    match(Op0, m_Sub(m_Value(Y), m_Specific(Op1))))
     return Y;
```

这里，第 1 个条件匹配子表达式(Y-X)和 Operand1： m_Value(Y)代表 Y，m_Specific(Op0)代表 X。一旦匹配成功，则合并表达式为 Y 并返回它。第 2 个条件的处理是类似的。

- **SimplifySubInst**：当操作数已知时，它尝试合并减法操作。合并方式的例子如下：

X - undef --> undef
X - X --> 0
X - 0 --> X
X - (X - Y) --> Y

减法操作的匹配与合并方法与 SimplifyAddInst 类似。

- **SimplifyAndInst**：与前面两者类似，它尝试合并逻辑与表达式。合并方式的例子如下：

A & ~A = ~A & A = 0

在这个函数中，实现上述合并方式的代码是这样的：

```
if (match(Op0, m_Not(m_Specific(Op1))) ||
    match(Op1, m_Not(m_Specific(Op0))))
    return Constant::getNullValue(Op0->getType());
```

这里，它尝试匹配 A 和~A，当匹配成功时返回一个 NULL 值，即 0。

至此，我们已经了解一些指令简化方法。那么，当一组指令可以被替换为一组更高效的指令时，我们该如何处理呢？

指令结合示例

指令结合是一种 LLVM 优化 Pass。这种编译器技术把一组指令序列替换为更高效的指令，消耗的机器周期数量减少，而运算结果不变。指令结合不改变程序的 CFG，主要用于代数运算简化。指令结合和指令简化是明显不同，前者可能生成新的指令，后者则不会。指定-instcombine 参数，就可以让 opt 工具运行指令结合 Pass。这个 Pass 的代码实现在 lib/Transforms/InstCombine 目录中。例如，它将结合

%Y = add i32 %X, 1

%Z = add i32 %Y, 1

为

%Z = add i32 %X, 2

它删除了一个多余的 add 指令，结合两个 add 指令为一个。

LLVM 文档指出，这个 Pass 将对程序执行如下规范化动作：

- 二元运算的常量操作数将移动到操作符右侧（RHS）。
- 含有常量操作数的位运算将按类别分组，移位运算放在最前，然后位运算，与运算，异或运算。
- 如果可能，将比较运算的比较方式从<，>，<=，>=转换为==或!=。
- 针对布尔值的比较运算将被替换为逻辑运算。
- X + X 将被转换为 X*2，即 X<<1。
- 和一个 2 的幂的常量参数相乘的乘法运算将被转换为移位运算。

这个 Pass 由函数 bool InstCombiner::runOnFunction(Function &F)开始运行，位于文件 InstructionCombining.cpp 中。在目录 lib/Transforms/InstCombine 目录下，有若干文件，它们为不同的指令执行指令结合。在 InstCombine 模块中，有些方法在指令结合之前尝试简化指令，例如：

- **SimplifyAssociativeOrCommutative** 函数：用于简化可结合或者可交换的运算。对于可交换运算，它交换操作数，以减小复杂度。对于可结合运算，形如“(X op Y) op Z”，它把它转换为”X op (Y op Z)”，如果(Y op Z)可以简化的话。
- **tryFactorization** 函数：利用运算的交换律和分配律，通过提取公共因子简化二元运算。例如，(A * B) + (A * C)简化为 A * (B + C)。

下面，我们来看指令结合是怎么工作的。如前所述，不同的文件实现了多种不同的功能。我们给出一段测试代码，看看如何添加代码，指令结合才会作用于它。

为程序模式(A | (B ^ C)) ^ ((A ^ C) ^ B)编写测试代码 test.ll，它可以简化为(A & (B ^ C))：

```
define i32 @testfunc(i32 %x, i32 %y, i32 %z) {
    %xor1 = xor i32 %y, %z
    %or = or i32 %x, %xor1
    %xor2 = xor i32 %x, %z
    %xor3 = xor i32 %xor2, %y
    %res = xor i32 %or, %xor3
    ret i32 %res
}
```

LLVM 中处理如“与”，“或”和“异或”的代码在这个文件中：lib/Transforms/InstCombine/InstCombineAndOrXor.cpp。

在上述文件的 InstCombiner::visitXor(BinaryOperator &I)函数中，在条件语句 if (Op0I && Op1I)处，添加如下代码：

```
    if (match(Op0I, m_Or(m_Xor(m_Value(B), m_Value(C)), m_Value(A))) &&
        match(Op1I, m_Xor(m_Xor(m_Specific(A), m_Specific(C)), m_Specific(B)))) {
        return BinaryOperator::CreateAnd(A, Builder->CreateXor(B, C));
    }
```

不难理解的是，这段代码用于匹配模式(A | (B ^ C)) ^ ((A ^ C) ^ B)，匹配成功后返回(A & (B ^ C))。

为了测试它，我们编译 LLVM，然后对 test.ll 运行 instcombine Pass，看看输出。

```
$ opt -instcombine -S test.ll
define i32 @testfunc(i32 %x, i32 %y, i32 %z) {
    %1 = xor i32 %y, %z
    %res = and i32 %1, %x
    ret i32 %res
}
```

如此，我们看到它只有一条 xor 指令和一条 and 指令，而不是如之前的四条 xor 指令和一条 or 指令。

你可以浏览 InstCombine 目录中的源代码，尝试理解和添加更多的指令转换。

总结

在本章中，我们学习了如何对 LLVM IR 执行基础的转换。我们深入了解了 opt 工具，LLVMPass 基础设施，Pass 管理器，以及一个 Pass 如何利用另一个 Pass 的信息。我们以指令简化和指令结合的例子结束了本章。下一章我们将学习更多的高级转换，例如循环优化，标量进化，等等。它们处理一个基本块的程序，而不是个别指令。

第 5 章 基本块高级转换

在前面的章节中，我们学习了若干优化方法，它们主要针对指令层次。在本章中，我们将讨论针对基本块层次的优化，简化基本块代码，让它们更高效。我们将从循环开始，看看在 LLVM 中如何表示循环，如何利用 CFG 和支配关系优化循环。在处理循环时，我们将用到循环简化（LoopSimplify）和循环不变量代码移动优化。然后，我们将研究一个标量在程序执行过程中如何变化，其它优化 Pass 如何利用标量进化优化的结果。我们将看到 LLVM 如何表示内建函数，即 LLVM 本质函数（intrinsic）。最后，我们将看到 LLVM 如何发挥并行化概念，理解它的向量化方法。

在本章中，我们将学习如下内容：

- 处理循环
- 标量进化
- LLVM 本质函数
- 向量化

处理循环

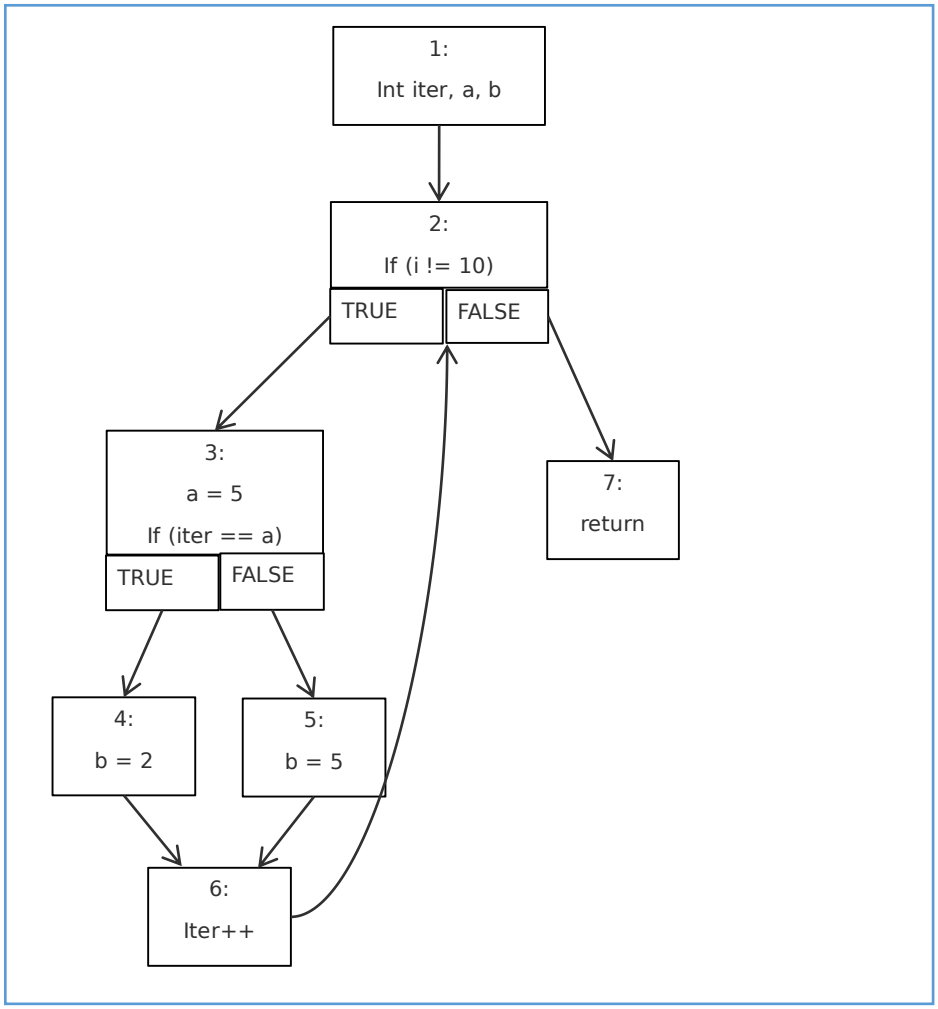
在开始讨论循环处理和优化之前，我们必须了解 CFG 和支配关系概念的大概内容。CFG 是程序的控制流图，揭示程序如何在执行过程中走过各个基本块。根据支配信息，我们能够知道 CFG 中各个基本块之间的关系。

在 CFG 中，我们说节点 d 支配节点 n，当经过 n 的所有路径（从起点到终点）都必须经过 d。这表示为 d -> n。图 G = (V, E)称为支配者树（dominator tree），其中 V 是基本块集合，E 是在 V 之上支配边集合。

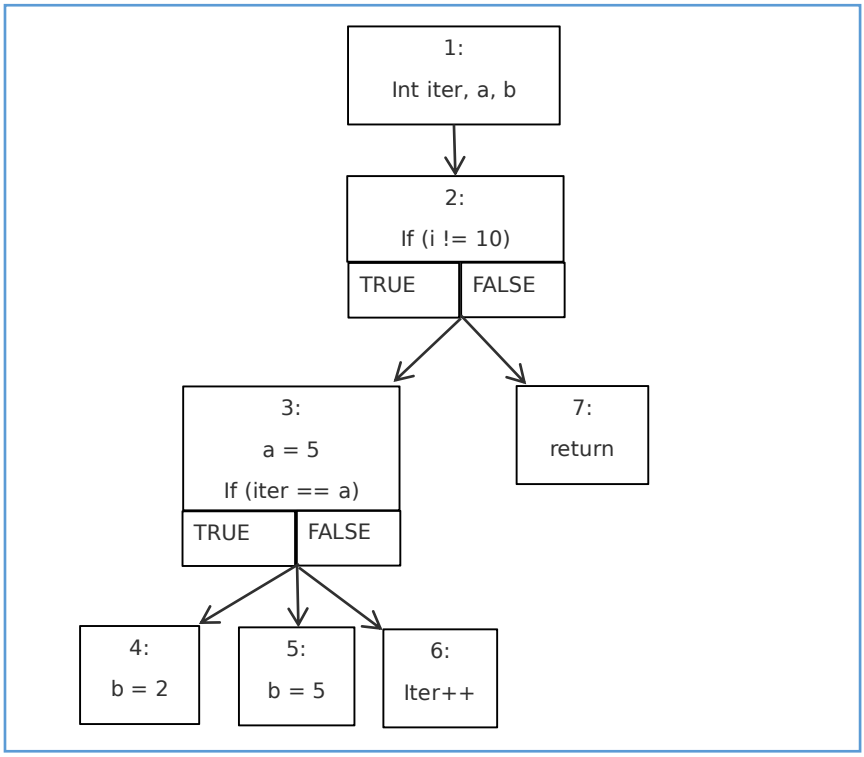
让我们举例说明程序的 CFG 和相应的支配者树。示例代码如下：

```
void fun() {
    int iter, a, b;
    for (iter = 0; iter < 10; iter++) {
        a = 5;
        if (iter == a)
            b = 2;
        else
            b = 5;
    }
}
```

上面的代码的 CFG 是这个样子：



根据支配关系和支配者树的知识，上面 CFG 的支配者树是这个样子：



第 1 个图是代码的 CFG，第 2 个图是这个 CFG 的支配者树。我们已对图中的基本块编号。在 CFG 中我们看到，2 支配 3，2 也支配 4，5，6。3 支配 4， 5， 6，而且 3 是它们的直接支配者。4 和 5 之间没有支配关系。6 不受 5 支配，因为有一条从 4 到 6 的路径。以相同的理由，6 也不受 4 支配。

LLVM 中的所有循环优化和转换都派生于 LoopPass 类，它的实现在文件 lib/Analysis/LoopPass.cpp 中。LPPassManager 负责处理所有的 LoopPass。

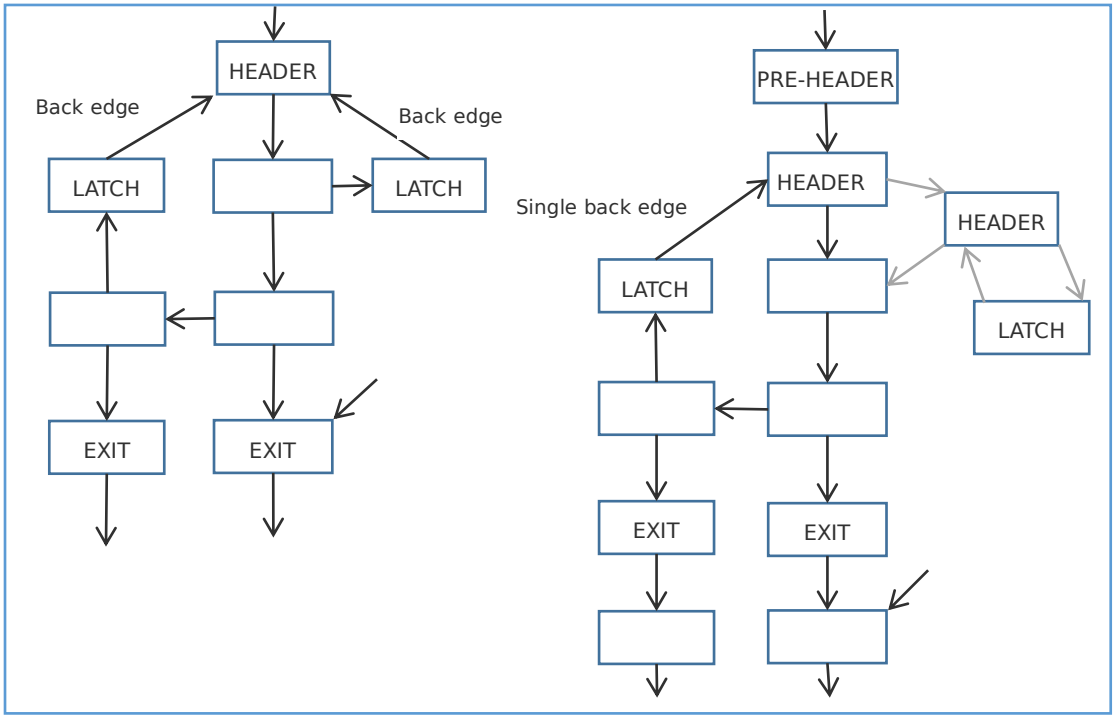
学习循环处理最适合从 LoopInfo 类开始。它用于识别程序中的自然循环，得出 CFG 中不同节点的深度。自然循环是 CFG 中的环结构。为了在 CFG 中定义自然循环，我们必须理解回边（backedge）：在 CFG 中一条终点支配起点的边。自然循环可以由回边 a -> d 定义，其中终点 d 是头节点（header node）。它定义了 CFG 的一个子图，包含所有不经过 d 可到达 a 的其它节点。

我们看到，在第 1 个图中，回边 6 -> 2 形成一个自然循环，包含节点 2，3，4，5 和

6。

下一个重要的步骤是简化循环，即将循环转换成一种规范的形式。为循环插入一个前置节点（preheader），使得有一条单独的边从循环外部到循环头节点。为循环插入退出节点，使得所有退出节点的前趋节点（predecessor）都来自循环内部。插入前置节点和退出节点有助于后续的循环优化，例如循环无关代码移动。

简化循环还保证一个循环只有一条回边。如果循环头节点的前趋节点多于两个（来自前置节点和多个循环回路），我们就得调整循环回路。一种方法是插入一个新的节点，所有回边都以它为目标，而这个新节点跳向循环头节点。让我们举例说明一个循环经过循环简化 Pass 后是什么样子。如下图所示，我们看到插入了一个前置节点，新建了退出节点，只有一条回边。



从 LoopInfo 获取所需的信息，把循环简化成一种规范的形式。接下来，我们将学习若干循环优化。

主要的循环优化 Pass 之一是循环不变量代码移动（LICM: loop invariant code

motion）。这个 Pass 尝试尽可能多地移出循环体中的代码。移出代码的条件是代码片段在循环中是不变的，也就是说，它的结果不依赖于循环的执行，每次循环迭代都保持不变。把不变代码移动到前置（preheader）基本块，或者移动到出口（exit）基本块。这个 Pass 的实现在文件 lib/Transforms/Scalar/ LICM.cpp 中。仔细看它的代码，我们看到，它需要先执行 LoopInfo 和 LoopSimplify Pass。还有，它需要 AliasAnalysis 的信息。这是为了用别名分析的结果从循环中移出循环不变的 load 和 call 指令。如果循环中的 load 和 call 不与任何 store 指令关联（alias），就能把它们移出循环。这也有助于内存标量提升。

让我们通过一个例子看看 LICM 是怎么做的。

```
$ cat licm.ll
define void @func(i32 %i) {
Entry:
  br label %Loop
Loop:
  %j = phi i32 [ 0, %Entry ], [ %Val, %Loop ]
  %loopinvar = mul i32 %i, 17
  %Val = add i32 %j, %loopinvar
  %cond = icmp eq i32 %Val, 0
  br i1 %cond, label %Exit, label %Loop
Exit:
  ret void
}
```

在上面的测试代码中，Loop 基本块表示一个循环，循环条件是 br i1 %cond, label %Exit, label %Loop（循环回路）。%j 是诱导变量，由 phi 指令引出。不言而喻，这个 phi 表示，当控制来自 Entry 基本块时，选择 0；当控制来自 Loop 基本块时，选择%Val。可以看出，这里的循环不变代码是%loopinvar = mul i32 %i, 17，因为%loopinvar 不依赖于循环迭代，它只依赖于函数的参数。因此，当我们运行 LICM Pass 时，我们预期这条指令将被提升（hoist）到循环之外，这样就避免每次循环迭代都执行这条指令。

让我们运行 LICM Pass，看看输出：

```
$ opt -licm licm.ll -o licm.bc
```

```
$ llvm-dis licm.bc -o licm_opt.ll
$ cat licm_opt.ll
; ModuleID = 'licm.bc'

define void @func(i32 %i) {
Entry:
  %loopinvar = mul i32 %i, 17
  br label %Loop

Loop:                                             ; preds = %Loop, %Entry
  %j = phi i32 [ 0, %Entry ], [ %Val, %Loop ]
  %Val = add i32 %j, %loopinvar
  %cond = icmp eq i32 %Val, 0
  br i1 %cond, label %Exit, label %Loop

Exit:                                             ; preds = %Loop
  ret void
}
```

我们看到，指令%loopinvar = mul i32 %i, 17 已经被提升到循环之外，这正是我们预期的结果。

LLVM 还有很多其它的循环优化，例如循环旋转（Loop Rotation），循环交换（Loop Interchange），循环不切换（Loop Unswitch）等。它们的源代码在 LLVM 目录 lib/Transforms/Scalar 中。为了更深地理解它们，我们需要阅读源代码。在下一章中，我们将学习标量进化。

标量进化

标量进化的意图，在于揭示一个标量在程序执行过程中如何变化。我们追踪一个特定的标量，看它是如何得到的，依赖于哪些元素，是否在编译时已知，将接受什么运算。我们需要分析一个基本块的代码，而不是单条指令。一个标量由两个元素建立，一个是变量，一个是常数次运算。变量元素的值在编译时是未知的，只有在运行时才能知道。另一个元素是常量部分。这些元素自己可能被递归地分解成其它元素，例如一个常量，一个未知量，或者一

个算术运算。

常量进化的主要概念，就是观察一个包含编译时未知元素的标量，分析它在程序执行过程中如何进化，寻找可优化之处。一个例子是，删除一个冗余的值，这个值的标量进化过程与同一程序中的别的值相似。

在 LLVM 中，我们可以用标量进化分析包含共同整数算术运算的代码。

在 LLVM 中，ScalarEvolution 类的实现在 include/llvm/Analysis 目录中，它是一个 LLVM Pass，可用于分析循环中的标量表达式。它能够识别普通的诱导变量（循环中的一个变量，其值是循环迭代次数的函数），并且把它们表达为 SCEV 类的对象。这个类用于表达程序中接受分析的表达式。利用这种分析，我们可以获得行程计数和其它分析结果。这种标量进化分析主要用于诱导标量替换和循环强度减弱。

下面我们来看一个例子，对它运行标量进化 Pass，看看它产生的输出。

编写一个测试程序 scalevl.ll，它有一个循环，循环中包含标量。

```
$ cat scalevl.ll
define void @fun() {
entry:
  br label %header
header:
  %i = phi i32 [ 1, %entry ], [ %i.next, %body ]
  %cond = icmp eq i32 %i, 10
  br i1 %cond, label %exit, label %body
body:
  %a = mul i32 %i, 5
  %b = or i32 %a, 1
  %i.next = add i32 %i, 1
  br label %header
exit:
  ret void
}
```

在这个测试程序中，循环体中的%a 和%b 是我们所关心的标量。对它运行标量进化 Pass，输出如下：

```
$ opt -analyze -scalar-evolution scalevl.ll
Printing analysis 'Scalar Evolution Analysis' for function 'fun':
Classifying expressions for: @fun
  %i = phi i32 [ 1, %entry ], [ %i.next, %body ]
  -->  {1,+,1}<%header> U: [1,11) S: [1,11)        Exits: 10
    LoopDispositions: { %header: Computable }
  %a = mul i32 %i, 5
  -->  {5,+,5}<%header> U: [5,51) S: [5,51)        Exits: 50
    LoopDispositions: { %header: Computable }
  %b = or i32 %a, 1
  -->  %b U: [1,0) S: full-set          Exits: 51          LoopDispositions:
{ %header: Variant }
  %i.next = add i32 %i, 1
  -->  {2,+,1}<%header> U: [2,12) S: [2,12)        Exits: 11
    LoopDispositions: { %header: Computable }
Determining loop execution counts for: @fun
Loop %header: backedge-taken count is 9
Loop %header: max backedge-taken count is 9
Loop %header: Predicated backedge-taken count is 9
```

我们看到，标量进化的输出显示一个特定变量的值的范围（U 代表无符号范围，S 代表有符号范围，此处两者相同）和当循环运行最后一次迭代时这个变量的值，即退出值。例如，%i 的值的范围是[1, 11)，也就是说，起始迭代值是 1，当%i 的值变成 11 时，条件%cond = icmp eq i32 %i, 10 变成假（false），循环跳出。因此，当循环退出时%i 的值是 10，在输出中表示为 Exits: 10。

在表达形式{x, +, y}中的值，例如{2, +, 1}，表示重复递增，也就是说，表达式的值在循环执行过程中变化，其中 x 表示第 0 此迭代时的基础值，y 表示每次后续迭代递增的值。

输出也显示了循环在第 1 次运行后迭代的次数。这里，它显示回边经过次数（backedge-taken count）是 9，也就是说，循环一共运行 10 次。最大回边经过次数（max backedge-taken）是永远不比回边经过次数小的最小值，这里是 9。

这是这个例子的输出，你可以试试其它的测试程序，看看它们的输出。

LLVM 本质函数

一个本质函数（intrinsic）是编译器内建的函数。编译器知道如何以最优化的方式实现它的功能，为特定的后端（backend）替换之为一组机器指令。经常地，这些机器指令内联地插入到代码中，避免函数调用的开销（在很多情况下，我们确实调用库函数。例如，对于 http://llvm.org/docs/LangRef.html#standard-c-library-intrinsics 列出的函数，我们调用 libc）。对于其它的编译器来说，它们也被称为内建函数（built-in）。

在 LLVM 中，这些本质函数是在 IR 层次的代码优化过程中产生的（程序中的本质函数可以由前端（frontend）直接生成）。这些函数的名字以前缀“llvm”开头，这是 LLVM 的保留字。这些函数总是外部的，使用者不能在代码中定义它们的函数体。在我们的代码中，我们只能调用这些本质函数。

在这个章节，我们不会深度探究细节。我们将举例说明 LLVM 如何用它自身的本质函数优化代码。

我们编写一段简单的代码：

```
$ cat intrinsic.cpp
int func() {
    int a[5];

    for (int i = 0; i != 5; ++i)
        a[i] = 0;

    return a[0];
}
```

下面我们用 Clang 生成 IR 文件。执行下面的命令，我们将得到 intrinsic.ll 文件，它是未优化的 IR，未使用任何本质函数。

```
$ clang -emit-llvm -S intrinsic.cpp
```

下面我们用 opt 工具 O1 优化级别优化 IR 文件。

```
$ opt -O1 intrinsic.ll -S -o -
; ModuleID = 'intrinsic.ll'
```

```
source_filename = "intrinsic.ll"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

; Function Attrs: norecurse nounwind readnone uwtable
define i32 @_Z4funcv() local_unnamed_addr #0 {
  %a = alloca [5 x i32], align 16
  %a4 = bitcast [5 x i32]* %a to i8*
  call void @llvm.memset.p0i8.i64(i8* nonnull %a4, i8 0, i64 20, i32 16, i1
false)
  %1 = getelementptr inbounds [5 x i32], [5 x i32]* %a, i64 0, i64 0
  %2 = load i32, i32* %1, align 16
  ret i32 %2
}

; Function Attrs: argmemonly nounwind
declare void @llvm.memset.p0i8.i64(i8* nocapture writeonly, i8, i64, i32, i1) #1

attributes #0 = { norecurse nounwind readnone uwtable
"disable-tail-calls"="false" "less-precise-fpmad"="false"
"no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf"
"no-infs-fp-math"="false" "no-nans-fp-math"="false"
"stack-protector-buffer-size"="8" "target-cpu"="x86-64"
"target-features"="+fxsr,+mmx,+sse,+sse2" "unsafe-fp-math"="false"
"use-soft-float"="false" }
attributes #1 = { argmemonly nounwind }

!llvm.ident = !{!0}

!0 = !{!"clang version 3.8.1-15 (tags/RELEASE_381/final)"}
```

这里值得注意的是，它调用了 LLVM 本质函数 llvm.memset.p0i8.i64 对数组填充 0，这是重要的优化。这个本质函数可能用于实现代码的向量化和并行化，以生成更优的代码。它可能调用 libc 库中最优的 memset 版本。它可能被完全省去，当没有代码引用这个数组时。

这个本质函数的第 1 个参数指定数组”a”，这是填充数值的目标数组。第 2 个参数指定被填充的值。第 3 个参数指定填充的字节数量。第 4 个参数指定目标的对齐方式。最后一个参数指定它是一个易变的操作（volatile，不可优化），或者不是。

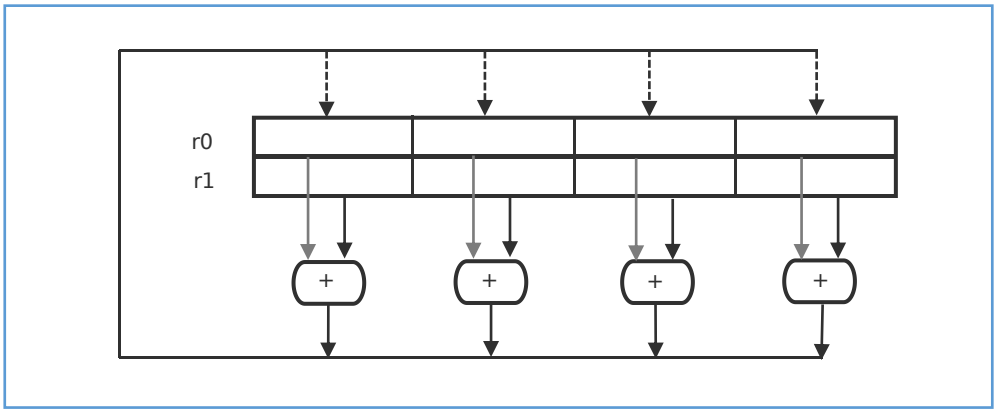
在 LLVM 中，这样的本质函数有很多，它们的列表见这个网页：http://llvm.org/docs/LangRef.html#intrinsic-functions。

向量化

向量化是编译器的一种重要的优化。我们用它向量化代码，使得指令一次执行能够操作多组数据。典型的高级目标架构都有向量寄存器集和向量指令，可以加载广泛的数据类型（以 128/256 位为典型）到向量寄存器，对它们执行运算，以一次标量运算的代价，同时操作两组、四组、甚至八组数据。

在 LLVM 中有两种向量化方法——超字层次并行化（SLP: Superword-Level Parallelism）和循环向量化。循环向量化处理循环中的向量化机会，而 SLP 方法向量化基本块中的线性代码。

向量指令执行单指令多数据（SIMD: Single-instruction multiple-data）操作，即对多路数据（并行地）执行相同操作。



让我们看看在 LLVM 基础设施中 SLP 向量化是如何实现的。

正如代码说明的那样，LLVM SLP 向量化的实现基于论文 *Loop-Aware SLP in GCC*，它的作者是 Ira Rosen，Dorit Nuzman，和 Ayal Zaks。LLVM SLP 向量化 Pass 采用自底向上的向量化方法。它探测可以被组合成向量存储的连续的存储，尝试将它们组合成向量

存储。然后，尝试用 use-def 链构建可向量树。如果找到这样的可获利的树，SLP 就向量化这棵树。

SLP 向量化经历三个阶段：

- 识别可向量化的模式
- 确定代码向量化后可获利
- 前面两个步骤成立时，对代码向量化

来看一个例子。考虑 3 个数组，各有 4 个元素，相加两个数组的元素，结果存到第 3 个数组。

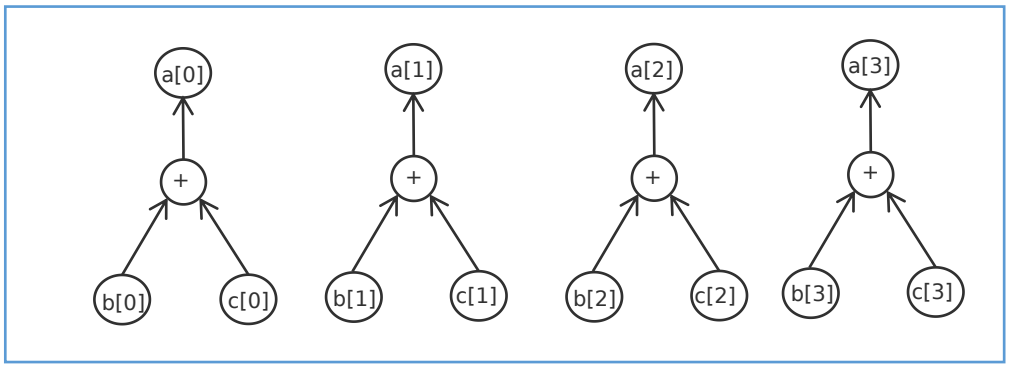
```
int a[4], b[4], c[4];

void addsub() {
    a[0] = b[0] + c[0];
    a[1] = b[1] + c[1];
    a[2] = b[2] + c[2];
    a[3] = b[3] + c[3];
}
```

这段程序的 IR 是这样的：

```
; ModuleID = 'addsub.cpp'
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-pc-linux-gnu"

@a = global [4 x i32] zeroinitializer, align 16
@b = global [4 x i32] zeroinitializer, align 16
@c = global [4 x i32] zeroinitializer, align 16

; Function Attrs: nounwind uwtable
define void @_Z6addsubv() #0 {
  %1 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @b, i64 0, i64 0), align 16
  %2 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @c, i64 0, i64 0), align 16
  %3 = add nsw i32 %1, %2
  store i32 %3, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @a, i64 0, i64 0), align 16
  %4 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @b, i64 0,
```

```
i64 1), align 4
  %5 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @c, i64 0, i64
1), align 4
  %6 = add nsw i32 %4, %5
  store i32 %6, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @a, i64 0, i64
1), align 4
  %7 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @b, i64 0,
i64 2), align 8
  %8 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @c, i64 0, i64
2), align 8
  %9 = add nsw i32 %7, %8
  store i32 %9, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @a, i64 0, i64
2), align 8
  %10 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @b, i64 0,
i64 3), align 4
  %11 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @c, i64 0,
i64 3), align 4
  %12 = add nsw i32 %10, %11
  store i32 %12, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @a, i64 0, i64
3), align 4
  ret void
}
```

上述程序模式的表达式树，可以可视化成 store 和 load 链：



对于上面的表达式树，自底向上的 SLP 向量化方法首先构建一个由 store 开始的链：

```
// Use the bottom up slp vectorizer to construct chains that start
// with store instructions.
BoUpSLP R(&F, SE, TTI, TLI, AA, LI, DT, AC);
```

然后扫描由上述代码构造的树，找到一个基本块中的所有 store：

```
// Scan the blocks in the function in post order.
for (auto BB: post_order(&F.getEntryBlock())) {
    // Vectorize trees that end at stores.
```

```
        if (unsigned count = collectStores(BB, R)) {
            (void)count;
            DEBUG(dbgs() << "SLP: Found" << count << " stores to
vectorize.\n");
            Changed |= vectorizeStoreChains(R);
        }
        // Vectorize trees that end at reductions.
        Changed |= vectorizeChainsInBlock(BB, R);
    }
```

函数 collectStores()收集所有的 store 引用。

```
unsigned
SLPVectorizer::collectStores(BasicBlock *BB, BoUpSLP &R) {
    unsigned count = 0;
    StoreRefs.clear();
    const DataLayout &DL = BB->getModule()->getDataLayout();
    for (Instruction &I : *BB) {
        StoreInst *SI = dyn_cast<StoreInst>(&I);
        if (!SI)
            continue;

        // Don't touch volatile stores.
        if (!SI->isSimple())
            continue;

        // Check that the pointer points to scalars.
        Type *Ty = SI->getValueOperand()->getType();
        if (!isValidElementType(Ty))
            continue;

        // Find the base pointer.
        Value *Ptr = GetUnderlyingObject(SI->getPointerOperand(), DL);

        // Save the store locations.
        StoreRefs[Ptr].push_back(SI);
        count++;
    }
    return count;
}
```

函数 SLPVectorizer::vectorizeStoreChains()依次执行三个步骤：

```
bool
SLPVectorizer::vectorizeStoreChain(ArrayRef<Value *> Chain, int
CostThreshold, BoUpSLP &R, unsigned VecRegSize) {
```

```
    ...
    ...
    R.buildTree(Operands);

    int Cost = R.getTreeCost();

    DEBUG(dbgs() << "SLP: Found cost=" << Cost << " for VF=" << VF <<
"\n");
    if (Cost < CostThreshold) {
        DEBUG(dbgs() << "SLP: Decided to vectorize cost=" << Cost << "\n");
        R.vectorizeTree();
    }
    ...
}
```

第 1 步是识别模式。之后函数 buildTree()递归地构造向量树，如前面的图示那样。

```
void
BoUpSLP::buildTree(ArrayRef<Value *> Roots, ArrayRef<Value *>
UserIgnoreLst) {
    ...
    ...
    buildTree_rec(Roots, 0);
    ...
    ...
}
```

对于我们给出的例子，它将识别出，所有的存储（store）操作都以加法（add）操作作为它们的操作数。

```
void
BoUpSLP::buildTree_rec(ArrayRef<Value *> VL, unsigned Depth) {
    ...
    ...
    case Instruction::Add:
    newTreeEntry(VL, true);
    DEBUG(dbgs() << "SLP: added a vector of bin op.\n");

    // Sort operands of the instructions so that each side is more
    // likely to have the same opcode
    if (isa<BinaryOperator>(VL0) && VL0->isCommutative()) {
        ValueList Left, Right;

        reorderInputsAccordingToOpcode(VL, Left, Right);
        buildTree_rec(Left, Depth + 1);
```

```
        buildTree_rec(Right, Depth + 1);
        return;
    }
    ...
    ...
}
```

当遇到加法（add）二元操作时，它再次递归地对加法操作的左右操作数构造树（调用相同的函数），这里左右两个操作数都是加载（load）操作：

```
    case Instruction::Load:
    // Check that a vectorized load would load the same memory as a
    // scalar load.
    // For example we don't want vectorize loads that are smaller than 8 bit.
    // Even though we have a packed struct {<i2, i2, i2, i2>} LLVM treats
    // loading/storing it as an i8 struct. If we vectorize loads/stores from
    // such a struct we read/write packed bits disagreeing with the
    // unvectorized version.
    const DataLayout &DL = F->getParent()->getDataLayout();
    Type *ScalarTy = VL[0]->getType();

    if (DL.getTypeSizeInBits(ScalarTy) != DL.getTypeAllocSizeInBits(ScalarTy)) {
        BS.cancelScheduling(VL);
        newTreeEntry(VL, false);
        DEBUG(dbgs() << "SLP: Gathering loads of non-packed type.\n");
        return;
    }
    // Check if the loads are consecutive or of we need to swizzle them.
    for (unsigned i = 0, e = VL.size() - 1; i < e; ++i) {
        LoadInst *L = cast<LoadInst>(VL[i]);
        if (!L->isSimple()) {
            BS.cancelScheduling(VL);
            newTreeEntry(VL, false);
            DEBUG(dbgs() << "SLP: Gathering non-simple loads.\n");
            return;
        }

        if (!isConsecutiveAccess(VL[i], VL[i + 1], DL)) {
            if (VL.size() == 2 && isConsecutiveAccess(VL[1], VL[0], DL)) {
                ++NumLoadsWantToChangeOrder;
            }
            BS.cancelScheduling(VL);
            newTreeEntry(VL, false);
            DEBUG(dbgs() << "SLP: Gathering non-consecutive loads.\n");
```

```
            return;
        }
        ++NumLoadsWantToKeepOrder;
        newTreeEntry(VL, true);
        DEBUG(dbgs() << "SLP: added a vector of loads.\n");
        return;
    }
```

当构造向量树时，有几个地方验证这棵树是否可向量化。例如，在前面的例子中，当遇到树中的 load 时，验证这些 load 是否连贯。在我们的表达式树中，整棵树中位于操作符左侧的 load 节点，b[0]，b[1]，b[2]和 b[3]，它们访问连续的内存地址。类似地，整棵树中位于操作符右侧的 load 节点，c[0]，c[1]，c[2]和 c[3]，它们也访问连续的内存地址。如果对于给出的操作任何一处验证失败，树构造动作将中止，代码不会被向量化。

在完成识别模式和构造向量树之后，下个步骤是估算向量化这棵树的成本。这实际上比较预期向量树的成本和当前标量树的成本。如果向量树的成本小于标量树，那么向量化这棵树是可获利的：

```
int BoUpSLP::getTreeCost() {
    int Cost = 0;
    DEBUG(dbgs() << "SLP: Calculating cost for tree of size " <<
VectorizableTree.size() << ".\n");

    // We only vectorize tiny trees if it is fully vectorizable.
    if (VectorizableTree.size() < 3 && !isFullyVectorizableTinyTree()) {
        if (VectorizableTree.empty()) {
            assert(!ExternalUses.size && "We should not have any external
users");
        }
        return INT_MAX;
    }

    unsigned BundleWidth = VectorizableTree[0].Scalars.size();

    for (unsigned i = 0, e = VectorizableTree.size(); i != e; ++i) {
        int C = getEntryCost(&VectorizableTree[i]);
        DEBUG(dbgs() << "SLP: Adding cost" << C << " for bundle that starts
with "
```

```
            << *VectorizableTree[i].Scalars[0] << ".\n");
        Cost += C;
    }

    SmallSet<Value *, 16> ExtractCostCalculated;
    int ExtractCost = 0;
    for (UserList::iterator I = ExternalUses.begin(), E = ExternalUses.end(); I !=
E; ++I) {
        // We only add extract cost once for the same scalar.
        if (!ExtractCostCalculated.insert(I->Scalar).second)
            continue;

        // Uses by ephemeral values are free (because the ephemeral value will
be
        // removed prior to code generation, and so the extraction will be
        // removed as well).
        if (EphValues.count(I->User))
            continue;

        VectorType *VecTy = VectorType::get(I->Scalar->getType(),
BundleWidth);
        ExtractCost += TTI->getVectorInstrCost(Instruction::ExtractElement,
VecTy, I->Lane);
    }

    Cost += getSpillCost();

    DEBUG(dbgs() << "SLP: Total Cost " << Cost + ExtractCost << ".\n");
    return Cost + ExtractCost;
}
```

这里值得关注的重要接口是 TargetTransformInfo(TTI)，它提供对代码生成接口的访问，这是 IR 层次转换所需要的。在我们的 SLP 向量化方法中，TTI 用于获得所建向量树的向量指令的成本：

```
int BoUpSLP::getEntryCost(TreeEntry *E) {
    ...
    ...
    case Instruction::Store: {
        // We know that we can merge the stores. Calculate the cost.
        int ScalarStCost = VecTy->getNumElements() *
            TTI->getMemoryOpCost(Instruction::Store, ScalarTy, 1, 0);
```

```
        int VecStCost = TTI->getMemoryOpCost(Instruction::Store, VecTy, 1, 0);
        return VecStCost - ScalarStCost;
    }
    ...
    ...
}
```

在同一函数中，也计算了向量加法的成本：

```
    case Instruction::Add: {
        // Calculate the cost of this instruction.
        int ScalarCost = 0;
        int VecCost = 0;
        if (Opcode == Instruction::FCmp ||
            Opcode == Instruction::ICmp ||
            Opcode == Instruction::Select) {
            VectorType *MaskTy = VectorType->get(Builder.getInt1Ty(),
VL.size());
            ScalarCost = VecTy->getNumElements() *
TTI->getCmpSelInstrCost(Opcode, VecTy, MaskTy);
        }
        else {
            // Certain instructions can be cheaper to vectorizer if they have
            // a constant second vector operand.
            TargetTransformInfo::OperandValueKind Op1VK =
TargetTransformInfo::OK_AnyValue;
            TargetTransformInfo::OperandValueKind Op2VK =
TargetTransformInfo::OK_UniformConstantValue;
            TargetTransformInfo::OperandValueProperties Op1VP =
TargetTransformInfo::OP_None;
            TargetTransformInfo::OperandValueProperties Op2VP =
TargetTransformInfo::OP_None;

            // If all operands are exactly the same ConstantInt then set the
            // operand kind to OK_UniformConstantValue.
            // If instead not all operands are constants, then set the operand
kind
            // to OK_AnyValue. If all operands are constants but not the
            // same, then set the operand kind to
OK_NonUniformConstantValue.
            ConstantInt *CInt = nullptr;
            for (unsigned i = 0; i < VL.size(); ++i) {
                const Instruction *I = cast<Instruction>(VL[i]);
                if (!isa<ConstantInt>(I->getOperand(1))) {
                    Op2VK = TargetTransformInfo::OK_AnyValue;
```

```
                    break;
                }
                if (i == 0) {
                    CInt = cast<ConstantInt>(I->getOperand(1));
                    continue;
                }
                if (Op2VK == TargetTransformInfo::OK_UniformConstantValue
&&
                    CInt != cast<ConstantInt>(I->getOperand(1)))
                    Op2VK                                                =
TargetTransformInfo::OK_NonUniformConstantValue;
                // FIXME: Currently cost of model modification for division by
                // power of 2 is handled only for X86. Add support for other
                // targets.
                if (Op2VK == TargetTransformInfo::OK_UniformConstantValue
&&
                    CInt &&
                    CInst->getValue().isPowerOf2())
                    Op2VK = TargetTransformInfo::OP_PowerOf2;

                ScalarCost = VecTy->getNumElements() *
                   TTI->getArithmeticInstrCost(Opcode, ScalarTy, Op1VK,
Op2VK, Op1VP, Op2VP);
                VecCost = TTI->getArithmeticInstrCost(Opcode, VecTy, Op1VK,
Op2VK, Op1VP, Op2VP);
                return VecCost - ScalarCost;
            }
        }
    }
```

在我们的例子中，整个表达式树的总成本得出为-12，这表明向量化它是可获利的。

最后，调用函数 vectorizeTree()对这棵树向量化：

```
Value *BoUpSLP::vectorizeTree() {
    ...
    ...
    vectorizeTree(&VectorizableTree[0]);
    ...
    ...
}
```

向量化 Pass 处理示例代码时经历了很多步骤。让我们看看有哪些步骤。注意，这需要调试版本的 opt 工具。

$ opt -S -basicaa -slp-vectorizer -mtriple=aarch64-unknown-linuxgnu -mcpu=cortex-a57 addsub.ll -debug

Features:+fxsr,+mmx,+sse,+sse2
CPU:cortex-a57

SLP: Analyzing blocks in addsub.
SLP: Found stores for 1 underlying objects.
SLP: Analyzing a store chain of length 4.
SLP: Analyzing a store chain of length 4
SLP: Analyzing 4 stores at offset 0
SLP: bundle: store i32 %3, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @a, i64 0, i64 0), align 16
SLP: initialize schedule region to store i32 %3, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @a, i64 0, i64 0), align 16
SLP: extend schedule region end to store i32 %6, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @a, i64 0, i64 1), align 4
SLP: extend schedule region end to store i32 %9, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @a, i64 0, i64 2), align 8
SLP: extend schedule region end to store i32 %12, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @a, i64 0, i64 3), align 4
SLP: try schedule bundle [store i32 %3, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @a, i64 0, i64 0), align 16; store i32 %6, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @a, i64 0, i64 1), align 4; store i32 %9, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @a, i64 0, i64 2), align 8; store i32 %12, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @a, i64 0, i64 3), align 4] in block
SLP: update deps of [store i32 %3, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @a, i64 0, i64 0), align 16; store i32 %6, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @a, i64 0, i64 1), align 4; store i32 %9, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @a, i64 0, i64 2), align 8; store i32 %12, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @a, i64 0, i64 3), align 4]
SLP: update deps of / store i32 %6, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @a, i64 0, i64 1), align 4
SLP: update deps of / store i32 %9, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @a, i64 0, i64 2), align 8
SLP: update deps of / store i32 %12, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @a, i64 0, i64 3), align 4
SLP: gets ready on update: store i32 %3, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @a, i64 0, i64 0), align 16
SLP: We are able to schedule this bundle.
SLP: added a vector of stores.
SLP: bundle: %3 = add nsw i32 %1, %2

SLP: extend schedule region start to %3 = add nsw i32 %1, %2
SLP: try schedule bundle [%3 = add nsw i32 %1, %2; %6 = add nsw i32 %4, %5; %9 = add nsw i32 %7, %8; %12 = add nsw i32 %10, %11] in block
SLP: update deps of [%3 = add nsw i32 %1, %2; %6 = add nsw i32 %4, %5; %9 = add nsw i32 %7, %8; %12 = add nsw i32 %10, %11]
SLP: update deps of / %6 = add nsw i32 %4, %5
SLP: update deps of / %9 = add nsw i32 %7, %8
SLP: update deps of / %12 = add nsw i32 %10, %11
SLP: schedule [store i32 %3, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @a, i64 0, i64 0), align 16; store i32 %6, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @a, i64 0, i64 1), align 4; store i32 %9, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @a, i64 0, i64 2), align 8; store i32 %12, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @a, i64 0, i64 3), align 4]
SLP: gets ready (def): [%3 = add nsw i32 %1, %2; %6 = add nsw i32 %4, %5; %9 = add nsw i32 %7, %8; %12 = add nsw i32 %10, %11]
SLP: We are able to schedule this bundle.
SLP: added a vector of bin op.
SLP: bundle: %1 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @b, i64 0, i64 0), align 16
SLP: extend schedule region start to %1 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @b, i64 0, i64 0), align 16
SLP: try schedule bundle [%1 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @b, i64 0, i64 0), align 16; %4 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @b, i64 0, i64 1), align 4; %7 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @b, i64 0, i64 2), align 8; %10 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @b, i64 0, i64 3), align 4] in block
SLP: update deps of [%1 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @b, i64 0, i64 0), align 16; %4 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @b, i64 0, i64 1), align 4; %7 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @b, i64 0, i64 2), align 8; %10 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @b, i64 0, i64 3), align 4]
SLP: update deps of / %4 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @b, i64 0, i64 1), align 4
SLP: update deps of / %7 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @b, i64 0, i64 2), align 8
SLP: update deps of / %10 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @b, i64 0, i64 3), align 4
SLP: schedule [%3 = add nsw i32 %1, %2; %6 = add nsw i32 %4, %5; %9 = add nsw i32 %7, %8; %12 = add nsw i32 %10, %11]
SLP: gets ready (def): [%1 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @b, i64 0, i64 0), align 16; %4 = load i32, i32* getelementptr

inbounds ([4 x i32], [4 x i32]* @b, i64 0, i64 1), align 4; %7 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @b, i64 0, i64 2), align 8; %10 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @b, i64 0, i64 3), align 4]
SLP: We are able to schedule this bundle.
SLP: added a vector of loads.
SLP: bundle: %2 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @c, i64 0, i64 0), align 16
SLP: try schedule bundle [%2 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @c, i64 0, i64 0), align 16; %5 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @c, i64 0, i64 1), align 4; %8 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @c, i64 0, i64 2), align 8; %11 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @c, i64 0, i64 3), align 4] in block
SLP: update deps of [%2 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @c, i64 0, i64 0), align 16; %5 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @c, i64 0, i64 1), align 4; %8 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @c, i64 0, i64 2), align 8; %11 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @c, i64 0, i64 3), align 4]
SLP: update deps of / %5 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @c, i64 0, i64 1), align 4
SLP: update deps of / %8 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @c, i64 0, i64 2), align 8
SLP: update deps of / %11 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @c, i64 0, i64 3), align 4
SLP: gets ready on update: %2 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @c, i64 0, i64 0), align 16
SLP: We are able to schedule this bundle.
SLP: added a vector of loads.
SLP: Checking user: store i32 %3, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @a, i64 0, i64 0), align 16.
SLP: Internal user will be removed: store i32 %3, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @a, i64 0, i64 0), align 16.
SLP: Checking user: store i32 %6, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @a, i64 0, i64 1), align 4.
SLP: Internal user will be removed: store i32 %6, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @a, i64 0, i64 1), align 4.
SLP: Checking user: store i32 %9, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @a, i64 0, i64 2), align 8.
SLP: Internal user will be removed: store i32 %9, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @a, i64 0, i64 2), align 8.
SLP: Checking user: store i32 %12, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @a, i64 0, i64 3), align 4.

SLP: Internal user will be removed: store i32 %12, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @a, i64 0, i64 3), align 4.
SLP: Checking user: %3 = add nsw i32 %1, %2.
SLP: Internal user will be removed: %3 = add nsw i32 %1, %2.
SLP: Checking user: %6 = add nsw i32 %4, %5.
SLP: Internal user will be removed: %6 = add nsw i32 %4, %5.
SLP: Checking user: %9 = add nsw i32 %7, %8.
SLP: Internal user will be removed: %9 = add nsw i32 %7, %8.
SLP: Checking user: %12 = add nsw i32 %10, %11.
SLP: Internal user will be removed: %12 = add nsw i32 %10, %11.
SLP: Checking user: %3 = add nsw i32 %1, %2.
SLP: Internal user will be removed: %3 = add nsw i32 %1, %2.
SLP: Checking user: %6 = add nsw i32 %4, %5.
SLP: Internal user will be removed: %6 = add nsw i32 %4, %5.
SLP: Checking user: %9 = add nsw i32 %7, %8.
SLP: Internal user will be removed: %9 = add nsw i32 %7, %8.
SLP: Checking user: %12 = add nsw i32 %10, %11.
SLP: Internal user will be removed: %12 = add nsw i32 %10, %11.
SLP: Calculating cost for tree of size 4.
SLP: Adding cost -3 for bundle that starts with store i32 %3, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @a, i64 0, i64 0), align 16.
SLP: Adding cost -3 for bundle that starts with %3 = add nsw i32 %1, %2.
SLP: Adding cost -3 for bundle that starts with %1 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @b, i64 0, i64 0), align 16.
SLP: Adding cost -3 for bundle that starts with %2 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @c, i64 0, i64 0), align 16.
SLP: #LV: 1 , Looking at %3 = add nsw i32 %1, %2
SLP: #LV: 2 , Looking at %1 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @b, i64 0, i64 0), align 16
SLP: #LV: 1 , Looking at %2 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @c, i64 0, i64 0), align 16
SLP: Spill Cost = 0.
SLP: Extract Cost = 0.
SLP: Total Cost = -12.
SLP: Found cost=-12 for VF=4
SLP: Decided to vectorize cost=-12
SLP: schedule block
SLP: initially in ready list: store i32 %3, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @a, i64 0, i64 0), align 16
SLP: schedule [store i32 %3, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @a, i64 0, i64 0), align 16; store i32 %6, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @a, i64 0, i64 1), align 4; store i32 %9, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @a, i64 0, i64 2), align 8; store i32 %12, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @a, i64 0, i64 3), align 4]

SLP: gets ready (def): [%3 = add nsw i32 %1, %2; %6 = add nsw i32 %4, %5; %9 = add nsw i32 %7, %8; %12 = add nsw i32 %10, %11]
SLP: schedule [%12 = add nsw i32 %1, %2; %11 = add nsw i32 %3, %4; %10 = add nsw i32 %5, %6; %9 = add nsw i32 %7, %8]
SLP: gets ready (def): [%1 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @b, i64 0, i64 0), align 16; %3 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @b, i64 0, i64 1), align 4; %5 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @b, i64 0, i64 2), align 8; %7 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @b, i64 0, i64 3), align 4]
SLP: gets ready (def): [%2 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @c, i64 0, i64 0), align 16; %4 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @c, i64 0, i64 1), align 4; %6 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @c, i64 0, i64 2), align 8; %8 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @c, i64 0, i64 3), align 4]
SLP: schedule [%8 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @c, i64 0, i64 0), align 16; %7 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @c, i64 0, i64 1), align 4; %6 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @c, i64 0, i64 2), align 8; %5 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @c, i64 0, i64 3), align 4]
SLP: schedule [%4 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @b, i64 0, i64 0), align 16; %3 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @b, i64 0, i64 1), align 4; %2 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @b, i64 0, i64 2), align 8; %1 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @b, i64 0, i64 3), align 4]
SLP: Extracting 0 values .
SLP: Erasing scalar: store i32 %15, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @a, i64 0, i64 0), align 16.
SLP: Erasing scalar: store i32 %14, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @a, i64 0, i64 1), align 4.
SLP: Erasing scalar: store i32 %13, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @a, i64 0, i64 2), align 8.
SLP: Erasing scalar: store i32 %11, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @a, i64 0, i64 3), align 4.
SLP: Erasing scalar: %15 = add nsw i32 %5, %10.
SLP: Erasing scalar: %14 = add nsw i32 %4, %9.
SLP: Erasing scalar: %13 = add nsw i32 %3, %8.
SLP: Erasing scalar: %11 = add nsw i32 %1, %6.
SLP: Erasing scalar: %5 = load i32, i32* getelementptr inbounds ([4 x i32], [4 x i32]* @b, i64 0, i64 0), align 16.
SLP: Erasing scalar: %4 = load i32, i32* getelementptr inbounds ([4 x i32],

```
[4 x i32]* @b, i64 0, i64 1), align 4.
SLP:      Erasing scalar:   %3 = load i32, i32* getelementptr inbounds ([4 x i32],
[4 x i32]* @b, i64 0, i64 2), align 8.
SLP:      Erasing scalar:   %1 = load i32, i32* getelementptr inbounds ([4 x i32],
[4 x i32]* @b, i64 0, i64 3), align 4.
SLP:      Erasing scalar:   %6 = load i32, i32* getelementptr inbounds ([4 x i32],
[4 x i32]* @c, i64 0, i64 0), align 16.
SLP:      Erasing scalar:   %5 = load i32, i32* getelementptr inbounds ([4 x i32],
[4 x i32]* @c, i64 0, i64 1), align 4.
SLP:      Erasing scalar:   %4 = load i32, i32* getelementptr inbounds ([4 x i32],
[4 x i32]* @c, i64 0, i64 2), align 8.
SLP:      Erasing scalar:   %2 = load i32, i32* getelementptr inbounds ([4 x i32],
[4 x i32]* @c, i64 0, i64 3), align 4.
SLP: Optimizing 0 gather sequences instructions.
SLP: vectorized "addsub"
```

最终的向量化输出是：

```
; ModuleID = 'addsub.ll'
source_filename = "addsub.ll"
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "aarch64-unknown-linuxgnu"

@a = global [4 x i32] zeroinitializer, align 16
@b = global [4 x i32] zeroinitializer, align 16
@c = global [4 x i32] zeroinitializer, align 16

; Function Attrs: nounwind uwtable
define void @addsub() #0 {
  %1 = load <4 x i32>, <4 x i32>* bitcast ([4 x i32]* @b to <4 x i32>*), align
16
  %2 = load <4 x i32>, <4 x i32>* bitcast ([4 x i32]* @c to <4 x i32>*), align
16
  %3 = add nsw <4 x i32> %1, %2
  store <4 x i32> %3, <4 x i32>* bitcast ([4 x i32]* @a to <4 x i32>*), align 16
  ret void
}
```

总结

在本章中，我们了解了基本块层次的优化，并以此结束了编译器中的优化器部分。我们举例说明了循环优化，标量进化，向量化，和 LLVM 本质函数。我们也见识了在 LLVM 中

SLP 向量化是如何处理的。然而，还有很多其它这样的优化，值得你去学习和掌握。

下一章我们将介绍中间表示如何被转换为有向无环图（Direct Acyclic Graph）。我们还将见识若干 selection DAG 层次的优化。

第 6 章 中间表示到 Selection DAG

到上一章为止，我们已经了解前端语言如何变换为 LLVM 中间表示。我们也看到了中间表示如何转换为更优化的代码。经过一系列的分析和转换 Pass，中间表示最终成为最优化的机器无关代码。然而，它依然是实际机器代码的抽象表示。编译器必须为目标架构生成可执行的机器代码。

LLVM 利用 DAG，即一种有向无环图表示，实现代码生成。它的思路是，将中间表示（IR）变换为 Selection DAG，经历一系列的阶段，包括 DAG 结合，合法化，指令选择，指令调度等，最后分配寄存器并输出机器代码。注意，寄存器分配和指令调度以交织的方式发生。

我们在本章中将介绍以下内容：

- 中间表示变换到 Selection DAG
- 合法化 Selection DAG
- 优化 Selection DAG
- 指令选择
- 调度和输出机器指令
- 寄存器分配
- 代码输出

中间表示变换到 Selection DAG

中间表示（IR）指令可以表示为 SDAG 节点。如此，整个指令序列形成一个相互连接的有向无环图，每个节点对应一条 IR 指令。

举例来说，考虑下面的 LLVM IR：

```
$ cat test.ll
define i32 @test(i32 %a, i32 %b, i32 %c) {
    %add = add nsw i32 %a, %b
    %div = sdiv i32 %add, %c
    ret i32 %div
}
```

LLVM 提供了接口 SelectionDAGBuilder，为 IR 指令创建相应的 DAG 节点。考虑二元操作：

```
%add = add nsw i32 %a, %b
```

处理这条指令时，将调用下面的函数：

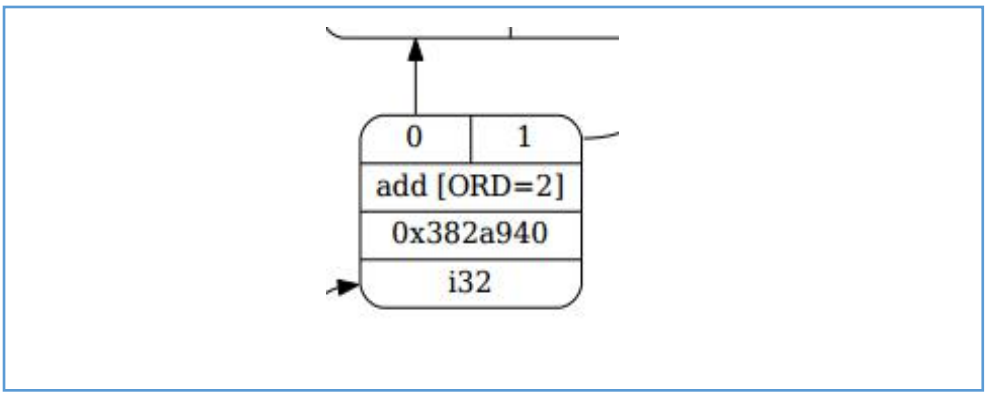
```
void SelectionDAGBuilder::visit(unsigned Opcode, const User &I) {
  // Note: this doesn't use InstVisitor, because it has to work with
  // ConstantExpr's in addition to instructions.
  switch (Opcode) {
  default: llvm_unreachable("Unknown instruction type encountered!");
    // Build the switch statement using the Instruction.def file.
#define HANDLE_INST(NUM, OPCODE, CLASS) \
    case Instruction::OPCODE: visit##OPCODE((const CLASS&)I); break;
#include "llvm/IR/Instruction.def"
  }
}
```

根据操作码(这里是 Add),将调用相应的访问函数。在这个例子中,将调用 visitAdd(),它进而调用 visitBinary()函数。visitBinary()函数如下：

```
void SelectionDAGBuilder::visitBinary(const User &I, unsigned OpCode) {
  SDValue Op1 = getValue(I.getOperand(0));
  SDValue Op2 = getValue(I.getOperand(1));

  bool nuw = false;
  bool nsw = false;
  bool exact = false;
  bool vec_redux = false;
  FastMathFlags FMF;

  if (const OverflowingBinaryOperator *OFBinOp =
          dyn_cast<const OverflowingBinaryOperator>(&I)) {
    nuw = OFBinOp->hasNoUnsignedWrap();
    nsw = OFBinOp->hasNoSignedWrap();
  }
  if (const PossiblyExactOperator *ExactOp =
```

```
            dyn_cast<const PossiblyExactOperator>(&I))
      exact = ExactOp->isExact();
    if (const FPMathOperator *FPOp = dyn_cast<const FPMathOperator>(&I))
      FMF = FPOp->getFastMathFlags();

    SDNodeFlags Flags;
    Flags.setExact(exact);
    Flags.setNoSignedWrap(nsw);
    Flags.setNoUnsignedWrap(nuw);
    Flags.setVectorReduction(vec_redux);
    if (EnableFMFInDAG) {
      Flags.setAllowReciprocal(FMF.allowReciprocal());
      Flags.setNoInfs(FMF.noInfs());
      Flags.setNoNaNs(FMF.noNaNs());
      Flags.setNoSignedZeros(FMF.noSignedZeros());
      Flags.setUnsafeAlgebra(FMF.unsafeAlgebra());
    }
    SDValue BinNodeValue = DAG.getNode(OpCode, getCurSDLoc(),
Op1.getValueType(),
                                       Op1, Op2, &Flags);
    setValue(&I, BinNodeValue);
}
```

这个函数获取二元操作的两个操作数，把它们存储到 SDValue 类型中。然后调用 DAG.getNode()函数，以二元操作的操作码为参数。这样就构造了一个 DAG 节点，它的样子如下图所示：



操作数 0 和操作数 1 是 load DAG 节点。

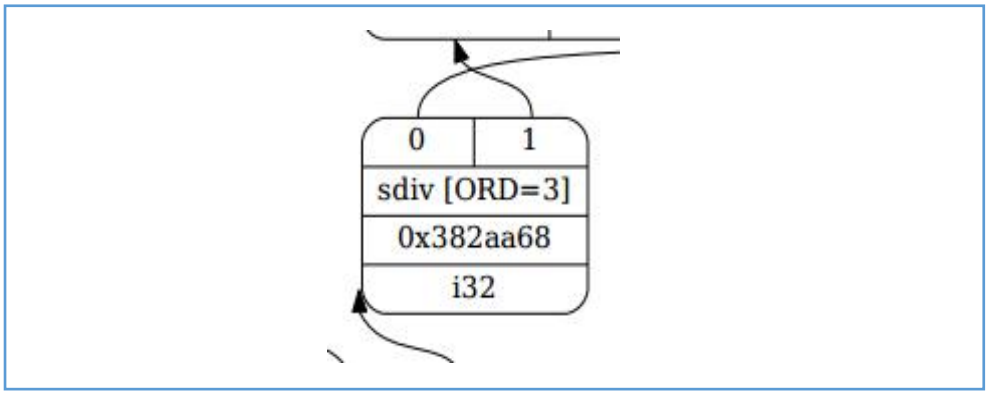
考虑这条指令：

```
%div = sdiv i32 %add, %c
```

遇到 sdiv 指令时，将调用 visitSDiv()函数。

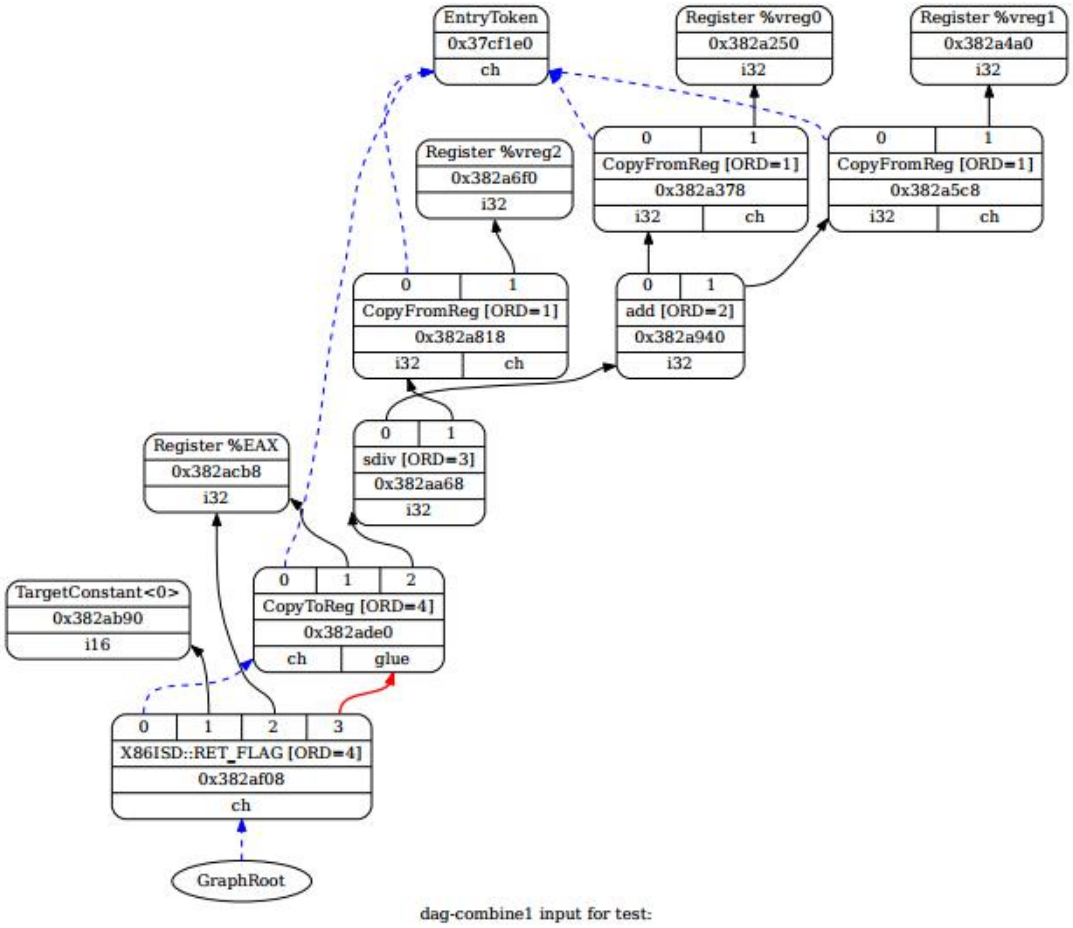
```
void SelectionDAGBuilder::visitSDiv(const User &I) {
  SDValue Op1 = getValue(I.getOperand(0));
  SDValue Op2 = getValue(I.getOperand(1));

  SDNodeFlags Flags;
  Flags.setExact(isa<PossiblyExactOperator>(&I) &&
                 cast<PossiblyExactOperator>(&I)->isExact());
  setValue(&I, DAG.getNode(ISD::SDIV, getCurSDLoc(), Op1.getValueType(),
Op1,
                           Op2, &Flags));
}
```

与 visitBinary()类似，这个函数也把两个操作数存储到 SDValue，以 ISD::SDIV 为操作符获取一个 DAG 节点。这个节点的样子如下图：



在中间表示中，其操作数 0 是%add。操作数 1 是%c，是传递给函数的一个参数，在中间表示变换到 SelectionDAG 时，它转换为一个 load 节点。visitLoad()函数用于构造 load DAG 节点。以上说到的访问函数，都在 lib/CodeGen/SelectionDAG/SelectionDAGBuilder.cpp 文件中。

访问了所有前面提到的 IR 指令之后，最终 IR 程序变换为 Selection DAG，如下图所示：



dag-combine1 input for test:

在上面的图中，注意这些图示：

- 黑色箭头表示数据流依赖（data flow dependency）
- 红色箭头表示粘合依赖（glue dependency）
- 蓝色点线箭头串联依赖（chain dependency）

粘合依赖防止两个节点在指令调度时被分离。串联依赖串联具有副作用（side effect）的节点。数据依赖表明一条指令依赖于前面指令的结果。

合法化 Selection DAG

在上一章中，我们看到中间表示如何变换为 Selection DAG。整个处理过程并未涉及任何目标架构的信息，而我们是在为它生成代码。对于给定的目标架构，一个 DAG 节点可

能是不合法的。例如，X86 架构不支持 sdiv 指令。作为替代，它支持 sdivrem 指令。通过 TargetLowering 接口，这种目标特定的信息被传送给 Selection DAG 结构。每个目标架构实现这个接口，描述 LLVM IR 应该如何低级化变换为合法的 Selection DAG 操作。

在我们的测试代码中，我们需要“展开”sdiv 指令为 sdivrem 指令。在函数 void SelectionDAGLegalize::LegalizeOp(SDNode *Node)中，处理这个特殊的节点将遇到 TargetLowering::Expand case，对它调用 ExpandNode()函数。

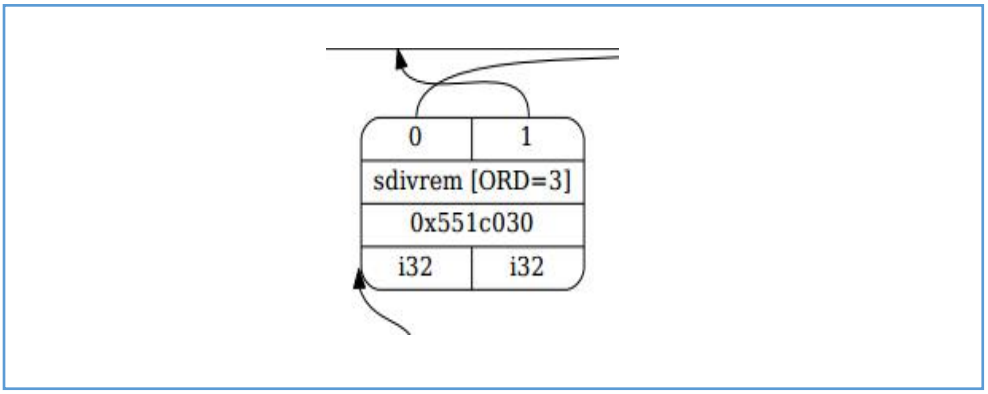
```
void SelectionDAGLegalize::LegalizeOp(SDNode *Node) {
    ...
    ...
    case TargetLowering::Expand:
      ExpandNode(Node);
      return;
    ...
    ...
}
```

这个函数展开 SDIV 到 SDIVREM 节点：

```
  case ISD::SDIV: {
    bool isSigned = Node->getOpcode() == ISD::SDIV;
    unsigned DivRemOpc = isSigned ? ISD::SDIVREM : ISD::UDIVREM;
    EVT VT = Node->getValueType(0);
    SDVTList VTs = DAG.getVTList(VT, VT);
    if (TLI.isOperationLegalOrCustom(DivRemOpc, VT) ||
        (isDivRemLibcallAvailable(Node, isSigned, TLI) &&
         useDivRem(Node, isSigned, true)))
      Tmp1 = DAG.getNode(DivRemOpc, dl, VTs, Node->getOperand(0),
                         Node->getOperand(1));
    else if (isSigned)
      Tmp1 = ExpandIntLibCall(Node, true,
                              RTLIB::SDIV_I8,
                              RTLIB::SDIV_I16, RTLIB::SDIV_I32,
                              RTLIB::SDIV_I64, RTLIB::SDIV_I128);
    else
      Tmp1 = ExpandIntLibCall(Node, false,
                              RTLIB::UDIV_I8,
                              RTLIB::UDIV_I16, RTLIB::UDIV_I32,
                              RTLIB::UDIV_I64, RTLIB::UDIV_I128);
    Results.push_back(Tmp1);
```

```
    break;
}
```

最后，经过合法化，这个节点变成 ISD::SDIVREM：



如此，上面的指令已经被合法化映射到目标架构支持的指令。这里我们看到的是展开合法化的例子。还有两类其它的合法化：提升和定制。前者提升一个类型到更高层的类型。后者处理目标特定的定义（可能是定制的操作，主要是中间表示本质函数）。它们都属于 CodeGen 编译阶段，留给读者自己去探索学习。

优化 Selection DAG

LLVM IR 变换为 Selection DAG 之后，可能出现很多优化 DAG 自身的机会。这些优化发生在 DAGCombiner 阶段。这些机会出现的原因在于架构特定的指令集。

来看一个例子：

```
#include <arm_neon.h>
unsigned hadd(uint32x4_t a) {
    return a[0] + a[1] + a[2] + a[3];
}
```

它的 LLVM IR 是这样的：

```
define i32 @hadd(<4 x i32> %a) nounwind {
    %vecext = extractelement <4 x i32> %a, i32 3
    %vecext1 = extractelement <4 x i32> %a, i32 2
    %add = add i32 %vecext, %vecext1
    %vecext2 = extractelement <4 x i32> %a, i32 1
    %add3 = add i32 %add, %vecext2
```

```
    %vecext4 = extractelement <4 x i32> %a, i32 0
    %add5 = add i32 %add3, %vecext4
    ret i32 %add5
}
```

这个例子简单地从一个<4 x i32>向量中提取单个元素，相加 4 个元素，得到一个标量结果。

像 ARM 这样的高级架构，有执行上述操作的单条指令，即单向量整体加。为了使用这条指令，需要在 Selection DAG 中识别前面的模式，将这些 DAG 结合为单个 DAG 节点。

这可以在 AArch64DAGToDAGISel 作指令选择时实现。

```
SDNode *AArch64DAGToDAGISel::Select(SDNode *Node) {
...
...
  case ISD::ADD:
    if (SDNode *I = SelectMLAV64LaneV128(Node))
      return I;
    if (SDNode *I = SelectADDV(Node))
      return I;
    break;
}
```

我们为 ADD 节点调用了函数 SelectADDV()。这是我们添加的函数，它的定义如下：

```
SDNode *AArch64DAGToDAGISel::SelectADDV(SDNode *N) {
  if (N->getValueType(0) != MVT::i32)
    return nullptr;
  SDValue SecondAdd;
  SDValue FirstExtr;
  if (!checkVectorElemAdd(N, SecondAdd, FirstExtr))
    return nullptr;

  SDValue Vector = FirstExtr.getOperand(0);
  if (Vector.getValueType() != MVT::v4i32)
    return nullptr;

  uint64_t LaneMask = 0;
  ConstantSDNode                    *LaneNode                    =
cast<ConstantSDNode>(FirstExtr->getOperand(1));
  LaneMask |= 1 << LaneNode->getZExtValue();

  SDValue ThirdAdd;
```

```
  SDValue SecondExtr;
  if (!checkVectorElemAdd(SecondAdd.getNode(), ThirdAdd, SecondExtr))
    return nullptr;
  if (Vector != SecondExtr.getOperand(0))
    return nullptr;
  ConstantSDNode *LaneNode2 = cast<ConstantSDNode>(SecondExtr->getOperand(1));
  LaneMask |= 1 << LaneNode2->getZExtValue();
  SDValue LHS = ThirdAdd.getOperand(0);
  SDValue RHS = ThirdAdd.getOperand(1);
  if (LHS.getOpcode() != ISD::EXTRACT_VECTOR_ELT ||
      RHS.getOpcode() != ISD::EXTRACT_VECTOR_ELT ||
      LHS.getOperand(0) != Vector ||
      RHS.getOperand(0) != Vector)
    return nullptr;
  ConstantSDNode *LaneNode3 = cast<ConstantSDNode>(LHS->getOperand(1));
  LaneMask |= 1 << LaneNode3->getZExtValue();
  ConstantSDNode *LaneNode4 = cast<ConstantSDNode>(RHS->getOperand(1));
  LaneMask |= 1 << LaneNode4->getZExtValue();
  if (LaneMask != 0x0F)
    return nullptr;
  return CurDAG->getMachineNode(AArch64::ADDVv4i32v, SDLoc(N), MVT::i32, Vector);
}
```

注意，我们早前定义了一个帮助函数 checkVectorElemAdd()，识别 Selection DAG 节点向量加法模式序列。

```
static bool checkVectorElemAdd(SDNode *N, SDValue &Add, SDValue &Extr) {
  SDValue Op0 = N->getOperand(0);
  SDValue Op1 = N->getOperand(1);
  const unsigned Opc0 = Op0->getOpcode();
  const unsigned Opc1 = Op1->getOpcode();

  const bool AddLeft = (Opc0 == ISD::ADD && Opc1 == ISD::EXTRACT_VECTOR_ELT);
  const bool AddRight = (Opc0 == ISD::EXTRACT_VECTOR_ELT && Opc1 == ISD::ADD);

  if (!(AddLeft || AddRight))
    return false;
```

```
  Add = AddLeft ? Op0 : Op1;
  Extr = AddLeft ? Op1 : Op0;
  return true;
}
```

让我们看看它是如何影响代码生成的。

```
$ llc -mtriple=aarch64-linux-gnu -verify-machineinstrs hadd.ll
```

这时没有用到我们定义的结合优化，最终生成的代码将是：

```
mov     w8, v0.s[3]
mov     w9, v0.s[2]
add     w8, w8, w9
mov     w9, v0.s[1]
add     w8, w8, w9
fmov    w9, s0
add     w0, w8, w9
ret
```

显然，这些代码是标量代码。加入我们定义的结合优化补丁，编译 LLVM，然后再次执行上面的命令，生成的代码将是：

```
addv s0, v0.4s
fmov w0, s0
ret
```

指令选择

至此，Selection DAG 在这个阶段已经被优化和合法化。然而，指令依然不是机器代码形式。这些指令自身需要在 Selection DAG 中被映射为架构特定的指令。TableGen 类帮助选择目标特定的指令。

CodeGenAndEmitDAG()函数调用 DoInstructionSelection()函数，它访问每个 DAG 节点，为它们调用 Select()函数。Select()函数是挂钩函数，指向特定实现，用以选择节点。它是虚函数，每个目标会实现它。

不妨假设我们的目标架构是 X86。X86DAGToDAGISel::Select()函数拦截一些节点，对它们作手动匹配，但是将大部分工作委托给 X86DAGToDAGISel::SelectCode()函数。

这个函数是由 TableGen 自动生成的。它包含匹配表，以这个表为参数，调用通用的

SelectionDAGISel:: SelectCodeCommon()函数。

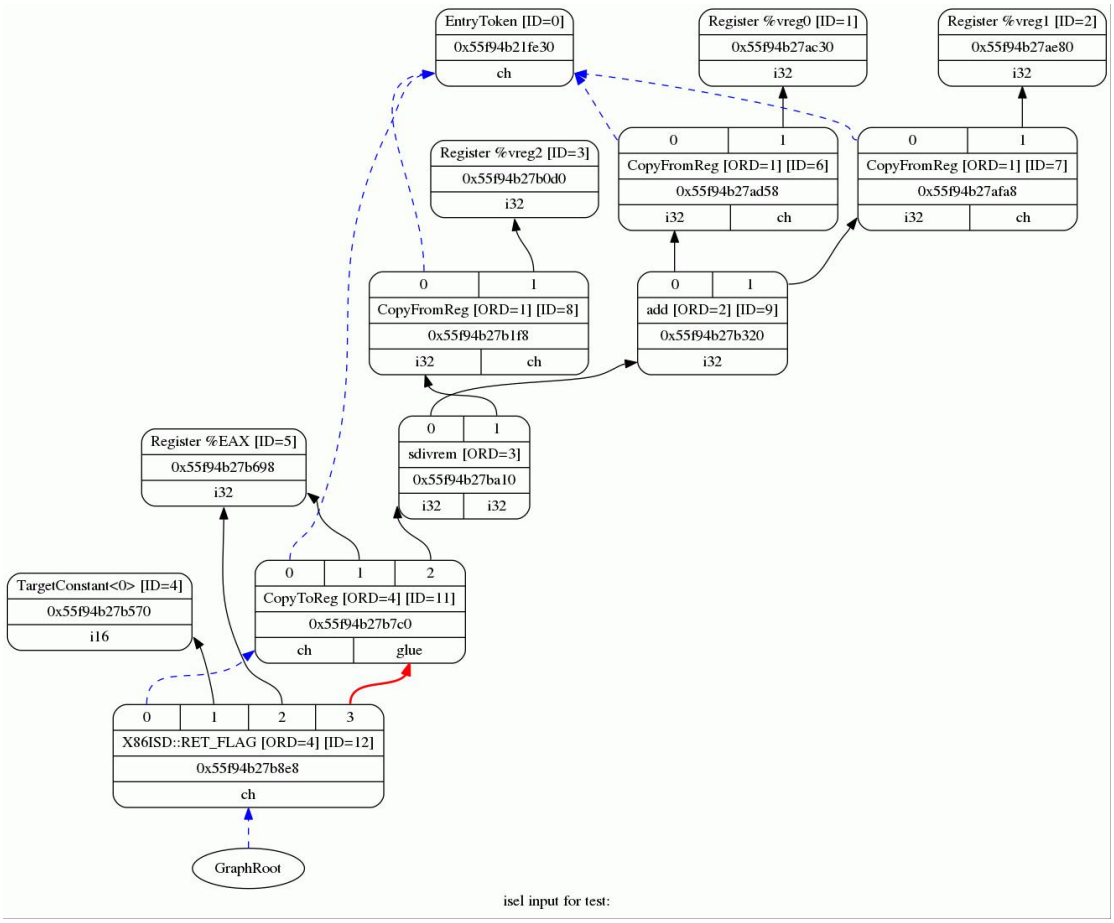
SDNode *ResNode = SelectCode(Node);

举例来说，考虑下面的程序：

```
$ cat test.ll
define i32 @test(i32 %a, i32 %b, i32 %c) {
  %add = add nsw i32 %a, %b
  %div = sdiv i32 %add, %c
  ret i32 %div
}
```

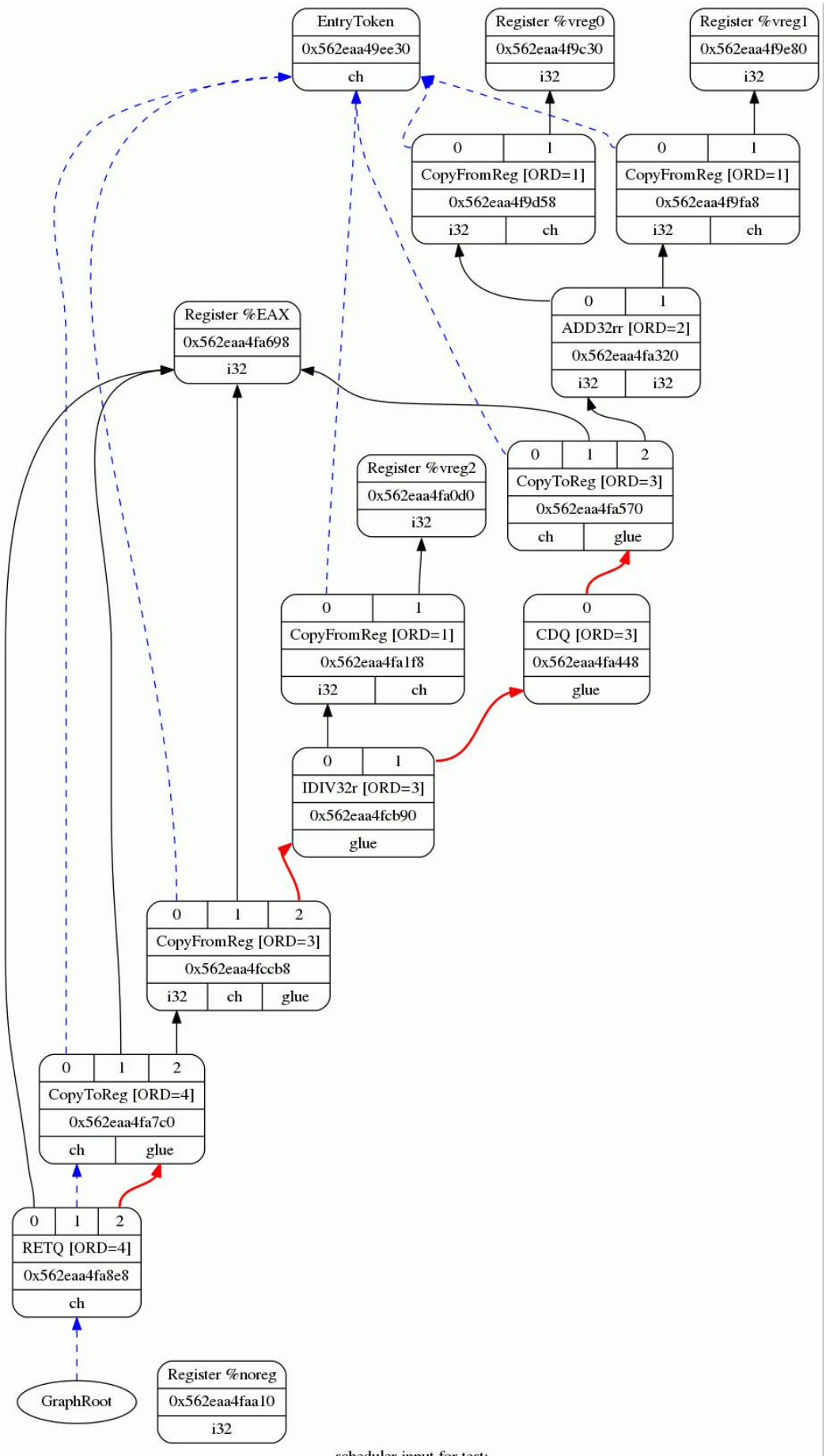
在指令选择之前，Selection DAG 是这样的：

```
$ llc -view-isel-dags test.ll
```



在指令选择之后，Selection DAG 是这样的：

```
$ llc -view-sched-dags test.ll
```



scheduler input for test:

调度和输出机器指令

至此，我们已经能够操作调整 DAG 了。下面，为了使程序能够在机器上运行，我们要将 DAG 变换为机器指令。首先一步是指令调度，即线性化 DAG，输出指令序列到 MachineBasicBlock。这是由 Scheduler 完成的。指令调度依赖于目标架构，因为目标有特定的挂钩函数，它们能够影响调度。

类函数 InstrEmitter::EmitMachineNode 以 SDNode *Node 为输入参数之一，为 Node 生成 MachineInstr 类的机器指令。这些指令被输出到 MachineBasicBlock。

它会调用 EmitSubregNode()函数以处理 EXTRACT_SUBREG，INSERT_SUBREG 和 SUBREG_TO_REG：

```
// Handle subreg insert/extract specially
if (Opc == TargetOpcode::EXTRACT_SUBREG ||
    Opc == TargetOpcode::INSERT_SUBREG ||
    Opc == TargetOpcode::SUBREG_TO_REG) {
  EmitSubregNode(Node, VRBaseMap, IsClone, IsCloned);
  return;
}
```

调用 EmitCopyToRegClassNode()和 EmitRegSequence()函数，以分别处理 COPY_TO_REGCLASS 和 REG_SEQUENCE：

```
// Handle COPY_TO_REGCLASS specially.
if (Opc == TargetOpcode::COPY_TO_REGCLASS) {
  EmitCopyToRegClassNode(Node, VRBaseMap);
  return;
}

// Handle REG_SEQUENCE specially.
if (Opc == TargetOpcode::REG_SEQUENCE) {
  EmitRegSequence(Node, VRBaseMap, IsClone, IsCloned);
  return;
}
```

调用 BuildMI 函数构造机器指令：

```
// Create the new machine instruction.
MachineInstrBuilder MIB = BuildMI(*MF, Node->getDebugLoc(), II);
```

为寄存器映射表添加新指令定义的结果寄存器：

```
// Add result register values for things that are defined by this
// instruction.
if (NumResults)
  CreateVirtualRegisters(Node, MIB, II, IsClone, IsCloned, VRBaseMap);
```

它用一个 for 循环生成指令的操作数：

```
for (unsigned i = NumSkip; i != NodeOperands; ++i)
  AddOperand(MIB, Node->getOperand(i), i-NumSkip+NumDefs, &II,
             VRBaseMap, /*IsDebug=*/false, IsClone, IsCloned);
```

将指令插入到 MachineBasicBlock 的特定位置：

```
// Insert the instruction into position in the block. This needs to
// happen before any custom inserter hook is called so that the
// hook knows where in the block to insert the replacement code.
MBB->insert(InsertPos, MIB);
```

下面的代码标记未使用的寄存器为死亡寄存器：

```
// Finally mark unused registers as dead.
if (!UsedRegs.empty() || II.getImplicitDefs())
  MIB->setPhysRegsDeadExcept(UsedRegs, *TRI);
```

正如我们早前讨论过的那样，目标特定的挂钩函数影响调度，这部分代码是这样的：

```
// Run post-isel target hook to adjust this instruction if needed.
if (II.hasPostISelHook())
  TLI->AdjustInstrPostInstrSelection(MIB, Node);
```

AdjustInstrPostInstrSelection()是一个虚函数，由目标实现。

我们用一个例子来看看在这一步中生成的机器指令。这需要给 llc 工具输入命令行选项 -print-machineinstrs。这是之前用过的测试程序：

```
$ cat test.ll
define i32 @test(i32 %a, i32 %b, i32 %c) {
  %add = add nsw i32 %a, %b
  %div = sdiv i32 %add, %c
  ret i32 %div
}
```

下面，运行 llc 工具，输入选项-print-machineinstrs。以 test.ll 为输入文件，存储输出到 outfile：

```
$ llc -print-machineinstrs test.ll > outfile 2>&1
```

这个 outfile 有点大，包含许多代码生成阶段，指令调度也是其中之一。我们关注的是 outfile 中”After Instruction Selection”之后的片段，内容如下：

```
# After Instruction Selection:
# Machine code for function test: SSA
Function Live Ins: %EDI in %vreg0, %ESI in %vreg1, %EDX in %vreg2

BB#0: derived from LLVM BB %0
    Live Ins: %EDI %ESI %EDX
    %vreg2<def> = COPY %EDX; GR32:%vreg2
    %vreg1<def> = COPY %ESI; GR32:%vreg1
    %vreg0<def> = COPY %EDI; GR32:%vreg0
    %vreg3<def,tied1> = ADD32rr %vreg0<tied0>, %vreg1, %EFLAGS<imp-def,dead>; GR32:%vreg3,%vreg0,%vreg1
    %EAX<def> = COPY %vreg3; GR32:%vreg3
    CDQ %EAX<imp-def>, %EDX<imp-def>, %EAX<imp-use>
    IDIV32r %vreg2, %EAX<imp-def>, %EDX<imp-def,dead>, %EFLAGS<imp-def,dead>, %EAX<imp-use>, %EDX<imp-use>; GR32:%vreg2
    %vreg4<def> = COPY %EAX; GR32:%vreg4
    %EAX<def> = COPY %vreg4; GR32:%vreg4
    RETQ %EAX

# End machine code for function test.
```

我们看到，有的地方用了物理寄存器，有的地方用了虚拟寄存器。它用到了机器指令 IDIV32r。下一节，我们将介绍这些代码中的虚拟寄存器如何被赋以物理寄存器。

寄存器分配

代码生成的第 2 步是寄存器分配。我们在前面的例子中看到，有的地方用了虚拟寄存器。寄存器分配的任务就是为这些虚拟寄存器分配物理寄存器。在 LLVM 中，虚拟寄存器的数量是无限的，而物理寄存器的数量是有限的，根据目标架构确定。因此，通过寄存器分配，我们的目的是将最多的物理寄存器分配给虚拟寄存器。我们必须明白物理寄存器数量有限，不总是能够为所有虚拟寄存器分配到物理寄存器。如果在某个点一个变量 V 分配不到可用的物理寄存器，我们就需要将一个已分配物理寄存器的变量移出到内存，释放这个物理寄存

器给变量 V 使用。这个过程，即将变量从物理寄存器移出到内存，称为 spilling。这要求找到一个合适的变量，把它从物理寄存器移出到内存。人们为此发明了多种算法。

寄存器分配的另一个重要的任务是解构 SSA 形式。至此在机器指令中的 Phi 指令需要被替换为常规指令。传统的做法是把它替换为复制指令。

必须注意的是，有些机器指令的操作数已经赋以了物理寄存器。这是因为目标架构要求某些指令使用固定的物理寄存器。寄存器分配器将处理其余的非固定寄存器，即虚拟寄存器。

寄存器分配映射虚拟寄存器到物理寄存器，它有两种方法：

- **直接映射：**它用到了 TargetRegisterInfo 类和 MachineOperand 类。这个方法需要在合适的位置插入 load 指令从内存读取变量，或者插入 store 指令存储变量到内存。
- **间接映射：**它用 VirtRegMap 类实现插入 load 和 store 指令。它同样需要从内存读取变量，或者存储变量到内存。它用 VirtRegMap::assignVirt2Phys(vreg, preg)函数来映射虚拟寄存器到物理寄存器。

LLVM 有四种寄存器分配技术。我们只简要介绍它们，不深入细节内容。这四种分配器如下：

- **基础寄存器分配器：**这是最基础的寄存器分配技术。它可以作为实现其它分配技术的起点。这个算法用挤出（spill）权重设置虚拟寄存器的优先级。权重最小的虚拟寄存器先得到寄存器分配。当没有物理寄存器可用时，虚拟寄存器被挤出（spill）到内存。
- **快速寄存器分配器：**这种分配方法逐次处理每个基本块。为了重用寄存器中的值，尽量保持它们在寄存器中更长时间。
- **PBQP 寄存器分配器：**正如这种分配方法的源代码（llvm/lib/CodeGen/RegAllocPBPQ.cpp）中提到的那样，它将寄存器分配表达为 PBPQ 问题，然后用 PBPQ 求解法解决它。

- **贪婪寄存器分配器：**这是 LLVM 中最高效的分配器，分配过程跨越函数。它分割变量的活跃范围（live range），最小化挤出（spill）代价。

让我们以 test.ll 为例子，看看寄存器分配器是如何将虚拟寄存器替换为实际物理寄存器的。我们选用贪婪寄存器分配器。你也可以选用其它的分配器。用到的目标机器是 x86-64。

```
$ llc test.ll -regalloc=greedy -o test1.s
$ cat test1.s
    .text
    .file "test.ll"
    .globl    test
    .align    16, 0x90
    .type    test,@function
test:                                   # @test
    .cfi_startproc
# BB#0:
    movl    %edx, %ecx
    leal (%rdi,%rsi), %eax
    cltd
    idivl    %ecx
    retq
.Lfunc_end0:
    .size    test, .Lfunc_end0-test
    .cfi_endproc

    .section ".note.GNU-stack","",@progbits
```

上面的代码中已经没有虚拟寄存器，它们被替换为了实际物理寄存器。这是 x86-64 机器。你可以试试 PBQP 分配器，看看分配的差别。leal(%rdi,%rsi), %eax 指令将被替换为下面的指令：

```
    movl   %esi, %edx
    movl   %edi, %eax
    leal    (%rax,%rdx), %eax
```

代码输出

第 1 节中我们从 LLVM 中间表示开始，将它变换为 Selection DAG，又变换为

MachineInstr。现在，我们需要输出代码。目前，我们可以使用 LLVM JIT 和 MC。JIT 是传统的在内存中即时为目标生成目标代码的方法。我们对 LLVM MC 层更感兴趣。

MC 层负责将从上一步传送过来的 MachineInstr 生成汇编文件或者目标文件。在 MC 层中，用 MCInst 表达指令，它是轻量的，因为它不像 MachineInstr 那样保存很多程序的信息。

代码输出从 AsmPrinter 类开始，它由目标特定的 AsmPrinter 类重载。这个类实现常规的低级化（lowering）过程。它利用目标特定的 MCInstLowering 接口（对于 X86 它是 X86MCInstLower 类，在 lib/Target/x86/X86MCInstLower.cpp 文件中），将 MachineFunction 函数变换为 MC 标签结构。

现在，我们已经得到 MCInst 指令，将它们输送给 MCStream 类进一步处理，以生成汇编文件或者目标代码。根据不同选择，MCStreamer 用它的子类 MCAsmStreamer 生成汇编代码，或者用它的子类 MCObjectStreamer 生成目标代码。

MCAsmStreamer 调用目标特定的 MCInstPrinter 打印汇编指令。MCObjectStreamer 使用 LLVM 目标代码汇编器生成二进制代码。这个汇编器进而调用 MCCodeEmitter::EncodeInstrction()生成二进制指令。

我们必须注意，MC 层是 LLVM 和 GCC 之间一个很大的不同之处。GCC 总是输出汇编代码，然后由一个外部的汇编器将它转换为目标文件。不同的是，LLVM 不仅易于用它自身的汇编器打印二进制指令，而且通过配置代理工具可以直接生成目标文件。这不仅保证生成的文本和二进制形式是一致的，而且相对 GCC 节省了时间，因为不必手工调用外部工具。

下面我们举一个例子，通过 llc 工具看看对应汇编的 MC 指令。我们使用前面用过的测试代码 test.ll。

为了查看 MC 指令，需要给 llc 输入命令行选项-asm-show-inst。MC 指令将以汇编

文件注释的形式给出。

```
$ llc test.ll -asm-show-inst -o -
    .text
    .file "test.ll"
    .globl    test
    .align    16, 0x90
    .type    test,@function
test:                                   # @test
    .cfi_startproc
# BB#0:
    movl    %edx, %ecx                # <MCInst #1660 MOV32rr
                                        #   <MCOperand Reg:22>
                                        #   <MCOperand Reg:24>>
    leal (%rdi,%rsi), %eax          # <MCInst #1268 LEA64_32r
                                        #   <MCOperand Reg:19>
                                        #   <MCOperand Reg:39>
                                        #   <MCOperand Imm:1>
                                        #   <MCOperand Reg:43>
                                        #   <MCOperand Imm:0>
                                        #   <MCOperand Reg:0>>
    cltd                                # <MCInst #386 CDQ>
    idivl    %ecx                     # <MCInst #899 IDIV32r
                                        #   <MCOperand Reg:22>>
    retq                                # <MCInst #2437 RETQ
                                        #   <MCOperand Reg:19>>
.Lfunc_end0:
    .size    test, .Lfunc_end0-test
    .cfi_endproc


    .section".note.GNU-stack","",@progbits
```

我们在汇编注释中看到 MCInst 和 MCOperand。通过给 llc 输入选项 -show-mc-encoding，我们也可以在汇编注释中看到二进制编码。

```
$ llc test.ll -show-mc-encoding -o -
    .text
    .file "test.ll"
    .globl    test
    .align    16, 0x90
    .type    test,@function
test:                                   # @test
    .cfi_startproc
# BB#0:
```

```
    movl    %edx, %ecx                  # encoding: [0x89,0xd1]
    leal (%rdi,%rsi), %eax          # encoding: [0x8d,0x04,0x37]
    cltd                                # encoding: [0x99]
    idivl   %ecx                        # encoding: [0xf7,0xf9]
    retq                                # encoding: [0xc3]
.Lfunc_end0:
    .size   test, .Lfunc_end0-test
    .cfi_endproc

    .section".note.GNU-stack","",@progbits
```

总结

在本章中，我们见识了 LLVM IR 是如何变换为 SelectionDAG 的。随后，SDAG 经历了多种转换。指令和数据类型被合法化。SelectionDAG 也经历了优化阶段，DAG 节点被结合为优化的节点。这可能是目标特定的。结合 DAG 之后，指令选择阶段将目标架构指令映射为 DAG 节点。然后，按线性顺序排列 DAG，使得 CPU 执行效率最高。这些 DAG 被变换为 MachineInstr，随后它们被销毁。然后，为代码中的所有虚拟寄存器分配物理寄存器。接着，MC 层出场，它用于生成目标代码和汇编代码。进入下一章，我们将学习如何定义目标；如何在 LLVM 中利用表描述文件和 TableGen 工具表示一个目标的各种属性。

第 7 章 为目标架构生成代码

由编译器生成的代码最终将在目标机器上运行。LLVM IR 的抽象形式帮助各种架构生成代码。目标机器是任意的，CPU，GPU，DSP 等都可以。目标机器给定了一些特性，例如寄存器集、指令集、函数调用惯例、指令流水线。这些特性或者属性由 tablegen 工具产生，这样方便机器代码生成程序使用它们。

LLVM 为后端设计了流水线结构，指令经历很多阶段转换，从 LLVM IR 到 SelectionDAG，然后到 MachineDAG，MachineInstr，最后到 MCInst。LLVM IR 变换为 SelectionDAG，SelectionDAG 经历合法化和优化。之后，DAG 节点经过指令选择被映射为目标指令。接着，DAG 经过指令调度，输出线性指令序列。然后，为虚拟寄存器分配目标机器寄存器，这要求优化寄存器分配方案，最小化内存 spill。

本章将介绍如何描述目标架构，如何输出汇编代码。

我们将学习以下内容：

- 定义寄存器和寄存器集
- 定义调用惯例
- 定义指令集
- 实现帧低层化
- 选择指令
- 打印指令
- 注册目标

后端示例

为了理解目标代码生成，我们定义一个简单的 RISC 架构机器 TOY，用最少的寄存器，

r0-r3，一个栈指针 SP，一个链接寄存器 LR（存储返回地址），和一个当前程序状态寄存器 CPSR。这个 TOY 后端的调用惯例类似于 ARM 类 thumb 架构——函数的参数存储在寄存器 r0-r1，返回值存储在 r0。

定义寄存器和寄存器集

寄存器集由 tablegen 工具定义。tablegen 帮助维护大量的域特定信息记录。它分解出这些记录的共同特征因子。这帮助减少重复描述，形成表示域信息的结构化方法。请访问页面 http://llvm.org/docs/TableGen 以详细理解 tablegen。TableGen 程序解释 TableGen 文件：llvm-tblgen。

在上个段落我们已经描述了我们的示例后端编译器，它有 4 个寄存器（r0-r3），一个栈寄存器（SP），和一个链接寄存器（LR）。这些可以在 TOYRegisterInfo.td 文件中指定。tablegen 函数提供 Register 类，它可扩展用于指定寄存器。创建一个新的文件，命名为 TOYRegisterInfo.td。

可以通过扩展 Register 类定义寄存器。

```
class TOYReg<bits<16> Enc, string n> : Register<n> {
  let HWEncoding = Enc;
  let Namespace = "TOY";
}
```

寄存器 r0-r3 属于常规用途 Register 类。可以通过扩展 RegisterClass 定义它们。

```
foreach i = 0-3 in {
  def R#i : R<i, "r"#i >;
}

def GRRegs : RegisterClass<"TOY", [i32], 32, (add R0, R1, R2, R3, SP)>;
```

其余的寄存器，SP，LR，和 CPSR，定义如下：

```
def SP : TOYReg<13, "sp">;
def LR : TOYReg<14, "lr">;
def CPSR : TOYReg<16, "cpsr">;
```

当上述代码放在一起时，TOYRegisterInfo.td 文件看起来是这样的：

```
class TOYReg<bits<16> Enc, string n> : Register<n> {
  let HWEncoding = Enc;
  let Namespace = "TOY";
}

foreach i = 0-3 in {
  def R#i : R<i, "r"#i >;
}

def SP : TOYReg<13, "sp">;
def LR : TOYReg<14, "lr">;
def CPSR : TOYReg<16, "cpsr">;
def GRRegs : RegisterClass<"TOY", [i32], 32, (add R0, R1, R2, R3, SP)>;
```

在 LLVM 目录 Target 中创建一个新的目录 TOY，把这个文件放在里面。如此，文件的路径是 llvm_root_directory/lib/Target/TOY/TOYRegisterInfo.td。

tablegen 工具 llvm-tablegen 处理这个.td 文件得到.inc 文件，一般地它包含这些寄存器的枚举值。这些枚举值可以被.cpp 文件使用，用于引用寄存器，例如 TOY::R0。

定义调用惯例

调用惯例指定如何将值传递给函数，如何从函数返回值。我们的 TOY 架构指定，用寄存器 r0 和 r1 传递两个参数，用栈传递其余的参数。指令选择阶段通过引用函数指针使用已定义的调用惯例。

当定义调用惯例时，我们需要定义两部分：返回值惯例和参数传递惯例。继承父类 CallingConv 以定义调用惯例。

在我们的 TOY 架构中，返回值存储在 r0 寄存器中。如果有更多的参数，整数值存储在栈单元中，每个单元 4 字节，4 字节对齐。这可以在 TOYCallingConv.td 中这样声明：

```
def RetCC_TOY : CallingConv<[
  CCIfType<[i32], CCAssignToReg<[R0]>>,
  CCIfType<[i32], CCAssignToStack<4, 4>>]>;
```

参数传递惯例可以这样定义：

```
def CC_TOY : CallingConv<[
  CCIfType<[i8, i16], CCPromoteToType<i32>>,
  CCIfType<[i32], CCAssignToReg<[R0, R1]>>,
  CCIfType<[i32], CCAssignToStack<4, 4>>]>;
```

上面的描述声明了如下三件事情：

- 如果参数的数据类型是 i8 或 i16，它将被提升为 i32；
- 头两个参数将被存储在寄存器 r0 和 r1 中；
- 如果有更多的参数，它们将被存储在栈（Stack）中。

我们也定义被调用者保存的寄存器，这些寄存器用于保存生命长久的值，这些值的作用域跨越函数调用。

```
def CC_Save : CalleeSavedRegs<(add R2, R3)>;
```

llvm-tablegen 工具处理.td 文件后生成一个 TOYCallingConv.inc 文件，它将被指令选择阶段的 TOYISelLowering.cpp 文件包含。

定义指令集

架构拥有丰富的指令集，表述目标机器支持的各种操作。典型地，表述指令的目标描述文件需要定义三个要素：

- 操作数
- 汇编字符串
- 指令模式

描述文件内容包含一个定义列表或输出列表，一个使用列表或输入列表。有不同的操作数类别，例如寄存器，立即数，和更复杂的（寄存器+立即数）操作数。

举例来说，我们在 TOYInstrInfo.td 中为 TOY 机器定义（寄存器+寄存器）操作如下：

```
def ADDrr : InstTOY<(outs GRRegs: $dst),
  (ins GRRegs:$src1, GRRegs:$src2),
```

```
  "add $dst, $src1, z$src2",
  [(set i32:$dst, (add i32:$src1, i32:$src2))]>;
```

在以上声明中，‘ins’有两个寄存器$src1 和$src2，它们属于常规用途寄存器类，用作操作数。操作的结果放在’outs’，它是$dst 寄存器，属于常规用途寄存器类。汇编字符串是”add $dst, $src1, z$src2”。$src1, $src2, $dst 的值将在分配寄存器时决定。如此，两个寄存器的 add 操作将被生成为如下的汇编：

```
add r0, r0, r1
```

以上我们看到如何利用 tablegen 表示一条简单的指令。类似于寄存器和寄存器 add 操作，寄存器和寄存器 subtract 操作也是如此定义。这留给读者自行尝试。详细的复杂指令的表示方法可以参考项目源代码中 ARM 或 X86 架构的指令描述文件。

实现帧低层化

帧低层化涉及输出函数的开场和结尾。开场出现在函数的开始，它设置被调用函数的栈帧。结尾出现在函数的最后，它复原调用函数（父）的栈帧。

栈（stack）在程序执行过程中有如下用途：

- 调用一个函数时记录返回地址
- 在函数调用上下文中存储局部变量
- 参数从调用者传递到被调用者

这样在实现帧低层化时需要定义两个主要的函数：emitPrologue() 和 emitEpilogue()。emitPrologue()定义如下：

```
void
TOYFrameLowering::emitPrologue(MachineFunction &MF) const {
  const TargetInstrInfo &TII = *MF.getSubtarget().getInstrInfo();
  MachineBasicBlock &MBB = MF.front();
  MachineBasicBlock::iterator MBBI = MBB.begin();

  uint64_t StackSize = computeStackSize(MF);
  if (!StackSize) {
```

```
    return;
  }
  unsigned StackReg = TOY::SP;
  unsigned OffsetReg = materializeOffset(MF, MBB, MBBI,
(unsigned)StackSize);
  if (OffsetReg) {
    BuildMI(MBB, MBBI, dl, TII.get(TOY::SUBrr), StackReg)
      .addReg(StackReg)
      .addReg(OffsetReg)
      .setMIFlag(MachineInstr::FrameSetup);
  } else {
    BuildMI(MBB, MBBI, dl, TII.get(TOY::SUBri), StackReg)
      .addReg(StackReg)
      .addImm(StackSize)
      .setMIFlag(MachineInstr::FrameSetup);
  }
}
```

上面的函数遍历 Machine Basic Block。它为函数计算栈长度和偏移，输出用栈寄存器设置帧的指令。

类似地，emitEpilogue()函数定义如下：

```
void
TOYFrameLowering::emitEpilogue(MachineFunction &MF, MachineBasicBlock
&MBB) const {
  const TargetInstrInfo &TII = *MF.getSubtarget().getInstrInfo();
  MachineBasicBlock::iterator MBBI = MBB.getLastNonDebugInstr();
  DebugLoc dl = MBBI->getDebugLoc();
  uint64_t StackSize = computeStackSize(MF);
  if (!StackSize) {
    return;
  }
  unsigned StackReg = TOY::SP;
  unsigned OffsetReg = materializeOffset(MF, MBB, MBBI,
(unsigned)StackSize);
  if (OffsetReg) {
    BuildMI(MBB, MBBI, dl, TII.get(TOY::ADDrr), StackReg)
      .addReg(StackReg)
      .addReg(OffsetReg)
      .setMIFlag(MachineInstr::FrameSetup);
  } else {
    BuildMI(MBB, MBBI, dl, TII.get(TOY::SUBri), StackReg)
      .addReg(StackReg)
```

```
        .addImm(StackSize)
        .setMIFlag(MachineInstr::FrameSetup);
  }
}
```

上面的函数同样计算栈长度，遍历 Machine Basic Block，为函数返回设置函数帧。注意，这里栈是下降的。

函数 emitPrologue()首先计算栈长度以决定是否需要开场。然后计算偏移以调整栈指针。对于 emitEpilogue()，它首先决定是否需要结尾。然后将栈指针复原为函数开始时的值。

举例来说，考虑以下输入 IR：

```
%p = alloca i32, align 4
store i32 2, i32* %p
%b = load i32* %p, align 4
%c = add nsw i32 %a, %b
```

生成的 TOY 汇编将是：

```
sub sp, sp, #4 ; prologue
movw r1, #2
str r1, [sp]
add r0, r0, #2
add sp, sp, #4 ; epilogue
```

低层化指令

在本章中，我们将学习实现三件事情：函数调用调用惯例，正式参数调用惯例，返回值调用惯例。我们创建一个文件 TOYISelLowering.cpp，用于实现指令低层化。

我们首先来看如何实现函数调用（call）调用惯例。

```
SDValue TOYTargetLowering::LowerCall(TargetLowering::CallLoweringInfo &CLI,
  SmallVectorImpl<SDValue> &InVals) const {
    SelectionDAG &DAG = CLI.DAG;
    SDLoc &Loc = CLI.DL;
    SmallVectorImpl<ISD::OutputArg> &Outs = CLI.Outs;
    SmallVectorImpl<SDValue> &OutVals = CLI.OutVals;
    SmallVectorImpl<ISD::InputArg> &Ins = CLI.Ins;
    SDValue Chain = CLI.Chain;
```

```
    SDValue Callee = CLI.Callee;
    CallingConv::ID CallConv = CLI.CallConv;
    const bool isVarArg = CLI.IsVarArg;

    CLI.IsTailCall = false;

    if (isVarArg) {
      llvm_unreachable("Unimplemented");
    }

    // Analyze operands of the call, assigning locations to each operand.
    SmallVector<CCValAssign, 16> ArgLocs;
    CCState CCInfo(CallConv, isVarArg, DAG.getMachineFunction(), ArgLocs,
*DAG.getContext());
    CCInfo.AnalyzeCallOperands(Outs, CC_TOY);

    // Get the size of the outgoing arguments stack space
    // requirement.
    const unsigned NumBytes = CCInfo.getNextStackOffset();

    Chain = DAG.getCALLSEQ_START(Chain, DAG.getIntPtrConstant(NumBytes,
Loc, true), Loc);

    SmallVector<std::pair<unsigned, SDValue>, 8> RegsToPass;
    SmallVector<SDValue, 8> MemOpChains;

    // Walk the register/memloc assignments, inserting copies/loads.
    for (unsigned i = 0, e = ArgLocs.size(); i != e; ++i) {
      CCValAssign &VA = ArgLocs[i];
      SDValue Arg = OutVals[i];

      // We only handle fully promoted arguments.
      assert(VA.getLocInfo() == CCValAssign::Full && "Unhandled loc info");

      if (VA.isRegLoc()) {
        RegsToPass.push_back(std::make_pair(VA.getLocReg(), Arg));
        continue;
      }

      assert(VA.isMemLoc() && "Only support passing arguments through
registers or via the stack");
      SDValue StackPtr = DAG.getRegister(TOY::SP, MVT::i32);
      SDValue PtrOff = DAG.getIntPtrConstant(VA.getLocMemOffset(), Loc);
      PtrOff = DAG.getNode(ISD::ADD, Loc, MVT::i32, StackPtr, PtrOff);
```

```
      MemOpChains.push_back(DAG.getStore(Chain, Loc, Arg, PtrOff,
MachinePointerInfo(), false, false, 0));
    }

    // Emit all stores, make sure they occur before the call.
    if (!MemOpChains.empty()) {
      Chain = DAG.getNode(ISD::TokenFactor, Loc, MVT::Other,
MemOpChains);
    }

    // Build a sequence of copy-to-reg nodes chained together with
    // token chain
    // and flag operands which copy the outgoing args into the
    // appropriate regs.
    SDValue InFlag;
    for (auto &Reg : RegsToPass) {
      Chain = DAG.getCopyToReg(Chain, Loc, Reg.first, Reg.second, InFlag);
      InFlag = Chain.getValue(1);
    }

    // We only support calling global addresses.
    GlobalAddressSDNode *G = dyn_cast<GlobalAddressSDNode>(Callee);
    assert(G && "We only support the calling of global addresses");
    EVT PtrVT = getPointerTy(DAG.getDataLayout());
    Callee = DAG.getGlobalAddress(G->getGlobal(), Loc, PtrVT, 0);

    std::vector<SDValue> Ops;
    Ops.push_back(Chain);
    Ops.push_back(Callee);

    // Add argument registers to the end of the list so that they
    // are known live into the call.
    for (auto &Reg : RegsToPass) {
      Ops.push_back(DAG.getRegister(Reg.first, Reg.second.getValueType()));
    }

    // Add a register mask operand representing the call-preserved
    // registers.
    const uint32_t *Mask;
    const TargetRegisterInfo *TRI = DAG.getSubtarget().getRegisterInfo();
    Mask = TRI->getCallPreservedMask(DAG.getMachineFunction(), CallConv);
    assert(Mask && "Missing call preserved mask for calling convertion");
    Ops.push_back(DAG.getRegisterMask(Mask));
```

```
    if (InFlag.getNode()) {
      Ops.push_back(InFlag);
    }
    SDVTList NodeTys = DAG.getVTList(MVT::Other, MVT::Glue);

    // Return a chain and a flag for retval copy to use.
    Chain = DAG.getNode(TOYISD::CALL, Loc, NodeTys, Ops);
    InFlag = Chain.getValue(1);
    Chain = DAG.getCALLSEQ_END(Chain,
      DAG.getIntPtrConstant(NumBytes, Loc, true),
      DAG.getIntPtrConstant(0, Loc, true),
      InFlag, Loc);
    if (!Ins.empty()) {
      InFlag = Chain.getValue(1);
    }

    // Handle result values, copying them out of physregs into vregs
    // that we return.
    return LowerCallResult(Chain, InFlag, CallConv, isVarArg, Ins, Loc, DAG,
InVals);
  }
```

在上面的函数中，我们首先分析调用指令的操作数，给它们分配位置，计算参数栈空间的长度。然后检查每个操作数，分别为寄存器操作数和内存操作数插入复制和加载指令。对于我们的示例目标，我们支持通过寄存器或者栈传递参数（记得上一节定义的调用惯例）。然后我们输出所有的加载指令，确保它们处于调用指令之前。我们构造了 copy-to-reg 节点序列，它们将输出的参数复制到正确的寄存器中。然后，添加一个寄存器掩码操作数，用于表示调用者保存的寄存器。最后，我们返回一个串联序列和一个标记用于复制返回值，将返回值从物理寄存器（physregs）复制到返回的虚拟寄存器（vregs）。

下面我们看看正式参数调用惯例的实现。

```
SDValue TOYTargetLowering::LowerFormalArguments(SDValue Chain,
    CallingConv::ID CallConv, bool isVarArg,
    const SmallVectorImpl<ISD::InputArg> &Ins,
    SDLoc dl, SelectionDAG &DAG,
    SmallVectorImpl<SDValue> &InVals) const {
      MachineFunction &MF = DAG.getMachineFunction();
```

```
    MachineRegisterInfo &RegInfo = MF.getRegInfo();
    assert(!isVarArg && "VarArg not supported");

    // Assign locations to all of the incoming arguments.
    SmallVector<CCValAssign, 16> ArgLocs;
    CCState CCInfo(CallConv, isVarArg, DAG.getMachineFunction(), ArgLocs,
*DAG.getContext());
    CCInfo.AnalyzeFormalArguments(Ins, CC_TOY);

    for (auto &VA :: ArgLocs) {
      if (VA.isRegLoc()) {
        // Arguments passed in registers
        EVT RegVT = VA.getLocVT();
        asseert(RegVT.getSimpleVT().SimpleTy == MVT::i32 && "Only
support MVT::i32 register passing");
        const unsigned VReg =
RegInfo.createVirtualRegister(&TOY::GRRegsRegClass);
        RegInfo.addLiveIn(VA.getLocReg(), VReg);
        SDValue ArgIn = DAG.getCopyFromReg(Chain, dl, VReg, RegVT);
        InVals.push_back(ArgIn);
        continue;
      }
      assert(VA.isMemLoc() && "Can only pass arguments as either registers
or via the stack");

      const unsigned Offset = VA.getLocMemOffset();
      const int FI = MF.getFrameInfo()->CreateFixedObject(4, Offset, true);
      EVT PtrTy = getPointerTy(DAG.getDataLayout());
      SDValue FIPtr = DAG.getFrameIndex(FI, PtrTy);
      assert(VA.getValVT() == MVT::i32 && "Only support passing arguments
as i32");
      SDValue Load = DAG.getLoad(VA.getValVT(), dl, Chain, FIPtr,
        MachinePointerInfo(), false, false, false, 0);
      InVals.push_back(Load);
    }
    return Chain;
  }
```

在上面的正式参数调用惯例实现中，我们为每个输入参数分配一个位置。我们只支持通过寄存器或者栈传递参数。下面我们来看返回值调用惯例的实现。

```
  SDValue TOYTargetLowering::LowerReturn(SDValue Chain,
                                         CallingConv::ID CallConv, bool
isVarArg,
```

```
                                              const
SmallVectorImpl<ISD::OutputArg> &Outs,
                                              const      SmallVectorImpl<SDValue>
&OutVals,
                                              SDLoc dl, SelectionDAG &DAG) const
{
    if (isVarArg) {
      report_fatal_error("VarArg not supported");
    }

    // CCValAssign - represent the assignment of
    // the return value to a location
    SmallVector<CCValAssign, 16> RVLocs;
    // CCState - Info about the registers and stack slot.
    CCState CCInfo(CallConv, isVarArg, DAG.getMachineFunction(), RVLocs,
*DAG.getContext());
    CCInfo.AnalyzeReturn(Outs, RetCC_TOY);
    SDValue Flag;
    SmallVector<SDValue, 4> RetOps(1, Chain);

    // Copy the result values into the output registers.
    for (unsigned i = 0, e = RVLocs.size(); i < e; ++i) {
      CCValAssign &VA = RVLocs[i];
      assert(VA.isRegLoc() && "Can only return in register!");

      Chain = DAG.getCopyToReg(Chain, dl, VA.getLocReg(), OutVals[i], Flag);
      Flag = Chain.getValue(1);
      RetOps.push_back(DAG.getRegister(VA.getLocReg(), VA.getLocVT()));
    }

    RetOps[0] = Chain; // Update chain.
    // Add the flag if we have it.
    if (Flag.getNode()) {
      RetOps.push_back(Flag);
    }

    return DAG.getNode(TOYISD::RET_FLAG, dl, MVT::Other, RetOps);
  }
```

首先我们检查能否低层化一个返回。然后收集寄存器和栈位置的信息。我们将返回值复制到输出寄存器中，最后我们为返回值返回一个 DAG 节点。

打印指令

打印汇编指令是生成目标代码的一个重要步骤。为了打印指令，我们定义各种类作为输出指令流的通道。

首先，在文件 TOYInstrFormats.td 中，初始化指令类，赋值操作数、汇编字符串、模式、输出变量等：

```
class InstTOY<dag outs, dag ins, string asmstr, list<dag> pattern>
    : Instruction {
  field bits<32> Inst;
  let Namespace = "TOY";
  dag OutOperandList = outs;
  dag InOperandList = ins;
  let AsmString = asmstr;
  let Pattern = pattern;
  let Size = 4;
}
```

然后，在文件 TOYInstPrinter.cpp 中定义打印操作数的函数。

```
void
TOYInstPrinter::printOperand(const MCInst *MI, unsigned opNo, raw_ostream
&O) {
  const MCOperand &Op = MI->getOperand(OpNo);
  if (Op.isReg()) {
    printRegName(O, Op.getReg());
    return;
  }
  if (Op.isImm()) {
    O << "#" << Op.getImm();
    return;
  }
  assert(Op.isExpr() && "unknown operand kind in printOperand");
  printExpr(Op.getExpr(), O);
}
```

这个函数简单地打印操作数，它可能是寄存器或者立即数。

在同一文件中，我们还定义了打印寄存器名字的函数：

```
void
TOYInstPrinter::printRegName(raw_ostream &OS, unsigned RegNo) const {
  OS << StringRef(getRegisterName(RegNo)).lower();
```

```
}
```

接着，我们定义打印指令的函数：

```
void
TOYInstPrinter::printInst(const MCInst *MI, raw_ostream &O, StringRef Annot) {
  printInstruction(MI, O);
  printAnnotation(O, Annot);
}
```

接着，我们声明和定义汇编信息如下：

创建文件 TOYMCAsmInfo.h，声明一个 ASMInfo 类：

```
#ifndef TOYTARGETASMINFO_H
#define TOYTARGETASMINFO_H
#include "llvm/MC/MCAsmInfoELF.h"
namespace llvm {
class StringRef;
class Target;
class TOYMCAsmInfo : public MCAsmInfoELF {
  virtual void anchor();

public:
  explicit TOYMCAsmInfo(StringRef TT);
}
} // namespace llvm
#endif
```

在文件 TOYMCAsmInfo.cpp 中定义类构造函数：

```
#include "TOYMCAsmInfo.h"
#include "llvm/ADT/StringRef.h"
using namespace llvm;
void TOYMCAsmInfo::anchor() {}
TOYMCAsmInfo::TOYMCAsmInfo(StringRef TT) {
  SupportsDebugInformation = true;
  Data16bitsDirective = "\t.short\t";
  Data32bitsDirective = "\t.long\t";
  Data64bitsDirective = 0;
  ZeroDirective = "\t.space\t";
  CommentString = "#";
  AscizDirective = ".asciiz";
  HiddenVisibilityAttr = MCSA_Invalid;
  ProtectedVisibilityAttr = MCSA_Invalid;
}
```

为了编译，我们定义 LLVMBuild.txt 如下：

```
[component_0]
type = Library
name = TOYAsmPrinter
parent = TOY
required_libraries = MC Support
add_to_library_groups = TOY
```

此外，定义 CMakeLists.txt 如下：

```
add_llvm_library(LLVMTOYAsmPrinter TOYInstPrinter.cpp)
```

当最后 TOY 编译器能够工作时，llc 工具（一个静态编译器）会生成 TOY 架构的汇编（向 llc 工具注册 TOY 架构之后）。

按照以下步骤，我们向静态编译器 llc 注册我们的 TOY 架构：

1. 首先，在文件 llvm_root_dir/CMakeLists.txt 中登记 TOY 后端：

```
set(LLVM_ALL_TARGETS
AArch64
ARM
...
...
TOY
)
```

2. 然后，在文件 llvm_root_dir/include/llvm/ADT/Triple.h 中添加 toy 条目：

```
class Triple {
public:
enum ArchType {
UnknownArch,
arm, // ARM (little endian): arm, armv.*, xscale
armeb, // ARM (big endian): armeb
aarch64, // AArch64 (little endian): aarch64
...
...
toy // TOY: toy
};
```

3. 在文件 llvm_root_dir/include/llvm/MC/MCExpr.h 中添加 toy 条目：

```
class MCSymbolRefExpr : public MCExpr {
public:
enum VariantKind {
...
VK_TOY_LO,
```

```
VK_TOY_HI,
};
```

4. 在文件 llvm_root_dir/include/llvm/Support/ELF.h 中添加 toy 条目：

```
enum {
EM_NONE = 0, // No machine
EM_M32 = 1, // AT&T WE 32100
...
...
EM_TOY = 220 // whatever is the next number
};
```

5. 在文件 lib/MC/MCExpr.cpp 中（省去根目录，下同）添加 toy 条目：

```
StringRef
MCSymbolRefExpr::getVariantKindName(VariantKind Kind) {
  switch (Kind) {
  ...
  ...
  case VK_TOY_LO: return "TOY_LO";
  case VK_TOY_HI: return "TOY_HI";
  }
  ...
}
```

6. 接着，在文件 lib/Support/Triple.cpp 中添加 toy 条目：

```
const char *Triple::getArchTypeName(ArchType Kind) {
  switch (Kind) {
  ...
  ...
  case toy: return "toy";
  }
}

const char *Triple::getArchTypePrefix(ArchType Kind) {
  switch (Kind) {
  ...
  ...
  case toy: return "toy";
  }
}

Triple::ArchType Triple::getArchTypeForLLVMName(StringRef Name) {
  ...
```

```
  ...
  .Case("toy", toy);
  ...
}

static unsigned getArchPointerBitWidth(llvm::Triple::ArchType Arch) {
  ...
  ...
  case llvm::Triple::toy:
  return 32;
  ...
  ...
}

Triple Triple::get32BitArchVariant() const {
  ...
  ...
  case Triple::toy:
  // Already 32-bit.
  break;
  ...
}

Triple Triple::get64BitArchVariant() const {
  ...
  ...
  case Triple::toy:
  T.setArch(UnknownArch);
  break;
  ...
  ...
}
```

7. 在文件 lib/Target/LLVMBuild.txt 中添加 toy 目录：

```
[common]
subdirectories = ARM AArch64 CppBackend Hexagon MSP430 ... ... TOY
```

8. 在目录 lib/Target/TOY 中创建一个文件命名 TOY.h：

```
#ifndef TARGET_TOY_H
#define TARGET_TOY_H
#include "MCTargetDesc/TOYMCTargetDesc.h"
#include "llvm/Target/TargetMachine.h"
namespace llvm {
  class TargetMachine;
```

```
  class TOYTargetMachine;
  FunctionPass *createTOYISelDag(TOYTargetMachine &TM,
CodeGenOpt::Level OptLevel);
} // end namespace llvm;
#endif
```

9. 在目录 lib/Target/TOY 下创建一个新目录，命名 TargetInfo。在其中创建一个新文件，命名 TOYTargetInfo.cpp，内容如下：

```
#include "TOY.h"
#include "llvm/IR/Module.h"
#include "llvm/Support/TargetRegistry.h"
using namespace llvm;
Target llvm::TheTOYTarget;
extern "C" void
LLVMInitializeTOYTargetInfo() {
  RegisterTarget<Triple::toy> X(TheTOYTarget, "toy", "TOY");
}
```

10. 在相同目录中，创建 CMakeLists.txt 文件：

```
add_llvm_library(LLVMTOYInfo TOYTargetInfo.cpp)
```

11. 在相同目录中，创建 LLVMBuild.txt 文件：

```
[component_0]
type = Library
name = TOYInfo
parent = TOY
required_libraries = Support
add_to_library_groups = TOY
```

12. 在目录 lib/Target/TOY 中，创建 TOYTargetMachine.cpp 文件：

```
#include "TOYTargetMachine.h"
#include "TOY.h"
#include "TOYFrameLowering.h"
#include "TOYInstrInfo.h"
#include "TOYISelLowering.h"
#include "TOYSelectionDAGInfo.h"
#include "llvm/CodeGen/Passes.h"
#include "llvm/IR/Module.h"
#include "llvm/PassManager.h"
#include "llvm/Support/TargetRegistry.h"
using namespace llvm;

TOYTargetMachine::TOYTargetMachine(const Target &T, StringRef TT,
```

```
      StringRef CPU, StringRef FS, const TargetOptions &Options,
      Reloc::Model RM, CodeModel::Model CM, CodeGenOpt::Level OL)
      : LLVMTargetMachine(T, TT, CPU, FS, Options, RM, CM, OL),
        Subtarget(TT, CPU, FS, *this) {
  initAsmInfo();
}

namespace {
  class TOYPassConfig : public TargetPassConfig {
  public:
    TOYPassConfig(TOYTargetMachine *TM, PassManagerBase &PM) :
TargetPassConfig(TM, PM) {}
    TOYTargetMachine &getTOYTargetMachine() const {
      return getTM<TOYTargetMachine>();
    }
    virtual bool addPreISel();
    virtual bool addInstSelector();
    virtual bool addPreEmitPass();
  };
} // namespace

TargetPassConfig *TOYTargetMachine::createPassConfig(PassManagerBase
&PM) {
  return new TOYPassConfig(this, PM);
}

bool TOYPassConfig::addPreISel() {
  return false;
}

bool TOYPassConfig::addInstSelector() {
  addPass(createTOYISelDag(getTOYTargetMachine(), getOptLevel()));
  return false;
}

bool TOYPassConfig::addPreEmitPass() {
  return false;
}

// Force static initialization.
extern "C" void LLVMInitializeTOYTarget() {
  RegisterTargetMachine<TOYTargetMachine> X(TheTOYTarget);
}
```

```
void TOYTargetMachine::addAnalysisPasses(PassManagerBase &PM) {}
```

13. 创建一个新目录 MCTargetDesc 和一个新文件 TOYMCTargetDesc.h：

```
#ifndef TOYMCTARGETDESC_H
#define TOYMCTARGETDESC_H
#include "llvm/Support/DataTypes.h"
namespace llvm {
class Target;
class MCInstrInfo;
class MCRegisterInfo;
class MCSubtargetInfo;
class MCContext;
class MCCodeEmitter;
class MCAsmInfo;
class MCCodeGenInfo;
class MCInstPrinter;
class MCObjectWriter;
class MCAsmBackend;
class StringRef;
class raw_ostream;
extern Target TheTOYTarget;

MCCodeEmitter *createTOYMCCodeEmitter(const MCInstrInfo &MCII,
  const MCRegisterInfo &MRI, const MCSubtargetInfo &STI, MCContext &Ctx);
MCAsmBackend *createTOYAsmBackend(const Target &T,
  const MCRegisterInfo &MRI, StringRef TT, StringRef CPU);
MCObjectWriter *createTOYELFObjectWriter(raw_ostream &OS, uint8_t OSABI);
} // End llvm namespace
#define GET_REGINFO_ENUM
#include "TOYGenInstrInfo.inc"
#define GET_SUBTARGETINFO_ENUM
#include "TOYGenSubtargetInfo.inc"
#endif
```

14. 在相同目录中再创建一个文件，命名 TOYMCTargetDesc.cpp：

```
#include "TOYMCTargetDesc.h"
#include "InstPrinter/TOYInstPrinter.h"
#include "TOYMCAsmInfo.h"
#include "llvm/MC/MCCodeGenInfo.h"
#include "llvm/MC/MCInstrInfo.h"
#include "llvm/MC/MCSubtargetInfo.h"
#include "llvm/MC/MCStreamer.h"
#include "llvm/Support/ErrorHandling.h"
#include "llvm/Support/FormattedStream.h"
```

```
#include "llvm/Support/TargetRegistry.h"
#define GET_INSTRINFO_MC_DESC
#include "TOYGenInstrInfo.inc"
#define GET_SUBTARGETINFO_MC_DESC
#include "TOYGenSubtargetInfo.inc"
#define GET_REGINFO_MC_DESC
#include "TOYGenRegisterInfo.inc"
using namespace llvm;

static MCInstrInfo *createTOYMCInstrInfo() {
  MCInstrInfo *X = new MCInstrInfo();
  InitTOYMCInstrInfo(X);
  return X;
}

static MCRegisterInfo *createTOYMCRegisterInfo(StringRef TT) {
  MCRegisterInfo *X = new MCRegisterInfo();
  InitTOYMCRegisterInfo(X, TOY::LR);
  return X;
}

static MCSubtargetInfo *createTOYMCSubtargetInfo(StringRef TT,
  StringRef CPU, StringRef FS) {
  MCSubtargetInfo *X = new MCSubtargetInfo();
  InitTOYMCSubtargetInfo(X, TT, CPU, FS);
  return X;
}

static MCAsmInfo *createTOYMCAsmInfo(const MCRegisterInfo &MRI, StringRef
TT) {
  MCAsmInfo *MAI = new TOYMCAsmInfo(TT);
  return MAI;
}

static MCCodeGenInfo *createTOYMCCodeGenInfo(StringRef TT, Reloc::Model
RM,
  CodeModel::Model CM, CodeGenOpt::Level OL) {
  MCCodeGenInfo *X = new MCCodeGenInfo();
  if (RM == Reloc::Default) {
    RM = Reloc::Static;
  }
  if (CM == CodeModel::Default) {
    CM = CodeModel::Small;
  }
```

```
  if (CM != CodeModel::Small && CM != CodeModel::Large) {
    report_fatal_error("Target only supports CodeModel Small or Large");
  }
  X->initMCCodeGenInfo(RM, CM, OL);
  return X;
}

static MCInstPrinter *createTOYMCInstPrinter(const Target &T, unsigned
SyntaxVariant,
  const MCAsmInfo &MAI, const MCInstrInfo &MII, const MCRegisterInfo &MRI,
  const MCSubtargetInfo &STI) {
  return new TOYInstPrinter(MAI, MII, MRI);
}

static MCStreamer *createMCAsmStreamer(MCContext &Ctx,
formatted_raw_ostream &OS,
  bool isVerboseAsm, bool useDwarfDirectory, MCInstPrinter *InstPrint,
  MCCodeEmitter *CE, MCAsmBackend *TAB, bool ShowInst) {
  return createAsmStreamer(Ctx, OS, isVerboseAsm,
  useDwarfDirectory, InstPrint, CE, TAB, ShowInst);
}

static MCStreamer *createMCStreamer(const Target &T, StringRef TT,
  MCContext &Ctx, MCAsmBackend &MAB, raw_ostream &OS, MCCodeEmitter
*Emitter,
  const MCSubtargetInfo &STI, bool RelaxAll, bool NoExecStack) {
  return createELFStreamer(Ctx, MAB, OS, Emitter, false, NoExecStack);
}

// Force static initialization.
extern "C" void LLVMInitializeTOYTargetMC() {
  // Register the MC asm info.
  RegisterMCAsmInfoFn X(TheTOYTarget, createTOYMCAsmInfo);
  // Register the MC codegen info.
  TargetRegistry::RegisterMCCodeGenInfo(TheTOYTarget,
createTOYMCCodeGenInfo);
  // Register the MC instruction info.
  TargetRegistry::RegisterMCInstrInfo(TheTOYTarget, createTOYMCInstrInfo);
  // Register the MC register info.
  TargetRegistry::RegisterMCRegInfo(TheTOYTarget, createTOYMCRegisterInfo);
  // Register the MC subtarget info.
  TargetRegistry::RegisterMCSubtargetInfo(TheTOYTarget,
createTOYMCSubtargetInfo);
  // Register the MCInstPrinter
```

```
  TargetRegistry::RegisterMCInstPrinter(TheTOYTarget,
createTOYMCInstPrinter);
  // Register the ASM Backend.
  TargetRegistry::RegisterMCAsmBackend(TheTOYTarget,
createTOYAsmBackend);
  // Register the assembly streamer.
  TargetRegistry::RegisterAsmStreamer(TheTOYTarget,
createMCAsmStreamer);
  // Register the object streamer.
  TargetRegistry::RegisterMCObjectStreamer(TheTOYTarget,
createMCStreamer);
  // Register the MCCodeEmitter
  TargetRegistry::RegisterMCCodeEmitter(TheTOYTarget,
createTOYMCCodeEmitter);
}
```

15. 在相同目录中创建 LLVMBuild.txt 文件：

```
[component_0]
type = Library
name = TOYDesc
parent = TOY
required_libraries = MC Support TOYAsmPrinter TOYInfo
add_to_library_groups = TOY
```

16. 创建 CMakeLists.txt 文件：

```
add_llvm_library(LLVMTOYDesc TOYMCTargetDesc.cpp)
```

最后，编译整个 LLVM 项目，如下：

```
$ cmake llvm_src_dir
 -DCMAKE_BUILD_TYPE=Release -DLLVM_TARGETS_TO_BUILD="TOY"
$ make
```

这里，我们（只）为 TOY 架构编译了 LLVM 编译器。编译结束之后，我们用 llc 检查 TOY 目标是否存在：

```
$ llc -version
...
...
Registered Targets :
toy - TOY
```

用 llc 工具编译下面的 IR，将生成相应的汇编，如下所示：

```
target datalayout =
```

```
"e-m:e-p:32:32-i1:8:32-i8:8:32-i16:16:32-i64:32-f64:32-a:0:32-n32"
target triple = "toy"
define i32 @foo(i32 %a, i32 %b) {
entry:
    %c = add nsw i32 %a, %b
    ret i32 %c
}

$ llc foo.ll
.text
.file "foo.ll"
.global foo
.type foo,@function
foo: # @foo
# BB#0: # %entry
add r0, r0, r1
b lr
.Ltmp0:
.size foo, .Ltmp0-foo
```

想要详细了解如何向 llc 工具注册一个目标架构，你可以访问 Chen Chung-Shu 和 Anoushe Jamshidi 写的文档：

http://llvm.org/docs/WritingAnLLVMBackend.html#target-registration
http://jonathan2251.github.io/lbd/llvmstructure.html#target-registration

总结

在本章中，我们简单讨论了在 LLVM 中如何表示一种目标架构的机器。我们利用 tablegen 工具组织目标架构的信息，包括寄存器集、指令集、调用惯例等。这是相当容易的。llvm-tablegen 将.td 目标描述文件变换为枚举值，为很多程序逻辑所用，例如帧低层化、指令选择、指令打印等。更精细复杂的架构，比如 ARM 和 X86，让我们洞察精细的目标架构描述方法。

在第 1 章中，我们动手练习运行 LLVM 基础设施提供的各种工具。在随后两章，即第 2 章构造 LLVM IR，和第 3 章高级 LLVM IR，我们用 LLVM 提供的 API 输出 IR。读者的前端

编译器可以利用这些 API 将语言变换为 LLVM IR。在第 5 章高级 IR 转换中，我们认识了 IR 优化 Pass 流水线，学习了几个例子。在第 6 章 IR 到 Selection DAG 中，我们熟悉了 IR 到 Selection DAG 的变换，这是实现输出机器代码的必要一步。在本章即最后一章中，我们学习了如何利用 tablegen 表示一个示例架构，并且用它输出代码。

阅读这本书之后，我们期望读者能够熟悉 LLVM 基础设施，准备继续深入探索 LLVM，为自己定制的架构或者语言设计编译器。编译快乐！