



Learn LLVM 12

A beginner's guide to learning LLVM compiler tools and core libraries with C++

Kai Nacke



Learn LLVM 12

A beginner's guide to learning LLVM compiler tools and core libraries with C++

作者: Kai Nacke

译者: 陈晓伟

本书概述

学习如何构建和使用编译器，包括前端、流水线优化和利用 LLVM 核心库的强大功能构建新的后端编译器。

LLVM 是为了弥合编译器理论和实际开发之间的差异而出现的。它提供了模块化的代码库和先进的工具，帮助开发人员轻松地构建编译器。本书提供了对 LLVM 的介绍，帮助读者在各种情况下构建和使用编译器。

本书将从配置、构建和安装 LLVM 库、工具和外部项目开始。接着，向您介绍 LLVM 的设计，以及在每个编译器阶段（前端、优化器和后端）的实际工作方式。以实际编程语言为例，学习如何使用 LLVM 开发前端编译器，并生成 LLVM IR，将其交给优化流水线，并从中生成机器码。后面的章节将展示如何扩展 LLVM，以及 LLVM 中的指令选择是如何工作的。在了解如何为 LLVM 开发新的后端编译器之前，将重点讨论即时编译问题和 LLVM 提供的 JIT 编译的支持情况。

阅读本书后，您将获得使用 LLVM 编译器开发框架的实际经验，并得到一些具有帮助性的实际示例和源代码片段。

关键特性：

- 学习如何有效地使用 LLVM
- 理解 LLVM 编译器的高级设计，并将原则应用到自己的编译器中
- 使用基于编译器的工具来提高 C++ 项目的代码质量

内容纲要：

- 配置、编译和安装 LLVM 框架
- 理解 LLVM 源码的结构
- 了解在项目中可以使用 LLVM 做什么
- 探索编译器是如何构造的，并实现一个小型编译器
- 为通用源语言构造生成 LLVM IR
- 建立优化流水线，并根据自己的需要进行调整
- 使用转换通道和 Clang 工具对 LLVM 进行扩展
- 添加新的机器指令和完整的后端编译器

作者简介

Kai Nacke 是一名专业 IT 架构师，目前居住在加拿大多伦多。毕业于德国多特蒙德技术大学的计算机专业。他关于通用哈希函数的毕业论文，被评为最佳论文。

他在 IT 行业工作超过 20 年，在业务和企业应用程序的开发和架构方面有丰富的经验。他在研发一个基于 LLVM/Clang 的编译器。

几年来，他一直是 LDC(基于 LLVM 的 D 语言编译器) 的维护者。在 Packt 出版过《D Web Development》一书，他也曾在自由和开源软件开发者欧洲会议 (FOSDEM) 的 LLVM 开发者室做过演讲。

审评者介绍

Suyog Sarda 是一名专业的软件工程师和开源爱好者，专注于编译器开发和编译器工具，是 LLVM 开源社区的积极贡献者。他毕业于印度浦那工程学院，具有计算机技术学士学位。Suyog 还参与了 ARM 和 X86 架构的代码性能改进，一直是 Tizen 项目编译团队的一员，对编译器开发的兴趣在于代码优化和向量化。之前，他写过一本关于 LLVM 的书，名为《LLVM Cookbook》，由 Packt 出版。除了编译器，Suyog 还对 Linux 内核开发感兴趣。他在迪拜 Birla Institute of Technology 的 2012 年 IEEE Proceedings of the International Conference on Cloud Computing, Technologies, Applications, and Management 上发表了一篇题为《VM pin and Page Coloring Secure Co-resident Virtualization in Multicore Systems》的技术论文。

本书相关

- github 翻译地址: <https://github.com/xiaoweiChen/Learn-LLVM-12>
- 本书代码: <https://github.com/PacktPublishing/Learn-LLVM-12>

写书是一项具有挑战性的任务。尤其是正计划移居加拿大时，突然一场流行病袭击了世界，改变了一切。

Packt 的团队在写作上给予指导，也对我缓慢的写作速度表示理解，并一直激励我坚持下去。我非常非常感谢他们。

没有家人的支持，就不可能有这本书。谢谢对我的信心！

目录

1 构建 LLVM	11
第 1 章 安装 LLVM	12
相关准备	12
使用 CMake 构建	14
定制化构建	17
总结	20
第 2 章 浏览 LLVM	21
相关代码	21
LLVM 代码的内容	21
LLVM 的项目结构	22
使用 LLVM 创建自己的项目	24
针对不同的 CPU 架构	34
总结	37
第 3 章 编译器结构	38
相关代码	38
编译器的构建块	38
算术表达式语言	39
语法分析	40
语义分析	51
使用 LLVM 后端编译器生成代码	53
总结	60
2 从源码到机器码	62
第 4 章 将源文件转换为抽象语法树	63
相关代码	63
定义一种编程语言	63
创建项目结构	66
管理源文件和用户消息	66
构建词法分析器	69
构建递归下降解析器	73
用 bison 和 flex 生成解析器和词法分析器	76
执行语义分析	79
总结	88
第 5 章 生成 IR——基础知识	89
相关代码	89
使用 AST 生成 IR	89
使用 AST 编码生成 SSA 形式的 IR	94
设置模块和驱动程序	103

总结	108
第 6 章 生成高级语言结构的 IR	109
相关代码	109
使用数组、结构体和指针	109
正确的获取二进制程序的接口	112
为类和虚函数创建 IR 代码	113
总结	119
第 7 章 生成高级语言结构的 IR	120
相关代码	120
抛出和捕获异常	120
基于类型的别名生成元数据的分析	129
添加调试元数据	134
总结	145
第 8 章 优化 IR	146
相关代码	146
介绍 LLVM Pass 管理器	146
使用新 Pass 管理器实现一个 Pass	147
使用旧 Pass 管理器适配 Pass	154
向编译器添加优化流水线	156
总结	163
3 LLVM 进阶	164
第 9 章 指令选择	165
相关代码	165
理解 LLVM 目标后端结构	165
使用 MIR 对后端进行测试和调试	166
如何选择指令	169
支持新的机器指令	180
总结	185
第 10 章 JIT 编译	186
相关代码	186
概述 LLVM 的 JIT 实现和用例	186
使用 JIT 编译直接执行	187
利用 JIT 编译器进行代码计算	200
总结	202
第 11 章 使用 LLVM 工具调试	203
相关代码	203
使用 sanitizer 安装应用程序	203
用 libFuzzer 找 Bug	208
使用 xRay 进行性能分析	211

使用 Clang 静态分析器检查源码	214
创建自己的基于 Clang 的工具	222
总结	227
第 12 章 自定义编译器后端	229
相关代码	229
为新的后端搭建舞台	229
将新的体系结构添加到 Triple 类中	230
在 LLVM 中扩展 ELF 文件格式定义	231
创建目标描述	232
实现 DAG 指令选择类	238
生成汇编指令	245
计算机编码	247
支持反汇编	249
自定义后端的汇总	252
总结	254

前言

构造编译器是一项复杂而有趣的任务。LLVM 为编译器提供了可重用组件，其核心库实现了一个世界级的优化代码生成器，它为所有主流 CPU 架构转换代码提供了独立的中间表示。许多编译器已经在使用 LLVM 的相关技术了。

这本书会教您如何实现自己的编译器，并通过 LLVM 实现它。您将了解前端编译器如何将源代码转换为抽象语法树，以及如何从中生成中间表示 (IR)。如何向编译器添加优化流水，对 IR 进行优化，并编译为性能不错的机器代码。

LLVM 框架可以通过多种方式进行扩展，您将学习如何添加新的通道、新的机器指令，甚至是全新的 LLVM 后端。还有高级主题，如编译不同的 CPU 架构和扩展 Clang、Clang 静态分析器与插件和检查器也涵盖其中。本书遵循最实用的方法，并附带了示例源代码，这可以让您更容易得在项目中应用所学习到的知识。

适宜读者

本书为刚接触 LLVM 并且对 LLVM 框架感兴趣的编译器开发人员、爱好者和工程师编写。对于希望使用基于编译器的工具进行代码分析和改进的 C++ 软件工程师，以及希望获得更多 LLVM 基本知识的 LLVM 库的用户也很有用。为了更有效地理解本书中所包含的概念，必须具备 C++ 中级水平。

本书内容

第 1 章，安装 LLVM，介绍如何设置开发环境。了解如何编译了 LLVM 库，并学习了如何自定义构建过程。

第 2 章，浏览 LLVM，介绍各种 LLVM 项目，并讨论所有项目的统一目录结构。您将使用 LLVM 核心库创建第一个项目，还可以为不同的 CPU 架构编译它。

第 3 章，编译器结构，提供编译器组件的概述。您将实现可以生成 LLVM IR 的编译器。

第 4 章，将源码转换为抽象语法树，详细地介绍了如何实现编译器前端。您将为小型编程语言创建前端，最后构建抽象语法树。

第 5 章，生成 IR——基础知识，展示了如何从抽象语法树生成 LLVM IR。本章的最后，您将为示例语言实现编译器，从而产生汇编文本或目标代码文件。

第 6 章，生成高级语言结构的 IR，介绍了如何将高级编程语言中常见的源语言特性转换为 LLVM IR。您将了解聚合数据类型的转换、实现类继承和虚函数的各种选项，以及如何遵循系统的 ABI。

第 7 章，生成 IR——进阶知识，介绍了如何在源语言中为异常处理语句生成 LLVM IR。了解如何为基于类型的别名分析添加元数据，以及如何向生成的 LLVM IR 添加调试信息，并且您将扩展编译器生成的元数据。

第 8 章，优化 IR，介绍 LLVM Pass 管理器。您将实现你自己的 Pass，作为 LLVM 的一部分或者插件，您将学习如何添加新 Pass 到优化流水中。

第 9 章，选择指令，介绍了 LLVM 如何减少生成机器指令的 IR。您将学习如何在 LLVM 中定义指令，并将向 LLVM 添加一条新的机器指令，以便在选择指令时使用新指令。

第 10 章, JIT 编译, 讨论如何使用 LLVM 来实现 JIT 编译器。本章结束时, 您将通过两种不同的方式实现自己的 LLVM IR JIT 编译器。

第 11 章, 使用 LLVM 工具调试, 探索了 LLVM 的各种库和组件, 这可以帮助您了解应用程序中的 bug。您将使用“杀虫剂”来识别内存溢出和其他错误。使用 libFuzzer 库, 将随机数据作为输入测试函数。XRay 也能帮助您找到性能瓶颈。您将使用 Clang 静态分析程序来识别源代码级的 bug, 并且可以将自己的检查程序添加到分析程序中。您还将学习如何使用自己的插件扩展 Clang。

第 12 章, 自定义编译器后端, 说明如何向 LLVM 添加新的后端。需要实现所有必要的类。在本章的最后, 将为某种 CPU 架构编译 LLVM IR。

编译环境

需要一台运行 *Linux*、*Windows*、*macOS* 或 *FreeBSD* 的计算机, 并为该操作系统安装了开发工具链。所需工具见表。所有工具都应该在 *shell* 的搜索路径中。

书中涉及的软件/硬件	操作系统
C/C++ 编译器: gcc 5.1.0 或更高版本, clang3.5 或更高版本 Apple clang 6.0 或更高版本, Visual Studio 2017 或更高版本	Linux(任意衍生版), Windows, macOS, 或 FreeBSD
CMake 3.13.4 或更高版本	
Ninja 1.9.0	
Python 3.6 或更高版本	
Git 1.7.10 或更高版本	

第 9 章“指令选择”中的 DAG 可视化, 必须安装<https://graphviz.org/>的 Graphviz。默认情况下, 生成的图像是 PDF 格式的, 所以需要 PDF 查看器来显示。

第 11 章“使用 LLVM 工具调试”中创建“火焰图”, 需要从<https://github.com/brendangregg/FlameGraph>获取安装脚本。要运行该脚本, 还需要安装最新版本的 Perl, 要查看图形, 还需要一个能够显示 SVG 文件的 Web 浏览器 (主流浏览器都没问题)。要在同一章中看到 Chrome 跟踪查看器可视化, 需要安装 Chrome 浏览器。

如果正在使用本书的数字版本, 我们建议您自己输入代码或通过 GitHub 存储库访问代码 (下一节提供链接), 将避免复制和粘贴代码。

下载示例

可以从 GitHub 网站<https://github.com/PacktPublishing/Learn-LLVM-12>下载本书的示例代码。如果有对代码的更新, 也会在现有的 GitHub 存储库中更新。

我们还在<https://github.com/PacktPublishing/>上提供了丰富的图书和视频目录中的其他代码包。可以一起拿来看看!

实例演示

本书代码的演示视频可以在<https://bit.ly/3nllhED>上查看

下载彩图

我们还提供了一个 PDF 文件，里面有本书中使用的屏幕截图/图表的彩色图像。可在此下载：https://static.packt-cdn.com/downloads/9781839213502_ColorImages.pdf。

联系方式

我们欢迎读者的反馈。

反馈：如果你对这本书的任何方面有疑问，需要在你的信息的主题中提到书名，并给我们发邮件到customercare@packtpub.com。

勘误：尽管我们谨慎地确保内容的准确性，但错误还是会发生。如果您在本书中发现了错误，请向我们报告，我们将不胜感激。请访问www.packtpub.com/support/errata，选择相应书籍，点击勘误表提交表单链接，并输入详细信息。

盗版：如果您在互联网上发现任何形式的非法拷贝，非常感谢您提供地址或网站名称。请通过copyright@packt.com与我们联系，并提供材料链接。

如果对成为书籍作者感兴趣：如果你对某主题有专长，又想写一本书或为之撰稿，请访问authors.packtpub.com。

欢迎评论

请留下评论。当您阅读并使用了本书，为什么不在购买网站上留下评论呢？其他读者可以看到您的评论，并根据您的意见来做出购买决定。我们在 Packt 可以了解您对我们产品的看法，作者也可以看到您对他们撰写书籍的反馈。谢谢你！

想要了解 Packt 的更多信息，请访问packt.com。

1 构建 LLVM

本节中，您将学习如何自己编译 LLVM，以及如何根据您的需要调整构建。您将了解 LLVM 项目是如何组织的，并将利用 LLVM 创建您的第一个项目。您还将学习如何为不同的 CPU 架构使用 LLVM，编译 LLVM 和应用程序。最后，您将探索编译器的结构，并创建一个小型编译器。

本节包括以下几章：

- 第 1 章，安装 LLVM
- 第 2 章，浏览 LLVM
- 第 3 章，编译器结构

第 1 章 安装 LLVM

要学习如何使用 LLVM，最好先从源代码开始编译 LLVM。LLVM 是一个伞形项目，它的 GitHub 存储库包含了属于 LLVM 的所有项目的源代码。每个 LLVM 项目都位于存储库的顶级目录中。除了克隆存储库之外，您的系统还必须安装构建系统所需的所有工具。

本章中，您将学习到以下内容：

- 准备好环境，并展示如何设置构建系统。
- 使用 CMake 构建，这将包括如何编译和安装 LLVM 核心库以及使用 CMake 和 Ninja 编译和安装 Clang。
- 定制化构建过程，了解影响构建过程的各种方式。

相关准备

要使用 LLVM，您的系统必须为常见的操作系统，如 Linux、FreeBSD、macOS 或 Windows。使用 Debug 构建 LLVM 和 Clang 很需要数十 GB 的磁盘空间，因此要确保您的系统有足够的可用磁盘空间，至少应该有 30GB 的空余空间。

所需的磁盘空间很大程度上取决于所选的构建选项。仅以发布模式构建 LLVM 核心库，而且只针对一个平台，需要大约 2GB 的空余空间，这是最低需求。为了减少编译时间，一个高速的 CPU(比如 2.5GHz 时钟速度的四核 CPU) 和一块 SSD 硬盘也会很有帮助。

甚至可以在小设备上构建 LLVM，比如树莓派——只是需要花费很多时间。我在笔记本电脑上开发了本书的示例，这台笔记本电脑配备了 2.7GHz 的英特尔四核 CPU，具有 40GB RAM 和 2.5TB SSD 硬盘空间。

您的开发系统必须安装一些必要的软件。让我们了解一下这些软件包的最低要求。

Note

Linux 发行版通常包含可以使用的最新版本，版本号为 LLVM 12。LLVM 的最新版本可能需要这里提到的软件包的最新版本。

从 GitHub 下载源代码，需要 git(<https://git-scm.com/>)，对版本没有要求。GitHub 帮助页面推荐使用 1.17.10 以上版本。

LLVM 项目使用 CMake(<https://cmake.org/>) 作为构建文件生成器，至少为 3.13.4。CMake 可以为各种构建系统生成构建文件。本书中，使用 Ninja(<https://ninja-build.org/>)，是因为它编译速度快，适用于所有平台，建议使用最新版本 1.9.0。

您还需要一个 C/C++ 编译器。LLVM 项目是用现代 C++ 编写的，基于 C++14 标准。需要符合标准的编译器和标准库。下面的编译器可以编译 LLVM 12:

- gcc 5.1.0 或更高版本
- Clang 3.5 或更高版本
- Apple Clang 6.0 或更高版本
- Visual Studio 2017 或更高版本

请注意，随着 LLVM 项目的开发，对编译器的需求很可能会发生变化。撰写本文时，有人讨论使用 C++17，并放弃对 Visual Studio 2017 的支持。所以，应该使用系统可用的最新编译器版本。

Python(<https://python.org/>) 用于生成构建文件并运行测试套件，版本至少应该是 3.6。

虽然在本书没有涉及到 Makefile，但可能有一些原因需要使用 Make 而不是 Ninja。这种情况下，您需要使用 GNU Make(<https://www.gnu.org/software/make/>) 3.79 或更高版本。这两种构建工具的用法非常相似。对于这里的场景，用 make 替换命令中的 ninja 就可以了。

要安装必备软件，最简单的方法是使用操作系统中的包管理器。下面将为主流的操作系统安装软件准备相应的命令行。

Ubuntu

Ubuntu 20.04 使用 APT 包管理器。大多数基础设施已经安装完毕，只缺少开发工具。输入以下命令：

```
$ sudo apt install -y gcc g++ git cmake ninja-build
```

Fedora 和 RedHat

Fedora 33 和 RedHat Enterprise Linux 8.3 的包管理器称为 DNF。和 Ubuntu 一样，大多数基本实用程序都已经安装好了。输入以下命令：

```
$ sudo dnf install -y gcc gcc-c++ git cmake ninja-build
```

FreeBSD

在 FreeBSD 12 或更高版本上，您必须使用 PKG 包管理器。FreeBSD 与基于 linux 的系统的不同之处在于，Clang 是首选编译器。输入以下命令：

```
$ sudo pkg install -y clang git cmake ninja
```

OS X

对于 OS X 上的开发，最好从 Apple 商店安装 Xcode。虽然本书中没有使用 XCode IDE，但它附带了所需的 C/C++ 编译器和实用程序。要安装其他工具，可以使用 Homebrew 软件包管理器 (<https://brew.sh/>)。输入以下命令：

```
$ brew install git cmake ninja
```

Windows

和 OS X 一样，Windows 没有包管理器。安装所有软件的最简单方法是使用 Chocolatey(<https://chocolatey.org/>) 包管理器。输入以下命令：

```
$ choco install visualstudio2019buildtools cmake ninja git gzip bzip2  
gnuwin32-coreutils.install
```

请注意，这只安装来自 Visual Studio 2019 的构建工具。如果你想获得 Community Edition(包含 IDE)，那么你必须安装 visualstudio2019community 包而不是 visualstudio2019 buildtools。Visual Studio 2019 安装的一部分是 VS 2019 的 x64 Native Tools 命令提示符。使用此命令提示符时，编译器将自动添加到搜索路径中。

配置 Git

LLVM 项目使用 Git 进行版本控制。如果没有使用过 Git，那么应该先做一些 Git 的基本配置；也就是说，设置用户名和电子邮件地址。如果提交更改，将使用这两条信息。以下命令中，将 Jane 替换为您的姓名，将 jane@email.org 替换为您的电子邮件：

```
$ git config --global user.email "jane@email.org"
```

```
$ git config --global user.name "Jane"
```

通常情况下，Git 使用 vi 编辑器提交消息。如果您更喜欢使用另一种编辑器，可以以类似的方式更改配置，例如：要使用 nano 编辑器：

```
$ git config --global core.editor nano
```

关于 git 的更多信息，请参阅由 Packt 出版的另一本书，《Git Version Control Cookbook - Second Edition》(<https://www.packtpub.com/product/git-version-control-cookbook/9781782168454>)。

使用 CMake 构建

准备好构建工具后，就可以从 GitHub 签出所有的 LLVM 项目。所有平台上执行此操作的命令基本相同。但在 Windows 上，建议关闭对行结束符的自动转译。

我们分三部分来回顾这个过程：克隆存储库、创建构建目录和生成构建系统文件。

克隆代码库

在所有非 Windows 平台上，输入以下命令克隆代码库：

```
$ git clone https://github.com/llvm/llvm-project.git
```

在 Windows 上，必须添加选项以禁用自动转译行结束符。在这里输入以下内容：

```
$ git clone --config core.autocrlf=false https://github.com/llvm/llvm-project.git
```


这将最新的源代码从 GitHub 克隆到一个名为 llvm-project 的本地目录中。现在,进入 llvm-project 目录:

```
$ cd llvm-project
```

这个目录包含所有的 LLVM 项目, 每个项目都有自己的目录。最值得注意的是, LLVM 核心库位于 LLVM 子目录中。LLVM 项目使用分支来进行后续版本开发 (“release/12.x”) 和标记 (“llvmorg-12.0.0”) 来标记某个版本。通过前面的 clone 命令, 可以获得当前的开发状态。本书使用 LLVM 12。要查看 LLVM 12 的第一个版本, 输入以下命令:

```
$ git checkout -b llvmorg-12.0.0
```

这样, 就克隆了整个存储库, 检出到对应的标记。

Git 还允许只克隆一个分支或标记 (包括历史记录)。使用 `git clone --branch llvmorg-12.0.0 https://github.com/llvm/llvm-project`。使用 `-depth=1` 选项, 可以防止历史信息的克隆。这节省了时间和空间, 但显然也限制了你能在本地可以做什么。

下一步就是创建构建目录。

创建构建目录

与许多其他项目不同, LLVM 不支持内联构建, 需要单独的构建目录。可以在 llvm-project 目录中创建一个目录。先进入 llvm-project 目录:

```
$ cd llvm-project
```

然后, 为了简单起见, 创建一个名为 build 的构建目录。Unix 和 Windows 系统的命令是不同, 在类 Unix 系统上, 应该使用以下命令:

```
$ mkdir build
```

在 Windows 上, 应该使用以下命令:

```
$ md build
```

然后, 切换到构建目录:

```
$ cd build
```

现在, 您可以在这个目录中使用 CMake 工具创建构建系统文件了。

生成构建系统文件

要生成将使用 Ninja 编译 LLVM 和 Clang 的构建系统文件, 请运行以下命令:

```
$ cmake -G Ninja -DLLVM_ENABLE_PROJECTS=clang ../llvm
```

Tip

在 Windows 上，反斜杠字符 \ 是目录名分隔符，CMake 会自动将 Unix 分隔符/转换为 Windows 分隔符。

-G 选项告诉 CMake 要为哪个系统生成构建文件。最常用的选项如下：

- Ninja: 对应 Ninja 的构建系统
- Unix Makefiles: 对应 GNU Make
- Visual Studio 15 VS2017 和 Visual Studio 16 VS2019: 对应 Visual Studio 和 MS Build
- Xcode: 对应 Xcode 工程

可以使用-D 选项设置各种变量来影响生成过程。通常，以 CMAKE_(由 CMAKE 定义) 或 LLVM_(由 LLVM 定义) 作为前缀。使用 LLVM_ENABLE_PROJECTS=clang 变量设置，CMake 为 LLVM 之外的 Clang 生成构建文件。命令的最后一部分告诉 CMake 在哪里可以找到 LLVM 核心库源代码。下一节中会有更多的相关内容。

当生成了构建文件，LLVM 和 Clang 可以用以下命令编译：

```
$ ninja
```

根据硬件资源的不同，该命令的运行时间在 15 分钟 (具有大量 CPU 内核、内存和快速存储的服务器) 到数小时 (内存有限的双核 Windows 笔记本) 之间。默认情况下，Ninja 使用了所有可用的 CPU 核。这有利于提高编译速度，但可能会阻止其他任务的运行。例如，在 Windows 笔记本上，Ninja 在运行时几乎不能上网。幸运的是，可以使用-j 选项限制资源的使用。

假设您有四个可用的 CPU 核，而 Ninja 应该只使用两个 (因为有并行任务要运行)。在这里，应该使用以下命令进行编译：

```
$ ninja -j2
```

当编译完成，可以运行测试套件，以检查是否一切正常：

```
$ ninja check-all
```

同样，该命令的运行时因可用硬件资源的不同而有很大差异。Ninja 检查目标运行所有测试用例，为每个包含测试用例的目录生成目标。使用 check-llvm(而不是 check-all) 是运行 LLVM 测试，而不是 Clang 测试，check-llvm-codegen 只运行来自 LLVM 的 CodeGen 目录中的测试 (即 llvm/test/CodeGen 目录)。

也可以做一个快速的手动检查。使用的 LLVM 的 llc，即 LLVM 编译器。如果使用-version 选项，会显示它的 LLVM 版本，主机 CPU，以及它所支持的所有架构：

```
$ bin/llc -version
```

如果您在编译 LLVM 时有困难,那么可以参考 LLVM 系统文档入门中的常见问题部分 (<https://llvm.org/docs/GettingStarted.html#common-problems>), 以获得常见问题的解决方案。

最后, 安装可执行文件:

```
$ ninja install
```

在类 Unix 系统上, 安装目录是 `/usr/local`。在 Windows 下, 使用 `C:\Program Files\LLVM`。当然可以修改, 下一节将说明如何操作。

定制化构建

CMake 系统使用 `CMakeLists.txt` 文件对项目进行描述。顶层文件在 `llvm` 目录中, 也就是 `llvm/CMakeLists.txt`。其他目录还包含 `CMakeLists.txt`, 在构建文件生成期间会递归地包含这些文件。

根据项目描述中提供的信息, CMake 检查已经安装了哪些编译器, 检测库和符号, 并创建构建系统文件, 如 `build.ninja` 或 `Makefile`(取决于选择的生成器)。还可以定义可重用的模块, 例如检测 LLVM 是否已安装的函数。这些脚本被放置在特殊的 `cmake` 目录 (`llvm/cmake`) 中, 在生成过程中会自动搜索该目录。

构建过程可以通过定义 CMake 变量来定制。命令行选项 `-D` 将为一个变量设置值, 这些变量会在 CMake 脚本中使用。CMake 自己定义的变量几乎总是以 `CMAKE_` 为前缀, 这些变量可以在所有项目中使用。由 LLVM 定义的变量前缀为 `LLVM_`, 但只能在项目定义中包含 LLVM 时使用。

CMake 定义的变量

有些变量是用环境变量的值初始化的。最值得注意的是 `CC` 和 `CXX`, 它们定义了用于构建的 C 和 C++ 编译器。CMake 尝试使用当前的 shell 搜索路径自动定位 C 和 C++ 编译器, 并选择找到的第一个编译器。如果你安装了多个编译器, 比如: `gcc` 和 `Clang` 或不同版本的 `Clang`, 那么默认找到的可能不是预期构建 LLVM 的编译器。

假设您想使用 `clang9` 作为 C 编译器, 使用 `clang++9` 作为 C++ 编译器。可以在 Unix shell 中使用 CMake:

```
$ CC=clang9 CXX=clang++9 cmake ../llvm
```

它会设置 `cmake` 调用的环境变量的值。如果需要, 可以为编译器指定绝对路径。

`CC` 是 `CMAKE_C_COMPILER` `cmake` 变量的默认值, 而 `CXX` 是 `CMAKE_CXX_COMPILER` `cmake` 变量的默认值。您可以直接设置 CMake 变量, 而不使用环境变量。这与前面的调用相同:

```
$ cmake -DCMAKE_C_COMPILER=clang9\  
-DCMAKE_CXX_COMPILER=clang++9 ../llvm
```

CMake 定义的其他常用变量如下:

- **CMAKE_INSTALL_PREFIX**: 在安装过程中添加到每个路径上的路径前缀。Unix 上默认为 `/usr/local`, Windows 上默认为 `C:\Program Files\`。如果要在 `/opt/LLVM` 目录下安装 LLVM, 必须指定 `-DCMAKE_INSTALL_PREFIX=/opt/LLVM`。可执行文件复制到 `/opt/llvm/bin`, 库文件复制到 `/opt/llvm/lib`, 以此类推。
- **CMAKE_BUILD_TYPE**: 不同类型的构建需要不同的设置, 例如: 调试构建需要指定用于生成调试符号的选项, 并且通常是针对系统库的调试版本进行链接。相比之下, 发布版本使用针对库的生产版本的优化标志和链接。此变量仅用于只能处理一种构建类型的构建系统, 如 Ninja 或 Make。对于 IDE 构建系统, 必须使用 IDE 的机制在构建类型之间进行切换。可能的值如下:
DEBUG: 使用调试符号构建
RELEASE: 以速度优化为主的构建
RELWITHDEBINFO: 使用调试符号的发布构建
MINSIZEREL: 以优化生成文件大小为主的构建
默认的构建类型是 DEBUG。要构建为发布版本, 必须指定 `-DCMAKE_BUILD_TYPE=RELEASE`。
- **CMAKE_C_FLAGS** 和 **CMAKE_CXX_FLAGS**: 当我们编译 C 和 C++ 源文件时, 这些是额外的标志。初始值取自 `CFLAGS` 和 `CXXFLAGS` 环境变量, 可以替代变量使用。
- **CMAKE_MODULE_PATH**: 指定在 CMAKE 模块中搜索的附加目录。在搜索默认目录之前搜索指定的目录, 以分号分隔的目录列表。
- **PYTHON_EXECUTABLE**: 如果没有找到 PYTHON 解释器, 或者如果安装了多个版本的 PYTHON 解释器。在 CMake 选择了错误的解释器时, 可以将该变量设置为正确 PYTHON 二进制文件的路径。这个变量只有在包含了 CMake 的 Python 模块时才会生效 (对于 LLVM 也是如此)。

CMake 为变量提供了内置帮助。 `--help-variable var` 选项打印 `var` 变量的帮助信息。例如, 您可以输入以下命令来获取 `CMAKE_BUILD_TYPE` 的帮助:

```
$ cmake --help-variable CMAKE_BUILD_TYPE
```

也可以用下面的命令列出所有的变量 (这个清单很长):

```
$ cmake --help-variablelist
```

LLVM 定义的变量

LLVM 定义的变量的工作方式与 CMake 定义的变量相同, 但没有内置帮助。常用的变量如下:

- **LLVM_TARGETS_TO_BUILD**: LLVM 支持不同的 CPU 架构。默认情况下, 构建所有目标。使用此变量指定要构建的目标列表, 由分号分隔。目前支持的目标有 AArch64、AMDGPU、ARM、BPF、Hexagon、Lanai、Mips、MSP430、NVPTX、PowerPC、RISCV、Sparc、SystemZ、

WebAssembly、X86、XCore。All 可以作为 All 目标的简写，并且名称区分大小写。若要只启用 PowerPC 和 SystemZ 目标，必须指定-DLLVM_TARGETS_To_BUILD="PowerPC;SystemZ"。

- LLVM_ENABLE_PROJECTS: 这是一个要构建的项目列表，由分号分隔。项目的源代码必须与 llvm 目录在同一级别（并排布局）。当前列表是 clang, clangtools-extra, compiler-rt, debuginfo-tests, lib, libclc, libcxx, libcxxabi, libunwind, lld, lldb, llgo, mlir, openmp, parallel-libs, polly 和 pstl。All 可以作为此列表中的所有项目的简写。要和 LLVM 一起构建 Clang 和 llgo，必须指定-DLLVM_ENABLE_PROJECT="Clang;llgo"。
- LLVM_ENABLE_ASSERTIONS: 如果设置为 ON，则启用断言检查。这些检查有助于发现错误，在开发过程中非常有用。对于 DEBUG 版本，默认值为 ON，否则为 OFF。要打开断言检查（对于 RELEASE 版本），必须指定-DLLVM_ENABLE_ASSERTIONS=ON。
- LLVM_ENABLE_EXPENSIVE_CHECKS: 这启用了一些检查，会降低编译速度或消耗大量内存，默认值为 OFF。要打开这些检查，必须设置-DLLVM_ENABLE_EXPENSIVE_CHECKS=ON。
- LLVM_APPEND_VC_REV: llc 等 LLVM 工具显示它们所基于的 LLVM 版本（如果提供了 version 命令行选项）。此版本信息基于 LLVM_REVISION C 宏。默认情况下，不仅 LLVM 版本，最新提交的 Git 哈希值也是版本信息的一部分。如果您正在跟踪主分支的开发，这将非常方便，因为它清楚地表明了该工具是基于哪个 Git 提交的。如果不是必需的，可以使用-DLLVM_APPEND_VC_REV=OFF 关闭。
- LLVM_ENABLE_THREADS: 如果检测到线程库（通常是 pthreads 库），LLVM 会自动包含线程支持。本例中，LLVM 假定编译器支持**线程本地存储 (TLS)**。如果不想让线程支持或者你的编译器不支持 TLS，那么可以使用-DLLVM_ENABLE_THREADS=OFF 来关闭它。
- LLVM_ENABLE_EH: LLVM 项目不使用 C++ 异常处理，所以默认关闭异常支持。此设置可能与您的项目正在链接的其他库不兼容。如果需要，可以通过指定-DLLVM_ENABLE_EH=ON 来启用异常支持。
- LLVM_ENABLE_RTTI: LLVM 使用一个轻量级的、自构建的系统来提供运行时类型信息。默认情况下，C++ RTTI 的生成是关闭的。与异常处理支持一样，这可能与其他库不兼容。要开启 C++ RTTI 的生成，必须设置-DLLVM_ENABLE_RTTI=ON。
- LLVM_ENABLE_WARNINGS: 如果可能的话，编译 LLVM 应该不会产生任何警告消息。默认情况下，打印警告消息的选项是打开的。要关闭它，必须设置-DLLVM_ENABLE_WARNINGS=OFF。
- LLVM_ENABLE_PEDANTIC: LLVM 源文件应该符合 C/C++ 标准。因此，默认情况下启用了对源的学究式检查。如果可能，也禁用编译器特定的扩展。要关闭此设置，必须指定-DLLVM_ENABLE_PEDANTIC=OFF。
- LLVM_ENABLE_WERROR: 如果设置为 ON，则所有警告都视为错误——发现警告，编译就会中止。它有助于在源代码中找到所有剩余的警告。默认情况下，是关闭的。要打开它，必须指定-DLLVM_ENABLE_WERROR=ON。
- LLVM_OPTIMIZED_TABLEGEN: 通常，TABLEGEN 工具与 LLVM 的其他部分使用相同的选项构建。同时，tablegen 用于生成大部分代码生成器。因此，tablegen 在调试构建中要慢得多，从而显著增加了编译时间。如果将此选项设置为 ON，则 tablegen 编译时会启用优化，即使是在调试构建中，也可能会减少编译时间，默认为 OFF。要打开此选项，必须指

定-`LLVM_OPTIMIZED_TABLEGEN=ON`。

- `LLVM_USE_SPLIT_DWARF`: 如果构建编译器是 `gcc` 或 `Clang`，那么打开这个选项编译器将在单独的文件中生成 DWARF 调试信息。减小的对象文件大小大大减少了调试构建的链接时间，默认为 `OFF`。要开启此功能，必须指定-`LLVM_USE_SPLIT_DWARF=ON`。

LLVM 定义了更多的 CMake 变量。可以在 CMake 的 LLVM 文档中找到完整的列表 (<https://releases.llvm.org/12.0.0/docs/CMake.html#llvm-specific-variables>)。前面的列表只包含常用的一些。

总结

本章中，您准备了开发机器来编译 LLVM，克隆了 LLVM GitHub 代码库，并编译了 LLVM 和 Clang。构建过程可以使用 CMake 变量进行定制。还学习了有用的变量以及如何更改它们。有了这些知识，您就可以根据需要调整 LLVM 的构建。

下一章中，我们将更详细地研究 LLVM 代码库的内容。将了解其中包含哪些项目以及这些项目是如何构建的。然后，将使用 LLVM 库创建自己的项目。最后，您将学习如何为不同的 CPU 架构编译 LLVM。

第 2 章 浏览 LLVM

LLVM 代码库包含 `llvm-project` 根目录下的所有项目。为了有效地使用 LLVM，知道哪些是可用的，以及在哪里可以找到。本章中，您将学习到以下内容：

- LLVM 代码库的内容，包括最重要的顶层项目
- LLVM 项目的结构，所有项目都统一的源码目录结构
- 如何使用 LLVM 库创建自己的项目，在自己的项目中使用 LLVM 的方法
- 如何针对不同的 CPU 体系结构，进行交叉编译

相关代码

本章的代码文件可在<https://github.com/PacktPublishing/Learn-LLVM-12/tree/master/Chapter02/tinylang>获取。

你可以在视频中找到代码<https://bit.ly/3nllhED>。

LLVM 代码的内容

在第 1 章中，您克隆了 LLVM 库。这个库包含所有 LLVM 顶层项目，可以分为以下几类：

- LLVM 核心库和附加内容
- 编译器和工具
- 运行时库

下一节中，我们将进一步研究这些。

LLVM 核心库和附加内容

LLVM 核心库位于 `llvm` 目录中。为主流的 CPU 提供了一组带有优化器和代码生成的库，还提供基于这些库的工具。LLVM 静态编译器 `llc` 将 LLVM 中间表示 (IR) 编写的文件作为输入，并将其编译为位码、汇编器输出或二进制对象文件。像 `llvm-objdump` 和 `llvm-dwarfdump` 这样的工具允许检查目标文件，而像 `llvm-ar` 这样的工具允许从一组目标文件创建静态库，还包括帮助开发 LLVM 本身的工具，例如：`bugpoint` 工具可以帮助找到 LLVM 中崩溃的最小测试用例。`llvm-mc` 可以对机器代码进行操作：该工具可以对机器指令进行汇编和反汇编，这对添加新的指令很有帮助。

LLVM 核心库由 C++ 编写的。此外，还提供了 C 接口和 Go、Ocaml 和 Python 接口。

Polly 项目位于 `polly` 目录中，向 LLVM 添加了另一组优化。它基于一种叫做多面体模型的数学表示，使用这种方法，可以进行复杂的优化，如使用缓存局部优化的循环。MLIR 项目旨在为 LLVM 提供多级中间表示。

MLIR 旨在为 LLVM 提供多级的中间表示。LLVM IR 已经属于底层，并包括源语言的某些信息（这些信息在编译器生成 IR 时丢失了）。MLIR 使 LLVM IR 具有可扩展性，并在特定领域可以捕获该信息，可以在 `mlir` 目录中找到相应的源码。

编译器和工具

名为 Clang(<http://clang.llvm.org/>) 的 C/C++/Objective-C/Object-C++ 编译器是 LLVM 项目的一部分，源码位于 clang 目录中。它提供了一组库，用于从 C、C++、Objective-C 和 Objective-C++ 源码进行词法分析、解析、语义分析和生成 LLVM IR。Clang 是基于这些库的编译器驱动程序。另一个工具是 clang-format，可以根据用户提供的规则格式化 C/C++ 源码。

Clang 的目标是兼容 GCC(GNU C/C++ 编译器) 和 CL(Microsoft C/C++ 编译器)。

C/C++ 的其他工具由同名目录下的 clang-tools-extra 项目提供。值得注意的是 clang-tidy，它是 C/C++ 的 Lint 样式检查器。clang-tidy 使用 clang 库来解析源代码，并使用静态分析检查源代码。与编译器相比，工具可以捕获更多的潜在错误，但会增加运行时间。

Llgo 是一个用于 Go 编程语言的编译器，位于 Llgo 目录下。用 Go 编写的，并使用 LLVM 核心库的 Go 绑定 LLVM 接口。Llgo 的目标是与参考编译器 (<https://golang.org/>) 兼容，但目前支持的架构是 64 位 x86 Linux。该项目似乎没有继续进行维护，并可能在未来删除。

编译器创建的对象文件必须与运行时库链接在一起，以形成可执行文件。这是 lld(<http://lld.llvm.org/>) 的任务，LLVM 链接器位于 lld 目录中。连接器支持 ELF、COFF、Mach-O 和 WebAssembly 格式。

没有调试器的编译器工具集是不完整的！LLVM 调试器名为 lldb(<http://lldb.llvm.org/>)，位于同名的目录中。该接口类似于 GDB、GNU 调试器，并且该工具支持 C、C++ 和 Objective-C。调试器可以扩展，因此可以添加对其他编程语言的支持。

运行时库

除了编译器，运行时库还需要编程语言支持。所有项目都位于同一个目录中：

- compiler-rt 项目提供了独立于编程语言的支持库。它包括泛型函数，例如：可在 32 位 (i386) 机上使用的 64 位除法、各种 sanitizer、模糊库和分析库。
- libunwind 库提供了基于 DWARF 标准的堆栈展开帮助函数。这通常用于 C++ 等语言的异常处理。该库用 C 编写，函数没有绑定到特定的异常处理模型上。
- libcxxabi 库在 libunwind 上实现了 C++ 的异常处理，并为其提供了标准的 C++ 函数。
- libcxx 是 C++ 标准库的实现，包括 iostreams 和 STL。另外，pstl 项目提供了并行版本的 STL 算法。
- libclc 是 OpenCL 的运行时库。OpenCL 是异构并行计算的标准，有助于将计算任务转移到 GPU 上。
- libc 旨在提供一个完整的 C 库。这个项目仍处于早期阶段。
- OpenMP 项目提供对 OpenMP API 的支持。OpenMP 可以帮助多线程编程，例如：可以基于源代码中的注释并行化循环。

尽管这是一个很长的项目列表，但所有项目的结构都是相似的。我们将在下一节中查看统一的目录结构。

LLVM 的项目结构

所有 LLVM 项目都有同意的目录结构。让我们比较一下 LLVM 和 GCC，即 GNU 编译器集合。几十年来，GCC 几乎为您能想到的每一个系统都提供了成熟的编译器。但是，除了编译器，没有任何工具可以用这些代码，原因是它不为重用而设计，而 LLVM 截然不同。

LLVM 的每个功能都有明确的 API 定义，并放在自己的库中。Clang 项目有一个库，可以将 C/C++ 源文件写入令牌流。解析器库将该令牌流转换为抽象语法树 (由库支持)。语义分析、代码生成，甚至编译器驱动程序都作为库提供。众所周知的 Clang 工具，只是一个链接到这些库的应用程序。

这样做的好处很明显: 想要构建一个需要 C++ 文件抽象语法树 (AST) 的工具时，可以重用这些库的功能来构建 AST。不需要语义分析和代码生成，也不需要链接这些库。所有 LLVM 项目都遵循这个原则，包括核心库!

每个项目都有类似的组织。因为 CMake 用于生成构建文件，所以每个项目都用 CMakeLists.txt 来描述项目的构建。如果需要额外的 CMake 模块或支持文件，可以将它们存储在 cmake 子目录中，而现成的模块则放在 cmake/modules 中。

库和工具大多是用 C++ 编写的。源文件放在 lib 目录下，头文件放在 include 目录下。因为一个项目通常由几个库组成，所以 lib 目录中有每个库的目录。如果有必要，还会套娃，例如：在 llvm/lib 目录中有 Target 目录，该目录包含特定于目标的更加底层的操作。除了一些源文件外，每个目标还有子目录，这些子目录会再次编译成库。每个目录都有一个 CMakeLists.txt 文件，该文件描述了如何构建库以及哪些子目录还包含源代码。

include 目录有级别。为了使包含文件的名称唯一，路径名包含项目名称，并且是 include 下的第一个子目录。只有在这个文件夹中，lib 目录的结构才会重复。

应用程序的源码位于 tools 和 utils 目录中。utils 目录中是在编译或测试期间使用的内部应用程序。它们通常不是用户安装的一部分，tools 目录包含用于最终用户的应用程序。这两个目录中，每个应用程序都有自己的子目录。与 lib 目录一样，每个包含 source 的子目录都有 CMakeLists.txt。

编译器必须正确的生成代码，这需要通过测试套件来实现。unittest 目录包含使用 Google Test 框架的单元测试。这主要用于单个函数和无法通过其他方式测试的独立功能。test 目录中是 LIT 测试，这些测试使用 llvm-lit 实用程序执行测试。llvm-lit 扫描文件中的 shell 命令并执行它们。该文件包含用作测试输入的源代码，例如：LLVM IR。文件中嵌入编译命令，由 llvm-lit 执行。然后，在 FileCheck 实用工具的帮助下，验证该步骤的输出。这个程序从文件中读取检查语句，并将它们与另一个文件进行匹配。LIT 测试本身位于 test 目录下的子目录中，对于 lib 的目录结构的遵循不是很严格。

文档 (通常是 reStructuredText) 放在 docs 目录中。如果项目提供了示例，则位于 examples 目录中。

根据项目的需要，还可以有其他目录。值得注意的是，一些提供运行时库的项目将源代码放在 src 目录中，并使用 lib 目录作为库导出定义。compiler-rt 和 libclc 项目包含与体系结构相关的代码，它总是放在以目标体系结构命名的子目录中 (例如，i386 或 ptx)。

总之，提供样例库并具有驱动工具项目的总体结构如下所示:

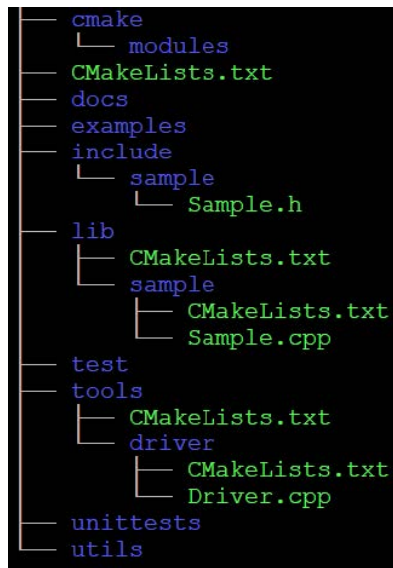


图 2.1 -项目的目录结构

我们自己的项目也将遵循这个结构。

使用 LLVM 创建自己的项目

根据上一节中的信息，您现在可以使用 LLVM 库创建自己的项目。下面几节介绍一种名为 Tiny 的小型语言，这个项目将称为 tinylang。这里定义了这样一个项目的结构。尽管本节中的工具只是一个“Hello, world”，但它的结构与实际编译器所需的所有部分一样。

创建目录结构

第一个问题，tinylang 项目是否应该与 LLVM 一起构建（就像 clang 一样）？还是应该是一个只使用 LLVM 库的独立项目？前一种情况下，还需要决定在哪里创建项目。

首先，假设 tinylang 应该与 LLVM 一起构建。在哪里放置项目有不同的选择。第一种解决方案是在 llvm-projects 目录中为项目创建一个子目录。此目录中的所有项目都将作为构建 LLVM 的一部分使用并构建。在并行项目布局创建之前，这是标准的构建方式，例如：clang。

第二种选择是将 tinylang 项目放在顶层目录中。因为它不是一个正式的 LLVM 项目，所以无需让 CMake 脚本知道。当运行 cmake 时，需要指定 `-DLLVM_ENABLE_PROJECTS=tinylang`，以便在构建中包含项目。

第三种选择是将项目目录放在其他地方，即 llvm-project 目录之外。当然，需要告诉 CMake 关于这个位置的信息。例如，若位置是 `/src/tinylang`，那么需要指定 `-DLLVM_ENABLE_PROJECTS=tinylang -DLLVM_EXTERNAL_TINYLANG_SOURCE_DIR=/src/tinylang`。

如果将项目作为独立项目构建，那么需要找到 LLVM 库。这是在 CMakeLists.txt 中完成的，本节稍后将讨论这个文件。

了解了可能的选择后，哪一个最好呢？使您的项目成为 LLVM 源代码树的一部分有点不灵活。要是您不打算将项目添加到顶层项目列表中，建议使用一个单独的目录。可以在 GitHub 或类似的服务上维护，而不用担心如何与 LLVM 项目同步。正如前面所示，您仍然可以与其他 LLVM 项目一起构建它。

让我们用一个非常简单的库和应用程序创建一个项目。第一步是创建目录布局，假设当前位于克隆 `llvm-project` 目录。使用 `mkdir(Unix)` 或 `md(Windows)` 创建以下目录：

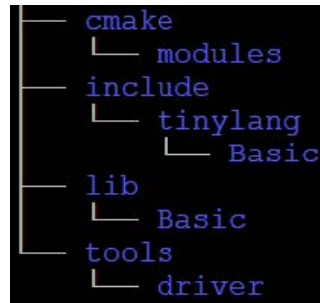


图 2.2 -项目所需的目录

接下来，我们将构建描述和源文件放在这些目录中。

添加 CMake 文件

上一节中确认了代码的基本结构。在 `tinylang` 目录中，按照以下步骤创建一个名为 `CMakeLists.txt` 的文件：

1. 该文件首先使用 `cmake_minimum_required()` 来声明 CMake 的最低版本：

```
cmake_minimum_required(VERSION 3.13.4)
```

2. 下一个语句是 `if()`。如果条件为真，那么项目将独立构建，并且需要额外的设置。该条件使用两个变量，`CMAKE_SOURCE_DIR` 和 `CMAKE_CURRENT_SOURCE_DIR`。`CMAKE_SOURCE_DIR` 变量是在 CMAKE 命令行中给出的顶层源目录。正如在关于目录布局的讨论中所看到的，每个带有源文件的目录都有一个 `CMakeLists.txt`。`CMakeLists.txt` 所位于的当前文件夹记录在 `CMAKE_CURRENT_SOURCE_DIR` 变量中。如果两个变量具有相同的字符串值，那么项目将独立构建。否则，`CMAKE_SOURCE_DIR` 将是 `llvm` 目录：

```
if(CMAKE_SOURCE_DIR STREQUAL CMAKE_CURRENT_SOURCE_DIR)
```

单独设置也很简单，每个 CMake 项目都需要一个名称。这里，将它设置为 `Tinylang`：

```
project(Tinylang)
```

3. 搜索 LLVM 包，将 LLVM 目录添加到 CMake 模块路径中：

```
find_package(LLVM REQUIRED HINTS
    "$LLVM_CMAKE_PATH")
list(APPEND CMAKE_MODULE_PATH $LLVM_DIR)
```

4. 然后，包含了 LLVM 提供的另外三个 CMake 模块。只有使用 Visual Studio 作为构建编译器，并设置正确的运行时库，再次链接时才需要第一个。另外两个模块添加 LLVM 使用的宏，并根据提供的选项配置构建：

```
include(ChooseMSVCCRT)
include(AddLLVM)
include(HandleLLVMOptions)
```

5. 接下来，将 LLVM 头文件的路径添加到 include 搜索路径中。新增两个目录。添加了构建目录中的 include 目录，因为这里保存了自动生成的文件。另一个 include 目录位于源目录中：

```
include_directories("${LLVM_BINARY_DIR}/include"
    "${LLVM_INCLUDE_DIR}")
```

6. 使用 link_directories()，LLVM 库的路径添加到链接器中：

```
link_directories("${LLVM_LIBRARY_DIR}")
```

7. 最后，设置一个标志来表示项目是独立构建的：

```
set(TINYLANG_BUILT_STANDALONE 1)
endif()
```

8. 现在遵循常见的设置。将 cmake/modules 目录添加到 CMake 模块搜索路径中。可以添加自己的 CMake 模块：

```
list(APPEND CMAKE_MODULE_PATH
    "${CMAKE_CURRENT_SOURCE_DIR}/cmake/modules")
```

9. 接下来，检查用户是否正在执行超出构建树的构建。与 LLVM 一样，我们要求用户使用单独的目录来构建项目：

```
if(CMAKE_SOURCE_DIR STREQUAL
    CMAKE_BINARY_DIR AND NOT
    MSVC_IDE)
    message(FATAL_ERROR "In-source builds are not allowed.")
endif()
```

10. tinylang 的版本号通过 configure_file() 命令写到生成文件中。版本号取自 TINYLANG_VERSION_STRING，configure_file() 命令会读取一个输入文件，用 CMake 变量的当前值替换它们，并写入一个输出文件。请注意，输入文件是从源目录读取的，并写入构建目录：


```
set(TINYLANG_VERSION_STRING "0.1")
configure_file($CMAKE_CURRENT_SOURCE_DIR/include/
    tinylang/Basic/Version.inc.in
    $CMAKE_CURRENT_BINARY_DIR/include/tinylang/Basic/Version.inc)
```

11. 接下来，将包含另一个 CMake 模块。AddTinylang 模块有一些辅助功能：

```
include(AddTinylang)
```

12. 接下来是另一个 `include_directories()` 语句。将把我们自己的 `include` 目录添加到搜索路径的开头。独立版本中，需要添加了两个目录：

```
include_directories(BEFORE
    $CMAKE_CURRENT_BINARY_DIR/include
    $CMAKE_CURRENT_SOURCE_DIR/include
)
```

13. 文件的末尾，在 `lib` 和 `tools` 目录其中找到 `CMakeLists.txt`。这个示例应用程序只有 `lib` 和 `tools` 目录下的源文件，因此不需要其他文件。更复杂的项目将添加更多的目录，例如：单元测试：

```
add_subdirectory(lib)
add_subdirectory(tools)
```

这就完成了您的项目描述。

AddTinylang.make 模块位于 `cmake/modules` 目录下。有以下内容：

```

macro(add_tinylang_subdirectory name)
    add_llvm_subdirectory(TINYLANG TOOL $name)
endmacro()

macro(add_tinylang_library name)
    if(BUILD_SHARED_LIBS)
        set(LIBTYPE SHARED)
    else()
        set(LIBTYPE STATIC)
    endif()

    llvm_add_library($name $LIBTYPE $ARGN)
    if(TARGET $name)
        target_link_libraries($name INTERFACE
            $LLVM_COMMON_LIBS)
        install(TARGETS $name
            COMPONENT $name
            LIBRARY DESTINATION lib$LLVM_LIBDIR_SUFFIX
            ARCHIVE DESTINATION lib$LLVM_LIBDIR_SUFFIX
            RUNTIME DESTINATION bin)
    else()
        add_custom_target($name)
    endif()
endmacro()

macro(add_tinylang_executable name)
    add_llvm_executable($name $ARGN )
endmacro()

macro(add_tinylang_tool name)
    add_tinylang_executable($name $ARGN)
    install(TARGETS $name
        RUNTIME DESTINATION bin
        COMPONENT $name)
endmacro()

```

包含该模块，可以使用 `add_tinylang_subdirectory()`、`add_tinylang_library()`、`add_tinylang_executable()` 和 `add_tinylang_tool()` 函数。这些是 LLVM(在 `AddLLVM` 模块中) 提供的函数包装器。`tinylang_subdirectory()` 为构建添加了一个新的源目录。此外，还添加了一个新的 CMake 选项。使

用此选项，用户可以控制是否应该编译目录的内容。使用 `add_tinylang_library()`，可以定义库并安装。`Add_tinylang_executable()` 定义了可执行文件，`Add_tinylang_tool()` 定义了同样安装的可执行文件。

`lib` 目录中，即使没有源代码，也需要 `CMakeLists.txt` 文件，必须包含这个项目库的源目录。打开文本编辑器，将以下内容保存在文件中：

```
add_subdirectory(Basic)
```

大型项目将创建几个库，源代码将放在 `lib` 的子目录中。每个目录都必须添加到 `CMakeLists.txt` 文件中。我们的小项目只有一个名为 `Basic` 的库，所以只需要一行代码。

`Basic` 库只有一个源文件 `Version.cpp`。这个目录中的 `CMakeLists.txt` 文件同样简单：

```
add_tinylang_library(tinylangBasic
    Version.cpp
)
```

定义了一个名为 `tinylangBasic` 的新库，并将编译后的 `Version.cpp` 添加到这个库中。LLVM 选项可以控制这是一个动态库还是静态库。默认情况下，会创建静态库。

在 `tools` 目录中重复相同的步骤。这个文件夹中的 `CMakeLists.txt` 文件几乎和 `lib` 目录中一样简单：

```
create_subdirectory_options(TINYLANG TOOL)
add_tinylang_subdirectory(driver)
```

首先，定义了一个 CMake 选项来控制该目录的内容是否编译。然后添加子目录 `driver`，这一次使用我们自己的模块函数。同样，我们可以控制这个目录是否包含在编译中。

驱动程序目录包含应用的源码 `driver.cpp`。这个目录中的 `CMakeLists.txt` 包含了编译和链接这个应用的所有步骤：

```

set(LLVM_LINK_COMPONENTS
    Support
)

add_tinylang_tool(tinylang
    Driver.cpp
)

target_link_libraries(tinylang
    PRIVATE
    tinylangBasic
)

```

首先，LLVM_LINK_COMPONENTS 变量设置为需要链接的 LLVM 组件列表 (LLVM 组件是一个或多个库的集合)。显然，这取决于工具实现的功能。这里，我们只需要 Support 组件。

使用 add_tinylang_tool() 定义了可安装应用程序。名称是 tinylang, 唯一的源文件是 Driver.cpp。要链接到自己的库，必须使用 target_link_libraries() 来指定。这里，只需要 tinylangBasic。

现在，CMake 所需的文件已经就绪。接下来，需要添加源文件。

创建目录结构

从 include/tinylang/Basic 目录开始。首先，创建 Version.inc.in 模板文件，其中包含配置版本号：

```

1 #define TINYLANG_VERSION_STRING "@TINYLANG_VERSION_STRING@"

```

TINYLANG_VERSION_STRING 周围的 @ 符号表示这是一个 CMake 变量，可以进行内容替换。

Version.h 头文件声明了一个可获取 version 字符串的函数：

```

1 #ifndef TINYLANG_BASIC_VERSION_H
2 #define TINYLANG_BASIC_VERSION_H
3
4 #include "tinylang/Basic/Version.inc"
5 #include <string>
6
7 namespace tinylang {
8     std::string getTinylangVersion();
9 }
10
11 #endif

```

这个函数的实现在 lib/Basic/Version.cpp 文件中。同样很简单：

```

1 #include "tinylang/Basic/Version.h"
2
3 std::string tinylang::getTinylangVersion() {
4     return TINYLANG_VERSION_STRING;
5 }

```

最后，在 tools/driver/Driver.cpp 文件中有应用程序的源代码：

```

1 #include "llvm/Support/InitLLVM.h"
2 #include "llvm/Support/raw_ostream.h"
3 #include "tinylang/Basic/Version.h"
4
5 int main(int argc_, const char **argv_) {
6     llvm::InitLLVM X(argc_, argv_);
7     llvm::outs() << "Hello, I am Tinylang "
8     << tinylang::getTinylangVersion()
9     << "\n";
10 }

```

尽管只是一个友好的工具，但源码使用了 LLVM 功能。调用 `llvm::InitLLVM()` 执行一些基本的初始化。Windows 上，参数会被转换为 Unicode，以便对命令行解析进行统一处理。在应用程序崩溃的情况下（希望不太可能），将安装打印堆栈跟踪处理程序。它输出调用层次结构，从发生崩溃的函数开始。要查看真正的函数名，而不是十六进制地址，需要提供调试符号。

LLVM 不使用 C++ 标准库的 `iostream` 类，它有自己的实现。`llvm::outs()` 是输出流，用来向用户发送消息。

编译 tinylang 应用程序

现在第一个应用程序的所有文件都已就绪，可以编译该应用程序了。概括一下，应该有以下目录和文件：

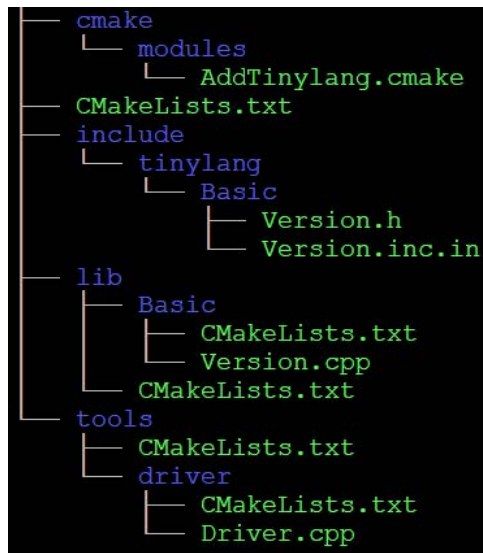


图 2.3 -tinylang 项目的所有目录和文件

如前所述，有几种方法可以构建 tinylang。下面是如何将 tinylang 构建为 LLVM 的一部分：

1. 进入构建目录：

```
$ cd build
```

2. 执行如下 CMake 命令：

```
$ cmake -G Ninja -DCMAKE_BUILD_TYPE=Release \
-DLLVM_EXTERNAL_PROJECTS=tinylang \
-DLLVM_EXTERNAL_TINYLANG_SOURCE_DIR=../tinylang \
-DCMAKE_INSTALL_PREFIX=../llvm-12 \
../llvm-project/llvm
```

通过这个命令，CMake 为 Ninja(-G Ninja) 生成构建文件。构建类型设置为 Release，从而生成优化的二进制文件 (-DCMAKE_BUILD_TYPE=Release)。Tinylang 作为一个外部项目与 LLVM(-DLLVM_EXTERNAL_PROJECTS=Tinylang) 一起构建，源代码位于与构建目录平行的目录中 (-DLLVM_EXTERNAL_TINYLANG_SOURCE_DIR=../Tinylang)。还提供了构建二进制文件的目标目录 (-DCMAKE_INSTALL_PREFIX=../llvm-12)。最后一个参数指定 LLVM 项目目录 (../llvm-project/llvm)。

3. 现在，进行构建和安装：

```
$ ninja
$ ninja install
```

4. 构建安装完成后，../llvm-12 目录会包含 LLVM 和 tinylang 二进制文件。请检查这些应用是否可以运行：


```
$ ../llvm-12/bin/tinylang
```

5. 您应该看到一些友好的消息。请同时检查 Basic 库是否安装:

```
$ ls ../llvm-12/lib/libtinylang*
```

这将显示有一个名为 libtinylangBasic.a 的文件。

当您关注 LLVM 的开发, 并且希望尽快了解 API 的变化时, 使用 LLVM 进行构建非常有用。第 1 章中, 检查了 LLVM 的特定版本。因此, 没有看到对 LLVM 源的修改。

这里, 只构建一次 LLVM, 然后使用已编译的 LLVM 版本, 将 tinylang 作为独立项目进行编译:

1. 重新开始, 再次进入构建目录:

```
$ cd build
```

这一次, CMake 仅用于构建 LLVM:

```
$ cmake -G Ninja -DCMAKE_BUILD_TYPE=Release \  
-DCMAKE_INSTALL_PREFIX=../llvm-12 \  
../llvm-project/llvm
```

2. 与前面的 CMake 命令比较, 除了 tinylang 的参数没了, 其他都一样。
3. 创建和安装 LLVM 与 Ninja:

```
$ ninja  
$ ninja install
```

4. 现在您已经在 llvm-12 目录中安装了一个 LLVM。接下来, 将对 tinylang 项目进行建成。由于它是一个独立的构建, 因此需要新的构建目录:

```
$ cd ..
```

5. 现在创建一个新的 build-tinylang 目录。Unix 上, 可以使用以下命令:

```
$ mkdir build-tinylang
```

Windows 上, 可以使用这个命令:

```
$ md build-tinylang
```

6. 任意一个操作系统上用以下命令输入新目录:

```
$ cd build-tinylang
```

7. 现在运行 CMake 来创建 tinylang 的构建文件。唯一的问题是如何发现 LLVM，因为 CMake 不知道 LLVM 的安装位置。解决方案是指定 LLVMConfig.cmake 的路径。使用 LLVM_DIR 变量指定该 cmake 文件所在的文件夹。命令如下：

```
$ cmake -G Ninja -DCMAKE_BUILD_TYPE=Release \  
-DLLVM_DIR=../llvm-12/lib/cmake/llvm \  
-DCMAKE_INSTALL_PREFIX=../tinylang ../tinylang/
```

8. 安装目录现在也是分开的。构建和安装如下：

```
$ ninja  
$ ninja install
```

9. 在命令完成后，应该运行 ../tinylang/bin/tinylang 应用程序检查应用程序是否正常。

包含 LLVM 的另一种方法

如果不想在项目中使用 CMake，那么需要找出包含文件和库的位置、要链接的库、使用的构建模式等等。此信息由 `llvm-config` 工具提供，该工具位于 LLVM 安装的 `bin` 目录中。假设这个目录包含在 `shell` 搜索路径中，可以运行 `$llvm-config` 查看所有选项。

例如，要让 LLVM 库链接到支持组件 (在前面的示例中使用)，可以运行以下命令：

```
$ llvm-config -libs support
```

输出一行带有库名的代码，包括编译器的链接选项，例如 `-lLLVMSupport`、`-lLLVMDemangle`。显然，这个工具可以很容易地与所选择的构建系统进行集成。

有了项目布局，您就有了一个可扩展到大型项目 (如编译器) 的结构。下一节将介绍另一个基础知识：如何针对不同的目标体系结构进行交叉编译。

针对不同的 CPU 架构

今天，许多小型计算机 (如树莓派) 在使用，只有有限的资源。在这样的计算机上运行编译器通常是不可能的，或者需要太多的运行时间。因此，编译器的一个常见需求是为不同的 CPU 体系结构生成代码，并创建可执行文件的整个过程称为**交叉编译**。前一节中，创建了一个基于 LLVM 库的小型示例应用。现在，我们将使用这个程序，并为不同的目标编译它。

使用交叉编译，涉及到两个系统：编译器在主机系统上运行，并为目标系统生成代码。为了表示这些系统，我们使用了所谓的“三元组表达式”。这是一个配置字符串，通常由 CPU 架构、供应商和操作系统组成。通常还会添加更多关于环境的信息。例如，`x86_64-pc-win32` 用于运行在 64 位 x86 CPU 上的 Windows 系统。CPU 架构是 `x86_64`，`pc` 是通用供应商，`win32` 是操作系统。各部分用连字符连接。在 ARMv8 CPU 上运行的 Linux 系统使用 `aarch64-unknown-linux-gnu` 作为三

元组表达式。aarch64 是 CPU 架构。操作系统是 linux，运行 gnu 环境。没有基于 linux 的系统供应商，所以这一部分是未知的。对于特定目的而言，那些不为人所知或不重要的部分通常会被省略，则 aarch64-linux-gnu 描述了相同的 Linux 系统。

假设您的开发机器在 x86 64 位 CPU 上运行 Linux，并且您希望交叉编译到一个运行 Linux 的 ARMv8 CPU 系统。主机表示为 x86_64-linux-gnu，目标三元组表达式是 aarch64-linux-gnu。不同的系统有不同的特点，应用程序必须以可移植的方式编写，否则失败的原因会很难查找。常见的陷阱如下：

- **字节顺序:** 多字节值存储在内存中的顺序可以不同。
- **指针大小:** 指针的大小随 CPU 架构的不同而不同 (通常为 16 位、32 位或 64 位)。C 类型 int 可能不够大，无法保存指针。
- **类型差异:** 数据类型通常与硬件密切相关。long double 类型可以使用 64 位 (ARM)、80 位 (X86) 或 128 位 (ARMv8)。PowerPC 系统可以对长双精度使用双精度算法，它通过使用两个 64 位双精度值的组合来提供更高的精度。

如果不注意这些要点，那么即使应用在您的主机系统上完美地运行，也可能在目标平台上表现惊人，或者会崩溃。LLVM 库在不同的平台上进行了测试，并包含了针对上述问题的可移植解决方案。

交叉编译时，需要使用以下工具：

- 为目标生成代码的编译器
- 可生成二进制文件的链接器
- 目标的头文件和库

Ubuntu 和 Debian 发行版都有支持交叉编译的软件包。下面的设置中，我们将利用这一点。gcc 和 g++ 编译器、ld 链接器和库都可以作为生成 ARMv8 代码和可执行文件的预编译可执行文件使用。输入以下命令：

```
$ sudo apt install gcc-8-aarch64-linux-gnu \
    g++-8-aarch64-linux-gnu binutils-aarch64-linux-gnu \
    libstdc++-8-dev-arm64-cross
```

新文件安装在 /usr/aarch64-linux-gnu 目录下。directory 目标系统的 (逻辑) 根目录，它包含通常的 bin、lib 和 include 目录，并且交叉编译器 (aarch64-linux-gnu-gcc-8 和 aarch64-linux-gnu-g++-8) 知道这个目录。

在其他系统上交叉编译

如果您的发行版没有附带所需的工具链，那么可以从源代码构建。必须配置 gcc 和 g++ 编译器来生成目标系统的代码，binutils 工具需要处理目标系统的文件。此外，C 和 C++ 库需要用这个工具链来编译。该步骤因所使用的操作系统以及主机和目标体系结构而异。可以通过网页上，搜索 gcc cross-compile <architecture>，从而找到找到相应的指令。

通过这些准备，已经准备好交叉编译示例应用程序 (包括 LLVM 库) 了，除了一个小细节外。

LLVM 在构建过程中使用 tablegen 工具。交叉编译期间，为目标体系结构编译所有内容，包括此工具。可以在第 1 章“安装 LLVM”中使用 llvm-tblgen，也可以只编译这个工具。假设从 GitHub 克隆代码目录下，键入如下命令：

```
$ mkdir build-host
$ cd build-host
$ cmake -G Ninja \
  -DLLVM_TARGETS_TO_BUILD="X86" \
  -DLLVM_ENABLE_ASSERTIONS=ON \
  -DCMAKE_BUILD_TYPE=Release \
  ../llvm-project/llvm
$ ninja llvm-tblgen
$ cd ..
```

这些步骤现在应该很熟悉了。创建并输入构建目录。CMake 命令仅为 X86 目标创建 LLVM 构建文件。为了节省空间和时间，已经完成了发布构建，但支持断言以捕获可能的错误。只有 llvmtblgen 工具是用 Ninja 编译的。

有了 llvm-tblgen 工具，就可以开始交叉编译了。CMake 命令行非常长，因此可能需要将该命令存储在脚本文件中。与之前的版本不同的是，需要提供更多的信息：

```
$ mkdir build-target
$ cd build-target
$ cmake -G Ninja \
  -DCMAKE_CROSSCOMPILING=True \
  -DLLVM_TABLEGEN=../build-host/bin/llvm-tblgen \
  -DLLVM_DEFAULT_TARGET_TRIPLE=aarch64-linux-gnu \
  -DLLVM_TARGET_ARCH=AArch64 \
  -DLLVM_TARGETS_TO_BUILD=AArch64 \
  -DLLVM_ENABLE_ASSERTIONS=ON \
  -DLLVM_EXTERNAL_PROJECTS=tinylang \
  -DLLVM_EXTERNAL_TINYLANG_SOURCE_DIR=../tinylang \
  -DCMAKE_INSTALL_PREFIX=../target-tinylang \
  -DCMAKE_BUILD_TYPE=Release \
  -DCMAKE_C_COMPILER=aarch64-linux-gnu-gcc-8 \
  -DCMAKE_CXX_COMPILER=aarch64-linux-gnu-g++-8 \
  ../llvm-project/llvm
$ ninja
```

同样，创建构建目录并进入。一些 CMake 参数之前没有使用过，需要解释一下：

- CMAKE_CROSSCOMPILING 设置为 ON，则告诉 CMake 我们正在交叉编译。

- LLVM_TABLEGEN 指定 llvm-tblgen 工具要使用的路径。这来自上一个构建。
- LLVM_DEFAULT_TARGET_TRIPLE 是目标架构的三元组表达式。
- LLVM_TARGET_ARCH 使用即时 (JIT) 代码生成，默认为主机的架构。对于交叉编译，必须将其设置为目标架构。
- LLVM_TARGETS_TO_BUILD 是一个目标列表，LLVM 应该包含这些目标的代码生成器。该列表至少应该包括目标体系结构。
- CMAKE_C_COMPILER 和 CMAKE_CXX_COMPILER 指定编译所用的 C 和 C++ 编译器。交叉编译器的二进制文件是用目标三元表达式，CMake 无法自动找到。

使用其他参数，发布版本将请求启用断言进行构建，我们的 tinylang 应用程序将作为 LLVM 的一部分构建 (如上一节所示)。编译过程完成后，可以使用 file 命令检查是否真的为 ARMv8 创建了一个二进制文件。运行 `$ file bin/tinylang`，检查输出是否显示它是 ARM aarch64 架构的 ELF 64 位对象。

使用 clang 进行交叉编译

由于 LLVM 为不同的体系结构生成代码，显然可以使用 clang 进行交叉编译。这里的障碍是，LLVM 不提供所有所需的组件，例如：C 库缺失。因此，必须混合使用 LLVM 和 GNU 工具，因此需要告诉 CMake 更多关于正在使用的环境的信息。--target=<target-triple> (为不同的目标启用代码生成)，--sysroot=<path> (目标的根目录的路径，参见前面)、-I(搜索头文件的路径) 和 -L(搜索库的路径)。在 CMake 运行期间，会编译一个小的应用程序。如果设置有问题，CMake 会报错。这个步骤足以检查您是否有一个正确的工作环境。常见的问题包括选择错误的头文件，由于不同的库名导致的链接失败，以及错误的搜索路径。

交叉编译非常复杂。有了本节的说明，您将能够针对选择的目标架构交叉编译相应的应用程序。

总结

本章中，您了解了作为 LLVM 存储库一部分的项目以及常用的布局。为自己的小型应用程序复制了这种结构，为构建更复杂的应用程序奠定了基础。作为编译器构造的最高原则，您还学习了如何为另一个目标体系结构交叉编译应用程序。

在下一章中，我们将概述示例语言 tinylang。您将了解编译器必须执行的任务，以及 LLVM 库支持哪些功能。

第 3 章 编译器结构

编译技术是计算机科学的一个重要领域，其的高级任务是将源语言翻译成机器代码。通常，这个任务分为两部分：前端和后端。前端主要处理源语言，而后端负责生成机器代码。

本章中，我们将讨论以下主题：

- 编译器的构建块，您将了解编译器中的常用组件。
- 算术表达式语言，将向您介绍一种示例语言。您将学习如何使用语法来定义一种语言。
- 词法分析，将讨论如何实现语言的词法分析器。
- 语法分析，包括如何构造语法解析器。
- 语义分析，将了解如何实现语义检查。
- LLVM 后端代码生成，将讨论如何与 LLVM 后端接口，以及如何将所有阶段聚合在一起，来创建一个完整的编译器。

相关代码

本章的代码文件可在 <https://github.com/PacktPublishing/Learn-LLVM-12/tree/master/Chapter03/calc> 获取。

你可以在视频中找到代码 <https://bit.ly/3nllhED>。

编译器的构建块

上世纪中叶计算机问世后，很快，一种比汇编语言更抽象的语言在编程方面就异军突起了。早在 1957 年，Fortran 作为第一种可用的高级程序设计语言问世。从那时起，成千上万种编程语言被开发出来。事实证明，所有的编译器都必须解决相同的任务，编译器的实现应该根据这些任务进行架构和设计。

抽象的来看，编译器由两部分组成：前端和后端。前端负责特定于语言的任务，读取源文件并计算语义分析表示，通常是带注释的抽象语法树 (AST)。后端从前端的结果创建优化的机器码。区分前端和后端的动机是可重用性。假设前端和后端之间的接口定义得很好，就可以将一个 C 和一个 Modula-2 前端连接到同一个后端。或者，当有一个 x86 后端和一个 Sparc 后端，那么可以将您的 C++ 前端与二者相连。

前端和后端有特定的结构。前端通常执行以下任务：

1. 词法分析器 (Lexical analyzer, 简称 Lexer) 读取源文件并生成一个令牌流。
2. 解析器从令牌流创建一个 AST。
3. 语义分析器向 AST 添加语义信息。
4. 代码生成器从 AST 生成一个中间表示 (IR)。

中间表示是后端接口。后端执行以下任务：

1. 后端在 IR 上执行与目标无关的优化。
2. 然后为 IR 代码选择指令。
3. 之后，对指令执行与目标相关的优化。
4. 最后，产生汇编程序代码或目标文件。

当然，这些指令只是概念上的，实现会有很大的不同。LLVM 核心库将中间表示定义为后端标准接口，其他工具可以使用带注释的 AST，并且 C 的预处理器是专用于 C 的。其可以实现为输出预处理 C 源的应用程序，也可以实现为词法分析器和解析器之间的中间件。某些情况下，AST 不能显式构造。如果要实现的语言不是太复杂，可以组合解析器和语义分析器。然后在解析时生成代码，即使给定的编程语言实现没有显式地命名这些组件。不过，以上的任务是必须要完成的。

在下面的小节中，我们将为表达式语言构造一个编译器，该语言从其输入生成 LLVM IR。LLVM 静态编译器 llc 表示后端，可以使用它将 IR 编译成目标代码。这一切都要从定义一种语言开始。

算术表达式语言

算术表达式是每一种编程语言的一部分。下面是一个叫做 calc 的算术表达式计算语言的例子。calc 表达式会编译到一个应用程序中，计算以下表达式：

```
with a, b: a * (4 + b)
```

表达式中使用的变量必须用 with 关键字声明。这个程序可以编译成应用程序，它向用户询问 a 和 b 变量的值并打印结果。

简单的示例总是受欢迎的，但作为编译器作者，需要比这更全面的规范来实现和测试。编程语言语法的载体是语法。

指定编程语言语法的形式

语言的元素，如：关键字、标识符、字符串、数字和操作符，称为标记。其实，程序是一个标记序列，语法指定哪些序列是有效的。

通常，语法是用扩展的巴科斯-诺尔范式 (EBNF) 编写的。其中一个语法规则是有一个左手边和一个右手边。左边只是一个叫做非终结符的符号，右手边包括用于替代和重复的非终结符、标记和元符号。来看看 calc 语言的语法：

```
calc : ("with" ident ("," ident)* ":")? expr ;  
expr : term (( "+" | "-" ) term)* ;  
term : factor (( "*" | "/" ) factor)* ;  
factor : ident | number | "(" expr ")" ;  
ident : ([a-zA-Z])+ ;  
number : ([0-9])+ ;
```

第一行中，calc 是非终止符。如果没有特别说明，则语法的第一个非结束符是开始符号。冒号，:，是规则左边和右边之间的分隔符。，"，" 和":" 是表示该字符串的标记。括号用于分组，组是可选的，也可以重复。右括号后的问号? 表示可选组。星号 * 表示零次或多次重复，而加号 + 表示一次或多次重复。ident 和 expr 是非终止符。每一个，都对应另一个规则。分号; 表示规则的结束。第二行中的管道 | 表示一个替代方案。最后，最后两行中的括号 [] 表示字符类。有效字符写在括号内，例如：[a-zA-Z] 字符类匹配一个大写字母或小写字母，([a-zA-Z])+ 匹配这些字母中的一个或多个。这相当于正则表达式。

语法对于编译器作者的帮助

首先，定义了所有标记，这是创建词法分析器所需要的，语法规则可以翻译成解析器。当然，如果出现关于解析器是否正确工作的问题，那么语法可以作为一个很好的规范。

然而，语法并不是定义编程语言的所有方面，语法的意义 (语义) 也必须定义。为此目的也开发了语法形式，类似于最初引入该语言，通常在纯文本中指定。

掌握了这些知识后，接下来的两节将展示词法分析如何将输入转换为标记序列，以及如何用 C++ 编写语法，从而进行语法分析。

语法分析

正如上一节的示例中所看到的，编程语言由许多元素组成，例如：关键字、标识符、数字、操作符等。词法分析的任务是获取文本输入并从中创建标记序列。calc 语言由 with、:、+、-、*、/、(, and) 标记和 ([a- za -z])+(标识符) 和 ([0-9])+(一个数字) 正则表达式组成。为了方便处理，为每个令牌分配了索引 (数字)。

手写的词法分析器

词法分析程序的实现通常称为 Lexer。创建一个名为 Lexer.h 的头文件，并开始定义 Token:

```
1 #ifndef LEXER_H
2 #define LEXER_H
3
4 #include "llvm/ADT/StringRef.h"
5 #include "llvm/Support/MemoryBuffer.h"
```

llvm::MemoryBuffer 类提供对一个内存块的只读访问，该内存块中填充了文件的内容。请求时，末尾的零字符 ('\x00') 添加到缓冲区的末尾。使用这个特性来读取整个缓冲区，而不检查每次访问缓冲区的长度。llvm::StringRef 类封装了一个指向 C 字符串及其长度的指针。因为长度已存储的，所以字符串不需要像普通的 C 字符串那样以 0 字符结束 ('\x00')。这允许 StringRef 的实例指向 MemoryBuffer 管理的内存。让我们更详细地了解一下：

1. 首先，令牌类包含前面提到的令牌编号 (枚举):

```
1 class Lexer;
2
3 class Token {
4     friend class Lexer;
5
6 public:
7     enum TokenKind : unsigned short {
8         eoi, unknown, ident, number, comma, colon, plus,
9         minus, star, slash, l_paren, r_paren, KW_with
10    };
```

除了为每个令牌定义成员外，我添加了两个额外的值: `eoi` 和 `unknown`。 `eoi` 表示输入的结束，如果处理完输入的所有字符，则返回 `eoi`。 `unknown` 用于词汇错误的情况，例如: `#` 不是语言标识，所以它会映射为 `unknown`。

- 除了枚举之外，该类还有一个成员 `Text`，指向令牌文本的开头 (使用前面提到的 `StringRef` 类):

```
1 private:
2     TokenKind Kind;
3     llvm::StringRef Text;
4
5 public:
6     TokenKind getKind() const { return Kind; }
7     llvm::StringRef getText() const { return Text; }
```

对于语义处理，知道标识符的名称很重要。

- `is()` 和 `isOneOf()` 方法用于测试令牌是否属于某种类型。 `isOneOf()` 使用可变参数模板，允许可变数量的参数:

```
1     bool is(TokenKind K) const { return Kind == K; }
2     bool isOneOf(TokenKind K1, TokenKind K2) const {
3         return is(K1) || is(K2);
4     }
5     template <typename... Ts>
6     bool isOneOf(TokenKind K1, TokenKind K2, Ts... Ks)
7     const {
8         return is(K1) || isOneOf(K2, Ks...);
9     }
10 };
```

- `Lexer` 类本身也有一个简单接口:

```
1 class Lexer {
2     const char *BufferStart;
3     const char *BufferPtr;
4     public:
5     Lexer(const llvm::StringRef &Buffer) {
6         BufferStart = Buffer.begin();
7         BufferPtr = BufferStart;
8     }
9     void next(Token &token);
10    private:
11    void formToken(Token &Result, const char *TokEnd,
12        Token::TokenKind Kind);
```

```

13 };
14 #endif

```

除了构造函数，公共接口只包含 `next()`，返回下一个令牌。该方法的作用类似于迭代器，总会指向下一个可用的令牌。该类的唯一成员是指向输入开头和下一个未处理字符的指针，并且假设缓冲区以 0 结尾 (类似于 C 字符串)。

5. 让我们在 `Lexer.cpp` 文件中实现 `Lexer` 类。从一些辅助函数开始，可以帮助分类字符:

```

1  #include "Lexer.h"
2
3  namespace charinfo {
4  LLVM_READNONE inline bool isWhitespace(char c) {
5      return c == ' ' || c == '\t' || c == '\f' ||
6         c == '\v' ||
7         c == '\r' || c == '\n';
8  }
9
10 LLVM_READNONE inline bool isDigit(char c) {
11     return c >= '0' && c <= '9';
12 }
13 LLVM_READNONE inline bool isLetter(char c) {
14     return (c >= 'a' && c <= 'z') ||
15         (c >= 'A' && c <= 'Z');
16 }
17 }

```

这些函数让条件判断的可读性更好。

Note

不使用 `<cctype>` 标准库头提供的函数，有两个原因。首先，这些函数根据环境中定义的语境来改变行为，例如：如果语境是德语，那么德语 umlauts 可以归类为字母。这在编译器中通常是不需要的。第二，由于函数的参数类型是 `int`，必须从 `char` 类型进行转换。这种转换的结果取决于 `char` 是当作有符号还是无符号类型，这会导致可移植性问题。

6. 上一节的语法中，我们知道了该语言的所有标记。但是语法没有定义需要忽略的字符，例如：空格或新行只会增加空格，并且经常被忽略。`next()` 方法首先要跳过这些字符:

```

1 void Lexer::next(Token &token) {
2     while (*BufferPtr &&
3         charinfo::isWhitespace(*BufferPtr)) {
4         ++BufferPtr;

```

```
5     }
```

7. 接下来，确保仍然有字符需要处理：

```
1     if (!*BufferPtr) {  
2         token.Kind = Token::eoi;  
3         return;  
4     }
```

至少要处理一个字符。

8. 因此，首先检查字符是小写还是大写。本例中，令牌要么是标识符，要么是 with 关键字。因为标识符的正则表达式也匹配关键字，所以常见的解决方案是收集正则表达式匹配的字符，并检查字符串是否是关键字：

```
1     if (charinfo::isLetter(*BufferPtr)) {  
2         const char *end = BufferPtr + 1;  
3         while (charinfo::isLetter(*end))  
4             ++end;  
5         llvm::StringRef Name(BufferPtr, end - BufferPtr);  
6         Token::TokenKind kind =  
7             Name == "with" ? Token::KW_with : Token::ident;  
8         formToken(token, end, kind);  
9         return;  
10    }
```

formToken() 方法用于填充令牌。

9. 接下来，我们检查数字。下面的代码与前面的代码非常相似：

```
1     else if (charinfo::isDigit(*BufferPtr)) {  
2         const char *end = BufferPtr + 1;  
3         while (charinfo::isDigit(*end))  
4             ++end;  
5         formToken(token, end, Token::number);  
6         return;  
7     }
```

10. 现在，只剩下由固定字符串定义的标记，这很容易用开关来完成。因为所有这些标记都只有一个字符，可以使用 CASE 宏来减少输入：

```
1     else {  
2         switch (*BufferPtr) {  
3 #define CASE(ch, tok) \  
4 case ch: formToken(token, BufferPtr + 1, tok); break  
5 CASE('+', Token::plus);
```

```

6 CASE('-', Token::minus);
7 CASE('*', Token::star);
8 CASE('/', Token::slash);
9 CASE('(', Token::Token::l_paren);
10 CASE(')', Token::Token::r_paren);
11 CASE(':', Token::Token::colon);
12 CASE(',', Token::Token::comma);
13 #undef CASE

```

11. 最后，我们需要检查“意外”的字符:

```

1     default:
2         formToken(token, BufferPtr + 1, Token::unknown);
3     }
4     return;
5 }
6 }

```

现在，只缺少 formToken() 了。

12. 这个 helper 方法填充 Token 实例的成员，并更新指向下一个未处理字符:

```

1 void Lexer::formToken(Token &Tok, const char *TokEnd,
2                          Token::TokenKind Kind) {
3     Tok.Kind = Kind;
4     Tok.Text = llvm::StringRef(BufferPtr, TokEnd -
5                                BufferPtr);
6     BufferPtr = TokEnd;
7 }

```

下一节中，我们将了解如何构造语法分析的解析器。

语法分析由解析器完成，下面我们将实现解析器，它的基础是前面几节中的语法和词法表。解析过程的结果是一个称为抽象语法树 (AST) 的动态数据结构。AST 是输入的一种非常浓缩的表现形式，非常适合进行语义分析。首先，我们将实现解析器。之后，再一起来了解一下 AST。

手写的解析器

解析器的接口定义在 parser.h 头文件中:

```

1 #ifndef PARSER_H
2 #define PARSER_H
3
4 #include "AST.h"
5 #include "Lexer.h"
6 #include "llvm/Support/raw_ostream.h"

```

AST.h 头文件声明了 AST 的接口，将在后面展示。LLVM 的编码指南禁止使用 `<iostream>`，所以必须包含相同功能的 LLVM 头文件。需要使用输出流输出一个错误信息，让我们更详细地了解一下：

1. 首先，Parser 类声明了一些私有成员：

```
1 class Parser {
2     Lexer &Lex;
3     Token Tok;
4     bool HasError;
```

Lex 和 Tok 是上一节中类的实例。Tok 存储下一个令牌 (前置)，而 Lex 用于检索下一个令牌。HasError 标志指示是否检测到错误。

2. 处理令牌的两个方法：

```
1 void error() {
2     llvm::errs() << "Unexpected: " << Tok.getText()
3     << "\n";
4     HasError = true;
5 }
6
7 void advance() { Lex.next(Tok); }
8
9 bool expect(Token::TokenKind Kind) {
10     if (Tok.getKind() != Kind) {
11         error();
12         return true;
13     }
14     return false;
15 }
16
17 bool consume(Token::TokenKind Kind) {
18     if (expect(Kind))
19         return true;
20     advance();
21     return false;
22 }
```

advance() 从词法分析器中检索下一个令牌。在 expect() 中测试，并提前检查是否为预期的类型，如果不是，则发出错误消息。最后，如果是预期的类型，consume() 将检索下一个令牌。如果发出错误消息，则将 HasError 标志设置为 true。

3. 对于语法中的每个非终结符，声明一个解析规则的方法：
-

```

1  AST *parseCalc();
2  Expr *parseExpr();
3  Expr *parseTerm();
4  Expr *parseFactor();

```

Note

没有识别和编号的方法。这些规则只返回令牌，并替换相应的令牌。

- 下面是公共接口。构造函数初始化所有成员并从解析器中检索第一个令牌:

```

1  public:
2  Parser(Lexer &Lex) : Lex(Lex), HasError(false) {
3      advance();
4  }

```

- 获取错误标志的值:

```

1  bool hasError() { return HasError; }

```

- 最后，parse() 是解析的主要入口点:

```

1  AST *parse();
2  };
3  #endif

```

下一节中，我们将学习如何实现解析器。

解析器的实现

深入分析解析器的实现:

- 解析器的实现可以在 parser.cpp 文件中找到，并以 parse() 为入口:

```

1  #include "Parser.h"
2
3  AST *Parser::parse() {
4      AST *Res = parseCalc();
5      expect(Token::eoi);
6      return Res;
7  }

```

parse() 的要点是处理整个输入。还记得第一部分中的解析示例添加了一个特殊符号来表示输入的结束吗? 在这里检查一下。

- parseCalc() 实现相应的规则。因为其他解析方法遵循相同的模式，所以值得仔细研究一下。回顾一下第一节中的规则:


```
1 calc : ("with" ident ("," ident)* ":")? expr ;
```

3. 该方法首先声明一些局部变量:

```
1 AST *Parser::parseCalc() {  
2     Expr *E;  
3     llvm::SmallVector<llvm::StringRef, 8> Vars;
```

4. 要做的第一个决定是是否必须解析可选组。组以 with 开始, 因此将令牌与以下值进行比较:

```
1     if (Tok.is(Token::KW_with)) {  
2         advance();
```

5. 接下来, 等待一个标识符:

```
1     if (expect(Token::ident))  
2         goto _error;  
3     Vars.push_back(Tok.getText());  
4     advance();
```

如果有标识符, 则将其保存在 Vars 中。否则, 为一个语法错误, 将被单独处理。

6. 现在的语法中, 后面跟着一个重复组, 它要解析更多用逗号分隔的标识符:

```
1     while (Tok.is(Token::comma)) {  
2         advance();  
3         if (expect(Token::ident))  
4             goto _error;  
5         Vars.push_back(Tok.getText());  
6         advance();  
7     }
```

重复组以 with 开始, 令牌的测试成为 while 循环的条件, 实现零或多次重复。循环内的标识符会和之前的处理一样。

7. 最后, 可选组需要在末尾加冒号:

```
1     if (consume(Token::colon))  
2         goto _error;  
3 }
```

8. 现在, 对 expr 规则进行解析:

```
1     E = parseExpr();
```

9. 通过调用, 规则已经成功解析。收集到的信息可以用于创建 AST 节点:

```
1     if (Vars.empty()) return E;  
2     else return new WithDecl(Vars, E);
```

现在，只缺少错误处理代码了。检测语法错误很容易，但从中错误中恢复却异常复杂。这里，使用了一种称为“恐慌模式”的简单方法。

恐慌模式下，将从令牌流中删除令牌，直到找到解析器可以使用的令牌才继续工作。大多数编程语言都有表示结束的符号，例如：C++，可以使用;(语句的结束) 或}(块的结束)。

另一方面，错误可能是正在寻找的符号不见了。这种情况下，许多标识可能在解析器继续之前就删除了，这并不像听起来那么糟糕。现今，编译器的速度非常重要。在出现错误的情况下，开发人员可以查看第一个错误消息，修复它，并重新启动编译器。这与使用穿孔卡片不同，在穿孔卡片中需要获取尽可能多的错误消息，因为等编译器运行完成就是第二天了。

错误处理

不使用一些任意的标记，而使用另一组标记。对于每个非终止符，会有一系列标识在规则中遵循这个规则。

1. calc 中，只有输入的结尾跟在这个非终止符之后:

```
1 _error:
2     while (!Tok.is(Token::eoi))
3         advance();
4     return nullptr;
5 }
```

2. 其他解析方法也以类似的方式构造。parseExpr() 是对 expr 规则的翻译:

```
1 Expr *Parser::parseExpr() {
2     Expr *Left = parseTerm();
3     while (Tok.isOneOf(Token::plus, Token::minus)) {
4         BinaryOp::Operator Op =
5             Tok.is(Token::plus) ? BinaryOp::Plus :
6                                     BinaryOp::Minus;
7         advance();
8         Expr *Right = parseTerm();
9         Left = new BinaryOp(Op, Left, Right);
10    }
11    return Left;
12 }
```

规则内的重复组会转换为 while 循环。注意，isOneOf() 的使用简化了对几个令牌的检查。

3. 术语规则的编码看起来都一样:

```
1 Expr *Parser::parseTerm() {
2     Expr *Left = parseFactor();
3     while (Tok.isOneOf(Token::star, Token::slash)) {
4         BinaryOp::Operator Op =
```

```

5     Tok.is(Token::star) ? BinaryOp::Mul :
6     BinaryOp::Div;
7     advance();
8     Expr *Right = parseFactor();
9     Left = new BinaryOp(Op, Left, Right);
10 }
11 return Left;
12 }

```

这个方法与 `parseExpr()` 非常地相似，可能会想将它们组合成一个。在语法中，可以有一条规则来处理乘法运算符和加法运算符。使用两个规则而不是一个规则的好处是，运算符的优先级与计算的数学顺序非常吻合。如果将两个规则结合起来，则需要其他地方找出计算顺序。

4. 最后，需要实现 `factor` 规则：

```

1 Expr *Parser::parseFactor() {
2     Expr *Res = nullptr;
3     switch (Tok.getKind()) {
4         case Token::number:
5             Res = new Factor(Factor::Number, Tok.getText());
6             advance(); break;

```

这里使用 `switch` 语句似乎更合适，而不是使用 `if` 和 `else if`，因为每个选项都只从一个令牌开始。通常，应该考虑更偏爱哪种翻译模式。如果以后需要改变解析方法，那么在每个方法都有相同的语法规则实现方式时，就是一种优势。

5. 如果使用 `switch` 语句，在默认情况下会进行错误处理：

```

1     case Token::ident:
2         Res = new Factor(Factor::Ident, Tok.getText());
3         advance(); break;
4     case Token::l_paren:
5         advance();
6         Res = parseExpr();
7         if (!consume(Token::r_paren)) break;
8     default:
9         if (!Res) error();

```

我们在这里避免发出错误消息。

6. 如果括号表达式中有语法错误，则会发出错误消息。这里可以避免第二个错误消息触发：

```

1     while (!Tok.isOneOf(Token::r_paren, Token::star,
2         Token::plus, Token::minus,
3         Token::slash, Token::eoi))
4         advance();

```

```
5     }  
6     return Res;  
7 }
```

这很简单，不是吗？当您记住了使用的模式，就会发现基于语法规则编写解析器很刻板。这种类型的解析器称为递归下降解析器。

递归下降解析器不能适配所有语法

语法必须满足某些条件才能适合构造递归下降解析器。这类语法叫做 LL(1)。事实上，您可以在网上找到的大多数语法都不属于这一类语法。大多数关于编译器构造理论的书籍都解释了这一原因。关于这个主题的经典书籍是所谓的“龙书”，由 Aho, Lam, Sethi 和 Ullman 编写的《编译器：原理，技术和工具》。

抽象语法树

解析的结果是 AST，是输入程序的另一种紧凑表示形式。它捕获了重要的信息。许多编程语言都有作为分隔符的符号，而这些符号没有意义。例如，在 C++ 中，分号; 表示单个语句的结束。当然，这些信息对解析器很重要。当我们将语句转换为内存中的表示，分号就可以删除了。

如果您查看示例表达式语言的第一条规则，那么很明显 with 关键字、逗号、顿号和冒号：对程序的意义来说并不真正重要。重要的是可以在表达式中使用的声明变量列表。结果只需要两个类来记录信息：Factor 保存数字或标识符，BinaryOp 保存算术运算符和表达式的左右两边，WithDecl 存储声明的变量和表达式的列表。AST 和 Expr 仅用于创建公共类的层次结构。

除了来自解析过的输入的信息外，还支持在使用访问者模式时进行树形遍历。这些都在 AST.h 头文件中：

1. 从访问者接口开始：

```
1 #ifndef AST_H  
2 #define AST_H  
3  
4 #include "llvm/ADT/SmallVector.h"  
5 #include "llvm/ADT/StringRef.h"  
6  
7 class AST;  
8 class Expr;  
9 class Factor;  
10 class BinaryOp;  
11 class WithDecl;  
12  
13 class ASTVisitor {  
14 public:
```

```

15 virtual void visit(AST &){};
16 virtual void visit(Expr &){};
17 virtual void visit(Factor &) = 0;
18 virtual void visit(BinaryOp &) = 0;
19 virtual void visit(WithDecl &) = 0;
20 };

```

访问者模式需要知道它必须访问的每个类。因为每个类也引用访问器，所以在文件的顶部声明所有类。请注意 AST 和 Expr 的 visit() 方法有一个默认实现，它什么也不做。

2. AST 类是层次结构的根类:

```

1 class AST {
2 public:
3     virtual ~AST() {}
4     virtual void accept(ASTVisitor &V) = 0;
5 };

```

3. 类似地, Expr 是与表达式相关的 AST 根类:

```

1 class Expr : public AST {
2 public:
3     Expr() {}
4 };

```

4. Factor 类存储一个数字或变量名:

```

1 class Factor : public Expr {
2 public:
3     enum ValueKind { Ident, Number };
4
5 private:
6     ValueKind Kind;
7     llvm::StringRef Val;
8
9 public:
10    Factor(ValueKind Kind, llvm::StringRef Val)
11        : Kind(Kind), Val(Val) {}
12    ValueKind getKind() { return Kind; }
13    llvm::StringRef getVal() { return Val; }
14    virtual void accept(ASTVisitor &V) override {
15        V.visit(*this);
16    }
17 };

```

本例中, 数字和变量的处理方法几乎相同, 因此可以只创建一个 AST 节点类来表示它们, Kind 成员告诉我们实例代表哪一种情况。在更复杂的语言中, 通常希望有不同的 AST 类, 例如: 用于数字的 NumberLiteral 类和用于变量引用的 VariableAccess 类。

5. BinaryOp 类保存了计算表达式所需的数据:

```
1 class BinaryOp : public Expr {
2 public:
3     enum Operator { Plus, Minus, Mul, Div };
4
5 private:
6     Expr *Left;
7     Expr *Right;
8     Operator Op;
9
10 public:
11     BinaryOp(Operator Op, Expr *L, Expr *R)
12         : Op(Op), Left(L), Right(R) {}
13     Expr *getLeft() { return Left; }
14     Expr *getRight() { return Right; }
15     Operator getOperator() { return Op; }
16     virtual void accept(ASTVisitor &V) override {
17         V.visit(*this);
18     }
19 };
```

与解析器相比, BinaryOp 类没有区分乘法运算符和加法运算符。操作符的优先级隐含在树型结构中。

6. 最后, WithDecl 存储声明的变量和表达式:

```
1 class WithDecl : public AST {
2     using VarVector =
3         llvm::SmallVector<llvm::StringRef, 8>;
4     VarVector Vars;
5     Expr *E;
6
7 public:
8     WithDecl(llvm::SmallVector<llvm::StringRef, 8> Vars,
9             Expr *E)
10         : Vars(Vars), E(E) {}
11     VarVector::const_iterator begin()
12         { return Vars.begin(); }
13     VarVector::const_iterator end() { return Vars.end(); }
```

```

14 Expr *getExpr() { return E; }
15 virtual void accept(ASTVisitor &V) override {
16     V.visit(*this);
17 }
18 };
19 #endif

```

AST 是在解析期间构造的。语义分析检查树是否符合语言的含义 (例如, 声明使用的变量), 并可能扩充树。之后, 使用树生成代码。

语义分析

语义分析器遍历 AST 并检查语言的各种语义规则, 例如: 在使用变量之前必须声明变量, 或者变量的类型必须在表达式中兼容。如果发现可以改进的情况, 还可以输出警告。对于表达式语言, 语义分析器必须检查每个使用的变量是否声明, 这是语言所需。一个可能的扩展 (这里不实现) 是在未使用声明的变量时输出警告消息。

语义分析器在 Sema 类中实现, 语义分析由 semantic() 执行。下面是完整的 Sema.h 头文件:

```

1 #ifndef SEMA_H
2 #define SEMA_H
3
4 #include "AST.h"
5 #include "Lexer.h"
6
7 class Sema {
8     public:
9     bool semantic(AST *Tree);
10 };
11 #endif

```

实现在 Sema.cpp 文件中。有趣的部分是语义分析, 并使用访问者实现。其基本思想是, 每个声明变量的名称存储在一个集合中。创建时, 可以检查每个命名是否唯一, 然后检查命名是否在集合中:

```

1 #include "Sema.h"
2 #include "llvm/ADT/StringSet.h"
3 namespace {
4     class DeclCheck : public ASTVisitor {
5         llvm::StringSet<> Scope;
6         bool HasError;
7
8         enum ErrorType { Twice, Not };

```

```

9     void error(ErrorType ET, llvm::StringRef V) {
10         llvm::errs() << "Variable " << V << " "
11             << (ET == Twice ? "already" : "not")
12             << " declared\n";
13         HasError = true;
14     }
15 public:
16     DeclCheck() : HasError(false) {}
17
18     bool hasError() { return HasError; }

```

与 Parser 类中一样，使用标记来指示发生了错误，这些名字存储在 Scope 的集合中。在持有一个变量名的 Factor 节点中，我们检查该变量名是否在这个集合中。

```

1     virtual void visit(Factor &Node) override {
2         if (Node.getKind() == Factor::Ident) {
3             if (Scope.find(Node.getVal()) == Scope.end())
4                 error(Not, Node.getVal());
5         }
6     };

```

对于 BinaryOp 节点，只需要检查两边是否存在，并且是否已经访问过：

```

1     virtual void visit(BinaryOp &Node) override {
2         if (Node.getLeft())
3             Node.getLeft()->accept(*this);
4         else
5             HasError = true;
6         if (Node.getRight())
7             Node.getRight()->accept(*this);
8         else
9             HasError = true;
10    };

```

在 WithDecl 节点中，填充集合并开始遍历表达式：

```

1     virtual void visit(WithDecl &Node) override {
2         for (auto I = Node.begin(), E = Node.end(); I != E;
3             ++I) {
4             if (!Scope.insert(*I).second)
5                 error(Twice, *I);
6         }
7         if (Node.getExpr())

```



```

8     Node.getExpr()->accept(*this);
9     else
10        HasError = true;
11    };
12 };
13 }

```

semantic() 方法会遍历树，并返回错误标志:

```

1 bool Sema::semantic(AST *Tree) {
2     if (!Tree)
3         return false;
4     DeclCheck Check;
5     Tree->accept(Check);
6     return Check.hasError();
7 }

```

如果没有使用声明的变量，也可以打印警告消息。这算是给读者留的课后作业。如果语义分析没有出现错误，那么就可以使用 AST 生成 LLVM IR。我们将在下一节中完成这项工作。

使用 LLVM 后端编译器生成代码

后端的任务是从模块的 IR 中创建优化的机器码。IR 是后端的接口，可以使用 C++ 接口或文本形式创建。同样，IR 由 AST 生成。

LLVM IR 的文本表示

尝试生成 LLVM IR 之前，需要了解我们想要生成的东西。对于表达式语言的例子，计划如下。

1. 询问每个变量的值。
2. 计算表达式的值。
3. 打印结果。

为了要求用户提供一个变量的值并打印出结果，使用了两个库函数，calc_read() 和 calc_write()。对于 *with a: 3*a* 表达式，生成的 IR 如下：

1. 标准库函数必须声明，语法类似于 C。函数名之前的类型是返回类型。由圆括号包围的类型名是参数类型。声明可以出现在文件的任何地方：

```

declare i32 @calc_read(i8*)
declare void @calc_write(i32)

```

2. calc_read() 函数将变量名作为参数。下面的定义了一个常数，持有 a 和空字节，空字节在 C 语言中作为字符串的结束符。

```
@a.str = private constant [2 x i8] c"a\00"
```

3. 接下来是 `main()` 函数。省略了参数的名字，因为没使用到。像 C 语言一样，函数的主体置于大括号中。

```
define i32 @main(i32, i8**) {
```

4. 每个基本块都必须有一个标签。因为这是函数的第一个基本块，我们把它命名为 `entry`。

```
entry:
```

5. 调用 `calc_read()` 函数来读取 `a` 变量的值。嵌套使用 `getelementptr` 指令进行了索引计算，以计算指向字符串常量第一个元素的指针。该函数的结果分配给未命名的 `%2` 变量。

```
%2 = call i32 @calc_read(i8* getelementptr inbounds  
    ([2 x i8], [2 x i8]* @a.str, i32 0, i32 0))
```

6. 接下来，该变量乘以 3。

```
%3 = mul nsw i32 3, %2
```

7. 结果通过 `calc_write()` 函数打印结果到控制台。

```
call void @calc_write(i32 %3)
```

8. 最后，`main()` 函数返回 0，表示成功执行。

```
ret i32 0  
}
```

LLVM IR 中的每个值都是类型化的，`i32` 表示 32 位的整数类型，`i8*` 表示一个字节的指针。IR 代码非常易读 (除了 `getelementptr` 操作，这将在第 5 章，IR 生成的基础知识中详细解释)。既然现在已经清楚了 IR 的样子，就使用 AST 生成它吧。

使用 AST 生成 IR

该接口在 `CodeGen.h` 头文件中提供：

```
1 #ifndef CODEGEN_H  
2 #define CODEGEN_H  
3  
4 #include "AST.h"  
5
```

```

6 class CodeGen
7 {
8 public:
9     void compile(AST *Tree);
10 };
11 #endif

```

因为 AST 包含了语义分析阶段的信息，所以基本思路是用访问者来遍历 AST。CodeGen.cpp 文件的实现方式如下：

1. 所需的包括在文件的顶端：

```

1 #include "CodeGen.h"
2 #include "llvm/ADT/StringMap.h"
3 #include "llvm/IR/IRBuilder.h"
4 #include "llvm/IR/LLVMContext.h"
5 #include "llvm/Support/raw_ostream.h"

```

2. LLVM 库的命名空间用于查找名称：

```

1 using namespace llvm;

```

3. 首先，在访问器中声明一些私有成员。每个编译单元在 LLVM 中由模块类表示，访问者有一个指向模块调用的指针 M。为了便于生成 IR，使用生成器 (IRBuilder<> 类型)。LLVM 用类的层次结构来表示 IR 中的类型，可以在 LLVM 的上下文中查找基本类型的实例，如 i32。这些基本类型会经常使用。为了避免重复查找，需要缓存类型实例，可以是 VoidTy、Int32Ty、Int8PtrTy、Int8PtrPtrTy 或 Int32Zero。V 是当前的计算值，通过树形遍历更新。最后，nameMap 将一个变量名映射为由 calc_read() 函数返回的值：

```

1 namespace {
2 class ToIRVisitor : public ASTVisitor {
3     Module *M;
4     IRBuilder<> Builder;
5     Type *VoidTy;
6     Type *Int32Ty;
7     Type *Int8PtrTy;
8     Type *Int8PtrPtrTy;
9     Constant *Int32Zero;
10    Value *V;
11    StringMap<Value *> nameMap;

```

4. 构造函数初始化了所有成员：

```

1 public:
2     ToIRVisitor(Module *M) : M(M), Builder(M->getContext())

```

```

3  {
4      VoidTy = Type::getVoidTy(M->getContext());
5      Int32Ty = Type::getInt32Ty(M->getContext());
6      Int8PtrTy = Type::getInt8PtrTy(M->getContext());
7      Int8PtrPtrTy = Int8PtrTy->getPointerTo();
8      Int32Zero = ConstantInt::get(Int32Ty, 0, true);
9  }

```

5. 对于每个函数，都必须创建一个 FunctionType 实例。在 C++ 中，这就是一个函数原型。函数本身是用 function 实例定义的。首先，run() 方法定义了 LLVM IR 中的 main() 函数：

```

1  void run(AST *Tree) {
2      FunctionType *MainFty = FunctionType::get(
3          Int32Ty, {Int32Ty, Int8PtrPtrTy}, false);
4      Function *MainFn = Function::Create(
5          MainFty, GlobalValue::ExternalLinkage,
6          "main", M);

```

6. 然后，创建带有入口标签的 BB 基本块，并将其附加到 IR 构建器上。

```

1      BasicBlock *BB = BasicBlock::Create(M->getContext(),
2          "entry", MainFn);
3      Builder.SetInsertPoint(BB);

```

7. 做好这些准备工作后，就可以开始进行树形遍历。

```

1      Tree->accept(*this);

```

8. 遍历树后，通过 calc_write() 函数打印计算值。同样，必须创建函数原型 (FunctionType 的实例)。唯一的参数是当前值 V：

```

1      FunctionType *CalcWriteFnTy =
2          FunctionType::get(VoidTy, {Int32Ty}, false);
3      Function *CalcWriteFn = Function::Create(
4          CalcWriteFnTy, GlobalValue::ExternalLinkage,
5          "calc_write", M);
6      Builder.CreateCall(CalcWriteFnTy, CalcWriteFn, {V});

```

9. 生成结束时，main() 函数返回 0：

```

1      Builder.CreateRet(Int32Zero);
2  }

```

10. WithDecl 节点保存声明变量的名称。首先，需要为 calc_read() 函数创建一个函数原型：

```

1  virtual void visit(WithDecl &Node) override {

```

```

2   FunctionType *ReadFty =
3       FunctionType::get(Int32Ty, {Int8PtrTy}, false);
4   Function *ReadFn = Function::Create(
5       ReadFty, GlobalValue::ExternalLinkage,
6       "calc_read", M);

```

11. 该方法遍历变量名:

```

1   for (auto I = Node.begin(), E = Node.end(); I != E;
2       ++I) {

```

12. 对于每个变量, 都会创建一个带有变量名称的字符串。

```

1   StringRef Var = *I;
2   Constant *StrText = ConstantDataArray::getString(
3       M->getContext(), Var);
4   GlobalVariable *Str = new GlobalVariable(
5       *M, StrText->getType(),
6       /*isConstant=*/true,
7       GlobalValue::PrivateLinkage,
8       StrText, Twine(Var).concat(".str"));

```

13. 然后, 创建调用 calc_read() 函数的 IR 代码。在上一步骤中创建的字符串会作为参数传递:

```

1   Value *Ptr = Builder.CreateInBoundsGEP(
2       Str, {Int32Zero, Int32Zero}, "ptr");
3   CallInst *Call =
4       Builder.CreateCall(ReadFty, ReadFn, {Ptr});

```

14. 返回值存储在 mapNames 中:

```

1   nameMap[Var] = Call;
2   }

```

15. 使用树形遍历继续遍历表达式:

```

1   Node.getExpr()->accept(*this);
2   };

```

16. Factor 节点可以是变量名, 也可以是数字。对于变量名, 将在 mapNames 中查找值。对于数字, 该值转换为常数值:

```

1   virtual void visit(Factor &Node) override {
2       if (Node.getKind() == Factor::Ident) {
3           V = nameMap[Node.getVal()];
4       } else {

```

```

5     int intval;
6     Node.getVal().getAsInteger(10, intval);
7     V = ConstantInt::get(Int32Ty, intval, true);
8 }
9 };

```

17. 最后，对于 BinaryOp 节点，必须使用正确的计算操作：

```

1  virtual void visit(BinaryOp &Node) override {
2      Node.getLeft()->accept(*this);
3      Value *Left = V;
4      Node.getRight()->accept(*this);
5      Value *Right = V;
6      switch (Node.getOperator()) {
7          case BinaryOp::Plus:
8              V = Builder.CreateNSWAdd(Left, Right); break;
9          case BinaryOp::Minus:
10             V = Builder.CreateNSWSub(Left, Right); break;
11          case BinaryOp::Mul:
12             V = Builder.CreateNSWMul(Left, Right); break;
13          case BinaryOp::Div:
14             V = Builder.CreateSDiv(Left, Right); break;
15      }
16  };
17 };
18 }

```

18. 这样，访问者类就完成了。compile() 方法创建了全局上下文和模块，执行了树形遍历，并将生成的 IR 转储到控制台：

```

1 void CodeGen::compile(AST *Tree) {
2     LLVMContext Ctx;
3     Module *M = new Module("calc.expr", Ctx);
4     ToIRVisitor ToIR(M);
5     ToIR.run(Tree);
6     M->print(outs(), nullptr);
7 }

```

我们实现了编译器的前端，从读取源代码到生成 IR。当然，所有这些组件必须在用户输入时协同工作，这是编译器驱动程序的任务，现在还需要实现运行时所需的功能。我们将在下一节中讨论这两个问题。

驱动程序和运行时库

前几节中的所有阶段都是由 Calc.cpp 驱动程序合在一起的，我们将在这里实现它。需要为输入表达式声明一个参数，初始化 LLVM，调用前几节中的所有阶段。让我们来看看：

1. 首先，必须包含必需的头文件：

```
1 #include "CodeGen.h"
2 #include "Parser.h"
3 #include "Sema.h"
4 #include "llvm/Support/CommandLine.h"
5 #include "llvm/Support/InitLLVM.h"
6 #include "llvm/Support/raw_ostream.h"
```

2. LLVM 自带了声明命令行选项的系统，只需要为所需的每个选项声明一个静态变量。在此过程中，该选项将被注册到全局命令行解析器中。这种方法的优点是，每个组件都可以在需要时添加命令行选项。我们必须为输入表达式声明一个选项：

```
1 static llvm::cl::opt<std::string>
2     Input(llvm::cl::Positional,
3           llvm::cl::desc("<input expression>"),
4           llvm::cl::init(""));
```

3. 在 main() 函数中，LLVM 库初始化，需要调用 ParseCommandLineOptions 来处理命令行上的选项。这也可以处理打印帮助信息。在出现错误的情况下，此方法会让应用程序终止运行：

```
1 int main(int argc, const char **argv) {
2     llvm::InitLLVM X(argc, argv);
3     llvm::cl::ParseCommandLineOptions(
4         argc, argv, "calc - the expression compiler\n");
```

4. 接下来，使用词法分析器和解析器。语法分析之后，检查是否发生了错误。如果有错，则使用返回代码退出编译器，表示失败：

```
1     Lexer Lex(Input);
2     Parser Parser(Lex);
3     AST *Tree = Parser.parse();
4     if (!Tree || Parser.hasError()) {
5         llvm::errs() << "Syntax errors occurred\n";
6         return 1;
7     }
```

5. 如果有语义错误，也是一样：

```
1     Sema Semantic;
2     if (Semantic.semantic(Tree)) {
3         llvm::errs() << "Semantic errors occurred\n";
```

```

4     return 1;
5 }

```

6. 最后，在驱动程序中，调用代码生成器:

```

1     CodeGen CodeGenerator;
2     CodeGenerator.compile(Tree);
3     return 0;
4 }

```

这样，就成功地为用户输入创建了 IR 代码。我们将目标代码的生成委托给 LLVM 静态编译器 llc，这样就完成了编译器的实现。我们必须将所有组件链接到一起，从而创建 calc 应用程序。

运行库由一个名为 rtcalc.c 的文件组成。它包含了用 C 语言编写的 calc_read() 和 calc_write() 函数的实现。

```

1 #include <stdio.h>
2 #include <stdlib.h>
3
4 void calc_write(int v)
5 {
6     printf("The result is: %d\n", v);
7 }

```

calc_write() 只把结果值输出到终端。

```

1 int calc_read(char *s)
2 {
3     char buf[64];
4     int val;
5     printf("Enter a value for %s: ", s);
6     fgets(buf, sizeof(buf), stdin);
7     if (EOF == sscanf(buf, "%d", &val))
8     {
9         printf("Value %s is invalid\n", buf);
10        exit(1);
11    }
12    return val;
13 }

```

calc_read() 从终端读取一个整数。用户可以有任意的输入，所以我们必须仔细检查输入。如果输入不是数字，则应用程序退出。一种更复杂的方法是让用户意识到这个问题，并再次请求输入数字。

现在，我们可以试试我们的编译器。calc 应用程序从一个表达式创建 IR。LLVM 静态编译器将 IR 编译为一个目标文件。然后，可以使用您喜欢的 C 编译器链接到小型运行时库。在 Unix 上，可以这样：

```
$ calc "with a: a*3" | llc -filetype=obj -o=expr.o
$ clang -o expr expr.o rtcalc.c
$ expr
  Enter a value for a: 4
  The result is: 12
```

在 Windows 上，很可能会使用 cl 编译器：

```
$ calc "with a: a*3" | llc -filetype=obj -o=expr.obj
$ cl expr.obj rtcalc.c
$ expr
  Enter a value for a: 4
  The result is: 12
```

这样，就创建了第一个基于 llvm 的编译器！请花点时间研究一下这些不同的表达方式。还要检查乘法运算符是否在加法运算符之前求值，以及使用括号是否会改变求值顺序，这与基本计算器的要求一致。

总结

本章中，了解编译器的组件。算术表达式语言用来介绍编程语言的语法。然后，学习了如何为该语言开发前端的典型组件：词法分析器、解析器、语义分析器和代码生成器。代码生成器只生成 LLVM IR，而 LLVM 静态编译器 llc 用于从它创建目标文件。最后，开发了第一个基于 llvm 的编译器！

下一章中，您将加深这方面的知识，以便为语言构建前端。

2 从源码到机器码

您将学习如何开发自己的编译器，我们将从构造前端开始，读取源文件并创建抽象语法树。然后，您将学习如何从源文件生成 LLVM IR。使用 LLVM 的优化功能，您将创建优化的机器码。您还将了解更多高级主题，包括为面向对象语言构造生成 LLVM IR，以及如何添加调试元数据。

本节包括以下几章：

- 第 4 章，将源码转换为抽象语法树
- 第 5 章，生成 IR——基础知识
- 第 6 章，生成高级语言结构的 IR
- 第 7 章，生成 IR——进阶知识
- 第 8 章，优化 IR

第 4 章 将源文件转换为抽象语法树

编译器通常分为两部分: 前端和后端。本章中, 我们将实现一个程序设计语言的前端, 就是处理源语言的部分。我们将学习实际的编译器使用的技术, 并应用到我们自己的编程语言中。

我们将从定义编程语言的语法开始, 并以抽象语法树 (AST) 结束 (将是代码生成的基础)。您可以将此方法用于任何想要为其实现编译器的编程语言。

本章中, 您将学习以下主题:

- 定义一种编程语言——tinylang 语言, 它是编程语言的子集, 必须为它实现一个编译器前端。
- 为编译器创建项目布局。
- 管理源文件和用户消息, 这使您了解如何处理多个输入文件, 以及如何以一种友善的方式告知用户问题所在。
- 构建词法分析器, 讨论如何将词法分析器分解成模块部分。
- 构建递归下降解析器, 讨论从语法派生解析器, 以及执行语法分析时可以使用的规则。
- 使用 bison 和 flex 解析器和分析器, 其中您将使用工具根据规范轻松地生成解析器和分析器。
- 执行语义分析, 您将创建 AST 并评估其属性, 这会与解析器有一些交集。

有了本章学到的技能, 您将能够为任何编程语言构建编译器前端。

相关代码

本章的代码文件可在<https://github.com/PacktPublishing/Learn-LLVM-12/tree/master/Chapter04>获取。

你可以在视频中找到代码<https://bit.ly/3nllhED>。

定义一种编程语言

实际的编程语言比前一章简单的 calc 语言更复杂。为了更详细地研究它, 我将在本章和接下来的章节中使用 Modula-2 的子集。Modula-2 设计良好, 可选地支持泛型和面向对象编程 (OOP)(并不是说要在本书中创建一个完整的 Modula-2 编译器)。因此, 我将把这个子集命名为 tinylang。

让我们快速浏览一下 tinylang 语法的子集。在接下来的小节中, 我们将从该语法派生词法分析器和解析器:

```
compilationUnit
    : "MODULE" identifier ";" ( import )* block identifier "." ;
Import : ( "FROM" identifier )? "IMPORT" identList ";" ;
Block
    : ( declaration )* ( "BEGIN" statementSequence )? "END" ;
```

在 Modula-2 中, 编译单元以 MODULE 关键字开头, 然后是模块的名称。模块的内容可以是导入的模块列表、声明和在初始化时运行的语句的块:

```
declaration
  : "CONST" ( constantDeclaration ";" )*
  | "VAR" ( variableDeclaration ";" )*
  | procedureDeclaration ";" ;
```

声明引入常量、变量和过程。已声明的常量以 CONST 关键字作为前缀。类似地，变量声明以 VAR 关键字开头。声明常量非常简单：

```
constantDeclaration : identifier "=" expression ;
```

标识符是常量的名称。该值派生自表达式，该表达式必须在编译时是可计算的。声明变量有点复杂：

```
variableDeclaration : identList ":" qualident ;
qualident : identifier ( "." identifier )* ;
identList : identifier ( "," identifier )* ;
```

为了能够一次声明多个变量，必须使用标识符列表，类型的名称可能来自另一个模块。本例中以模块名作为前缀，称为限定标识符。而过程需要更多详细信息：

```
procedureDeclaration
  : "PROCEDURE" identifier ( formalParameters )? ";"
  block identifier ;
formalParameters
  : "(" ( formalParameterList )? ")" ( ":" qualident )? ;
formalParameterList
  : formalParameter ( ";" formalParameter )* ;
formalParameter : ( "VAR" )? identList ":" qualident ;
```

在前面的代码中，您可以看到常量、变量和过程是如何声明的。过程可以有参数和返回类型。普通参数作为值传递，而 VAR 参数通过引用传递。前面的块规则缺少的另一部分是 statementSequence，只是单个语句的列表：

```
statementSequence
  : statement ( ";" statement )* ;
```

如果语句后面跟着另一个语句，则该语句用分号分隔。同样，Modula-2 语句支持这种方式：

```
statement
  : qualident ( "!=" expression | "(" ( expList )? ")" )? )
  | ifStatement | whileStatement | "RETURN" ( expression )? ;
```

该规则的第一部分描述了一个赋值或过程调用。后跟`!=` 的限定标识符是一个赋值。另一方面，如果后面跟着 `(`，则它是一个过程调用。其他语句是常用的控制语句：

```
ifStatement
  : "IF" expression "THEN" statementSequence
    ( "ELSE" statementSequence )? "END" ;
```

IF 语句也有一个简化的语法，因为它只能有一个 ELSE 块。可以通过这个语句，我们有条件地保护语句：

```
whileStatement
  : "WHILE" expression "DO" statementSequence "END" ;
```

WHILE 语句描述了一个由条件保护的循环。加上 IF 语句，就可以用 `tinylang` 编写简单的算法。最后，没有表达式的定义：

```
expList
  : expression ( "," expression )* ;
expression
  : simpleExpression ( relation simpleExpression )? ;
relation
  : "=" | "#" | "<" | "<=" | ">" | ">=" ;
simpleExpression
  : ( "+" | "-" )? term ( addOperator term )* ;
addOperator
  : "+" | "-" | "OR" ;
term
  : factor ( mulOperator factor )* ;
mulOperator
  : "*" | "/" | "DIV" | "MOD" | "AND" ;
factor
  : integer_literal | "(" expression ")" | "NOT" factor
  | qualident ( "(" ( expList )? ")" )? ;
```

表达式语法与前一章的 `calc` 相似，只支持 `INTEGER` 和 `BOOLEAN` 数据类型。

此外，还使用 `identifier` 和 `integer_literal` 标记。**标识符**以字母或下划线开头，后面跟着字母、

数字和下划线。**整数字面值**可以是一个十进制数字序列，也可以是一个十六进制数字序列，后面跟着字母 H。

虽然已经介绍了很多规则，但只涉及 Modula-2 的一部分! 尽管如此，还是可以使用这个子集中编写小型应用程序。接下来，让我们为 tinylang 实现一个编译器!

创建项目结构

tinylang 的项目布局遵循了我们在第 2 章，LLVM 源的访问中列出的方式。每个组件的源代码在 lib 目录的子目录中，而头文件在 include/tinylang 的子目录中，子目录以组件命名。在第 2 章，我们只创建了基本组件。

在前一章中，我们知道需要实现词法分析器、解析器、AST 和语义分析器。其都有自己的组件，称为 Lexer、Parser、AST 和 Sema。上一章中使用的目录布局如下:

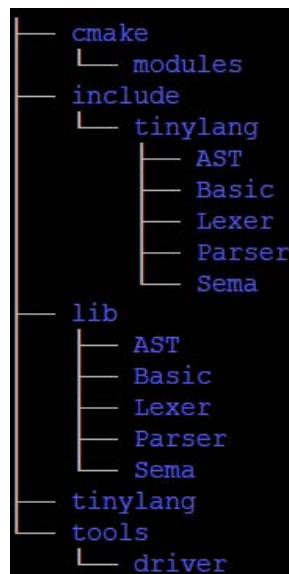


图 4.1 -tinylang 项目的目录布局

组件有明确定义的依赖项。在这里，Lexer 只依赖于 Basic。Parser 依赖于 Basic、Lexer、AST 和 Sema。最后，Sema 只依赖于 Basic 和 AST，这些定义良好的依赖有助于对组件的重用。

让我们仔细了解一下它们是如何实现的。

管理源文件和用户消息

实际的编译器必须能够处理许多文件。通常，开发人员用主编译单元的名称来调用编译器。这个编译单元可以引用其他文件，例如：通过 C 语言中的 `#include` 指令，或 Python 或 Modula-2 中的 `import` 语句。导入的模块可以导入其他模块等。所有这些文件都必须加载到内存中，并通过编译器的分析阶段运行。在开发过程中，开发人员可能会犯语法或语义错误。当检测到错误信息时，应该打印出错误信息，包括源行和一个标记，很明显，这个基本组件很重要。

幸运的是，LLVM 附带了一个解决方案:LLVM::SourceMgr 类。通过调用 `AddNewSourceBuffer()` 方法，将新源文件添加到 SourceMgr。或者，通过调用 `AddIncludeFile()` 来加载文件。这两个方法都返回一个 ID 来标识缓冲区，可以使用这个 ID 来检索指向关联文件的内存缓冲区的指针。要在

文件中定义位置，必须使用 `llvm::SMLoc` 类，这个类将指针封装到缓冲区中。各种 `PrintMessage()` 会向用户发出错误和其他信息消息。

只缺少一种集中定义消息的方法。大型软件 (如编译器) 中，您不希望到处都是消息字符串。如果有人要求更改消息或将其翻译成另一种语言，那么最好把它们放在中心位置!

一种简单的方法是，每个消息都有一个 ID (一个 `enum` 成员)、一个严重性级别和一个包含消息的字符串。代码中，只引用消息 ID。严重性级别和消息字符串仅在打印消息时使用。必须一致地管理这三个项 (ID、严重性级别和消息)。LLVM 库使用一个预处理器来解决这个问题。数据存储在在一个后缀为 `a.def` 的文件中，并封装在一个宏名称中。该文件通常包含多次，宏的定义不同。它的定义在 `include/tinylang/Basic/Diagnostic.def` 文件路径中，如下所示:

```
1 #ifndef DIAG
2 #define DIAG(ID, Level, Msg)
3 #endif
4
5 DIAG(err_sym_declared, Error, "symbol {0} already declared")
6 #undef DIAG
```

第一个宏参数 ID 是枚举标签，第二个参数 Level 是严重性，第三个参数 Msg 是消息文本。这样，就可以定义一个 `DiagnosticsEngine` 类发出的错误消息。该接口位于 `include/tinylang/Basic/Diagnostic.h` 文件中:

```
1 #ifndef TINYLANG_BASIC_DIAGNOSTIC_H
2 #define TINYLANG_BASIC_DIAGNOSTIC_H
3
4 #include "tinylang/Basic/LLVM.h"
5 #include "llvm/ADT/StringRef.h"
6 #include "llvm/Support/FormatVariadic.h"
7
8 #include "llvm/Support/SMLoc.h"
9 #include "llvm/Support/SourceMgr.h"
10 #include "llvm/Support/raw_ostream.h"
11 #include <utility>
12
13 namespace tinylang {
```

包含必要的头文件之后，现在使用 `Diagnostic.def` 来定义枚举。为了不污染全局命名空间，需要使用命名空间 `diag`:

```
1 namespace diag {
2     enum {
3         #define DIAG(ID, Level, Msg) ID,
4         #include "tinylang/Basic/Diagnostic.def"
5     };
```

```
6 } // namespace diag
```

DiagnosticsEngine 类使用 SourceMgr 实例通过 report() 发出消息，消息可以有参数。要实现这个功能，必须使用 LLVM 的可变格式支持。在静态方法的帮助下检索消息文本和严重性级别。此外，还会计算发出的错误消息的数量：

```
1 class DiagnosticsEngine {
2     static const char *getDiagnosticText(unsigned DiagID);
3     static SourceMgr::DiagKind
4     getDiagnosticKind(unsigned DiagID);
```

消息字符串由 getDiagnosticText() 返回，级别由 getDiagnosticKind() 返回。这两种方法都将在以后的.cpp 文件中实现：

```
1     SourceMgr &SrcMgr;
2     unsigned NumErrors;
3
4 public:
5     DiagnosticsEngine(SourceMgr &SrcMgr)
6         : SrcMgr(SrcMgr), NumErrors(0) {}
7
8     unsigned numErrors() { return NumErrors; }
```

因为消息可以有可变数量的参数，所以 C++ 中的解决方案是使用可变参数模板。当然，LLVM 提供的 formatv() 函数也使用这种方法。要获得格式化的消息，我们只需要转发模板参数：

```
1     template <typename... Args>
2     void report(SMLoc Loc, unsigned DiagID,
3                 Args &&... Arguments) {
4         std::string Msg =
5             llvm::formatv(getDiagnosticText(DiagID),
6                           std::forward<Args>(Arguments)...)
7             .str();
8         SourceMgr::DiagKind Kind = getDiagnosticKind(DiagID);
9         SrcMgr.PrintMessage(Loc, Kind, Msg);
10        NumErrors += (Kind == SourceMgr::DK_Error);
11    }
12};
13
14 } // namespace tinylang
15
16 #endif
```


这样，就实现了类的大部分内容。只缺少 `getDiagnosticText()` 和 `getDiagnosticKind()`。它们在 `lib/Basic/Diagnostic.cpp` 文件中定义，并使用 `Diagnostic.def` 文件：

```
1 #include "tinylang/Basic/Diagnostic.h"
2 using namespace tinylang;
3 namespace {
4     const char *DiagnosticText[] = {
5         #define DIAG(ID, Level, Msg) Msg,
6         #include "tinylang/Basic/Diagnostic.def"
7     };
```

与头文件中一样，`DIAG` 宏定义为检索所需的部分。这里，将定义一个数组来保存文本消息。因此，`DIAG` 宏只返回 `Msg` 部分。我们将使用相同的方法：

```
1 SourceMgr::DiagKind DiagnosticKind[] = {
2 #define DIAG(ID, Level, Msg) SourceMgr::DK_##Level,
3 #include "tinylang/Basic/Diagnostic.def"
4 };
5 } // namespace
```

不出所料，这两个函数都只是简单地对数组进行索引，并返回所需的数据：

```
1 const char *
2 DiagnosticsEngine::getDiagnosticText(unsigned DiagID) {
3     return DiagnosticText[DiagID];
4 }
5
6 SourceMgr::DiagKind
7 DiagnosticsEngine::getDiagnosticKind(unsigned DiagID) {
8     return DiagnosticKind[DiagID];
9 }
```

`SourceMgr` 和 `DiagnosticsEngine` 类的这种组合为其他组件提供了良好的基础。让我们先在词法分析器中使用一下！

构建词法分析器

正如前一章中所知，我们需要一个 `Token` 类和一个 `Lexer` 类。此外，需要一个 `TokenKind` 枚举来给每个令牌类一个的数字。拥有一体化的头文件和一个实现文件是无法扩展的，所以需要重构这些内容。`TokenKind` 枚举可以复用，并放置在 `Basic` 组件中。`Token` 和 `Lexer` 类属于 `Lexer` 组件，但是放在不同的头文件和实现文件中。

有三种不同类型的标记：关键字、标点符号和表示多值集的标记，例如：`CONST` 关键字，分号分隔符和标识符，它们表示源中的标识符，每个令牌都需要枚举的成员名。关键字和标点符号具有可以用于消息的显示名称。

与许多编程语言一样，关键字是标识符的子集。为了将标记分类为关键字，我们需要一个关键字过滤器，它检查所找到的标识符是否确实是关键字。这与 C 或 C++ 中的行为相同，其中关键字也是标识符的子集。编程语言随着时间的推移而发展，可能会引入新的关键字，例如：K&R C 语言没有使用 enum 关键字定义的枚举。因此，应该有指示关键字的语言级别的标志。

我们收集了几条信息，它们都属于 TokenKind 枚举的一个成员：枚举成员的标签、标点符号的拼写和关键字的标志。至于调试消息，我们将信息集中存储在一个名为 include/tinylang/Basic/TokenKinds.def 的文件中，文件的内容如下所示。需要注意的是，关键字的前缀是 kw_：

```
1 #ifndef TOK
2 #define TOK(ID)
3 #endif
4 #ifndef PUNCTUATOR
5 #define PUNCTUATOR(ID, SP) TOK(ID)
6 #endif
7 #ifndef KEYWORD
8 #define KEYWORD(ID, FLAG) TOK(kw_ ## ID)
9 #endif
10
11 TOK(unknown)
12 TOK(eof)
13 TOK(identifier)
14 TOK(integer_literal)
15
16 PUNCTUATOR(plus, "+")
17 PUNCTUATOR(minus, "-")
18 // ...
19
20 KEYWORD(BEGIN, KEYALL)
21 KEYWORD(CONST, KEYALL)
22 // ...
23
24 #undef KEYWORD
25 #undef PUNCTUATOR
26 #undef TOK
```

有了这些集中的定义，就很容易在 include/tinylang/Basic/TokenKinds.h 中创建 TokenKind 枚举。同样，枚举也可以放在自己的命名空间 tok 中：

```
1 #ifndef TINYLANG_BASIC_TOKENKINDS_H
2 #define TINYLANG_BASIC_TOKENKINDS_H
3
4 namespace tinylang {
```

```

5
6 namespace tok {
7 enum TokenKind : unsigned short {
8 #define TOK(ID) ID,
9 #include "TokenKinds.def"
10     NUM_TOKENS
11 };

```

填充数组使用的模式已经很熟悉了，TOK 宏定义为仅返回枚举标签的 ID。作为一个有用的加法，我们还将 NUM_TOKENS 定义为枚举的最后一个成员，表示已定义标记的数量：

```

1     const char *getTokenName(TokenKind Kind);
2     const char *getPunctuatorSpelling(TokenKind Kind);
3     const char *getKeywordSpelling(TokenKind Kind);
4 }
5 }
6
7 #endif

```

实现文件 lib/Basic/TokenKinds.cpp 也使用 .def 文件来检索名称：

```

1 #include "tinylang/Basic/TokenKinds.h"
2 #include "llvm/Support/ErrorHandling.h"
3
4 using namespace tinylang;
5
6 static const char * const TokNames[] = {
7 #define TOK(ID) #ID,
8 #define KEYWORD(ID, FLAG) #ID,
9 #include "tinylang/Basic/TokenKinds.def"
10     nullptr
11 };

```

令牌的文本名称派生自其枚举标签的 ID，其有两个特点。首先，需要定义两个 TOK 和 KEYWORD 宏，因为 KEYWORD 的默认定义不使用 TOK 宏。其次，会在数组的末尾添加一个 nullptr 值，用于计算 NUM_TOKENS 枚举成员：

```

1 const char *tok::getTokenName(TokenKind Kind) {
2     return TokNames[Kind];
3 }

```

对于 getPunctuatorSpelling()getKeywordSpelling() 函数，我们采用了不同的方法。这些函数只返回枚举子集的有意义的值。可以通过 switch 语句实现，该语句默认返回 nullptr 值：

```

1 const char *tok::getPunctuatorSpelling(TokenKind Kind) {
2     switch (Kind) {
3 #define PUNCTUATOR(ID, SP) case ID: return SP;
4 #include "tinylang/Basic/TokenKinds.def"
5         default: break;
6     }
7     return nullptr;
8 }
9
10 const char *tok::getKeywordSpelling(TokenKind Kind) {
11     switch (Kind) {
12 #define KEYWORD(ID, FLAG) case kw_ ## ID: return #ID;
13 #include "tinylang/Basic/TokenKinds.def"
14         default: break;
15     }
16     return nullptr;
17 }

```

Tip

注意如何定义宏从文件中检索所需的信息片段。

前一章中，Token 类在与 Lexer 类相同的头文件中声明。为了使其更加模块化，我们将把 Token 类放到 include/Lexer/Token.h 中。前面的例子中，Token 存储了一个指针，指向标记的开始、长度和标记的类型：

```

1 class Token {
2     friend class Lexer;
3
4     const char *Ptr;
5     size_t Length;
6     tok::TokenKind Kind;
7
8 public:
9     tok::TokenKind getKind() const { return Kind; }
10    size_t getLength() const { return Length; }

```

SMLoc 实例表示源在消息中的位置，它是使用指向令牌的指针创建：

```

1 SMLoc getLocation() const {
2     return SMLoc::getFromPointer(Ptr);
3 }

```

getIdentifier() 和 getLiteralData() 允许我们访问标识符和文字数据的标记文本。没有必要访问任何其他令牌类型的文本，因为这是隐式的令牌类型：

```
1 StringRef getIdentifier() {
2      assert(is(tok::identifier) &&
3              "Cannot get identifier of non-identifier");
4      return StringRef(Ptr, Length);
5  }
6  StringRef getLiteralData() {
7      assert(isOneOf(tok::integer_literal,
8                      tok::string_literal) &&
9              "Cannot get literal data of non-literal");
10     return StringRef(Ptr, Length);
11 }
12};
```

我们在 include/Lexer/Lexer.h 头文件中声明 Lexer 类，并将实现放在 lib/Lexer/Lexer.cpp 文件中。其结构与前一章的 calc 语言相同。这里，我们必须留意两个细节：

- 首先，有些操作符共享相同的前缀，例如：< 和 <=。当我们正在查看的当前字符是 < 时，首先检查下一个字符，然后再决定我们找到了哪个标记。记住，我们要求输入以空字节结束。因此，如果当前字符有效，则可以使用下一个字符：

```
1 case '<':
2     if (*(CurPtr + 1) == '=')
3         formTokenWithChars(token, CurPtr + 2, tok::lessequal);
4     else
5         formTokenWithChars(token, CurPtr + 1, tok::less);
6     break;
```

- 另一个细节是，如有更多的关键词，我们该如何处理？一个简单而快速的解决方案是用关键字填充哈希表，这些关键字都存储在 tokenkind.def 文件中，可在实例化 Lexer 类时完成。这种方法还可以支持不同级别的语言，因为可以用附加的标志过滤关键字。这里，还不需要这种灵活性。头文件中，关键字过滤器定义如下，使用一个实例 llvm::StringMap 的哈希表：

```
1 class KeywordFilter {
2     llvm::StringMap<tok::TokenKind> HashTable;
3     void addKeyword(StringRef Keyword,
4                     tok::TokenKind TokenCode);
5 public:
6     void addKeywords();
```

getKeyword() 方法返回给定字符串的标记类型，如果字符串不代表关键字，则返回默认值：

```

1 tok::TokenKind getKeyword(
2    StringRef Name,
3     tok::TokenKind DefaultTokenCode = tok::unknown) {
4     auto Result = HashTable.find(Name);
5     if (Result != HashTable.end())
6         return Result->second;
7     return DefaultTokenCode;
8 }
9 };

```

在实现文件中，关键字表填充为:

```

1 void KeywordFilter::addKeyword(StringRef Keyword,
2 tok::TokenKind TokenCode)
3 {
4     HashTable.insert(std::make_pair(Keyword, TokenCode));
5 }
6
7 void KeywordFilter::addKeywords() {
8 #define KEYWORD(NAME, FLAGS)
9 addKeyword(StringRef(#NAME), tok::kw_##NAME);
10 #include "tinylang/Basic/TokenKinds.def"
11 }

```

使用这些技术，编写一个高效的 lexer 类并不困难。由于编译速度很重要，许多编译器都使用手写的分析器，Clang 就是其中之一。

构建递归下降解析器

如前一章所示，解析器是从语法派生出来的。让我们回顾一下所有的构造规则：对于每个语法规则，都要创建一个以规则左侧的非终结符命名的方法，以便解析规则的右侧。根据右边的定义，必须做以下事情：

- 对于每个非终结符，调用相应的方法。
- 处理每个令牌。
- 对于可选组和可选组或重复组，将检查前瞻性令牌 (下一个未使用的令牌)，以决定从哪里继续。

让我们将这些结构规则应用到以下语法规则中：

```
ifStatement
    : "IF" expression "THEN" statementSequence
      ( "ELSE" statementSequence )? "END" ;
```

可以很容易地将其转换为以下 C++ 方法:

```
1 void Parser::parseIfStatement() {
2     consume(tok::kw_IF);
3     parseExpression();
4     consume(tok::kw_THEN);
5     parseStatementSequence();
6     if (Tok.is(tok::kw_ELSE)) {
7         advance();
8         parseStatementSequence();
9     }
10    consume(tok::kw_END);
11 }
```

tinylang 的整个语法可以用这种方式转换成 C++。不过，必须小心并避免一些陷阱。

需要注意的一个问题是左递归规则。如果右边和左边以相同的终端开始，规则是左递归的。表达式的语法中可以找到一个典型的例子:

```
expression : expression "+" term ;
```

如果还不清楚语法，那么下面对 C++ 的翻译应该可以清楚地表明，这将导致无限递归:

```
1 void Parser::parseExpression() {
2     parseExpression();
3     consume(tok::plus);
4     parseTerm();
5 }
```

左递归也可能间接发生，涉及更多规则，这很难发现。这就是为什么存在一种可以检测和消除左递归的算法。

每个步骤中，解析器决定如何仅通过使用预先标记继续。如果不能确定，语法就会发生冲突。为了说明这一点，让我们看看 C# 中的 using 语句。与 C++ 类似，using 语句可用于在名称空间中使符号可见，例如在 using Math; 中。还可以为导入的符号定义别名; 也就是说，使用 $M = Math$; 在语法中，可以这样表示:

```
usingStmt : "using" (ident "=")? ident ";"
```

显然，这里有个问题。解析器使用了 using 关键字之后，就标识了预先标记。但是，这些信息

不足以让我们决定是否必须跳过或解析可选组。如果可选组开始的一组标记与可选组后面的一组标记重叠，则总是会出现这种情况。

我们用一个替代方案来重写这个规则：

```
usingStmt : "using" ( ident "=" ident | ident ) ";" ;
```

现在，有一个不同的冲突：两个选择都以相同的标记开始。只看预先标记，解析器不能决定哪个选项是正确的。

这些冲突非常普遍。因此，需要了解如何处理它们。一种方法是以冲突消失的方式重写语法。在前面的示例中，两个选项都以相同的标记开始。这可以分解，得到以下规则：

```
usingStmt : "using" ident ("=" ident)? ";" ;
```

这种表示没有冲突。然而，还应该注意的，它的表达性较差。其他两个公式中，很明显哪个标识是别名，哪个标识是名称空间名称。这个无冲突规则中，最左边的标识改变了它的角色。首先，是名称空间名称，但是如果后面跟着等号 (=)，那么就变成别名。

第二种方法是添加一个额外的谓词来区分这两种情况。这个谓词通常称为解释器，它可以使用上下文信息进行决策（例如符号表中的名称查找），但可以查看多个令牌。让我们假设 lexer 有一个 *Token & peek(int n)* 方法，在当前查找令牌之后返回第 n 个令牌。这里，等号的存在可以作为判定中的附加谓词：

```
1 if (Tok.is(tok::ident) && Lex.peek(0).is(tok::equal)) {  
2     advance();  
3     consume(tok::equal);  
4 }  
5 consume(tok::ident);
```

现在，让我们加入错误恢复。前一章中，介绍了恐慌模式作为一种错误恢复技术。基本思想是跳过标记，直到找到适合继续解析的标记为止。例如，在 tinylang 中，语句后面跟着分号 (;)。

如果 If 语句中存在语法问题，则跳过所有标记，直到找到分号。然后，继续下一个。与其为令牌集的特别定义标记，不如使用系统方法。

对于每个非终结符，计算可以在跟随该非终结符的标记集（称为 FOLLOW 集）。对于非终结符语句，可以跟、ELSE 和 END 标记。因此，可以在 parseStatement() 的错误恢复部分中使用这个集合。当然，此方法假设语法错误可以在本地处理。通常来说，这不太可能。因为解析器会跳过标记，所以可能会跳过很多标记，以至于到达输入的末尾。所以，无法进行本地恢复。

为了防止无意义的错误消息，需要通知调用方法错误恢复仍未完成。这可以通过 bool 返回值来完成：true 表示错误恢复尚未完成，false 表示解析（包括可能的错误恢复）成功。

有许多方法可以扩展这个错误恢复方案，一种流行的方法是也使用活动调用者的 FOLLOW 集合。一个简单的例子，让我们假设 parseStatement() 由 parseStatementSequence() 调用，而 parseBlock() 本身调用 parseBlock()，而 parseModule() 调用 parseStatement()。

这里，每个对应的非终端都有一个 FOLLOW 集。如果解析器检测到 parseStatement() 中的语

法错误，那么将跳过令牌，直到令牌至少出现在活动调用者的一个 FOLLOW 集合中。如果令牌在语句的 FOLLOW 集合中，则在本地恢复错误，并将一个假值返回给调用者。否则，返回一个真值，这意味着错误恢复必须继续。这个扩展的一个可能的实现策略是，传递 `std::bitset` 或 `std::tuple` 来表示当前 FOLLOW 的集合返回给调用者。

最后一个问题仍然未解决：如何调用错误恢复？前一章中，使用 `goto` 跳转到错误恢复块。这可行，但不是很优雅的解决方案。根据前面的讨论，可以在单独的方法中跳过标记。为此，Clang 有一个名为 `skipUntil()` 的方法，我们也可以将其用于 `tinylang`。

因为下一步是向解析器添加语义操作，所以如果有必要，最好有找个位置完成清理代码（嵌套函数是不错的选择）。C++ 没有嵌套函数，所以 Lambda 函数可以用于这项任务。具有错误恢复的 `parseIfStatement()` 如下所示：

```
1 bool Parser::parseIfStatement() {
2     auto _errorhandler = [this] {
3         return SkipUntil(tok::semi, tok::kw_ELSE, tok::kw_END);
4     };
5     if (consume(tok::kw_IF))
6         return _errorhandler();
7     if (parseExpression(E))
8         return _errorhandler();
9     if (consume(tok::kw_THEN))
10        return _errorhandler();
11    if (parseStatementSequence(IfStmts))
12        return _errorhandler();
13    if (Tok.is(tok::kw_ELSE)) {
14        advance();
15        if (parseStatementSequence(ElseStmts))
16            return _errorhandler();
17    }
18    if (expect(tok::kw_END))
19        return _errorhandler();
20    return false;
21 }
```

用 bison 和 flex 生成解析器和词法分析器

手动构造词法分析器和解析器并不困难，通常会产生处理速度很快的组件。但缺点是不容易更改，特别是在解析器中。如果正在创建一种新的编程语言的原型，那么这一点非常重要。使用特定的工具可以缓解这个问题。

有许多工具可以根据规范文件生成词法分析器或解析器。在 Linux 中，flex (<https://github.com/westes/flex>) 和 bison (<https://www.gnu.org/software/bison/>) 是常用工具。Flex 从一组

正则表达式生成分析器，而 bison 从语法描述生成解析器。通常，这两种工具会一起使用。

Bison 根据语法描述生成 LALR(1) 解析器。LALR(1) 解析器是一种自下而上的解析器，并且使用自动机实现。bison 的输入是一个语法文件，与本章开始介绍的语法文件非常相似。主要的区别是右侧规则不支持正则表达式，可选的组和重复检查必须为规则重写。tinylang 的 bison 规范，存储在 tinylang.yy 文件中，以以下内容开始：

```
%require "3.2"
%language "c++"
%defines "Parser.h"
%define api.namespace tinylang
%define api.parser.class Parser
%define api.token.prefix T_
%token
identifier integer_literal string_literal
PLUS MINUS STAR SLASH
```

我们指示 bison 使用 %language 指令生成 C++ 代码。使用 %define 指令，重写了代码生成的一些默认值：生成的类应该命名为 Parser，并位于 tinylang 命名空间中。此外，表示标记类型的枚举的成员应该以 T_ 作为前缀。我们需要 3.2 或更高版本，因为其中一些变量在这个版本中引入。为了能够与 flex 交互，我们告诉 bison 用 %define 指令编写一个 Parser.h 头文件。最后，我们必须使用 %token 指令声明所有已使用的令牌。%% 后面是具体的语法规则：

```
%%
compilationUnit
    : MODULE identifier SEMI imports block identifier PERIOD ;
imports : %empty | import imports ;
import
    : FROM identifier IMPORT identList SEMI
    | IMPORT identList SEMI ;
```

请将这些规则与本章第一节所示的语法规则进行比较。bison 不知道重复组，因此需要添加一个名为 imports 的新规则来对这种重复。在导入规则中，必须引入一个替代方法来对可选组建模。

我们还需要用这种风格重写 tinylang 语法的其他规则。例如，IF 语句的规则如下：

```
ifStatement
    : IF expression THEN statementSequence
      elseStatement END ;
elseStatement : %empty | ELSE statementSequence ;
```

同样，我们必须引入一个新的规则来处理可选的 ELSE 语句。可以省略 %empty 指令，不过这

里使用的是空分支。

当我们用 `bison` 风格重写了所有语法规则，就可以用以下命令生成解析器：

```
$ bison tinylang.yy
```

这就是创建与上一节中手写解析器类似解析器的全部内容！

类似地，`flex` 很容易使用。`flex` 的规范是一个正则表达式和相关操作的列表，如果正则表达式匹配，则执行该列表。`tinylang.l` 文件指定 `tinylang` 的词法分析器。与 `bison` 规范一样，也有一个标准开头：

```
%
#include "Parser.h"
%
%option noyywrap nounput noininput batch
id [a-zA-Z][a-zA-Z_0-9]*
digit [0-9]
hexdigit [0-9A-F]
space [ \t\r]
```

在 `%{ }%` 里面的文本复制到 `flex` 生成的文件中，我们使用这种机制来包含由 `bison` 生成的头文件。使用 `%option` 指令，我们控制生成的解析器应该具有哪些特性。只读取一个文件，并且在读取完一个文件后不需要继续读取另一个文件，所以可以指定 `noyywrap` 来禁用这个特性。我们也不需要访问底层的文件流，使用 `nounput` 和 `noinout` 禁用它。因为我们已经不需要一个交互式的解析器了，所以需要生成一个批扫描程序。

在开头的内容中，我们还可以定义字符模式以供以后使用。在 `%%` 后面跟着定义部分：

```
%%
space+
digit+      return
            tinylang::Parser::token::T_integer_literal;
```

定义部分中，指定正则表达式模式和在模式与输入匹配时执行的操作。动作也可以是空的。

`{space}+` 模式使用序言中定义的空格字符模式，它匹配一个或多个空格字符。我们没有定义任何操作，因此将忽略所有空白。

要匹配一个数字，我们使用 `digit+` 模式。作为一个动作，我们只返回关联的令牌类型。对所有令牌都做了同样的操作。例如，对算术运算符执行以下操作：

```
"+"      return tinylang::Parser::token::T_PLUS;
"_"      return tinylang::Parser::token::T_MINUS;
"*"      return tinylang::Parser::token::T_STAR;
"/"      return tinylang::Parser::token::T_SLASH;
```

如果有几个模式匹配输入，则选择匹配时间最长的模式。如果仍然有多个模式与输入匹配，那么将选择规范文件中按字典顺序最先出现的模式。这就是为什么必须首先定义关键字的模式，而只在所有关键字之后定义标识符模式：

```
"VAR"      return tinylang::Parser::token::T_VAR;
"WHILE"     return tinylang::Parser::token::T_WHILE;
id          return tinylang::Parser::token::T_identifier;
```

这些操作不仅仅局限于 `return` 语句。如果您的代码有多行，那么您必须用大括号括起来 `{}`。扫描通过以下命令生成：

```
$ flex -c++ tinylang.l
```

在语言项目中你应该使用哪种方法？解析器生成器通常生成 LALR(1) 解析器。LALR(1) 类比 LL(1) 类大，可以为其构造递归下降解析器。如果不能调整语法以适应 LL(1) 类，那么应该考虑使用解析器生成器，手工构造一个自底向上的解析器是不可行的。即使语法是 LL(1)，解析器生成器在生成与您手工编写代码类似的代码时也会提供更多的便利。通常，这受许多因素影响的选择。Clang 使用手写解析器，而 GCC 使用 bison 生成的解析器。

执行语义分析

上一节构造的解析器只检查输入的语法，下一步是添加执行语义分析的能力。在前一章的 `calc` 例子中，解析器构造了 AST。在另一个阶段，语义分析器会处理这棵树。在本节中，我们将使用一种不同的方法，并将解析器和语义分析器进一步融合在一起。

以下是语义分析器必须执行的一些任务：

- 对于每个声明，语义分析器必须检查所使用的名称是否已经声明过。
- 对于表达式或语句中每次出现的名称，语义分析程序必须检查名称是否已声明，以及期望的用途是否符合声明。
- 对于每个表达式，语义分析器必须计算得到的类型。还需要计算表达式是否为常量。如果是，它的值是多少。
- 对于赋值和参数传递，语义分析器必须检查类型是否兼容。此外，必须检查 IF 和 WHILE 语句中的条件是否为布尔类型。

对于这样一个编程语言的子集来说，要检查的内容已经很多了！

处理名称的作用域

让我们先看看名称的作用域，名称的范围是名称可见的范围。与 C 语言一样，tinylang 也使用了先声明后使用的模型，例如：B 和 X 变量是在模块级声明的，因此它们是 INTEGER 类型：

```
VAR B, X: INTEGER;
```

在声明之前，变量是未知的，不能使用。这只有在声明之后才有可能。在过程中，可以声明更多的变量：

```
PROCEDURE Proc;  
VAR B: BOOLEAN;  
BEGIN  
    (* Statements *)  
END Proc;
```

在这个过程中，在注释所在的地方，使用 B 指的是局部变量 B，而使用 X 指的是全局变量 X。局部变量 B 的作用域是 Proc 过程。如果在当前作用域中找不到名称，则在外围作用域中继续搜索。因此，可以在过程中使用 X 变量。在 tinylang 中，只有模块和过程才能打开一个新的作用域。其他语言构造，如 struct 和 class，通常也打开作用域。预定义 (如 INTEGER 类型或 TRUE 字面值) 在全局作用域中声明，该作用域中包含模块的作用域。

在 tinylang 中，只有名字才是关键。因此，作用域可以实现为从名称到其声明的映射，只有在新名称不存在的情况下才能插入新名称。对于查找，封闭或父作用域也必须已知。该接口 (在 include/tinylang/Sema/Scope.h 文件中) 如下所示：

```
1 #ifndef TINYLANG_SEMA_SCOPE_H  
2 #define TINYLANG_SEMA_SCOPE_H  
3  
4 #include "tinylang/Basic/LLVM.h"  
5 #include "llvm/ADT/StringMap.h"  
6 #include "llvm/ADT/StringRef.h"  
7  
8 namespace tinylang {  
9  
10 class Decl;  
11  
12 class Scope {  
13     Scope *Parent;  
14     StringMap<Decl *> Symbols;  
15  
16 public:  
17     Scope(Scope *Parent = nullptr) : Parent(Parent) {}  
18
```

```

19  bool insert(Decl *Declaration);
20  Decl *lookup(StringRef Name);
21
22  Scope *getParent() { return Parent; }
23 };
24 } // namespace tinylang
25 #endif

```

lib/Sema/Scope.cpp 文件中的实现如下所示:

```

1  #include "tinylang/Sema/Scope.h"
2  #include "tinylang/AST/AST.h"
3
4  using namespace tinylang;
5
6  bool Scope::insert(Decl *Declaration) {
7      return Symbols
8          .insert(std::pair<StringRef, Decl *>(
9              Declaration->getName(), Declaration))
10         .second;
11 }

```

请注意, StringMap::insert() 方法并不覆盖现有的规则条目。生成的 std::pair 的第二个成员表示表是否更新, 该信息将返回给调用者。

为了实现对符号声明的搜索, lookup() 方法搜索当前范围。如果没有找到, 则搜索由父成员链接的作用域:

```

1  Decl *Scope::lookup(StringRef Name) {
2      Scope *S = this;
3      while (S) {
4          StringMap<Decl *>::const_iterator I =
5              S->Symbols.find(Name);
6          if (I != S->Symbols.end())
7              return I->second;
8          S = S->getParent();
9      }
10     return nullptr;
11 }

```

然后对变量声明进行如下处理:

- 当前作用域是模块作用域。
- 查找 INTEGER 类型声明。如果没有找到声明, 或者不是类型声明, 则为错误。

- 实例化了新的 AST 节点 VariableDeclaration，其中重要的属性是名称 B 和类型。
- 将名称 B 插入到当前作用域，并映射到声明实例。如果名称已经在作用域中，那么是一个错误。这种情况下，当前范围的内容不会改变。
- 对 X 变量也做了同样的处理。

这里执行两个任务。与在 calc 示例中一样，构造了 AST 节点。同时，计算节点的属性，比如它的类型。为什么这样可行呢？

语义分析器可以依赖两组不同的属性。作用域是从调用者继承的，可以通过计算类型声明的名称来计算 (或合成) 类型声明。这种语言的设计方式使这两组属性足以计算 AST 节点的所有属性。

其中一个重要的方面是使用前声明模型。如果一种语言允许在声明之前使用名称，比如 C++ 中的类中的成员，不可能一次计算 AST 节点的所有属性。这种情况下，AST 节点必须仅使用部分计算的属性或纯信息构造 (例如 calc 示例)。

必须访问 AST 一次或多次才能确定丢失的信息。在 tinylang(和 Modula-2) 的情况下，也可以不使用 AST 构造——AST 是通过 parseXXX() 方法的调用层次结构间接表示。从 AST 生成代码要常见得多，因此在这里构造了 AST。

把所有的部分合一起前，需要理解使用 LLVM 风格的运行时类型信息 (RTTI)。

AST 中使用 LLVM 风格的 RTTI

当然，AST 节点是类层次结构的一部分。声明总是有名字，其他属性取决于声明的内容。如果声明了一个变量，则需要一个类型。常量声明需要类型和值等。当然，在运行时需要找出使用的是哪种声明。可以使用 dynamic_cast<> C++ 操作符。问题是，必需的 RTTI 只有在 C++ 类附加了一个虚表时才可用；也就是说，使用虚函数。另一个缺点是 C++ RTTI 过于臃肿。为了避免这些缺点，LLVM 开发人员引入了一种自制的 RTTI 风格。

层次结构的 (抽象) 基类是 Decl。要实现 LLVM 风格的 RTTI，需要添加包含每个子类标签的公共枚举。此外，还需要此类型的私有成员和公共 getter，私有成员通常称为 Kind。我们的例子中，它看起来像这样：

```

1 class Decl {
2 public:
3     enum DeclKind { DK_Module, DK_Const, DK_Type,
4         DK_Var, DK_Param, DK_Proc };
5 private:
6     const DeclKind Kind;
7 public:
8     DeclKind getKind() const { return Kind; }
9 };

```

每个子类现在都需要一个名为 classof 的特殊函数成员。此函数的目的是确定给定实例是否属于请求的类型，对于 VariableDeclaration，实现如下：

```

1 static bool classof(const Decl *D) {
2     return D->getKind() == DK_Var;

```

```
3 }
```

现在,可以使用 `llvm::isa<>` 特殊模板来检查对象是否为所请求的类型,并使用 `llvm::dyn_cast<>` 来强制转换对象。还有更多的模板,但这两个是最常用的。其他模板请参阅<https://llvm.org/docs/ProgrammersManual.html#the-isa-cast-and-dyn-cast-templates>。关于 LLVM 样式的更多信息,包括更高级的用法,请参阅<https://llvm.org/docs/HowToSetUpLLVMStyleRTTI.html>。

创建语义分析器

有了这些知识,现在可以实现语义分析器,在分析器创建的 AST 节点上操作。首先,将为变量实现 AST 节点的定义,该变量存储在 `include/llvm/tinylang/AST/AST.h` 文件中。除了支持 LLVM 风格的 RTTI 之外,基类还存储声明的名称、名称的位置和一个指向声明的指针,后者是代码生成嵌套过程所必需的。Decl 基类的声明如下:

```
1 class Decl {
2     public:
3         enum DeclKind { DK_Module, DK_Const, DK_Type,
4                         DK_Var, DK_Param, DK_Proc };
5
6     private:
7         const DeclKind Kind;
8
9     protected:
10         Decl *EnclosingDecl;
11         SMLoc Loc;
12        StringRef Name;
13
14     public:
15         Decl(DeclKind Kind, Decl *EnclosingDecl, SMLoc Loc,
16             StringRef Name)
17             : Kind(Kind), EnclosingDecl(EnclosingDecl), Loc(Loc),
18               Name(Name) {}
19
20         DeclKind getKind() const { return Kind; }
21         SMLoc getLocation() { return Loc; }
22         StringRef getName() { return Name; }
23         Decl *getEnclosingDecl() { return EnclosingDecl; }
24 };
```

变量的声明只添加指向类型声明的指针:

```
1 class TypeDeclaration;
2
```



```

3         Decl *D) {
4     if (TypeDeclaration *Ty = dyn_cast<TypeDeclaration>(D)) {
5         for (auto I = Ids.begin(), E = Ids.end(); I != E; ++I) {
6             SMLoc Loc = I->first;
7            StringRef Name = I->second;
8             VariableDeclaration *Decl = new VariableDeclaration(
9                 CurrentDecl, Loc, Name, Ty);
10            if (CurrentScope->insert(Decl))
11                Decls.push_back(Decl);
12            else
13                Diags.report(Loc, diag::err_symbold_declared, Name);
14        }
15    } else if (!Ids.empty()) {
16        SMLoc Loc = Ids.front().first;
17        Diags.report(Loc, diag::err_vardecl_requires_type);
18    }
19 }

```

首先，使用 `llvm::dyn_cast<TypeDeclaration>` 检查类型声明。如果不是类型声明，则打印错误消息。否则，对于 `Ids` 列表中的每个名称，将实例化一个 `VariableDeclaration` 并添加到声明列表中。如果将变量添加到当前作用域失败，因为名称已经声明，则打印一条错误消息。

大多数其他实体都是以相同方式构造，唯一不同的是它们语义分析的复杂性。模块和过程需要更多的工作，因为它们开辟了一个新的范围。开辟新的作用域很容易：只需要实例化新的 `scope` 对象。当模块或过程被解析，就必须删除作用域。

这必须以靠谱的方式完成，因为我们不想在语法错误的情况下将名称添加到错误的作用域。这是 C++ 中资源获取即初始化 (RAII) 习惯用法的经典用法。另一个复杂之处在于过程可以递归地调用自己。因此，在使用过程之前，必须将过程的名称添加到当前作用域，语义分析器有两种方法来进入和离开一个范围。作用域与声明相关联：

```

1 void Sema::enterScope(Decl *D) {
2     CurrentScope = new Scope(CurrentScope);
3     CurrentDecl = D;
4 }
5
6 void Sema::leaveScope() {
7     Scope *Parent = CurrentScope->getParent();
8     delete CurrentScope;
9     CurrentScope = Parent;
10    CurrentDecl = CurrentDecl->getEnclosingDecl();
11 }

```

简单的 helper 类用于实现 RAII:

```
1 class EnterDeclScope {
2     Sema &Semantics;
3     public:
4     EnterDeclScope(Sema &Semantics, Decl *D)
5     : Semantics(Semantics) {
6         Semantics.enterScope(D);
7     }
8     ~EnterDeclScope() { Semantics.leaveScope(); }
9 };
```

在解析模块或过程中, 有两个与语义分析器的交互。第一个是在名称解析之后。这里, 构造了(几乎为空的)AST 节点, 并建立了一个新的作用域:

```
1 bool Parser::parseProcedureDeclaration(/* ... */) {
2     /* ... */
3     if (consume(tok::kw_PROCEDURE)) return _errorhandler();
4     if (expect(tok::identifier)) return _errorhandler();
5     ProcedureDeclaration *D =
6         Actions.actOnProcedureDeclaration(
7             Tok.getLocation(), Tok.getIdentifier());
8     EnterDeclScope S(Actions, D);
9     /* ... */
10 }
```

语义分析器不只是检查当前作用域中的名称, 还要返回 AST 节点:

```
1 ProcedureDeclaration *
2 Sema::actOnProcedureDeclaration(SMLoc Loc, StringRef Name) {
3     ProcedureDeclaration *P =
4         new ProcedureDeclaration(CurrentDecl, Loc, Name);
5     if (!CurrentScope->insert(P))
6         Diags.report(Loc, diag::err_symboldeclared, Name);
7     return P;
8 }
```

当解析了所有的声明和过程的主体, 实际工作就完成了。基本上, 语义分析器必须只检查过程声明末尾的名称是否等于过程的名称, 以及用于返回类型的声明是否真的是类型声明:

```
1 void Sema::actOnProcedureDeclaration(
2     ProcedureDeclaration *ProcDecl, SMLoc Loc,
3     StringRef Name, FormalParamList &Params, Decl *RetType,
4     DeclList &Decls, StmtList &Stmts) {
```

```

5
6  if (Name != ProcDecl->getName()) {
7      Diags.report(Loc, diag::err_proc_identifier_not_equal);
8      Diags.report(ProcDecl->getLocation(),
9                  diag::note_proc_identifier_declaration);
10 }
11 ProcDecl->setDecls(Decls);
12 ProcDecl->setStmts(Stmts);
13
14 auto RetTypeDecl =
15     dyn_cast_or_null<TypeDeclaration>(RetType);
16 if (!RetTypeDecl && RetType)
17     Diags.report(Loc, diag::err_returntype_must_be_type,
18                 Name);
19 else
20     ProcDecl->setRetType(RetTypeDecl);
21 }

```

有些声明是固有的，不能由开发人员定义。这包括 BOOLEAN 和 INTEGER 类型以及 TRUE 和 FALSE 字面值。这些声明存在于全局作用域中，必须通过编程方式添加。Modula-2 还预定义了一些，如 INC 或 DEC，也应该添加到全局范围。给定我们的类，全局作用域的初始化很简单：

```

1 void Sema::initialize() {
2     CurrentScope = new Scope();
3     CurrentDecl = nullptr;
4     IntegerType =
5         new TypeDeclaration(CurrentDecl, SMLoc(), "INTEGER");
6     BooleanType =
7         new TypeDeclaration(CurrentDecl, SMLoc(), "BOOLEAN");
8     TrueLiteral = new BooleanLiteral(true, BooleanType);
9     FalseLiteral = new BooleanLiteral(false, BooleanType);
10    TrueConst = new ConstantDeclaration(CurrentDecl, SMLoc(),
11                                       "TRUE", TrueLiteral);
12    FalseConst = new ConstantDeclaration(
13        CurrentDecl, SMLoc(), "FALSE", FalseLiteral);
14    CurrentScope->insert(IntegerType);
15    CurrentScope->insert(BooleanType);
16    CurrentScope->insert(TrueConst);
17    CurrentScope->insert(FalseConst);
18 }

```

有了这个方案，tinylang 所需的所有计算都可以完成。例如，要计算表达式是否产生常量值，必须确保：

- 常量声明的文字或引用是常量。
- 如果表达式的两边都是常量，使用该操作符也会得到一个常量。

在为表达式创建 AST 节点时，可以轻松地将这些规则嵌入到语义分析器中。同样，可以计算类型和常量值。

需要指出的是，并不是所有类型的计算都可以用这种方法来完成，例如：要检测未初始化变量的使用，可以使用一种称为符号解释的方法。在其一般形式中，该方法需要通过 AST 的特殊遍历顺序（这在构造期间是不可能的）。好消息是，本文介绍的方法创建了一个经过修饰的 AST，它可以用于代码生成。当然，这个 AST 可以用于进一步的分析，可以根据需要打开或关闭分析。

要使用前端，还需要更新驱动程序。由于缺少代码生成，正确的 tinylang 程序不会产生输出。尽管如此，它仍可用于探索错误恢复和引发语义错误：

```
1 #include "tinylang/Basic/Diagnostic.h"
2 #include "tinylang/Basic/Version.h"
3 #include "tinylang/Parser/Parser.h"
4 #include "llvm/Support/InitLLVM.h"
5 #include "llvm/Support/raw_ostream.h"
6
7 using namespace tinylang;
8
9 int main(int argc_, const char **argv_) {
10     llvm::InitLLVM X(argc_, argv_);
11     llvm::SmallVector<const char *, 256> argv(argv_ + 1,
12                                               argv_ + argc_);
13     llvm::outs() << "Tinylang "
14                 << tinylang::getTinylangVersion() << "\n";
15
16     for (const char *F : argv) {
17         llvm::ErrorOr<std::unique_ptr<llvm::MemoryBuffer>>
18             FileOrErr = llvm::MemoryBuffer::getFile(F);
19         if (std::error_code BufferError =
20             FileOrErr.getError()) {
21             llvm::errs() << "Error reading " << F << ": "
22                         << BufferError.message() << "\n";
23             continue;
24         }
25
26         llvm::SourceMgr SrcMgr;
27         DiagnosticsEngine Diags(SrcMgr);
```

```

28     SrcMgr.AddNewSourceBuffer(std::move(*FileOrErr),
29                               llvm::SMLoc());
30     auto lexer = Lexer(SrcMgr, Diags);
31     auto sema = Sema(Diags);
32     auto parser = Parser(lexer, sema);
33     parser.parse();
34 }
35 }

```

恭喜你! 已经完成了 tinylang 的前端实现!

现在, 让我们尝试一下目前所学到的东西。以下是欧几里得最大公约数算法的实现的源代码, 保存为 Gcd.mod 文件:

```

MODULE Gcd;

PROCEDURE GCD(a, b: INTEGER):INTEGER;
VAR t: INTEGER;
BEGIN
    IF b = 0 THEN RETURN a; END;
    WHILE b # 0 DO
        t := a MOD b;
        a := b;
        b := t;
    END;
    RETURN a;
END GCD;

END Gcd.

```

让我们用下面的命令运行编译器:

```

$ tinylang Gcm.mod
Tinylang 0.1

```

除了打印的版本号, 没有输出, 这是因为只实现了前端部分。但是, 如果您更改源代码, 使其包含语法错误, 则将打印错误消息。

下一章中, 我们将通过添加代码生成来继续这项任务。

总结

本章中，您学习了在前端使用的编译器技术。从项目的布局开始，您为词法分析器、解析器和语义分析器创建了单独的库。为了向用户输出消息，您扩展了现有的 LLVM 类，允许集中存储消息。词法分析器目前已经分成几个接口。

然后，您学习了如何根据语法描述构造递归下降解析器、要避免哪些缺陷，以及如何使用生成器来完成这项工作。您构建的语义分析器执行语言所需的所有语义检查，同时与解析器和 AST 构造交织在一起。

编码工作的结果是一个修饰过的 AST，下一章中将使用它生成 IR 代码和目标代码。

第 5 章 生成 IR——基础知识

在为编程语言创建了修饰的抽象语法树 (AST) 之后，下一个任务是从它生成 LLVM IR 代码。LLVM IR 代码类似于三地址代码，具有人类可读的表示形式。因此，我们需要一种系统的方法，将诸如控制结构之类的语言概念翻译到较底层次的 LLVM IR 中。

本章中，您将学习 LLVM IR 的基础知识，以及如何从 AST 中为控制流结构生成 IR。还将学习如何使用现代算法，为静态单分配 (SSA) 形式的表达式生成 LLVM IR。最后，您将学习如何生成汇编文本和目标代码。

本章将涵盖以下内容：

- 使用 AST 生成 IR
- 使用 AST 编码生成 SSA 形式的 IR 代码
- 设置模块和驱动

本章结束时，将了解为自己的编程语言创建代码生成器的知识，以及如何将它集成到自己的编译器中。

相关代码

本章的代码文件可在<https://github.com/PacktPublishing/Learn-LLVM-12/tree/master/Chapter05/tinylang>获取。

你可以在视频中找到代码<https://bit.ly/3nllhED>。

使用 AST 生成 IR

LLVM 代码生成器接受 IR 中描述的模块作为输入，并将其转换为目标代码或汇编文本，我们需要将 AST 表示转换为 IR。为了实现 IR 代码生成器，我们将首先查看一个简单的示例，然后开发代码生成器所需的类。完整的实现将分为三个类：CodeGenerator、CGModule 和 CGProcedure 类。CodeGenerator 类是编译器驱动程序使用的通用接口。CGModule 和 CGProcedure 类保存了为编译单元和单个函数生成 IR 代码所需的状态。

下一节中，我们首先看看 Clang 生成的 IR。

了解 IR 代码

生成 IR 代码之前，最好先了解 IR 语言的主要元素。在第 3 章中，我们已经简要地了解了 IR。获得更多 IR 的一个简单方法是研究 clang 的输出。例如，保存以下 C 源代码，它实现了计算两个数的最大公约数的欧几里得算法：

```
1 unsigned gcd(unsigned a, unsigned b) {  
2     if (b == 0)  
3         return a;  
4     while (b != 0) {  
5         unsigned t = a % b;  
6         a = b;
```



```
7     b = t;  
8 }  
9 return a;  
10 }
```

然后可以使用以下命令，创建 IR 文件 gcd.ll:

```
$ clang -target=aarch64-linux-gnu -O1 -S -emit-llvm gcd.c
```

IR 代码与目标有关，该命令用于编译 Linux 环境下 ARM 64 位 CPU 的源文件。-S 选项指示 clang 输出一个程序集文件，通过附加的-emit-llvm，创建一个 IR 文件。优化级别-O1 用于获得易读的 IR 代码。看一下生成的文件，并理解 C 源代码如何映射到 IR。在文件的顶部，有一些基本信息:

```
; ModuleID = 'gcd.c'  
source_filename = "gcd.c"  
target datalayout = "e-m:e-i8:8:32-i16:16:32-i64:64-  
                    i128:128-n32:64-S128"  
target triple = "aarch64-unknown-linux-gnu"
```

第一行是注释，说明使用了哪个模块标识符。下一行中，注明了源文件的文件名。对于 clang，两者一样。

target datalayout 字符串建立了一些基本属性。它的各个部分用-隔开。包括以下信息:

- 小 e 意味着内存中的字节使用小端模式存储。要指定大的端序，可以使用大写的 E。
- 指定应用于符号的名称转换。这里，m:e 表示使用 ELF 名称 mangling。
- 在 iN:A:P 形式中的条目 i8:8:32，指定了数据的对齐方式，以位为单位。第一个数字是 ABI 所需的对齐方式，第二个数字是首选的对齐方式。对于 bytes (i8)，ABI 对齐是 1 字节 (8)，首选对齐是 4 字节 (32)。
- n 指定可用的本机寄存器大小。n32:64 意味着本地支持 32 位和 64 位宽整数。
- s 指定堆栈的对齐方式，同样以位为单位。S128 表示堆栈保持 16 字节对齐。

Note

目标数据的更多的信息，你以在参考手册<https://llvm.org/docs/LangRef.html#data-layout>。

最后，target triple 字符串指定了我们要编译的体系结构。对于我们在命令行中给出的信息来说，这是必不可少的。这个在第 2 章，已经进行了深入的讨论。

接下来，在 IR 文件中定义 gcd 函数:

```
define i32 @gcd(i32 %a, i32 %b) {
```

这类似于 C 文件中的函数签名。unsigned 数据类型被转换为 32 位整数类型 i32。函数名以 @ 作为前缀，参数名以 % 作为前缀。函数体用大括号括起来。正文代码如下：

```
entry:
    %cmp = icmp eq i32 %b, 0
    br i1 %cmp, label %return, label %while.body
```

IR 代码组织在基本块中。结构良好的**基本块**是指令的线性序列，它以可选标签开始，以终止指令（例如，分支或返回指令）结束。因此，每个基本块都有一个入口点和一个出口点。LLVM 允许在构造时出现畸形的基本块。第一个基本块的标签是 entry。块中的代码很简单：第一个指令比较参数 %b 和 0。第二个指令分支到 label return，如条件为真，则分支到 label while；若条件为假，则返回。

IR 代码的另一个特点是 SSA 形式的。代码使用无限数量的虚拟寄存器，但每个寄存器只编写一次。比较的结果分配给指定的虚拟寄存器 %cmp。这个寄存器随后会使用，但它永远不会再写入。诸如常量传播和公共子表达式消除之类的优化在 SSA 表单中工作得非常好，所有现代编译器都在使用它。

下一个基本块是 while 的循环体：

```
while.body:
    %b.addr.010 = phi i32 [ %rem, %while.body ],
                        [ %b, %entry ]
    %a.addr.09 = phi i32 [ %b.addr.010, %while.body ],
                        [ %a, %entry ]
    %rem = urem i32 %a.addr.09, %b.addr.010
    %cmp1 = icmp eq i32 %rem, 0
    br i1 %cmp1, label %return, label %while.body
```

在 gcd 循环中，赋予 a 和 b 参数以新值。如果一个寄存器只能写入一次，那么是无法完成的。解决方法是使用特殊的 phi 指令，phi 指令有一个基本块列表和值作为参数。一个基本块表示从哪边的基本块进入的，值就是那些基本块的值。在运行时，phi 指令将前面执行的基本块的标签与参数列表中的标签进行比较。

指令的值与标签的值相关联。对于第一个 phi 指令，如果之前执行的基本块是 while.body，则值为 %rem。如果 entry 是前面执行的基本块，则该值为 %b。这些值位于基本块的开始部分。%b.addr.010 从第一个 phi 指令中得到一个值。在第二个 phi 指令的参数列表中使用了相同的寄存器，但该值假定为通过第一个 phi 指令改变它之前的值。

在循环体之后，必须选择返回值：

```

return:
    %retval.0 = phi i32 [ %a, %entry ],
                    [ %b.addr.010, %while.body ]
    ret i32 %retval.0
}

```

同样，使用 phi 指令来选择所需的值。ret 指令不仅可以结束这个基本块，还表示该函数在运行时的结束。它将返回值作为参数。

对于 phi 指令的使用有一些限制，必须是基本块的第一个指令。第一个基本块是特殊的：没有块在它之前执行过。因此，不能以 phi 指令开始。

IR 代码本身看起来很像 C 语言和汇编语言的混合。尽管风格类似，我们还不清楚如何从 AST 轻松生成 IR 代码。特别是 phi 指令看起来很难生成，在下一节中，我们将介绍一个简单的算法来实现这一点！

了解加载-存储

LLVM 中的所有本地优化都基于这里显示的 SSA 形式。对于全局变量，使用内存引用。IR 语言知道用于获取和存储这些值的加载和存储指令，也可以将此用于局部变量。这些指令不是 SSA 形式的，LLVM 知道如何将它们转换成所需的 SSA 形式。因此，可以为每个局部变量分配内存，并使用加载-存储指令更改它们的值。您只需要记住指向存储变量的内存的指针，clang 编译器使用的就是这种方法。

让我们看看带有加载和存储的 IR 代码。再次编译 gcd.c，这次不启用优化：

```
$ clang -target=aarch64-linux-gnu -S -emit-llvm gcd.c
```

gcd 函数现在看起来不同了。这是第一个基本块：

```

define i32 @gcd(i32, i32) {
    %3 = alloca i32, align 4
    %4 = alloca i32, align 4
    %5 = alloca i32, align 4
    %6 = alloca i32, align 4
    store i32 %0, i32* %4, align 4
    store i32 %1, i32* %5, align 4
    %7 = load i32, i32* %5, align 4
    %8 = icmp eq i32 %7, 0
    br i1 %8, label %9, label %11
}

```

IR 编码现在可以自动传递寄存器和标签的编号，为未指定参数名称。默认情况下，它们是 %0 和 %1。基本块没有标签，所以赋值为 2。第一个指令为 4 个 32 位值分配内存。之后，参数 %0 和 %1 存储在寄存器 %4 和 %5 所指向的内存中。要执行参数 %1 与 0 的比较，显式地从内存加载该值（使

用这种方法，而不是 phi 指令)! 相反，可以从内存加载一个值，对其执行计算，并将新值存储回内存中。下一次读取内存时，将得到最后计算的值。gcd 函数的所有其他基本块都遵循这个模式。

以这种方式使用加载-存储指令的好处是，生成 IR 代码相当容易。缺点是，在将基本块转换为 SSA 形式之后，将生成大量 IR 指令，LLVM 将在第一个优化步骤中使用 mem2reg 通道删除这些指令。因此，我们直接以 SSA 的形式生成 IR 代码。

我们将控制流映射到基本块开始，来完成开发 IR 代码生成。

将控制流映射到基本块

如前一节所述，格式良好的基本块只是指令的线性序列。一个基本块可以从 phi 指令开始，必须以一个分支指令结束。基本块中，不允许使用 phi 和分支指令。每个基本块都有一个标签，标记基本块的第一条指令。标签是分支指令的目标。可以将分支视为两个基本块之间的有向边，从而得到控制流图 (CFG)。一个基本模块可以有前身和继任者，不过函数的第一个基本块是特殊的 (没有任何块在它之前)。

由于这些限制，源语言的控制语句 (如 WHILE 或 IF) 会生成几个基本块。让我们看看 WHILE 语句，WHILE 语句的条件控制是执行循环体还是执行下一条语句。条件必须在它自己的基本块中生成，因为前面有两个块：

- 由 WHILE 循环之前的语句产生的基本块
- 循环体的末端条件分支

还有两个后继：

- 循环体的开始部分
- 由 WHILE 循环后的语句产生的基本块

循环体本身至少有一个基本块：

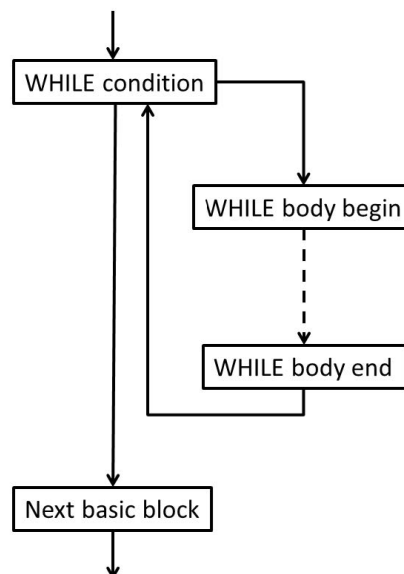


图 5.1 -WHILE 语句的基本块

IR 代码生成遵循这个结构。在 `CGProcedure` 类中存储一个指向当前基本块的指针，并使用 `llvm::IRBuilder<>` 的实例，将指令插入到基本块中。首先，创建基本块：

```
1 void emitStmt(WhileStatement *Stmt) {
2     llvm::BasicBlock *WhileCondBB = llvm::BasicBlock::Create(
3         getLLVMCtx(), "while.cond", Fn);
4     llvm::BasicBlock *WhileBodyBB = llvm::BasicBlock::Create(
5         getLLVMCtx(), "while.body", Fn);
6     llvm::BasicBlock *AfterWhileBB =
7         llvm::BasicBlock::Create(
8             getLLVMCtx(), "after.while", Fn);
```

`Fn` 变量表示当前函数，`getLLVMCtx()` 返回 LLVM 上下文，两者都会在以后设置。我们用一个基本块的分支来结束当前的基本块，保存条件：

```
1     Builder.CreateBr(WhileCondBB);
```

条件的基本块成为新的当前基本块。我们生成条件并以条件分支结束代码块：

```
1     setCurr(WhileCondBB);
2     llvm::Value *Cond = emitExpr(Stmt->getCond());
3     Builder.CreateCondBr(Cond, WhileBodyBB, AfterWhileBB);
```

接下来，生成循环体。作为最后一条指令，我们向条件的基本块添加一个分支：

```
1     setCurr(WhileBodyBB);
2     emit(Stmt->getWhileStmts());
3     Builder.CreateBr(WhileCondBB);
```

这就结束了 WHILE 语句的生成。WHILE 语句后面的空基本块将成为新的当前基本块：

```
1     setCurr(AfterWhileBB);
2 }
```

按照这个模式，您可以为源语言的每个语句创建 `emit()` 方法。

使用 AST 编码生成 SSA 形式的 IR

为了从 AST 生成 SSA 形式的 IR 编码，我们使用了一种称为 **AST 编码** 的方法。基本思想是，对于每个基本块，我们存储这个基本块中局部变量的当前值。

虽然很简单，但仍然需要几个步骤。我们将首先介绍所需的数据结构，然后实现对基本块局部值的读写。然后，将处理在几个基本块中使用的值，并通过优化所创建的 `phi` 指令得出结论。

定义保存值的数据结构

使用 `struct BasicBlockDef` 来保存单个块的信息：

```

1 struct BasicBlockDef {
2     llvm::DenseMap<Decl *, llvm::TrackingVH<llvm::Value>> Defs;
3     // ...
4 };

```

LLVM 类 `llvm::Value` 表示一个 SSA 形式的值，`Value` 类的作用类似于计算结果上的标签。它通常通过 IR 指令创建一次，然后使用。在各种优化过程中可能会有更改。例如，如果优化器检测到值 `%1` 和 `%2` 总是相同的，那么可以将 `%2` 的使用替换为 `%1`，这会改变标签，但不会改变计算。要注意这些变化，不能直接使用 `Value` 类。相反，我们需要一个值句柄，有不同功能的值句柄。要跟踪替换，要使用 `llvm::TrackingVH<>` 类。因此，`Defs` 成员将 AST 的声明 (变量或形式参数) 映射到其当前值。现在我们需要为每个基本块存储这些信息：

```

1 llvm::DenseMap<llvm::BasicBlock *, BasicBlockDef>
2     CurrentDef;

```

有了这个数据结构，就能够处理局部值了。

读写基本块的局部值

为了将局部变量的当前值存储在一个基本块中，只需在映射中创建一个条目：

```

1 void writeLocalVariable(llvm::BasicBlock *BB, Decl *Decl,
2                         llvm::Value *Val) {
3     CurrentDef[BB].Defs[Decl] = Val;
4 }

```

变量值的查找稍微复杂一点，因为值可能不在基本块中。这种情况下，需要使用可能的递归搜索将搜索扩展到前面几个基本块：

```

1 llvm::Value *
2 readLocalVariable(llvm::BasicBlock *BB, Decl *Decl) {
3     auto Val = CurrentDef[BB].Defs.find(Decl);
4     if (Val != CurrentDef[BB].Defs.end())
5         return Val->second;
6     return readLocalVariableRecursive(BB, Decl);
7 }

```

真正的工作是搜索前几个基本块，这将在下一节中实现。

搜索前一个块以查找值

如果我们正在查看的当前基本块只有一个前块，那么就在那里搜索变量的值。如果基本块有几个前块，那么需要搜索所有这些块中的值，并将结果合并。为了说明这种情况，可以查看上一节中带有 `WHILE` 语句条件的基本块。

这个基本块有两个前块——一个来自 WHILE 循环之前的语句，另一个来自 WHILE 循环主体的分支。条件中使用的变量应该有一个初始值，并且很可能在循环体中更改。所以，需要收集这些定义并从中创建一条 phi 指令。由 WHILE 语句创建的基本块包含一个循环。

因为我们递归地搜索前一个块，所以必须打破这个循环。为此，需要使用一个简单的技巧——插入一条空的 phi 指令，并将其记录为变量的当前值。如果在搜索中再次看到这个基本块，那么就会发现这个变量有值，我们会使用它，并且搜索到此为止。在我们收集了所有的值之后，必须对 phi 指令进行更新。

我们仍将面临一个问题。查找时，并不是所有基本块的前块都是已知的。这是怎么可能呢？看看 WHILE 语句的基本块的创建。首先生成循环条件的 IR，但从 body 的末端返回到包含条件的基本块的分支只能在生成 body 的 IR 之后添加，因为这个基本块之前是不知道的。如果需要在条件中读取一个变量的值，就会卡住，因为不是所有的前一个变量都是已知的。

要解决这个问题，就必须再多做一点：

- 首先，给基本块附加一个标志。
- 然后，如果知道基本块的所有前块，就定义一个基本块为“密封的”。基本块没有密封，我们需要查找这个基本块中尚未定义的变量值，然后插入一个空的 phi 指令。
- 我们还需要记住这个标识。该块之后密封时，需要用实际值更新指令。IncompletePhis 的 map 记录了需要更新的 phi 指令，Sealed 标志表明基本块是否密封：

```
1 llvm::DenseMap<llvm::PHINode *, Decl *>  
2   IncompletePhis;  
3 unsigned Sealed : 1;
```

- 然后，该方法的实现如下：

```
1 llvm::Value *readLocalVariableRecursive(  
2     llvm::BasicBlock *BB,  
3     Decl *Decl) {  
4     llvm::Value *Val = nullptr;  
5     if (!CurrentDef[BB].Sealed) {  
6         llvm::PHINode *Phi = addEmptyPhi(BB, Decl);  
7         CurrentDef[BB].IncompletePhis[Phi] = Decl;  
8         Val = Phi;  
9     } else if (auto *PredBB = BB  
10        ->getSinglePredecessor()) {  
11         Val = readLocalVariable(PredBB, Decl);  
12     } else {  
13         llvm::PHINode *Phi = addEmptyPhi(BB, Decl);  
14         Val = Phi;  
15         writeLocalVariable(BB, Decl, Val);  
16         addPhiOperands(BB, Decl, Phi);  
17     }
```

```

18     writeLocalVariable(BB, Decl, Val);
19     return Val;
20 }

```

- addEmptyPhi() 方法在基本块的开头插入一条空 phi 指令:

```

1 llvm::PHINode *addEmptyPhi(llvm::BasicBlock *BB, Decl
2 *Decl) {
3     return BB->empty()
4         ? llvm::PHINode::Create(mapType(Decl), 0,
5             "", BB)
6         : llvm::PHINode::Create(mapType(Decl), 0,
7             "", &BB->front());
8 }

```

- 为了将缺失的操作数加到 phi 指令中，我们首先搜索基本块的所有前任，然后将操作数对值和基本块加到 phi 指令中。然后，尝试优化指令:

```

1 void addPhiOperands(llvm::BasicBlock *BB, Decl *Decl,
2                     llvm::PHINode *Phi) {
3     for (auto I = llvm::pred_begin(BB),
4         E = llvm::pred_end(BB);
5         I != E; ++I) {
6         Phi->addIncoming(readLocalVariable(*I, Decl), *I);
7     }
8     optimizePhi(Phi);
9 }

```

这种算法会产生不必要的 phi 指令。下一节将尝试实现一种优化方法。

优化生成的 phi 指令

我们如何优化 phi 指令，为什么要这样做？虽然 SSA 形式对许多优化是有利的，但算法通常无法理解 phi 指令，所以会阻碍优化。因此，我们生成的 phi 指令是越少越好:

- 如果指令只有一个操作数或所有操作数都有相同的值，则用这个值替换指令。如果指令没有操作数，则用特殊值 Undef 替换该指令。只有当指令有两个或多个不同的操作数时，才需要保留指令:

```

1 void optimizePhi(llvm::PHINode *Phi) {
2     llvm::Value *Same = nullptr;
3     for (llvm::Value *V : Phi->incoming_values()) {
4         if (V == Same || V == Phi)
5             continue;

```



```

6     if (Same && V != Same)
7         return;
8     Same = V;
9 }
10 if (Same == nullptr)
11     Same = llvm::UndefValue::get(Phi->getType());

```

- 删除一个 phi 指令可能会导致其他 phi 指令有优化的机会。我们搜索值在其他 phi 指令中的用法后，尝试优化这些指令：

```

1     llvm::SmallVector<llvm::PHINode *, 8> CandidatePhis;
2     for (llvm::Use &U : Phi->uses()) {
3         if (auto *P =
4             llvm::dyn_cast<llvm::PHINode>(U.getUser()))
5             CandidatePhis.push_back(P);
6     }
7     Phi->replaceAllUsesWith(Same);
8     Phi->eraseFromParent();
9     for (auto *P : CandidatePhis)
10         optimizePhi(P);
11 }

```

如果需要，该算法还可以进一步改进。可以选择并记住两个不同的值，而不是迭代每个 phi 指令的值列表。优化函数中，可以检查这两个值是否仍然在 phi 指令的列表中。如果是，那么就on知道没有什么需要优化的了。但即使没有这个优化，这个算法也运行得很快，所以我们现在不打算实现它。

我们快实现完了。只有密封基本块的操作还没有实现，我们将在下一节中继续实现。

密封块

只要我们知道一个块的所有前块，就可以密封该块。如果源语言只包含结构化语句，比如 `tinylang`，那么很容易确定块可以被密封的位置。再次查看为 `WHILE` 语句生成的基本块。包含条件的基本块可以密封后，从分支体的末端添加，因为这是最后一个缺失的前块。要封住一个块，只需将缺少的操作数添加到不完整的 phi 指令中，并设置标志：

```

1 void sealBlock(llvm::BasicBlock *BB) {
2     for (auto PhiDecl : CurrentDef[BB].IncompletePhis) {
3         addPhiOperands(BB, PhiDecl.second, PhiDecl.first);
4     }
5     CurrentDef[BB].IncompletePhis.clear();
6     CurrentDef[BB].Sealed = true;
7 }

```

有了这些方法，就可以为表达式生成 IR 代码了。

为表达式创建 IR 代码

一般来说，你可以按照第 3 章编译器的结构来翻译表达式，唯一有难度的部分是如何访问变量。上一节介绍了局部变量，但还有其他类型的变量。让我们简单讨论一下需要做什么：

- 对于过程的局部变量，使用上一节中的 `readLocalVariable()` 和 `writeLocalVariable()`。
- 对于封闭过程中的局部变量，需要指向封闭过程框架的指针。后面的部分将对此进行处理。
- 对于全局变量，生成加载和存储指令。
- 对于形式参数，必须区分按值传递和按引用传递 (tinylang 中的 VAR 参数)。按值传递的参数被视为局部变量，按引用传递的参数将视为全局变量。

把所有这些放在一起，我们可以得到下面的代码，从而可以读取变量或形式参数：

```
1 llvm::Value *CGProcedure::readVariable(llvm::BasicBlock
2                                     *BB,
3                                     Decl *D) {
4     if (auto *V = llvm::dyn_cast<VariableDeclaration>(D)) {
5         if (V->getEnclosingDecl() == Proc)
6             return readLocalVariable(BB, D);
7         else if (V->getEnclosingDecl() ==
8                 CGM.getModuleDeclaration()) {
9             return Builder.CreateLoad(mapType(D),
10                                     CGM.getGlobal(D));
11         } else
12             llvm::report_fatal_error(
13                 "Nested procedures not yet supported");
14     } else if (auto *FP =
15                 llvm::dyn_cast<FormalParameterDeclaration>(
16                 D)) {
17         if (FP->isVar()) {
18             return Builder.CreateLoad(
19                 mapType(FP->getPointerElementType(),
20                 FormalParams[FP]));
21         } else
22             return readLocalVariable(BB, D);
23     } else
24         llvm::report_fatal_error("Unsupported declaration");
25 }
```

写入变量或形式参数是对称的，只需要将读的方法与写的方法交换，并使用 `store` 指令，而不是 `load` 指令。

接下来，在为函数生成 IR 代码时应用这些函数，我们将在接下来实现这些函数。

生成函数的 IR 代码

大多数 IR 代码将存在于函数中。IR 代码中的函数类似于 C 中的函数，它指定了名称、参数类型、返回值和其他属性。要在不同的编译单元中调用函数，需要声明函数。这类似于 C 语言中的原型。如果向函数中添加基本块，那么就可以定义函数。我们将在下一节中完成所有这些工作，首先讨论关于符号名的可见性。

使用连接和名称修饰的方式控制可见性

函数 (以及全局变量) 有一个附加的链接样式。使用链接样式，我们定义了符号名的可见性，以及如果多个符号具有相同的名称应该发生什么。最基本的链接样式是私有和外部可访问的。具有私有链接的符号仅在当前编译单元中可见，而具有外部链接的符号则是全局可用的。

对于没有适当模块概念的语言，如 C，这足够了。对于模块，我们需要做更多的工作。假设有一个名为 Square 的模块提供了一个 Root() 函数，而 Cube 模块也提供了一个 Root() 函数。如果函数是私有的，没有问题。该函数获取名称 Root 和私有链接。如果导出函数，情况就不同了，这样就可以在其他模块中调用它。仅使用函数名是不够的，因为函数名不唯一。

解决方案是调整名称，使其具有全局唯一性，这叫做命名修饰。如何做到这一点取决于语言的需求和特征。我们的例子中，基本思想是使用模块名和函数名的组合来创建一个全局唯一名。使用 Square.Root 作为名称看起来是一个简单的解决方案，但可能会导致汇编程序的问题，因为点可能有特殊的含义。我们不需要在名称组件之间使用分隔符，而是可以用名称组件的长度作为前缀:6Square4Root，从而获得类似的效果。这不是 LLVM 的合法标识符，但可以通过在整个名称前加上_t (t 代表 tinylang):_t6Square4Root 来解决这个问题。通过这种方式，我们可以为导出的符号，并创建唯一的名称:

```
1 std::string CGModule::mangleName(Decl *D) {
2     std::string Mangled;
3     llvm::SmallString<16> Tmp;
4     while (D) {
5         llvm::StringRef Name = D->getName();
6         Tmp.clear();
7         Tmp.append(llvm::itostr(Name.size()));
8         Tmp.append(Name);
9         Mangled.insert(0, Tmp.c_str());
10        D = D->getEnclosingDecl();
11    }
12    Mangled.insert(0, "_t");
13    return Mangled;
14 }
```

如果你的源语言支持类型重载，那么需要用类型名来扩展这个方案。例如，为了区分 C++ 函数 *int root(int)* 和 *double root(double)*，形参的类型和返回值需要添加到函数名中。

您还需要考虑生成的名称的长度，因为一些链接器对长度进行了限制。在 C++ 中使用嵌套的名称空间和类，修饰后的名称可能会相当长。所以，C++ 定义了一个压缩方案来避免一遍又一遍地重复命名组件。

接下来，看看如何处理参数类型。

将类型从 AST 描述转换为 LLVM 类型

函数的参数也需要考虑。首先，需要将源语言的类型映射到 LLVM 类型。因为 tinylang 目前只有两种类型：

```
1 llvm::Type *convertType(TypeDeclaration *Ty) {
2     if (Ty->getName() == "INTEGER")
3         return Int64Ty;
4     if (Ty->getName() == "BOOLEAN")
5         return Int1Ty;
6     llvm::report_fatal_error("Unsupported type");
7 }
```

Int64Ty、Int1Ty 和后来的 void 都是类成员，它们持有 LLVM 类型 i64、i1 和 void 的类型表示。

对于通过引用传递的形式参数是不够的，该参数的 LLVM 类型为指针。我们将函数推广并考虑形式参数：

```
1 llvm::Type *mapType(Decl *Decl) {
2     if (auto *FP = llvm::
3         dyn_cast<FormalParameterDeclaration>(
4             Decl)) {
5         llvm::Type *Ty = convertType(FP->getType());
6         if (FP->isVar())
7             Ty = Ty->getPointerTo();
8         return Ty;
9     }
10    if (auto *V = llvm::dyn_cast<VariableDeclaration>(Decl))
11        return convertType(V->getType());
12    return convertType(llvm::cast<TypeDeclaration>(Decl));
13 }
```

有了这些助手函数，接下就来创建 LLVM IR 函数。

创建 LLVM IR 函数

要在 LLVM IR 中计算函数，需要一个函数类型，这类似于 C 中的原型。创建函数类型需要映射类型，然后调用工厂方法来创建函数类型：

```
1 llvm::FunctionType *createFunctionType(
```

```

2  ProcedureDeclaration *Proc) {
3  llvm::Type *ResultTy = VoidTy;
4  if (Proc->getRetType()) {
5      ResultTy = mapType(Proc->getRetType());
6  }
7  auto FormalParams = Proc->getFormalParams();
8  llvm::SmallVector<llvm::Type *, 8> ParamTypes;
9  for (auto FP : FormalParams) {
10     llvm::Type *Ty = mapType(FP);
11     ParamTypes.push_back(Ty);
12 }
13 return llvm::FunctionType::get(ResultTy, ParamTypes,
14                                /* IsVarArgs */ false);
15 }

```

根据函数类型，我们还创建了 LLVM 函数，将函数类型与链接和修饰后的名称关联起来：

```

1  llvm::Function *
2  createFunction(ProcedureDeclaration *Proc,
3                llvm::FunctionType *FTy) {
4  llvm::Function *Fn = llvm::Function::Create(
5      FTy, llvm::GlobalValue::ExternalLinkage,
6      mangleName(Proc), getModule());

```

getModule() 方法返回当前的 LLVM 模块，稍后我们将对其进行设置。

创建这个函数后，可以向它添加更多的信息。首先，可以给出参数的名称。这使得 IR 更具可读性。其次，可以向函数和参数添加属性，以指定一些特征。例如，可以对通过引用传递的参数进行这样的操作。

从 LLVM 层来看，这些参数是指针。但是从源语言设计来看，这些都是非常受限的指针。类似于 C++ 中的引用，需要为 VAR 参数指定一个变量。因此，通过设计，我们知道这个指针永不为空，而且总是可解引用的，这意味着可以冒着保护故障的风险读取指向的值。所以，这个指针不能传递。特别地，没有指针的副本比函数调用的时间周期长。因此，我们说指针没有捕获。

AttributeBuilder 类用于为形参构建属性集。要获得参数类型的存储大小，可以简单地查询数据结构：

```

1  size_t Idx = 0;
2  for (auto I = Fn->arg_begin(), E = Fn->arg_end(); I != E;
3      ++I, ++Idx) {
4      llvm::Argument *Arg = I;
5      FormalParameterDeclaration *FP =
6      Proc->getFormalParams()[Idx];
7      if (FP->isVar()) {

```

```

8     llvm::AttrBuilder Attr;
9     llvm::TypeSize Sz =
10         CGM.getModule()
11         ->getDataLayout().getTypeStoreSize(
12             CGM.convertType(FP->getType()));
13     Attr.addDereferenceableAttr(Sz);
14     Attr.addAttribute(llvm::Attribute::NoCapture);
15     Arg->addAttrs(Attr);
16 }
17 Arg->setName(FP->getName());
18 }
19 return Fn;
20 }

```

现在我们已经创建了 IR 功能。下一节中，我们将函数体的基本块添加到函数中。

生成函数体

我们几乎完成了函数的 IR 代码生成! 我们只需要将各个部分组合在一起就可以生成一个函数，包括函数体:

- 给定 tinylang 中的过程声明，首先创建函数类型和函数:

```

1 void run(ProcedureDeclaration *Proc) {
2     this->Proc = Proc;
3     Fty = createFunctionType(Proc);
4     Fn = createFunction(Proc, Fty);

```

- 接下来，创建函数的第一个基本块，并将其设置为当前块:

```

1     llvm::BasicBlock *BB = llvm::BasicBlock::Create(
2         CGM.getLLVMCtx(), "entry", Fn);
3     setCurr(BB);

```

- 然后遍历所有形式参数。为了正确处理 VAR 参数，需要初始化 FormalParams 成员 (在 readVariable() 中使用)。与局部变量不同，形式参数在第一个基本块中有一个值，所以这些值是已知的:

```

1     size_t Idx = 0;
2     auto &Defs = CurrentDef[BB];
3     for (auto I = Fn->arg_begin(), E = Fn->arg_end(); I !=
4         E; ++I, ++Idx) {
5         llvm::Argument *Arg = I;
6         FormalParameterDeclaration *FP = Proc->
7             getParams()[Idx];

```

```

8     FormalParams[FP] = Arg;
9     Defs.Defs.insert(
10         std::pair<Decl *, llvm::Value *>(FP, Arg));
11 }

```

- 按照这个设定，可以调用 emit() 方法来开始为语句生成 IR 代码:

```

1     auto Block = Proc->getStmts();
2     emit(Proc->getStmts());

```

- 生成 IR 代码后的最后一个块可能还没有密封，因此现在调用 sealBlock()。tinylang 中的过程可能有隐式的 return，所以还要检查最后一个基本块是否有合适的终止符，如果没有，就添加一个:

```

1     sealBlock(Curr);
2     if (!Curr->getTerminator()) {
3         Builder.CreateRetVoid();
4     }
5 }

```

这就完成了函数 IR 代码的生成。我们仍然需要创建 LLVM 模块，它会将所有 IR 代码保存在一起。

设置模块和驱动程序

我们在 LLVM 模块中收集编译单元的所有函数和全局变量。为了方便 IR 的生成，我们将前面几节中的所有函数包装在代码生成器类中。要获得一个编译器，还需要定义要为其生成代码的目标体系结构，并添加生成代码的 Pass。我们将在下一章中实现所有这些，就先从代码生成器开始。

包装代码生成器

IR 模块是我们为编译单元生成的所有元素的大括号。在全局级别，我们遍历模块级别的声明，并创建全局变量，并为过程调用代码生成。tinylang 中的全局变量会映射到 llvm::GlobalValue 类的实例。这个映射保存在 Globals 中，并可用于过程代码的生成:

```

1 void CGModule::run(ModuleDeclaration *Mod) {
2     for (auto *Decl : Mod->getDecls()) {
3         if (auto *Var =
4             llvm::dyn_cast<VariableDeclaration>(Decl)) {
5             llvm::GlobalVariable *V = new llvm::GlobalVariable(
6                 *M, convertType(Var->getType()),
7                 /*isConstant=*/false,
8                 llvm::GlobalValue::PrivateLinkage, nullptr,
9                 mangleName(Var));

```

```

10     Globals[Var] = V;
11 } else if (auto *Proc =
12     llvm::dyn_cast<ProcedureDeclaration>(
13     Decl)) {
14     CGProcedure CGP(*this);
15     CGP.run(Proc);
16 }
17 }
18 }

```

该模块还持有 LLVMContext 类，并缓存最常用的 LLVM 类型。后者需要初始化，例如：对于 64 位整数类型：

```

1 Int64Ty = llvm::Type::getInt64Ty(getLLVMCtx());

```

CodeGenerator 类初始化 LLVM IR 模块，并调用该模块的代码生成。最重要的是，这个类必须知道我们为哪个目标体系结构生成代码。这个信息在 llvm::TargetMachine 类中传递，在驱动程序中设置：

```

1 void CodeGenerator::run(ModuleDeclaration *Mod, std::string
2 FileName) {
3     llvm::Module *M = new llvm::Module(FileName, Ctx);
4     M->setTargetTriple(TM->getTargetTriple().getTriple());
5     M->setDataLayout(TM->createDataLayout());
6     CGModule CGM(M);
7     CGM.run(Mod);
8 }

```

为了方便使用，还为代码生成器引入了工厂方法：

```

1 CodeGenerator *CodeGenerator::create(llvm::TargetMachine *TM) {
2     return new CodeGenerator(TM);
3 }

```

CodeGenerator 类提供了创建 IR 代码的接口，非常适合在编译器驱动程序中使用。在集成之前，需要支持实现对机器码的生成。

初始化目标机器类

现在，只缺少创建目标机器类。在目标机器上，我们定义了用于生成代码的 CPU 体系结构。对于每个 CPU，还可以使用一些可用的特性来影响代码生成，例如：CPU 体系结构家族中较新的 CPU 可以支持向量指令。通过特性，我们可以打开或关闭向量指令的使用。为了支持从命令行设置所有这些选项，LLVM 提供了一些支持代码。在 Driver 类中，添加了以下 include 变量：

```

1 #include "llvm/CodeGen/CommandFlags.h"

```


这个 include 变量为编译器驱动程序添加了常用的命令行选项。许多 LLVM 工具也使用这些命令行选项，好处是为用户提供了一个公共界面 (只缺少指定目标三元组的选项)。由于这非常好用，所以我们添加了这个：

```
1 static cl::opt<std::string>
2   MTriple("mtriple",
3   cl::desc("Override target triple for module"));
```

创建目标机器码：

- 为了显示错误消息，应用程序的名称必须传递给函数：

```
1 llvm::TargetMachine *
2 createTargetMachine(const char *Argv0) {
```

- 收集命令行提供的所有信息。以下是代码生成器的选项、CPU 的名称、应该开启 (或关闭) 的可能特性以及目标的“三元组”：

```
1   llvm::Triple = llvm::Triple(
2       !MTriple.empty()
3       ? llvm::Triple::normalize(MTriple)
4       : llvm::sys::getDefaultTargetTriple());
5
6   llvm::TargetOptions =
7       codegen::InitTargetOptionsFromCodeGenFlags(Triple);
8   std::string CPUStr = codegen::getCPUStr();
9   std::string FeatureStr = codegen::getFeaturesStr();
```

- 在目标注册表中查找目标。如果发生错误，则显示错误消息并退出。一个可能的错误是，用户指定的不支持的三元组：

```
1   std::string Error;
2   const llvm::Target *Target =
3       llvm::TargetRegistry::lookupTarget(
4           codegen::getMArch(), Triple,
5           Error);
6
7   if (!Target) {
8       llvm::WithColor::error(llvm::errs(), Argv0) <<
9           Error;
10      return nullptr;
11  }
```

- 在 Target 类的帮助下，使用用户请求的所有已知选项配置目标机器：

```

1  llvm::TargetMachine *TM = Target->
2      createTargetMachine(
3          Triple.getTriple(), CPUStr, FeatureStr,
4          TargetOptions,
5          llvm::Optional<llvm::Reloc::Model>(
6              codegen::getRelocModel()));
7  return TM;
8  }

```

通过目标机器实例，可以生成针对我们选择的 CPU 架构的 IR 代码。缺少的是对程序集文本的转换或目标代码文件的生成。我们将在下一节中添加这种支持。

生成汇编程序文本和目标代码

在 LLVM 中，IR 代码通过 Pass 运行。每一遍执行一个任务，例如：删除死代码。我们将在第 8 章，优化 IR 中学习更多关于 Pass 的内容。输出汇编程序代码或目标文件也可以实现为 Pass。

需要 `llvm::legacy::PassManager` 类来保存 Pass，以便将代码发送到文件。还希望能够输出 LLVM IR 代码，因此也需要一个 Pass 来做这个。最后，使用 `llvm::ToolOutputFile` 类进行文件操作：

```

1  #include "llvm/IR/IRPrintingPasses.h"
2  #include "llvm/IR/LegacyPassManager.h"
3  #include "llvm/Support/ToolOutputFile.h"

```

输出 LLVM IR 的另一个命令行选项也是必需的：

```

1  static cl::opt<bool>
2      EmitLLVM("emit-llvm",
3          cl::desc("Emit IR code instead of assembler"),
4          cl::init(false));

```

新的 `emit()` 方法中的第一个任务是处理输出文件的名称。如果从 `stdin` 读取输入 (用减号-表示)，则将结果输出到 `stdout`。`ToolOutputFile` 类知道如何处理特殊文件名，`-`：

```

1  bool emit(StringRef Argv0, llvm::Module *M,
2          llvm::TargetMachine *TM,
3          StringRef InputFilename) {
4      CodeGenFileType FileType = codegen::getFileType();
5      std::string OutputFilename;
6      if (InputFilename == "-") {
7          OutputFilename = "-";
8      }

```

否则，根据用户给出的命令行选项，我们将删除输入文件名的可能扩展名，并附加.ll、.s 或.o 作为扩展名。FileType 选项在 llvm/CodeGen/CommandFlags.inc 头文件中定义，我们之前包含了它。这个选项不支持发出 IR 代码，所以我们添加了新的选项-emit-llvm，只有在与汇编文件一起使用时才会生效：

```
1  else {
2      if (InputFilename.endswith(".mod"))
3          OutputFilename = InputFilename.drop_back(4).str();
4      else
5          OutputFilename = InputFilename.str();
6      switch (FileType) {
7      case CGFT_AssemblyFile:
8          OutputFilename.append(EmitLLVM ? ".ll" : ".s");
9          break;
10     case CGFT_ObjectFile:
11         OutputFilename.append(".o");
12         break;
13     case CGFT_Null:
14         OutputFilename.append(".null");
15         break;
16     }
17 }
```

有些平台区分文本文件和二进制文件，所以必须在打开输出文件时提供正确的标志：

```
1  std::error_code EC;
2  sys::fs::OpenFlags = sys::fs::OF_None;
3  if (FileType == CGFT_AssemblyFile)
4      OpenFlags |= sys::fs::OF_Text;
5  auto Out = std::make_unique<llvm::ToolOutputFile>(
6      OutputFilename, EC, OpenFlags);
7  if (EC) {
8      WithColor::error(errs(), Argv0) << EC.message() <<
9          '\n';
10     return false;
11 }
```

现在我们可以将所需的 Pass 添加到 PassManager 中。TargetMachine 类有一个实用程序方法，用于添加请求的类。因此，我们只需要检查用户是否请求输出 LLVM IR 代码：

```
1  legacy::PassManager PM;
2  if (FileType == CGFT_AssemblyFile && EmitLLVM) {
3      PM.add(createPrintModulePass(Out->os()));
```

```

4 } else {
5     if (TM->addPassesToEmitFile(PM, Out->os(), nullptr,
6                               FileType)) {
7         WithColor::error() << "No support for file type\n";
8         return false;
9     }
10 }

```

所有这些准备工作完成后，生成文件就可以归结为一个函数调用：

```

1 PM.run(*M);

```

如果不显式地保留该文件，那么 `ToolOutputFile` 类会自动删除该文件。这使得错误处理更容易，因为可能有很多地方需要处理错误，而在一切顺利的情况下只需要到达相应的期望。这里，我们成功地生成了代码，所以想要保留这个文件：

```

1 Out->keep();

```

最后，我们向调用者报告成功：

```

1 return true;
2 }

```

使用 `llvm::Module` 调用 `emit()` 方法，并调用 `CodeGenerator` 类，将按照请求生成代码。

假设在 `tinylang` 中有最大公约数算法存储在 `gcd.mod` 文件中。把它翻译成 `gcd.os` 目标文件，需要输入以下内容：

```
$ tinylang -filetype=obj gcd.mod
```

如果想在屏幕上直接检查生成的 IR 代码，可以输入以下代码：

```
$ tinylang -filetype=asm -emit-llvm -o -gcd.mod
```

让我们好好庆祝一下吧！至此，已经创建了一个完整的编译器，从读取源语言到生成汇编代码或目标文件。

总结

本章中，学习了如何为 LLVM IR 代码实现你自己的代码生成器。基本块是一种重要的数据结构，包含所有的指令并表示分支。您学习了如何为源语言的控制语句创建基本块，以及如何向基本块添加指令。您应用了一种现代算法来处理函数中的局部变量，从而减少了 IR 代码。编译器的目标是为输入生成汇编文本或目标文件，因此还添加了一个简单的编译 Pass。有了这些知识，您将能够生成 LLVM IR，并随后为您自己的语言编译器生成汇编文本或目标代码。

下一章中，您将学习如何处理聚合数据结构，以及如何确保函数调用符合规则。

第 6 章 生成高级语言结构的 IR

目前，高级语言通常使用聚合数据类型和面向对象编程 (OOP) 构造。LLVM IR 对聚合数据类型有一定的支持，必须自己实现 OOP 构造。添加聚合类型会引起如何传递聚合类型参数的问题。不同的平台有不同的规则，这也体现在 IR 中。遵循调用约定可以确保系统函数可以调用。

本章中，您将学习如何转换聚合数据类型和指向 LLVM IR 的指针，以及如何以系统兼容的方式将参数传递给函数。您还将学习如何在 LLVM IR 中实现类和虚函数。

本章将包含以下内容：

- 使用数组、结构和指针
- 获取正确的二进制接口
- 为类和虚函数创建 IR 代码

在本章结束时，您将学习如何为聚合数据类型和 OOP 创建 LLVM IR。您还会了解如何根据平台规则传递聚合数据类型。

相关代码

本章的代码文件可在<https://github.com/PacktPublishing/Learn-LLVM-12/tree/master/Chapter06/tinylang>获取。

你可以在视频中找到代码<https://bit.ly/3nllhED>。

使用数组、结构体和指针

对于所有的应用程序，基本类型 (如 INTEGER) 是不够用的。例如，要表示矩阵或复数等数学对象，必须基于现有数据类型构造新的数据类型。这些新的数据类型通常称为聚合或复合类型。

数组是同一类型元素的序列。在 LLVM 中，数组总是静态的：元素的数量是常量。整数数组 [10] 的 tinylang 类型，或长整数数组 [10] 的 C 类型，可以用 IR 表示为：

```
[10 x i64]
```

结构是不同类型的复合。在编程语言中，通常用命名成员来表示，例如：在 tinylang 中，结构写为 RECORD x, y: REAL; color: INTEGER; END; 同样的结构在 C 中是 struct {float x, y; long color;};。在 LLVM IR 中，只列出了类型名：

```
{ float, float, i64 }
```

要访问成员，需要使用数字索引。与数组一样，第一个元素的索引号是 0。

此结构的成员根据数据布局字符串中的规范在内存中进行布局。如果有必要，将插入未使用的填充字节。如果需要控制内存布局，可以使用打包结构，其中所有元素都是 1 字节对齐的。语法略有不同：

```
< float, float, i64 >
```

数组和结构可以作为一个单元来处理，例如：不能将%x 数组的单个元素以%x[3] 引用。这时因为 SSA 的形式不能分辨%x[i] 和%x[j] 是否指相同的元素。相反，需要特殊的指令来提取和插入单个元素的值到数组中。要读取第二个元素，可以使用以下语句：

```
%el2 = extractvalue [10 x i64] %x, 1
```

我们也可以更新一个元素，例如：第一个元素：

```
%xnew = insertvalue [10 x i64] %x, i64 %el2, 0
```

这两种指令也适用于结构体。例如，要从%pt 寄存器访问 color 成员，需要编写以下代码：

```
%color = extractvalue float, float, i64 %pt, 2
```

这两个指令都有一个限制：索引必须是常数。对于结构来说，这很容易解释。索引号只是名称的替代，而 C 等语言没有动态计算结构成员名称的概念。对于数组，只是不能有效地实现。这两种指令在特定的情况下都有意义，当元素的数量很小且已知时，例如：复数可以建模为两个浮点数的数组。传递这个数组是合理的，并且在计算过程中访问数组的哪一部分是很清楚的。

对于前端的一般使用，需要求助于指向内存的指针。LLVM 中的所有全局值都表示为指针，声明一个全局变量 @arr，作为一个包含 8 个 i64 元素的数组，等价于 long arr[8] 的 C 声明：

```
@arr = common global [8 x i64] zeroinitializer
```

要访问数组的第二个元素，必须执行地址计算，以确定索引的地址。然后，可以从该地址加载该值。将其放入 @second 函数中：

```
define i64 @second()  
    %1 = getelementptr [8 x i64], [8 x i64]* @arr, i64 0, i64  
    1    %2 = load i64, i64* %1  
    ret i64 %2
```

getelementptr 指令是地址计算的主要工具，所以需要对其进行更多的介绍。第一个操作数 [8 x i64] 是该指令操作的基类型。第二个操作数 [8 x i64]* @arr 指定基指针。注意这里的细微差别：我们声明了一个包含 8 个元素的数组，而所有全局值都视为指针，所以有一个指向数组的指针。C 语法中，我们使用 long (*arr)[8]！其结果是，在对元素进行索引之前，首先必须对指针进行解引用，例如：C 中的 arr[0][1]。第三个操作数 i64 0 对指针进行解引用，第四个操作数 i64 1 是元素的下标。计算的结果是索引元素的地址。请注意，本指令没有涉及内存。

除了结构之外，索引参数不需要是常量。因此，可以在循环中使用 getelementptr 指令来检索数组的元素。这里对结构的处理是不同的：只能使用常量，类型必须是 i32。

有了这些知识，数组很容易集成到第 5 章的代码生成器中。必须扩展 `convertType()` 方法来创建类型，如果 `Arr` 变量持有数组的类型指示符，那么可以在方法中添加以下内容：

```
1 llvm::Type *Component = convertType(Arr->getComponentType());
2 uint64_t NumElements = Arr->getNumElem();
3 return llvm::ArrayType::get(Component, NumElements);
```

该类型可用于声明全局变量。对于局部变量，需要为数组分配内存。在过程的第一个基本块中可以完成这个操作：

```
1 for (auto *D : Proc->getDecls()) {
2     if (auto *Var =
3         llvm::dyn_cast<VariableDeclaration>(D)) {
4         llvm::Type *Ty = mapType(Var);
5         if (Ty->isAggregateType()) {
6             llvm::Value *Val = Builder.CreateAlloca(Ty);
7             Defs.Defs.insert(
8                 std::pair<Decl *, llvm::Value *>(Var, Val));
9         }
10    }
11 }
```

要读写一个元素，必须生成 `getelemtptr` 指令。这个指令会添加到 `emitExpr()`(读取值) 和 `emitAssign()`(写入值) 的方法中。要读取数组元素，首先读取变量的值，然后处理变量的选择器。对于每个索引，计算表达式并存储其值。基于这个列表，计算引用元素的地址并加载值：

```
1 auto &Selectors = Var->getSelectorList();
2 for (auto *I = Selectors.begin(),
3     *E = Selectors.end();
4     I != E;) {
5     if (auto *Idx = llvm::dyn_cast<IndexSelector>(*I)) {
6         llvm::SmallVector<llvm::Value *, 4> IdxList;
7         IdxList.push_back(emitExpr(Idx->getIndex()));
8         for (++I; I != E;) {
9             if (auto *Idx2 =
10                 llvm::dyn_cast<IndexSelector>(*I)) {
11                 IdxList.push_back(emitExpr(Idx2->getIndex()));
12                 ++I;
13             } else
14                 break;
15         }
16         Val = Builder.CreateGEP(Val, IdxList);
17         Val = Builder.CreateLoad(
```

```

18     Val->getType()->getPointerElementType(), Val);
19 } else {
20     llvm::report_fatal_error("Unsupported selector");
21 }
22 }

```

写入数组元素使用相同的代码，但不生成加载指令。相反，在存储指令中使用指针作为目标。对于记录，可以使用类似的方法。记录成员的选择器包含常量字段索引，名为 `Idx`。可以通过以下方法将这个常量转换为 LLVM 常量值：

```

1 llvm::Value *FieldIdx = llvm::ConstantInt::get(Int32Ty, Idx);

```

然后，您可以像使用数组一样使用 `Builder.CreateGEP()` 方法中的值。

现在您已经掌握了将聚合数据类型转换为 LLVM IR 的方法。以系统兼容的方式传递这些类型的值需要注意一些细节，在下一节中您将学习如何正确地实现它。

正确的获取 ABI（应用程序二进制接口）

使用最新添加的数组和记录到代码生成器中，您可能会注意到有时生成的代码不能按预期执行。原因是目前为止我们忽略了平台的调用规则。对于同一个程序或库中的一个函数如何调用另一个函数，每个平台都定义了自己的规则。这些规则在应用程序二进制接口 (ABI) 文档中进行了总结。通常的信息包括：

- 是否使用机器寄存器传递参数？如果是，是哪个？
- 如何将数组和结构体等聚合体传递给函数？
- 如何处理返回值？

在使用中有各种各样的规则时，在一些平台上，聚合体总是间接传递，是将聚合体的副本放在堆栈上，只将指向该副本的指针作为参数传递。在其他平台上，在寄存器中传递小的聚合体（比如 128 位或 256 位宽），只有超过阈值才使用间接参数传递。有些平台还使用浮点和向量寄存器传递参数，而有些平台要求浮点值在整数寄存器中传递。

当然，这些都是很有趣的、很底层的东西。不幸的是，它与 LLVM IR 相关。这还挺令人惊讶的，毕竟我们在 LLVM IR 中定义了函数的所有参数的类型！事实证明，这还不够。为了理解这一点，来考虑下复数。有些语言有用于复数的内置数据类型，例如：C99 有 `float_Complex`(等等)。旧版本的 C 没有复数类型，但可以很容易地定义 `struct complex {float re, im;}` 并在该类型上创建算术运算。这两种类型都可以映射到 `{float, float}` LLVM IR 类型。如果 ABI 现在声明内置复数类型的值在两个浮点寄存器中传递，但用户定义的聚合体总是间接传递，那么该函数提供的信息不足以让 LLVM 决定如何传递这个特定参数。不幸的是，我们需要向 LLVM 提供更多信息，而这些是高度特定于 ABI 的信息。

有两种方法可以将此信息指定给 LLVM：参数属性和类型重写。您需要使用的内容取决于目标平台和代码生成器，常用的参数属性如下：

- `inreg` 指定参数在寄存器中传递。
- `byval` 指定按值传递参数。形参必须是指针类型。指向数据的隐藏副本，并传递给调用的函数。

- zeroext 和 signext 指定传递的整数值应为零或扩展符号。
- sret 此形参保存一个指向内存的指针，该指针用于从函数返回聚合体类型。

虽然所有代码生成器都支持 zeroext、signext 和 sret 属性，但只有部分代码生成器支持 inreg 和 byval。可以使用 addAttr() 方法将属性添加到函数的参数中，例如：要在 Arg 参数上设置 inreg 属性，可以调用以下方法：

```
1 Arg->addAttr(llvm::Attribute::InReg);
```

要设置多个属性，可以使用 llvm::AttrBuilder 类。

提供附加信息的另一种方法是使用类型重写。使用这种方法，可以隐藏原始类型：

- 拆分参数，例如：可以传递两个浮点参数，而不是传递一个复杂参数。
- 将参数转换为不同的表示形式，例如：将大小为 64 位或更小的结构体转换为 i64 整数。

在不改变值位数的情况下，在不同类型之间进行转换，可以使用位转换指令。bitcast 指令不会对聚合类型进行操作，但这不是一个限制，因为您可以使用指针。如果一个聚合体为一个具有两个 int 成员的结构体，在 LLVM 中表示为类型 {i32, i32}，那么这个聚合体可以按以下方式位转换为 i64：

```
%intpoint = bitcast i32, i32* %point to i64*
```

这将把指向结构体的指针转换为指向 i64 整数值的指针。随后，可以加载该值并将其作为参数传递。必须确保这两种类型具有相同的大小。

向参数添加属性或更改类型并不复杂。但是，如何知道需要实现什么呢？首先，应该大致了解目标平台上使用的调用约定，例如：Linux 上的 ELF ABI 记录了每种支持的 CPU 平台（只要只需要查一下文件）。有关于 LLVM 代码生成器需求的文档，信息的来源是 Clang 实现，在<https://github.com/llvm/llvm-project/blob/main/clang/lib/CodeGen/TargetInfo.cpp>文件中。这个文件包含所有受支持平台的特定于 ABI 的操作。

本节中，您学习了如何为函数调用生成符合平台 ABI 的 IR。下一节将介绍为类和虚函数创建 IR 的不同方法。

为类和虚函数创建 IR 代码

许多现代编程语言都支持使用类面向对象方式。类是一种高级语言构造，在本节中，我们将探讨如何将类构造映射到 LLVM IR 中。

实现单继承

类是数据和方法的集合。一个类可以从另一个类继承，可能会添加更多的数据字段和方法，或者覆盖现有的虚方法。我们用 Oberon-2 中的类来说明这一点，它也是 tinylang 的一个模型。Shape 类定义了一个带有颜色和区域的抽象形状：

```
TYPE Shape = RECORD
    color: INTEGER;
    PROCEDURE (VAR s: Shape) GetColor():
        INTEGER;
    PROCEDURE (VAR s: Shape) Area(): REAL;
END;
```

GetColor 方法只返回颜色编号:

```
PROCEDURE (VAR s: Shape) GetColor(): INTEGER;
BEGIN RETURN s.color; END GetColor;
```

抽象的形狀的面积是无法计算的，所以这是一个抽象方法:

```
PROCEDURE (VAR s: Shape) Area(): REAL;
BEGIN HALT; END;
```

Shape 类型可以扩展为表示 Circle 类:

```
TYPE Circle = RECORD (Shape)
    radius: REAL;
    PROCEDURE (VAR s: Circle) Area(): REAL;
END;
```

对于圆，面积公式为:

```
PROCEDURE (VAR s: Circle) Area(): REAL;
BEGIN RETURN 2 * radius * radius; END;
```

还可以在运行时查询该类型。如果 shape 是 Shape 类型的变量，则可以这样制定类型测试:

```
IF shape IS Circle THEN (* ...*) END;
```

除了语法不同之外，工作原理与 C++ 非常相似。与 C++ 的明显区别是，Oberon-2 语法使隐式的 this 指针显式化，将其称为接收方方法。

需要解决的基本问题是如何在内存中布局类，以及如何实现方法的动态调用和运行时类型检查。对于内存布局，这是相当容易的。Shape 类只有一个数据成员，可以将它映射到相应的 LLVM 结构类型:

```
@Shape = type { i64 }
```

Circle 类添加了另一个数据成员，直接在末尾添加新的数据成员：

```
@Circle = type { i64, float }
```

原因是 一个类可以有很多子类。使用这种策略，公共基类的数据成员总是具有相同的内存偏移量，并使用相同的索引通过 `getelementptr` 指令访问字段。

为了实现方法的动态调用，必须进一步扩展 LLVM 结构。如果在 Shape 对象上调用 `Area()` 函数，则会调用抽象方法，导致应用程序停止。如果在 Circle 对象上调用该函数，则调用相应的方法来计算圆的面积。这两个类的对象都可以调用 `GetColor()` 函数。实现这个的基本思想是将表与函数指针与每个对象关联起来。这里，该表有两个条目：一个用于 `GetColor()` 方法，一个用于 `Area()` 函数。Shape 类和 Circle 类都有这样一个表。这两个表在 `Area()` 函数的条目上有所不同，该函数根据对象的类型调用不同的代码。这个表称为虚方法表，通常缩写为 `vtable`。

单独使用 `vtable` 是没有用的，我们必须把它和一个物体联系起来。为此，可以添加一个指向虚函数表的指针，该指针总是作为结构的第一个数据成员。在 LLVM 层，`@Shape` 类型变成如下表示：

```
@Shape = type { [2 x i8*]*, i64 }
```

`@Circle` 类型的扩展也是类似的，得到的内存结构如图 6.1 所示：

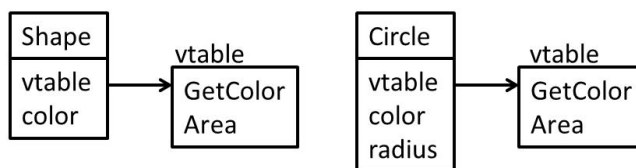


图 6.1 -类和虚拟方法表的内存布局

LLVM 没有空指针，而是使用指向字节的指针。引入了隐藏的虚函数表字段后，现在还需要一种方法来初始化它。在 C++ 中，这是构造函数的一部分。在 Oberon-2 中，该字段在分配内存时自动初始化。

然后按以下步骤执行对方法的动态调用：

1. 通过 `getelementptr` 指令计算虚表指针的偏移量。
2. 加载指向虚函数表的指针。
3. 计算虚函数表中函数的偏移量。
4. 加载函数指针。
5. 通过带有调用指令的指针间接调用函数。

这听起来不是很高效，但事实上，大多数 CPU 架构只需要两条指令就可以执行这个动态调用。因此，真正低效的是 LLVM 层。

要将函数转换为方法，需要对对象数据的引用。这是通过将指向数据的指针，作为函数的第一个参数来实现的。在 Oberon-2 中，这就是显式接收器。类似于 C++ 的语言中的 `this` 指针。

有了虚函数表，每个类在内存中都有唯一的地址。这对运行时类型测试有帮助吗？的确有帮助，但帮助有限。为了说明这个问题，我们用继承自 Circle 类的 Ellipse 类来扩展类层次结构（这不是

数学意义上的 is-a 关系)。如果有 Shape 类型的 shape 变量，那么可以实现 shape IS Circle 的类型测试，将 shape 变量中存储的虚函数表指针与 Circle 类中的虚函数表指针进行比较。只有当 shape 具有精确的 Circle 类型时，这个比较才会得到 true。但是如果 shape 是 Ellipse 类型的，那么将返回 false(即使 Ellipse 类型的对象，可以在只需要 Circle 类型的对象的地方使用)。

显然，我们需要做得更多。解决方案是使用运行时类型信息扩展虚拟函数表，需要存储多少信息取决于源语言。为了支持运行时类型检查，只要存储一个指向基类的虚函数表的指针就足够了，如图 6.2 所示：

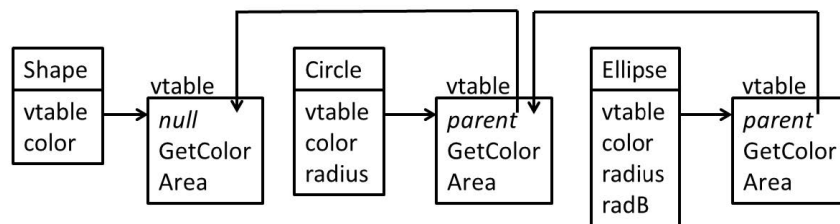


图 6.2 -支持简单类型测试的类和虚函数表布局

若测试像前面描述的那样失败，则使用指向基类的虚函数表的指针重复测试。一直重复，直到测试结果为 true，如果没有基类，则为 false。与调用动态函数相比，类型测试是一种开销很大的操作，因为在最坏的情况下，继承层次结构会逐步上升到根类。

如果您知道整个类层次结构，那么就可能有一种有效的方法：按照深度优先的顺序为类层次结构的每个成员编号。然后，类型测试就变成了对一个数字或一个区间的比较，可以在固定时间内完成。事实上，这就是 LLVM 自己的运行时类型测试的方法，我们在前一章已经学习过了。

将运行时类型信息与虚函数表耦合是一个设计上的决策，要么由源语言强制执行，要么只是一个实现。如果您需要详细的运行时类型信息，因为源语言支持运行时反射，而您的数据类型没有虚函数表，那么耦合两者就不是一个好主意。在 C++ 中，这种耦合出现在一种情况下，即带有虚函数 (因此没有虚函数表) 的类没有附加运行时类型数据。

编程语言通常支持接口，接口是虚拟方法的集合。接口很重要，它们添加了有用的抽象。我们将在下一节中讨论接口的可能实现。

使用接口扩展单继承

像 Java 这样的语言支持接口。接口是抽象方法的集合，类似于没有数据成员且只定义抽象方法的基类。接口带来了一个有趣的问题，因为每个实现接口的类在虚函数表的不同位置都有相应的方法。原因很简单，虚函数表中函数指针的顺序派生自源语言中类定义中函数的顺序。接口中的定义与此无关，不同的顺序才是重点。

因为在接口中定义的方法可以有不同的顺序，所以将每个实现的接口的表附加到类中。对于接口的每个方法，该表既可以指定方法在虚表中的索引，也可以指定存储在虚表中的函数指针的副本。如果在接口上调用方法，则搜索接口对应的虚函数表，然后获取函数的指针并调用方法。将两个接口 I1 和 I2 添加到 Shape 类中会得到如下布局：

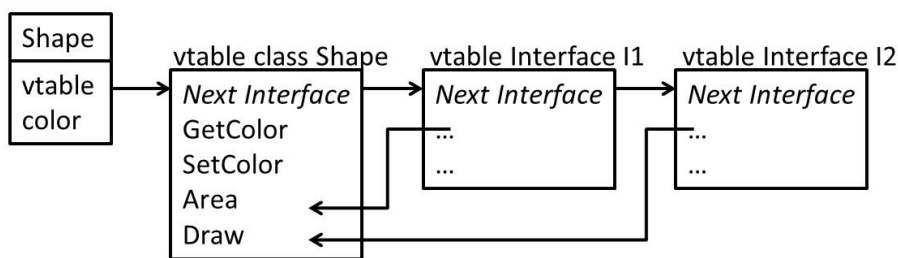


图 6.3 -接口的虚函数表布局

需要注意的是，我们必须找到正确的虚变量表。可以使用类似于运行时类型测试的方法：可以通过接口虚函数表列表执行线性搜索。使用这个数字标识虚函数表，可以为每个接口分配一个唯一的数字（例如，内存地址）。这种模式的缺点很明显：通过接口调用方法要比在类上调用相同的方法花费更多的时间。要解决这个问题并不容易。

一个好的方法是用哈希表代替线性搜索。在编译时，类实现的接口已知。因此，可以构造一个哈希函数，它将接口编号映射到接口的虚函数表。在构造过程中可能需要一个已知标识接口的数字，因此内存没有帮助。但还有其他计算唯一数字的方法。如果源中的符号名是唯一的，那么可以计算加密哈希值，如符号的 MD5，并使用哈希值作为数字。计算发生在编译时，因此没有运行时成本。

结果比线性搜索快得多，只需要常数时间。尽管如此，仍然涉及对数字的几个算术运算，比类类型的方法调用要慢。

通常，接口也会参与运行时类型测试，这使得需要搜索的列表更长。当然，如果实现了哈希表方法，那么它也可以用于运行时类型测试。

有些语言允许有多个父类。这对实现有一些挑战，我们将在下一节学习。

对多重继承的支持

多重继承增加了另一个挑战。如果一个类继承了两个或更多的基类，那么我们需要组合数据成员，使它们仍然可以从函数中访问。与单继承的情况一样，解决方案是附加所有数据成员，包括隐藏的虚函数表指针。Circle 类不仅是一个几何形状，也是一个图形对象。为了对此进行建模，让 Circle 类继承 Shape 类和 GraphicObj 类。在类布局中，来自 Shape 类的字段最先出现。然后，附加 GraphicObj 类的所有字段，包括隐藏的虚表指针。之后，添加了 Circle 类的新数据成员，得到了如图 6.4 所示的整体结构：

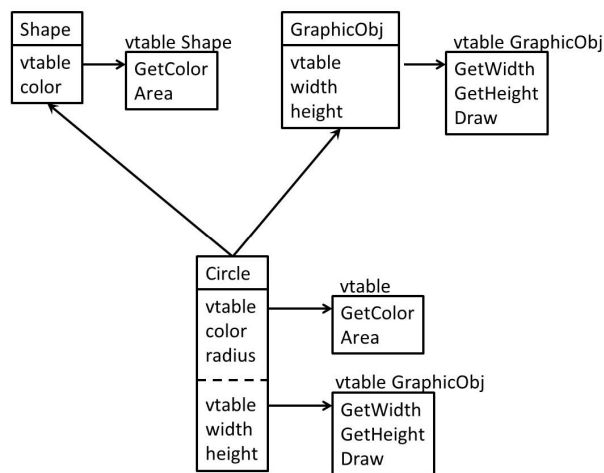


图 6.4 - 具有多重继承的类和虚函数表

这种函数有几个含义。现在可以有几个指向该对象的指针。指向 Shape 或 Circle 类的指针指向对象的顶部，而指向 GraphicObj 类的指针指向对象内部，指向嵌入的 GraphicObj 对象的开头。在比较指针时必须考虑这一点。

调用虚函数也会受到影响。如果函数是在 GraphicObj 类中定义的，那么这个函数需要 GraphicObj 类的类布局。如果这个函数没有在 Circle 类中重写，那么有两种可能。简单的情况是，如果函数调用是通过一个指向 GraphicObj 实例的指针完成的：在 GraphicObj 类的虚表中查找方法的地址，然后调用函数。更复杂的情况是使用指向 Circle 类的指针调用函数。同样，可以在 Circle 类的虚表中查找函数的地址。调用的函数需要一个指向 GraphicObj 类实例的 this 指针，因此我们也必须调整该指针。我们可以这样做，因为已知 GraphicObj 类在 Circle 类中的偏移量。

如果在 Circle 类中重写了 GraphicObj 的函数，通过指向 Circle 类的指针调用该方法，则不需要做任何特殊操作。但是，如果该方法是通过指向 GraphicObj 实例的指针调用的，就需要进行另一个调整，因为该方法需要一个 this 指针指向 Circle 实例。编译时，我们无法计算这个调整，因为不知道这个 GraphicObj 实例是否是多重继承层次结构的一部分。为了解决这个问题，我们将在调用函数之前对 this 指针所做的调整与虚函数表中的每个函数指针一起存储起来，如图 6.5 所示：

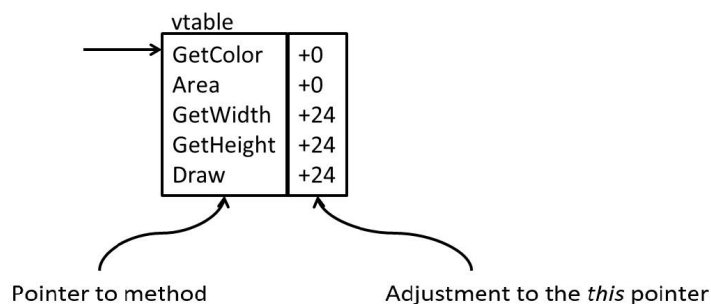


图 6.5 - 调整 this 指针的虚函数表

函数调用现在变成如下方式：

1. 在虚函数表中查找函数指针。
2. 调整 this 指针。

3. 调用该方法。

这种方法还可以用于实现接口。因为接口只有方法，所以每个实现的接口都会向对象添加一个新的虚函数表指针。这更容易实现，而且可能更快，但它增加了每个对象实例的开销。在最坏的情况下，如果你的类有一个 64 位数据字段，实现了 10 个接口，那么你的对象需要 96 个字节的内存：类本身的 vtable 指针需要 8 个字节，数据成员需要 8 个字节，每个接口的 vtable 指针需要 $10 * 8$ 个字节。

为了支持对对象的有意义的比较并执行运行时类型测试，需要首先对对象的指针进行标准化。如果我们在虚表中添加一个额外的字段，在对象的顶部包含一个偏移量，那么我们总是可以调整指针指向实际对象。在 Circle 类的虚函数表中，这个偏移量是 0，但在嵌入式 GraphicObj 类的虚函数表中不是。当然，这是否需要实现取决于源语言的语义。

LLVM 本身并不支持面向对象特性的特殊实现。如本节所见，可以使用可用的 LLVM 数据类型实现所有方法。如果想尝试一种新的方法，那么一个好方法是先用 C 做一个原型。所需的指针操作可以快速地转换为 LLVM IR，但是在高级语言中进行功能性的推理会更容易。

通过本节，您可以在自己的代码生成器中将编程语言中所有 OOP 构造下沉至 LLVM IR 中。您已经了解了如何表示单继承、使用接口的单继承或内存中的多继承，以及如何实现类型测试和如何查找虚函数的方法，这些都是 OOP 语言的核心概念。

总结

本章中，您学习了如何将聚合数据类型和指针转换为 LLVM IR 代码。您还了解了 ABI 的复杂性。最后，您了解了将类和虚函数转换为 LLVM IR 的不同方法。有了本章的知识，您将能够为大多数真实的编程语言创建一个 LLVM IR 代码生成器。

下一章中，您将学习一些高级技术。异常处理在现代编程语言中相当常见，LLVM 对它有一定的支持。将类型信息附加到指针可以进行某些优化，所以我们将添加支持。最后，调试应用程序的能力对于许多开发人员来说是必不可少的，因此我们将在代码生成器中添加调试元数据的生成。

第 7 章 生成高级语言结构的 IR

通过前面章节介绍的中间表示 (IR) 生成，您已经可以实现编译器中所需的大部分功能。本章中，我们将讨论一些实际编译器中出现的主题，例如：许多现代语言都使用的异常处理，我们将研究如何将其转换为底层 (LLVM)IR。

为了支持 LLVM 优化器在某些情况下生成更好的代码，我们向 IR 代码添加了额外的类型元数据，并附加调试元数据，使编译器的用户能够利用源代码级调试工具。

本章中，您将学习以下内容：

- 在抛出和捕获异常中，您将学习如何在编译器中实现异常处理
- 在为基于类型的别名分析生成元数据中，将添加额外的元数据到 LLVM IR，这有助于 LLVM 更好地优化代码。
- 添加调试元数据，实现向生成的 IR 代码添加调试信息所需的类。

本章的最后，将了解关于异常处理和基于类型别名分析，以及调试信息的元数据的知识。

相关代码

本章的代码文件可在<https://github.com/PacktPublishing/Learn-LLVM-12/tree/master/Chapter07>获取。

你可以在视频中找到代码<https://bit.ly/3nllhED>。

抛出和捕获异常

LLVM IR 中的异常处理与平台的支持密切相关。这里，我们将了解 libunwind 中最常见的异常处理方式。它能在 C++ 中使用了，所以我们首先看一个 C++ 中的例子，其中 bar() 函数抛出一个 int 或 double 值，如下所示：

```
1 int bar(int x) {  
2     if (x == 1) throw 1;  
3     if (x == 2) throw 42.0;  
4     return x;  
5 }
```

foo() 函数调用 bar()，但只处理抛出的 int 值。foo() 函数声明它只抛出 int 值，如下所示：

```
1 int foo(int x) throw(int) {  
2     int y = 0;  
3     try {  
4         y = bar(x);  
5     }  
6     catch (int e) {  
7         y = e;  
8     }  
9     return y;  
}
```


抛出异常需要对运行时库进行两次调用。首先，调用 `_cxa_allocate_exception()` 为异常分配内存，这个函数以要分配的字节数作为参数，异常 (示例中的 `int` 或 `double` 值) 会复制到分配的内存中，然后调用 `_cxa_throw()` 引发异常。这个函数有三个参数: 指向已分配异常的指针; 关于异常类型信息; 以及指向析构函数的指针 (如果异常有效负载有的话)。函数的作用是: `_cxa_throw()` 函数启动堆栈展开过程，并且不返回。在 LLVM IR 中，对 `int` 值做如下处理:

```
%eh = tail call i8* @_cxa_allocate_exception(i64 4)
%payload = bitcast i8* %eh to i32*
store i32 1, i32* %payload
tail call void @_cxa_throw(i8* %eh,
                          i8* bitcast (i8** @_ZTIi to i8*), i8*
                          null)
unreachable
```

`_ZTIi` 是描述 `int` 类型的信息。对于双精度类型，使用的是 `ZTIId`。对 `_cxa_throw()` 的调用标记在尾部，因为它是这个函数中的最后一个调用，或许可以重用当前堆栈帧。

到目前为止，还没有针对 `llvm` 进行任何操作。变化会发生在 `foo()` 函数中，因为对 `bar()` 的调用可能会引发异常。如果是 `int` 类型异常，则必须将控制流转移到 `catch` 子句的 IR 代码上。为了实现这一点，必须使用 `invoke` 调用指令而不是 `call` 调用指令，具体的方式如下面的代码所示:

```
%y = invoke i32 @_Z3bari(i32 %x) to label %next
      unwind label %lpad
```

这两个指令之间的区别在于 `invoke` 调用有两个关联的标签。第一个标签是调用函数正常结束后继续执行的地方，通常是 `ret` 指令。在之前的示例中，这个标签称为 `%next`。如果产生异常，则在带有 `%lpad` 标签，所谓的“着陆垫”上继续执行。

“着陆垫”是一个基本模块，必须以 `landingpad` 指令开始，着陆指令向 LLVM 提供关于已处理异常类型的信息。对于 `foo()` 函数，给出了以下信息:

```
lpad:
%exc = landingpad i8*, i32
      cleanup
      catch i8* bitcast (i8** @_ZTIi to i8*)
      filter [1 x i8*] [i8* bitcast (i8** @_ZTIi to
      i8*)]
```

这里有三种可能的行动类型，概述如下:

1. `cleanup`: 这表示清除当前状态的代码。通常，这用于调用局部对象的析构函数。如果有此标记，则在堆栈展开期间始终调用着陆垫。

2. `catch`: 这是一个类型-值对列表，表示可以处理的异常类型。如果在此列表中发现抛出的异常类型，则调用着陆垫。在 `foo()` 函数中，值是指向 `int` 类型的 C++ 运行时的指针，类似于 `_cxa_throw()` 函数的形参。
3. `filter`: 指定异常类型数组。如果在数组中找不到当前异常的异常类型，则调用着陆垫。这用于实现 `throw()` 的规范，对于 `foo()` 函数，数组只有一个成员——`int` 类型的类型信息。

着陆指令的结果类型是一个 `{i8*, i32}` 结构。第一个元素是指向抛出异常的指针，而第二个元素是类型选择器。让我们从结构中提取这两个元素：

```
%exc.ptr = extractvalue { i8*, i32 } %exc, 0
%exc.sel = extractvalue { i8*, i32 } %exc, 1
```

使用数字对类型进行选择，它可以帮助我们确定为什么要使用着陆垫。如果当前异常类型与着陆指令 `catch` 部分中给出的异常类型匹配，则它具有正值。如果当前异常类型不匹配过滤器中的任何值，则该值为负值，如果应该调用清理代码，则该值为 0。

基本上，类型选择器是类型信息表中偏移信息，该表由 `landingpad` 指令的 `catch` 和 `filter` 中给定的值构造。在优化过程中，多个着陆垫可以组合成一个，这意味着该表格的结构在 IR 层面上是未知的。要检索给定类型的类型选择器，需要调用 `@llvm.eh.typeid.for` 指令函数。我们需要检查类型选择器的值是否对应 `int` 的类型信息，以便执行 `catch (int e) {}` 块中的代码：

```
%tid.int = tail call i32 @llvm.eh.typeid.for(
                                i8* bitcast (i8** @_ZTIi to
                                i8*))
%tst.int = icmp eq i32 %exc.sel, %tid.int
br i1 % tst.int, label %catchint, label %filterorcleanup
```

异常的处理是通过调用 `_cxa_begin_catch()` 和 `_cxa_end_catch()` 来实现的。`_cxa_begin_catch()` 函数需要一个参数：当前异常。这是着陆指令返回的值之一。它返回一个指向异常的指针——我们的例子中是一个 `int` 值。函数的作用是：`_cxa_end_catch()` 标志着异常处理的结束，并释放由 `_cxa_allocate_exception()` 分配的内存。请注意，如果在 `catch` 块中抛出另一个异常，则运行时行为会复杂很多。异常处理如下：

```
catchint:
%payload = tail call i8* @_cxa_begin_catch(i8* %exc.ptr)
%payload.int = bitcast i8* %payload to i32*
%retval = load i32, i32* %payload.int
tail call void @_cxa_end_catch()
br label %return
```

如果当前异常类型与 `throws()` 声明中的列表不匹配，则调用“意外”的异常处理程序。首先，我们需要再次检查类型选择器，如下所示：

```
filterorcleanup:  
%tst.blzero = icmp slt i32 %exc.sel, 0  
br i1 %tst.blzero, label %filter, label %cleanup
```

如果类型选择器的值小于 0，则调用处理程序：

```
filter:  
tail call void @_cxa_call_unexpected(i8* %exc.ptr) #4  
unreachable
```

同样，处理程序不会进行返回。

在这种情况下不需要清理工作，所以清理代码所做的全部工作就是继续执行堆栈展开器 (stack unwinder)：

```
cleanup:  
resume i8*, i32 %exc
```

还有一点还不清楚：libunwind 驱动堆栈展开，但它并没有绑定到语言，语言依赖处理是在函数中完成的。对于 Linux 上的 C++，相应函数称为 `_gxx_personality_v0()`。根据平台或编译器的不同，这个名称可能有所不同。每一个需要参与堆栈展开的功能都附加了一个功能，相应函数分析函数是否捕获异常、是否有不匹配的筛选列表或是否需要清理调用。它将这些信息返回给展开器，展开器相应地进行操作。在 LLVM IR 中，函数的指针作为函数定义的一部分给出，如下代码片段所示：

```
define i32 @_Z3fooi(i32) personality i8* bitcast  
    (i32 (...)* @_gxx_personality_v0 to  
    i8*)
```

这样，异常处理功能就完成了。

要在编程语言的编译器中使用异常处理，最简单的策略是利用现有的 C++ 运行时函数。这样做还有一个好处，即可以与 C++ 互操作。缺点是将一些 C++ 运行时绑定到您的语言的运行时中——最明显的是内存管理。如果想避免这种情况，需要创建自己的 `_cxa_` 函数，或等效函数。不过，还是可以使用 libunwind，因为提供了堆栈展开的机制。

1. 了解一下如何创建 IR。我们在第 3 章中创建了 calc 表达式编译器。现在，我们将扩展表达式编译器的代码生成器，以便在执行除 0 时处理异常。生成的 IR 将检查一个除数的除数是否为 0，如果为 true，则会引发异常。我们还将为该函数添加一个着陆垫，可以捕获异常，在控制台打印“Divide by zero!”，最后结束计算。这种简单的情况下，使用异常处理并不是那么必要，但它可以让我们集中精力于代码生成。我们将所有代码添加到 `CodeGenerator.cpp` 文件中。首先，添加所需的新字段和一些帮助器方法，需要存储 `_cxa_allocate_exception()` 和 `_cxa_throw()` 函数的 LLVM 声明，包括函数类型和函数本身，需要 `GlobalVariable` 实例来保存类型信息。我们还需要引用包含着陆台的基本块和只包含不可达指令的基本块：

```

1 GlobalVariable *TypeInfo = nullptr;
2 FunctionType *AllocEHFty = nullptr;
3 Function *AllocEHFn = nullptr;
4 FunctionType *ThrowEHFty = nullptr;
5 Function *ThrowEHFn = nullptr;
6 BasicBlock *LPadBB = nullptr;
7 BasicBlock *UnreachableBB = nullptr;

```

- 我们还添加了一个新的辅助函数来创建用于比较两个值的 IR。createICmpEq() 函数接受左值和右值作为参数进行比较，它创建一个比较指令，用于测试值的相等性，并创建一个分支指令，用于相等和不相等情况下的两个基本块。这两个基本块通过 TrueDest 和 FalseDest 参数中的引用返回。新基本块的标签可以在 TrueLabel 和 FalseLabel 参数中给出：

```

1 void createICmpEq(Value *Left, Value *Right,
2     BasicBlock *&TrueDest,
3     BasicBlock *&FalseDest,
4     const Twine &TrueLabel = "",
5     const Twine &FalseLabel = "") {
6     Function *Fn =
7         Builder.GetInsertBlock()->getParent();
8     TrueDest = BasicBlock::Create(M->getContext(),
9         TrueLabel, Fn);
10    FalseDest = BasicBlock::Create(M->getContext(),
11        FalseLabel, Fn);
12    Value *Cmp = Builder.CreateCmp(CmpInst::ICMP_EQ,
13        Left, Right);
14    Builder.CreateCondBr(Cmp, TrueDest, FalseDest);
15 }

```

- 要使用运行时中的函数，我们需要创建几个函数声明。在 LLVM 中，必须构造给出签名的函数类型（以及函数本身）。我们使用 createFunc() 方法来创建这两个对象。函数需要引用 FunctionType 和 Function 指针、新声明函数的名称和结果类型。参数类型列表是可选的，表示变量参数列表的标志设置为 false，表示参数列表中没有变量部分：

```

1 void createFunc(FunctionType *&Fty, Function *&Fn,
2     const Twine &N, Type *Result,
3     ArrayRef<Type *> Params = None,
4     bool IsVarArgs = false) {
5     Fty = FunctionType::get(Result, Params, IsVarArgs);
6     Fn = Function::Create(
7         Fty, GlobalValue::ExternalLinkage, N, M);

```

```
8 }

```

这些准备工作完成后，我们继续生成 IR 来引发异常。

引发异常

为了生成引发异常的 IR 代码，添加了一个 `addThrow()` 方法。这个新方法需要初始化新字段，然后生成 IR，通过 `_cxa_` 函数引发异常。所引发异常的有效负载是 `int` 类型的，可以设置为任意值。下面是我们需要编码的内容：

1. 新的 `addThrow()` 方法首先检查 `TypeInfo` 字段是否已初始化。如果不是，则创建一个 `i8*` 类型的全局外部常量和一个 `_ZTIi` 名称。它表示描述 C++ `int` 类型的 C++ 元数据：

```
1 void addThrow(int PayloadVal) {
2     if (!TypeInfo) {
3         TypeInfo = new GlobalVariable(
4             *M, Int8PtrTy,
5             /*isConstant=*/true,
6             GlobalValue::ExternalLinkage,
7             /*Initializer=*/nullptr, "_ZTIi");

```

2. 初始化继续使用 `createFunc()` 辅助方法为 `_cxa_allocate_exception()` 和 `_cxa_throw functions()` 创建 IR 声明：

```
1     createFunc(AllocEHFty, AllocEHFn,
2                 "__cxa_allocate_exception",
3                 Int8PtrTy,
4                 {Int64Ty});
5     createFunc(ThrowEHFty, ThrowEHFn, "__cxa_throw",
6                 VoidTy,
7                 {Int8PtrTy, Int8PtrTy, Int8PtrTy});

```

3. 使用异常处理的函数需要一个特殊函数，它有助于堆栈展开。我们添加 IR 代码来声明 C++ 库中的 `_gxx_personality_v0()` 函数，并将其设置为当前特殊函数。当前函数不存储为字段，但可以使用 `Builder` 实例来查询当前的基本块，它将函数存储为父字段：

```
1     FunctionType *PersFty;
2     Function *PersFn;
3     createFunc(PersFty, PersFn,
4                 "__gxx_personality_v0", Int32Ty, None,
5                 true);
6     Function *Fn =
7         Builder.GetInsertBlock()->getParent();
8     Fn->setPersonalityFn(PersFn);

```

4. 接下来，创建并填充着陆垫的基本块。首先，我们需要保存指向当前基本块的指针。然后，创建一个新的基本块，在构建器中设置它作为插入指令的基本块，并调用 `addLandingPad()` 方法。此方法生成用于处理异常的 IR 代码，将在下一节捕获异常中进行描述。下面的代码填充了着陆垫的基本块：

```
1 BasicBlock *SaveBB = Builder.GetInsertBlock();
2 LPadBB = BasicBlock::Create(M->getContext(),
3                               "lpad", Fn);
4 Builder.SetInsertPoint(LPadBB);
5 addLandingPad();
```

5. 初始化部分已经完成创建保存不可达指令的基本块。同样，我们创建了一个基本块，并将其设置为构建器上的插入点。然后，向它添加一个不可访问的指令。最后，将构建器的插入点设置回已保存的 `SaveBB` 实例，以便将以下 IR 添加到正确的基本块中：

```
1 UnreachableBB = BasicBlock::Create(
2     M->getContext(), "unreachable", Fn);
3 Builder.SetInsertPoint(UnreachableBB);
4 Builder.CreateUnreachable();
5 Builder.SetInsertPoint(SaveBB);
6 }
```

6. 要引发异常，需要通过调用 `_cxa_allocate_exception()` 函数来为异常和负载分配内存。我们的异常是 C++ 的 `int` 类型，其大小通常为 4 字节。我们为其创建一个常量 `unsigned` 值，并将其作为参数调用函数。函数类型和函数声明已经初始化，所以只需要创建一个调用指令：

```
1 Constant *PayloadSz =
2     ConstantInt::get(Int64Ty, 4, false);
3 CallInst *EH = Builder.CreateCall(
4     AllocEHFty, AllocEHFn, {PayloadSz});
```

7. 接下来，我们将 `PayloadVal` 值存储到分配的内存中。为此，需要通过调用 `ConstantInt::get()` 函数来创建一个 LLVM IR 常量。指向分配内存的指针是 `i8*` 类型的，但是要存储 `i32` 类型的值，需要创建一个 `bitcast` 指令来强制转换该类型：

```
1 Value *PayloadPtr =
2     Builder.CreateBitCast(EH, Int32PtrTy);
3 Builder.CreateStore(
4     ConstantInt::get(Int32Ty, PayloadVal, true),
5     PayloadPtr);
```

8. 最后，通过调用 `_cxa_` 抛出函数引发异常。因为这个函数实际上会在处理函数中引发异常，所以我们需要使用 `invoke` 指令而不是 `call` 指令。与 `call` 指令不同，`invoke` 指令结束一个基本块，因为它有两个后续基本块。这里，是 `UnreachableBB` 和 `LPadBB` 基本块。如果函数没有

引发异常，控制流则转移到 UnreachableBB 基本块。由于 _cxa_throw() 函数的设计，这种情况永远不会发生。控制流会传输到 LPadBB 基本块来处理异常。这样就完成了 addThrow() 方法的实现：

```
1  Builder.CreateInvoke(  
2      ThrowEHFty, ThrowEHFn, UnreachableBB, LPadBB,  
3      {EH, ConstantExpr::getBitCast(TypeInfo,  
4          Int8PtrTy),  
5          ConstantPointerNull::get(Int8PtrTy)});  
6  }
```

接下来，我们添加代码来生成处理异常的 IR。

捕获异常

为了生成 IR 代码来捕获异常，添加了一个 addLandingPad() 方法。生成的 IR 从异常中提取类型信息。如果匹配 C++ 的 int 类型，则通过输出 “Divide by zero!”，并从函数返回。如果类型不匹配，则只需执行一个 resume 指令，它将控制权转移回运行时。因为在调用层次结构中没有其他函数来处理此异常，所以运行时将终止应用程序。以下是生成 IR 来捕获异常需要的步骤：

1. 生成的 IR 中，需要调用 C++ 运行库中的 _cxa_begin_catch() 和 _cxa_end_catch() 函数。要打印错误消息，将使用 C 运行时库的 puts() 函数，而要从异常中获取类型信息，必须生成对 llvm.eh.typeid.for 指令的调用。所有的函数都需要 FunctionType 和 Function 实例，我们使用 createFunc() 方法进行创建：

```
1  void addLandingPad() {  
2      FunctionType *TypeIdFty; Function *TypeIdFn;  
3      createFunc(TypeIdFty, TypeIdFn,  
4          "llvm.eh.typeid.for", Int32Ty,  
5          {Int8PtrTy});  
6      FunctionType *BeginCatchFty; Function  
7          *BeginCatchFn;  
8      createFunc(BeginCatchFty, BeginCatchFn,  
9          "__cxa_begin_catch", Int8PtrTy,  
10         {Int8PtrTy});  
11     FunctionType *EndCatchFty; Function *EndCatchFn;  
12     createFunc(EndCatchFty, EndCatchFn,  
13         "__cxa_end_catch", VoidTy);  
14     FunctionType *PutsFty; Function *PutsFn;  
15     createFunc(PutsFty, PutsFn, "puts", Int32Ty,  
16         {Int8PtrTy});
```

2. 着陆指令是我们生成的第一条指令。结果类型是一个包含 i8* 和 i32 类型字段的结构。这个结构是通过调用 StructType::get() 函数生成的。我们处理一个 C++ 的 int 类型的异常，必

须将 this 作为子句添加到 landingpad 指令中。子句必须是 i8* 类型的常量，因此需要生成一个 bitcast 指令将 TypeInfo 值转换为这种类型。将指令返回的值存储在 Exc 变量中，以备以后使用：

```
1 LandingPadInst *Exc = Builder.CreateLandingPad(  
2     StructType::get(Int8PtrTy, Int32Ty), 1, "exc");  
3 Exc->addClause(ConstantExpr::getBitCast(TypeInfo,  
4     Int8PtrTy));
```

3. 接下来，从返回值中提取类型选择器。通过 llvm.eh.typeid.for 指令，我们获取 TypeInfo 字段的类型 ID，其表示为 C++ 的 int 类型。通过这个 IR，我们现在已经生成了两个需要比较的值，以确定是否可以处理异常：

```
1 Value *Sel = Builder.CreateExtractValue(Exc, {1},  
2     "exc.sel");  
3 CallInst *Id =  
4     Builder.CreateCall(TypeIdFty, TypeIdFn,  
5         {ConstantExpr::getBitCast(  
6             TypeInfo, Int8PtrTy)}));
```

4. 为了生成用于比较的 IR，调用 createICmpEq()。这个函数还生成两个基本块，存储在 TrueDest 和 FalseDest 变量中：

```
1 BasicBlock *TrueDest, *FalseDest;  
2 createICmpEq(Sel, Id, TrueDest, FalseDest,  
3     "match",  
4     "resume");
```

5. 如果这两个值不匹配，控制流将继续在 FalseDest 基本块上运行。这个基本块只包含一个 resume 指令，用于将控制权交还给 C++ 运行时：

```
1 Builder.SetInsertPoint(FalseDest);  
2 Builder.CreateResume(Exc);
```

6. 如果这两个值相等，控制流将继续在 TrueDest 基本块上运行。首先生成 IR 代码，从存储在 Exc 变量中的 landingpad 指令的返回值中提取指向异常的指针。然后，生成对 _cxa_begin_catch() 函数的调用，将异常的指针作为参数传递，这表明运行时开始处理异常：

```
1 Builder.SetInsertPoint(TrueDest);  
2 Value *Ptr =  
3     Builder.CreateExtractValue(Exc, {0},  
4     "exc.ptr");  
5 Builder.CreateCall(BeginCatchFty, BeginCatchFn,  
6     {Ptr});
```


7. 通过调用 puts() 函数来处理异常,将消息打印到控制台。为此,首先通过调用 CreateGlobalString Ptr() 函数生成一个指向该字符串的指针,然后在生成的调用 puts() 函数中将该指针作为参数传入:

```
1   Builder.CreateCall(EndCatchFty, EndCatchFn);
2   Builder.CreateRet(Int32Zero);
3 }
```

通过 addThrow() 和 addLandingPad() 函数,可以生成 IR 来触发异常并处理异常。我们仍然需要添加 IR 来检查除数是否为 0,这是下一节的主题。

将异常处理代码集成到应用程序中

除法的 IR 是在 visit(BinaryOp&) 中生成的。我们不只是生成一个 sdiv 指令,而是首先生成 IR 来将除数与 0 进行比较。如果除数为 0,则控制流继续在基本块中引发异常。否则,控制流将继续使用 sdiv 指令在基本块中运行。在 createICmpEq() 和 addThrow() 函数的帮助下,我们可以很容易地编写代码:

```
1  case BinaryOp::Div:
2      BasicBlock *TrueDest, *FalseDest;
3      createICmpEq(Right, Int32Zero, TrueDest,
4                  FalseDest, "divbyzero", "notzero");
5      Builder.SetInsertPoint(TrueDest);
6      addThrow(42); // Arbitrary payload value.
7      Builder.SetInsertPoint(FalseDest);
8      V = Builder.CreateSDiv(Left, Right);
9      break;
```

代码生成部分现在已经完成。为了构建应用程序,可以切换到 build 目录并运行 ninja,如下所示:

```
$ ninja
```

构建完成后,您可以检查生成的 IR——例如,使用 with a: 3/a 表达式,如下所示:

```
$ src/calc "with a: 3/a"
```

您将看到引发和捕获异常所需的 IR。

生成的 IR 现在依赖于 C++ 运行时。链接所需库的最简单方法是使用 clang++ 编译器,将 rtdcalc.c 文件重命名为 rtdcalc.cpp 表达式计算器的运行时函数,并在文件中的每个函数前面添加 extern "C"。然后可以使用 llc 工具将生成的 IR 转换为一个目标文件,并使用 clang++ 编译器创建一个可执行文件:

```
$ src/calc "with a: 3/a" | llc -filetype obj -o exp.o
$ clang++ -o exp exp.o ../rtcalc.cpp
```

然后，可以用不同的值运行应用：

```
$ ./exp
Enter a value for a: 1
The result is: 3

$ ./exp
Enter a value for a: 0
Divide by zero!
```

第二次运行中，输入为 0，这将引发一个异常。能够正常工作！

我们已经学习了如何引发和捕获异常，生成 IR 的代码可以用作其他编译器的蓝图。当然，所使用的类型信息和 catch 子句的数量取决于编译器的输入，不过我们需要生成的 IR 遵循本节中的模式。

添加元数据是向 LLVM 提供进一步信息的一种方法。下一节中，我们将添加类型元数据，并在某些情况下支持 LLVM 优化器。

基于类型的别名生成元数据的分析

两个指针可以指向同一个内存单元，它们互为别名。在 LLVM 模型中没有输入内存，这使得优化器很难决定两个指针是否为别名。如果编译器能够证明两个指针没有别名，那么就有进行更多优化的可能。下一节中，在实现这种方法之前，我们将更仔细地研究这个问题，并研究添加的元数据将如何发挥作用。

理解添加元数据的需求

为了演示这个问题，让我们看看下面的函数：

```
1 void doSomething(int *p, float *q) {
2     *p = 42;
3     *q = 3.1425;
4 }
```

优化器不能确定 p 和 q 指针是否指向相同的内存单元。优化过程中，这是一个重要的分析，称为别名分析。如果 p 和 q 指向同一个存储单元，那么它们就是别名。如果优化器能够证明两个指针从来没有别名，这将提供优化机会，例如：在 soSomething() 函数中，可以在不改变结果的情况下重新排序存储。

一种类型的变量是否可以另一种类型的变量的别名，这取决于语言的定义。请注意，语言也可能包含打破基于类型的别名假设的表达式——例如，不相关类型之间的类型转换。

LLVM 开发人员选择的解决方案是添加元数据来加载和存储指令。元数据有两个用途：

- 首先，根据可以别名其他类型的类型层次结构定义类型层次结构
- 其次，描述了加载或存储指令中的内存访问

让我们看看 C 中的类型层次结构。每种类型的层次结构都以一个根节点开始，根节点可以命名，也可以匿名。LLVM 假设具有相同名称的根节点描述相同类型的层次结构。您可以在同一个 LLVM 模块中使用不同的类型层次结构，并且 LLVM 做了一个安全的假设，即这些类型可以别名。在根节点之下，有用于标量类型的节点。聚合类型的节点不附加到根节点，但它们引用标量类型和其他聚合类型。Clang 为 C 定义了如下层次结构：

- 根节点称为 Simple C/C++ TBAA
- 根节点下面是用于字符类型的节点。这是 C 语言中的一种特殊类型，因为所有指针都可以转换为 char 类型的指针。
- 在 char 节点下面是用于其他标量类型的节点和用于所有指针的类型。

聚合类型定义为成员类型和偏移量的序列。

这些元数据定义用于加载和存储指令的访问标记。访问标记由三部分组成：基类型、访问类型和偏移量。根据基类型的不同，访问标签描述内存访问有两种可能的方式：

1. 如果基类型是聚合类型，则访问标记描述结构成员的内存访问，具有访问类型并位于给定偏移量。
2. 如果基类型是标量类型，则访问类型必须与基类型相同，并且偏移量必须为 0。

有了这些定义，我们现在可以在访问标记上定义一个关系，该关系用于计算两个指针是否可以别名。元组 (基类型, 偏移量) 的直接父类由基类型和偏移量决定：

- 如果基类型是标量类型并且偏移量为 0，则直接父类型是 (父类型, 0)，父类型是类型层次结构中定义的父亲节点的类型。如果偏移量不为 0，则直接父节点未定义。
- 基类型是一个聚合类型，然后 tuple(基类型, 偏移量) 的直接父类型是 tuple(新类型, 新偏移量)，而新类型是偏移量处成员的类型。新偏移量会根据新开始调整的新类型的起始位置。

这个关系的传递闭包是父关系。双内存访问类型——例如，(基类型 1, 访问类型 1, 偏移量 1) 和 (基类型 2, 访问类型 2, 偏移量 2)——可以别名为 (基类型 1, 偏移量 1) 和 (基类型 2, 偏移量 2)，反之亦然。这其实与父关系相关。

用一个例子来说明：

```
1 struct Point { float x, y; }
2 void func(struct Point *p, float *x, int *i, char *c) {
3     p->x = 0; p->y = 0; *x = 0.0; *i = 0; *c = 0;
4 }
```

使用前面对标量类型的内存访问标记定义，参数 i 的访问标记是 (int, int, 0)，参数 c 的访问标记是 (char, char, 0)。在类型层次结构中，int 类型的节点的父亲节点是 char 节点，因此 (int, 0) 的直接父节点是 (char, 0)，两个指针都可以别名。参数 x 和参数 c 也是一样的，但是参数 x 和 i 不相关，因此它们不会互为别名。struct Point 的 y 成员的访问是 (Point, float, 4)，4 是结构中 y 成员的偏移量。(Point, 4) 的直接父对象是 (float, 0)，因此对 p->y 和 x 的访问可能会别名，而且出于同样的原因，参数 c 也是如此。

要创建元数据，我们使用 `llvm::MDBuilder` 类，在 `llvm/IR/MDBuilder.h` 头文件中声明。数据本身存储在 `llvm::MDNode` 和 `llvm::MDString` 类的实例中。使用构建器类可以让我们避开构造的内部细节。

通过调用 `createTBAARoot()` 方法创建根节点，该方法将类型层次结构的名称作为参数并返回根节点。可以使用 `createAnonymousTBAARoot()` 方法可以创建一个匿名的根节点。

使用 `createTBAAScalarTypeNode()` 方法将标量类型添加到层次结构中，该方法以类型的名称和父节点作为参数。为聚合类型添加类型节点稍微复杂一些，`createTBAAStructTypeNode()` 方法以类型名称和字段列表作为参数。指定为 `std::pair<llvm::mdnode*, uint64_t>` instance。第一个元素表示成员的类型，第二个元素表示结构类型中的偏移量。

使用 `createTBAAStructTagNode()` 方法创建访问标记，该方法接受基类型、访问类型和偏移量作为参数。

最后，元数据必须附加到加载或存储指令。指令类有一个 `setMetadata()` 方法，用于添加各种元数据。第一个参数必须是 `llvm::LLVMContext::MD_tbaa`，第二个参数必须是访问标记。

有了这些知识，我们将在下一节中为 `tinylang` 添加基于类型的别名分析 (TBAA) 的元数据。

向 `tinylang` 添加 TBAA 元数据

为了支持 TBAA，需要添加了一个新的 `CGTBAA` 类，该类负责生成元数据节点。我们让它成为 `CGModule` 类的成员，称它为 TBAA。每个加载和存储指令都可能注释，为此我们在 `CGModule` 类中也放置了一个新函数。该函数试图创建标记访问信息。如果成功，则将元数据附加到指令。这种设计还允许我们在不需要元数据时关闭元数据生成——例如，在关闭了优化的构建中：

```
1 void CGModule::decorateInst(llvm::Instruction *Inst,
2                             TypeDenoter *TyDe) {
3     if (auto *N = TBAA.getAccessTagInfo(TyDe))
4         Inst->setMetadata(llvm::LLVMContext::MD_tbaa, N);
5 }
```

我们将新 `CGTBAA` 类的声明放入 `include/tinylang/CodeGen/CGTBAA.h` 头文件中，并将定义放入 `lib/CodeGen/CGTBAA.cpp` 文件中。除了抽象语法树 (AST) 定义之外，头文件还需要包含定义元数据节点和构建器的文件：

```
1 #include "tinylang/AST/AST.h"
2 #include "llvm/IR/MDBuilder.h"
3 #include "llvm/IR/Metadata.h"
```

`CGTBAA` 类需要存储一些数据成员，看看如何一步步地做到这些：

1. 首先，需要缓存类型层次结构的根：

```
1 class CGTBAA {
2     llvm::MDNode *Root;
```

2. 为了构造元数据节点，需要 `MDBuilder` 类的实例：

```
1  llvm::MDBuilder MDHelper;
```

3. 最后，存储为类型生成的元数据以供重用:

```
1  llvm::DenseMap<TypeDenoter *, llvm::MDNode *>
2      MetadataCache;
3  // ...
4  };
```

在定义了构造所需的变量之后，现在添加创建元数据所需的方法:

1. 构造函数初始化数据成员:

```
1  CGTBAA::CGTBAA(llvm::LLVMContext &Ctx)
2      : MDHelper(llvm::MDBuilder(Ctx)), Root(nullptr) {}
```

2. 惰性地实例化类型层次结构的根，将其命名为 Simple tinylang TBAA:

```
1  llvm::MDNode *CGTBAA::getRoot() {
2      if (!Root)
3          Root = MDHelper.createTBAARoot("Simple tinylang
4                                          TBAA");
5      return Root;
6  }
```

3. 对于标量类型，在 MDBuilder 类的帮助下根据类型的名称创建元数据节点。新的元数据节点存储在缓存中:

```
1  llvm::MDNode *
2  CGTBAA::createScalarTypeNode(TypeDeclaration *Ty,
3                              StringRef Name,
4                               llvm::MDNode *Parent) {
5      llvm::MDNode *N =
6          MDHelper.createTBAAScalarTypeNode(Name, Parent);
7      return MetadataCache[Ty] = N;
8  }
```

4. 为记录创建元数据的方法更为复杂，必须枚举记录的所有字段:

```
1  llvm::MDNode *CGTBAA::createStructTypeNode(
2      TypeDeclaration *Ty, StringRef Name,
3      llvm::ArrayRef<std::pair<llvm::MDNode *,
4                              uint64_t>>
5      Fields) {
6      llvm::MDNode *N =
```

```

7     MDHelper.createTBAAStructTypeNode(Name, Fields);
8     return MetadataCache[Ty] = N;
9 }

```

5. 要返回 tinylang 类型的元数据，需要创建类型层次结构。由于 tinylang 的类型系统非常有限，可以使用一种简单的方法。每个标量类型映射到附加到根节点的唯一类型，并将所有指针映射到单个类型。结构化类型然后引用这些节点。如果不能映射类型，则返回 nullptr:

```

1 llvm::MDNode *CGTBAA::getTypeInfo(TypeDeclaration *Ty) {
2     if (llvm::MDNode *N = MetadataCache[Ty])
3         return N;
4
5     if (auto *Pervasive =
6         llvm::dyn_cast<PervasiveTypeDeclaration>(Ty)) {
7        StringRef Name = Pervasive->getName();
8         return createScalarTypeNode(Pervasive, Name,
9             getRoot());
10    }
11    if (auto *Pointer =
12        llvm::dyn_cast<PointerTypeDeclaration>(Ty)) {
13        StringRef Name = "any pointer";
14        return createScalarTypeNode(Pointer, Name,
15            getRoot());
16    }
17    if (auto *Record =
18        llvm::dyn_cast<RecordTypeDeclaration>(Ty)) {
19        llvm::SmallVector<std::pair<llvm::MDNode *,
20            uint64_t>,
21            4>
22            Fields;
23        auto *Rec =
24            llvm::cast<llvm::StructType>(
25                CGM.convertType(Record));
26        const llvm::StructLayout *Layout =
27            CGM.getModule()->getDataLayout()
28                .getStructLayout(Rec);
29
30        unsigned Idx = 0;
31        for (const auto &F : Record->getFields()) {
32            uint64_t Offset = Layout->getElementOffset(Idx);
33            Fields.emplace_back(getTypeInfo(F.getType()),

```

```

34         Offset);
35         ++Idx;
36     }
37    StringRef Name = CGM.mangleName(Record);
38     return createStructTypeNode(Record, Name, Fields);
39 }
40 return nullptr;
41 }

```

6. 获取元数据的一般方法是 `getAccessTagInfo()`。因为只需要查找指针类型，所以我们检查它。否则，返回 `nullptr`：

```

1 llvm::MDNode *CGTBAA::getAccessTagInfo(TypeDenoter *TyDe)
2 {
3     if (auto *Pointer = llvm::dyn_cast<PointerType>(TyDe))
4     {
5         return getTypeInfo(Pointer->getTyDen());
6     }
7     return nullptr;
8 }

```

要启用 TBAA 元数据的生成，只需要将元数据附加到生成的加载和存储指令。例如，在 `CGProcedure::writeVariable()` 中，对全局变量进行存储，使用存储指令：

```

1 Builder.CreateStore(Val, CGM.getGlobal(D));

```

为了修饰指令，需要将前面的行替换为以下行：

```

1 auto *Inst = Builder.CreateStore(Val,
2                                 CGM.getGlobal(Decl));
3 CGM.decorateInst(Inst, V->getTypeDenoter());

```

有了这些更改，就完成了 TBAA 元数据的生成。

下一节中，我们将讨论一个非常类似的主题：对元数据的生成进行调试。

添加调试元数据

为了允许源代码级调试，我们必须添加调试信息。LLVM 中对调试信息的支持使用调试元数据来描述源语言的类型和其他静态信息，以及跟踪变量值的内在特性。LLVM 核心库在 Unix 系统上以 DWARF 格式生成调试信息，在 Windows 系统上以 PDB(Protein Data Bank) 格式生成调试信息。我们将在下一节中查看其总体结构。

了解调试元数据的结构

为了描述静态结构, LLVM 以类似于用于基于类型的分析的元数据的方式使用元数据。静态结构描述文件、编译单元、函数、词法块和使用的数据类型。

我们使用的类是 `llvm::DIBuilder`, 需要使用 `llvm/IR/DIBuilder` 中的文件来获得类声明。这个构建器类, 提供了易于使用的接口来创建可调试的元数据。元数据要么添加到 LLVM 对象 (如全局变量) 中, 要么在调试时使用调试指令。这里列出了构建器类可以创建的元数据:

- `llvm::DIFile`: 使用文件名和包含该文件的目录的绝对路径来描述一个文件, 可以使用 `createFile()` 方法来创建。一个文件可以包含主编译单元, 也可以包含导入的声明。
- `llvm::DICompileUnit`: 这用于描述当前的编译单元, 还需要指定源语言、特定于编译器的生成器字符串、是否启用了优化, 当然还有编译单元所在的 `DIFile`。您可以通过调用 `createCompileUnit()` 来创建。
- `llvm::DISubprogram`: 描述一个函数。其中重要的信息有: 作用域 (通常是嵌套函数的 `DICompileUnit` 或 `DISubprogram`)、函数名、修饰过的函数名和函数类型。可以通过调用 `createFunction()` 来创建。
- `llvm::DILexicalBlock`: 描述了如何在语言中对块作用域进行建模的词汇块。可以通过调用 `createLexicalBlock()` 来创建。

LLVM 对编译器翻译的语言没有任何假设。因此, 没有关于该语言的数据类型的信息。要支持源代码级调试, 还要在调试器中显示变量值, 就必须添加类型信息。这里列出了相对重要的几个:

- `createBasicType()` 函数返回一个指向 `llvm::DIBasicType` 类的指针, 创建元数据来描述基本类型, 如 `tinylang` 中的 `INTEGER` 或 C++ 中的 `int`。除了类型的名称之外, 所需参数是以位为单位大小和编码的。例如, 有符号类型还是无符号类型。
- 有几种方法可以构造复合数据类型的元数据, 用 `llvm::DIComposite` 类表示。可以使用 `createArrayType()`、`createStructType()`、`createUnionType()` 和 `createVectorType()` 函数实例化数组、结构、联合和向量数据类型的元数据。这些函数需要相应的参数—例如, 数组类型的基类型和下标, 或者结构类型的字段成员列表。
- 还有一些方法支持枚举、模板、类等。

函数列表展示了必须将源语言的每个细节添加到调试信息中。假设 `llvm::DIBuilder` 类的实例名为 `DBuilder`。进一步假设有一些 `tinylang` 源文件, 名为 `File.mod` 的文件在 `/home/llvmuser` 文件夹下。文件中第 5 行是一个 `Func():INTEGER` 函数, 在第 7 行包含一个 `VAR i:INTEGER` 局部声明。我们为它创建元数据, 从文件的信息开始。这里需要指定文件所在文件夹的文件名和绝对路径:

```
1 llvm::DIFile *DbgFile = DBuilder.createFile("File.mod",
2                                           "/home/llvmuser");
```

该文件是 `tinylang` 中的一个模块, 因此是 LLVM 的编译单元。这包含了很多信息:

```
1 bool IsOptimized = false;
2 llvm::StringRef CUFlags;
3 unsigned ObjCRunTimeVersion = 0;
4 llvm::StringRef SplitName;
5 llvm::DICompileUnit::DebugEmissionKind EmissionKind =
```



```

6 llvm::DICompileUnit::DebugEmissionKind::FullDebug;
7 llvm::DICompileUnit *DbgCU = DBuilder.createCompileUnit(
8     llvm::dwarf::DW_LANG_Modula2, DbgFile,  tinylang",
9     IsOptimized, CUFlags,  ObjCRunTimeVersion, SplitName,
10     EmissionKind);

```

调试器需要知道源语言。DWARF 标准定义了一个包含所有公共值的枚举，就是不能简单地添加新的源语言中。为此，必须通过 DWARF 代理创建一个请求。请注意，调试器和其他调试工具也需要对新语言的支持，所以仅仅向枚举添加一个新成员是不够的。

许多情况下，选择一种与源语言相近的语言就够了。tinylang 中，我们选择了 Modula-2，使用 DW_LANG_Modula2 来进行语言识别。编译单元存放在文件中，该文件由之前创建的 DbgFile 变量标识，调试信息可以携带关于生成器的信息。这可以是编译器的名称和版本信息。这里，只是传递一个 tinylang 字符串。如果不想添加此信息，可以简单地使用空字符串作为参数。

下一组信息包括 IsOptimized 标志，指示编译器是否打开了优化。通常，这个标志派生自 -O 选项，可以使用 CUFlags 参数将附加参数设置传递给调试器。这里不使用这个，我们传递一个空字符串。这里不使用 Objective-C，所以传递 0 作为 Objective-C 的运行时代版本。通常，调试信息嵌入到我们创建的对象文件中。如果想要将调试信息写入一个单独的文件，那么 SplitName 参数必须包含该文件的名称；否则，只需传递一个空字符串。最后，可以定义应该发出的调试信息的级别。默认设置是完整的调试信息，通过使用 FullDebug 枚举值来表示。如果只想知道行号信息，还可以选择 LineTablesOnly 值，或者选择 NoDebug 值来表示根本没有调试信息。对于后者，最好一开始就不创建调试信息。

我们的最小源代码只使用 INTEGER 数据类型，是一个有符号的 32 位值。为这种类型创建元数据很简单：

```

1 llvm::DIBasicType *DbgIntTy =
2     DBuilder.createBasicType("INTEGER", 32,
3     llvm::dwarf::DW_ATE_signed);

```

要为函数创建调试元数据，必须为签名创建类型，然后为函数本身创建元数据。这类似于为函数创建 IR。函数的签名是一个数组，其中参数的所有类型按源顺序排列，函数的返回类型作为索引 0 处的第一个元素。通常，这个数组是动态构造的。本例中，我们还可以静态地构造元数据。这对于内部函数非常有用——例如，初始化模块。通常，这些函数的参数是已知的，编译器作者可以对它们进行硬编码：

```

1 llvm::Metadata *DbgSigTy = {DbgIntTy};
2 llvm::DITypeRefArray DbgParamsTy =
3     DBuilder.getOrCreateTypeArray(DbgSigTy);
4 llvm::DISubroutineType *DbgFuncTy =
5     DBuilder.createSubroutineType(DbgParamsTy);

```

我们的函数有一个 INTEGER 返回类型，没有其他参数，所以 DbgSigTy 数组只包含指向该类型元数据的指针。该静态数组转换为类型数组，然后使用该类型数组创建函数的类型。

函数本身需要更多的数据:

```
1 unsigned LineNo = 5;
2 unsigned ScopeLine = 5;
3 llvm::DISubprogram *DbgFunc = DBuilder.createFunction(
4     DbgCU, "Func", "_t4File4Func", DbgFile, LineNo,
5     DbgFuncTy, ScopeLine,
6     llvm::DISubprogram::FlagPrivate,
7     llvm::DISubprogram::SPFlagLocalToUnit);
```

一个函数属于一个编译单元，例子中存储在 DbgCU 变量中。需要在源文件中指定函数的名称，即 Func，而修饰的名称存储在对象文件中。此信息有助于调试器稍后定位函数的机器码。根据 tinylang 的规则，修饰的名为_t4File4Func。我们还必须指定包含该函数的文件。

这听起来可能令人惊讶，但考虑一下 C 和 C++ 中的包含机制：一个函数可以存储在不同的文件中，然后在主编译单元中用 #include 包含该函数。这里的情况并非如此，我们使用的文件与编译单元使用的文件相同。接下来，传递函数的行号和函数类型。函数的行号可能不是函数词法作用域开始的行号。在这种情况下，可以指定一个不同的 ScopeLine。函数也有保护，在这里用 FlagPrivate 值指定它来指示一个私有函数。其他可能的值是 FlagPublic 和 FlagProtected，用于 public 和 protected 函数。

除了保护级别，这里还可以指定其他标志，例如：FlagVirtual 表示虚函数，而 FlagNoReturn 表示函数不返回给调用者。可以在 llvm/include/llvm/IR/DebugInfoFlags.def 中找到 llvm 需要包含文件的完整列表。最后，可以指定特定于函数的标志。最常用的是 SPFlagLocalToUnit 值，该值表明该函数是这个编译单元的本地函数。还经常使用的是 MainSubprogram 值，这表明该功能是应用程序的主要功能。还可以在前面提到的 LLVM 包括文件中找到所有可能的值。

到目前为止，只创建了引用静态数据的元数据。变量本质上是动态的，下一节中，我们将探讨如何将静态元数据附加到 IR 代码，以及对变量的访问。

跟踪变量

上一节中描述的类型元数据需要与源程序的变量相关联。对于全局变量，这非常简单。DIBuilder 类的 createGlobalVariableExpression() 函数创建了描述全局变量的元数据。这包括源文件中变量的名称、修饰的名称、源文件等等。LLVM IR 中的全局变量由 GlobalVariable 类的一个实例表示。这个类有一个 addDebugInfo() 方法，它将 createGlobalVariableExpression() 返回的元数据节点与全局变量关联起来。

对于局部变量，我们需要采用另一种方法。LLVM IR 不知道一个表示局部变量的类型，它只知道值。LLVM 社区开发的解决方案是对内在函数的调用插入到函数的 IR 代码中，内在函数是 LLVM 知道的函数，因此可以对其进行一些修改。大多数情况下，内在函数不会导致机器级别的例程调用。这里，函数调用是将元数据与值关联起来的工具。

调试元数据最重要的内部函数是 llvm.dbg.declare 和 llvm.dbg.value。调用前者来声明局部变量的地址，调用后者将变量设置为新值时。

以后的 LLVM 版本将用 `llvm.dbg.addr` 指令替换 `llvm.dbg.declare`

`llvm.dbg.declare` 内在函数做了一个非常大胆的假设: 内在函数调用中描述的变量的地址, 在函数的整个生命周期内都是有效的。这种假设使得在优化期间很难保留调试元数据, 因为实际存储地址可能会改变。为了解决这一问题, 设计了新的指令 `llvm.dbg.addr`。这个指令的参数与 `llvm.dbg.declare` 相同, 但是它的语义没有 `llvm.dbg.declare` 那么严格。它仍然描述一个局部变量的地址, 前端应该生成对它的调用。

优化期间, Pass 可以用 (可能是多个) 对 `llvm.dbg.value` 和/或 `llvm.dbg.addr` 的调用来替换这个原有的调用, 以保留调试信息。

当对 `llvm.dbg.addr` 的构建工作完成时, `llvm.dbg.declare` 指令将弃用, 之后将删除该指令。

它是如何工作的呢? LLVM IR 的表示和通过 `LLVM::DIBuilder` 类进行的编程创建略有不同, 因此将两者都考虑。继续上一节的例子, 我们使用 `alloca` 指令在 `Func` 函数中为 `i` 变量分配本地存储, 如下所示:

```
@i = alloca i32
```

之后, 我们对 `llvm.dbg.declare` 指令添加调用:

```
call void @llvm.dbg.declare(metadata i32* %i,  
                             metadata !1, metadata  
                             !DIExpression())
```

第一个参数是局部变量的地址。第二个参数是本地变量的元数据, 可以通过调用本地变量的 `createAutoVariable()` 或 `llvm::DIBuilder` 类的参数的 `createParameterVariable()` 创建的。第三个参数是地址表达式, 稍后将对此进行解释。

来实现 IR 的创建, 可以通过调用 `llvm::IRBuilder<>` 类的 `CreateAlloca()` 方法来为 `@i` 局部变量分配存储:

```
1 llvm::Type *IntTy = llvm::Type::getInt32Ty(LLVMCtx);  
2 llvm::Value *Val = Builder.CreateAlloca(IntTy, nullptr, "i");
```

`LLVMCtx` 变量是使用的上下文类, 而 `Builder` 是 `llvm::IRBuilder<>` 类的使用实例。

局部变量也需要用元数据来描述:

```
1 llvm::DILocalVariable *DbgLocalVar =  
2   DBuilder.createAutoVariable(DbgFunc, "i", DbgFile,  
3                               7, DbgIntTy);
```

使用上一节中的值, 我们指定变量是 `DbgFunc` 函数的一部分, 变量的名称为 `i`, 在第 7 行由 `DbgFile` 命名的文件中定义, 变量的类型为 `DbgIntTy`。

最后,我们使用 `llvm.dbg.declare` 指令将调试元数据与变量的地址关联起来。使用 `llvm::DIBuilder` 可以屏蔽添加调用的细节:

```
1 llvm::DILocation *DbgLoc =
2   llvm::DILocation::get(LLVMCtx, 7, 5,
3                           DbgFunc);
4 DBuilder.insertDeclare(Val, DbgLocalVar,
5                           DBuilder.createExpression(), DbgLoc,
6                           Val.getParent());
```

同样,必须为变量指定一个源位置。`llvm::DILocation` 的实例是一个容器,用于保存与作用域关联的位置的行和列。`insertDeclare()` 方法向 LLVM IR 添加了对固有函数的调用。作为参数,需要变量的地址(存储在 `Val` 中)和变量的调试元数据(存储在 `DbgValVar` 中)。我们还传递一个空的地址表达式和之前创建的调试位置。与普通指令一样,需要指定将调用插入到哪个基本块中。如果指定了一个基本块,那么调用将在末尾插入。或者,可以指定一条指令,调用插入到该指令之前。我们有一个指向 `alloca` 指令的指针,它是插入底层基本块的最后一个指令。所以,我们使用这个基本块,调用附加在 `alloca` 之后的指令。

如果局部变量的值发生了变化,则必须将对 `llvm.dbg.value` 的调用添加到 IR 中。可以使用 `llvm::DIBuilder` 的 `insertValue()` 来实现。这与 `llvm.dbg.addr` 的工作原理类似。不同之处在于,现在指定的不是变量的地址,而是新值。

在实现函数的 IR 生成时,我们使用了一种高级算法,主要使用值,避免为局部变量分配存储空间。对于添加调试信息,这只意味着我们使用 `llvm.dbg.value` 的次数要比在 clang 生成的 IR 中多得多。

如果变量没有专用的存储空间,但属于较大的聚合类型,该怎么办?可能出现这种情况的原因是使用嵌套函数。要实现对调用方堆栈框架的访问,需要收集结构中所有使用过的变量,并将指向该记录的指针传递给被调用的函数。在调用的函数内部,您可以引用调用方的变量,就像是函数的局部变量一样。不同的是,这些变量现在是整体的一部分。

在对 `llvm.dbg.declare` 的调用中,如果调试元数据描述了第一个参数所指向的整个内存,则使用空表达式。如果它只描述内存的一部分,则需要添加一个表达式,指示元数据应用于内存的哪一部分。在嵌套的情况下,需要计算到帧的偏移量。需要访问 `DataLayout` 实例,您可以从创建 IR 代码的 LLVM 模块获取该实例。如果 `llvm::Module` 实例名为 `Mod`,则包含嵌套框架结构的变量名为 `frame`,类型为 `llvm::StructType`,然后访问该框架的第三个成员。这样就可以得到成员的偏移量:

```
1 const llvm::DataLayout &DL = Mod->getDataLayout();
2 uint64_t Ofs = DL.getStructLayout(Frame)
3   ->getElementOffset(3);
```

表达式是由一系列操作创建的。要访问 `Frame` 的第三个成员,调试器需要向基指针添加偏移量。需要创建一个数组和这个信息:

```
1 llvm::SmallVector<int64_t, 2> AddrOps;
2 AddrOps.push_back(llvm::dwarf::DW_OP_plus_uconst);
3 AddrOps.push_back(Offset);
```

从这个数组中，可以创建一个表达式，然后传递给 `llvm.dbg.declare`，而不是空表达式：

```
1 llvm::DIExpression *Expr = DBuilder.createExpression(AddrOps);
```

不必受此偏移操作的限制。DWARF 知道许多不同的操作符，所以可以创建相当复杂的表达式。可以在 `llvm/include/llvm/BinaryFormat/Dwarf.def` 文件中找到需要包含的 LLVM 文件操作符的完整列表。

现在可以为变量创建调试信息。要使调试器能够跟踪源中的控制流，还需要提供行号信息，这是下一节的主题。

添加行号

调试器允许程序员逐行调试应用程序。为此，调试器需要知道哪个机器指令属于源代码中的哪一行，LLVM 允许在每个指令中添加一个源位置。上一节中，创建了 `llvm::DILocation` 类型的位置信息。调试位置包含的信息不仅仅是行、列和范围。如果需要，可以指定这一行内联到的范围。还可以指出此调试位置属于隐式代码，即前端生成的但不在源代码中的代码。

将它附加到指令之前，必须将调试位置包装在 `llvm::DebugLoc` 对象中。为此，只需将从 `llvm::DILocation` 类获得的位置信息传递给 `llvm::DebugLoc` 构造函数。通过这种包装，LLVM 可以跟踪位置信息。虽然源代码中的位置显然没有改变，但在优化期间可以删除为源代码级语句或表达式生成的机器码。封装有助于处理这些可能的更改。

添加行号信息主要可以归结为从 AST 检索行号信息并将其添加到生成的指令中。指令类有 `setDebugLoc()` 方法，将位置信息附加到指令上。

下一节中，我们将调试信息的生成添加到 `tinylang` 编译器中。

为 `tinylang` 添加调试支持

我们将调试元数据的生成封装在新的 `CGDebugInfo` 类中。把声明放到 `tinylang/CodeGen/CGDebugInfo.h` 头文件中，把定义放到 `tinylang/CodeGen/CGDebugInfo.cpp` 文件中。

`CGDebugInfo` 类有五个重要成员。我们需要引用模块 CGM 的代码生成器，因为我们需要将类型从 AST 表示转换为 LLVM 类型。当然，还需要 `llvm::DIBuilder` 类的一个实例，称为 `DBuilder`，如上一节所述。还需要一个指向编译单元实例的指针，我们将它存储在名为 `CU` 的成员中。

为了避免重复创建类型的调试元数据，我们还添加了一个映射来缓存该信息，该成员称为 `TypeCache`。最后，需要一种管理作用域信息的方法，为此我们基于 `llvm::SmallVector<>` 类创建一个名为 `ScopeStack` 的堆栈：

```
1 CGModule &CGM;  
2 llvm::DIBuilder DBuilder;  
3 llvm::DICompileUnit *CU;  
4 llvm::DenseMap<TypeDeclaration *, llvm::DIType *>  
5   TypeCache;  
6 llvm::SmallVector<llvm::DIScope *, 4> ScopeStack;
```

`CGDebugInfo` 类的以下方法都使用了这些成员：

1. 首先，需要创建编译单元，这是在构造函数中完成的。我们还在这里创建了一个包含 compile 单元的文件。稍后，可以通过 CU 成员引用该文件。构造函数的代码如下所示：

```
1 CGDebugInfo::CGDebugInfo(CGModule &CGM)
2   : CGM(CGM), DBuilder(*CGM.getModule()) {
3     llvm::SmallString<128> Path(
4       CGM.getASTCtx().getFilename());
5     llvm::sys::fs::make_absolute(Path);
6
7     llvm::DIFile *File = DBuilder.createFile(
8       llvm::sys::path::filename(Path),
9       llvm::sys::path::parent_path(Path));
10
11     bool IsOptimized = false;
12     unsigned ObjCRunTimeVersion = 0;
13     llvm::DICompileUnit::DebugEmissionKind EmissionKind =
14       llvm::DICompileUnit::DebugEmissionKind::FullDebug;
15     CU = DBuilder.createCompileUnit(
16       llvm::dwarf::DW_LANG_Modula2, File, "tinylang",
17       IsOptimized,StringRef(), ObjCRunTimeVersion,
18       StringRef(), EmissionKind);
19 }
```

2. 通常，我们需要提供行号。这可以从源管理器位置派生，大多数 AST 节点都可以使用该位置。源管理器可以将其转换为行号：

```
1 unsigned CGDebugInfo::getLineNumber(SMLoc Loc) {
2   return CGM.getASTCtx().getSourceMgr().FindLineNumber(
3     Loc);
4 }
```

3. 有关作用域的信息保存在堆栈中。我们需要一些方法来打开和关闭一个作用域，以及检索当前作用域。编译单元是全局作用域，我们自动添加它：

```
1 llvm::DIScope *CGDebugInfo::getScope() {
2   if (ScopeStack.empty())
3     openScope(CU->getFile());
4   return ScopeStack.back();
5 }
6
7 void CGDebugInfo::openScope(llvm::DIScope *Scope) {
8   ScopeStack.push_back(Scope);
9 }
```

```

10
11 void CGDebugInfo::closeScope() {
12     ScopeStack.pop_back();
13 }

```

- 我们为需要转换的类型的每个类别创建一个方法。getPervasiveType() 方法为基本类型创建调试元数据。注意以下代码片段中编码参数的使用，声明 INTEGER 类型为 signed 类型，BOOLEAN 类型为 Boolean 类型：

```

1  llvm::DType *
2  CGDebugInfo::getPervasiveType(TypeDeclaration *Ty) {
3      if (Ty->getName() == "INTEGER") {
4          return DBuilder.createBasicType(
5              Ty->getName(), 64, llvm::dwarf::DW_ATE_signed);
6      }
7      if (Ty->getName() == "BOOLEAN") {
8          return DBuilder.createBasicType(
9              Ty->getName(), 1,
10                 llvm::dwarf::DW_ATE_boolean);
11      }
12      llvm::report_fatal_error(
13          "Unsupported pervasive type");
14  }

```

- 如果简单地重命名了类型名，则将其映射到类型定义。这里，我们需要 (第一次) 使用范围和行号信息：

```

1  llvm::DType *
2  CGDebugInfo::getAliasType(AliasTypeDeclaration *Ty) {
3      return DBuilder.createTypedef(
4          getType(Ty->getType()), Ty->getName(),
5          CU->getFile(), getLineNumber(Ty->getLocation()),
6          getScope());
7  }

```

- 为数组创建调试信息需要关于大小和对齐方式的说明，从 DataLayout 类中检索这些数据。我们还需要指定数组的下标范围：

```

1  llvm::DType *
2  CGDebugInfo::getArrayType(ArrayTypeDeclaration *Ty) {
3      auto *ATy =
4          llvm::cast<llvm::ArrayType>(CGM.convertType(Ty));
5      const llvm::DataLayout &DL =

```

```

6     CGM.getModule()->getDataLayout();
7
8     uint64_t NumElements = Ty->getUpperIndex();
9     llvm::SmallVector<llvm::Metadata *, 4> Subscripts;
10    Subscripts.push_back(
11        DBuilder.getOrCreateSubrange(0, NumElements));
12    return DBuilder.createArrayType(
13        DL.getTypeSizeInBits(ATy) * 8,
14        DL.getABITypeAlignment(ATy),
15        getType(Ty->getType()),
16        DBuilder.getOrCreateArray(Subscripts));
17 }

```

7. 使用所有这些单一方法，创建一个中心方法来创建类型的元数据。该元数据还负责缓存数据。代码可以在以下代码片段中看到:

```

1  llvm::DIType *
2  CGDebugInfo::getType(TypeDeclaration *Ty) {
3      if (llvm::DIType *T = TypeCache[Ty])
4          return T;
5
6      if (llvm::isa<PervasiveTypeDeclaration>(Ty))
7          return TypeCache[Ty] = getPervasiveType(Ty);
8      else if (auto *AliasTy =
9          llvm::dyn_cast<AliasTypeDeclaration>(Ty))
10         return TypeCache[Ty] = getAliasType(AliasTy);
11     else if (auto *ArrayTy =
12         llvm::dyn_cast<ArrayTypeDeclaration>(Ty))
13         return TypeCache[Ty] = getArrayType(ArrayTy);
14     else if (auto *RecordTy =
15         llvm::dyn_cast<RecordTypeDeclaration>(
16             Ty))
17         return TypeCache[Ty] = getRecordType(RecordTy);
18     llvm::report_fatal_error("Unsupported type");
19     return nullptr;
20 }

```

8. 我们还需要添加一个方法来生成全局变量的元数据:

```

1  void CGDebugInfo::emitGlobalVariable(
2      VariableDeclaration *Decl,
3      llvm::GlobalVariable *V) {

```



```

4   llvm::DIGlobalVariableExpression *GV =
5       DBuilder.createGlobalVariableExpression(
6           getScope(), Decl->getName(), V->getName(),
7           CU->getFile(),
8           getLineNumber(Decl->getLocation()),
9           getType(Decl->getType()), false);
10  V->addDebugInfo(GV);
11 }

```

9. 要发出过程的调试信息，首先需要为过程类型创建元数据。为此，需要一个参数类型列表，返回类型是第一个条目。如果过程没有返回类型，则使用未指定的类型 `void`，就像 C 中那样。如果形参是引用，则需要添加引用类型；否则，将该类型添加到列表中：

```

1  llvm::DISubroutineType *
2  CGDebugInfo::getType(ProcedureDeclaration *P) {
3      llvm::SmallVector<llvm::Metadata *, 4> Types;
4      const llvm::DataLayout &DL =
5          CGM.getModule()->getDataLayout();
6      // Return type at index 0
7      if (P->getRetType())
8          Types.push_back(getType(P->getRetType()));
9      else
10         Types.push_back(
11             DBuilder.createUnspecifiedType("void"));
12     for (const auto *FP : P->getFormalParams()) {
13         llvm::DIType *PT = getType(FP->getType());
14         if (FP->isVar()) {
15             llvm::Type *PTy = CGM.convertType(FP->getType());
16             PT = DBuilder.createReferenceType(
17                 llvm::dwarf::DW_TAG_reference_type, PT,
18                 DL.getTypeSizeInBits(PTy) * 8,
19                 DL.getABITypeAlignment(PTy));
20         }
21         Types.push_back(PT);
22     }
23     return DBuilder.createSubroutineType(
24         DBuilder.getOrCreateTypeArray(Types));
25 }

```

10. 对于过程本身，可以使用上一步中创建的过程类型创建调试信息。过程还会打开一个新的作用域，因此我们将该过程压入作用域堆栈。我们还将 LLVM 函数对象与新的调试信息关联起

来:

```
1 void CGDebugInfo::emitProcedure(  
2     ProcedureDeclaration *Decl, llvm::Function *Fn) {  
3     llvm::DISubroutineType *SubT = getType(Decl);  
4     llvm::DISubprogram *Sub = DBuilder.createFunction(  
5         getScope(), Decl->getName(), Fn->getName(),  
6         CU->getFile(), getLineNumber(Decl->getLocation()),  
7         SubT, getLineNumber(Decl->getLocation()),  
8         llvm::DINode::FlagPrototyped,  
9         llvm::DISubprogram::SPFlagDefinition);  
10    openScope(Sub);  
11    Fn->setSubprogram(Sub);  
12 }
```

11. 当到达一个过程的末尾时, 必须通知构建器完成此过程调试信息的构建。我们还需要从作用域堆栈中删除该过程:

```
1 void CGDebugInfo::emitProcedureEnd(  
2     ProcedureDeclaration *Decl, llvm::Function *Fn) {  
3     if (Fn && Fn->getSubprogram())  
4         DBuilder.finalizeSubprogram(Fn->getSubprogram());  
5     closeScope();  
6 }
```

12. 最后, 当完成添加调试信息时, 我们需要将 finalize() 方法添加到构建器中。然后验证生成的调试信息。这是开发过程中的一个重要步骤, 因为它可以帮助您查找错误生成的元数据:

```
1 void CGDebugInfo::finalize() { DBuilder.finalize(); }
```

只有在用户请求时才应该生成调试信息。为此, 需要一个新的命令行选项。我们将把它添加到 CGModule 类的文件中:

```
1 static llvm::cl::opt<bool>  
2 Debug("g", llvm::cl::desc("Generate debug information"),  
3     llvm::cl::init(false));
```

CGModule 类持有 std::unique_ptr<cgdebuginfo> 类的实例。关于命令行选项的设置, 指针在构造函数中初始化:

```
1 if (Debug)  
2     DebugInfo.reset(new CGDebugInfo(*this));
```

在 getter 方法中, 这样返回指针:

```

1 CGDebugInfo *getDbgInfo() {
2     return DebugInfo.get();
3 }

```

生成调试元数据时，一个常见的模式是检索指针并检查它是否有效。例如，在创建了一个全局变量之后，会以这种方式添加调试信息：

```

1 VariableDeclaration *Var = ...;
2 llvm::GlobalVariable *V = ...;
3 if (CGDebugInfo *Dbg = getDbgInfo())
4     Dbg->emitGlobalVariable(Var, V);

```

为了添加行号信息，需要添加一个 `getDebugLoc()` 转换方法到 `CGDebugInfo` 类中，它将 AST 中的位置信息转换为调试元数据：

```

1 llvm::DebugLoc CGDebugInfo::getDebugLoc(SMLoc Loc) {
2     std::pair<unsigned, unsigned> LineAndCol =
3         CGM.getASTCtx().getSourceMgr().getLineAndColumn(Loc);
4     llvm::DILocation *DILoc = llvm::DILocation::get(
5         CGM.getLLVMCtx(), LineAndCol.first, LineAndCol.second,
6         getCU());
7     return llvm::DebugLoc(DILoc);
8 }

```

然后，可以调用 `CGModule` 类中的实用函数，将行号信息添加到指令中：

```

1 void CGModule::applyLocation(llvm::Instruction *Inst,
2                             llvm::SMLoc Loc) {
3     if (CGDebugInfo *Dbg = getDbgInfo())
4         Inst->setDebugLoc(Dbg->getDebugLoc(Loc));
5 }

```

通过这种方式，可以为您自己的编译器添加调试信息。

总结

本章中，您学习了如何在 LLVM 中抛出和捕获异常，以及需要生成哪些 IR 代码来利用该特性。为了增强 IR 的范围，了解了如何将各种元数据附加到指令中。基于类型的别名的元数据为 LLVM 优化器提供了额外的信息，并有助于进行某些优化，以生成更好的机器码。用户总是喜欢使用源代码级调试器，通过向 IR 代码添加调试信息，就能够支持这一重要特性。

优化 IR 代码是 LLVM 的核心任务。下一章中，我们将学习 Pass 管理器是如何工作的，以及如何影响 Pass 管理器所管理的优化管道。

第 8 章 优化 IR

LLVM 使用一系列 Pass 来优化中间表示 (IR)，通过对 IR 的单元 (函数或模块) 执行操作。操作可以是转换 (以定义的方式更改 IR)，也可以是分析 (收集依赖关系等信息)。一系列的 Pass 称为 **Pass 流水线**。Pass 管理器在编译器生成的 IR 上执行 Pass 管道。因此，了解 Pass 管理器做什么，以及如何构建 Pass 管道很重要。支持编程语言的语义可能需要开发新的 Pass，所以必须将这些 Pass 添加到管道中。

本章中，我们将包括以下内容：

- 介绍 LLVM Pass 管理器
- 使用新 Pass 管理器实现一个 Pass
- 使用旧 Pass 管理器中使用 Pass
- 向编译器添加优化流水线

本章结束时，将了解如何开发一个新的 Pass，以及如何将它添加到 Pass 流水线中。还将了解如何在自己的编译器中设置 Pass 管道。

相关代码

本章的代码文件可在<https://github.com/PacktPublishing/Learn-LLVM-12/tree/master/Chapter08>获取。

你可以在视频中找到代码<https://bit.ly/3nllhED>。

介绍 LLVM Pass 管理器

LLVM 核心库优化编译器创建的 IR，并将其转换为目标代码。这个巨大的任务可以分解成几个单独的步骤，称为 Pass。这些 Pass 需要按照正确的顺序执行，这是 Pass 管理器的目标。

但为什么不硬编码 Pass 的顺序呢？编译器的用户通常期望编译器提供不同级别的优化。开发人员在开发期间进行优化，喜欢更快的编译速度。最终的应用程序应该尽可能快地运行，编译器应该能够执行复杂的优化，并接受更长的编译时间。不同的优化级别意味着需要执行的优化通过的数量不同。而且，作为编译器作者，您可能希望提供自己的 Pass，可以充分展现您对源语言的了解。例如，您可能想用内联 IR 或 (如果可能的话) 用该函数的计算结果替换已知的库函数。对于 C 语言，这样的 Pass 是 LLVM 核心库的一部分，但是对于其他语言，您需要单独提供。在介绍自己的 Pass 时，您可能需要重新排序或添加一些 Pass。例如，如果知道 Pass 的操作使某些 IR 代码不可访问，那么您应该运行完自己的 Pass 之后删除 Pass。这里，Pass 管理器可以帮助您进行管理。

Pass 通常根据作用范围进行分类：

- Pass 函数接受单个函数作为输入，并仅对该函数执行。
- Pass 模块接受整个模块作为输入。这样的 Pass 在给定的模块上执行，并可用于该模块中的过程内操作。
- 调用 Pass 图以自底向上的顺序遍历调用图的函数。

除了 IR 代码之外，Pass 还可能消耗、生成或使一些分析结果无效。有很多不同的分析，例如：别名分析或支配树的构造。支配树帮助将不变代码移出循环，因此执行这种转换的 Pass 只能在创

建支配树之后运行。另一个 Pass 可能执行可能使现有支配树无效的转换。

在编译器内部，Pass 管理器有以下功能：

- 分析结果由各 Pass 共享。这需要您指定跟踪哪个 Pass 需要哪个分析，以及每个分析的状态。其目标是避免不必要的分析重新计算，并尽快释放分析结果所占用的内存。
- Pass 以流水线方式执行，例如：如果需要依次执行几个 Pass 函数，那么 Pass 管理器将在第一个函数上运行这些函数中的每个 Pass。然后，将在第二个函数上运行所有的 Pass 函数，以此类推。这里的基本思想是改进缓存行为，因为编译器只对有限的数据集（即一个 IR 函数）执行转换，然后转向下一个（有限的）数据集。

LLVM 中有两个 Pass 管理器：

- 旧（或遗留的）Pass 管理器
- 新 Pass 管理器

未来是属于新的 Pass 管理器的，但目前过渡尚未完成。许多关键的 Pass（例如目标代码生成）还没有迁移到新 Pass 管理器，因此理解这两个 Pass 管理器还挺重要的。

旧 Pass 管理器要求 Pass 从基类继承，例如：Pass 函数从 `llvm::FunctionPass` 类继承。相比之下，新 Pass 管理器基于概念的方法，只需要从特殊的 `llvm::PassInfo<>` 语言特性（mixin）类继承。旧 Pass 管理器没有明确表示 Pass 之间的依赖关系，而新 Pass 管理器中，依赖关系需要显式编码。新 Pass 管理器还具有处理分析的不同方法，并允许通过命令行上的文本表示规范优化流水。一些 LLVM 用户报告说，仅仅从旧 Pass 管理器切换到新 Pass 管理器，编译量就减少了 10%，这是使用新 Pass 管理器非常有力的理由。

首先，我们将为新的 Pass 管理器实现一个 Pass，并探索如何将它添加到优化流水中。稍后，我们还将看看如何使用旧 Pass 管理器。

使用新 Pass 管理器实现一个 Pass

Pass 可以在 LLVM IR 上执行任意的转换。为了说明添加新 Pass 的机制，我们的新 Pass 只计算 IR 指令和基本块的数量，我们将这个 Pass 命名为 `countir`。将 Pass 添加到 LLVM 源树或作为一个独立的 Pass 略有不同，因此我们将在以下部分中进行这两种操作。让我们从向 LLVM 源树添加一个新的 Pass 开始。

向 LLVM 源树添加一个 Pass

我们从将新的 Pass 添加到 LLVM 源开始。如果稍后想要在 LLVM 树中发布新的 Pass，这是一种正确的方法。

在 LLVM IR 上执行转换的 Pass 的源代码位于 `llvm-project/llvm/lib/Transforms` 文件夹，而头文件位于 `llvm-project/llvm/include/llvm/Transforms` 文件夹。因为有这么多的 Pass，他们已分类到对应类别的子文件夹。

对于我们的新 Pass，需要在两个位置都创建了一个名为 `CountIR` 的新文件夹。首先，实现 `CountIR.h` 头文件：

1. 像往常一样，需要确保文件可以多次包含。另外，我们需要包含 Pass 管理器定义：

```
1 #ifndef LLVM_TRANSFORMS_COUNTIR_COUNTIR_H
```

```

2 #define LLVM_TRANSFORMS_COUNTIR_COUNTIR_H
3
4 #include "llvm/IR/PassManager.h"

```

- 因为在 LLVM 源代码中，所以可以将新的 CountIR 类放入 LLVM 命名空间中。该类继承自 PassInfoMixin 模板。这个模板只添加了一些样板代码，比如 name() 方法。不过，它不用于确定 Pass 的类型：

```

1 namespace llvm {
2 class CountIRPass : public PassInfoMixin<CountIRPass> {

```

- 在运行时，任务将调用的 run() 方法。run() 方法的签名决定了 Pass 的类型。这里，第一个参数是函数类型的引用，所以这是一个 Pass 函数：

```

1 public:
2     PreservedAnalyses run(Function &F,
3     FunctionAnalysisManager &AM);

```

- 最后，我们需要闭合类、命名空间和头文件的宏：

```

1 };
2 } // namespace llvm
3 #endif

```

当然，新 Pass 的定义非常简单，只执行了一项简单的任务。

继续在 CountIR.cpp 文件中实现 Pass。如果在调试模式下编译，LLVM 会收集关于 Pass 的统计信息。对于我们的 Pass，会使用这个基础组件。

- 通过包含我们自己的头文件和所需的 LLVM 头文件来开始编写源代码：

```

1 #include "llvm/Transforms/CountIR/CountIR.h"
2 #include "llvm/ADT/Statistic.h"
3 #include "llvm/Support/Debug.h"

```

- 为了缩短源代码，我们告诉编译器我们使用的是 llvm 名称空间：

```

1 using namespace llvm;

```

- LLVM 的内置调试基础设施要求我们定义一个调试类型，它是一个字符串。这个字符串稍后会显示在打印的统计信息中：

```

1 #define DEBUG_TYPE "countir"

```

- 我们用 STATISTIC 宏定义两个计数器变量。第一个参数是计数器变量的名称，第二个参数是将在统计中打印的文本：

```

1 STATISTIC(NumOfInst, "Number of instructions.");
2 STATISTIC(NumOfBB, "Number of basic blocks.");

```

9. 在 `run()` 方法中，我们循环遍历函数的所有基本块，并递增相应的计数器，并对基本块的所有指令都做同样的操作。为了防止编译器对未使用的变量发出警告，我们插入了 `I` 变量做空操作。因为我们只计算 IR 而不更改 IR，所以我们告诉调用者使用了 Pass，并且保留了所有的分析：

```
1 PreservedAnalyses
2 CountIRPass::run(Function &F,
3 FunctionAnalysisManager &AM) {
4     for (BasicBlock &BB : F) {
5         ++NumOfBB;
6         for (Instruction &I : BB) {
7             (void)I;
8             ++NumOfInst;
9         }
10    }
11    return PreservedAnalyses::all();
12 }
```

目前为止，已经实现了新 Pass 的功能。稍后将对 out-of-tree Pass 重用此实现。对于 LLVM 树中的解决方案，必须更改 LLVM 中的几个文件来声明新的 Pass：

1. 首先，需要将 `CMakeLists.txt` 添加到源文件夹。这个文件包含一个新的 LLVM 库名称 LLVM CountIR 的构建说明。新库需要链接到 LLVM Support 组件，因为我们使用了调试和统计基础设施，还需要链接到 LLVM Core 组件，其中包含 LLVM IR 的定义：

```
add_llvm_component_library(LLVMCountIR
    CountIR.cpp
    LINK_COMPONENTS Core Support )
```

2. 为了使这个新的库成为构建的一部分，我们需要将这个文件夹添加到父文件夹的 `CMakeList.txt` 中，即 `llvm-project/llvm/lib/Transforms/CMakeList.txt` 文件。然后，添加以下行：

```
add_subdirectory(CountIR)
```

3. `PassBuilder` 类需要知道我们的新 Pass。为此，在 `llvm-project/llvm/lib/Passes/PassBuilder.cpp` 文件的 `include` 部分添加以下代码：

```
1 #include "llvm/Transforms/CountIR/CountIR.h"
```

4. 最后一步，需要更新 Pass 注册表，它位于 `llvm-project/llvm/lib/Passes/PassRegistry.def` 文件中。查找定义 Pass 函数的部分，例如：通过搜索 `function _PASS` 宏。本节中，需要添加以下行：

```
1 FUNCTION_PASS("countir", CountIRPass())
```


5. 我们现在已经做了所有必要的改变。按照第 1 章的构建说明，使用 CMake 重新编译 LLVM。为了测试新的 Pass，我们在演示中存储以下 IR 代码。L1 文件在构建文件夹中。代码有两个函数，三个指令和两个基本块：

```
define internal i32 @func() {
    ret i32 0
}

define dso_local i32 @main() {
    %1 = call i32 @func()
    ret i32 %1
}
```

6. 我们可以通过 opt 实用程序使用新 Pass。要运行新 Pass，我们要使用 `--passes="countir"` 选项。要获得统计输出，需要添加 `--stats` 选项。因为我们不需要生成的比特码，所以我们也指定了 `--disable-output` 选项：

```
$ bin/opt --disable-output --passes="countir" --stats
demo.ll
=====
-----
... Statistics Collected ...
=====
-----
2 countir - Number of basic blocks.
3 countir - Number of instructions.
```

7. 运行我们的新 Pass，其输出符合期望。我们已经成功扩展了 LLVM！

运行单个 Pass 有助于调试。使用 `--passes` 选项，不仅可以命名单个 Pass，还可以描述整个流水。例如，优化级别 2 的默认管道名为 `default<O2>`。可以使用 `--passes="module(countir),default<O2>"` 参数在默认管道之前运行 countir Pass，这样的流水描述中的 Pass 名称必须是相同类型的。默认的流水是一个模块 Pass，我们的 countir Pass 是一个 Pass 函数。要从两者创建模块管道。首先，必须创建一个包含 countir Pass 的 Pass 模块。通过模块 (countir) 可以完成，通过在逗号分隔的列表中指定函数，可以向模块 Pass 中添加更多的 Pass 函数。以同样的方式，可以组合 Pass 模块。想要研究效果的话，可以使用内联和 countir Pass，以不同的顺序运行它们，或者作为 Pass 模块，结果会是不同的统计输出。

如果您计划将 Pass 作为 LLVM 的一部分发布，那么向 LLVM 源树中添加一个新的 Pass 是有意义的。如果不打算这样做，或者想独立于 LLVM 分发 Pass，那么可以创建一个 Pass 插件。下一节中，我们将查看执行此操作的步骤。

添加一个新 Pass 作为插件

为了提供一个新的 Pass 作为插件，我们将创建一个使用 LLVM 的新项目：

1. 我们首先在源文件夹中创建一个名为 countirpass 的文件夹。该文件夹将具有以下结构和文件：

```
|-- CMakeLists.txt
|-- include
|   |-- CountIR.h
|-- lib
    |-- CMakeLists.txt
    |-- CountIR.cpp
```

2. 注意，我们重用了前一节中的功能，并进行了一些调整。CountIR.h 头文件现在位于不同的位置，因此我们更改了用作保护宏的名称。我们也不使用 llvm 名称空间，因为现在在 llvm 源之外。头文件如下所示：

```
1 #ifndef COUNTIR_H
2 #define COUNTIR_H
3
4 #include "llvm/IR/PassManager.h"
5
6 class CountIRPass
7 : public llvm::PassInfoMixin<CountIRPass> {
8     public:
9         llvm::PreservedAnalyses
10         run(llvm::Function &F,
11             llvm::FunctionAnalysisManager &AM);
12 };
13
14 #endif
```

3. 可以复制上一节中的 CountIR.cpp 实现文件。这里也需要一些小的变动。因为头文件已经改变了，需要用下面的代码替换 include 指令：

```
1 #include "CountIR.h"
```

4. 我们还需要在 Pass 构建器中注册新 Pass，当加载插件时就会发生这种情况。Pass 插件管理器调用特殊函数 llvmGetPassPluginInfo()，该函数会执行注册。对于这个实现，需要额外的包含两个文件：

```
1 #include "llvm/Passes/PassBuilder.h"
2 #include "llvm/Passes/PassPlugin.h"
```

用户使用 `--passes` 选项指定要在命令行上运行的 Pass。PassBuilder 类从字符串中提取 Pass 名称。为了创建一个名为 Pass 的实例，PassBuilder 类维护一个回调列表。其实，调用回调时使用的是 Pass 名称和 Pass 管理器。如果回调知道 Pass 名称，那么将此 Pass 的一个实例添加到 Pass 管理器中。对于 Pass，需要提供这样一个回调函数：

```
1 bool PipelineParsingCB(  
2  StringRef Name, FunctionPassManager &FPM,  
3   ArrayRef<PassBuilder::PipelineElement>) {  
4     if (Name == "countir") {  
5       FPM.addPass(CountIRPass());  
6       return true;  
7     }  
8     return false;  
9 }
```

5. 当然，需要将这个函数注册为 PassBuilder 实例。加载插件后，注册回调函数就是为了这个目的。我们的注册功能如下：

```
1 void RegisterCB(PassBuilder &PB) {  
2     PB.registerPipelineParsingCallback(PipelineParsingCB);  
3 }
```

6. 最后，每个插件都需要提供前面提到的 `llvmGetPassPluginInfo()` 函数。这个函数返回一个包含四个元素的结构：我们的插件使用的 LLVM 插件 API 版本、一个名称、插件的版本号和注册回调。插件 API 要求函数使用 `extern "C"`。这是为了避免 C++ 修饰命名的问题。功能非常简单：

```
1 extern "C" ::llvm::PassPluginLibraryInfo LLVM_ATTRIBUTE_  
2 WEAK  
3 llvmGetPassPluginInfo() {  
4     return {LLVM_PLUGIN_API_VERSION, "CountIR", "v0.1",  
5         RegisterCB};  
6 }
```

为每个回调函数实现一个单独的函数有助于理解发生了什么。如果插件提供了几个 Pass，那么可以扩展 RegisterCB 回调函数来注册所有 Pass。通常，可以找到一种紧凑的方法。下面的 `llvmGetPassPluginInfo()` 函数将前面的 `PipelineParsingCB()`、`RegisterCB()` 和 `llvmGetPassPluginInfo()` 组合成一个函数，并通过 Lambda 函数来实现：

```
1 extern "C" ::llvm::PassPluginLibraryInfo LLVM_ATTRIBUTE_  
2 WEAK  
3 llvmGetPassPluginInfo() {  
4     return {LLVM_PLUGIN_API_VERSION, "CountIR", "v0.1",  
5         RegisterCB};  
6 }
```

```

5     [](PassBuilder &PB) {
6         PB.registerPipelineParsingCallback(
7             [](StringRef Name, FunctionPassManager
8                 &FPM,
9                 ArrayRef<PassBuilder::PipelineElement>)
10            {
11                if (Name == "countir") {
12                    FPM.addPass(CountIRPass());
13                    return true;
14                }
15                return false;
16            });
17    });
18 }

```

7. 现在，只需要添加构建文件。这个 lib/CMakeLists.txt 文件只包含一个编译源文件的命令。特定于 llvm 的命令 `add_llvm_library()` 确保使用了与构建 llvm 时相同的编译器标志：

```
add_llvm_library(CountIR MODULE CountIR.cpp)
```

顶层 CMakeLists.txt 文件更复杂。

8. 通常，我们设置了所需的 CMake 版本和项目名称。另外，将 `LLVM_EXPORTED_SYMBOL_FILE` 变量设置为 `ON`。这是插件在 Windows 上工作的必要条件：

```

cmake_minimum_required(VERSION 3.4.3)
project(countirpass)

set(LLVM_EXPORTED_SYMBOL_FILE ON)

```

9. 接下来，安装 LLVM。我们还要找到的版本信息，并打印到控制台：

```

find_package(LLVM REQUIRED CONFIG)
message(STATUS "Found LLVM ${LLVM_PACKAGE_VERSION}")
message(STATUS "Using LLVMConfig.cmake in: ${LLVM_DIR}")

```

10. 现在，可以将 LLVM 中的 `cmake` 文件夹添加到搜索路径中。包括了特定于 llvm 的文件 `ChooseMSVCCRT` 和 `AddLLVM`，并提供了额外的命令：

```
list(APPEND CMAKE_MODULE_PATH ${LLVM_DIR})
include(ChooseMSVCCRT)
include(AddLLVM)
```

11. 编译器需要知道所需的定义和 LLVM 的路径:

```
include_directories("${LLVM_INCLUDE_DIR}")
add_definitions("${LLVM_DEFINITIONS}")
link_directories("${LLVM_LIBRARY_DIR}")
```

12. 最后, 我们添加自己的 include 和 source 文件夹:

```
include_directories(BEFORE include)
add_subdirectory(lib)
```

13. 实现了所有必需的文件之后, 现在可以在 countirpass 文件夹旁创建 build 文件夹了。首先, 切换到构建目录并创建构建文件:

```
$ cmake -G Ninja ../countirpass
```

14. 然后, 可以编译插件:

```
$ ninja
```

15. 使用插件与 opt 工具, 这是模块化的 LLVM 优化器和分析器。其中, opt 生成输入文件的优化版本。使用插件时, 需要指定一个参数来加载插件:

```
$ opt --load-pass-plugin=lib/CountIR.so
    --passes="countir" \
    --disable-output --stats demo.ll
```

输出与以前的版本相同。恭喜你, Pass 插件工作了!

到目前为止, 我们只为新 Pass 管理器创建了一个 Pass。在下一节中, 我们还将扩展旧 Pass 管理器中的 Pass。

使用旧 Pass 管理器适配 Pass

未来是属于新的 Pass 管理器, 专为旧 Pass 管理器开发一个新的 Pass 没有什么意义。然而, 在正在进行的转换阶段, 如果 Pass 可以与两个 Pass 管理器一起工作, 就很有用, 因为 LLVM 中的大多数 Pass 已经这样做了。

旧 Pass 管理器需要从某些基类派生的 Pass，例如：Pass 函数必须派生自 FunctionPass 基类。还有更多的差异，Pass 管理器运行的方法名为 runOnFunction()，还必须提供 Pass 的 ID。在这里遵循的策略是创建一个单独的类，可以在旧 Pass 管理器中使用，并以一种可以在两个 Pass 管理器中使用的功能的方式重构源码。

我们使用 Pass 插件作为基础。在 include/CountIR.h 头文件中，我们添加了一个新的类定义，如下所示：

1. 这个新类需要从 FunctionPass 类派生，所以包含了一个额外的头文件来获取类定义：

```
1 #include "llvm/Pass.h"
```

2. 我们将这个新类命名为 CountIRLegacyPass。类需要一个 ID 作为内部 LLVM 机制，用它初始化父类：

```
1 class CountIRLegacyPass : public llvm::FunctionPass {
2 public:
3     static char ID;
4     CountIRLegacyPass() : llvm::FunctionPass(ID) {}
```

3. 为了实现 Pass 功能，必须重写两个函数。每个 LLVM IR 函数都会调用 runOnFunction() 方法，并实现计数功能。getAnalysisUsage() 方法用来声明所有的分析结果都保存了：

```
1     bool runOnFunction(llvm::Function &F) override;
2     void getAnalysisUsage(llvm::AnalysisUsage &AU) const
3     override;
4 };
```

4. 对头文件的更改完成后，可以增强 lib/CountIR.cpp 文件中的实现。为了重用计数功能，我们将源代码移动到一个新函数中：

```
1 void runCounting(Function &F) {
2     for (BasicBlock &BB : F) {
3         ++NumOfBB;
4         for (Instruction &I : BB) {
5             (void)I;
6             ++NumOfInst;
7         }
8     }
9 }
```

5. 为了使用新函数，新 Pass 管理器的方法需要更新：

```
1 PreservedAnalyses
2 CountIRPass::run(Function &F, FunctionAnalysisManager
3 &AM) {
4     runCounting(F);
```

```

5     return PreservedAnalyses::all();
6 }

```

6. 以同样的方式，为旧 Pass 管理器实现方法。使用 false 返回值，表示 IR 没有改变:

```

1 bool CountIRLegacyPass::runOnFunction(Function &F) {
2     runCounting(F);
3     return false;
4 }

```

7. 为了保留现有的分析结果，必须以以下方式实现 getAnalysisUsage() 方法。这类似于新 Pass 管理器中的 PreservedAnalyses::all() 返回值。如果不实现这个方法，那么默认情况下所有的分析结果都会丢弃:

```

1 void CountIRLegacyPass::getAnalysisUsage(
2     AnalysisUsage &AU) const {
3     AU.setPreservesAll();
4 }

```

8. ID 字段可以用任意值初始化，因为 LLVM 使用该字段的地址。公共值是 0，所以可以直接使用:

```

1 char CountIRLegacyPass::ID = 0;

```

9. 现在只有 Pass 注册没有了。要注册新 Pass，需要提供 RegisterPass<> 模板的静态实例。第一个参数是调用新 Pass 的命令行选项的名称。第二个参数是 Pass 的名称，是在使用-help 选项时为用户提供信息使用。

```

1 static RegisterPass<CountIRLegacyPass>
2 X("countir", "CountIR Pass");

```

10. 这些更改足以让我们在旧 Pass 管理器和新 Pass 管理器下使用新的 Pass。要测试添加的内容，请回到构建文件夹并编译 Pass:

```
$ ninja
```

11. 为了在旧 Pass 管理器中加载插件，需要使用--load 选项。新 Pass 是可以通过--countir 选项调用:

```
$ opt --load-pass-plugin=lib/CountIR.so --countir --stats\
    --disable-output demo.ll
```

Tip

还请在前一节的命令行中检查，使用新 Pass 管理器调用我们的 Pass 仍然可以正常工作!

能够使用 llvm 提供的工具运行新 Pass 非常香，但最终，我们希望在我们的编译器中运行它。在下一节中，我们将探讨如何设置优化和如何自定义流水。

向编译器添加优化流水线

tinylang 编译器 (在前几章中开发的) 没有对创建的 IR 代码进行优化。在接下来的部分中，我们将向编译器添加一个优化流水来精确地执行此操作。

使用新 Pass 管理器创建一个优化流水

设置优化流水的核心是 PassBuilder 类。这个类知道所有已注册的 Pass，并可以根据文本描述构造 Pass 流水。我们使用这个类根据命令行上给出的描述创建 Pass 流水，或者根据请求的优化级别使用默认流水。我们还支持使用 Pass 插件，例如 countir Pass 插件。这样，就模拟了 opt 工具的部分功能，并为命令行选项使用类似的名称。

PassBuilder 类填充了 ModulePassManager 类的实例，ModulePassManager 类是保存已构造的 Pass 流水，并运行它的 Pass 管理器。代码生成 Pass 仍然使用旧 Pass 管理器，因此必须为此目的保留旧 Pass 管理器。

为了实现功能，我们将 tinylang 编译器中的 tools/driver/Driver.cpp 文件进行了扩展：

1. 我们使用新的类，因此从添加新的包含文件开始。llvm/Passes/PassBuilder.h 文件提供了 PassBuilder 类的定义。llvm/Passes/PassPlugin.h 文件是插件支持所必需的。最后，llvm/Analysis/TargetTransformInfo.h 文件提供了一个连接 IR 级转换和特定目标信息的 Pass：

```
1 #include "llvm/Passes/PassBuilder.h"
2 #include "llvm/Passes/PassPlugin.h"
3 #include "llvm/Analysis/TargetTransformInfo.h"
```

2. 为了使用新 Pass 管理器的某些特性，我们添加了三个命令行选项，它们的名称与 opt 工具相同。--passes 选项启用 Pass 管道的文本规范，而--load-pass-plugin 选项启用相应的 Pass 插件。如果给定了--debug-pass-manager 选项，Pass 管理器将输出关于已执行的 Pass 的信息：

```
1 static cl::opt<bool>
2     DebugPM("debug-pass-manager", cl::Hidden,
3             cl::desc("Print PM debugging
4                       information"));
5 static cl::opt<std::string> PassPipeline(
6     "passes",
7     cl::desc("A description of the pass pipeline"));
8 static cl::list<std::string> PassPlugins(
9     "load-pass-plugin",
10    cl::desc("Load passes from plugin library"));
```

3. 用户对优化程度的设定影响着对 Pass 流水的构建。PassBuilder 类支持六种不同的优化级别：一种是不进行优化，三种是优化速度，两种是减小大小。我们可以使用命令行选项捕获所有这些级别：

```

1 static cl::opt<signed char> OptLevel(
2     cl::desc("Setting the optimization level:"),
3     cl::ZeroOrMore,
4     cl::values(
5         clEnumValN(3, "0", "Equivalent to -O3"),
6         clEnumValN(0, "00", "Optimization level 0"),
7         clEnumValN(1, "01", "Optimization level 1"),
8         clEnumValN(2, "02", "Optimization level 2"),
9         clEnumValN(3, "03", "Optimization level 3"),
10        clEnumValN(-1, "0s",
11            "Like -O2 with extra
12            optimizations "
13            "for size"),
14        clEnumValN(
15            -2, "0z",
16            "Like -O0s but reduces code size further")),
17    cl::init(0));

```

4. LLVM 的插件机制支持静态插件注册表，它是在项目配置期间创建的。为了使用这个注册表，包含了 `llvm/Support/Extension.def` 数据库文件来创建函数的原型，并返回插件信息：

```

1 #define HANDLE_EXTENSION(Ext) \
2     llvm::PassPluginLibraryInfo get##Ext##PluginInfo();
3 #include "llvm/Support/Extension.def"

```

5. 我们将重新实现 `emit()` 函数。函数的顶部声明了必需的 `PassBuilder` 实例：

```

1 bool emit(StringRef Argv0, llvm::Module *M,
2           llvm::TargetMachine *TM,
3           StringRef InputFilename) {
4     PassBuilder PB(TM);

```

6. 为了实现对命令行中给定的 `Pass` 插件支持，我们循环遍历用户给出的插件库列表，并尝试加载插件。如果失败，会发出一个错误消息；否则，就注册 `Pass`：

```

1 for (auto &PluginFN : PassPlugins) {
2     auto PassPlugin = PassPlugin::Load(PluginFN);
3     if (!PassPlugin) {
4         WithColor::error(errs(), Argv0)
5             << "Failed to load passes from '"
6             << PluginFN
7             << "'. Request ignored.\n";

```



```

8         continue;
9     }
10    PassPlugin->registerPassBuilderCallbacks(PB);
11 }

```

7. 静态插件注册表中的信息以类似的方式用于在 PassBuilder 实例中注册这些插件:

```

1 #define HANDLE_EXTENSION(Ext) \
2     get##Ext##PluginInfo().RegisterPassBuilderCallbacks( \
3         PB);
4 #include "llvm/Support/Extension.def"

```

8. 我们需要为不同的分析管理器声明变量。唯一的参数是调试标志:

```

1 LoopAnalysisManager LAM(DebugPM);
2 FunctionAnalysisManager FAM(DebugPM);
3 CGSCCAnalysisManager CGAM(DebugPM);
4 ModuleAnalysisManager MAM(DebugPM);

```

9. 接下来,我们将调用 PassBuilder 实例上相应的注册方法来填充分析管理器。通过这个调用,分析管理器将填充默认的分析 Pass,并运行注册回调。还需要确保函数分析管理器使用默认的别名-分析管道,并且所有的分析管理器都互相知道:

```

1 FAM.registerPass(
2     [&] { return PB.buildDefaultAAPipeline(); });
3 PB.registerModuleAnalyses(MAM);
4 PB.registerCGSCCAnalyses(CGAM);
5 PB.registerFunctionAnalyses(FAM);
6 PB.registerLoopAnalyses(LAM);
7 PB.crossRegisterProxies(LAM, FAM, CGAM, MAM);

```

10. MPM 模块 Pass 管理器保存着我们构造的 Pass 流水。实例用 debug 标志初始化:

```

1 ModulePassManager MPM(DebugPM);

```

11. 我们实现了两种不同的方法来使用 Pass 流水填充模块 Pass 管理器。如果用户在命令行上提供了一个 Pass 流水,也就是说,使用了--passes 选项,那么就使用这个作为 Pass 流水:

```

1 if (!PassPipeline.empty()) {
2     if (auto Err = PB.parsePassPipeline(
3         MPM, PassPipeline)) {
4         WithColor::error(errs(), Argv0)
5         << toString(std::move(Err)) << "\n";
6         return false;
7     }

```

```
8     }
```

12. 否则，根据使用所选的优化级别来确定要构造的 Pass 管道。默认 Pass 管道的名称为 default，它会将优化级别作为参数：

```
1     else {
2        StringRef DefaultPass;
3         switch (OptLevel) {
4             case 0: DefaultPass = "default<00>"; break;
5             case 1: DefaultPass = "default<01>"; break;
6             case 2: DefaultPass = "default<02>"; break;
7             case 3: DefaultPass = "default<03>"; break;
8             case -1: DefaultPass = "default<0s>"; break;
9             case -2: DefaultPass = "default<0z>"; break;
10        }
11        if (auto Err = PB.parsePassPipeline(
12            MPM, DefaultPass)) {
13            WithColor::error(errs(), Argv0)
14                << toString(std::move(Err)) << "\n";
15            return false;
16        }
17    }
```

13. 现在已经建立了在 IR 代码上运行转换的 Pass 流水。我们需要一个打开的文件来写入结果。系统汇编器和 LLVM IR 输出都是基于文本的，所以应该为它们都设置 OF_text 标志：

```
1     std::error_code EC;
2     sys::fs::OpenFlags OpenFlags = sys::fs::OF_None;
3     CodeGenFileType FileType = codegen::getFileType();
4     if (FileType == CGFT_AssemblyFile)
5         OpenFlags |= sys::fs::OF_Text;
6     auto Out = std::make_unique<llvm::ToolOutputFile>(
7         outputFilename(InputFilename), EC, OpenFlags);
8     if (EC) {
9         WithColor::error(errs(), Argv0)
10             << EC.message() << '\n';
11         return false;
12    }
```

14. 对于代码生成，必须使用旧 Pass 管理器。我们只需声明 CodeGenPM 实例，并添加 Pass，使特定于目标的信息在 IR 转换级别可用：

```
1     legacy::PassManager CodeGenPM;
```

```

2   CodeGenPM.add(createTargetTransformInfoWrapperPass(
3       TM->getTargetIRAnalysis()));

```

15. 为了输出 LLVM IR，添加了一个 Pass，只是将 IR 打印到流中：

```

1   if (FileType == CGFT_AssemblyFile && EmitLLVM) {
2       CodeGenPM.add(createPrintModulePass(Out->os()));
3   }

```

16. 否则，我们让 TargetMachine 实例添加所需的代码生成 Pass，由作为参数传递的 FileType 值指导：

```

1   else {
2       if (TM->addPassesToEmitFile(CodeGenPM, Out->os(),
3           nullptr, FileType)) {
4           WithColor::error()
5               << "No support for file type\n";
6           return false;
7       }
8   }

```

17. 在所有这些准备之后，现在就可以执行 Pass 了。首先，在 IR 模块上运行优化流水。接下来，运行代码生成 Pass。当然，在所有这些工作之后，我们希望保留输出文件：

```

1   MPM.run(*M, MAM);
2   CodeGenPM.run(*M);
3   Out->keep();
4   return true;
5 }

```

18. 代码很多，但很简单。当然，还必须更新 tools/driver/CMakeLists.txt 构建文件中的依赖项。除了添加目标组件外，还添加了来自 LLVM 的所有转换和代码生成组件。这些名称与源代码所在的目录名称大致相似。组件名在配置过程中会转换为链接库的名称：

```

set(LLVM_LINK_COMPONENTS ${LLVM_TARGETS_TO_BUILD}
    AggressiveInstCombine Analysis AsmParser
    BitWriter CodeGen Core Coroutines IPO IRReader
    InstCombine Instrumentation MC ObjCARCOpts Remarks
    ScalarOpts Support Target TransformUtils Vectorize
    Passes)

```

19. 我们的编译器驱动支持插件，可以声明支持插件：

```
add_tinylang_tool(tinylang Driver.cpp SUPPORT_PLUGINS)
```

20. 和以前一样，必须链接我们自己的库：

```
target_link_libraries(tinylang PRIVATE tinylangBasic tinylangCodeGen  
tinylangLexer tinylangParser tinylangSema)
```

这些都需要添加到源代码和构建系统中的。

21. 要构建扩展的编译器，请切换到构建目录并输入以下内容：

```
$ ninja
```

对构建系统文件的更改会自动检测到，在编译和链接更改的源代码之前，运行 `cmake`。如果您需要重新运行配置步骤，请按照第 2 章的步骤，编译 `tinylang`。

既然已经将 `opt` 工具的选项作为蓝图使用，那么应该尝试运行 `tinylang` 来加载 Pass 插件并运行 Pass，就像在前几节中所做的那样。

使用当前的实现，我们既可以运行默认的 Pass 流水，也可以自己构造一个。后者是非常灵活的。在所有情况下，都是多余的。默认流水在类 C 语言中运行得非常好。缺少的是一种扩展 Pass 管道的方法。在下一节中，我们将解释如何实现它。

扩展 Pass 流水

在前一节中，可以通过用户提供的描述或预定义的名称，使用 `PassBuilder` 类创建了 Pass 流水。现在，我们来了解一下定制 Pass 流水的另一种方法：使用扩展点。

在构造 Pass 流水期间，Pass 构建器允许您添加由用户贡献的 Pass。这些位置称为扩展点。LLVM 有许多扩展点，如：

- 流起始扩展点允许您在管道的开头添加 Pass。
- 窥视孔扩展点允许您在指令组合符 Pass 的每个实例之后添加 Pass。

还有其他扩展点。要使用扩展点，需要注册一个回调。在构造 Pass 流水期间，回调将在定义的扩展点运行，并可以向给定的 Pass 管理器添加一个 Pass。

要为流水起始扩展点注册回调，可以调用 `PassBuilder` 类的 `registerPipelineStartEPCallback()` 方法。例如，将 `CountIRPass` Pass 添加到流水的开头，需要通过调用 `createModuleToFunctionPassAdaptor()` 模板函数来调整 Pass 作为模块 Pass 使用，然后将 Pass 添加到模块 Pass 管理器中：

```
1 PB.registerPipelineStartEPCallback(  
2     [](ModulePassManager &MPM) {  
3         MPM.addPass(  
4             createModuleToFunctionPassAdaptor(  
5                 CountIRPass()));  
6     });
```

可以在 Pass 流水创建之前的任何时刻，也就是调用 `parsePassPipeline()` 方法之前，在 Pass 流水设置代码中添加此代码。

对于上一节中所做的工作，一个非常自然的扩展是让用户在命令行上传递一个扩展点的 Pass 流水描述 (opt 工具也允许这样做)。先为流水起始扩展点这样做。首先，将以下代码添加到 `tools/driver/Driver.cpp` 文件中：

1. 为用户添加了一个新的命令行来指定管道描述。同样，从 opt 工具中获取选项名：

```
1 static cl::opt<std::string> PipelineStartEPPipeline(  
2     "passes-ep-pipeline-start",  
3     cl::desc("Pipeline start extension point));
```

2. 使用 Lambda 函数作为回调函数是最方便的方法。为了解析流水描述，调用 `PassBuilder` 实例的 `parsepasspipeline()` 方法。Pass 会添加到 PM Pass 管理器，并作为 Lambda 函数的参数给出。如果出现错误，则在不停止应用程序的情况下打印错误消息。可以在调用 `crossRegisterProxies()` 方法之后添加以下代码：

```
1 PB.registerPipelineStartEPCallback(  
2     [&PB, Argv0](ModulePassManager &PM) {  
3         if (auto Err = PB.parsePassPipeline(  
4             PM, PipelineStartEPPipeline)) {  
5             WithColor::error(errs(), Argv0)  
6                 << "Could not parse pipeline "  
7                 << PipelineStartEPPipeline.ArgStr  
8                 << " : "  
9                 << toString(std::move(Err)) << "\n";  
10        }  
11    });
```

Tip

要允许用户在每个扩展点添加 Pass，需要为每个扩展点添加前面的代码。

3. 现在是尝试不同 Pass 管理器的好时机。使用 `--debug-pass-manager` 选项，可以按照顺序执行哪些 pass。可以使用 `--print-before-all` 和 `--print-after-all` 选项在调用每个 Pass 之前或之后打印 IR。如果创建了自己的 Pass 流水，那么可以将打印 Pass 插入到感兴趣的点中。例如，尝试一下 `--passes="print,inline,print"` 选项。还可以使用 `print` Pass 来探索各种扩展点。

LLVM 12 的新打印选项

LLVM 12 支持 `-print-changed` 选项，与之前 Pass 的结果相比，该选项仅在 IR 代码发生更改时打印。大大减少的输出，使得跟踪 IR 转换更加容易。

`PassBuilder` 类有一个嵌套的 `OptimizationLevel` 类来表示六个不同的优化级别。而不是使

用 "default" 流水描述作为 `parsePassPipeline()` 方法的参数，我们也可以调用 `buildPerModuleDefaultPipeline()` 方法，为优化级别构建相应的优化管道——除了级别 O0。优化级别为 O0，表示不执行优化。因此，Pass 管理器中不会添加任何 Pass。如果我们仍然想运行某个 Pass，可以手动将它添加到 Pass 管理器中。这个级别运行的是一个简单的 Pass: `AlwaysInlinerPass`。它将一个标记有 `always_` 内联属性的函数内联到调用者中。将优化级别的命令行选项值转换为 `OptimizationLevel` 类的相应成员，实现如下：

```
1 PassBuilder::OptimizationLevel Olevel = ...;
2 if (Olevel == PassBuilder::OptimizationLevel::O0)
3     MPM.addPass(AlwaysInlinerPass());
4 else
5     MPM = PB.buildPerModuleDefaultPipeline(Olevel,
6     DebugPM);
```

当然，可以通过这种方式向 Pass 管理器添加多个 Pass。PassBuilder 类还在构造 Pass 流水期间使用 `addPass()` 方法。

LLVM 12 中的新功能——运行扩展点回调

因为 Pass 流水不是为优化级别 O0 填充的，所以不会调用注册的扩展点。如果使用扩展点来注册 Pass(它也应该在 O0 级别运行)，那么这就有问题了。在 LLVM 12 中，可以调用新的 `runRegisteredEPCallbacks()` 方法来运行注册的扩展点回调，导致 Pass 管理器只填充通过扩展点注册的 Pass。

通过向 `tinylang` 添加优化流水，您可以创建一个优化编译器，比如 Clang。LLVM 社区致力于改进每个版本的优化和优化流水。因此，很少使用默认流水。大多数情况下，添加新 Pass 是为了实现编程语言的某些语义。

总结

本章中，您学习了如何为 LLVM 创建一个新的 Pass。您使用一个 Pass 流水描述和一个扩展点运行 Pass。通过构造和执行类似于 Clang 的 Pass 流水，扩展了编译器，将 `tinylang` 变成了带优化的编译器。Pass 流水允许在扩展点添加 Pass，并且您了解了如何在这些点注册 Pass。这使您能够使用自己开发的 Pass，或对现有 Pass 进行扩展或优化流水。

下一章中，我们将探索 LLVM 如何从优化的 IR 生成机器指令。

3 LLVM 进阶

您将了解如何在 LLVM 中实现指令选择，并通过添加对新机器指令的支持来应用这些知识。LLVM 有一个即时 (JIT) 编译器，您将了解如何使用它，以及如何根据您的需要对它进行裁剪。您还将尝试各种工具和库，这些工具和库有助于识别应用程序中的错误。最后，将使用一个新的后端扩展 LLVM，这将使您掌握利用 LLVM 尚未支持的新体系结构所需的知识。

本节包括以下几章：

- 第 9 章，选择指令
- 第 10 章，JIT 编译
- 第 11 章，使用 LLVM 工具调试
- 第 12 章，自定义编译器后端

第 9 章 指令选择

到目前为止使用的 LLVM IR 仍然需要转换成机器指令，这称为**指令选择**，通常缩写为 **ISel**。指令选择是后端的重要组成部分，LLVM 有三种不同的指令选择方法：选择 DAG、快速指令选择和全局指令选择。

在本章中，您将学习以下内容：

- 了解 LLVM 目标后端结构，将介绍目标后端执行的任务，并检查要运行的机器。
- 使用**机器 IR(MIR)** 来测试和调试后端，这有助于您输出 MIR 后指定的通过和运行一个通过的 MIR 文件。
- 指令选择是如何工作的，在其中您将了解 LLVM 执行指令选择的不同方式。
- 支持新的机器指令，在其中添加一个新的机器指令，并使其用于指令选择。

在本章结束时，您将了解目标后端是如何构造的，以及指令选择是如何工作的。您还将获得将当前不支持的机器指令添加到汇编程序和指令选择的知识，以及如何测试您添加的指令。

相关代码

要查看图形可视化，必须安装 Graphviz 软件，可以从 Graphviz 官网下载该软件<https://graphviz.org/>。源代码可在 <http://gitlab.com/graphviz/graphviz/> 获取。

本章的代码文件可在<https://github.com/PacktPublishing/Learn-LLVM-12/tree/master/Chapter09> 获取。

你可以在视频中找到代码<https://bit.ly/3nllhED>。

理解 LLVM 目标后端结构

优化 LLVM IR 后，使用所选的 LLVM 目标生成机器码。其中，以下任务在后端执行：

1. 构造了用于指令选择的有向无环图 (DAG)，通常称为 SelectionDAG。
2. 选择与 IR 代码相对应的机器指令。
3. 选定的机器指令按最优顺序排列。
4. 机器寄存器取代虚拟寄存器。
5. 函数中添加头尾代码。
6. 按最优顺序排列基本块。
7. 运行目标特定的 Pass。
8. 使用对象源码或汇编进行触发。

所有这些步骤都是由 MachineFunctionPass 类派生的机器函数 Pass 来实现的。这是 FunctionPass 类的一个子类，该类是旧 Pass 管理器使用的基类之一。在 LLVM 12 中，机器功能 Pass 到新 Pass 管理器的转换仍在进行中。

在所有这些步骤中，LLVM 指令都要进行转换。在代码层，一个 LLVM IR 指令由 Instruction 类的实例表示。在指令选择阶段，它转换为 MachineInstr 实例。这是一个更接近实际机器表示。它已经包含了对目标有效的指令，但仍然在虚拟寄存器上操作（到寄存器分配），还可以包含某些伪

指令。指令选择之后的传递会对其进行改进，最后创建 MCInstr 实例，它是真实机器指令的表示。MCInstr 实例可以写入目标文件或打印为汇编代码。

为了探索后端 Pass，可以创建一个包含以下内容的小 IR 文件：

```
define i16 @sum(i16 %a, i16 %b) {  
    %res = add i16 %a, 3  
    ret i16 %res  
}
```

将此代码保存为 sum.ll，使用 llc(LLVM 静态编译器) 编译 MIPS 架构。该工具将 LLVM IR 编译为汇编文本或目标文件。可以在命令行中用 -mtriple 选项覆盖编译的目标平台。使用 -debug-pass=Structure 选项调用 llc 工具：

```
$ llc -mtriple=mips-linux-gnu -debug-pass=Structure < sum.ll
```

除了生成的程序集代码之外，您还将看到一长串要运行的机器 Pass 列表。其中，MIPS DAG->DAG 指令选择模式 Pass 执行指令选择，MIPS 延迟槽填充器是针对特定目标的 Pass，清理前的最后一个 Pass 是 MIPS 汇编打印机负责打印汇编代码。在所有这些 Pass 中，指令选择 Pass 是最有趣的，我们将在下一节详细讨论它。

使用 MIR 对后端进行测试和调试

在上一节中，您看到许多 Pass 在后端运行。然而，这些 Pass 中的大多数不是在 LLVM IR 上运行，而是在 MIR 上运行。这是依赖于目标的指令表示，因此比 LLVM IR 更低层。它仍可以包含对虚拟寄存器的引用，因此它还不是纯 CPU 指令。

例如，要查看 IR 级别的优化，可以告诉 llc 在每个 Pass 完成后转储 IR。因为它们不在 IR 上工作，所以并不适用于后台的机器 Pass。不过，MIR 也有类似的功能。

MIR 是当前模块中机器指令当前状态的文本表示。它利用了 YAML 格式，允许序列化和反序列化。基本思想是，可以在某个点停止传递 Pass，并以 YAML 格式检查状态。您还可以修改 YAML 文件，或者创建您自己的 YAML 文件，传递它，并检查结果。这样可以方便地调试和测试。

让我们看一下 MIR。使用 --stop-after=finalize-iseloption 和我们之前使用的测试输入文件运行 llc 工具：

```
$ llc -mtriple=mips-linux-gnu \  
    -stop-after=finalize-isel < sum.ll
```

这指示 llc 在指令选择完成后转储 MIR。缩短后的输出如下所示：

```

---
name:                                sum
body:                                |
    bb.0 (%ir-block.0):

    liveins: $a0, $a1
    %1:gpr32 = COPY $a1
    %0:gpr32 = COPY $a0
    %2:gpr32 = ADDu %0, %1
    $v0 = COPY %2
    RetRA implicit $v0
...

```

您可以注意几个属性。首先，虚拟寄存器 (如%0) 和真实机器寄存器 (如 \$a0) 的混合使用，原因在使用了更底层的 ABI。为了跨不同的编译器和语言移植，函数遵循调用约定，这是应用程序二进制接口 (ABI) 的一部分。该输出用于 MIPS 机器上的 Linux 系统，需要使用系统使用的调用规则，第一个参数在寄存器 \$a0 中传递。MIR 输出是在指令选择之后，但在寄存器分配之前生成的，所以仍然可以看到虚拟寄存器的使用。

MIR 文件中使用的是机器指令 ADDu，而不是 LLVM IR 中的 add 指令。您还可以看到虚拟寄存器附加了一个寄存器调用，本例中是 gpr32。MIPS 体系结构上没有 16 位寄存器，因此必须使用 32 位寄存器。

bb.0 标签为第一个基本块，标签后的缩进内容是基本块的一部分。第一个语句指定在进入基本块时处于活动状态的寄存器。在本例中，只有 \$a0 和 \$a1 这两个参数在输入时是活动的。

MIR 文件中还有很多其他细节可以在 LLVM MIR 文档中进行了解 <https://llvm.org/docs/MIRLangRef.html>。

遇到的第一个问题可能是如何找到一个 Pass 的名称，特别是当只需要检查该 Pass 之后的输出而不主动处理时。当使用带有 llc 的 -debug-pass=Structure 选项时，激活 Pass 的选项会打印在顶部。如果想在 Mips 延迟槽填充器 Pass 之前停止，那么您需要查看打印的列表，并找到 -mipsdelay-slot-filler 选项，其也代表相应 Pass 的名称。

MIR 文件格式的主要应用是在后端辅助测试机通过。使用 llc 和 --stop-after 选项，在指定 Pass 后得到 MIR。通常，使用它作为预期测试用例的基础。您要注意的第一件事是，MIR 输出非常冗长。例如，许多字段是空的。为了减少这种混乱，可以在 llc 命令行中添加 -simplify-mir 选项。

您可以根据测试用例的需要保存和更改 MIR。llc 工具可以运行一个 Pass，这是与 MIR 文件测试的完美匹配。我们假设您想要测试 MIPS 延迟槽填充器 Pass。延迟槽是 RISC 体系结构 (如 MIPS 或 SPARC) 的一个特殊属性: 跳转后的下一条指令总是是执行。因此，编译器必须确保在每次跳转后都有合适的指令，而这个 Pass 就是执行这个任务的。

在运行 Pass 之前生成 MIR:

```
$ llc -mtriple=mips-linux-gnu \  
      -stop-before=mips-delay-slot-filler -simplify-mir \  
      < sum.ll >delay.mir
```

输出要小很多，因为使用了-simplify-mir 选项。函数体如下所示：

```
body:          |  
  bb.0 (%ir-block.0):  
    liveins: $a0, $a1  
  
    renamable $v0 = ADDu killed renamable $a0,  
                      killed renamable $a1  
    PseudoReturn undef $ra, implicit $v0
```

最值得注意的是，将看到 ADDu 指令后面是用于返回的 apseudo 指令。
delay.ll 文件作为输入，现在运行延迟槽填充器 Pass:

```
$ llc -mtriple=mips-linux-gnu \  
      -run-pass=mips-delay-slot-filler -o - delay.mir
```

现在将输出的函数与之前的函数进行比较：

```
body:          |  
  bb.0 (%ir-block.0):  
    PseudoReturn undef $ra, implicit $v0 {  
      renamable $v0 = ADDu killed renamable $a0,  
                      killed renamable $a1
```

您可以看到，用于返回的 ADDu 和伪指令已经更改了顺序，ADDu 指令现在嵌套在返回语句中：传递标识了适合于延迟槽的 ADDu 指令。

如果您刚接触到延迟槽的概念，也会想要看看生成的组件，这可以通过 llc 很轻松的完成：

```
$ llc -mtriple=mips-linux-gnu < sum.ll
```

输出包含了很多细节，但是在 bb.0 基础块的帮助下，可以很容易地找到为它生成的汇编代码：

```
# %bb.0:  
      jr      $ra  
      jaddu   $2, $4, $5
```

的确，指令的顺序改变了！

有了这些知识，我们将了解后端的核心，并检查如何在 LLVM 中执行机器指令的选择。

如何选择指令

LLVM 后端的任务是从 LLVM IR 创建机器指令，这个过程叫做指令选择或指令降级。出于尽可能地自动化这个任务的想法，LLVM 开发人员发明了 TableGen 语言来捕获目标描述的所有细节。在深入学习指令选择算法之前，我们先来看看这种语言。

用 TableGen 语言指定目标

机器指令有很多属性：汇编程序和反汇编程序使用的助记符、在内存中表示指令的位模式、输入和输出操作数，等等。LLVM 开发人员决定在一个地方捕获所有这些信息，即目标描述。一种新的语言，TableGen 语言，就是为此目的而发明的。其想法是使用代码生成器从目标描述创建各种源片段，然后可以在不同的工具中使用这些片段。目标描述存储在使用 .td 文件中。

原则上，TableGen 语言非常简单，您所能做的就是定义记录。一个记录有唯一的名称，包含一个值列表和一个超类列表。定义是一个所有值的记录，而类是一个可以有未定义值的记录。类的主要目的是有一个抽象记录，可以用来构建其他抽象或具体的记录。例如，Register 类定义了一个寄存器的公共属性，则可以为寄存器 R0 定义一个具体的记录：

```
1 class Register {
2     string name;
3 }
4
5 def R0 : Register {
6     let name = "R0";
7     string altName = "$0";
8 }
```

使用 let 关键字覆盖一个值。

TableGen 语言有很多语法糖，可以使处理记录更容易。一个类可以有一个模板实参，例如：

```
1 class Register<string n> {
2     string name = n;
3 }
4
5 def R0 : Register<"R0"> {
6     string altName = "$0";
7 }
```

TableGen 语言是静态类型的，您必须指定每个值的类型。一些受支持的类型如下：

- bit: 一个位
- int: 64 位整数值
- bits<n>: 一种由 n 位组成的整数类型

- string: 一个字符串
- list<t>: 类型为 t 的元素列表
- dag: 有向无环图 (DAG: 由指令选择使用)

类的名称也可以用作类型。例如，list<register> 指定 Register 类的元素列表。

该语言允许使用 include 关键字包含其他文件。对于条件编译，支持预处理指令 #define、#ifdef 和 #ifndef。

LLVM 中的 TableGen 库可以解析用 TableGen 语言编写的文件，并创建记录的内存表示。您可以使用这个库创建自己的生成器。

LLVM 自带了自己的生成工具 LLVM-tblgen 和一些 .td 文件。后端目标描述包括 llvm/target/target.td 文件。该文件定义诸如 Register、Target 或 Processor 等类。llvm-tblgen 工具了解这些类，并从定义的记录中生成 C++ 代码。

以 MIPS 后端为例来看看。目标描述在 Mips.td 中，文件位于 llvm/lib/Target/Mips 文件夹，该文件包含 Target.td 文件。它还定义了目标特性，例如：

```
def FeatureMips64r2
  : SubtargetFeature<"mips64r2", "MipsArchVersion",
                    "Mips64r2", "Mips64r2 ISA Support",
                    [FeatureMips64, FeatureMips32r2]>;
```

这些特性可以用来定义 CPU 模型，例如：

```
def : Proc<"mips64r2", [FeatureMips64r2]>;
```

还包括定义寄存器、指令、调度模型等的其他文件。

llvm-tblgen 工具可以显示此目标描述定义的记录。如果在 build 目录下，下面的命令会将记录打印到控制台：

```
$ bin/llvm-tblgen \
-I../llvm-project/llvm/lib/Target/Mips/ \
-I../llvm-project/llvm/include \
../llvm-project/llvm/lib/Target/Mips/Mips.td
```

与 Clang 一样，-I 选项在包含文件时添加一个要搜索的目录。查看记录对调试有帮助。这个工具的真正目的是从记录中生成 C++ 代码。例如，使用 -gen-subtarget 选项，解析 llc 的 -mcpu= 和 -mtarget= 所需的数据将发送到控制台：

```
$ bin/llvm-tblgen \  
-I../llvm-project/llvm/lib/Target/Mips/ \  
-I../llvm-project/llvm/include \  
../llvm-project/llvm/lib/Target/Mips/Mips.td \  
-gen-subtarget
```

将该命令生成的代码保存在一个文件中，并探索如何在生成的代码中使用该特性和 CPU！

指令的编码通常遵循一些模式。因此，将指令的定义分为定义位编码的类和指令的具体定义类。MIPS 指令编码在 `llvm/Target/Mips/MipsInstrFormats.td` 文件中。让我们来看看 `ADD_FM` 格式的定义：

```
class ADD_FM<bits<6> op, bits<6> funct> : StdArch {  
    bits<5> rd;  
    bits<5> rs;  
    bits<5> rt;  
  
    bits<32> Inst;  
  
    let Inst31-26 = op;  
    let Inst25-21 = rs;  
    let Inst20-16 = rt;  
    let Inst15-11 = rd;  
    let Inst10-6 = 0;  
    let Inst5-0 = funct;  
}
```

在记录主体中，定义了几个新的位域：`rd`、`rs` 等。它们用于覆盖 `Inst` 字段的部分，该字段保存指令的位模式。`rd`、`rs` 和 `rt` 位域对指令操作的寄存器进行编码，而 `op` 和 `funct` 参数表示操作码和函数号。`StdArch` 超类只添加一个字段，声明该格式遵循的编码标准。

MIPS 目标中的大多数指令编码不指向 DAG 节点，也不指定汇编助记符。为此定义了一个单独的类。MIPS 体系结构中的一个指令是 `nor` 指令，它按位计算第一和第二输入寄存器的值，将结果的位倒转，并将结果赋给输出寄存器。该指令有几种变体，下面的 `LogicNOR` 类可以避免多次使用相同的定义：

```
class LogicNOR<string opstr, RegisterOperand RO>:
    InstSE<(outs RO:$rd), (ins RO:$rs, RO:$rt),
           !strconcat(opstr, "$rd, $rs, $rt"),
           [(set RO:$rd, (not (or RO:$rs, RO:$rt)))],
           II_NOR, FrmR, opstr> {
    let isCommutable = 1;
}
```

哇，记录这个简单的概念现在看起来很复杂。让我们解析一下这个定义：这个类派生自 `InstSE` 类，它用于具有标准编码的指令。如果进一步跟踪超类的层次结构，就会看到这个类派生自指令类，指令类是预定义的类，表示目标的一条指令。`(outs RO:$rd)` 参数将最后一条指令的结果定义为 DAG 节点。`RO` 部分是指与 `LogicNOR` 类同名的参数，表示寄存器操作数，`$rd` 是要使用的寄存器。在 `rd` 字段中，这是以后要放到指令编码中的值。第二个参数定义指令将操作的值。总之，这个类用于在三个寄存器上操作的指令。`strconcat(opstr, "\t$rd, $rs, $rt")` 参数汇编指令的文本表示。`strconcat` 操作符是 TableGen 预定义的函数，用于连接两个字符串。可以在 TableGen 开发者指南中查找所有预定义的操作符：<https://llvm.org/docs/TableGen/ProgRef.html>。

记录遵循模式定义，类似于 `nor` 指令的文本描述，并描述该指令的计算。模式的第一个元素是操作，后面是用逗号分隔的操作数列表。操作数引用 DAG 参数中的寄存器名称，并指定 LLVM IR 值类型。LLVM 有一组预定义的操作符，例如 `add` 和 `and`，可以在模式中使用。这些操作符属于 `SDNode` 类，也可以用作参数。您可以在 `llvm/Target/TargetSelectionDAG.td` 中查找预定义的操作符。

`II_NOR` 参数指定在调度模型中使用的路线类，而 `FrmR` 参数是定义来标识这种指令格式的值。最后，`opstr` 助记符传递给超类。这个类的主体非常简单：它只指定 `nor` 操作是可交换的，这意味着操作数的顺序可以交换。

最后，这个类用于定义一条指令的记录，例如：64 位模式下的 `nor` 指令：

```
def NOR64 : LogicNOR<"nor", GPR64Opnd>, ADD_FM<0, 0x27>,
               GPR_64;
```

这是最终的定义，可以通过 `def` 关键字识别。它使用 `LogicNOR` 类来定义 DAG 操作数和模式，并使用 `ADD_FM` 类来指定二进制指令编码。附加的 `GPR_64` 谓词可以确保该指令仅在 64 位寄存器上可用。

开发人员努力避免多次重复定义，一种常用的方法是使用 `multiclass` 类。一个 `multiclass` 类可以一次定义多个记录。

例如，MIPS CPU 的浮点单元可以使用单精度或双精度浮点值进行加法运算。这两条指令的定义非常相似，因此 `multiclass` 类定义为一次创建两条指令：

```
multiclass ADDS_M<...> {  
    def _D32 : ADDS_FT<...>, FGR_32;  
    def _D64 : ADDS_FT<...>, FGR_64;  
}
```

FT 类定义了指令格式，类似于 LogicNOR 类。FGR_32 和 FGR_64 谓词在编译时决定可以使用哪条指令。重要的是_D32 和_D64 记录的定义，这些是记录的模板。然后用 defm 关键字定义指令记录：

```
defm FADD : ADDS_M<...>;
```

这将同时定义来自多类的两条记录，并将名称 FADD_D32 和 FADD_D64 赋给它们。这是避免代码重复的一种非常强大的方法，它经常用于目标描述，但与其他 TableGen 特性相结合，可能导致定义模糊。

有了目标描述是如何组织的知识，就可以在下一节中探索指令选择。

指令选择与选择 DAG

LLVM 将 IR 转换为机器指令的标准方法是通过 DAG。使用与目标描述中提供的模式匹配的模式，并使用自定义代码，IR 指令将转换为机器指令。这种方法并不像听起来那么简单：IR 主要是独立于目标的，并且可以包含目标不支持的数据类型。例如，表示单个位的 i1 类型在大多数目标上不是有效类型。

selectionDAG 由 SDNode 类型的节点组成，在 llvm/CodeGen/SelectionDAGNodes.h 文件中定义。该节点表示的操作称为 OpCode，目标独立代码定义在 llvm/CodeGen/ISDOpcodes.h 文件中。除了操作之外，节点还存储操作数及其生成的值。

节点的值和操作数形成数据流依赖关系。控制流依赖由链边表示，链边具有特殊类型 MVT::Other。这使得保留具有副作用的指令的顺序成为可能，例如：load 指令。

使用选择 DAG 进行指令选择的步骤如下：

1. 构建 DAG。
2. 优化 DAG。
3. 检查 DAG 中类型的合法性。
4. 优化 DAG。
5. 检查 DAG 中操作的合法性。
6. 优化 DAG。
7. 选择指令。
8. 指令排序。

让我们检查一下每个步骤对选择 DAG 所做的更改。

如何进行指令选择

可以通过两种不同的方式看到指令选择的工作。如果将`-debug-only=isel` 选项传递给 `llc` 工具, 则每一步的结果都会以文本格式打印出来。如果需要调查为什么选择机器指令, 这会有很大的帮助。例如, 运行以下命令查看总和的输出.ll 文件来自理解 LLVM 目标后端结构一节:

```
$ llc -mtriple=mips-linux-gnu -debug-only=isel < sum.ll
```

这会打印出很多信息。在输出的顶部, 您可以看到为输入创建的初始 DAG 的描述:

```
Initial selection DAG: %bb.0 'sum:'
SelectionDAG has 12 nodes:
  t0: ch = EntryToken
      t2: i32,ch = CopyFromReg t0, Register:i32 %0
      t5: i16 = truncate t2
      t4: i32,ch = CopyFromReg t0, Register:i32 %1
      t6: i16 = truncate t4
      t7: i16 = add t5, t6
      t8: i32 = any_extend t7
  t10: ch,glue = CopyToReg t0, Register:i32 $v0, t8
  t11: ch = MipsISD::Ret t10, Register:i32 $v0, t10:1
```

与上一节的 MIR 输出类似, 可以在这里看到 `CopyFromReg` 指令, 它将 ABI 使用的寄存器的内容转移到虚拟节点。截断节点是必需的, 因为示例使用 16 位值, 但是 MIPS 体系结构只支持 32 位值。在 16 位虚拟寄存器上执行添加操作, 并将结果扩展并返回给调用者。上面提到的每个步骤都会打印这个部分。

LLVM 还可以在 Graphviz 软件的帮助下生成选择 DAG 的可视化。如果将`-view-dag-combine1-dags` 选项传递给 `llc` 工具, 则会打开一个窗口, 显示构建的 DAG。例如, 使用前面的小文件运行 `llc`:

```
$ llc -mtriple=mips-linux-gnu -view-dag-combine1-dags sum.ll
```

在 Windows PC 上运行, 将会看到 DAG:

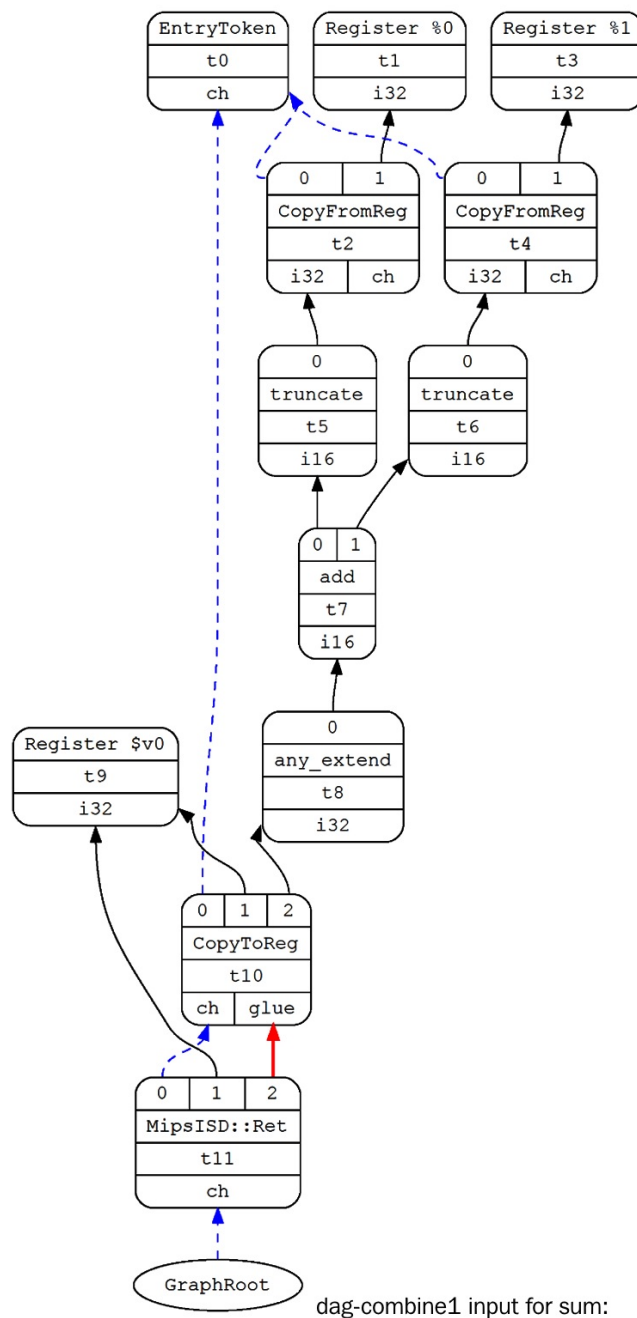


图 9.1 -使用 sum.ll 构造选择 DAG 图

一定要比较文本表示和这个图包含相同的信息。EntryToken 是 DAG 的开始，GraphRoot 是最后一个节点。控制流的链用蓝色虚线箭头标记，黑色箭头表示数据流，红色箭头将节点粘在一起，防止它们重新排序。即使对于中等大小的函数，图也会变得非常大。它并不比带有`-debug-only=isel`选项的文本输出更多或其他信息，只是表示的方式更友好。也可以在其他时间点生成图表，例如：

- 添加`--view-legalize-types-dags` 选项，在类型合法化之前查看 DAG。
- 添加`-view-isel-dags` 选项以查看指令选择。

可以使用`--help-hidden` 选项查看所有可用的 DAG 选项。因为 DAG 可能会变得很大，而且容易混淆，所以可以使用`-filter-view-dags` 选项将输出限制为一个基本块。

检查指令选择

知道了如何形象化 DAG，我们现在可以深入到细节。选择 DAG 是由 IR 构建的。对于 IR 中的每个函数，SelectionDAG 类的一个实例由 SelectionDAGBuilder 类填充。不过，在此步骤中没有进行特殊的优化。然而，目标需要提供一些函数来降级调用、参数处理、返回跳转等。为此，目标必须实现 targetlower 接口。在目标文件夹中，源文件通常在 XXXISelLowering.h 和 XXXISelLowering.cpp 文件中。targetlower 接口的实现提供指令过程所需的所有信息，例如：目标上支持哪些数据类型和哪些操作。

优化步骤会运行几次。优化器执行简单的优化，例如：在支持这些操作的目标上进行标识重排。这里的基本原理是生成一个清理过的 DAG，这里简化了其他步骤。

在类型合法化步骤期间，目标上不支持的类型将被支持的类型替换，例如：如果目标本机只支持 32 位范围的整数，那么较小的值必须通过符号或 0 扩展名转换为 32 位。如果一个 64 位的值不能被这个目标处理，那么这个值必须被分割成一对 32 位的值。向量类型也以类似的方式处理，vector 类型可以使用额外的元素进行扩展，也可以将其分解为几个值。例如，一个有四个值的向量可以分成两个各有两个值的向量。如果拆分过程以单个值结束，则找不到合适的向量，而使用标量类型。有关支持类型的信息是在 targetlower 接口的特定目标实现中配置的。在类型合法化之后，选择 DAG 转存文本表示到 sum.ll 文件中：

```
Optimized type-legalized selection DAG: %bb.0 'sum:'  
SelectionDAG has 9 nodes:  
  t0: ch = EntryToken  
    t2: i32,ch = CopyFromReg t0, Register:i32 %0  
    t4: i32,ch = CopyFromReg t0, Register:i32 %1  
  t12: i32 = add t2, t4  
  t10: ch,glue = CopyToReg t0, Register:i32 $v0, t12  
  t11: ch = MipsISD::Ret t10, Register:i32 $v0, t10:1
```

如果与初始构造的 DAG 进行比较，则这里只使用了 32 位寄存器。因为本地只支持 32 位值，所以提升了 16 位。

操作合法化类似于类型合法化。这个步骤是必要的，因为目标不是所有的操作都能支持，或者即使目标本地支持某种类型，也不能对所有操作都有效。例如，并不是所有的目标都有一个针对总体计数的本机指令。在这种情况下，操作被一系列实现功能的操作所取代。如果该类型不适合操作，那么可以将该类型提升为更大的类型。后端作者也可以提供自定义代码。如果合法化操作设置为 Custom，那么 targetlower 类中的 LowerOperation() 方法将用于这些操作。然后，该方法必须创建该操作的合法版本。sum.ll 的例子中，添加操作是合法的，因为平台上支持添加两个 23 位寄存器，所以没有任何改变。

在类型和操作合法化之后，就会进行指令选择，大部分的选择是自动化的。请记住，在上一节中，指令的描述中提供了一个模式。根据这些描述，llvm-tblgen 工具生成一个模式匹配器。基本上，模式匹配器试图找到与当前 DAG 节点匹配的模式。然后选择与模式相关联的指令。模式匹配器实现为字节码解释器，解释器的可用代码定义在 llvm/CodeGen/SelectionDAGISel.h 头文件中。XXXISelDAGToDAG 类实现了目标的指令选择。对每个 DAG 节点调用 Select() 方法。默认情况

是调用生成的匹配器，但也可以为它不支持的情况添加支持。

值得注意的是，选择 DAG 节点和选择的指令之间没有一对一的关系。一个 DAG 节点可以扩展成几个指令，而几个 DAG 节点可以分解成一条指令。前者的一个例子是合成直接值，特别是在 RISC 体系结构中，直接值的位长是受到限制的。一个 32 位的目标可能只支持 16 位长度的立即数。要执行需要 32 位常量值的操作，通常将其分割为两个 16 位值，然后生成两个或更多使用 16 位值的指令。其中，您可以在 MIPS 目标中找到这种模式。位域指令是后一种情况的一个常见例子：and、or 和 shift DAG 节点的组合通常可以匹配到特殊的位域指令，导致两个或多个 DAG 节点只使用一条指令。

通常，可以在目标描述中指定一个模式来组合两个或多个 DAG 节点。对于较复杂的、不易用模式处理的情况，可以将顶层节点的操作标记为需要特殊的 DAG 组合处理。对于这些节点，将调用 XXXISelLowering 类中的 PerformDAGCombine() 方法。然后可以检查任意的复杂模式，如果找到匹配，则可以返回表示组合 DAG 节点的操作。在为 DAG 节点运行生成的匹配器之前调用此方法。

您可以按照选择过程的打印输出来生成 sum.ll 文件。对于 add 操作，可以在这里找到以下几行：

```
ISEL: Starting selection on root node: t12: i32 = add t2, t4
ISEL: Starting pattern match
  Initial Opcode index to 27835
  ...
  Morphed node: t12: i32 = ADDu t2, t4
ISEL: Match complete!
```

索引号指向生成的匹配器的数组。从索引 27835 开始 (一个任意值，可以在不同的版本中更改)，经过一些步骤后，会选择 ADDu 指令。

模式匹配

如果遇到模式问题，还可以通过读取生成的字节码来回溯匹配。可以在 lib/Target/XXX/XXXGenDAGIsel.inc 中找到源代码，在文本编辑器中打开文件并在前面的输出中搜索索引。每一行都以索引号作为前缀，因此可以很容易地在数组中找到正确的位置。所使用的谓词也会以注释的形式打印出来，这些谓词可以帮助您理解为什么没有选择某个模式。

将 DAG 转换为指令序列

在指令选择之后，代码仍然是一个图形。这个数据结构需要序列化，这意味着指令必须按顺序排列。图中包含数据和控制流的依赖关系，但总是有几种可能以满足这些依赖关系的方式对指令进行排序。我们想要的是一份能充分利用硬件的顺序。现代硬件可以并行地发出多条指令，但总是有限制。这种限制的一个简单例子是一条指令需要另一条指令的结果。在这种情况下，硬件可能不能同时发出两个指令，而是按顺序执行指令。

您可以将调度模型添加到目标描述中，该描述描述可用的单元及其属性。例如，如果一个 CPU 有两个整数算术单元，那么该信息将在模型中捕获。对于每个指令，有必要知道模型的哪个部分被使用。较新的方法是使用机器指令调度程序定义一个调度模型，需要为目标描述中的每个子目标定义一个 `SchedMachineModel` 记录。基本上，这个模型由指令的输入和输出操作数以及处理器资源的定义组成。然后将这两个定义与延迟值关联在一起，可以在 `llvm/Target/TargetSched.td` 中查找此模型的预定义类型。`Lanai` 中有比较简单的模型，`SystemZ` 中比较复杂的调度模型。

还有一种基于所谓路线的旧模式。在这个模型中，将处理器单元定义为 `FuncUnit` 记录。使用这样一个单元的步数定义为一个 `InstrStage` 记录。每个指令都与一个 `itinerary` 类相关联。对于每个 `itinerary` 类，定义了由 `InstrStage` 记录组成的使用的处理器管道，以及执行所需的处理器周期数。可以在 `llvm/Target/targetitinerary.td` 中找到路线模型的预定义类型。

一些目标同时使用两种模型 (由于历史原因)。基于路线的模型是第一个添加到 LLVM 的模型，目标开始使用这个模型。5 年多的时间里，当新的机器指令调度程序被添加进来时，没有人迁移已经存在的模型。另一个原因是，使用路线模型，您不仅可以为使用多个处理器单元的指令建模，还可以指定在哪个周期中使用这些单元。然而，这个细节很少需要，如果需要，可以参考机器指令调度程序模型来定义路线，基本上也可以将该信息拉入新模型中。

如果存在，则使用调度模型以最优方式对指令进行排序。这一步之后，不再需要 DAG，并且被销毁。

使用选择 DAG 执行指令选择可以产生几乎最优的结果，但这是以运行时和内存使用为代价的。因此，我们开发了可供选择的方法，下面将对其进行研究。在下一节中，我们将介绍快速指令选择方法。

快速指令选择-FastISel

使用选择 DAG 进行指令选择会消耗编译时间。如果正在开发一个应用程序，那么编译器的运行时很重要。可能也不太关心生成的代码，因为发出完整的调试信息更为重要。由于这些原因，LLVM 开发人员决定实现一个特殊的指令选择器，它有一个快速的运行时，但产生的最优代码较少，只用于-O0 优化级别。该组件称为快速指令选择，简称 `FastISel`。

实现在 `XXXFastISel` 类中。不是每个目标都支持这种指令选择方法，在这种情况下，选择 DAG 方法也用于-O0。实现很简单：一个特定于目标的类派生自一个 `FastISel` 类，并且必须实现两个方法。`TableGen` 工具从目标描述生成大部分所需的代码。然而，实现这个指令选择器还需要一些工作量。根本原因是您需要获得正确的调用规则，所以比较复杂。

MIPS 目标的特点是实现快速指令选择。您可以使用快速指令选择通过传递 `-fast-isel` 选项使用 `llc` 工具。使用 `sum.ll` 文件：

```
$ llc -mtriple=mips-linux-gnu -fast-isel -O0 sum.ll
```

快速指令选择运行非常快，但它是一个完全不同的代码路径。一些 LLVM 开发人员决定寻找一种既能快速运行又能生成良好代码的解决方案，目标是在未来同时替换选择 DAG 和快速指令选择器。我们将在下一节中介绍这种方法。

新的全局指令选择-GlobalISel

使用选择 DAG，我们可以生成非常好的机器代码。缺点是它对于软件来说，它非常复杂。这意味着很难开发、测试和维护。FastISel 指令选择工作迅速且不那么复杂，但不能生成良好的代码。除了 TableGen 生成的代码外，这两种方法都不共享太多代码。

能两全其美吗？一个指令选择算法，是快速，容易实现，并产生良好的代码？这就是在 LLVM 框架中添加另一种指令选择算法——全局指令选择的动机。短期目标是首先取代 FastISel，长期目标是选择 DAG。

全局指令选择的方法是在现有的基础上进行，整个任务被分解成一系列的机器功能传递。另一个主要的设计决策是不引入另一个中间表示，而是使用现有的 MachineInstr 类。但是，添加了新的泛型操作码。

当前的步骤顺序如下：

1. IRTranslator 通过使用通用操作码构建初始机器指令。
2. Legalizer 可以将类型和操作合法化。这与选择 DAG 不同，后者使用两个不同的步骤。真实的 CPU 架构有时会很奇怪，而且有可能某一数据类型只被一条指令支持。这种情况不能由选择 DAG 很好地处理，但是在全局指令选择的组合步骤中很容易处理。
3. 生成的机器指令仍然在虚拟寄存器上运行。在 RegBankSelect 通道中，选择一个注册。寄存器组表示 CPU 上的一种寄存器，例如：通用寄存器。这比目标描述中的寄存器定义更粗粒度。重要的一点是，它将类型信息与指令关联起来。类型信息基于目标中可用的类型，因此这已经低于 LLVM IR 中的泛型类型。
4. 此时，已知类型和操作对目标是合法的，并且类型信息与每个指令相关联。下面的 InstructionSelect Pass 可以轻松地用机器指令替换通用指令。

在全局指令选择之后，后端程序（如指令调度、寄存器分配和基本块放置）就会运行。

全局指令选择编译到 LLVM 中，但默认情况下不启用。如果要使用它，需要将-global-isel 选项赋予 llc，或将-mllvm global-isel 选项赋予 clang。如果全局指令选择不能处理 IR 构造，则可以进行手动控制。当给 llc 提供-global-isel-abort=0 选项时，选择 DAG 会作为备选。当 =1 时，应用程序终止。为了防止这种情况，可以给 llc 提供-global-isel-abort=0 选项。如果是 =2，选择 DAG 将作为备选，并打印一条诊断信息来进行通知。

要将全局指令选择添加到目标，只需要覆盖目标的 TargetPassConfig 类中的相应函数。这个类由 XXXTargetMachine 类实例化，实现通常在同一个文件中找到。例如，重写 addIRTranslator() 方法，可以将 IRTranslator 传递添加到目标的机器传递中。

开发主要在 AArch64 目标上进行，该目标目前对全局指令选择有最好的支持。许多其他目标，包括 x86 和 Power，也增加了对全局指令选择的支持。这里挑战是，从表描述中生成的代码并不多，因此仍然需要手工编写大量代码。另一个挑战是目前还不支持大端目标，所以像 SystemZ 这样的纯大端目标目前还不能使用全局指令选择。随着时间的推移，两者都肯定会得到改善。

Mips 目标的特点是实现了全局指令选择，但有提到的限制，即它只能用于小端目标。你可以通过将-global-isel 选项传递给 llc 工具来启用全局指令选择。使用 sum.ll:

```
$ llc -mtriple=mipsel-linux-gnu -global-isel sum.ll
```

请注意，目标 mipsel-linux-gnu 是小端目标。使用大端位的 mips-linux-gnu 目标会导致错误。

全局指令选择器的工作速度比选择 DAG 快得多，并且已经产生了比快速指令选择更高质量的代码。

支持新的机器指令

您的目标 CPU 可能有 LLVM 还不支持的机器指令。例如，使用 MIPS 架构的制造商经常向核心 MIPS 指令集添加特殊指令。RISC-V 指令集的规范明确允许制造商添加新的指令，或者要添加一个全新的后端，然后必须添加 CPU 指令。在下一节中，我们将为 LLVM 后端添加对单个新机器指令的汇编器支持。

向汇编程序和代码生成添加新指令

新的机器指令通常与特定的 CPU 特性绑定在一起。只有当用户使用 `--mattr=` 选项选择 `llc` 时，新指令才能识别。

作为一个例子，我们将添加一个新的机器指令到 MIPS 后端。新的机器指令首先将两个输入寄存器的值平方 `$2` 和 `$3`，然后将两个平方和的和赋给输出寄存器 `$1`：

```
sqsumu $1, $2, $3
```

该指令的名称是 `sqsumu`，由平方和求和运算派生而来。名称中的最后一个 `u` 表示该指令适用于无符号整数。

我们首先添加的 CPU 特性称为 `sqsum`。这将允许使用 `--mattr+=sqsum` 选项调用 `llc`，以启用对新指令进行识别。

我们将添加的大部分代码都在描述 MIPS 后端的 TableGen 文件中。所有文件都位于 `llvm/lib/Target/Mips` 文件夹中。顶层文件为 `Mips.td`，查看该文件并找到定义各种特性的部分。这里添加了我们新特性的定义：

```
def FeatureSQSum
  : SubtargetFeature<"sqsum", "HasSQSum", "true",
    "Use square-sum instruction">;
```

`SubtargetFeature` 类接受四个模板参数。第一个是 `sqsum`，它是特性的名称，用于命令行。第二个参数 `HasSQSum` 是表示此特性的 `Subtarget` 类中的属性名称。下一个参数是默认值和特性的描述，用于在命令行上提供帮助。TableGen 为 `MipsSubtarget` 类生成基类，其在 `MipsSubtarget.h` 文件中定义。这个文件中，类的私有部分添加了新属性，所有其他属性都在这里定义：

```
1 // Has square-sum instruction.
2 bool HasSQSum = false;
```

在 `public` 部分，还使用了一个方法来检索属性的值：

```
1 bool hasSQSum() const { return HasSQSum; }
```

通过这些那方法，我们已经能够在命令行上设置 `sqsum` 特性，尽管目前还没有效果。

要将新指令绑定到 `sqsum` 特性，需要定义一个谓词来指示是否选择了该特性。我们将其添加到 `MipsInstrInfo.td` 文件中，或者在定义所有其他谓词的部分，亦或者简单地放在结尾：

```
def HasSQSum : Predicate<"Subtarget->hasSQSum()">,
    AssemblerPredicate<(all_of FeatureSQSum)>;
```

谓词使用前面定义的 `hasSQSum()` 方法。此外，`AssemblerPredicate` 模板指定为汇编程序生成源代码时所使用的条件，只是简单地引用前面定义的特性。

我们还需要更新调度模型。MIPS 目标同时使用行程和机器指令调度程序，对于路线模型，为 `MipsSchedule` 中的每条指令定义一个 `instritclass.td` 记录文件。在此文件中定义所有路线的部分中添加以下行：

```
def II_SQSUMU : InstrItinClass;
```

我们还需要提供说明费用的细节。通常，可以在 CPU 的文档中找到这些信息。对于我们的指令，乐观地假设它在 ALU 中只需要一个循环。该信息能添加到的 `MipsGenericItineraries` 定义中：

```
InstrItinData<II_SQSUMU, [InstrStage<1, [ALU]>]>
```

这样，对基于路线的调度模型的更新就完成了。MIPS 目标还在 `MipsScheduleGeneric.td` 文件中，定义了基于 `MipsScheduleGeneric` 中的机器指令调度器模型的通用调度模型。因为这是一个包含所有指令的完整模型，所以还需要添加指令 `add`。因为它是基于乘法的，所以我们简单地扩展了 `MULT` 和 `MULTu` 指令的定义：

```
def : InstRW<[GenericWriteMul], (instrs MULT, MULTu, SQSUMu)>;
```

MIPS 目标还在 `MipsScheduleP5600.td` 中定义了 P5600 CPU 的调度模型。这个目标显然不支持我们的新指令，所以把它添加到不支持的特性列表中：

```
list<Predicate> UnsupportedFeatures = [HasSQSum, HasMips3, ...
```

现在，我们准备在 `Mips64InstrInfo.td` 文件的末尾添加新指令。`TableGen` 的定义总是简洁的，因此需要对其进行剖析。该定义使用了来自 MIPS 目标描述的一些预定义类，我们的新指令是一个算术指令。通过设计，它适合于 `ArithLogicR` 类。第一个参数 `sqsumu` 指定指令的汇编助记符，下一个参数 `GPR64Opnd` 表示指令使用 64 位寄存器作为操作数，下面的 1 个参数表示操作数是可交换的。最后，给出了路线指令。添加 `_FM` 类用于指定指令的二进制编码。对于真正的指令，必须根据文档选择参数。然后跟随 `ISA_MIPS64` 谓词，该谓词指示指令对哪个指令集是有效的。最后，我们的 `SQSUM` 声明，只有在启用我们的特性时，指令才有效。完整的定义如下：


```
def SQSUMu : ArithLogicR<"sqsumu", GPR64Opnd, 1, II_SQSUMU>,  
    ADD_FM<0x1c, 0x28>, ISA_MIPS64, SQSUM
```

如果您的目标只是支持新指令，那么这个定义就够了，所以一定要完成的定义。通过添加选择 DAG 模式，代码生成器可以使用该指令。该指令使用两个操作数寄存器 \$rs 和 \$rt 以及目标寄存器 \$rd，这三个寄存器都由 ADD_FM 二进制格式类定义。理论上，要匹配的模式很简单：使用乘数运算符将每个寄存器的值平方，然后使用 add 运算符将两个乘积相加，并将它们赋给目标寄存器 \$rd。这个模式变得有点复杂，因为使用 MIPS 指令集，乘法的结果存储在特殊的寄存器对中。为了便于使用，结果必须移到通用寄存器中。在操作的合法化过程中，通用 mul 操作符被 MIPS 特定的 MipsMult 操作所取代，用于乘法运算和 MipsMFLO 运算，以将结果的下一部分移动到通用寄存器中。在编写模式时，必须考虑到这一点，模式如下所示：

```
{  let Pattern = [(set GPR64Opnd:$rd,  
                    (add (MipsMFLO (MipsMult  
                                GPR64Opnd:$rs,  
                                GPR64Opnd:$rs)),  
                        (MipsMFLO (MipsMult  
                                GPR64Opnd:$rt,  
                                GPR64Opnd:$rt)))  
                    )];  
}
```

正如在带选择 DAG 部分的指令选择中所描述的，如果该模式与当前 DAG 节点匹配，则选择我们的新指令。由于 SQSUM 谓词，这只在激活 SQSUM 特性时发生。我们用一个测试来检查它！

测试新指令

如果扩展了 LLVM，那么最好使用自动化测试来验证它。如果想将您的扩展贡献给 LLVM 项目，那么就需要良好的测试。

在像上一节一样添加一个新的机器指令之后，我们必须进行两个检查：

- 首先，必须验证指令编码是否正确。
- 其次，必须确保代码生成按照预期工作。

LLVM 项目使用 LIT(LLVM Integrated Tester) 作为测试工具。基本上，测试用例是一个包含输入、要运行的命令和应该执行的检查的文件。添加新测试就像将一个文件复制到测试目录中一样简单。为了验证新指令的编码，使用 llvm-mc 工具。除了其他任务外，该工具还可以显示指令的编码。对于临时检查，可以执行以下命令显示编码指令：

```
$ echo "sqsumu \$1,\$2,\$3" | \  
  llvm-mc --triple=mips64-linux-gnu -mattr=+sqsum \  
    -show-encoding
```

这已经显示了要在自动化测试用例中运行的部分输入和命令。要验证结果，可以使用 FileCheck 工具，llvm-mc 的输出通过管道传输到这个工具中。另外，FileCheck 读取测试用例文件。测试用例文件包含用 CHECK: 关键字标记的行，其后是预期的输出。FileCheck 尝试将这些行与传入它的数据进行匹配。如果没有找到匹配，则显示一个错误。将包含测试用例文件 sqsumu.s 放到 llvm/test/MC/Mips 目录中：

```
# RUN: llvm-mc %s -triple=mips64-linux-gnu -mattr=+sqsum \  
# RUN: --show-encoding | FileCheck %s  
# CHECK: sqsumu $1, $2, $3 # encoding: [0x70,0x43,0x08,0x28]  
  
sqsumu $1, $2, $3
```

如果位于 llvm/test/Mips/MC 文件夹中，那么可以使用以下命令运行测试，并在最后报告成功：

```
$ llvm-lit sqsumu.s  
-- Testing: 1 tests, 1 workers --  
PASS: LLVM :: MC/Mips/sqsumu.s (1 of 1)  
Testing Time: 0.11s  
Passed: 1
```

LIT 工具解释 RUN:line，用当前文件名替换%s。FileCheck 工具读取文件，并解析 CHECK:line，尝试匹配来自流水的输入。这是一种非常有效的测试方法。

如果位于构建目录中，则可以使用以下命令调用 LLVM 测试：

要为代码生成构建一个测试用例，需要遵循相同的策略。sqsum.ll 文件包含计算斜边平方的 LLVM IR 代码：

```
define i64 @hyposquare(i64 %a, i64 %b) {  
    %asq = mul i64 %a, %a  
    %bsq = mul i64 %b, %b  
    %res = add i64 %asq, %bsq  
    ret i64 %res  
}
```

要查看生成的汇编代码，可以使用 llc 工具：

```
$ llc -mtriple=mips64-linux-gnu -mattr=+sqsum < sqsum.ll
```

确信在输出中看到了新的 `sqsum` 指令。如果删除 `-mattr=+sqsum` 选项，还请检查该指令是否未生成。

有了这些，就可以构建测试用例了。这一次，使用两个 `RUN:line:` 一行检查新指令是否生成，另一行检查是否没有生成。我们可以用一个测试用例文件做到这两点，因为可以告诉 `FileCheck` 工具寻找与“`CHECK:`”不同的标签。将测试文件 `sqsum.ll` 和以下内容放置到 `llvm/test/CodeGen/Mips` 文件夹：

```
; RUN: llc -mtriple=mips64-linux-gnu -mattr=+sqsum < %s |\n; RUN: FileCheck -check-prefix=SQSUM %s\n; RUN: llc -mtriple=mips64-linux-gnu < %s |\n; RUN: FileCheck -check-prefix=NOSQSUM %s\n\ndefine i64 @hyposquare(i64 %a, i64 %b) {\n; SQSUM-LABEL: hyposquare:\n; SQSUM: sqsumu $2, $4, $5\n; NOSQSUM-LABEL: hyposquare:\n; NOSQSUM: dmult $5, $5\n; NOSQSUM: mflo $1\n; NOSQSUM: dmult $4, $4\n; NOSQSUM: mflo $2\n; NOSQSUM: addu $2, $2, $1\n    %asq = mul i64 %a, %a\n    %bsq = mul i64 %b, %b\n    %res = add i64 %asq, %bsq\n    ret i64 %res\n}
```

与其他测试一样，可以使用以下命令在文件夹中单独运行测试：

```
$ llvm-lit squm.ll
```

或者，可以使用以下命令从构建目录运行：

```
$ ninja check-llvm-mips-codegen
```

通过这些步骤，可以使用新指令增强 LLVM 汇编程序，允许指令选择使用这个新指令，并验证编码是否正确，是否按照预期完成代码生成的工作。

总结

本章中，您学习了 LLVM 目标的后端是如何构造的。使用 MIR 检查通过后的状态，并使用机器 IR 运行单次通过。有了这些知识，您就可以研究后端传递中的问题。

了解了如何在 LLVM 中使用选择 DAG 实现指令选择，还介绍了使用 FastISel 和 GlobalISel 进行指令选择的替代方法，如果平台提供了所有这些方法，这有助于决定选择哪一种算法。

您扩展了 LLVM 以在汇编程序和指令选择中支持新的机器指令，这有助于添加对当前不受支持的 CPU 特性的支持。为了验证扩展，需要为它添加自动化测试用例。

下一章中，我们将研究 LLVM 的另一个独特特性：一步生成和执行代码，也称为 JIT 编译。

第 10 章 JIT 编译

LLVM 核心库附带了 ExecutionEngine 组件，允许在内存中编译和执行 IR 代码。使用这个组件，我们可以构建即时 (JIT) 编译器，它允许直接执行 IR 代码。JIT 编译器的工作方式更像解释器，因为不需要将目标代码存储在辅助存储器上。

在本章中，您将了解 JIT 编译器的应用程序，以及 LLVM JIT 编译器的原理。您将探索 LLVM 动态编译器和解释器，还将学习如何自己实现 JIT 编译器工具。了解如何使用 JIT 编译器作为静态编译器的一部分，以及与之相关的挑战。

本章将包含以下内容：

- 概述 LLVM 的 JIT 实现和用例
- 使用 JIT 编译直接执行
- 利用 JIT 编译器进行代码计算

在本章结束时，您将知道如何开发 JIT 编译器，可以使用预先配置的种类，也可以使用符合您需求的定制版本。您还将了解如何在传统静态编译器中使用 JIT 编译器。

相关代码

本章的代码文件可在<https://github.com/PacktPublishing/Learn-LLVM-12/tree/master/Chapter10>获取。

你可以在视频中找到代码<https://bit.ly/3nllhED>。

概述 LLVM 的 JIT 实现和用例

目前为止，我们只研究了提前 (AOT) 编译器，这些编译器会编译整个应用程序。只有编译完成后，应用程序才能运行。如果编译是在应用程序的运行时执行的，那么编译器就是一个 JIT 编译器。JIT 编译器有一些有趣的用例：

- 虚拟机的实现：可以用 AOT 编译器将编程语言翻译成字节代码。在运行时，使用 JIT 编译器将字节代码编译为机器码。这种方法的优点是字节码是独立于硬件的，而且由于 JIT 编译器，与 AOT 编译器相比，没有性能损失。今天的 Java 和 C# 使用这个模型，但其出现的时间非常久远：1977 年的 USCD Pascal 编译器已经使用了类似的方法。
- 表达式解析：电子表格应用程序可以使用 JIT 编译器编译经常执行的表达式，这可以加速金融模拟。LLVM 调试器 LLDB 就是使用这种方法在调试时计算表达式的。
- 数据库查询：数据库从数据库查询创建执行计划。执行计划描述表和列上的操作，这些操作在运行时导致查询答案。可以使用 JIT 编译器将执行计划转换为机器码，从而加速查询

LLVM 的静态编译模型与 JIT 模型之间的差异并不像您想象的那么大。LLVM 静态编译器将 LLVM IR 编译成机器码，并将结果保存为对象文件。如果目标文件不是存储在磁盘上而是存储在内存中，那么代码是可执行的吗？不能直接执行，因为对全局函数和全局数据的引用使用的是重定位的方式，而不是绝对地址。

从概念上讲，重定位描述了如何计算地址，例如：作为已知地址的偏移量。如果将重定位解析为地址，就像链接器和动态加载器那样，那么就可以执行目标代码。让静态编译器将 IR 代码编译

成内存中的对象文件，对内存中的对象文件执行链接步骤，然后运行该代码，就得到了一个 JIT 编译器。LLVM 核心库中的 JIT 实现就是基于这种思想。

在 LLVM 的开发历史中，有几种 JIT 实现，它们具有不同的特性集。最新的 JIT API 是随请求编译 (ORC, on request compilation) 引擎。这个缩写有个小故事：在 ELF(可执行和链接格式) 和 DWARF(调试标准) 已经存在之后，首席开发人员打算创造另一个基于 Tolkien universe 的缩写。

ORC 引擎构建并扩展了在内存对象文件上使用静态编译器和动态链接器的思想。该实现使用分层的方法，两个基本层次如下：

1. 编译层
2. 连接层

在编译层之上可以有一个提供惰性编译支持的层。转换层可以堆叠在惰性编译层的顶部或下方，允许开发人员添加任意转换，或者只是得到某些事件的通知。这种分层方法优点是，JIT 引擎可以针对不同的需求进行定制，例如：高性能虚拟机可能选择预先编译所有内容，而不使用惰性编译层。其他虚拟机将强调启动时间和对用户的响应，并借助惰性编译层实现这一点。

旧 MCJIT 引擎仍然可用。这个 API 派生自一个更古老的、已经删除的 JIT 引擎。随着时间的推移，这个 API 变得有点臃肿，而且缺乏 ORC API 的灵活性。我们的目标是删除这个实现，因为 ORC 引擎现在可以提供了 MCJIT 引擎的所有功能。新的项目可以使用 ORC API。

下一节中，我们将讨论 lli, LLVM 解释器和动态编译器，然后再讨论 JIT 编译器的实现。

使用 JIT 编译直接执行

在考虑 JIT 编译器时，首先想到的是直接运行 LLVM IR。这就是 lli 工具、LLVM 解释器和动态编译器所做的工作。我们将在下一节中探索 lli 工具，然后实现一个类似的工具。

探索 lli 工具

让我们用一个非常简单的例子来试试 lli 工具。将以下源代码存储为 hello.ll 文件，它相当于 C 程序的 hello world，其声明了来自 C 库的 printf() 函数，hellostr 常量包含要打印的消息。在 main() 函数内部，通过 getelementptr 指令计算消息的第一个字符的指针，并将该值传递给 printf() 函数，程序总是返回 0。完整的源代码如下：

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

@hellostr = private unnamed_addr constant [13 x i8] c"Hello
                                         world\0A\00"

define i32 @main(i32 %argc, i8** %argv) {
    %res = call i32 @printf(
        i8* getelementptr inbounds ([13 x i8],
                                     [13 x i8]* @hellostr, i64 0, i64 0))
    ret i32 0 }
```

这个 LLVM IR 文件是通用的，对所有平台都有效。我们可以通过以下命令直接使用 lli 工具执行 IR:

```
$ lli hello.ll
Hello world
```

这里有趣的一点是如何找到 printf() 函数。IR 代码编译为机器码，并触发对 printf 符号的查找。这个符号在 IR 中找不到，所以当前进程会搜索它。lli 工具动态链接 C 库，可以在那里找到符号。

当然，lli 工具不会链接您创建的库。为了使用这些函数，lli 工具支持加载共享库和对象。下面的 C 源代码只打印了一个友好的消息:

```
1 #include <stdio.h>
2
3 void greetings() {
4     puts("Hi!");
5 }
```

存储在 greeting.c 文件中，我们使用它来使用 lli 工具探索对象的加载。将此源代码编译为动态库。-fPIC 选项指示 clang 生成位置无关的代码，这是动态库所必需的。通过给定的-shared 选项，编译器创建 greetings.so 动态库:

```
$ clang -fPIC -shared -o greetings.so greetings.c
```

我们还将该文件编译为一个 greetings.o 对象文件:

```
$ clang -c -o greetings.o greetings.c
```

我们现在有两个文件，greetings.so 和 greetings.o 文件，将他们加载到 lli 工具中。

我们还需要一个 LLVM IR 文件，它调用 greetings() 函数。为此，创建 main.ll 文件，其中包含对函数的单独调用:

```
declare void @greetings(...)

define dso_local i32 @main(i32 %argc, i8** %argv) {
    call void (...) @greetings()
    ret i32 0
}
```

如果尝试像以前一样执行 IR，那么 lli 工具无法定位 greetings 符号，就会崩溃:

```
$ lli main.ll
```

PLEASE submit a bug report to <https://bugs.llvm.org/> and
include the crash backtrace.

`greeting()` 函数定义在一个外部文件中，为了修复崩溃，必须告诉 `lli` 工具需要加载哪个文件。为了使用动态库，必须使用 `-load` 选项，它将动态库的路径作为参数：

```
$ lli -load ./greetings.so main.ll  
Hi!
```

如果包含动态库的目录不在动态加载器的搜索路径中，那么指定动态库的路径是很重要的。如果省略，则将找不到库。

或者，可以使用 `-extobject` 选项来指示 `lli` 工具加载的对象文件：

```
$ lli -extra-object greetings.o main.ll  
Hi!
```

其他支持的选项是 `-extra-archive` (加载静态库文件)，和 `-extramodule` (加载比特码文件)。这两个选项都要求将文件路径作为参数。

现在，知道了如何使用 `lli` 工具直接执行 LLVM IR。在下一节中，将实现自己的 JIT 工具。

用 LLJIT 实现自己的 JIT 编译器

`lli` 工具只不过是 LLVM API 的一个包装器。在之前的内容中，我们了解到 ORC 引擎使用分层方法，`ExecutionSession` 类表示一个正在运行的 JIT 程序。除了其他项外，这个类还保存使用的 `JITDylib` 实例。`JITDylib` 实例是一个符号表，将符号名映射到地址，例如：可以是 LLVM IR 文件中定义的符号，也可以是加载的动态库的符号。

要执行 LLVM IR，不需要自己创建一个 JIT 堆栈，`LLJIT` 类提供了这个功能。您还可以在从旧的 `MCJIT` 实现迁移时使用这个类，这个类本质上提供了相同的功能。在下一小节中，我们将从 JIT 引擎的初始化开始实现。

为编译 LLVM IR 初始化 JIT 引擎

我们首先实现设置 JIT 引擎的函数，编译 LLVM IR 模块，并在该模块中执行 `main()` 函数。稍后，我们将使用这个核心功能构建一个小型 JIT 工具。先是 `jitmain()` 函数：

1. 该函数需要带有 IR 的 LLVM 模块来执行。这个模块还需要 LLVM context 类，因为 context 类包含重要的类型信息。我们的目标是调用 `main()` 函数，所以也传递了 `argc` 和 `argv` 参数：

```
1 Error jitmain(std::unique_ptr<Module> M,  
2             std::unique_ptr<LLVMContext> Ctx, int  
3             argc,  
4             char *argv[]) {
```


2. 我们使用 LLJITBuilder 类创建一个 LLJIT 实例。如果发生错误，则返回错误。一个可能的错误原因是平台还不支持 JIT 编译:

```
1 auto JIT = orc::LLJITBuilder().create();
2 if (!JIT)
3     return JIT.takeError();
```

3. 然后将模块添加到主 JITDylib 实例中。如果配置了，那么 JIT 编译可以使用多个线程。因此，需要将模块和上下文包装在 ThreadSafeModule 实例中。如果发生错误，则返回错误:

```
1 if (auto Err = (*JIT)->addIRModule(
2     orc::ThreadSafeModule(std::move(M),
3         std::move(Ctx))))
4     return Err;
```

4. 与 lli 工具一样,我们也支持来自 C 库的符号。DefinitionGenerator 类公开符号,DynamicLibrarySearchGenerator 子类公开在动态库中找到的名称。这个类提供了两个工厂方法: Load() 方法可用于加载动态库,而 GetForCurrentProcess() 方法公开当前进程的符号。这里,我们使用后一个函数。符号名可以有前缀,这取决于平台。我们检索数据布局并将前缀传递给 GetForCurrentprocess() 函数。然后以正确的方式处理符号名,我们不需要关心它。像往常一样,如果发生错误,则返回函数:

```
1 const DataLayout &DL = (*JIT)->getDataLayout();
2 auto DLSG = orc::DynamicLibrarySearchGenerator::
3     GetForCurrentProcess(DL.getGlobalPrefix());
4 if (!DLSG)
5     return DLSG.takeError();
```

5. 然后将生成器添加到主 JITDylib 实例中。如果需要查找一个符号,也会搜索加载的动态库中的符号:

```
1 (*JIT)->getMainJITDylib().addGenerator(
2     std::move(*DLSG));
```

6. 接下来,我们需要查找主符号。这个符号必须在命令行中给出的 IR 模块中,查找触发 IR 模块的编译。如果 IR 模块中引用了其他符号,则使用前一步中添加的生成器解析它们。结果类型是 JITEvaluatedSymbol 类:

```
1 auto MainSym = (*JIT)->lookup("main");
2 if (!MainSym)
3     return MainSym.takeError();
```

7. 我们向返回的 JIT 符号询问函数的地址,并将这个地址转换为 C main() 函数的原型:

```
1 auto *Main = (int (*)(
2     int, char **))MainSym->getAddress();
```

8. 现在我们可以调用 IR 模块中的 main() 函数，并传递 argc 和 argv 参数。这里，忽略返回值:

```
1 (void)Main(argc, argv);
```

9. 报告函数确定函数是否执行成功:

```
1 return Error::success();  
2 }
```

使用 JIT 编译还挺容易。除了公开当前进程的符号或动态库中的符号外，还有许多其他的方法可以公开名称。StaticLibraryDefinitionGenerator 类公开在静态归档文件中找到的符号，可以像 DynamicLibrarySearchGenerator 类一样使用。LLJIT 类还有一个 addObjectFile() 方法来公开对象文件的符号。如果现有的实现不能满足您的需要，还可以提供自己的 DefinitionGenerator 实现。在下一小节中，将实现扩展到 JIT 编译器中。

创建 JIT 编译器工具

jitmain() 函数可以很容易地扩展为一个小工具。源代码保存在 JIT.cpp 文件中，是一个简单的 JIT 编译器:

1. 我们必须包含几个头文件 LLJIT.h 头文件定义了 LLJIT 类和 ORC API 的核心类。IRReader.h 头文件，定义了一个读取 LLVM IR 文件的函数。CommandLine.h 允许以 LLVM 风格解析命令行选项。最后，InitLLVM.h 用于工具的基本初始化，TargetSelect.h 用于初始化本机目标:

```
1 #include "llvm/ExecutionEngine/Orc/LLJIT.h"  
2 #include "llvm/IRReader/IRReader.h"  
3 #include "llvm/Support/CommandLine.h"  
4 #include "llvm/Support/InitLLVM.h"  
5 #include "llvm/Support/TargetSelect.h"
```

2. 将 llvm 命名空间添加到当前作用域:

```
1 using namespace llvm;
```

3. 我们的 JIT 工具只需要一个输入文件，用 cl::opt<> 类声明它:

```
1 static cl::opt<std::string>  
2 InputFile(cl::Positional, cl::Required,  
3 cl::desc("<input-file>"));
```

4. 要读取 IR 文件，调用 parseIRFile() 函数。该文件可以是文本 IR，也可以是位码文件，函数返回指向所创建模块的指针。错误处理略有不同，因为文本 IR 文件可以解析，但语法不一定正确。SMDiagnostic 实例保存语法错误时的错误信息，输出错误消息，并退出应用程序:

```
1 std::unique_ptr<Module>  
2 loadModule(StringRef Filename, LLVMContext &Ctx,  
3 const char *ProgName) {
```

```

4   SMDiagnostic Err;
5   std::unique_ptr<Module> Mod =
6       parseIRFile(Filename, Err, Ctx);
7   if (!Mod.get()) {
8       Err.print(ProgName, errs());
9       exit(-1);
10  }
11  return std::move(Mod);
12 }

```

5. jitmain() 函数放在这里:

```

1 Error jitmain(...) { ... }

```

6. 然后添加 main() 函数, 它初始化工具和本机目标, 并解析命令行:

```

1 int main(int argc, char *argv[]) {
2     InitLLVM X(argc, argv);
3
4     InitializeNativeTarget();
5     InitializeNativeTargetAsmPrinter();
6     InitializeNativeTargetAsmParser();
7
8     cl::ParseCommandLineOptions(argc, argv,
9                                 "JIT\n");

```

7. 接下来, 初始化 LLVM context 类:

```

1 auto Ctx = std::make_unique<LLVMContext>();

```

8. 然后加载命令行中命名的 IR 模块:

```

1 std::unique_ptr<Module> M =
2     loadModule(InputFile, *Ctx, argv[0]);

```

9. 然后可以调用 jitmain() 函数, 我们使用 ExitOnError 实用程序类处理错误。当发生错误时, 该类打印错误消息并退出应用程序。还可以设置了一个带有应用程序名称的标识, 它会将之前的错误消息打印出来:

```

1 ExitOnError ExitOnErr(std::string(argv[0]) + ": ");
2 ExitOnErr(jitmain(std::move(M), std::move(Ctx),
3                   argc, argv));

```

10. 如果控制流到达此处, 则成功执行了 IR。返回 0 就表示成功:

```

1 return 0;
2 }

```

这已经是完整的实现了! 我们只需要添加构建描述。

添加 CMake 构建

为了编译这个源文件，我们还需要创建一个带有构建描述的 CMakeLists.txt 文件，它保存在 JIT.cpp 文件之外:

1. 我们将最低要求的 CMake 版本设置为 LLVM 所需的版本，并将项目命名为 jit:

```
cmake_minimum_required (VERSION 3.13.4)
project ("jit")
```

2. 需要加载 LLVM 包，将 LLVM 提供的 CMake 模块目录添加到搜索路径中。然后包含 ChooseMSVCCRT 模块，确保与 LLVM 使用的 C 运行时相同:

```
find_package(LLVM REQUIRED CONFIG)
list(APPEND CMAKE_MODULE_PATH ${LLVM_DIR})
include(ChooseMSVCCRT)
```

3. 还需要从 LLVM 添加定义和包含路径。使用的 LLVM 组件通过函数调用映射到库名:

```
add_definitions(${LLVM_DEFINITIONS})
include_directories(SYSTEM ${LLVM_INCLUDE_DIRS})
llvm_map_components_to_libnames(llvm_libs Core OrcJIT
                                Support
                                native)
```

4. 最后，定义可执行文件的名称、要编译的源文件和要链接的库:

```
add_executable(JIT JIT.cpp)
target_link_libraries(JIT ${llvm_libs})
```

5. 这就是 JIT 工具所需要的一切。创建并更改到 build 目录，然后运行以下命令来创建和编译应用程序:

```
$ cmake -G Ninja <path to source directory>
$ ninja
```

这将用来编译 JIT 工具，并且可以用 hello.ll 检查工具的功能:

```
$ JIT hello.ll
Hello world
```

创建 JIT 编译器就是这么简单!

该示例使用 LLVM IR 作为输入,但这不是必需的。LLJIT 类使用 `ircompilayer` 类,负责将 IR 编译成机器码。您可以定义自己的层,可以接受需要的输入,例如 Java 字节码。

使用预定义的 LLJIT 类很方便,但是限制了灵活性。下一节中,我们将讨论如何使用 ORC API 提供的层来实现 JIT 编译器。

从头开始构建 JIT 编译器类

使用 ORC 的分层方法,可以很容易地构建一个定制的 JIT 编译器。没有适合所有人的 JIT 编译器,本章的第一节给出了一些示例。让我们看看如何设置 JIT 编译器。

ORC API 将这些层是堆叠在一起的。最低的一层是对象链接层,由 `llvm::orc::RTDyldObjectLinkingLayer` 类表示,负责链接内存中的对象并将它们转换为可执行代码。此任务所需的内存由 `MemoryManager` 接口的一个实例管理,有一个默认实现,如果需要,也可以使用自定义版本。

在对象链接层之上是编译层,它负责创建内存中的对象文件。`llvm::orc::IRCompileLayer` 类将一个 IR 模块作为输入,并将其编译为对象文件。`IRCompileLayer` 类是 `IRLayer` 类的子类,`IRLayer` 类是一个通用类,用于接受 LLVM IR 的层实现。

这两层已经构成了 JIT 编译器的核心。添加一个 LLVM IR 模块作为输入,在内存中编译和链接。为了添加更多的功能,可以在它们之上添加更多的层,例如:`CompileOnDemandLayer` 类拆分一个模块,以便只编译被请求的函数。这可以用来实现延迟编译。`CompileOnDemandLayer` 类也是 `IRLayer` 类的子类。以一种非常通用的方式,`IRTransformLayer` 类 (也是 `IRLayer` 类的子类) 允许我们对模块应用转换。

另一个重要的类是 `ExecutionSession` 类,这个类表示一个正在运行的 JIT 程序。基本上,这意味着该类管理 `JITDylib` 符号表,提供符号查找功能,并跟踪所使用的资源管理器。

JIT 编译器的一般流程如下:

1. 初始化 `ExecutionSession` 类的实例。
2. 初始化层,至少包括 `RTDyldObjectLinkingLayer` 类和 `IRCompileLayer` 类。
3. 创建第一个 `JITDylib` 符号表,通常使用 `main` 或类似的名称。

它的用法与上一节提到的 LLJIT 类非常相似:

4. 将 IR 模块添加到符号表中。
5. 查找一个符号,触发关联函数的编译,可能还有整个模块。
6. 执行函数。

在下一小节中,我们将基于泛型配置实现一个 JIT 编译器类。

创建 JIT 编译器类

为了保持 JIT 编译器类的实现简单,我们将所有内容都放到 `JIT.h` 头文件中。类的初始化稍微复杂一些。由于要处理错误,需要一个工厂方法在调用构造函数之前预先创建一些对象。创建类的步骤如下:

1. 首先,使用 `JIT_H` 预处理器定义来保护头文件不被重复包含:

```
1 #ifndef JIT_H
```

```
2 #define JIT_H
```

2. 需要一堆包含文件。它们中的大多数都提供了与头文件同名的类。Core.h 头文件提供了两个基本类，包括 ExecutionSession 类。ExecutionUtils.h 头文件提供了 DynamicLibrarySearch Generator 类来搜索库中的符号，我们已经在使用 LLJIT 实现我们自己的 JIT 编译器部分使用过。CompileUtils.h 头文件提供了 ConcurrentIRCompiler 类：

```
1 #include "llvm/Analysis/AliasAnalysis.h"
2 #include "llvm/ExecutionEngine/JITSymbol.h"
3 #include "llvm/ExecutionEngine/Orc/CompileUtils.h"
4 #include "llvm/ExecutionEngine/Orc/Core.h"
5 #include "llvm/ExecutionEngine/Orc/ExecutionUtils.h"
6 #include "llvm/ExecutionEngine/Orc/IRCompileLayer.h"
7 #include "llvm/ExecutionEngine/Orc/IRTransformLayer.h"
8 #include "llvm/ExecutionEngine/Orc/JITTargetMachineBuilder.h"
9 #include "llvm/ExecutionEngine/Orc/Mangling.h"
10 #include "llvm/ExecutionEngine/Orc/RTDyldObjectLinkingLayer.h"
11 #include "llvm/ExecutionEngine/Orc/TargetProcessControl.h"
12 #include "llvm/ExecutionEngine/SectionMemoryManager.h"
13 #include "llvm/Passes/PassBuilder.h"
14 #include "llvm/Support/Error.h"
```

3. 我们的新类是 JIT 类：

```
1 class JIT {
```

4. 私有数据成员为 ORC 层和一个助手类。ExecutionSession、ObjectLinkingLayer、CompileLayer、OptIRLayer 和 MainJITDylib 实例表示正在运行的 JIT 程序、各层和符号表。TargetProcessControl 实例用于与 JIT 目标流程进行交互，这可以是相同的进程，也可以是同一台机器上的另一个进程，或者是不同机器上的远程进程（可能具有不同的体系结构）。DataLayout 和 MangleAndInterner 类需要以正确的方式篡改符号名称。符号名是内化的，这意味着所有相等的名称都有相同的地址。为了检查两个符号名是否相等，比较地址就足够了，这是非常快速的操作：

```
1     std::unique_ptr<llvm::orc::TargetProcessControl>
2         TPC;
3     std::unique_ptr<llvm::orc::ExecutionSession> ES;
4     llvm::DataLayout DL;
5     llvm::orc::MangleAndInterner Mangle;
6     std::unique_ptr<llvm::orc::RTDyldObjectLinkingLayer>
7         ObjectLinkingLayer;
8     std::unique_ptr<llvm::orc::IRCompileLayer>
9         CompileLayer;
```

```

10 std::unique_ptr<llvm::orc::IRTransformLayer>
11     OptIRLayer;
12 llvm::orc::JITDylib &MainJITDylib;

```

5. 初始化分为三个部分。在 C++ 中，构造函数不能返回错误。一个简单且推荐的解决方案是创建一个静态工厂方法，该方法可以在构造对象之前进行错误处理。层的初始化更加复杂，因此我们也为它们引入了工厂方法。

在 create() 工厂方法中，首先创建一个 SymbolStringPool 实例，该实例用于实现字符串内部化，并由几个类共享。为了控制当前进程，创建了一个 SelfTargetProcessControlinstance。如果想要瞄准一个不同的流程，则需要更改这个实例。

然后，构造一个 JITTargetMachineBuilder 实例，为此需要知道 JIT 进程的目标三元组。接下来，我们向目标机器生成器查询数据布局，例如：如果构建器不能基于提供的三元组实例化目标机器，这个步骤可能会失败，因为对这个目标的支持没有编译到 LLVM 库中：

```

1 public:
2 static llvm::Expected<std::unique_ptr<JIT>> create() {
3     auto SSP =
4         std::make_shared<llvm::orc::SymbolStringPool>();
5     auto TPC =
6         llvm::orc::SelfTargetProcessControl::Create(SSP);
7     if (!TPC)
8         return TPC.takeError();
9     llvm::orc::JITTargetMachineBuilder JTMB(
10         (*TPC)->getTargetTriple());
11     auto DL = JTMB.getDefaultDataLayoutForTarget();
12     if (!DL)
13         return DL.takeError();

```

6. 至此，我们已经处理了所有可能失败的调用，现在能够初始化 ExecutionSession 实例。最后，使用所有实例化的对象调用 JIT 类的构造函数，并将结果返回给调用者：

```

1     auto ES =
2         std::make_unique<llvm::orc::ExecutionSession>(
3             std::move(SSP));
4
5     return std::make_unique<JIT>(
6         std::move(*TPC), std::move(ES),
7         std::move(*DL),
8         std::move(JTMB));
9 }

```

7. JIT 类的构造函数将传递的参数移动到私有数据成员，层对象是通过调用带有 create 前缀的

静态工厂名称来构造的。每个层工厂方法都需要对 ExecutionSession 实例的引用，将该层连接到正在运行的 JIT 会话。除了位于层栈底部的对象链接层外，每一层还需要参考上一层：

```
1  JIT(std::unique_ptr<llvm::orc::TargetProcessControl>
2      TPCtrl,
3      std::unique_ptr<llvm::orc::ExecutionSession> ExeS,
4      llvm::DataLayout DataL,
5      llvm::orc::JITTargetMachineBuilder JTMB)
6  : TPC(std::move(TPCtrl)), ES(std::move(ExeS)),
7      DL(std::move(DataL)), Mangler(*ES, DL),
8      ObjectLinkingLayer(std::move(
9          createObjectLinkingLayer(*ES, JTMB))),
10     CompileLayer(std::move(createCompileLayer(
11         *ES, *ObjectLinkingLayer,
12         std::move(JTMB)))),
13     OptIRLayer(std::move(
14         createOptIRLayer(*ES, *CompileLayer))),
15     MainJITDylib(ES->createBareJITDylib("<main>")) {
```

8. 构造函数的主体中，我们添加了一个生成器来搜索当前进程中的符号。GetForCurrentProcess() 方法是特殊的，因为返回值包装在一个 Expected<> 模板中，这表明也可以返回一个 Error 对象。但是我们知道不会发生错误，也就是当前进程最终会运行！因此，我们使用 cantFail() 函数展开结果，如果出现错误，该函数将终止应用程序：

```
1  MainJITDylib.addGenerator(llvm::cantFail(
2      llvm::orc::DynamicLibrarySearchGenerator::
3      GetForCurrentProcess(DL.getGlobalPrefix())));
4  }
```

9. 为了创建对象链接层，需要提供一个内存管理器。这里我们坚持使用默认的 SectionMemoryManager 类，但如果需要，也可以提供不同的实现：

```
1  static std::unique_ptr<
2      llvm::orc::RTDyldObjectLinkingLayer>
3  createObjectLinkingLayer(
4      llvm::orc::ExecutionSession &ES,
5      llvm::orc::JITTargetMachineBuilder &JTMB) {
6      auto GetMemoryManager = []() {
7          return std::make_unique<
8              llvm::SectionMemoryManager>();
9      };
10     auto OLLayer = std::make_unique<
11         llvm::orc::RTDyldObjectLinkingLayer>(<
```



```
12     ES, GetMemoryManager);
```

10. COFF 对象文件格式稍微有点复杂，它在 Windows 上使用。这种文件格式不允许将函数标记为导出。这将导致对象链接层内部的检查失败：存储在符号中的标志将与 IR 中的标志进行比较，这将会由于缺少导出标记而导致不匹配。解决方案是只覆盖该文件格式的标志。这样就完成了对象层的构造，并将对象返回给调用者：

```
1     if (JTMB.getTargetTriple().isOSBinFormatCOFF()) {
2         OLLayer
3             ->setOverrideObjectFlagsWithResponsibilityFlags(
4                 true);
5         OLLayer
6             ->setAutoClaimResponsibilityForObjectSymbols(
7                 true);
8     }
9     return std::move(OLLayer);
10 }
```

11. 要初始化编译器层，需要一个 IRCompiler 实例。IRCompiler 实例负责将 IR 模块编译成目标文件。如果 JIT 编译器不使用线程，那么可以使用 SimpleCompiler 类，它使用给定的目标机器来编译 IR 模块。TargetMachine 类不是线程安全的，SimpleCompiler 类也是如此。为了支持多线程编译，使用了 ConcurrentIRCompiler 类，它为每个要编译的模块创建了新的 TargetMachine 实例。这种方法解决了多线程问题：

```
1     static std::unique_ptr<llvm::orc::IRCompileLayer>
2     createCompileLayer(
3         llvm::orc::ExecutionSession &ES,
4         llvm::orc::RTDyldObjectLinkingLayer &OLLayer,
5         llvm::orc::JITTargetMachineBuilder JTMB) {
6         auto IRCompiler = std::make_unique<
7             llvm::orc::ConcurrentIRCompiler>(
8             std::move(JTMB));
9         auto IRCLayer =
10             std::make_unique<llvm::orc::IRCompileLayer>(
11                 ES, OLLayer, std::move(IRCompiler));
12         return std::move(IRCLayer);
13     }
```

12. 我们不是直接将 IR 模块编译成机器码，而是首先安装一个优化 IR 的层。这是一个深思熟虑的设计决策：将 JIT 编译器转换为优化 JIT 编译器，这将产生更快的代码，但需要更长的时间来生成，这对用户来说意味着等待。我们没有添加惰性编译，因此当只查找一个符号时，会编译整个模块。在用户看到代码执行之前，这可能需要很长时间。

Note

请注意，引入惰性编译并不是所有情况下的正确解决方案。

惰性编译是通过将每个函数移动到它自己的模块中来实现的，该模块在查找函数名时进行编译。这阻止了内联等过程间优化，因为内联传递需要访问为内联而调用的函数体。因此，使用惰性编译，用户会看到更快的启动速度，但生成的代码并不是最优的。这些设计决策取决于预期的用途，我们决定使用快速代码，接受较慢的启动时间。优化层作为转换层实现。IRTransformLayer 类将转换委托给一个函数，我们的例子中，会委托给 optimizeModule 函数：

```
1  static std::unique_ptr<llvm::orc::IRTransformLayer>
2  createOptIRLayer(
3      llvm::orc::ExecutionSession &ES,
4      llvm::orc::IRCompileLayer &CompileLayer) {
5      auto OptIRLayer =
6          std::make_unique<llvm::orc::IRTransformLayer>(
7              ES, CompileLayer,
8              optimizeModule);
9      return std::move(OptIRLayer);
10 }
```

13. optimizeModule() 函数是 IR 模块上的一个转换示例。该函数获取要转换的模块作为参数，并返回转换后的模块。因为 JIT 可以使用多个线程运行，IR 模块包装在 ThreadSafeModule 实例中：

```
1  static llvm::Expected<llvm::orc::ThreadSafeModule>
2  optimizeModule(
3      llvm::orc::ThreadSafeModule TSM,
4      const llvm::orc::MaterializationResponsibility
5          &R) {
```

14. 为了优化 IR，我们回顾第 8 章中的一些信息，在向编译器添加优化管道一节。我们需要一个 PassBuilder 实例来创建一个优化流水。首先，我们定义几个分析管理器，然后在 Pass 构建器中注册它们。然后，使用 O2 级别的默认优化流水填充 ModulePassManager 实例。这又是一个设计决策：O2 已经产生了快速的机器码，甚至比 O3 更快。之后，我们在模块上运行管道。最后，优化后的模块返回给调用者：

```
1  TSM.withModuleDo([](llvm::Module &M) {
2      bool DebugPM = false;
3      llvm::PassBuilder PB(DebugPM);
4      llvm::LoopAnalysisManager LAM(DebugPM);
5      llvm::FunctionAnalysisManager FAM(DebugPM);
6      llvm::CGSCCAnalysisManager CGAM(DebugPM);
```


将 JIT 编译器组合在一起非常容易。初始化类有点棘手，因为它涉及 JIT 类的工厂方法和构造函数调用，以及每一层的工厂方法。尽管代码本身很简单，但其是因为 C++ 的限制造成的。

下一小节中，我们将使用新的 JIT 编译器类来实现一个命令行实用程序。

使用新的 JIT 编译器类

我们新的 JIT 编译器类的接口，类似于 (用 LLJIT 实现我们自己的 JIT 编译器部分中使用的)LLJIT 类。为了测试我们的实现，可以从上一节复制 LIT.cpp 中的类，并进行以下更改：

1. 为了能够使用我们的新类，我们包含了 JIT.h 头文件。这将替换 llvm/ExecutionEngine/Orc/LLJIT.h 头文件，因为这里不再使用 LLJIT 类，所以不再需要这个头文件。
2. 在 jitmain() 函数中，用一个新的 JIT::create() 方法来替换对 orc::LLJITBuilder().create() 的调用。
3. 同样，在 jitmain() 函数中，删除了添加 DynamicLibrarySearchGenerator 类的代码。确切地说，这个生成器集成在 JIT 类中。

我们可以像上一节一样编译和运行更改后的应用程序，结果是相同的。在内部，新类使用了固定的优化级别，因此对于足够大的模块，可以注意到启动和运行时的差异。

手边有一个 JIT 编译器时，就可以随时验证新的想法。在下一节中，我们将研究如何使用 JIT 编译器作为静态编译器的一部分，在编译时计算代码。

利用 JIT 编译器进行代码计算

编译器编写者为了生成最优代码付出了巨大的努力。一种简单而有效的优化方法是用运算的结果值替换对两个常量的算术运算。为了能够执行计算，嵌入了常量表达式的解释器。为了得到相同的结果，解释器必须实现与生成的机器代码相同的规则！当然，这可能是出错的原因。

另一种方法是使用相同的代码生成方法将常量表达式编译到 IR，然后让 JIT 编译并执行 IR。这个想法甚至可以更进一步。数学中对于相同的输入，函数总是产生相同的结果。对于计算机语言中的函数，这就不一定了。一个例子是 rand() 函数，它为每次调用返回一个随机值。计算机语言中的函数与数学中的函数具有相同的特性，称为纯函数。在优化表达式期间，我们可以 JIT 编译和执行纯函数 (只有常量参数)，并使用 JIT 执行返回的结果替换对函数的调用。实际上，我们将函数的执行从运行时移动到编译时！

交叉编译

使用 JIT 编译器作为静态编译器的一部分是一个有趣的选择。但是，如果编译器要支持交叉编译，那么就应该仔细考虑这种方法。引起麻烦的常见候选者是浮点类型，C 语言中的长双精度通常取决于硬件和操作系统。有些系统使用 128 位浮点数，而有些系统只使用 64 位浮点数。80 位浮点类型只在 x86 平台上可用，通常只在 Windows 上使用。以不同的精度执行相同的浮点运算可能会导致巨大的差异。在这种情况下不能使用通过 JIT 编译进行的计算。

一个函数纯不纯不容易判定，常见的解决方案是应用启发式。如果函数不通过指针或间接使用聚合类型读取或写入堆内存，并且只调用其他纯函数，那么它就是一个纯函数。开发人员可以帮助编译器标记纯函数，例如：用特殊的关键字或符号。在语义分析阶段，编译器可以检查是否存在违

规。

在下一小节中，我们将进一步研究在编译时尝试 JIT 执行函数时对语义的影响。

识别语义

困难的部分是在语义级别，需要决定语言的哪些部分适合在编译时计算。排除对堆内存的访问非常严格，它排除了字符串处理。当分配的内存存在 JIT 执行函数的生命周期内仍然存在时，使用堆内存就会出现这个问题。这是一种程序状态，可能会影响其他结果。另一方面，如果有对 `malloc()` 和 `free()` 函数的匹配调用，那么内存只用于内部计算。在这种情况下，使用堆内存是安全的。但确切地说，这并不容易证明。

在类似的级别上，JIT 执行函数内部的无限循环可能会冻结编译器。1936 年，艾伦·图灵 (Alan Turing) 证明，没有机器可以决定一个函数是否会产生结果，或者是否陷入了无休止的循环。必须采取一些预防措施来避免这种情况，例如：在运行时限制之后，JIT 执行的函数需要停止。

最后，允许的功能越多，就必须更多地考虑安全性，因为编译器现在执行别人编写的代码。想象一下，这些代码从互联网上下载并运行文件，或者试图擦除硬盘：对于 JIT 执行的函数，允许的状态太多了，因此我们也需要考虑这样的场景。

这个想法并不新。D 编程语言有一个叫做编译时执行函数的特性。引用编译器 `dmd` 通过在 AST 级别解释函数来实现这个特性。基于 LLVM 的 LDC 编译器实验性的特性，可以为它使用 LLVM JIT 引擎。可以在<https://dlang.org/>找到更多关于语言和编译器的信息。

忽略语义挑战，实现并不是那么困难。在从零开始构建 JIT 编译器类的章节中，我们使用 JIT 类开发了一个 JIT 编译器。在类中输入一个 IR 模块，然后可以从这个模块查找并执行函数。看看 `tinylang` 编译器的实现，我们可以清楚地识别对常量的访问，因为 AST 中有一个 `ConstantAccess` 节点。例如：

```
1 if (auto *Const = llvm::dyn_cast<ConstantAccess>(Expr)) {  
2     // Do something with the constant.  
3 }
```

不需要解释表达式中的运算来推导常量的值，我们可以做以下操作：

1. 创建一个新的 IR 模块。
2. 在模块中创建一个 IR 函数，返回一个预期类型的值。
3. 使用现有的 `emitExpr()` 函数为表达式创建 IR，并使用最后一条指令返回计算值。
4. JIT 执行函数来计算值。

这是否值得实现？作为优化流水的一部分，LLVM 执行常量传播和函数内联。像 `4+5` 这样的简单表达式在构建 IR 时已经替换了，像计算最大公约数这样的小函数是内联的。如果所有参数都是常量值，那么内联代码将通过常量传播。

根据这一观察，只有在有足够的语言特性可在编译时执行时，这种方法的实现才有用。如果是这样，那么使用给定的示意图就相当容易实现了。

了解如何使用 LLVM 的 JIT 编译器组件使您能够以全新的方式使用 LLVM。除了实现像 Java VM 这样的 JIT 编译器外，JIT 编译器还可以嵌入到其他应用程序中。这允许使用创造性的方法，例如：在静态编译器中使用它（这一节中已经介绍过了）。

总结

本章中，您学习了如何开发 JIT 编译器。从 JIT 编译器应用开始，探索了 LLVM 动态编译器和解释器 `lli`。使用预定义的 `LLJIT` 类，您可以自己构建一个类似于 `lli` 的工具。为了能够利用 ORC API 的分层结构，实现了一个优化 JIT 类。掌握了所有这些知识后，了解了在静态编译器中使用 JIT 编译器的可能性，有一些语言可以从中受益。

下一章中，您将了解如何为 LLVM 添加一个新 CPU 体系架构的后端。

第 11 章 使用 LLVM 工具调试

LLVM 附带了一组工具，可以帮助您查找应用程序中的某些错误。所有这些工具都会使用 LLVM 和 Clang 的库。

本章中，您将学习如何使用 sanitizer 来检测应用程序，如何使用最常见的 sanitizer 来识别各种各样的错误，以及如何为您的应用程序实现模糊测试 (fuzz testing)。这将帮助您识别单元测试通常无法发现的 bug。您还将学习如何在应用程序中识别性能瓶颈，运行静态分析程序来识别编译器通常没有发现的问题，以及创建您自己的 (基于 Clang) 的工具，您可以使用它来扩展 Clang 的新功能。

本章将涵盖以下内容：

- 使用 sanitizer 检查应用程序
- 用 libFuzzer 找到 bug
- 使用 xRay 进行性能分析
- 使用 Clang 静态分析器检查源代码
- 创建基于 Clang 的工具

本章结束时，您将了解如何使用各种 LLVM 和 Clang 工具来识别应用程序中的错误。您还将获得使用新功能扩展 Clang 的知识，例如：执行命名约定或添加新的源码分析工具。

相关代码

要在使用 XRay 的性能分析部分创建火焰图，需要从<https://github.com/brendangregg/FlameGraph>安装脚本。有些系统，如 Fedora 和 FreeBSD，为这些脚本提供了安装包，可以直接使用安装包安装。

要查看 Chrome 可视化，需要安装 Chrome 浏览器。可以从<https://www.google.com/chrome/>下载浏览器，或者使用系统的包管理器来安装 Chrome 浏览器。本章的代码文件可以在<https://github.com/PacktPublishing/Learn-LLVM-12/tree/master/Chapter11>获取。

可以在视频中找到代码<https://bit.ly/3nllhED>。

使用 sanitizer 安装应用程序

LLVM 自带一些 sanitizer。这些 Pass 以某种方式检测中间表示 (IR)，以检查应用程序的某些不当行为。通常，需要库支持，这是 compiler-rt 项目的一部分。可以在 Clang 中启用 sanitizer，这让它们使用起来更加方便。下面的小节中，我们将介绍可用的 sanitizer，即地址、内存和线程。我们先来看看地址 sanitizer。

用地址 sanitizer 检测内存访问问题

可以使用地址 sanitizer 来检测应用程序中的两个内存访问错误。这包括一些常见的错误，比如：在释放动态分配的内存后使用它，或者在已分配内存的边界之外写入动态分配的内存。

当启用地址 sanitizer 时，地址 sanitizer 将用它自己的版本替换对 malloc() 和 free() 函数的调用，并使用检查保护程序检测所有内存访问。当然，这给应用程序增加了很多开销，您将只在应用程序的测试阶段使用地址消毒剂。如果对实现细节感兴趣，可以在 `llvm/lib/Transforms/Instrumentation/`

AddressSanitizer.cpp 文件中找到 Pass 源，并在<https://github.com/google/sanitizers/wiki/AddressSanitizerAlgorithm>中找到算法描述。

让我们运行一个简短的示例来演示地址 sanitizer 的功能。下面的示例应用程序 outofbounds.c 分配了 12 字节的内存，但初始化了 14 字节：

```
1 #include <stdlib.h>
2 #include <string.h>
3
4 int main(int argc, char *argv[]) {
5     char *p = malloc(12);
6     memset(p, 0, 14);
7     return (int)*p;
8 }
```

您可以编译并运行此应用程序，而不会注意到任何问题。这是这中错误的典型特征。即使在较大的应用程序中，这种错误也可能在很长一段时间内不被注意。但是，如果使用-fsanitize=address 选项启用地址 sanitizer，那么应用程序在检测到错误后将停止。

使用-g 选项启用调试符号也很有用，因为它有助于识别源文件中错误的位置。下面的代码示例说明了如何在启用地址消毒剂和调试符号的情况下编译源文件：

```
$ clang -fsanitize=address -g outofbounds.c -o outofbounds
```

现在，当运行应用程序时，会得到一个冗长的错误报告：

```
$ ./outofbounds
=====
===
==1067==ERROR: AddressSanitizer: heap-buffer-overflow on
address 0x60200000001c at pc 0x00000023a6ef bp 0x7ffffffeb10
sp 0x7ffffffe2d8
WRITE of size 14 at 0x60200000001c thread T0
    #0 0x23a6ee in _asan_memset /usr/src/contrib/llvm-project/
compiler-rt/lib/asan/asan_interceptors_memintrinsics.cpp:26:3
    #1 0x2b2a03 in main /home/kai/sanitizers/outofbounds.c:6:3
    #2 0x23331f in _start /usr/src/lib/csu/amd64/crt1.c:76:7
```

报告还包含关于内存内容的详细信息。重要的信息是错误的类型（在本例中是堆缓冲区溢出）和出错的源行。要找到源码行，可以查看位置 #1 的堆栈跟踪，这是地址 sanitizer 拦截应用程序执行之前的最后一个位置。它显示了 outofbounds.c 文件中的第 6 行，这一行包含了对 memset() 的调用——实际上，这就是发生内存溢出的位置。

替换包含 memset(p, 0, 14); 在 outofbounds.c 文件中使用以下代码，然后在释放内存后引入对

内存的访问。并将源代码存储在 useafterfree.c 中:

```
1 memset(p, 0, 12);
2 free(p);
```

同样, 如果编译并运行, 会检测到内存释放后指针的使用情况:

```
$ clang -fsanitize=address -g useafterfree.c -o useafterfree
$ ./useafterfree

=====
===
==1118==ERROR: AddressSanitizer: heap-use-after-free on address
0x602000000010 at pc 0x0000002b2a5c bp 0x7ffffffeb00 sp
0x7ffffffeaf8
READ of size 1 at 0x602000000010 thread T0
    #0 0x2b2a5b in main /home/kai/sanitizers/
useafterfree.c:8:15
    #1 0x23331f in _start /usr/src/lib/csu/amd64/crt1.c:76:7
```

这一次, 报告指向第 8 行, 其中包含 p 指针的释放。

在 x86_64 Linux 和 macOS 上, 也可以启用泄漏检测器。如果在运行应用程序之前将 ASAN_OPTIONS 环境变量设置为 detect_leaks=1, 还会得到一个关于内存泄漏的报告。可以这样做:

```
$ ASAN_OPTIONS=detect_leaks=1 ./useafterfree
```

地址 sanitizer 非常有用, 因为它捕获了一类用其他方法很难检测到的 bug。内存 sanitizer 执行类似的任务, 我们将在下一节中研究它。

使用内存 sanitizer 查找未初始化的内存访问

使用未初始化的内存是另一类难以发现的错误。在 C 和 C++ 中, 一般的内存分配例程不会用默认值初始化内存缓冲区, 对于堆栈上的变量也是如此。

出现错误的机会很多, 而内存 sanitizer 有助于找到错误。如果对实现细节感兴趣, 可以在 llvm/lib/Transforms/Instrumentation/MemorySanitizer.cpp 中找到内存 sanitizer Pass 的源文件。文件顶部的注释解释了实现思想。

让我们运行一个小示例, 并将下面的源代码保存为 memory.c 文件。你应该注意到 x 变量没有初始化, 而是用作返回值:

```
1 int main(int argc, char *argv[]) {
2     int x;
3     return x;
4 }
```

没有 sanitizer，应用程序将运行得很好。如果你使用 `-fsanitize=memory` 选项，就会得到一个错误报告：

```
$ clang -fsanitize=memory -g memory.c -o memory
$ ./memory
==1206==WARNING: MemorySanitizer: use-of-uninitialized-value
    #0 0x10a8f49 in main /home/kai/sanitizers/memory.c:3:3
    #1 0x1053481 in _start /usr/src/lib/csu/amd64/crt1.c:76:7

SUMMARY: MemorySanitizer: use-of-uninitialized-value /home/kai/
sanitizers/memory.c:3:3 in main
Exiting
```

与地址 sanitizer 一样，内存 sanitizer 在发现第一个错误时停止应用程序。

下一节中，我们将了解如何使用线程 sanitizer 来检测多线程应用程序中的数据竞争。

用线程 sanitizer 指出数据竞争

为了充分利用现代 CPU 的功能，应用程序现在使用多线程。这是一项强大的技术，但它也引入了新的错误来源。多线程应用程序中一个常见的问题是，对全局数据的访问没有保护，例如：没有使用互斥锁或信号量。这样的问题称为数据竞争。线程 sanitizer 可以检测基于 pthread 的应用程序和使用 LLVM libc++ 实现的应用程序中的数据竞争。可以在 `llvm/lib/Transforms/Instrumentation/ThreadSanitize.cpp` 文件中找到实现。

为了演示线程 sanitizer 的功能，我们将创建一个简单的生产者/消费者的应用程序。生产者线程增加一个全局变量，而消费者线程减少同一个变量。对全局变量的访问不受保护，因此这显然是一场数据竞争。在 `thread.c` 文件中保存以下源代码：

```
1 #include <pthread.h>
2
3 int data = 0;
4
5 void *producer(void *x) {
6     for (int i = 0; i < 10000; ++i) ++data;
7     return x;
8 }
9
10 void *consumer(void *x) {
11     for (int i = 0; i < 10000; ++i) --data;
12     return x;
13 }
14
15 int main() {
```

```
16 pthread_t t1, t2;
17 pthread_create(&t1, NULL, producer, NULL);
18 pthread_create(&t2, NULL, consumer, NULL);
19 pthread_join(t1, NULL);
20 pthread_join(t2, NULL);
21 return data;
22 }
```

前面的代码中，数据变量在两个线程之间共享。这里，它是 `int` 类型，以使示例简单。最常见的情况是使用 `std::vector` 类或类似的数据结构。这两个线程运行 `producer()` 和 `consumer()` 函数。

`producer()` 函数只增加数据变量，而 `consumer()` 函数减少数据变量。没有实现访问保护，因此这构成了一场数据竞争。`main()` 函数使用 `pthread_create()` 函数启动两个线程，使用 `pthread_join()` 函数等待线程结束，并返回数据变量的当前值。

如果编译并运行此应用程序，则不会注意到任何错误，返回值总是 0。如果执行的循环次数增加 100 倍，则会出现一个错误。本例中，返回值不等于 0，您将看到显示其他值。

您可以使用线程 sanitizer 来标识数据竞争。要在启用了线程 sanitizer 的情况下进行编译，需要将 `-fsanitize=thread` 选项传递给 Clang。使用 `-g` 选项添加调试符号可以在报告中提供行号，这很有帮助。注意，还需要链接 `pthread` 库：

```

$ clang -fsanitize=thread -g thread.c -o thread -lpthread
$ ./thread
=====
WARNING: ThreadSanitizer: data race (pid=1474)
  Write of size 4 at 0x000000cdf8f8 by thread T2:
    #0 consumer /home/kai/sanitizers/thread.c:11:35 (thread+0x2b0fb2)

  Previous write of size 4 at 0x000000cdf8f8 by thread T1:
    #0 producer /home/kai/sanitizers/thread.c:6:35 (thread+0x2b0f22)
  Location is global 'data' of size 4 at 0x000000cdf8f8 (thread+0x000000cdf8f8)
  Thread T2 (tid=100437, running) created by main thread at:
    #0 pthread_create /usr/src/contrib/llvm-project/
  compiler-rt/lib/tsan/rtl/tsan_interceptors_posix.cpp:962:3 (thread+0x271703)
    #1 main /home/kai/sanitizers/thread.c:18:3 (thread+0x2b1040)

  Thread T1 (tid=100436, finished) created by main thread at:
    #0 pthread_create /usr/src/contrib/llvm-project/
  compiler-rt/lib/tsan/rtl/tsan_interceptors_posix.cpp:962:3 (thread+0x271703)
    #1 main /home/kai/sanitizers/thread.c:17:3 (thread+0x2b1021)

SUMMARY: ThreadSanitizer: data race /home/kai/sanitizers/ thread.c:11:35 in consumer
=====
ThreadSanitizer: reported 1 warnings

```

报告指向源文件的第 6 行和第 11 行，其中访问全局变量。它还显示了两个名为 T1 和 T2 的线程访问了该变量，以及分别调用 `pthread_create()` 函数的文件和行号。

本节中，我们学习了如何使用三种 sanitizer 来查找应用程序中的常见问题。地址 sanitizer 帮助我们识别常见的内存访问错误，例如：越界访问或在释放后使用内存。使用内存 sanitizer，可以找到对未初始化内存的访问，线程 sanitizer 帮助我们查找数据竞争。

下一节中，我们将尝试通过在随机数据上运行应用程序（称为模糊测试）来触发 sanitizer。

用 libFuzzer 找 Bug

要测试应用程序，需要编写单元测试。这是确保软件正常运行的好方法。然而，可能的输入呈指数级增长，很可能会错过某些奇怪的输入，以及一些 bug。

模糊测试可以在这里提供帮助。其思想是为应用程序提供随机生成的数据，或基于有效输入但随机更改的数据。这是一遍又一遍地进行的，因此您的应用程序将使用大量输入进行测试。这是一种非常强大的测试方法。通过模糊测试，发现了网络浏览器和其他软件中的数百个漏洞。

LLVM 自带了自己的模糊测试库。libFuzzer 实现最初是 LLVM 核心库的一部分，后来移到了

compiler-rt 上。该库旨在测试小而快速的函数。

让我们运行一个小示例。需要提供 LLVMFuzzerTestOneInput() 函数。这个函数由 fuzzer 驱动程序调用，并为您提供一些输入。下面的函数计算输入中的连续 ASCII 数字，然后将随机输入输入给它。需要将示例保存在 fuzzer.c 文件中：

```
1 #include <stdint.h>
2 #include <stdlib.h>
3
4 int count(const uint8_t *Data, size_t Size) {
5     int cnt = 0;
6     if (Size)
7         while (Data[cnt] >= '0' && Data[cnt] <= '9') ++cnt;
8     return cnt;
9 }
10
11 int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t
12                             Size) {
13     count(Data, Size);
14     return 0;
15 }
```

代码中，count() 函数计算 Data 变量所指向的内存中的位数。只检查数据的大小以确定是否有可用的字节。在 while 循环中，不检查数据长度。

与普通 C 字符串一起使用时，不会出现错误，因为 C 字符串总是以 0 字节结束。LLVMFuzzerTestOneInput() 函数就是 fuzz 目标，它是 libFuzzer 调用的函数。调用我们想要测试的函数并返回 0，这是目前唯一允许的值。

要使用 libFuzzer 编译文件，需要添加-fsanitize=fuzzer 选项。建议还启用地址 sanitizer 和调试符号的生成：

```
$ clang -fsanitize=fuzzer,address -g fuzzer.c -o fuzzer
```

当您运行测试时，会产生一个冗长的报告。报告包含比堆栈跟踪更多的信息，所以让我们仔细来看看：

1. 第一行告诉您用于初始化随机数生成器的种子。可以使用 -seed= 来重复执行：

```
INFO: Seed: 1297394926
```

2. 默认情况下，libFuzzer 将输入限制为最多 4,096 字节。可以使用 -max_len= 来更改默认值：

```
INFO: -max_len is not provided; libFuzzer will not
generate inputs larger than 4096 bytes
```

3. 现在，在不提供示例输入的情况下运行测试。所有样本输入的集合称为语料库，在这次运行中它是空的：

```
INFO: A corpus is not provided, starting from an empty corpus
```

4. 下面是一些关于生成的测试数据的信息。尝试了 28 个输入，找到了 6 个输入，总长度为 19 字节，总共覆盖了 6 个点或基本块：

```
#28 NEW cov: 6 ft: 9 corp: 6/19b lim: 4 exec/s: 0  
rss: 29Mb L: 4/4 MS: 4 CopyPart-PersAutoDict-CopyPart-  
ChangeByte- DE: "1\x00"-
```

5. 之后，检测到内存溢出，然后就是地址 sanitizer 的信息。最后，报告显示导致内存溢出的输入在哪里：

```
artifact_prefix='./'; Test unit written to ./crash-17ba0791499db908433b80f37c5fbc89b87  
0084b
```

保存了输入，就可以用崩溃的输入再次执行测试用例：

```
$ ./fuzzer crash-17ba0791499db908433b80f37c5fbc89b870084b
```

这显然对识别问题有很大帮助。只是，使用随机数据通常不是很有帮助。如果您尝试对 `tinylang` 词法分析器或解析器进行模糊测试，因为无法找到有效的标记，所以纯随机数据将导致输入立即被拒绝。

这种情况下，提供一小组称为语料库的有效输入更有用。然后，对语料库中的文件进行随机变异并作为输入。您可以认为输入基本上是有效的，只翻转了几个位。这也适用于其他必须具有特定格式的输入，例如：对于处理 JPEG 和 PNG 文件的库，可以将提供一些小的 JPEG 和 PNG 文件作为语料库。

您可以将语料库文件保存在一个或多个目录中，并可以在 `printf` 命令的帮助下为 fuzz 测试创建一个简单的语料库：

```
$ mkdir corpus  
$ printf "012345\0" >corpus/12345.txt  
$ printf "987\0" >corpus/987.txt
```

当运行测试时，可以在命令行上提供语料库目录：

```
$ ./fuzzer corpus/
```

然后使用语料库作为生成随机输入的基础，如报告所示：

INFO: seed corpus: files: 2 min: 4b max: 7b total: 11b rss: 29Mb

如果您正在测试一个作用于令牌或其他魔数值 (如编程语言) 的函数, 那么可以通过提供一个带有令牌的字典来加快这个过程。对于编程语言, 字典将包含该语言中使用的所有关键字和特殊符号。字典定义遵循简单的键-值样式, 例如: 要在字典中定义 if 关键字, 可以添加以下内容:

```
kw1="if"
```

但是, 这个键是可选的, 可以省略。然后可以使用 `-dict=` 在命令行上指定字典文件。下一节中, 我们将了解 libFuzzer 实现的限制和替代方案。

限制和替代

libFuzzer 实现速度很快, 但对测试目标有许多限制:

- 测试函数必须在内存中以数组的形式接受输入。有些库函数需要数据的文件路径, 不能用 libFuzzer 测试它们。
- 不应该调用 `exit()` 函数。
- 全局状态不应改变。
- 不应该使用硬件随机数生成器。

从前面提到的限制中可以看出, 前两个限制是 libFuzzer 作为库实现的一个含义。在后两个限制是必需的, 以避免在评估算法中的混淆。如果没有满足其中一个限制, 那么对模糊目标的两次相同调用可能会得到不同的结果。

最著名的模糊测试替代工具是 AFL, 可以在<https://github.com/google/AFL>找到。AFL 需要一个检测的二进制文件 (提供了一个用于检测的 LLVM 插件), 并要求应用程序将输入作为命令行上的文件路径。AFL 和 libFuzzer 可以共享相同的语料库和字典文件。因此, 可以使用这两种工具来测试应用程序。在 libFuzzer 不适用的情况下, AFL 是一个很好的替代方案。

有很多方法可以影响 libFuzzer 的工作方式, 可以在<https://llvm.org/docs/LibFuzzer.html>了解更多细节。

下一节中, 我们将讨论应用程序可能存在的一个完全不同的问题, 试图查找性能瓶颈。

使用 XRay 进行性能分析

如果应用程序运行缓慢, 那么很可能想知道代码中的时间都花费在什么地方了。在这种情况下, 使用 XRay 检测代码会有所帮助。基本上, 在每个函数的入口和出口, 都会插入一个对运行时库的特殊调用。这允许计算函数被调用的频率, 以及在函数中花费的时间。可以在 `llvm/lib/XRay/` 目录中找到 Pass 的实现, 运行时是 `compiler-rt` 的一部分。

下面的示例源代码中, 通过调用 `usleep()` 函数来模拟实际工作。函数的作用是: 休眠 10s。func2() 函数要么调用 func1(), 要么休眠 100s, 这取决于 n 参数是奇数还是偶数。在 main() 函数中, 两个函数都是在循环中调用的。在 `xraydemo.c` 文件中保存以下代码:

```
1 #include <unistd.h>
2
```

```

3 void func1() { usleep(10); }
4
5 void func2(int n) {
6     if (n % 2) func1();
7     else usleep(100);
8 }
9
10 int main(int argc, char *argv[]) {
11     for (int i = 0; i < 100; i++) { func1(); func2(i); }
12     return 0;
13 }

```

要在编译期间启用 XRay 检测，需要指定 `-fxray-instrument` 选项，不测试小于 200 条指令的函数。这是一个由开发人员定义的任意阈值，在例子中，函数不会被检测。该阈值可以通过 `-fxrayinstruction-threshold=` 指定，或者可以添加一个 `function` 属性来控制函数是否应该检测，例如：添加以下修饰会让函数始终执行检测：

```

1 void func1() __attribute__((xray_always_instrument));

```

同样，通过使用 `xray_never_instrument` 属性，可以关闭函数的检测功能。
现在，将使用命令行选项编译 `xraydemo.c` 文件：

```

$ clang -fxray-instrument -fxray-instruction-threshold=1 -g \
    xraydemo.c -o xraydemo

```

现在，将使用命令行选项编译 `xraydemo.c` 文件在生成的二进制文件中，检测功能默认关闭。如果运行该二进制文件，您将不会注意到与未插装的二进制文件的区别。`XRAY_OPTIONS` 环境变量用于控制运行时数据的记录。要启用数据收集，运行应用程序如下所示：

```

$ XRAY_OPTIONS="patch_premain=true xray_mode=xray-basic "\
./xraydemo

```

`xray_mode=xray-basic` 选项告诉运行时我们想要使用基本模式。在这种模式下，将收集所有的运行时数据，这可能生成巨大的日志文件。当给出 `patch_premain=true` 选项时，那么在 `main()` 函数之前运行的函数也会被检测。

执行该命令后，将在收集的数据所在的目录中看到一个新文件，需要使用 `llvm-xray` 工具从这个文件中提取可读的信息。

`llvm-xray` 工具支持各种子命令。可以使用 `account` 子命令提取一些基本统计信息，例如：要获得被调用最多的前 10 个函数，可以添加 `-top=10` 选项来限制输出，并添加 `-sort=count` 选项来指定函数调用计数作为排序标准。可以使用 `-sortorder=` 选项影响排序顺序。执行如下命令获取统计信息：


```
$ llvm-xray account xray-log.xraydemo.xVsWiE -sort=count\  
-sortorder=dsc -instr_map ./xraydemo
```

Functions with latencies: 3

funcid	count	sum	function
1	150	0.166002	demo.c:4:0: func1
2	100	0.543103	demo.c:9:0: func2
3	1	0.655643	demo.c:17:0: main

可以看到，func1() 函数调用得最频繁，以及在这个函数中花费的累计时间。这个示例只有三个函数，所以-top= 在这里没有明显的效果，但对于实际应用程序来说，它非常有用。

从收集的数据中，可以重构运行时发生的所有堆栈帧。可以使用 stack 子命令查看排名前 10 的堆叠。为了简洁起见，此处显示的输出简化了：

```
$ llvm-xray stack xray-log.xraydemo.xVsWiE -instr_map\  
./xraydemo
```

Unique Stacks: 3

Top 10 Stacks by leaf sum:

Sum: 1325516912

lvl	function	count	sum
#0	main	1	1777862705
#1	func2	50	1325516912

Top 10 Stacks by leaf count:

Count: 100

lvl	function	count	sum
#0	main	1	1777862705
#1	func1	100	303596276

堆栈帧是一个函数调用的序列。func2() 函数由 main() 函数调用，这是累积耗时最长的堆栈帧。深度取决于调用多少函数，堆栈帧通常很大。

这个子命令还可以用于从堆栈帧创建火焰图，可以很容易地识别哪些函数具有较大的累积运行时。输出是包含计数和运行时信息的堆栈帧。使用 flamegraph.pl 脚本，可以将数据转换为可缩放矢量图形 (Scalable Vector Graphics, SVG) 文件，并且可以在浏览器中查看该文件。

使用下面的命令，可以指示 llvm-xray 使用-all-stacks 选项输出所有堆栈帧。使用-stack-format=flame 选项，输出将以 flamegraph.pl 脚本所期望的格式显示。使用-aggregationtype 选项，可以选择是按总时间还是按调用次数聚合堆栈帧。llvm-xray 的输出通过管道传输到 flamegraph.pl 脚本中，结果输出保存在 flame.svg 文件中：

```
$ llvm-xray stack xray-log.xraydemo.xVsWiE -all-stacks\  
-stack-format=flame --aggregation-type=time\  
-instr_map ./xraydemo | flamegraph.pl >flame.svg
```

在浏览器中打开生成的 flame.svg 文件。图表如下所示:

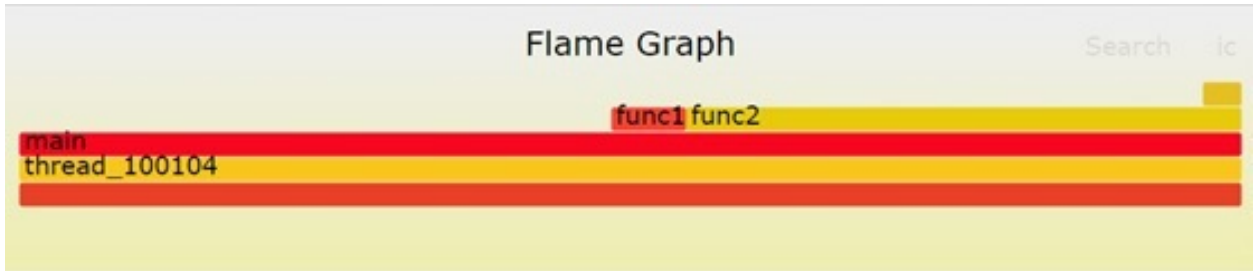


图 11.1 - 由 llvm-x 射线生成的火焰图

火焰图第一眼看起来可能会令人困惑，因为 x 轴没有通常的流逝时间的含义。相反，函数只是简单地按名称排序。颜色的选择要有良好的对比，没有其他意义。从上面的图中，可以很容易地确定调用层次结构和在函数中花费的时间。

只有将鼠标移到表示堆栈帧的矩形上，才会显示堆栈帧的相关信息。用鼠标单击框架，可以放大此堆栈框架。如果想要确定值得优化的函数，火焰图是很有帮助的。想了解更多关于火焰图的信息，请访问火焰图的作者 Brendan Gregg 的网站，<http://www.brendangregg.com/flamegraphs.html>。

您可以使用 convert 子命令将数据转换为.yaml 格式或 Chrome 跟踪查看器可视化使用的格式。后者是另一种从数据创建图形的好方法。将数据保存在 xray.evt 文件中:

```
$ llvm-xray convert -output-format=trace_event\  
-output=xray.evt -symbolize -sort\  
-instr_map=./xraydemo xray-log.xraydemo.xVsWiE
```

如果不指定-symbolic 选项，则结果图中不会显示函数名。

一旦完成，打开 Chrome 浏览器，输入 Chrome:///跟踪。然后，点击加载按钮来加载 xray.evt 文件。您将看到以下可视化数据:

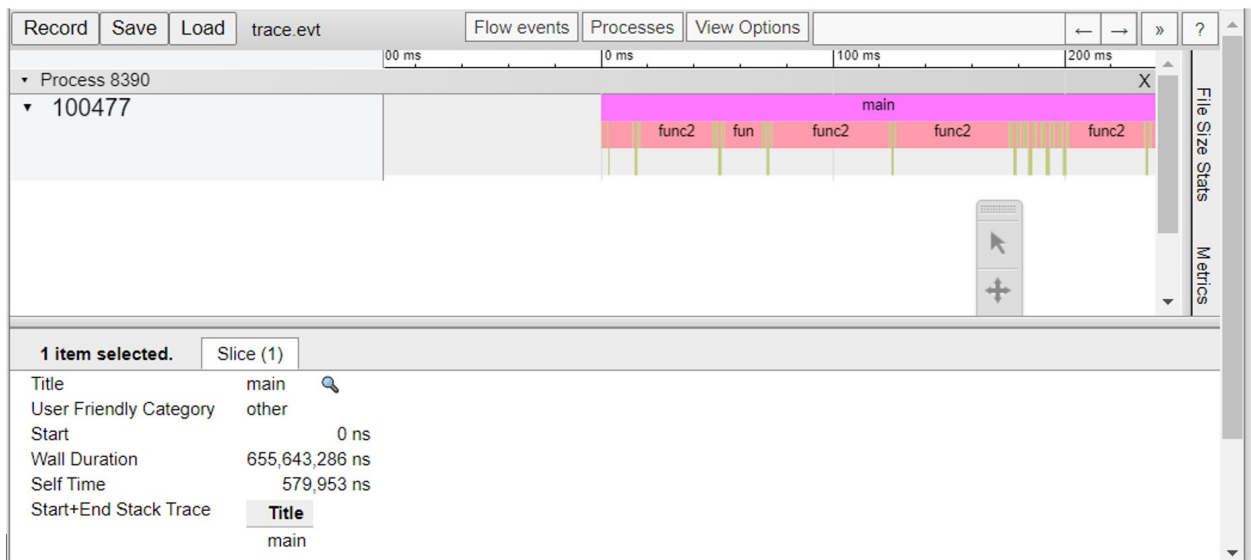


图 11.2 -Chrome 跟踪查看器可视化生成的 llvm-xray

在这个视图中，堆栈帧按函数调用发生的时间排序。关于可视化的进一步解释，请阅读教程 <https://www.chromium.org/developers/how-tos/trace-event-profiling-tool>。

Tip

llvm-xray 工具有更多的功能。可以在 LLVM 网站上找到它，网址是<https://llvm.org/docs/XRay.html>和<https://llvm.org/docs/XRayExample.html>。

本节中，我们学习了如何使用 XRay 检测应用程序，如何收集运行时信息，以及如何可视化数据。我们可以利用这些知识来发现应用程序中的性能瓶颈。

识别应用程序中的错误的另一种方法是通过静态分析器分析源代码。

使用 Clang 静态分析器检查源码

Clang Static Analyzer 是一个对 C、C++ 和 Objective-C 源代码执行额外检查的工具，静态分析程序执行的检查比编译器执行的检查更彻底。它们在时间和所需资源方面的成本也更高。静态分析程序有一组检查程序，用于检查某些 bug。

该工具对源代码执行符号解释，查看整个应用程序的所有代码路径，并从中派生应用程序中使用的值的约束。符号解释是编译器中常用的一种技术，例如：用于标识常量。在静态分析器的上下文中，检查器应用于派生值。

例如，如果除法的除数是 0，那么静态分析程序就会发出警告。我们可以用下面这个存储在 div.c 文件中的例子来检查：

```
1 int divbyzero(int a, int b) { return a / b; }
2
3 int bug() { return divbyzero(5, 0); }
```

在这个例子中，静态分析程序会对被 0 除法发出警告。然而，在编译时，带有 clang -Wall -c div.c 命令的文件将不会显示任何警告。

有两种方法可以从命令行调用静态分析程序。旧工具是扫描构建，包含在 LLVM 中，可以用于简单的场景。新工具是 CodeChecker，可在<https://github.com/Ericsson/codechecker/>。对于检查单个文件，扫描构建工具是更简单的解决方案。只需将 compile 命令传递给工具，其他工作就会自动完成：

```
$ scan-build clang -c div.c
scan-build: Using '/usr/local/llvm12/bin/clang-12' for static
analysis
div.c:2:12: warning: Division by zero [core.DivideZero]
    return a / b;
           ~~~~
1 warning generated.
scan-build: Analysis run complete.
scan-build: 1 bug found.
scan-build: Run 'scan-view /tmp/scanbuild-2021-03-01-023401-8721-1'
to examine bug reports.
```

屏幕上的输出已经发现了一个问题，即触发了名为 core.DivideZero 的检查器。但这还不是全部，可以在 /tmp 目录的子目录中找到一个完整的 HTML 报告。可以使用 scan-view 命令查看报表，也可以在浏览器的子目录中打开 index.html 文件。

报告的第一页显示了 bug 的摘要：

sanitizers - scan-build results

User:	kai@freebsd
Working Directory:	/usr/home/kai/sanitizers
Command Line:	clang-12 -c div.c
Clang Version:	clang version 12.0.0
Date:	Sat Apr 3 22:47:20 2021

Bug Summary

Bug Type	Quantity	Display?
All Bugs	1	<input checked="" type="checkbox"/>
Logic error		
Division by zero	1	<input checked="" type="checkbox"/>

Reports

Bug Group	Bug Type ▾	File	Function/Method	Line	Path Length
Logic error	Division by zero	div.c	divbyzero	2	3 View Report

图 11.3 -摘要页面

对于每个错误，摘要页面显示错误的类型、源中的位置，以及分析器找到错误的路径长度。提供了指向错误详细报告的链接。

下面的截图显示了错误的详细报告：

Bug Summary

File: /home/kai/sanitizers/div.c

Warning: [line 2, column 12](#)
Division by zero

Annotated Source Code

Press ['?'](#) to see keyboard shortcuts

[Show analyzer invocation](#)

☐ Show only relevant lines

```
1 int divbyzero(int a, int b) {  
2     return a / b;  
3 }  
4  
5 int bug() {  
6     return divbyzero(5, 0);  
7 }
```

3 ← Division by zero

1 Passing the value 0 via 2nd parameter 'b' →

2 ← Calling 'divbyzero' →

图 11.4 - 详细报告

有了详细的报告，就能够通过跟踪编号的气泡来验证错误。我们的示例中，以三个步骤展示了如何将 0 作为参数值传递导致除 0 错误。

确认工作需要人工进行。如果派生的约束对于某个检查器来说不够精确，那么可能会出现误报。也就是说，对于完全正确的代码会报告错误。根据这份报告，可以通过人工来确定假阳性。

您不限于该工具提供的检查器，还可以添加新的检查器。下一节将展示如何做到这一点。

向 Clang 静态分析器添加新检查器

要向 Clang Static Analyzer 添加一个新的检查器，需要创建一个 checker 类的新子类。静态分析程序尝试代码中所有可能的路径。分析器引擎在某些点生成事件，例如：在函数调用之前或之后。如果需要处理这些事件，则类必须为它们提供回调。Checker 类和事件的注册在 clang/include/clang/StaticAnalyzer/Core/Checker.h 头文件中提供。

通常，检查器需要跟踪一些符号。但是检查器不能管理状态，因为它不知道分析器引擎当前尝试的代码路径。因此，跟踪的状态必须在引擎中注册，并且只能用于 ProgramStateRef 实例。

许多库提供了必须成对使用的函数，例如：C 标准库提供了 malloc() 和 free() 函数。由 malloc() 函数分配的内存必须由 free() 函数精确地释放一次。没有调用 free() 函数，或者多次调用它，是一个编程错误。这种编码模式还有很多实例，静态分析器为其中一些提供了检查器。

iconv 库提供了将文本从一种编码转换为另一种编码的函数，例如：从 Latin-1 编码转换为 UTF-16 编码。要执行转换，实现需要分配内存。为了透明地管理内部资源，iconv 库提供了 iconv_open() 和 iconv_close() 函数，它们必须成对使用。您可以实现一个检查器来进行检查。

为了检测错误，检查器需要跟踪从 iconv_open() 函数返回的描述符。分析引擎为 iconv_open()

函数的返回值返回一个 SymbolRef 实例。我们将这个符号与一个状态关联起来，以反映是否调用了 iconv_close()。对于状态，创建了 IconvState 类，其中只封装了一个 bool 值。

新 IconvChecker 类需要处理的四个事件：

- PostCall，发生在函数调用之后。调用 iconv_open() 函数后，检索返回值的符号，并将其视为打开状态。
- PreCall，发生在函数调用之前。在调用 iconv_close() 函数之前，检查描述符的符号是否处于打开状态。如果不是，则说明描述符已经调用了 iconv_close() 函数，并且检测到对该函数的双重调用。
- DeadSymbol，当未使用的符号清除时发生。我们检查描述符的未使用符号是否仍处于打开状态。如果是，则检测到对 iconv_close() 的未调用，这是一个资源泄漏。
- PointerEscape，当分析器不能再跟踪符号时调用时，我们从状态中删除了符号，因为不能再推断描述符是否关闭了。

新的检查器在 Clang 项目中实现。让我们从将新的检查器添加到所有检查器集合开始，即 clang/include/clang/StaticAnalyzer/Checkers/Checkers.td 文件。每个检查器都与包相关联，我们的新检查器正在开发中，因此它属于 alpha 包。iconv API 是 posix 标准化的 API，所以它也属于 unix 包。在 Checkers.td 文件中找到 UnixAlpha 部分，并添加以下代码注册新的 IconvChecker：

```
1 def IconvChecker : Checker<"Iconv">,  
2   HelpText<"Check handling of iconv functions">,  
3   Documentation<NotDocumented>;
```

这将把新的检查器添加到已知检查器集合中，为命令行选项设置帮助文本，并声明没有关于此检查器的文档。

接下来，在 clang/lib/StaticAnalyzer/Checkers/IconvChecker.cpp 文件中实现检查器：

1. 为了实现，我们需要包含几个头文件。注册检查器需要 BuiltinCheckerRegistration.h 文件。Checker.h 文件提供了 Checker 类的声明和事件的回调。CallEvent.h 文件声明用于调用事件的类，CheckerContext 类的声明需要 CheckerContext.h 文件，CheckerContext 类是提供访问分析器状态的中心类：

```
1 #include "clang/StaticAnalyzer/Checkers/  
2 BuiltinCheckerRegistration.h"  
3 #include "clang/StaticAnalyzer/Core/Checker.h"  
4 #include "clang/StaticAnalyzer/Core/  
5 PathSensitive/CallEvent.h"  
6 #include "clang/StaticAnalyzer/Core/PathSensitive/  
7 CheckerContext.h"
```

2. 为了避免输入命名空间名称：

```
1 using namespace clang;  
2 using namespace ento;
```

3. 我们将状态与代表 iconv 描述符的每个符号关联起来。状态可以是打开的，也可以是关闭的，我们使用了一个 boolean 类型的变量，该变量的真值表示打开状态。状态值封装在 IconvState 结构体中。此结构与 FoldingSet 数据结构一起使用，后者是过滤重复条目的散列表。为了使用这个数据结构实现，这里添加了 Profile() 方法，它设置这个结构的唯一位。我们将结构体放入一个匿名命名空间，以避免对全局命名空间的污染:

```
1 namespace {
2 struct IconvState {
3     const bool IsOpen;
4     public:
5     IconvState(bool IsOpen) : IsOpen(IsOpen) {}
6     bool isOpen() const { return IsOpen; }
7     bool operator==(const IconvState &O) const {
8         return IsOpen == O.IsOpen;
9     }
10    void Profile(llvm::FoldingSetNodeID &ID) const {
11        ID.AddInteger(IsOpen);
12    }
13 };
14 }
```

4. IconvState 结构体表示 iconv 描述符的状态，该描述符由 SymbolRef 类的符号表示。这最好使用映射来完成，将符号作为键，将状态作为值。如前所述，检查器不能保持状态。相反，状态必须注册到全局程序状态，这是通过 REGISTER_MAP_WITH_PROGRAMSTATE 宏完成的。这个宏引入了 IconvStateMap 名称，我们稍后将使用它来访问映射:

```
1 REGISTER_MAP_WITH_PROGRAMSTATE(IconvStateMap, SymbolRef,
2                                 IconvState)
```

5. 我们还在匿名命名空间中实现了 IconvChecker 类。请求的 PostCall、PreCall、DeadSymbols 和 PointerEscape 事件是 Checker 基类的模板参数:

```
1 namespace {
2     class IconvChecker
3     : public Checker<check::PostCall, check::PreCall,
4                     check::DeadSymbols,
5                     check::PointerEscape> {
```

6. IconvChecker 类只有 CallDescription 类型的字段，用于识别程序中 iconv_open()、iconv() 和 iconv_close() 函数调用:

```
1     CallDescription IconvOpenFn, IconvFn, IconvCloseFn;
```


7. report() 的作用是: 生成错误报告。该方法的重要参数是符号数组、错误类型和错误描述。该方法中, 为每个符号创建一个错误报告, 并将该符号标记为该错误感兴趣的符号。如果源范围作为参数提供, 将添加到报表中。最后, 生成报告:

```
1 void
2 report(ArrayRef<SymbolRef> Syms, const BugType &Bug,
3        StringRef Desc, CheckerContext &C,
4        ExplodedNode *ErrNode,
5        Optional<SourceRange> Range = None) const {
6     for (SymbolRef Sym : Syms) {
7         auto R = std::make_unique
8             <PathSensitiveBugReport>(
9             Bug, Desc, ErrNode);
10        R->markInteresting(Sym);
11        if (Range)
12            R->addRange(*Range);
13        C.emitReport(std::move(R));
14    }
15 }
```

8. IconvChecker 类的构造函数只使用函数名初始化 CallDescription 字段:

```
1 public:
2     IconvChecker()
3         : IconvOpenFn("iconv_open"), IconvFn("iconv"),
4           IconvCloseFn("iconv_close", 1) {}
```

9. checkPostCall() 在分析器执行函数调用后调用。如果执行的函数不是一个全局的 C 函数, 并且没有命名为 iconv_open, 那么就没有什么可做的了:

```
1 void checkPostCall(const CallEvent &Call,
2                   CheckerContext &C) const {
3     if (!Call.isGlobalCFunction() ||
4         !Call.isCalled(IconvOpenFn))
5         return;
```

10. 否则, 将尝试以符号的形式获取函数的返回值。为了在全局程序状态中存储具有打开状态的符号, 我们需要从 CheckerContext 实例中获取一个 ProgramStateRef 实例。状态是不可变的, 所以将符号添加到状态会产生一个新的状态。通过调用 addTransition() 方法, 分析器引擎会更新状态:

```
1 if (SymbolRef Handle =
2     Call.getReturnValue().getAsSymbol()) {
3     ProgramStateRef State = C.getState();
```



```

4     State = State->set<IconvStateMap>(
5         Handle, IconvState(true));
6     C.addTransition(State);
7 }
8 }

```

11. 调用 `checkDeadSymbols()` 方法来清理未使用的符号。我们循环遍历我们跟踪的所有符号，并询问 `SymbolReaper` 实例当前的符号是否已死：

```

1  void checkDeadSymbols(SymbolReaper &SymReaper,
2                          CheckerContext &C) const {
3      ProgramStateRef State = C.getState();
4      SmallVector<SymbolRef, 8> LeakedSyms;
5      for (auto SymbolState :
6           State->get<IconvStateMap>()) {
7          SymbolRef Sym = SymbolState.first;
8          IconvState &St = SymbolState.second;
9
10         if (SymReaper.isDead(Sym)) {

```

12. 如果符号已死，那么需要检查状态。如果状态仍然是打开的，那么这是一个潜在的资源泄漏。有一个例外：`iconv_open()` 在出现错误时返回-1。如果分析器处于处理此错误的代码路径中，则假设资源泄漏是错误的，因为函数调用失败。我们尝试从 `ConstraintManager` 实例获取该符号的值，如果该值为-1，则不认为该符号存在资源泄漏。向 `SmallVector` 实例添加一个泄漏的符号，以便稍后生成错误报告。最后，我们将死符号从程序状态中移除：

```

1      if (St.isOpen()) {
2          bool IsLeaked = true;
3          if (const llvm::APSIInt *Val =
4              State->getConstraintManager()
5                  .getSymVal(State, Sym))
6              IsLeaked = Val->getExtValue() != -1;
7          if (IsLeaked)
8              LeakedSyms.push_back(Sym);
9      }
10
11     State = State->remove<IconvStateMap>(Sym);
12 }
13 }

```

13. 在循环之后，我们调用 `generateNonFatalErrorNode()` 方法。此方法转换到新的程序状态，如果此路径还没有错误节点，则返回一个错误节点。`LeakedSyms` 容器保存泄漏符号的列表（可

能为空), 调用 report() 方法生成错误报告:

```
1  if (ExplodedNode *N =
2      C.generateNonFatalErrorNode(State)) {
3      BugType LeakBugType(this, "Resource Leak",
4                          "iconv API Error", true);
5      report(LeakedSyms, LeakBugType,
6            "Opened iconv descriptor not closed", C,
7            N);
8  }
9  }
```

14. 当分析器检测到无法跟踪参数的函数调用时, 将调用 checkPointerEscape() 函数。这种情况下, 必须假设不知道函数内部是否关闭了 iconv 描述符。唯一的例外是调用 iconv() 函数, 该函数执行转换, 并且已知不调用 iconv_close() 函数。这就完成了 IconvChecker 类的实现:

```
1  ProgramStateRef
2  checkPointerEscape(ProgramStateRef State,
3                    const InvalidatedSymbols &Escaped,
4                    const CallEvent *Call,
5                    PointerEscapeKind Kind) const {
6      if (Kind == PSK_DirectEscapeOnCall &&
7          Call->isCalled(IconvFn))
8          return State;
9      for (SymbolRef Sym : Escaped)
10         State = State->remove<IconvStateMap>(Sym);
11     return State;
12 }
13 };
14 }
```

15. 最后, 需要在 CheckerManager 实例上注册新的检查器。shouldRegisterIconvChecker() 方法返回 true, 表示默认情况下应该注册 IconvChecker, registerIconvChecker() 方法执行注册。这两个方法都是通过 Checkers.td 文件生成的代码调用:

```
1  void ento::registerIconvChecker(CheckerManager &Mgr) {
2      Mgr.registerChecker<IconvChecker>();
3  }
4
5  bool ento::shouldRegisterIconvChecker(
6  const CheckerManager &Mgr) {
7      return true;
8  }
```

这就完成了新检查器的实现。只需要将文件名添加到 clang/lib/StaticAnalyzer/Checkers/CmakeLists.txt 文件中的源文件名列表中:

```
add_clang_library(clangStaticAnalyzerCheckers
...
    IconvChecker.cpp
...)
```

要编译新的检查器，要切换到 build 目录，并运行 ninja 命令:

```
$ ninja
```

可以使用保存在 conf.c 文件中的以下源代码测试新的检查器，该文件有两次对 iconv_close() 函数的调用:

```
1 #include <iconv.h>
2
3 void doconv() {
4     iconv_t id = iconv_open("Latin1", "UTF-16");
5     iconv_close(id);
6     iconv_close(id);
7 }
```

学习了如何使用自己的检查器扩展 Clang 静态分析器，就可以使用这些知识来创建新的通用检查器，或将它们贡献给社区，或可以创建专门为您的需求构建的检查器，以提高自己产品的质量。

静态分析器是利用 Clang 基础设施构建的，下一节将介绍如何使用自己的插件来扩展 Clang。

创建自己的基于 Clang 的工具

静态分析器是一个令人印象深刻的例子，说明了使用 Clang 基础结构可以做什么。也可以通过插件来扩展 Clang，这样你就可以将自己的功能添加到 Clang 中。该技术与在 LLVM 中添加 Pass 插件非常相似。

用一个简单的插件来探索它的功能。LLVM 编码标准要求函数名以小写字母开头。然而，随着时间的推移，编码标准也在不断发展。在许多情况下，需要函数以大写字母开头。一个产生违反命名规则的警告插件可以帮助解决这个问题，所以让我们尝试一下实现这样一个插件。

因为希望在抽象语法树 (AST) 上运行用户定义的操作，所以需要定义 PluginASTAction 类的一个子类。如果使用 Clang 库来编写自己的工具，可以将动作定义为 ASTFrontendAction 类的子类。PluginASTAction 类是 ASTFrontendAction 类的一个子类，具有解析命令行选项的额外功能。

需要的另一个类是 ASTConsumer 类的子类。AST 消费者是一个类，可以使用它在 AST 上运行操作，而不管 AST 的来源是什么。我们的第一个插件不需要太多组件，可以在 NamingPlugin.cpp 文件中创建如下实现:

1. 首先包括必需的头文件。除了上面提到的 `ASTConsumer` 类，还需要一个编译器的实例和插件注册表：

```
1 #include "clang/AST/ASTConsumer.h"
2 #include "clang/Frontend/CompilerInstance.h"
3 #include "clang/Frontend/FrontendPluginRegistry.h"
```

2. 使用 `clang` 命名空间，并将实现放到一个匿名命名空间中，以避免名称冲突：

```
1 using namespace clang;
2 namespace {
```

3. 接下来，定义 `ASTConsumer` 类的子类。稍后，如果检测到违反命名规则，将发出警告。为此，需要一个对 `DiagnosticsEngine` 实例的引用。
4. 需要在类中存储一个 `CompilerInstance` 实例，然后可以请求一个 `DiagnosticsEngine` 实例：

```
1 class NamingASTConsumer : public ASTConsumer {
2     CompilerInstance &CI;
3
4 public:
5     NamingASTConsumer(CompilerInstance &CI) : CI(CI) {}
```

5. `ASTConsumer` 实例有几个入口方法。`HandleTopLevelDecl()` 方法最符合我们的要求，在顶层的每个声明都调用该方法。这不仅仅包括函数，例如：变量。因此，可以使用 LLVM RTTI `dyn_cast<>()` 函数来确定该声明是否为函数声明。`HandleTopLevelDecl()` 方法有一个声明组作为参数，它可以包含多个声明。这需要对声明进行循环。下面的代码向我们展示了 `HandleTopLevelDecl()` 方法的实现：

```
1 bool HandleTopLevelDecl(DeclGroupRef DG) override {
2     for (DeclGroupRef::iterator I = DG.begin(),
3         E = DG.end();
4         I != E; ++I) {
5         const Decl *D = *I;
6         if (const FunctionDecl *FD =
7             dyn_cast<FunctionDecl>(D)) {
```

6. 找到函数声明之后，需要检索函数的名称。还需要确保名称不为空：

```
1     std::string Name =
2         FD->getNameInfo().getName().getAsString();
3     assert(Name.length() > 0 &&
4         "Unexpected empty identifier");
```

如果函数名不是以小写字母开头，就违反了命名规则：

```
1     char &First = Name.at(0);
```

```
2         if (!(First >= 'a' && First <= 'z')) {
```

7. 要发出警告，需要一个 DiagnosticsEngine 实例。此外，还需要一个消息 ID。在 Clang 内部，消息 ID 定义为枚举。因为插件不是 Clang 的一部分，需要创建自定义 ID，然后使用它发出警告：

```
1         DiagnosticsEngine &Diag =
2             CI.getDiagnostics();
3         unsigned ID = Diag.getCustomDiagID(
4             DiagnosticsEngine::Warning,
5             "Function name should start with "
6             "lowercase letter");
7         Diag.Report(FD->getLocation(), ID);
```

8. 除了关闭所有的开花括号外，还需要从这个函数返回 true，表示处理可以继续：

```
1         }
2     }
3 }
4 return true;
5 }
6 };
```

9. 接下来，需要创建 PluginASTAction 子类，实现了 Clang 调用的接口：

```
1 class PluginNamingAction : public PluginASTAction {
2 public:
```

必须实现的第一个方法是 CreateASTConsumer() 方法，返回 NamingASTConsumer 类的一个实例。这个方法由 Clang 调用，传递的 CompilerInstance 实例可以访问编译器的所有重要类：

```
1     std::unique_ptr<ASTConsumer>
2     CreateASTConsumer(CompilerInstance &CI,
3                      StringRef file) override {
4         return std::make_unique<NamingASTConsumer>(CI);
5     }
```

10. 插件还可以访问命令行选项。我们的插件没有命令行参数，只需要返回 true 来表示成功：

```
1     bool ParseArgs(const CompilerInstance &CI,
2                   const std::vector<std::string> &args)
3                   override {
4         return true;
5     }
```

11. 插件的操作类型描述了何时调用该操作。默认值是 `Cmdline`，这意味着插件必须在命令行上命名才能调用。需要重写该方法并将其值更改为 `AddAfterMainAction`，插件才能自动运行：

```
1 PluginASTAction::ActionType getActionType() override {  
2     return AddAfterMainAction;  
3 }
```

12. `PluginNamingAction` 类的实现完成了：

```
1 };  
2 }
```

13. 最后，需要注册插件。第一个参数是插件的名称，第二个参数是帮助文本：

```
1 static FrontendPluginRegistry::Add<PluginNamingAction>  
2 X("naming-plugin", "naming plugin");
```

这就完成了插件的实现。要编译插件，在 `CMakeLists.txt` 文件中创建一个构建描述。该插件位于 Clang 源代码树之外，因此需要设置一个完整的项目。可以遵循以下步骤：

1. 首先定义所需的 CMake 版本和项目名称：

```
cmake_minimum_required(VERSION 3.13.4)  
project(namingplugin)
```

2. 接下来，包括 LLVM 文件。如果 CMake 不能自动找到文件，需要设置 `LLVM_DIR` 变量，以指向包含 CMake 文件的 LLVM 目录：

```
find_package(LLVM REQUIRED CONFIG)
```

3. 将带有 CMake 文件的 LLVM 目录添加到搜索路径中，并包含一些必需的模块：

```
list(APPEND CMAKE_MODULE_PATH ${LLVM_DIR})  
include(ChooseMSVCCRT)  
include(AddLLVM)  
include(HandleLLVMOptions)
```

4. 然后，加载 Clang 的 CMake 定义。如果 CMake 不能自动找到文件，那么必须设置 `Clang_DIR` 变量指向包含 CMake 文件的 Clang 目录：

```
find_package(Clang REQUIRED)
```

5. 接下来，定义头文件和库文件的位置，以及使用哪些定义：

```
include_directories("${LLVM_INCLUDE_DIR}"
    "${CLANG_INCLUDE_DIRS}")
add_definitions("${LLVM_DEFINITIONS}")
link_directories("${LLVM_LIBRARY_DIR}")
```

6. 前面的定义设置了构建环境。插入以下命令，定义插件的名称，插件的源文件，以及说明它是一个 Clang 插件：

```
add_llvm_library(NamingPlugin MODULE NamingPlugin.cpp
    PLUGIN_TOOL clang)
```

在 Windows 上，插件支持不同于 Unix 平台，必须链接所需的 LLVM 和 Clang 库：

```
if(LLVM_ENABLE_PLUGINS AND (WIN32 OR CYGWIN))
    set(LLVM_LINK_COMPONENTS Support)
    clang_target_link_libraries(NamingPlugin PRIVATE
        clangAST clangBasic clangFrontend clangLex)
endif()
```

7. 将这两个文件保存在 NamingPlugin 目录中。创建一个与 NamingPlugin 目录同级的 build-NamingPlugin 目录，并使用以下命令构建插件：

```
$ mkdir build-naming-plugin
$ cd build-naming-plugin
$ cmake -G Ninja ../NamingPlugin
$ ninja
```

这些步骤创建 NamingPlugin，构建目录中的动态库。

要测试插件，请将下面的源代码保存为 named.c 文件。Func1 函数名违反了命名规则，但主函数没有违反规则：

```
1 int Func1() { return 0; }
2 int main() { return Func1(); }
```

要调用插件，需要使用 -fplugin= 指定：

```
$ clang -fplugin=./NamingPlugin.so naming.c
naming.c:1:5: warning: Function name should start with
lowercase letter
int Func1() return 0;
    ^
1 warning generated.
```

这种调用需要覆盖 PluginASTAction 类的 getActionType() 方法,并且返回一个不同于 Cmdline 默认值的值。

如果没有这样做,但还想对插件操作的调用有更多的控制,那么可以从编译器命令行运行插件:

```
$ clang -cc1 -load ./NamingPlugin.so -plugin naming-plugin\
naming.c
```

祝贺你,已经构建了第一个 Clang 插件!

这种方法的缺点是它有一定的局限性。ASTConsumer 类有不同的入口方法,但它们都是粗粒度的,这可以通过使用 RecursiveASTVisitor 类来解决。这个类遍历所有 AST 节点,可以覆盖感兴趣的 VisitXXX() 方法。可以按照以下步骤来为使用“访问者”重写插件:

1. 定义 RecursiveASTVisitor 类需要一个额外的 include:

```
1 #include "clang/AST/RecursiveASTVisitor.h"
```

2. 然后,将访问者定义为匿名名称空间中的第一个类。只存储对 AST 上下文的引用,将使您访问所有用于 AST 操作的重要方法,包括发出警告所需的 DiagnosticsEngine 实例:

```
1 class NamingVisitor
2   : public RecursiveASTVisitor<NamingVisitor> {
3 private:
4     ASTContext &ASTCtx;
5 public:
6     explicit NamingVisitor(CompilerInstance &CI)
7       : ASTCtx(CI.getASTContext()) {}
```

3. 遍历过程中,只要发现函数声明,就调用 VisitFunctionDecl() 方法。将内部循环的主体复制到 HandleTopLevelDecl() 函数中:

```
1 virtual bool VisitFunctionDecl(FunctionDecl *FD) {
2     std::string Name =
3         FD->getNameInfo().getName().getAsString();
4     assert(Name.length() > 0 &&
5            "Unexpected empty identifier");
6     char &First = Name.at(0);
```



```

7     if (!(First >= 'a' && First <= 'z')) {
8         DiagnosticsEngine &Diag =
9             ASTCtx.getDiagnostics();
10        unsigned ID = Diag.getCustomDiagID(
11            DiagnosticsEngine::Warning,
12            "Function name should start with "
13            "lowercase letter");
14        Diag.Report(FD->getLocation(), ID);
15    }
16    return true;
17 }
18 };

```

4. 这就完成了访问者实现。在 `NamingASTConsumer` 类中，现在只会存储一个访问者实例：

```

1     std::unique_ptr<NamingVisitor> Visitor;
2 public:
3     NamingASTConsumer(CompilerInstance &CI)
4         : Visitor(std::make_unique<NamingVisitor>(CI)) {}

```

5. 您将删除 `HandleTopLevelDecl()` 方法，因为该功能现在在访问者类中，所以需要重写 `HandleTranslationUnit()` 方法。每个翻译单元调用一次，会在这里开始 AST 遍历：

```

1     void
2     HandleTranslationUnit(ASTContext &ASTCtx) override {
3         Visitor->TraverseDecl(
4             ASTCtx.getTranslationUnitDecl());
5     }

```

这个新的实现具有完全相同的功能。优点是它更容易扩展，例如：想检查变量声明，可以实现 `VisitVarDecl()` 方法。或者，如果想使用语句，那么可以实现 `VisitStmt()` 方法。基本上，对于 C、C++ 和 Objective-C 语言的每个实体都有一个 visitor 方法。

访问 AST 允许构建执行复杂任务的插件。如本节所述，强制执行命名约定是对 Clang 的一个补充。另一个可以作为插件实现的有用的附加功能是计算软件度量，比如：圈复杂度。您还可以添加或替换 AST 节点，例如：允许添加运行时检测。添加插件可以以需要的方式，对 Clang 进行扩展。

总结

本章中，学习了如何使用各种 sanitizer。使用地址 sanitizer 检测到指针错误，使用内存 sanitizer 检测到未初始化的内存访问，并使用线程 sanitizer 检测到数据竞争。应用程序错误通常是由不正确的输入触发的，实现了模糊测试来使用随机数据测试应用程序。

使用 XRay 对应用程序进行了测试，以确定性能瓶颈，还了解了可视化数据的各种方法。本章中，还使用了 Clang 静态分析器来通过解释源代码来查找可能的错误，并学习了如何构建自己的 Clang 插件。

这些技能将帮助您提高所构建的应用程序的质量。在应用程序用户抱怨运行时错误之前，找到它们当然是好的。应用本章中获得的知识，不仅可以找到广泛的常见错误，还可以用新的功能扩展 Clang。

下一章中，您将学习如何想 LLVM 添加一个新的后端。

第 12 章 自定义编译器后端

LLVM 有一个非常灵活的架构。您可以向它添加一个新的后端，后端的核心是目标描述，大部分代码都是从它生成的。但是，还不可能生成一个完整的后端，并且实现调用规则需要进行手动编码。本章中，我们将学习如何添加对老 CPU 的支持。

本章中，我们将学习以下内容：

- 设置一个新的后端时，我们将向您介绍 M88k CPU 体系架构，并向您展示在哪里可以找到您需要的信息。
- 将新的体系结构添加到 Triple 类中，将教会您如何让 LLVM 了解新的 CPU 体系架构。
- 在扩展 LLVM 中的 ELF 文件格式定义时，可以向处理 ELD 对象文件的库和工具中添加对 m88k 特定重定位的支持。
- 创建目标描述时，您将使用 TableGen 语言开发目标描述的所有部分。
- 实现 DAG 指令选择类时，您将创建指令选择所需的 Pass 和支持的类。
- 生成汇编指令时，会带您了解如何实现汇编打印，生成汇编文本。
- 生成的机器码时，您将了解必须提供哪些附加类才能使机器码 (MC) 层将代码写入目标文件。
- 添加反汇编支持时，您将了解如何实现对反汇编程序的支持。
- 将这些功能整合在一起时，就可以将新后端的源代码集成到构建系统中。

本章结束时，您将了解如何开发一个新的和完整的后端。您将了解后台的不同组成部分，从而更深入地了解 LLVM 体系结构。

相关代码

本章的代码文件可在<https://github.com/PacktPublishing/Learn-LLVM-12/tree/master/Chapter12>获取。

你可以在视频中找到代码<https://bit.ly/3nllhED>。

为新的后端搭建舞台

无论是商业上需要支持一个新的 CPU，还是只是一个业余项目需要添加对一些旧架构的支持，为 LLVM 添加一个新后端都是一项重要任务。以下部分概述了开发新后端所需的内容。我们将为摩托罗拉 M88k 架构添加一个后端，这是一个 20 世纪 80 年代的 RISC 架构。

References

可以在维基百科上阅读更多关于该架构的信息:https://en.wikipedia.org/wiki/Motorola_88000。关于这个体系结构的重要信息仍然可以在互联网上找到,可以在<http://www.bitsavers.org/components/motorola/88000/>找到带有指令集和计时信息的 CPU 手册,和 SystemV ABI M88k 处理器补充 ELF 格式的定义和调用规则可以在https://archive.org/details/bitsavers_attunixSysa0138776555SystemVRelease488000ABI1990_8011463找到。

OpenBSD(可在<https://www.openbsd.org/>获得) 仍然支持 LUNA-88k 系统。在 OpenBSD 系统上,很容易为 M88k 创建一个 GCC 交叉编译器。GXemul 可在<http://gavarese/gxemul/>查看相应资料,并且其有一个模拟器能够运行针对 M88k 体系结构的某些 OpenBSD 版本。

总的来说, M88k 体系结构已经淘汰很久了,但是我们找到了足够的信息和工具,可以为它添加一个 LLVM 后端。我们将从一个非常基本的任务开始,并扩展到 Triple 类。

将新的体系结构添加到 Triple 类中

Triple 类的一个实例表示 LLVM 为其生成代码的目标平台。为了支持新的体系架构,第一个任务是扩展 Triple 类。在 `llvm/include/llvm/ADT/Triple.h` 文件中,可向 `ArchType` 枚举添加了一个成员和一个新的谓词:

```
1 class Triple {
2     public:
3     enum ArchType {
4         // Many more members
5         m88k, // M88000 (big endian): m88k
6     };
7     /// Tests whether the target is M88k.
8     bool isM88k() const {
9         return getArch() == Triple::m88k;
10    }
11    // Many more methods
12};
```

在 `llvm/lib/Support/Triple.cpp` 文件中,有许多使用 `ArchType` 枚举的地方。需要扩展它们,例如:在 `getArchTypeName()` 方法中,添加一个新的 case:

```
1 switch (Kind) {
2     // Many more cases
3     case m88k: return "m88k";
4 }
```

大多数情况下，如果忘记在某个函数中处理新的 m88k 枚举成员，编译器会发出警告。接下来，我们将扩展可执行和可链接格式 (ELF) 定义。

在 LLVM 中扩展 ELF 文件格式定义

ELF 文件格式是 LLVM 支持读写的二进制对象文件格式之一。ELF 本身是为许多 CPU 架构结构定义的，对于 M88k 体系架构也有一个定义。我们需要做的就是添加重定位的定义和一些标志。重定位可在 System V ABI M88k 处理器手册的第 4 章看到：

1. 需要在 `llvm/include/llvm/BinaryFormat/ELFRelocs/M88k.def` 文件中输入以下内容：

```
1 #ifndef ELF_RELOC
2 #error "ELF_RELOC must be defined"
3 #endif
4 ELF_RELOC(R_88K_NONE, 0)
5 ELF_RELOC(R_88K_COPY, 1)
6 // Many more...
```

2. 还需要在 `llvm/include/llvm/BinaryFormat/ELF.h` 文件中添加了一些标志，并包含了重定位定义：

```
1 // M88k Specific e_flags
2 enum : unsigned {
3     EF_88K_NABI = 0x80000000, // Not ABI compliant
4     EF_88K_M88110 = 0x00000004 // File uses 88110-
5     // specific
6     // features
7 };
8 // M88k relocations.
9 enum {
10 #include "ELFRelocs/M88k.def"
11 };
```

代码可以添加到文件中的任何地方，但最好保持排序顺序，并在 MIPS 体系结构的代码之前插入它。

3. 还需要扩展其他一些方法。在 `llvm/include/llvm/Object/ELFObjectFile.h` 文件中，有一些方法可以在枚举成员和字符串之间进行转换，例如：必须在 `getFileFormatName()` 方法中添加新的 case 语句：

```
1 switch (EF.getHeader()->e_ident[ELF::EI_CLASS]) {
2     // Many more cases
3     case ELF::EM_88K:
4         return "elf32-m88k";
5 }
```

4. 类似地，扩展了 `getArch()` 方法。
5. 最后，在 `llvm/lib/Object/ELF.cpp` 文件的 `getELFRelocationTypeName()` 方法中使用重定位定义：

```
1  switch (Machine) {
2      // Many more cases
3      case ELF::EM_88K:
4          switch (Type) {
5      #include "llvm/BinaryFormat/ELFRelocs/M88k.def"
6          default:
7              break;
8      }
9      break;
10 }
```

6. 为了完成支持，还可以在 `llvm/lib/ObjectYAML/ELFYAML.cpp` 文件中，映射 `ELFYAML::ELF_REL` 枚举的方法中添加重定位。
7. 至此，我们已经以 ELF 文件格式完成了对 m88k 体系结构的支持。可以使用 `llvm-readobj` 工具检查 ELF 对象文件，例如：由 OpenBSD 上的交叉编译器创建的文件。同样，可以使用 `yaml2obj` 工具为 m88k 体系结构创建一个 ELF 对象文件。

是否必须添加对对象文件格式的支持？

将架构支持集成到 ELF 文件格式实现中只需要几行代码。如果为其创建 LLVM 后端的体系结构使用 ELF 格式，那么应该采用此方法。另一方面，添加对全新二进制文件格式的支持本身就是一项复杂的任务。在这种情况下，一种可能的方法是只输出汇编程序文件，并使用外部汇编程序创建对象文件。

通过这些添加，ELF 文件格式的实现现在支持 M88k 体系结构。下一节中，我们将创建 M88k 体系结构的描述，该描述介绍了该体系结构的指令、寄存器、调用规则，以及其他细节。

创建目标描述

目标描述是后端实现的核心。理想情况下，可以从目标描述生成整个后端。这个目标还没有达到，因此，需要在以后扩展生成的代码。让我们从顶层文件开始剖析目标。

实现目标描述的顶层文件

我们将新后端文件放入 `llvm/lib/Target/M88k` 目录，目标描述在 `M88k.td` 文件中：

1. 这个文件中，首先需要包含由 LLVM 预定义的基本目标描述类，然后是在下一节中要创建的文件：

```
include "llvm/Target/Target.td"

include "M88kRegisterInfo.td"
include "M88kCallingConv.td"
include "M88kSchedule.td"
include "M88kInstrFormats.td"
include "M88kInstrInfo.td"
```

2. 接下来，还要定义受支持的处理器。除此之外，也可以转换为-mcpu=option 的参数:

```
def : ProcessorModel<"mc88110", M88kSchedModel, []>;
```

3. 完成所有这些定义后，就可以将目标拼凑在一起了。定义了这些子类，为了可以修改默认值。M88kInstrInfo 类保存了关于指令的所有信息:

```
def M88kInstrInfo : InstrInfo;
```

4. 为.s 汇编文件定义了一个解析器，声明寄存器名称时总是以% 作为前缀:

```
def M88kAsmParser : AsmParser;
def M88kAsmParserVariant : AsmParserVariant {
    let RegisterPrefix = "%";
}
```

5. 接下来，为程序集编写器定义一个类，负责编写.s 汇编文件:

```
def M88kAsmWriter : AsmWriter;
```

6. 最后，将所有这些记录放在一起定义目标:

```
def M88k : Target {
    let InstructionSet = M88kInstrInfo;
    let AssemblyParsers = [M88kAsmParser];
    let AssemblyParserVariants = [M88kAsmParserVariant];
    let AssemblyWriters = [M88kAsmWriter];
    let AllowRegisterRenaming = 1;
}
```

现在已经实现了顶层文件，我们将创建包含的文件，从寄存器定义开始。

添加寄存器定义

CPU 体系架构通常定义一组寄存器，这些寄存器的特性可以有很大的不同。一些架构允许访问子寄存器，例如：x86 体系结构有特殊的寄存器名，只能访问寄存器值的一部分，其他体系结构没有实现这一点。除了通用寄存器、浮点寄存器和向量寄存器外，体系结构还可以定义特殊寄存器，例如：状态码寄存器或浮点操作配置寄存器。您需要为 LLVM 定义这些信息。

M88k 体系结构定义了通用寄存器、浮点寄存器和控制寄存器。为了使示例简短，我们将只定义通用寄存器。我们从为寄存器定义一个超类开始，寄存器的编码只使用 5 位，这也限制了保存编码的字段。我们还定义所有生成的 C++ 代码应该放在 M88k 命名空间中：

```
class M88kReg<bits<5> Enc, string n> : Register<n> {  
    let HWEncoding15-5 = 0;  
    let HWEncoding4-0 = Enc;  
    let Namespace = "M88k";  
}
```

M88kReg 类用于所有寄存器类型，为通用寄存器定义了一个特殊的类：

```
class GRi<bits<5> Enc, string n> : M88kReg<Enc, n>;
```

现在可以定义所有通用寄存器 (32 个) 了：

```
foreach I = 0-31 in {  
    def R#I : GRi<I, "r"#I>;  
}
```

单个寄存器需要分组到寄存器类中，寄存器的顺序还定义了寄存器分配器中的分配顺序。这里，可以简单地添加所有寄存器：

```
def GPR : RegisterClass<"M88k", [i32], 32,  
    (add (sequence "R%u", 0, 31))>;
```

最后，需要定义一个基于寄存器类的操作数。该操作数用于选择与寄存器匹配的 DAG 节点，它还可以扩展为在汇编代码中打印和匹配寄存器的方法名：

```
def GPROpnd : RegisterOperand<GPR>;
```

这就完成了寄存器的定义。下一节中，将使用这些定义来定义调用规则。

定义调用规则

调用规则定义了如何将参数传递给函数。通常，第一个参数在寄存器中传递，其余的参数在堆栈上传递。还必须有关于如何传递聚合和如何从函数返回值的规则。根据这里给出的定义，将生成分析程序类，稍后在底层调用时使用它们。

您可以在 System V ABI M88k Processor 手册的第 3 章，“底层系统信息”中阅读有关 M88k 架构上使用的调用规则。这里，我们把它翻译成 TableGen 语法：

1. 为调用规则定义了一个记录：

```
def CC_M88k : CallingConv<[
```

2. M88k 架构只有 32 位寄存器，因此较小的数据类型的值需要提升到 32 位：

```
CCIfType<[i1, i8, i16], CCPromoteToType<i32>»,
```

3. 调用规则规定，对于聚合返回值，指向内存的指针会传递到 r12 寄存器中：

```
CCIfSRet<CCIfType<[i32], CCAssignToReg<[R12]>»>,
```

4. 寄存器 r2 到 r9 用于传递参数：

```
CCIfType<[i32,i64,f32,f64],  
  CCAssignToReg<[R2, R3, R4, R5, R6, R7, R8,  
    R9]>»,
```

5. 每个附加的参数在堆栈上进行传递，需要 4 字节对齐：

```
CCAssignToStack<4, 4>,  
>;
```

6. 另一条记录定义如何将结果传递给调用函数。32 位值在 r2 寄存器中传递，64 位值使用 r2 和 r3 寄存器：

```
def RetCC_M88k : CallingConv<[  
  CCIfType<[i32,f32], CCAssignToReg<[R2]>»,  
  CCIfType<[i64,f64], CCAssignToReg<[R2, R3]>»  
>;
```

7. 最后，调用规则还规定了调用函数必须保留哪些寄存器：

```
def CSR_M88k :  
  CalleeSavedRegs<(add (sequence "R%d", 14,  
    25), R30)>;
```

如果需要，还可以定义多个调用规则。下一节中，我们将简要介绍调度模型。

创建调度模型

使用调度模型代码生成，可以以最优方式对指令进行排序。定义一个调度模型可以提高所生成代码的性能，但不是代码生成所必需的。因此，我们只为模型定义一个占位符。我们添加的 CPU 一次最多可以发出两条指令的信息，并且它是有序 CPU:

```
def M88kSchedModel : SchedMachineModel {  
    let IssueWidth = 2;  
    let MicroOpBufferSize = 0;  
    let CompleteModel = 0;  
    let NoModel = 1;  
}
```

可以在 YouTube 上的“编写伟大的调度程序”(Writing Great Schedulers) 讲座中找到关于如何创建完整调度模型的方法，网址是<https://www.youtube.com/watch?v=brpomKUynEA>。

接下来，来定义指令格式和指令。

定义指令格式和指令信息

我们已经在第 9 章，支持新机器指令的章节中了解了指令格式和指令信息。为了定义 M88k 体系结构的指令，我们采用相同的方法。首先，为指令记录定义一个基类。这个类最重要的字段是 Inst 字段，保存指令的编码。大多数其他字段定义只是给指令超类中定义的字段赋值:

```
class InstM88k<dag outs, dag ins, string asmstr,  
    list<dag> pattern, InstrItinClass itin =  
        NoItinerary>  
    : Instruction {  
    field bits<32> Inst;  
    field bits<32> SoftFail = 0;  
    let Namespace = "M88k";  
    let Size = 4;  
    dag OutOperandList = outs;  
    dag InOperandList = ins;  
    let AsmString = asmstr;  
    let Pattern = pattern;  
    let DecoderNamespace = "M88k";  
    let Itinerary = itin;  
}
```

这个基类用于所有的指令格式，所以也用于 F_JMP 格式。您需要根据处理器用户手册的介绍，对处理器进行编码。类有两个参数 (是编码的一部分)，func 参数定义了编码的第 11 位到第 15 位，它将指令定义为带有或不保存返回地址的跳转。下一个参数是一个位，它定义下一条指令是否无条

件执行。这类似于 MIPS 架构的延迟槽。

该类还定义了 rs2 字段，该字段保存保存目标地址的寄存器的编码。其他参数包括 DAG 输入和输出操作数、文本汇编字符串、用于选择该指令的 DAG 模式，以及用于调度程序模型的 itinerary 类：

```
class F_JMP<bits<5> func, bits<1> next,  
          dag outs, dag ins, string asmstr,  
          list<dag> pattern,  
          InstrItinClass itin = NoItinerary>  
  : InstM88k<outs, ins, asmstr, pattern, itin> {  
    bits<5> rs2;  
    let Inst31-26 = 0b111101;  
    let Inst25-16 = 0b0000000000;  
    let Inst15-11 = func;  
    let Inst10 = next;  
    let Inst9-5 = 0b00000;  
    let Inst4-0 = rs2;  
  }
```

有了这个，就可以定义指令了。跳转指令是基本块中的最后一条指令，因此需要设置 isTerminator 标志。因为控制流不能通过这条指令，所以还必须设置 isBarrier 标志。我们从处理器的用户手册中获取 func 和 next 参数的值。

输入 DAG 操作数是一个通用寄存器，它引用前面寄存器的信息中的操作数。编码存储在 rs2 字段中，来自前面的类定义，输出操作数为空。汇编字符串给出指令的文本语法，也引用寄存器操作数。DAG 模式使用预定义的 brind 操作符，如果 DAG 包含一个间接分支节点，目标地址保存在寄存器中，则选择此指令：

```
let isTerminator = 1, isBarrier = 1 in  
  def JMP : F_JMP<0b11000, 0, (outs), (ins GPROpnd:$rs2),  
              "jmp $rs2", [(brind GPROpnd:$rs2)]>;
```

我们需要以这种方式为所有指令进行定义。

这个文件中，还实现了用于指令选择的其他模式。一个典型的应用是不断的合成，M88k 体系架构是 32 位的，但可以将作为操作数的指令也能支持 16 位范围的常量。因此，诸如按位运算以及寄存器和 32 位常量之间的运算，必须分割成两个使用 16 位常量的指令。

幸运的是，and 指令中的一个标志定义了一个操作是用于寄存器的下半部分还是上半部分。使用操作符 LO16 和 HI16 来提取一个常数的下半部分或上半部分，我们可以为一个寄存器和一个 32 位宽常数之间的和运算建立一个 DAG 模式：

```
def : Pat<(and GPR:$rs1, uimm32:$imm),
          (ANDriu (ANDriu GPR:$rs1, (HI16 i32:$imm)),
                (LO16 i32:$imm))>;
```

ANDriu 运算符是将常数应用到寄存器下半部分的 and 指令，ANDriu 运算符使用寄存器上半部分。当然，在使用这些模式之前，必须像定义 jmp 指令一样定义指令。该模式使用带有 and 操作的 32 位常量来解决问题，在指令选择期间为其生成两条机器指令。

不是所有的操作都可以用预定义的 DAG 节点来表示，例如：M88k 体系结构定义了位域操作，可以将其视为正常和/或操作的一般化。对于这样的操作，可以引入新的节点类型，例如：set 指令：

```
def m88k_set : SDNode<"M88kISD::SET", SDTIntBinOp>;
```

这定义了一个 SDNode 类的新记录。第一个参数是表示新操作的 C++ 枚举成员。第二个参数是所谓的类型概要文件，它定义了参数的类型、数量和结果类型。预定义的 SDTIntBinOp 类定义了两个整型参数和一个整型结果类型。可以在 llvm/include/llvm/Target/TargetSelectionDAG.td 文件中查找预定义的类。如果没有合适的预定义类型说明文件，那么可以定义一个新的。

对于调用函数，LLVM 需要某些无法预定义的函数，因为它们不是完全独立于目标的，例如：对于 returns，我们需要指定一个 retflag 记录：

```
def retflag : SDNode<"M88kISD::RET_FLAG", SDTNone,
                    [SDNPHasChain, SDNPOptInGlue, SDNPVariadic]>;
```

与 m88k_set 相比，这还为 DAG 节点定义了一些标志：使用了链和胶水序列，操作符可以接受可变数量的参数。

以迭代的方式实现指令

现代的 CPU 可以很容易地拥有数千条指令，不一次性实现所有指令是有意义的。相反，您应该首先关注基本指令，如逻辑操作和调用和返回指令。这足以使一个非常基本的后端工作，然后在这个基础上添加越来越多的指令定义和模式。

这就完成了目标描述的实现。从目标描述中，使用 llvm-tblgen 工具自动生成大量代码。为了完成指令选择和后台的其他部分，我们仍需要使用生成的代码开发 C++ 代码。下一节中，我们将实现 DAG 指令的选择。

实现 DAG 指令选择类

DAG 指令选择器的很大一部分是由 llvm-tblgen 工具生成的。不过，仍然需要使用生成的代码创建类，并将所有内容放在一起。让我们从初始化过程开始。

初始化目标机器

每个后端必须提供至少一个 TargetMachine 类，通常是 LLVMTargetMachine 类的子类。M88k TargetMachine 类包含代码生成所需的许多细节，还充当其他后端类的工厂，尤其是 Subtarget 类和 TargetPassConfig 类。Subtarget 类保存代码生成的配置，比如启用了哪些特性。TargetPassConfig 类配置后端机器的通行证。M88kTargetMachine 类的声明在 M88kTargetMachine.h 文件中：

```
1 class M88kTargetMachine : public LLVMTargetMachine {
2 public:
3     M88kTargetMachine(/* parameters */);
4     ~M88kTargetMachine() override;
5     const M88kSubtarget *getSubtargetImpl(const Function &)
6                                         const override;
7     const M88kSubtarget *getSubtargetImpl() const = delete;
8     TargetPassConfig *createPassConfig(PassManagerBase &PM)
9                                         override;
10 };
```

请注意，每个功能可以有不同的子目标。

M88kTargetMachine.cpp 文件中的实现非常简单。最有趣的是这个后端机器 Pass 的设置。这会创建到选择 DAG 的连接 (如果需要，还会创建到全局指令选择的连接)。在类中创建的 Pass 随后添加到 Pass 流水中，以便从 IR 生成目标文件或汇编程序：

```
1 namespace {
2 class M88kPassConfig : public TargetPassConfig {
3 public:
4     M88kPassConfig(M88kTargetMachine &TM, PassManagerBase
5                     &PM)
6         : TargetPassConfig(TM, PM) {}
7     M88kTargetMachine &getM88kTargetMachine() const {
8         return getTM<M88kTargetMachine>();
9     }
10
11     bool addInstSelector() override {
12         addPass(createM88kISelDag(getM88kTargetMachine(),
13                                   getOptLevel()));
14         return false;
15     }
16 };
17 } // namespace
18
19 TargetPassConfig *M88kTargetMachine::createPassConfig(
20     PassManagerBase &PM) {
21     return new M88kPassConfig(*this, PM);
22 }
```

22 }

从 M88kTargetMachine 类返回的 SubTarget 实现允许访问其他重要类。M88kInstrInfo 类返回关于指令的信息，包括寄存器。m88ktargetlower 类提供了与调用相关的指令的降低，还允许添加自定义 DAG 规则。这个类的大部分是由 llvm-tblgen 工具生成的，而且需要包含生成的头文件。

M88kSubTarget.h 文件的定义如下：

```
1 #define GET_SUBTARGETINFO_HEADER
2 #include "M88kGenSubtargetInfo.inc"
3
4 namespace llvm {
5 class M88kSubtarget : public M88kGenSubtargetInfo {
6     Triple TargetTriple;
7     virtual void anchor();
8
9     M88kInstrInfo InstrInfo;
10    M88kTargetLowering TLInfo;
11    M88kFrameLowering FrameLowering;
12
13 public:
14     M88kSubtarget(const Triple &TT, const std::string &CPU,
15                  const std::string &FS,
16                  const TargetMachine &TM);
17
18     void ParseSubtargetFeatures(StringRef CPU, StringRef FS);
19
20     const TargetFrameLowering *getFrameLowering() const
21     override
22     { return &FrameLowering; }
23     const M88kInstrInfo *getInstrInfo() const override
24     { return &InstrInfo; }
25     const M88kRegisterInfo *getRegisterInfo() const override
26     { return &InstrInfo.getRegisterInfo(); }
27     const M88kTargetLowering *getTargetLowering() const
28     override
29     { return &TLInfo; }
30 };
31 } // end namespace llvm
```

接下来，实现选择 DAG。

添加选择 DAG 的实现

选择 DAG 是在同名文件中的 M88kDAGtoDAGISel 类中实现的。这里，我们可以从创建目标机器描述中受益：大多数功能都是从这个描述中生成的。第一个实现中，只需要覆盖 Select() 函数并将其转发给生成的 SelectCode 函数。某些情况下，可以重写更多函数，例如：如果需要扩展 DAG 的预处理，或者如果需要添加特殊的内联汇编器规则。

因为这个类是一个机器函数的 Pass，所以我们也为 Pass 提供了一个名称。实现的主要部分来自生成的文件，将其包含在类的中间：

```
1 class M88kDAGtoDAGISel : public SelectionDAGISel {
2     const M88kSubtarget *Subtarget;
3 public:
4     M88kDAGtoDAGISel(M88kTargetMachine &TM,
5                       CodeGenOpt::Level OptLevel)
6         : SelectionDAGISel(TM, OptLevel) {}
7
8    StringRef getPassName() const override {
9         return "M88k DAG->DAG Pattern Instruction Selection";
10    }
11
12    #include "M88kGenDAGISel.inc"
13    void Select(SDNode *Node) override {
14        SelectCode(Node);
15    }
16};
```

我们还添加了工厂函数来创建这个文件中的 Pass：

```
1 FunctionPass *llvm::createM88kISelDag(M88kTargetMachine &TM,
2                                       CodeGenOpt::Level
3                                       OptLevel) {
4     return new M88kDAGtoDAGISel(TM, OptLevel);
5 }
```

现在可以实现特定于目标的操作，这些操作不能在目标描述中表示。

支持针对性的操作

转向 m88ktargetlower 类，在 M88kISelLowering.h 文件中定义。这个类配置指令 DAG 选择过程，并增强底层对特定于目标的操作。

目标描述中，我们定义了新的 DAG 节点。此文件中还定义了新类型一起使用的枚举，并继续使用最后一个预定义数字进行编号：

```
1 namespace M88kISD {
2     enum NodeType : unsigned {
```

```

3  FIRST_NUMBER = ISD::BUILTIN_OP_END,
4  RET_FLAG,
5  SET,
6 };
7 } // end namespace M88kISD

```

类需要为函数调用提供所需的降级方法。为了简单起见，只看返回值。类还可以为需要自定义处理的操作定义 LowerOperation() 钩子方法。还可以启用自定义 DAG 组合方法，为此我们定义了 PerformDAGCombine() 方法：

```

1 class M88kTargetLowering : public TargetLowering {
2     const M88kSubtarget &Subtarget;
3
4 public:
5     explicit M88kTargetLowering(const TargetMachine &TM,
6                                 const M88kSubtarget &STI);
7
8     SDValue LowerOperation(SDValue Op, SelectionDAG &DAG) const
9                             override;
10
11     SDValue PerformDAGCombine(SDNode *N, DAGCombinerInfo &DCI)
12                             const override;
13
14     SDValue LowerReturn(SDValue Chain, CallingConv::ID CallConv,
15                         bool IsVarArg,
16                         const SmallVectorImpl<ISD::OutputArg> &Outs,
17                         const SmallVectorImpl<SDValue> &OutVals,
18                         const SDLoc &DL,
19                         SelectionDAG &DAG) const override;
20 };

```

该类的实现在 m88kisellowering.cpp 中。首先，我们看看如何降级返回值：

1. 为调用规则生成的函数是需要的，因此需要包含生成的文件：

```

1 #include "M88kGenCallingConv.inc"

```

2. LowerReturn() 方法接受许多参数，这些参数都由 targetlower 超类定义。最重要的是 out 向量，保存了返回参数的描述，以及 outval 向量，还保存了返回值的 DAG 节点：

```

1 SDValue M88kTargetLowering::LowerReturn(SDValue Chain,
2     CallingConv::ID CallConv,
3     bool IsVarArg,
4     const SmallVectorImpl<ISD::OutputArg>

```



```

5     &Outs,
6     const SmallVectorImpl<SDValue> &OutVals,
7     const SDLoc &DL, SelectionDAG &DAG) const {

```

3. 我们在 CCState 类的帮助下分析返回参数，将引用传递给生成的 RetCC_M88k 函数。因此，需要对所有返回参数进行分类：

```

1     MachineFunction &MF = DAG.getMachineFunction();
2     SmallVector<CCValAssign, 16> RetLocs;
3     CCState RetCCInfo(CallConv, IsVarArg, MF, RetLocs,
4                       *DAG.getContext());
5     RetCCInfo.AnalyzeReturn(Outs, RetCC_M88k);

```

4. 对于 void 函数，没有什么可做的，返回就好。请注意，返回的节点类型是 RET_FLAG。在目标描述中将其定义为新的 ret_flag 节点：

```

1     if (RetLocs.empty())
2         return DAG.getNode(M88kISD::RET_FLAG, DL,
3                             MVT::Other, Chain);

```

5. 否则，就需要循环遍历返回参数。对于每个返回参数，都有一个 CCValAssign 类的实例，它会告诉我们如何处理参数：

```

1     SDValue Glue;
2     SmallVector<SDValue, 4> RetOps;
3     RetOps.push_back(Chain);
4     for (unsigned I = 0, E = RetLocs.size(); I != E;
5          ++I) {
6         CCValAssign &VA = RetLocs[I];
7         SDValue RetValue = OutVals[I];

```

6. 这些值可能需要提升。如果有必要，可以添加一个带有所需扩展操作的 DAG 节点：

```

1     switch (VA.getLocInfo()) {
2     case CCValAssign::SExt:
3         RetValue = DAG.getNode(ISD::SIGN_EXTEND, DL,
4                                 VA.getLocVT(), RetValue);
5         break;
6     case CCValAssign::ZExt:
7         RetValue = DAG.getNode(ISD::ZERO_EXTEND, DL,
8                                 VA.getLocVT(), RetValue);
9         break;
10    case CCValAssign::AExt:
11        RetValue = DAG.getNode(ISD::ANY_EXTEND, DL,

```

```

12         VA.getLocVT(), RetValue);
13     break;
14     case CCValAssign::Full:
15         break;
16     default:
17         llvm_unreachable("Unhandled VA.getLocInfo()");
18 }

```

7. 当值具有正确的类型时，将该值复制到寄存器中以返回该值，并与副本的链接并粘合在一起。这样循环就结束了：

```

1     Register Reg = VA.getLocReg();
2     Chain = DAG.getCopyToReg(Chain, DL, Reg, RetValue,
3                               Glue);
4     Glue = Chain.getValue(1);
5     RetOps.push_back(DAG.getRegister(Reg,
6                                       VA.getLocVT()));
7 }

```

8. 最后，我们需要更新连接和粘合：

```

1     RetOps[0] = Chain;
2     if (Glue.getNode())
3         RetOps.push_back(Glue);

```

9. 然后，返回 `re_flag` 节点，连接降级后的结果：

```

1     return DAG.getNode(M88kISD::RET_FLAG, DL,
2                        MVT::Other,
3                        RetOps);
4 }

```

为了能够调用函数，必须实现 `LowerFormalArguments()` 和 `LowerCall()` 方法。这两种方法都遵循类似的方法，因此这里没有展示。

降级配置目标

降级函数调用和参数的方法必须实现，因为它们依赖于目标。其他操作可能在目标体系结构中得到支持，也可能没有。为了让降级过程了解它，我们在 `m88ktargetlower` 类的构造函数中设置了配置：

1. 构造函数将 `TargetMachine` 和 `M88kSubtarget` 实例作为参数，并用它们初始化了相应的字段：

```

1 M88kTargetLowering::M88kTargetLowering(
2     const TargetMachine &TM, const M88kSubtarget &STI)
3 : TargetLowering(TM), Subtarget(STI) {

```

2. 首先添加所有的注册类。这里，只定义了通用寄存器，因此它只是一个简单的调用：

```
1 addRegisterClass(MVT::i32, &M88k::GPRRegClass);
```

3. 添加所有寄存器类之后，计算寄存器的派生属性，例如：由于寄存器是 32 位宽的，这个函数将 64 位数据类型标记为需要两个寄存器：

```
1 computeRegisterProperties(Subtarget.getRegisterInfo());
```

4. 还需要告诉堆栈指针使用了哪个寄存器。在 M88k 架构中，使用 r31 寄存器：

```
1 setStackPointerRegisterToSaveRestore(M88k::R31);
```

5. 还需要定义布尔值是如何表示的。基本上，就是这里使用的值是 0 和 1。其他可能的选项是只查看值的第 0 位，忽略其他所有位，并将值的所有位设置为 0 或 1：

```
1 setBooleanContents(ZeroOrOneBooleanContent);
```

6. 对于每个需要特殊处理的操作，必须调用 setOperationAction() 方法。该方法采用要采用的操作、值类型和操作作为输入。如果操作有效，则使用 Legal action 值。如果类型应该提升，则使用 Promote 操作值，如果操作导致库调用，则使用 LibCall 对值进行操作。

如果给定 Expand 操作值，则指令选择首先尝试将该操作扩展为其他操作。如果不能，则使用库调用。最后，如果使用 Custom 操作值，我们可以实现自己的操作。本例中，将为具有此操作的节点调用 LowerOperation() 方法，例如：将 CTTZ 计数末尾为零的操作设置为 Expand 操作。这个操作将使用一系列基本的位操作所替代：

```
1 setOperationAction(ISD::CTTZ, MVT::i32, Expand);
```

7. M88k 体系结构具有位字段操作，在目标描述中定义模式并不容易。这里，告诉指令选择，我们想要在或 DAG 节点上执行额外的匹配：

```
1 setTargetDAGCombine(ISD::OR);
2 }
```

根据目标体系结构，在构造函数中设置配置可能要长得多。我们只定义了最小值，忽略值，例如：浮点运算。

我们在上面标记了或操作来执行自定义组合。因此，指令选择器在调用生成的指令选择之前调用 PerformDAGCombine() 方法。这个函数会在指令选择的各个阶段调用，但通常只在操作合法化后执行匹配。常见的实现是查看操作和处理分支函数：

```
1 SDValue M88kTargetLowering::PerformDAGCombine(SDNode *N,
2 DAGCombinerInfo &DCI) const {
3     if (DCI.isBeforeLegalizeOps())
4         return SDValue();
5     switch (N->getOpcode()) {
```

```

6     default:
7         break;
8     case ISD::OR:
9         return performORCombine(N, DCI);
10 }
11 return SDValue();
12 }

```

performcombine() 方法中，尝试检查是否可以为 or 操作生成一个 set 指令。set 指令将从指定的位偏移量开始的连续位数设置为 1。这是 or 操作的一种特殊情况，第二个操作数是匹配这种格式的常量。因为 M88k 体系结构的 or 指令只能在 16 位常量上工作，所以这种匹配是有益的；否则，将合成常量，从而产生两个 or 指令。该方法使用 isShiftedMask() 帮助函数来确定常量值是否具有所需的数据。

如果第二个操作数是必需形式的常量，则该函数返回一个表示 set 指令的节点。否则，返回值 SDValue() 表示没有找到匹配的模式，需要调用生成的 DAG 模式匹配器：

```

1 SDValue performORCombine(SDNode *N,
2     TargetLowering::DAGCombinerInfo &DCI) {
3     SelectionDAG &DAG = DCI.DAG;
4     uint64_t Width, Offset;
5     ConstantSDNode *Mask =
6         dyn_cast<ConstantSDNode>(N->:getOperand(
7             1));
8     if (!Mask ||
9         !isShiftedMask(Mask->getZExtValue(), Width, Offset))
10        return SDValue();
11
12     EVT ValTy = N->getValueType(0);
13     SDLoc DL(N);
14     return DAG.getNode(M88kISD::SET, DL, ValTy,
15         N->getOperand(0),
16         DAG.getConstant(Width << 5 | Offset, DL,
17             MVT::i32));
18 }

```

为了完成整个降低过程的实现，需要实现 M88kFrameLowering 类。这个类负责处理堆栈帧，这包括生成头部和尾部代码、处理寄存器溢出等等。

对于第一个实现，可以只提供空函数。显然，为了实现完整的功能，必须实现这个类，这就完成了指令选择的实现。接下来，看看最后的指令如何产生。

生成汇编指令

前几节中实现的指令选择将 IR 指令降到 MachineInstr 实例中。这是一个非常底层的指令，但还不是机器代码。后端 Pass 中的最后一个步骤是发出指令，要么作为汇编文本，要么作为目标文件。M88kAsmPrinter 机器 Pass 负责这项任务。

基本上，这个 Pass 将一个 MachineInstr 实例降到一个 MCInst 实例，然后该实例发到一个 streamer 中。因为 MachineInstr 类没有所需的细节，所以 MCInst 类表示实际机器代码指令。

对于第一种方法，我们可以将实现限制为 emitInstruction() 方法。需要重写更多方法来支持几种操作数类型，主要是为了发出正确的重定位。这个类还负责处理内联汇编程序，如果需要，也要实现内联汇编程序。

因为 M88kAsmPrinter 类是机器函数 Pass，所以我们还重写了 getPassName() 方法。类的声明如下：

```
1 class M88kAsmPrinter : public AsmPrinter {
2 public:
3     explicit M88kAsmPrinter(TargetMachine &TM,
4                             std::unique_ptr<MCStreamer>
5                             Streamer)
6         : AsmPrinter(TM, std::move(Streamer)) {}
7    StringRef getPassName() const override
8     { return "M88k Assembly Printer"; }
9
10    void emitInstruction(const MachineInstr *MI) override;
11};
```

基本上，必须在 emitInstruction() 方法中处理两种不同的情况。MachineInstr 实例仍然可以有操作数，这不是真正的机器指令，例如：对于 RET 操作码值的返回 ret_flag 节点，就是这种情况。在 M88k 架构上，没有返回指令。所以，会跳转到 r1 寄存器中的地址存储。因此，当检测到 RET 操作码时，需要构造分支指令。默认情况下，降级只需要来自 MachineInstr 实例的信息，我们将这个任务委托给 M88kMCInstLower 类：

```
1 void M88kAsmPrinter::emitInstruction(const MachineInstr *MI) {
2     MCInst LoweredMI;
3     switch (MI->getOpcode()) {
4     case M88k::RET:
5         LoweredMI = MCInstBuilder(M88k::JMP).addReg(M88k::R1);
6         break;
7
8     default:
9         M88kMCInstLower Lower(MF->getContext(), *this);
10        Lower.lower(MI, LoweredMI);
11        break;
```

```

12 }
13 EmitToStreamer(*OutStreamer, LoweredMI);
14 }

```

M88kMCInstLower 类是没有预定义的超类。它的主要目的是处理各种操作数类型。由于目前只支持一组非常有限的操作数类型，可以将这个类简化为只有一个方法。lower() 方法设置 MCInst 实例的操作码和操作数。只处理寄存器和立即操作数，其他操作数类型将会忽略。对于完整的实现，还需要处理内存地址：

```

1 void M88kMCInstLower::lower(const MachineInstr *MI, MCInst
2 &OutMI) const {
3     OutMI.setOpcode(MI->getOpcode());
4     for (unsigned I = 0, E = MI->getNumOperands(); I != E; ++I)
5     {
6         const MachineOperand &MO = MI->getOperand(I);
7         switch (MO.getType()) {
8             case MachineOperand::MO_Register:
9                 if (MO.isImplicit())
10                     break;
11                 OutMI.addOperand(MCOperand::createReg(MO.getReg()));
12                 break;
13
14             case MachineOperand::MO_Immediate:
15                 OutMI.addOperand(MCOperand::createImm(MO.getImm()));
16                 break;
17
18             default:
19                 break;
20         }
21     }
22 }

```

汇编打印器需要一个工厂方法，该方法在初始化时调用，例如：在 InitializeAllAsmPrinters() 中初始化：

```

1 extern "C" LLVM_EXTERNAL_VISIBILITY void
2 LLVMInitializeM88kAsmPrinter() {
3     RegisterAsmPrinter<M88kAsmPrinter> X(getTheM88kTarget());
4 }

```

最后，将指令降级到真正的机器码指令，但这还没有完成。我们需要对 MC 层的实现进行补充，这将在下一节中进行讨论。

计算机编码

MC 层负责以文本或二进制形式发出机器码。大多数功能要么在各种 MC 类中实现 (只需要配置), 要么从目标描述生成实现。

MC 层的初始化在 MCTargetDesc/M88kMCTargetDesc.cpp 中进行。以下类是在 targeregistry 单例中注册的:

1. M88kMCAsmInfo: 该类提供基本信息, 如代码指针的大小、堆栈增长的方向、注释符号或汇编指令的名称。
2. M88MCInstrInfo: 该类包含指令的信息, 例如: 指令名称。
3. M88kRegInfo: 该类提供有关寄存器的信息, 例如: 寄存器的名称, 或哪个寄存器是堆栈指针。
4. M88kSubtargetInfo: 这个类保存调度模型的数据, 以及解析和设置 CPU 特性的方法。
5. M88kMCAsmBackend: 这个类提供了助手方法来获取与目标相关的重新定位数据。它还包含对象书写器类的工厂方法。
6. M88kMCInstPrinter: 该类包含以文本形式打印指令和操作数的助手方法。如果操作数在目标描述中定义了自定义打印方法, 则要在该类中实现该方法。
7. M88kMCCodeEmitter: 该类将指令的编码写入流。

根据后端实现的范围, 我们不需要注册和实现所有这些类。如果不支持文本汇编器输出, 可以忽略注册 MCInstPrinter 子类。如果不添加对写入对象文件的支持, 可以省略 MCAsmBackend 和 MCCodeEmitter 子类。

文件开始包括生成的部分和提供它的工厂方法:

```
1 #define GET_INSTRINFO_MC_DESC
2 #include "M88kGenInstrInfo.inc"
3 #define GET_SUBTARGETINFO_MC_DESC
4 #include "M88kGenSubtargetInfo.inc"
5 #define GET_REGINFO_MC_DESC
6 #include "M88kGenRegisterInfo.inc"
7
8 static MCInstrInfo *createM88kMCInstrInfo() {
9     MCInstrInfo *X = new MCInstrInfo();
10    InitM88kMCInstrInfo(X);
11    return X;
12 }
13
14 static MCRegisterInfo *createM88kMCRegisterInfo(
15     const Triple &TT) {
16     MCRegisterInfo *X = new MCRegisterInfo();
17     InitM88kMCRegisterInfo(X, M88k::R1);
18     return X;
19 }
```

```

20
21 static MCSubtargetInfo *createM88kMCSubtargetInfo(
22     const Triple &TT,StringRef CPU,StringRef
23     FS) {
24     return createM88kMCSubtargetInfoImpl(TT, CPU, FS);
25 }

```

我们还为在其他文件中实现的类提供了一些工厂方法:

```

1 static MCAsmInfo *createM88kMCAsmInfo(
2     const MCRegisterInfo &MRI, const Triple &TT,
3     const MCTargetOptions &Options) {
4     return new M88kMCAsmInfo(TT);
5 }
6
7 static MCInstPrinter *createM88kMCInstPrinter(
8     const Triple &T, unsigned SyntaxVariant,
9     const MCAsmInfo &MAI, const MCInstrInfo &MII,
10    const MCRegisterInfo &MRI) {
11    return new M88kInstPrinter(MAI, MII, MRI);
12 }

```

为了初始化 MC 层,只需要将所有的工厂方法注册到 TargetRegistry 单例中即可:

```

1 extern "C" LLVM_EXTERNAL_VISIBILITY
2 void LLVMInitializeM88kTargetMC() {
3     TargetRegistry::RegisterMCAsmInfo(getTheM88kTarget(),
4         createM88kMCAsmInfo);
5     TargetRegistry::RegisterMCCodeEmitter(getTheM88kTarget(),
6         createM88kMCCodeEmitter);
7     TargetRegistry::RegisterMCInstrInfo(getTheM88kTarget(),
8         createM88kMCInstrInfo);
9     TargetRegistry::RegisterMCRegInfo(getTheM88kTarget(),
10        createM88kMCRegisterInfo);
11    TargetRegistry::RegisterMCSubtargetInfo(getTheM88kTarget(),
12        createM88kMCSubtargetInfo);
13    TargetRegistry::RegisterMCAsmBackend(getTheM88kTarget(),
14        createM88kMCAsmBackend);
15    TargetRegistry::RegisterMCInstPrinter(getTheM88kTarget(),
16        createM88kMCInstPrinter);
17 }

```


另外，在 MCTargetDesc/M88kTargetDesc.h 头文件中，还需要包含生成的源文件的头文件部分：

```
1 #define GET_REGINFO_ENUM
2 #include "M88kGenRegisterInfo.inc"
3 #define GET_INSTRINFO_ENUM
4
5 #include "M88kGenInstrInfo.inc"
6 #define GET_SUBTARGETINFO_ENUM
7 #include "M88kGenSubtargetInfo.inc"
```

我们将注册类的源文件全部放在 MCTargetDesc 目录中。对于第一个实现，只提供这些类的部分就足够了，例如：只要不将内存地址支持添加到目标描述中，就不会生成补丁。M88kMCAsmInfo 类可以很快实现，只需要在构造函数中设置一些属性：

```
1 M88kMCAsmInfo::M88kMCAsmInfo(const Triple &TT) {
2     CodePointerSize = 4;
3     IsLittleEndian = false;
4     MinInstAlignment = 4;
5     CommentString = "#";
6 }
```

实现了 MC 层的支持类之后，就可以将机器码生成到文件中了。

下一节中，我们实现反汇编所需的类，这是反向的操作：将对象文件转换回汇编文本。

支持反汇编

目标描述中指令的定义允许构造解码器表，解码器表用于将目标文件反汇编成文本汇编程序。解码器表和解码器函数是由 llvm-tblgen 工具生成的。除了生成的代码外，我们只需要提供注册和初始化 M88kDisassembler 类的代码，以及一些解码寄存器和操作数的帮助函数。

我们将实现放在 Disassembler/M88kDisassembler.cpp 文件中。M88kDisassembler 类的 getInstruction() 方法执行解码工作。它接受一个字节数组作为输入，并将下一条指令解码到 MCInst 类的实例中。类声明如下：

```
1 using DecodeStatus = MCDisassembler::DecodeStatus;
2
3 namespace {
4
5 class M88kDisassembler : public MCDisassembler {
6 public:
7     M88kDisassembler(const MCSubtargetInfo &STI, MCContext &Ctx)
8         : MCDisassembler(STI, Ctx) {}
9     ~M88kDisassembler() override = default;
```

```

10
11 DecodeStatus getInstruction(MCInst &instr, uint64_t &Size,
12                             ArrayRef<uint8_t> Bytes,
13                             uint64_t Address,
14                             raw_ostream &CStream) const
15                             override;
16 };
17 }

```

生成的类以非限定的方式引用 DecodeStatus 枚举，因此必须使名称可见。

为了初始化反汇编器，我们定义了一个工厂函数，只是实例化一个新对象：

```

1 static MCDisassembler *
2 createM88kDisassembler(const Target &T,
3                         const MCSubtargetInfo &STI,
4                         MCContext &Ctx) {
5     return new M88kDisassembler(STI, Ctx);
6 }

```

LLVMInitializeM88kDisassembler() 函数中，将工厂函数注册到目标注册表中：

```

1 extern "C" LLVM_EXTERNAL_VISIBILITY void
2 LLVMInitializeM88kDisassembler() {
3     TargetRegistry::RegisterMCDisassembler(
4         getTheM88kTarget(), createM88kDisassembler);
5 }

```

当 LLVM 核心库初始化时，会调用 initializealldisassemblers() 函数或 InitializeNativeTargetDisassembler() 函数。

生成的解码器函数需要助手函数来解码寄存器和操作数，这些元素的编码通常涉及目标描述中没有表示的特殊情况，例如：两个指令之间的距离总是偶数，所以最小的位可以忽略，因为它总是零。

要解码寄存器，必须定义 DecodeGPRRegisterClass() 函数。32 个寄存器用 0 到 31 之间的数字进行编码，我们可以使用静态 GPRDecoderTable 表来映射编码和生成的寄存器枚举之间的关系：

```

1 static const uint16_t GPRDecoderTable[] = {
2     M88k::R0, M88k::R1, M88k::R2, M88k::R3,
3     M88k::R4, M88k::R5, M88k::R6, M88k::R7,
4     M88k::R8, M88k::R9, M88k::R10, M88k::R11,
5     M88k::R12, M88k::R13, M88k::R14, M88k::R15,
6     M88k::R16, M88k::R17, M88k::R18, M88k::R19,
7     M88k::R20, M88k::R21, M88k::R22, M88k::R23,
8     M88k::R24, M88k::R25, M88k::R26, M88k::R27,

```

```

9   M88k::R28, M88k::R29, M88k::R30, M88k::R31,
10  };
11
12  static DecodeStatus
13  DecodeGPRRegisterClass(MCInst &Inst, uint64_t RegNo,
14                        uint64_t Address,
15                        const void *Decoder) {
16      if (RegNo > 31)
17          return MCDisassembler::Fail;
18
19      unsigned Register = GPRDecoderTable[RegNo];
20      Inst.addOperand(MCOperand::createReg(Register));
21      return MCDisassembler::Success;
22  }

```

所有其他所需的解码器函数遵循与 DecodeGPRRegisterClass() 函数相同的模式:

1. 检查要解码的值是否符合所需的大小限制。如果不是，则返回 MCDisassembler::Fail。
2. 解码该值并将其添加到 MCInst 实例中。
3. 返回 MCDisassembler::Success 表示成功。

然后，需要可以包括生成的解码器表和函数:

```

1  #include "M88kGenDisassemblerTables.inc"

```

最后，可以定义 getInstruction() 方法。这个方法有两个结果值，解码的指令和指令的大小。如果字节数组太小，则必须将大小设置为 0。这很重要，因为即使解码失败，size 参数也会将指针推进到下一个内存位置。

在 M88k 架构的情况下，方法很简单，因为所有指令都是 4 个字节长。因此，从数组中提取 4 个字节后，可以调用生成的解码器函数:

```

1  DecodeStatus M88kDisassembler::getInstruction(
2      MCInst &MI, uint64_t &Size, ArrayRef<uint8_t> Bytes,
3      uint64_t Address, raw_ostream &CS) const {
4      if (Bytes.size() < 4) {
5          Size = 0;
6          return MCDisassembler::Fail;
7      }
8      Size = 4;
9
10     uint32_t Inst = 0;
11     for (uint32_t I = 0; I < Size; ++I)
12         Inst = (Inst << 8) | Bytes[I];

```

```
13     return decodeInstruction(DecoderTableM88k32, MI, Inst,  
14                             Address, this, STI);  
15 }
```

这就完成了反汇编程序的实现。

在实现了所有类之后，只需要设置构建系统来获取新的目标后端，我们将在下一节中添加它。

自定义后端的汇总

我们的新目标位于 `llvm/lib/Target/M88k` 目录中，需要集成到构建系统中。为了简化开发，我们将其作为实验目标添加到 `llvm/CMakeLists.txt` 文件中。将现有的空字符串替换为目标的名称：

```
set(LLVM_EXPERIMENTAL_TARGETS_TO_BUILD "M88k" ...)
```

我们还需要提供 `llvm/lib/Target/M88k/CMakeLists.txt` 来构建目标。除了列出目标的 C++ 文件外，还定义了从目标描述生成源文件的方法。

从目标描述生成所有类型的源码

`llvm-tblgen` 工具的不同运行方式，会生成 C++ 代码的不同部分。但是，建议将所有部件的生成添加到 `CMakeLists.txt` 文件中。这样做的原因是能提供了更好的检查，例如：如果您在指令编码方面犯了错误，那么这只会反汇编程序生成代码期间捕获。因此，即使不打算支持反汇编程序，仍值得为其生成源码。

该文件如下所示：

1. 首先，定义一个名为 `M88k` 的新 LLVM 组件：

```
add_llvm_component_group(M88k)
```

2. 接下来，命名目标描述文件，添加语句以使用 `TableGen` 生成各种源片段，并为其定义一个公共目标：

```

set(LLVM_TARGET_DEFINITIONS M88k.tdtablegen(LLVM
M88kGenAsmMatcher.inc -gen-asm-matcher)
tablegen(LLVM M88kGenAsmWriter.inc -gen-asm-writer)
tablegen(LLVM M88kGenCallingConv.inc -gen-callingconv)
tablegen(LLVM M88kGenDAGISel.inc -gen-dag-isel)
tablegen(LLVM M88kGenDisassemblerTables.inc
                                -gen-disassembler)

tablegen(LLVM M88kGenInstrInfo.inc -gen-instr-info)
tablegen(LLVM M88kGenMCCodeEmitter.inc -gen-emitter)
tablegen(LLVM M88kGenRegisterInfo.inc -gen-register-info)
tablegen(LLVM M88kGenSubtargetInfo.inc -gen-subtarget)
add_public_tablegen_target(M88kCommonTableGen)

```

3. 必须列出新组件的所有源文件:

```

add_llvm_target(M88kCodeGen
    M88kAsmPrinter.cpp M88kFrameLowering.cpp
    M88kISelDAGToDAG.cpp M88kISelLowering.cpp
    M88kRegisterInfo.cpp M88kSubtarget.cpp
    M88kTargetMachine.cpp )

```

4. 最后, 在构建中包含包含 MC 和反汇编类的目录:

```

add_subdirectory(MCTargetDesc)
add_subdirectory(Disassembler)

```

现在我们准备用新的后端目标编译 LLVM。在构建目录下, 可以运行如下代码:

```
$ ninja
```

这将检测更改的 CMakeLists.txt 文件, 再次运行配置步骤, 并编译新的后端。要检查是否一切正常, 可以运行以下命令:

```
$ bin/llc -version
```

注册目标的输出:

```
m88k          - M88k
```

噢耶! 我们完成了后端实现。下面 LLVM IR 中的 f1 函数在函数的两个参数之间执行一个按位的 AND 操作并返回结果。将其保存成 example.ll 文件:

```
target triple = "m88k-openbsd"
define i32 @f1(i32 %a, i32 %b) {
    %res = and i32 %a, %b
    ret i32 %res
}
```

运行 llc 工具，在控制台上查看生成的汇编文本：

```
$ llc < example.ll
        .text
        .file      "<stdin>"
        .globl     f1                                # -- Begin
function f1
        .align     3
        .type      f1,@function
f1:                                           # @f1
        .cfi_startproc
# %bb.0:
        and %r2, %r2, %r3
        jmp %r1
.Lfunc_end0:
        .size      f1, .Lfunc_end0-f1
        .cfi_endproc
                                           # -- End function
        .section   ".note.GNU-stack","",@progbits
```

输出是有效的 GNU 语法。对于 f1 函数，将生成和和 jmp 指令。参数在 %r2 和 %r3 寄存器中传递，这两个寄存器在 and 指令中使用。结果存储在 %r2 寄存器中，该寄存器也是返回 32 位值的寄存器。函数的返回通过分支到 %r1 寄存器中的 hold 地址实现，该地址也与 ABI 匹配。一切看起来都很好！

学习了本章中的内容，现在可以实现自己的 LLVM 后端。对于许多相对简单的 CPU，如：数字信号处理器 (DSP)，只需要实现这些功能就够用了。当然，M88k CPU 体系结构的实现还不支持该体系架构的所有特性，例如：浮点寄存器。不过，现在已经了解了 LLVM 后端开发中所有重要概念，有了这些知识，您将能够添加任何缺失的部分！

总结

本章中，学习了如何为 LLVM 开发一个新的后端目标。首先收集了所需的文档，并通过增强 Triple 类使 LLVM 了解了新的体系结构。该文档还包括 ELF 文件格式的重定位定义，并向 LLVM 添加了对此的支持。

了解了目标描述包含的不同部分，并使用它生成 C++ 代码，了解了如何实现指令选择。为了输出生成的代码，开发了一个汇编打印器，并了解了需要哪些支持类来写入目标文件。还学习了如何添加对反汇编的支持，反汇编用于将目标文件转换回汇编文本。最后，扩展了构建系统以在构建中包含新目标。

现在，您已经具备了在自己的项目中创造性地使用 LLVM 所需的一切。LLVM 生态系统是非常活跃的，一直在添加新功能，所以请务必跟上它的发展!

作为一名编译器开发人员，我很高兴能够写关于 LLVM 的文章，并在此过程中发现了一些新的特性。希望你也能在 LLVM 中玩得开心! :)