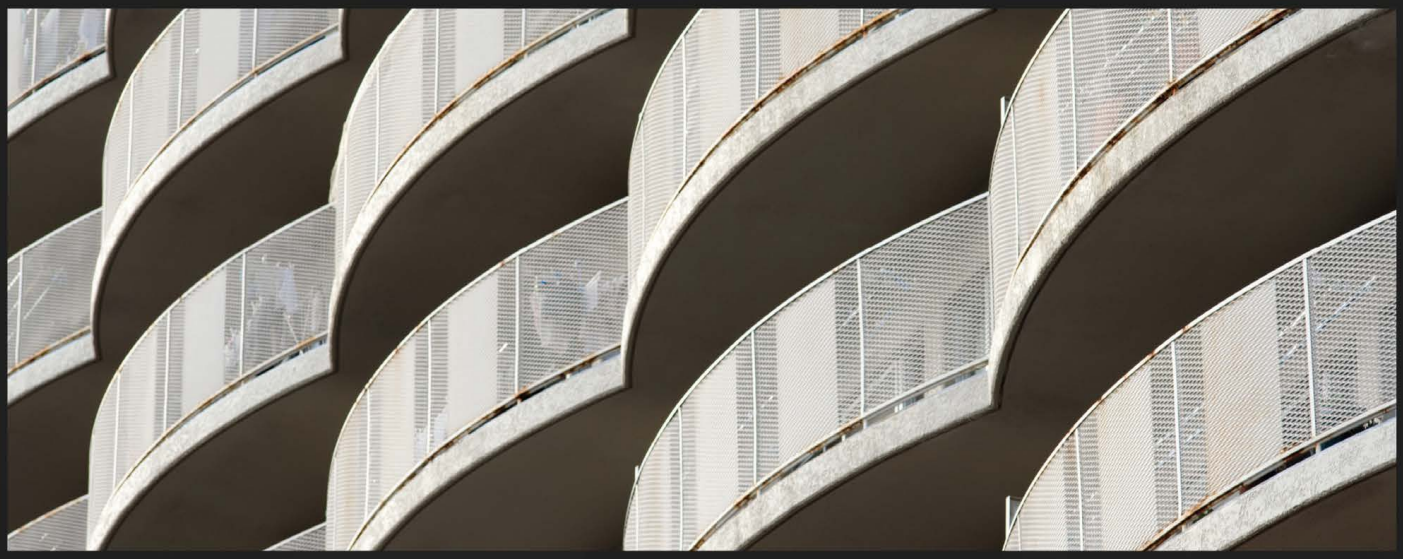
packt



2ND EDITION

Learn LLVM 17

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

KAI NACKE | AMY KWAN

Learn LLVM 17

学习 LLVM 编译器工具和核心库的初学者指南 (C++ 版)

作者: Kai Nacke 和 Amy Kwan

译者：陈晓伟

目录

作者致谢

写一本书需要很多时间和精力。没有我的妻子坦尼娅和女儿波琳娜的支持和理解，这本书是不可能完成的。谢谢你们一直鼓励我*!*

由于一些个人的挑战，这个项目处于危险之中，很感谢艾米作为作者加入这个项目。若没有她，这本书就不会像现在这样好。

再次感谢 *Packt* 的团队，不仅对我的写作提供指导，而且对我缓慢的写作表现出理解，并一直激励我坚持下去。我欠他们一个大大的感谢。

- Kai Nacke

2023 年对我来说是非常具有变革意义的一年，我将自己的 *LLVM* 知识贡献给这本书，是今年如此重要的原因之一。我从来没有想过 *Kai* 会找到我，开始这段激动人心的旅程，与大家分享 *LLVM 17 !* 感谢 *Kai*，感谢他的技术指导和指导，感谢 *Packt* 的团队，当然还有我的家人和亲人，感谢他们在我写这本书的过程中给予的支持和动力。

- Amy Kwan

关于作者

Kai Nacke 是一名专业的 IT 架构师，目前居住在加拿大多伦多，拥有德国多特蒙德技术大学 (Technical University of Dortmund) 计算机科学文凭。他关于通用散列函数的毕业论文是当时最好的论文之一。

Kai 在 IT 行业拥有超过 20 年的经验，在业务和企业应用程序的开发和架构方面拥有丰富的专业知识。在他目前的职位上，开发了一个基于 LLVM/clang 的编译器。

几年来，Kai 一直担任 LDC(基于 llvm 的 D 编译器) 的维护者。他是《D Web Development》和《Learn LLVM 12》的作者，这两本书都由 Packt 出版。过去，他还是自由和开源软件开发者欧洲会议 (FOSDEM) 的 LLVM 开发者工作室的讲师。

Amy Kwan 是一名编译器开发人员，目前居住在加拿大多伦多。Amy 来自加拿大大草原，拥有萨斯喀彻温大学计算机科学学士学位。她在职位上，利用 LLVM 技术作为后端编译器开发人员。此前，Amy 曾与 Kai Nacke 一起在 2022 年的 LLVM 开发者大会上发表演讲。

关于审稿者

Akash Kothari 是伊利诺伊 LLVM 编译器研究实验室的研究助理。他在伊利诺伊大学厄巴纳-香槟分校获得计算机科学博士学位。Akash 专注于性能工程、程序合成、形式语义和验证，他还对探索计算和编程系统的历史非常感兴趣。

Shuo Niu 拥有计算机工程硕士学位，是编译技术领域的一股活跃力量。在英特尔 PSG 专注于 FPGA HLD 编译器的五年里，领导了编译器中端优化器的创新。他在开发尖端功能方面的专业知识，使用户能够在 FPGA 板上实现显着的性能提升。

前言

构造编译器是一项复杂而迷人的任务。LLVM 项目为编译器提供了可重用的组件，LLVM 核心库实现了世界级的优化代码生成器，可以为所有主流 CPU 架构翻译与源语言无关的机器码中间表示，许多编程语言的编译器已经在使用 LLVM。

本书将介绍如何实现自己的编译器，以及如何使用 LLVM 来实现。您将了解编译器的前端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)。此外，还将探索在编译器中添加一个优化管道，可将 IR 编译为高性能的机器码。

LLVM 框架可以通过多种方式进行扩展，读者将了解如何向 LLVM 添加通道，甚至是一个全新的后端。高级主题，如编译不同的 CPU 架构和扩展 clang 和 clang 静态分析器与自己的插件和检查器也包括在内。本书遵循一种实用的方法，并附有示例源代码，读者可以在自己的项目中应用相应的代码。

新版本增加的内容

有一个新的章节，专门介绍在 LLVM 中使用的 TableGen 语言的概念和语法，读者可以利用它来定义类，记录和整个 LLVM 后端。此外，本书还介绍了后端开发的重点，其中讨论了可以为 LLVM 后端实现的各种新后端概念，例如：实现 GlobalISel 指令框架和开发机器功能通道。

适读人群

本书是为有兴趣学习 LLVM 框架的编译器开发人员、爱好者和工程师编写的。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 库的工程师。要理解本书所涵盖的概念，必须具有 C++ 中级编程经验。

本书内容

第 1 章，安装 LLVM，解释了如何设置和使用你的开发环境，了解如何自行编译 LLVM 库，以及自定义构建过程。

第 2 章，编译器的结构，对编译器组件的概述，最后可以实现第一个生成 LLVM IR 的编译器。

第 3 章，将源码文件转换为抽象语法树，详细介绍了如何实现编译器的前端。为一种小型编程语言创建前端编译器，最后构建抽象语法树。

第 4 章，生成 IR 代码的基础知识，展示了如何从抽象语法树生成 LLVM IR。将实现语言的编译器，生成汇编文本或目标代码文件。

第 5 章，高级语言结构生成的 IR，说明了如何将高级编程语言中常见的源语言特性转换为 LLVM IR。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应用程序二进制接口。

第 6 章，生成 IR 代码的进阶知识，展示了如何在源语言中为异常处理语句生成 LLVM IR。还将学习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，并扩展编译器生成的元数据。

第 7 章，优化 IR，解释了 LLVM 通道管理器。将实现自己的通道，既作为 LLVM 的一部分，也作为插件，将了解如何将通道添加到优化通道流水线中。

第 8 章，TableGen 语言，介绍了 LLVM 自己的领域特定语言 TableGen。该语言用于减少开发人员的编码工作，将了解在 TableGen 语言中定义数据的不同方法，以及如何在后端使用。

第 9 章，JIT 编译，讨论了如何使用 LLVM 实现一个即时 (JIT) 编译器。将以两种不同的方式为 LLVM IR 实现自己的 JIT 编译器。

第 10 章，使用 LLVM 工具进行调试，探讨了 LLVM 的各种库和组件的细节，可以识别应用程序中的错误，使用消毒器来识别缓冲区溢出和其他错误；使用 libFuzzer 库，使用随机数据作为输入来测试函数；XRay 将帮助程序找到性能瓶颈。可使用 clang 静态分析器在源代码级别识别错误，并且将了解到可以向分析器添加自己的检查器，还将了解如何使用自己的插件对 clang 进行扩展。

第 11 章，目标描述，解释了如何添加对新的 CPU 架构的支持。本章讨论了必要的和可选的步骤，如定义寄存器和指令，开发指令选择，以及支持汇编和反汇编程序。

第 12 章，指令选择，演示了两种不同的指令选择方法，特别解释了 SelectionDAG 和 GlobalISel 是如何工作的，并展示了如何在目标中实现这些功能，基于前一章的例子。此外，将了解如何对指令选择进行调试和测试。

第 13 章，超越指令选择，解释了如何通过探索超越指令选择的概念来完成后端实现。这包括添加新的机器通道来实现特定于目标的任务，这些主题对于简单后端来说是不必要的，但对于高度优化的后端来说是有趣的，例如：交叉编译到另一个 CPU 架构。

编译环境

您需要一台运行 Linux、Windows、Mac OS X 或 FreeBSD 的计算机，并为操作系统安装了开发工具链。所需工具请参见表格，所有工具都需要配置到 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 7.1.0 或更高版本, clang 3.0 或更高版本, Apple clang 10.0 或更高版本, Visual Studio 2019 16.7 或更高版本	Linux(any), Windows, Mac OS X 或 FreeBSD
CMake 3.20.0 或更高版本	
Ninja 1.11.1	
Python 3.6 或更高版本	
Git 2.39 或更高版本	

要创建第 10 章“使用 LLVM 工具调试”中的火焰图，需要从https://github.com/brendangregg/FlameGraph获取安装脚本。要运行安装脚本，还需要安装最新版本的 Perl。要查看图形，还需要能够显示 SVG 文件的 Web 浏览器，这是所有现代浏览器都能做到的。要在同一章中查看 Chrome Trace Viewer 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-17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!

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

我们很想听听读者们对本书的看法! 欢迎请点击这里直接进入这本书的亚马逊评论页面，分享你的反馈。

您的评论对我们和技术社区都很重要，将帮助确保书籍内容的品质。

第一部分 使用 LLVM 构建编译器的基础知识

将了解如何编译 LLVM，并根据需要定制构建；了解 LLVM 项目是如何组织的，将使用 LLVM 创建一个项目。最后，将探索编译器的总体结构，并创建一个小型编译器。

- 第 1 章，安装 LLVM
- 第 2 章，编译器的结构

第 1 章 安装 LLVM

为了学习如何使用 LLVM，最好先从源代码开始编译 LLVM。LLVM 是一个伞形项目，GitHub 库包含属于 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。除了克隆库之外，本地环境还必须安装构建系统所需的所有工具。

本章中，将学习以下主题:

- 准备环境，将展示如何设置构建系统
- 克隆库并使用源码构建，了解如何获得 LLVM 源代码，以及如何编译和安装 LLVM 核心库和 clang 与 CMake 和 Ninja
- 自定义构建过程，将讨论影响构建过程的各种可能性

1.1. 编译与直接安装 LLVM

可以使用各种源来安装 LLVM 二进制文件。若使用的是 Linux，则其发行版包含 LLVM 库。为什么要自己编译 LLVM 呢?

首先，并非所有安装包都包含使用 LLVM 进行开发所需的所有文件，自己编译和安装 LLVM 可以避免这个问题。另一个原因是，LLVM 可定制，通过构建 LLVM，将了解如何自定义 LLVM，这将使读者能够诊断将 LLVM 应用程序放到另一个平台运行时可能出现的问题。最后，本书的第三部分，将会对 LLVM 进行扩展，所以需要有能自行构建 LLVM 的能力。

但在开始使用时，完全可以避免编译 LLVM。若想要走这条路，只需要安装下一节中描述的相关工具即可。

> **Note**
>
> 许多 Linux 发行版将 LLVM 分成几个包。请确保安装了开发包。以 Ubuntu 为例，需要安装 llvm-dev 包。请确定安装了 LLVM 17。对于其他版本，本书中的示例可能需要修改。

1.2. 配置环境

要使用 LLVM，开发系统应该运行一个通用的操作系统，如 Linux、FreeBSD、macOS 或 Windows。可以在不同的模式下构建 LLVM 和 clang，启用调试符号的构建最多需要 30 GB 的空间。所需的磁盘空间在很大程度上取决于所选择的构建选项。例如，在发布模式下仅构建 LLVM 核心库，只针对一个平台，需要最少 2 GB 的可用磁盘空间。

为了减少编译时间，快速的 CPU(例如：时钟速度为 2.5 GHz 的四核 CPU) 和快速的 SSD 也很有帮助。甚至可以在小型设备 (如 Raspberry Pi) 上构建 LLVM——需要花费很多时间。本书中的示例是在一台笔记本电脑上开发的，该笔记本电脑采用 Intel 四核 CPU，时钟速度为 2.7 GHz，具有 40GB RAM 和 2.5TB SSD 磁盘空间。

您的开发系统必须安装一些必备软件，来回顾一下这些软件包的最低要求版本。

要从 GitHub 查看源代码，需要 Git(https://git-scm.com/)。GitHub 帮助页面建议至少使用 1.17.10 版本。由于过去发现的已知安全问题，建议使用最新的可用版本，在撰写本文时为 2.39.1。

LLVM 项目使用 CMake(https://cmake.org/) 作为构建文件生成器，至少为 3.20.0。CMake 可以为各种构建系统生成构建文件。本书中，使用了 Ninja(https://ninja-build.org/)，因为它速度快，并且可以在所有平台上使用，建议使用最新版本 1.11.1。

显然，还需要一个 C/C++ 编译器。LLVM 项目是基于 C++17 标准，用现代 C++ 编写的。需要一个兼容的编译器和标准库。以下编译器可以与 LLVM 17 一起工作 (已测试):

- gcc 7.1.0 或更高版本
- clang 5.0 或更高版本
- Apple clang 10.0 或更高版本
- Visual Studio 2019 16.7 或更高版本

Tip

随着 LLVM 项目的进一步发展，编译器的需求很可能会发生变化。一般来说，应该使用系统可用的最新编译器版本。

Python(https://python.org/) 用于生成构建文件和运行测试套件，至少为 3.8。

虽然本书没有涉及，但可能需要使用 Make，而非 Ninja，所以需要使用 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。对于下面描述的场景，将每个命令中的 ninja 替换为 make 就可以了。

LLVM 还依赖于 zlib 库 (https://www.zlib.net/)，至少为 1.2.3.4 版本。与往常一样，建议使用最新版本 1.2.13。

要安装必备软件，最简单的方法是从操作系统中使用包管理器。下面几节中，将为主流操作系统显示安装软件所需的命令。

1.2.1. Ubuntu

Ubuntu 22.04 使用 apt 包管理器。大多数基本的工具都已经安装好了，只缺少开发工具。要一次安装所有软件包，可以输入以下命令:

```
$ sudo apt -y install gcc g++ git cmake ninja-build zlib1g-dev
```

1.2.2. Fedora 和 RedHat

Fedora 37 和 RedHat Enterprise Linux 9 的包管理器名为 dnf。和 Ubuntu 一样，大多数基本的工具都已经安装好了。要一次安装所有软件包，可以输入以下命令:

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

1.2.3. FreeBSD

在 FreeBSD 13 或更高版本上，必须使用 pkg 包管理器。FreeBSD 与基于 linux 的系统的不同之处在于已经安装了 clang 编译器。要一次安装所有其他软件包，可以输入以下命令:

```
$ sudo pkg install -y git cmake ninja zlib-ng
```

1.2.4. OS X

在 OS X 上开发，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++ 编译器和相关工具。对于其他工具的安装，可以使用包管理器 Homebrew(https://brew.sh/)。要一次安装所有软件包，可以输入以下命令:

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

1.2.5. Windows

和 OS X 一样，Windows 没有包管理器。对于 C/C++ 编译器，需要下载个人免费使用的 Visual Studio Community 2022(https://visualstudio.microsoft.com/vs/community/)。请确保安装了名为 Desktop Development with C++ 的工作负载。可以使用包管理器 Scoop(https://scoop.sh/) 来安装其他包。按照网站上的描述安装 Scoop 之后，从 Windows 菜单中打开 VS 2022 的 x64 Native Tools Command Prompt。要安装所需的软件包，输入以下命令:

```
$ scoop install git cmake ninja python gzip bzip2 coreutils
$ scoop bucket add extras
$ scoop install zlib
```

请密切关注 Scoop 的输出。对于 Python 和 zlib 包，建议添加一些注册表项。其他软件需要这些条目才能找到这些软件包。要添加注册表项，最好复制并粘贴来自 Scoop 的输出，如下所示:

```
$ %HOMEPATH%\scoop\apps\python\current\install-pep-514.reg
$ %HOMEPATH%\scoop\apps\zlib\current\register.reg
```

每个命令之后，注册表编辑器将弹出一个消息窗口，询问是否真的要导入这些注册表项，需要单击 Yes 以完成导入。现在，已经安装了所有所需的工具。

对于本书中的所有示例，必须在 VS 2022 中使用 x64 本机工具命令提示符。使用此命令提示符，编译器将自动添加到搜索路径中。

Tip

LLVM 代码库非常大。为了方便地导航源代码，建议使用一个 IDE，可以跳转到类的定义，并搜索源代码。我们发现 Visual Studio Code(https://code.visualstudio.com/download) 是一个可扩展的跨平台 IDE，使用起来非常舒服，但这不是运行本书示例的必要条件。

1.3. 使用代码库源码进行构建

准备好构建工具后，可以从 GitHub 中下载 LLVM 项目，并构建 LLVM。这一过程在所有平台上都是相同的:

1. 配置 Git
2. 克隆库
3. 创建构建目录
4. 生成构建系统文件
5. 最后，编译和安装 LLVM

先从配置 Git 开始吧！

1.3.1. 配置 Git

LLVM 项目使用 Git 进行版本控制。若以前没有使用过 Git，在继续之前应该先做一些 Git 的基本配置: 设置用户名和电子邮件地址。提交修改，将使用这两个信息。

可以使用以下命令检查是否已经在 Git 中配置了电子邮件和用户名:

```
$ git config user.email
$ git config user.name
```

前面的命令将输出在使用 Git 时已经设置的电子邮件和用户名，但若是第一次设置用户名和电子邮件，可以输入以下命令进行第一次配置。以下命令中，可以简单地将 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"
```

这些命令更改全局 Git 配置。在 Git 存储库中，可以不指定--global 选项在本地覆盖这些值。

默认情况下，Git 使用 vi 编辑器提交消息。若喜欢其他编辑器，可以以类似的方式更改配置。例如：要使用 nano 编辑器，可以输入以下命令:

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

有关 Git 的更多信息，请参阅 Git 版本控制手册 (https://www.packtpub.com/product/git-version-control-cookbook-secondedition/9781789137545)。

现在可以从 GitHub 克隆 LLVM 了。

1.3.2. 克隆库

克隆库的命令在所有平台上基本上是相同的。仅在 Windows 上，建议关闭行结束符的自动翻译。

在所有非 windows 平台上，输入以下命令克隆库:

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

仅在 Windows 上，添加禁用行结束符自动翻译的选项，可输入以下内容:

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

这个 Git 命令将最新的源代码从 GitHub 克隆到一个名为 llvm-project 的本地目录中。现在使用以下命令，将当前目录更改为 llvm-project 目录:

```
$ cd llvm-project
```

目录中是所有 LLVM 项目，每个项目都在自己的目录中。最值得注意的是，LLVM 核心库位于 LLVM 子目录中。LLVM 项目使用分支用于后续版本开发 (“release/17.x”) 和标签 (“llvmg -17.0.1”) 来标记某个版本。使用前面的 clone 命令，可以获得当前的开发状态。本书使用 LLVM 17。要将 LLVM 17 的第一个版本检出到一个名为 llvm-17 的分支中，可以输入以下命令:

```
$ git checkout -b llvm-17 llvmorg-17.0.1
```

通过前面的步骤，就克隆了整个库，并从标记创建了分支。这是最灵活的方法。

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 release/X https://github.com/llvm/llvm-project，只克隆版本 release/17.x 分支及其历史。然后，就可以看到 LLVM 17 发布分支的最新状态。若需要确切的发布版本，只需要像之前一样从发布标签创建一个分支。

使用--depth=1 选项，就是 Git 的浅克隆，可以避免下载太多历史记录。这节省了时间和空间，但显然限制了在本地可以做的事情，包括根据发布标记签出分支。

1.3.3. 创建构建目录

LLVM 不支持内联构建，并且需要一个单独的构建目录。最简单的方法是在 llvm-project 目录中创建，这是当前目录。简单起见，将构建目录命名为 build。这里，Unix 和 Windows 系统的命令不同。在类 Unix 系统上，可以使用以下命令:

```
$ mkdir build
```

在 Windows 上，使用以下命令:

```
$ md build
```

现在，就已经准备好使用该目录下的 CMake 工具创建构建系统文件了。

1.3.4. 生成构建系统文件

为了使用 Ninja 生成编译 LLVM 和 clang 的构建系统文件，可以运行以下命令:

```
$ cmake -G Ninja -DCMAKE_BUILD_TYPE=Release \
  -DLLVM_ENABLE_PROJECTS=clang -B build -S llvm
```

-G 选项告诉 CMake 为哪个系统生成构建文件。该选项的常用值如下:

- Ninja –for the Ninja build system
- Unix Makefiles –for GNU Make
- Visual Studio 17 VS2022 –for Visual Studio and MS Build
- Xcode –for Xcode projects

使用-B 选项，告诉 CMake 构建目录的路径。类似地，可以使用-S 选项指定源目录。可以通过使用-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) 为前缀。

如前所述，我们也对与 LLVM 一起编译 clang 感兴趣。使用 LLVM_ENABLE_PROJECTS=clang，允许 CMake 除了为 LLVM 生成构建文件外，还为 clang 生成构建文件。此外，CMAKE_BUILD_TYPE=Release 变量告诉 CMAKE 它应该为“发布”构建生成构建文件。

-G 选项的默认值取决于平台，而构建类型的默认值取决于工具链，但可以使用环境变量定义自己的首选项。CMAKE_GENERATOR 变量控制生成器，CMAKE_BUILD_TYPE 变量指定构建类型。若使用 bash 或类似的 shell，可以这样设置变量:

```
$ export CMAKE_GENERATOR=Ninja
$ export CMAKE_BUILD_TYPE=Release
```

若使用的是 Windows 命令提示符，可以这样设置变量:

```
$ set CMAKE_GENERATOR=Ninja
$ set CMAKE_BUILD_TYPE=Release
```

有了这些设置，创建构建系统文件的命令就变成了下面这样，这样更容易输入:

```
$ cmake -DLLVM_ENABLE_PROJECTS=clang -B build -S llvm
```

将在自定义构建过程一节中找到更多关于 CMake 变量的信息。

1.3.5. 编译和安装 LLVM

生成构建文件后，可以使用以下命令编译 LLVM 和 clang:

```
$ cmake --build build
```

我们告诉 CMake 在配置步骤中生成 Ninja 文件，所以这个命令在底层运行 Ninja。但若为支持多构建配置的生成器 (如 Visual Studio) 生成构建文件，则需要使用--config 选项指定要用于构建的配置。根据硬件资源的不同，该命令的运行时间从 15 分钟 (具有大量 CPU 内核、内存和快速存储的服务器) 到几个小时 (具有有限内存的双核 Windows 笔记本) 不等。

默认情况下，Ninja 使用所有可用的 CPU 内核。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 操作系统的笔记本电脑上，几乎不可能在运行 Ninja 时上网。幸运的是，可以使用-j 选项限制资源使用。

假设有四个可用的 CPU 内核，而 Ninja 应该只使用两个 (因为你有并行任务要运行)。就使用以下命令进行编译:

```
$ cmake --build build -j2
```

编译完成后，运行测试检查是否一切都如预期的那样工作:

```
$ cmake --build build --target check-all
```

同样，该命令的运行时随着可用硬件资源的不同而变化很大。checkall Ninja 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 会运行 LLVM 测试，但不会运行 clang 测试，checkllvm-codegen 仅从 LLVM(即 llvm/test/CodeGen 目录) 运行 CodeGen 目录中的测试。

也可以进行快速手动检查。LLVM 应用程序之一是 LLVM 编译器 llc。若使用-version 选项运行，会显示 LLVM 的版本，主机 CPU 和所有支持的架构:

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

若在编译 LLVM 时遇到困难，应该查阅入门 LLVM 系统文档 (https://releases.llvm.org/17.0.1/docs/GettingStarted.html#common-problems) 中的常见问题部分，了解常见问题的解决方案。

最后一步，安装二进制文件:

```
$ cmake --install build
```

类 Unix 系统上，安装目录是/usr/local。在 Windows 上，使用 C:\Program Files\LLVM。这可以修改，下一节将解释如何实现。

1.4. 自定义构建

CMake 系统使用 CMakeLists.txt 文件中的项目描述。顶层文件位于 llvm 目录“llvm/CMakeLists.txt” 。其他目录也有 CMakeLists.txt 文件，这些文件在生成过程中递归包含。

根据项目描述中提供的信息，CMake 检查安装了哪些编译器，检测库和符号，并创建构建系统文件，例如 build.ninja 或 Makefile(取决于所选择的生成器)。也可以定义可重用的模块，例如: 检测

LLVM 是否已安装的函数。这些脚本放在特殊的 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 用于将变量设置为一个值，这些变量在 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 定义的变量以 LLVM_为前缀，但只能在项目定义中包含 LLVM 时使用。

1.4.1. 可定义的 CMake 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，定义了用于构建的 C 和 C++ 编译器。CMake 尝试使用当前 shell 搜索路径自动定位 C 和 C++ 编译器，会选择找到的第一个编译器。若本地安装了多个编译器，例如 gcc 和 clang 或不同版本的 clang，那么这可能不是您构建 LLVM 所需的编译器。

假设使用 clang17 作为 C 编译器，clang++17 作为 C++ 编译器。可以在 Unix shell 中调用 CMake，方法如下:

```
$ CC=clang17 CXX=clang++17 cmake -B build -S llvm
```

这只会为 cmake 的调用设置环境变量的值，可以为编译器可执行文件指定一个绝对路径。

CC 是 CMAKE_C_COMPILER CMAKE 变量的默认值，CXX 是 CMAKE_CXX_COMPILER CMAKE 变量的默认值。可以直接设置 CMake 变量，而不是使用环境变量:

```
$ cmake -DCMAKE_C_COMPILER=clang17 \
  -DCMAKE_CXX_COMPILER=clang++17 -B build -S llvm
```

CMake 定义的其他相关变量如下所示:

变量名	功能
CMAKE_INSTALL_PREFIX	这是安装过程中每个安装路径的前缀。Unix 上默认为/usr/local 和 Windows 上默认为 C:\Program Files\<Project>。要在/opt/llvm 目录下安装 LLVM，需要设置-DCMAKE_INSTALL_PREFIX=/opt/llvm。二进制文件将复制到/opt/llvm/bin 文件夹下，库文件将复制到 /opt/llvm/lib 文件夹下，以此类推。

CMAKE_BUILD_TYPE	不同类型的构建需要不同的设置。例如，调试构建需要指定生成调试符号的选项，通常链接到系统库的调试版本。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，例如:Ninja 或 Make。对于 IDE 构建系统，将生成所有配置，并且需要使用 IDE 在构建类型之间进行切换。取值范围如下所示: DEBUG: 构建带有调试符号 RELEASE: 构建优化运行速度 RELWITHDEBINFO: 构建发布版本，但带有调试符号 MINSIZEREL: 构建会生成尺寸最小的二进制文件 (也是一种优化) 默认构建类型由 CMAKE_BUILD_TYPE 变量设置。若未设置此变量，则默认值取决于所使用的工具链，并且通常为空。为了生成发布版本的构建，可以设置-DCMAKE_BUILD_TYPE=RELEASE。
CMAKE_C_FLAGS CMAKE_CXX_FLAGS	这些是编译 C 和 C++ 源文件时使用的编译选项，初始值取自 CFLAGS 和 CXXFLAGS 环境变量。
CMAKE_MODULE_PATH	指定 CMake 模块搜索的其他目录。在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	若没有找到 Python 解释器，或者在安装了多个版本的情况下选择了错误的解释器，则可以将此变量设置为 Python 二进制文件的路径。这个变量只有在包含 CMake 的 Python 模块时才有效 (LLVM 就是这种情况)。

表 1.1 - CMake 定义的其他相关变量

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。可以输入以下命令来获取 CMAKE_BUILD_TYPE 的信息:

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

也可以用下面的命令列出所有的变量:

```
$ cmake --help-variable-list
```

这个清单很长，可能将输出管道传输到相应的处理程序中会更合适。

1.4.2. 使用 LLVM 定义的构建配置变量

除了没有内置帮助之外，LLVM 定义的构建配置变量的工作方式与 CMake 定义的相同。最有用的变量如下表所示，对首次安装 LLVM 的用户有用的变量和对更高级的 LLVM 用户有用的变量。

变量名	功能
LLVM_TARGETS_TO_BUILD	LLVM 支持不同 CPU 架构的代码生成。默认情况下，所有这些目标都已构建。使用此变量指定要构建的目标列表，以分号分隔。目前支持的目标有 AArch64、AMDGPU、ARM、AVR、BPF、Hexagon、Lanai、LoongArch、Mips、MSP430、NVPTX、PowerPC、RISCV、Sparc、SystemZ、VE、WebAssembly、X86 和 XCore。 all 可以作为所有目标的简写。名称区分大小写。要只启用 PowerPC 和 SystemZ 目标，可以指定-DLLVM_TARGETS_TO_BUILD=”PowerPC;SystemZ”。
LLVM_EXPERIMENTAL_TARGETS_TO_BUILD	除了官方支持的架构外，LLVM 源代码树还包含实验目标。这些目标还在开发中，通常还不支持后端的全部功能。目前的实验得架构有 ARC、CSKY、DirectX、M68k、SPIRV 和 Xtensa。要构建 M68k 目标，可以设置-DLLVM_EXPERIMENTAL_TARGETS_TO_BUILD=M68k。
LLVM_ENABLE_PROJECTS	这是要构建的项目列表，以分号分隔。项目的源代码必须与 llvm 目录在同一级 (并排布局)。当前的列表是 bolt、clang、clang-tools-extra、compiler-rt、cross-project-tests、libc、libclc、lld、lldb、mlir、openmp、polly 和 pstl。 all 可以用作此列表中所有项目的简写，可以在这里指定 flang 项目。由于一些特殊的构建需求，它还不是 all 列表的一部分。 要将 clang 和 bolt 与 LLVM 一起构建，可以设置-DLLVM_ENABLE_PROJECT=”clang;bolt”。

表 1.2 - 对于首次安装 LLVM 的用户有用的变量

变量名	功能
LLVM_ENABLE_ASSERTIONS	若设置为 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 (例如，对于 RELEASE 版本)，可以设置 -DLLVM_ENABLE_ASSERTIONS=ON。
LLVM_ENABLE_EXPENSIVE_CHECKS	这将启用一些大开销的检查，这些检查可能会减慢编译速度或消耗大量内存，默认值为 OFF。要打开这些检查，可以指定-DLLVM_ENABLE_EXPENSIVE_CHECKS=ON。
LLVM_APPEND_VC_REV	若使用-version 命令行选项，llc 等 LLVM 工具除了显示其他信息外，还会显示它们所基于的 LLVM 版本。该版本信息基于 C 代码中的 LLVM_REVISION 宏。默认情况下，不仅 LLVM 版本，而且当前 Git 哈希值也是版本信息的一部分。若正在跟踪主分支的开发，就能清楚地表明该工具基于哪个 Git 提交。若不需要，则可以使用 -DLLVM_APPEND_VC_REV=OFF 关闭此功能。
LLVM_ENABLE_THREADS	若检测到线程库 (通常是 pthread 库)，LLVM 会自动包含线程支持。在这种情况下，LLVM 会假定编译器支持 TLS (线程本地存储)。若不想要线程支持或者本地编译器不支持 TLS，那么可以设置-DLLVM_ENABLE_THREADS=OFF 关闭该功能。
LLVM_ENABLE_EH	LLVM 项目不使用 C++ 异常处理，因此在默认情况下关闭异常支持。此设置可能与项目所链接的其他库不兼容。若需要，可以指定-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系统来获取运行时类型信息，C++ RTTI 的生成在默认情况下是关闭的。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，可以设置-DLLVM_ENABLE_RTTI=ON。
LLVM_ENABLE_WARNINGS	可能的话，编译 LLVM 应该不会生成任何警告消息，所以打印警告消息的选项在默认情况下是打开的。要关闭它，可以设置-DLLVM_ENABLE_WARNINGS=OFF。
LLVM_ENABLE_PEDANTIC	LLVM 源代码应符合 C/C++ 语言标准，默认情况下启用对源代码进行严格检查。若可能的话，还会禁用特定于编译器的扩展。要关闭此设置，可以设置 -DLLVM_ENABLE_PEDANTIC=OFF。

LLVM_ENABLE_WERROR	若设置为 ON，那么所有警告都被视为错误——发现警告，编译就会中止。这有助于在源代码中找到所有剩余的警告。默认情况下，它是关闭的。要打开它，可以设置 -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 是用来生成大部分代码的生成器。因此，tablegen 在调试构建中要慢得多，显著增加了编译时间。若将此选项设置为 ON，则即使在调试构建中，tablegen 也会在编译时打开优化，这可能会减少编译时间。默认为 OFF。要打开它，可以指定 -DLLVM_OPTIMIZED_TABLEGEN=ON。
LLVM_USE_SPLIT_DWARF	若构建编译器是 gcc 或 clang，那么打开该选项将指示编译器在单独的文件中生成 DWARF 调试信息。目标文件大小的减小，减少了调试构建的链接时间，默认为 OFF。要打开它，可以设置-DLLVM_USE_SPLIT_DWARF=ON。

表 1.3 - LLVM 进阶用户的变量

Note

LLVM 定义了更多的 CMake 变量，可以在关于 CMake 的 LLVM 文档https://releases.llvm.org/17.0.1/docs/CMake.html#llvm-specific-variables中找到完整的列表。前面的列表只包含最可能需要的部分。

1.5. 总结

本章中，准备了开发机器来编译 LLVM，克隆了 GitHub 库，并编译了 LLVM 和 clang 版本。构建过程可以用 CMake 变量自定义，了解了有用的变量以及如何更改它们。有了这些知识，就可以根据自己的需要调整 LLVM 了。

下一节中，我们将进一步了解编译器的结构，将探索编译器内部的不同组件，以及其中发生的不同类型的分析——特别是词法、语法和语义分析。最后，还会简要介绍使用 LLVM 后端进行代码生成的接口。

第 2 章 编译器的结构

编译器技术是计算机科学中一个研究领域，其任务是将源语言翻译成机器码。通常，此任务分为三个部分: 前端、中端和后端。前端主要处理源语言，而中端执行转换并改进代码，后端负责生成机器码。由于 LLVM 核心库提供了中端和后端，我们将在本章中专注于前端的介绍。

本章中，将学习以下主题:

- 编译器的构建块，将了解编译器中的常用组件
- 算术表达式语言，将介绍一种示例语言，并展示如何使用语法来定义语言
- 词法分析，讨论如何实现语言的词法分析器
- 语法分析，包括从语法构造解析器
- 语义分析，将了解如何实现语义检查
- 使用 LLVM 后端生成代码，讨论如何与 LLVM 后端进行接口，并将前面的所有阶段粘合在一起，创建完整的编译器

2.1. 编译器的构建块

自从有了计算机，编程语言就开发了数千种。事实证明，所有编译器都必须解决相同的任务，并且编译器的实现最好根据这些任务进行结构化。总得来说，有三个组成部分。前端将源代码转换为中间表示 (IR)；中端在 IR 上执行转换，其目标是提高性能或减少代码的大小；后端将 IR 生成为机器码。LLVM 核心库为所有主流平台提供了一个由复杂转换和后端组成的中端。

此外，LLVM 核心库还定义了一个中间表示，用作中间端和后端的输入。这种设计的优点是，只需要关心编程语言的前端即可。

前端的输入是源代码，通常是一个文本文件。为了使它更有意义，前端首先标识语言的单词，例如数字和标识符，通常称为标记。这一步由词法分析器执行。其次，分析了符号构成的句法结构。所谓的解析器执行这个步骤，结果是生成了抽象语法树 (AST)。

最后，前端需要由语义分析器检查，代码是否遵守编程语言的规则。若没有检测到错误，则将 AST 转换为 IR，并移交给中端处理。

下面几节中，将为表达式语言构造一个编译器，该编译器将根据其输入生成 LLVM IR。LLVM llc 静态编译器表示后端，将 IR 编译成目标代码，这一切都从定义语言开始。切记，本章中所有文件的 C++ 实现，都将包含在 src/目录中。

2.2. 算术表达式语言

算术表达式是每一种编程语言的一部分。下面是一个算术表达式计算语言 calc 的例子，calc 表达式会编译成一个计算以下表达式的应用程序:

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

表达式中使用的变量必须用关键字 with 声明。该程序会编译成一个应用程序，该程序向用户询问 a 和 b 变量的值，并输出结果。

示例总是简单并容易理解的，但作为编译器作者，需要比这更全面的实现和测试规范。首先，编程语言的载体是语法。

2.2.1. 描述程序设计语言语法的形式

语言中的元素，例如关键字、标识符、字符串、数字和操作符，称为标记 (token)。从这个意义上说，程序是一个标记序列，语法指定了哪些序列是有效的。

通常，语法以扩展的 Backus-Naur 形式 (EBNF) 编写。语法规则有左边和右边，左边只有一个符号，叫做非终结符；右侧由非终结符、记号和用于替代和重复的元符号组成。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-zAZ])+ ;
number : ([0-9])+ ;
```

第一行中，calc 是非终结符。若没有特别说明，则语法的第一个非终结符是开始符号。冒号 (:) 是规则左右两边之间的分隔符，” with”、” 和”:” 是表示这个字符串的标记，括号用于分组。组可以是可选的，也可以是重复的。右括号后的问号 (?) 表示一个可选的组。星号 * 表示零次或多次重复，加号 + 表示一次或多次重复。ident 和 expr 是非终结符，处理它们使用的是另一种规则。分号 (;) 表示规则的结束，第二行中的管道 | 表示另一种选择。最后，最后两行的括号 [] 表示字符类。有效字符写在括号内。例如，字符类 [a-zA-Z] 匹配一个大写或小写字母，而 ([a-zA-Z])+ 匹配这些字母中的一个或多个。看上去，这就是一个正则表达式。

2.2.2. 语法如何帮助编译器作者?

这样的语法看起来像一个玩具，但对编译器作者很有意义。首先，定义所有标记，这是创建词法分析器所需的，语法规则可以转化为解析器。若有解析器是否正确工作的问题，则语法就可以作为标准进行衡量。

然而，语法并不能定义编程语言的所有方面。语法的含义——语义——也必须定义。为此目的的形式也开发出来了，因为是在最初引入该语言时拟定的，所以通常在纯文本中指定。

了解了这些知识，接下来的两节将介绍词法分析如何将输入转换为标记序列，以及语法如何使用 C++ 进行语法分析。

2.3. 词法分析

正如在前一节的示例，编程语言由许多元素组成，如关键字、标识符、数字、操作符等。词法分析器的任务是获取文本输入并从中创建一个标记序列。calc 语言由，:，+，-，*，/，(，) 和正则表达式 ([a- za -z])+(标识符) 和 ([0-9])+(数字) 组成。需要为每个标记分配唯一编号，以使标记更容易处理。

2.3.1. 手写词法分析器

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

```
#ifndef LEXER_H
#define LEXER_H

#include "llvm/ADT/StringRef.h"
#include "llvm/Support/MemoryBuffer.h"
```

llvm::MemoryBuffer 类提供对内存块的只读访问，内存块由文件的内容填充，在缓冲区的末尾添加一个尾零字符 ('\x00')。我们使用这个特性来读取缓冲区，而不必在每次访问时检查缓冲区的长度。StringRef 类封装了一个指向 C 字符串及其长度的指针。因为对长度进行了保存，所以字符串不需要像普通的 C 字符串那样以零字符 ('\x00') 结束。StringRef 实例允许指向由 MemoryBuffer 管理的内存。

了解了这一点后，我们开始实现 Lexer 类:

1. 首先，Token 类包含前面提到的唯一标记编号的枚举定义:

```
class Lexer;

class Token {
    friend class Lexer;

public:
    enum TokenKind : unsigned short {
        eoi, unknown, ident, number, comma, colon, plus,
        minus, star, slash, l_paren, r_paren, KW_with
    };
```

除了为每个标记定义成员外，还添加了两个值:eoi 和 unknown。eoi 表示输入结束，当处理完输入的所有字符时返回。unknown 在词法级别发生错误时使用，例如，# 不是该语言的标记，将映射为 unknown。

2. 除了枚举之外，该类还有一个 Text 成员，该成员指向标记文本的开头，使用了前面提到的 StringRef 类:

```
private:
    TokenKind Kind;
    llvm::StringRef Text;

public:
    TokenKind getKind() const { return Kind; }
    llvm::StringRef getText() const { return Text; }
```

这对于语义处理很有意义，例如：对于标识符，了解名称很有用。

3. is() 和 isOneOf() 方法用于测试标识是否属于某种类型。isOneOf() 方法使用可变的模板，允许可变数量的参数:

```
    bool is(TokenKind K) const { return Kind == K; }
    bool isOneOf(TokenKind K1, TokenKind K2) const {
        return is(K1) || is(K2);
    }
    template <typename... Ts>
    bool isOneOf(TokenKind K1, TokenKind K2, Ts... Ks) const {
        return is(K1) || isOneOf(K2, Ks...);
    }
};
```

4. 头文件中，Lexer 类本身也有类似的接口:

```
class Lexer {
    const char *BufferStart;
    const char *BufferPtr;

public:
    Lexer(const llvm::StringRef &Buffer) {
        BufferStart = Buffer.begin();
        BufferPtr = BufferStart;
    }

    void next(Token &token);

private:
    void formToken(Token &Result, const char *TokEnd,
                   Token::TokenKind Kind);
};
#endif
```

除了构造函数之外，公共接口只有 next() 方法，该方法返回下一个标识。该方法的作用类似于迭代器，总是前进到下一个可用标识。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，假设缓冲区以 0 结束 (就像 C 字符串一样)。

5. 在 Lexer.cpp 文件中实现 Lexer 类，从一些帮助函数开始分类字符:

```
#include "Lexer.h"
namespace charinfo {
    LLVM_READNONE inline bool isWhitespace(char c) {
        return c == ' ' || c == '\t' || c == '\f' ||
        c == '\v' ||
        c == '\r' || c == '\n';
    }
    LLVM_READNONE inline bool isDigit(char c) {
        return c >= '0' && c <= '9';
    }
    LLVM_READNONE inline bool isLetter(char c) {
        return (c >= 'a' && c <= 'z') ||
        (c >= 'A' && c <= 'Z');
```

```
}
}
```

这些函数使条件更具有可读性。

> **Note**
>
> 没有使用 <cctype> 标准库头文件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言环境改变行为。例如，区域设置是德语区域，那么德语的变音符可以分类为字母。这在编译器中通常是不需要的。其次，由于函数的形参类型为 int，因此需要从 char 类型进行转换。这种转换的结果取决于 char 是视为有符号类型还是无符号类型，这会导致移植性问题。

6. 前一节对语法的了解中，知道了该语言的所有符号，但语法没有定义应该忽略的字符。例如，空格或换行字符只添加空白，通常应该忽略。next() 一开始就会忽略这些字符:

```
void Lexer::next(Token &token) {
    while (*BufferPtr &&
    charinfo::isWhitespace(*BufferPtr)) {
        ++BufferPtr;
    }
```

7. 接下来，确保仍然有字符需要处理:

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

至少有一个字符需要处理。

8. 首先检查字符是小写还是大写。所以，标记要么是标识符，要么是 with 关键字，因为标识符的正则表达式也匹配关键字。最常见的解决方案是，收集正则表达式匹配的字符，并检查字符串是否恰好是关键字:

```
    if (charinfo::isLetter(*BufferPtr)) {
        const char *end = BufferPtr + 1;
        while (charinfo::isLetter(*end))
            ++end;
        llvm::StringRef Name(BufferPtr, end - BufferPtr);
        Token::TokenKind kind =
            Name == "with" ? Token::KW_with : Token::ident;
        formToken(token, end, kind);
        return;
    }
```

formToken() 私有方法用于填充标记。

9. 接下来，检查数字。与前面的代码片段非常相似:

```
    else if (charinfo::isDigit(*BufferPtr)) {
        const char *end = BufferPtr + 1;
```

```
        while (charinfo::isDigit(*end))
            ++end;
        formToken(token, end, Token::number);
        return;
    }
```

现在只剩下由固定字符串定义的标记。

10. 这很容易用 switch 完成。由于所有这些标记都只有一个字符，因此使用 CASE 预处理器宏来减少编码量:

```
else {
    switch (*BufferPtr) {
        #define CASE(ch, tok) \
        case ch: formToken(token, BufferPtr + 1, tok); break
        CASE('+', Token::plus);
        CASE('-', Token::minus);
        CASE('*', Token::star);
        CASE('/', Token::slash);
        CASE('(', Token::Token::l_paren);
        CASE(')', Token::Token::r_paren);
        CASE(':', Token::Token::colon);
        CASE(',', Token::Token::comma);
        #undef CASE
```

11. 最后，需要检查不支持的字符:

```
        default:
            formToken(token, BufferPtr + 1, Token::unknown);
        }
        return;
    }
}
```

只有 formToken() 私有辅助方法仍然缺失。

12. 填充 Token 实例的成员，并更新指向下一个未处理字符的指针:

```
void Lexer::formToken(Token &Tok, const char *TokEnd,
                      Token::TokenKind Kind) {
    Tok.Kind = Kind;
    Tok.Text = llvm::StringRef(BufferPtr,
                               TokEnd - BufferPtr);
    BufferPtr = TokEnd;
}
```

下一节中，我们将了解如何构造用于语法分析的解析器。

2.4. 语法分析

语法分析由解析器完成，接下来我们将实现解析器。它的基础是前面小节中的语法和词法分析器。解析过程的结果是一个动态数据结构，称为抽象语法树 (AST)。AST 是输入的一个非常紧凑的表示，非常适合语义分析。

首先，将实现解析器，再了解 AST 的解析过程。

2.4.1. 手写解析器

解析器的接口在头文件 Parser.h 中定义，以一些 include 声明开始:

```
#ifndef PARSER_H
#define PARSER_H

#include "AST.h"
#include "Lexer.h"
#include "llvm/Support/raw_ostream.h"
```

AST.h 头文件声明 AST 的接口，将在后面展示。LLVM 的编码指南禁止使用 <iostream> 库，所以包含了等效 LLVM 功能的头文件。这里需要发出一个错误消息:

1. Parser 类首先声明一些私有成员:

```
class Parser {
    Lexer &Lex;
    Token Tok;
    bool HasError;
```

Lex 和 Tok 是前一节中类的实例。Tok 存储下一个标记 (预检)，Lex 用于从输入中检索下一个标记。HasError 标志表示是否检测到错误。

2. 有几个方法处理这个标记:

```
    void error() {
        llvm::errs() << "Unexpected: " << Tok.getText()
        << "\n";
        HasError = true;
    }

    void advance() { Lex.next(Tok); }

    bool expect(Token::TokenKind Kind) {
        if (Tok.getKind() != Kind) {
            error();
            return true;
        }
        return false;
    }

```

```
    bool consume(Token::TokenKind Kind) {
        if (expect(Kind))
            return true;
        advance();
        return false;
    }
```

advance() 从词法分析器检索下一个标记。Expect() 测试预检是否具有预期的类型，否则发出错误消息。最后，若预检具有预期的类型，则 consume() 将检索下一个标记。若发出错误消息，则将 HasError 标志设置为 true。

3. 对于语法的每个非终结符，声明一个解析规则的方法:

```
    AST *parseCalc();
    Expr *parseExpr();
    Expr *parseTerm();
    Expr *parseFactor();
```

> **Note**
>
> 没有标识和编号的方法。这些规则只返回标记，并用相应的标记替换。

4. 接下来是公共接口。构造函数初始化所有成员，并从词法分析器获取第一个标记:

```
public:
    Parser(Lexer &Lex) : Lex(Lex), HasError(false) {
        advance();
    }
```

5. 需要一个函数来获取错误标志的值:

```
    bool hasError() { return HasError; }
```

6. 最后，parse() 方法是解析的主要入口:

```
    AST *parse();
};

#endif
```

2.4.1.1 解析器的实现

让我们深入了解解析器的实现!

1. Parser.cpp 文件的实现中，以 parse() 开始:

```
#include "Parser.h"

AST *Parser::parse() {
    AST *Res = parseCalc();
    expect(Token::eoi);
```

```
    return Res;
}
```

parse() 的要点是使用了整个输入。还记得第一节中的解析示例中，添加了一个特殊符号来表示输入的结束吗? 需要在这里进行检查。

2. parseCalc() 实现相应的规则。因为其他解析方法也遵循相同的模式，所以有必要仔细研究一下这个方法。回顾一下第一节的规则:

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

3. 该方法首先声明一些局部变量:

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

4. 第一个决定是是否必须解析可选组。组以 with 标记开始，将标记与以下值进行比较:

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

5. 接下来，需要一个标识符:

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

若标识符存在，则将其保存在 Vars vector 中。否则，就是一个语法错误，单独处理。

6. 接下来是一个重复组，解析了更多的标识符，并用逗号分隔:

```
    while (Tok.is(Token::comma)) {
        advance();
        if (expect(Token::ident))
            goto _error;
        Vars.push_back(Tok.getText());
        advance();
    }
```

重复组以标记 (，) 开头，标记的测试成为 while 循环的条件，实现零或更多次的重复。循环内的标识符和以前一样进行处理。

7. 最后，可选组需要在末尾加一个冒号:

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

8. 最后，必须解析 expr 规则:

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

9. 有了这个调用，规则的解析就完成了。收集到的信息可以用于为该规则创建 AST 节点:

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

现在只缺少错误处理。检测语法错误很容易，但从中恢复却异常复杂。这里使用了一种简单的方法，称为恐慌模式。

恐慌模式下，标记会从标记流中删除，直到找到一个解析器可以使用它继续工作。大多数编程语言都有表示结束的符号，例如：C++ 中，a;(语句结束) 或}(语句块结束)。这样的标记是很好的候选对象。

另一方面，错误可能是正在寻找的符号丢失了，所以可能在解析器继续之前删除了许多标记。这并不像听起来那么糟糕，现在更重要的是编译器的速度。若出现错误，开发人员查看第一条错误消息，修复它，然后重新启动编译器。这与使用打孔卡完全不同，打孔卡非常重要的是获取尽可能多的错误消息，因为编译器的下一次运行可能要在第二天才可以进行。

2.4.1.2 错误处理

这里没有使用一些标记来查找，而是使用了另一组标记。对于每个非终结符，在规则中都有一组标记可以跟随该非终结符:

1. 在 calc 中，只有输入的结尾跟在这个非终结符后面。实现很简单:

```
_error:
    while (!Tok.is(Token::eoi))
        advance();
    return nullptr;
}
```

2. 其他解析方法的构造也类似。parseExpr() 是对 expr 规则的翻译:

```
Expr *Parser ::parseExpr() {
    Expr *Left = parseTerm() ;
    while (Tok.isOneOf(Token::plus, Token::minus)) {
        BinaryOp::Operator Op =
            Tok.is(Token::plus) ? BinaryOp::Plus :
                                  BinaryOp::Minus;
        advance();
        Expr *Right = parseTerm();
        Left = new BinaryOp(Op, Left, Right);
    }
    return Left;
}
```

规则中的重复组可转换为 while 循环，使用 isOneOf() 方法简化了对多个标记的检查。

3. 术语规则的编码看起来都一样:

```
Expr *Parser::parseTerm() {
    Expr *Left = parseFactor();
    while (Tok.isOneOf(Token::star, Token::slash)) {
```

```
        BinaryOp::Operator Op =
            Tok.is(Token::star) ? BinaryOp::Mul :
                                  BinaryOp::Div;
        advance();
        Expr *Right = parseFactor();
        Left = new BinaryOp(Op, Left, Right);
    }
    return Left;
}
```

此方法与 parseExpr() 非常相似，有的读者可能想将它们合并为一个方法。在语法中，可以使用一个规则处理乘法和加性操作符。使用两个规则的优点是，操作符的优先级与求值的数学顺序非常吻合。若结合了这两个规则，需要在其他地方弄清楚求值顺序。

4. 最后，需要实现解析 factor 规则:

```
Expr *Parser::parseFactor() {
    Expr *Res = nullptr;
    switch (Tok.getKind()) {
        case Token::number:
        Res = new Factor(Factor::Number, Tok.getText());
        advance(); break;
```

这里不使用 if 和 else if 语句链，而使用 switch 语句，因为每个选项都只以一个标记开头。一般来说，应该考虑喜欢使用哪种翻译模式。若以后需要更改解析方法，并不是每个方法都有不同的实现语法规则的方式，这就很方便修改了。

5. 若使用 switch 语句，错误处理将在默认情况下发生:

```
    case Token::ident:
        Res = new Factor(Factor::Ident, Tok.getText());
        advance(); break;
    case Token::l_paren:
        advance();
        Res = parseExpr();
        if (!consume(Token::r_paren)) break;
    default:
        if (!Res) error();
```

由于失败，需要在这里避免发出错误消息。

6. 若括号表达式中存在语法错误，则已经发出错误消息。这个守卫可以防止第二个错误消息:

```
        while (!Tok.isOneOf(Token::r_paren, Token::star,
                            Token::plus, Token::minus,
                            Token::slash, Token::eoi))
            advance();
    }
    return Res;
}
```

这很简单，不是吗? 记住使用的模式之后，根据语法规则编写解析器是一项乏味的工作。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从语法中构造出来

语法必须满足某些条件才能适合于构造递归下降解析器，这类语法称为 LL(1)。事实上，能在网上能找到的大多数语法都不属于这类语法。大多数关于编译器结构理论的书籍都解释了这一点的原因。关于这个话题的经典书籍是所谓的“龙书”——《编译器: 原则，技术和工具》，由 Aho, Lam, Sethi 和 Ullman 编写。

2.4.2. 抽象语法树

解析过程的结果是 AST，AST 是输入程序的另一种紧凑表示，捕获了基本信息。许多编程语言都有作为分隔符的符号，但这些符号没有进一步的含义。例如：在 C++ 中，分号; 表示单个语句的结束，这些信息对解析器很重要。当将语句转换为内存中的表示形式后，分号就不重要，可以删除了。

看一下示例表达式语言的第一条规则，with 关键字、逗号 (，) 和冒号 (:) 对于程序的含义并不重要。重要的是声明的变量列表，可以在表达式中使用。结果是只需要两个类来记录信息:Factor 保存一个数字或标识符，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 仅用于创建公共类层次结构。

除了来自解析输入的信息外，还支持使用访问者模式对树进行遍历。这都在 AST.h 头文件中有所体现:

1. 开始于访问者接口:

```
#ifndef AST_H
#define AST_H

#include "llvm/ADT/SmallVector.h"
#include "llvm/ADT/StringRef.h"

class AST;
class Expr;
class Factor;
class BinaryOp;
class WithDecl;

class ASTVisitor {
    public:
    virtual void visit(AST &){};
    virtual void visit(Expr &){};
    virtual void visit(Factor &) = 0;
    virtual void visit(BinaryOp &) = 0;
    virtual void visit(WithDecl &) = 0;
};
```

访问者模式需要知道要访问的每个类。每个类也引用访问者，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，但不做任何事情。

2. AST 类是层次结构的根:

```
class AST {
public:
    virtual ~AST() {}
    virtual void accept(ASTVisitor &V) = 0;
};
```

3. 类似地，Expr 是与表达式相关的 AST 类的根:

```
class Expr : public AST {
public:
    Expr() {}
};
```

4. Factor 类存储一个数字或一个变量的名称:

```
class Factor : public Expr {
public:
    enum ValueKind { Ident, Number };

private:
    ValueKind Kind;
    llvm::StringRef Val;

public:
    Factor(ValueKind Kind, llvm::StringRef Val)
        : Kind(Kind), Val(Val) {}
    ValueKind getKind() { return Kind; }
    llvm::StringRef getVal() { return Val; }
    virtual void accept(ASTVisitor &V) override {
        V.visit(*this);
    }
};
```

本例中，数字和变量的处理方式几乎相同，所以可以只创建一个 AST 节点类来表示。Kind 成员实例代表两种情况中的哪一种。在更复杂的语言中，通常希望使用不同的 AST 类，例如：用于数字的 NumberLiteral 类和用于变量引用的 VariableAccess 类。

5. BinaryOp 类保存表达式求值所需的数据:

```
class BinaryOp : public Expr {
public:
    enum Operator { Plus, Minus, Mul, Div };

private:
    Expr *Left;
    Expr *Right;
    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  };
```

与解析器相反，BinaryOp 类不区分乘法运算符和加法运算符。操作符的优先级在树结构中隐式可用。

6. 最后，WithDecl 类存储了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 是在解析过程中构造的。语义分析检查树是否符合语言的含义 (例如，使用已声明的变量)，并可能对树进行扩充，树将用于代码生成。

2.5. 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：变量必须在使用前声明，或者变量的类型必须在表达式中兼容。若语义分析器发现可以改进的情况，还可以输出警告。对于示例表达式语言，语义分析器必须检查是否声明了每个使用的变量 (这是该语言所要求的)。一个可能的扩展 (这里没有实现) 是，在声明的变量未使用时输出警告。

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

```
#ifndef SEMA_H
#define SEMA_H

#include "AST.h"
#include "Lexer.h"

class Sema {
    public:
    bool semantic(AST *Tree);
};

#endif
```

实现在 Sema.cpp 文件中。有趣的部分是语义分析，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。创建集合时，可以检查每个名字的唯一性，再检查给定的名字是否在集合中:

```
#include "Sema.h"
#include "llvm/ADT/StringSet.h"

namespace {
class DeclCheck : public ASTVisitor {
    llvm::StringSet<> Scope;
    bool HasError;
    enum ErrorType { Twice, Not };
    void error(ErrorType ET, llvm::StringRef V) {
        llvm::errs() << "Variable " << V << " "
                     << (ET == Twice ? "already" : "not")
                     << " declared\n";
        HasError = true;
    }
public:
    DeclCheck() : HasError(false) {}
    bool hasError() { return HasError; }
```

与 Parser 类中一样，使用一个标志来指示发生了错误。这些名称存储在一个名为 Scope 的集合中。在包含变量名的 Factor 节点上，检查变量名是否在集合中:

```
virtual void visit(Factor &Node) override {
    if (Node.getKind() == Factor::Ident) {
        if (Scope.find(Node.getVal()) == Scope.end())
        error(Not, Node.getVal());
    }
};
```

对于 BinaryOp 节点，除了两边都存在并访问之外，没什么需要检查:

```
virtual void visit(BinaryOp &Node) override {
    if (Node.getLeft())
        Node.getLeft()->accept(*this);
    else
        HasError = true;
    if (Node.getRight())
        Node.getRight()->accept(*this);
    else
        HasError = true;
};
```

在 WithDecl 节点上，填充该集合并开始遍历表达式:

```
    virtual void visit(WithDecl &Node) override {
        for (auto I = Node.begin(), E = Node.end(); I != E;
            ++I) {
            if (!Scope.insert(*I).second)
            error(Twice, *I);
        }
        if (Node.getExpr())
            Node.getExpr()->accept(*this);
        else
            HasError = true;
    };
};
}
```

semantic() 方法只启动树遍历，并返回错误标志:

```
bool Sema::semantic(AST *Tree) {
    if (!Tree)
        return false;
    DeclCheck Check;
    Tree->accept(Check);
    return Check.hasError();
}
```

还可以做得更多。若没有使用声明的变量，也可以打印警告。我们把这个留给读者当做练习来实现。若语义分析没有错误，则可以从 AST 生成 LLVM IR。这将在下一节中完成。

2.6. 使用 LLVM 后端生成代码

后端的任务是从模块的 LLVM IR 中创建优化的机器码。IR 是后端接口，可以使用 C++ 接口或以文本形式创建，IR 也是从 AST 生成的。

2.6.1. LLVM IR 的文本表示

1. 向用户询问每个变量的值。
2. 计算表达式的值。
3. 输出结果。

为了让用户提供一个变量的值并打印结果，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 如下所示:

1. 必须声明库函数，就像在 C 中一样，语法也类似于 C。函数名之前的类型是返回类型。用括号括起来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(ptr)
declare void @calc_write(i32)
```

2. calc_read() 函数将变量名作为参数。下面的构造定义了一个常量，保存 a 和空字节作为 C 中的字符串结束符:

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

3. 其遵循 main() 函数。省略参数名，因为没有使用。和 C 语言一样，函数体用大括号括起来:

```
define i32 @main(i32, ptr) {
```

4. 每个基本块必须有一个标签，这是函数的第一个基本块，将其命名为 entry:

```
entry:
```

5. calc_read() 用来读取 a 变量的值。嵌套的 getelemenptr 指令执行索引计算，以计算指向字符串常量第一个元素的指针。函数结果赋值给未命名的%2 变量。

```
    %2 = call i32 @calc_read(ptr @a.str)
```

6. 接下来，将变量乘以 3:

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

7. 结果通过调用 calc_write() 函数输出在控制台上:

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

8. 最后，main() 函数返回 0，表示执行成功:

```
    ret i32 0
}
```

LLVM IR 中的每个值都是有类型的，i32 表示 32 位整数类型，ptr 表示指针。

Note

以前版本的 LLVM 使用类型化指针。例如：在 LLVM 中，指向字节的指针表示为 i8*。从 LLVM 16 开始，不透明指针是默认的。不透明指针只是一个指向内存的指针，不携带关于类型的信息。LLVM IR 中的符号是 ptr。

既然现在已经清楚了 IR 的样子，让我们使用 AST 生成它吧。

2.6.2. 使用 AST 生成 IR

CodeGen.h 头文件中提供的接口非常少:

```
#ifndef CODEGEN_H
#define CODEGEN_H

#include "AST.h"

class CodeGen
{
    public:
    void compile(AST *Tree);
};
#endif
```

AST 包含了这些信息，所以使用一个访问器来遍历 AST。CodeGen.cpp 文件的实现如下所示:

1. 所需的包含在文件的顶部:

```
#include "CodeGen.h"
#include "llvm/ADT/StringMap.h"
#include "llvm/IR/IRBuilder.h"
#include "llvm/IR/LLVMContext.h"
#include "llvm/Support/raw_ostream.h"
```

2. LLVM 库的命名空间用于名称查找:

```
using namespace llvm;
```

3. 首先，在访问者中声明一些私有成员。在 LLVM 中，每个编译单元都由 Module 类表示，访问者有一个指向模块 M 的指针。为了便于生成 IR，使用了 Builder (IRBuilder<> 类型)。LLVM 有一个类层次结构来表示 IR 中的类型，可以从 LLVM 上下文中查找 i32 等基本类型的实例。这些基本类型经常使用。为了避免重复查找，这里缓存了所需的类型实例:VoidTy、Int32Ty、PtrTy 和 Int32Zero。成员 V 是当前计算的值，通过树遍历更新。最后，nameMap 将变量名映射到 calc_read() 函数返回的值:

```
namespace {
class ToIRVisitor : public ASTVisitor {
    Module *M;
    IRBuilder<> Builder;
    Type *VoidTy;
    Type *Int32Ty;
    PointerType *PtrTy;
    Constant *Int32Zero;
    Value *V;
    StringMap<Value *> nameMap;
```

4. 构造函数初始化所有成员:

```
public:
    ToIRVisitor(Module *M) : M(M), Builder(M->getContext())
    {
        VoidTy = Type::getVoidTy(M->getContext());
        Int32Ty = Type::getInt32Ty(M->getContext());
        PtrTy = PointerType::getUnqual(M->getContext());
        Int32Zero = ConstantInt::get(Int32Ty, 0, true);
    }
```

5. 对于每个函数，必须创建一个 FunctionType 实例。在 C++ 术语中，这是一个函数原型。函数本身是用函数实例定义的。run() 方法首先在 LLVM IR 中定义 main() 函数:

```
    void run(AST *Tree) {
        FunctionType *MainFty = FunctionType::get(
            Int32Ty, {Int32Ty, PtrTy}, false);
        Function *MainFn = Function::Create(
            MainFty, GlobalValue::ExternalLinkage,
            "main", M);
```

6. 然后用 entry 标签创建 BB 基本块，并将其添加到 IR 构建器:

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

7. 准备工作完成后，就可以开始遍历树了:

```
            Tree->accept(*this);
```

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

```
            Tree->accept(*this);FunctionType *CalcWriteFnTy =
                FunctionType::get(VoidTy, {Int32Ty}, false);
            Function *CalcWriteFn = Function::Create(
                CalcWriteFnTy, GlobalValue::ExternalLinkage,
                "calc_write", M);
            Builder.CreateCall(CalcWriteFnTy, CalcWriteFn, {V});
```

9. 生成 main() 函数，并返回 0 结束:

```
            Builder.CreateRet(Int32Zero);
        }
```

10. WithDecl 节点保存声明的变量的名称，为 calc_read() 函数创建一个函数原型:

```
    virtual void visit(WithDecl &Node) override {
        FunctionType *ReadFty =
            FunctionType::get(Int32Ty, {PtrTy}, false);
        Function *ReadFn = Function::Create(
```

```
            ReadFty, GlobalValue::ExternalLinkage,
            "calc_read", M);
```

11. 该方法循环遍历变量名:

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

12. 对于每个变量，创建一个带有变量名的字符串:

```
        StringRef Var = *I;
        Constant *StrText = ConstantDataArray::getString(
            M->getContext(), Var);
        GlobalVariable *Str = new GlobalVariable(
            *M, StrText->getType(),
            /*isConstant=*/true,
            GlobalValue::PrivateLinkage,
            StrText, Twine(Var).concat(".str"));
```

13. 创建调用 calc_read() 函数的 IR 代码，使用上一步中创建的字符串作为参数传递:

```
        CallInst *Call =
            Builder.CreateCall(ReadFty, ReadFn, {Str});
```

14. 返回值存储在 mapNames 映射中供以后使用:

```
        nameMap[Var] = Call;
    }
```

15. 树的遍历将继续使用下面的表达式:

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

16. Factor 节点可以是变量名，也可以是数字。对于变量名，将在 mapNames 映射中查找该值。对于数字，将值转换为整数并转换为常数值:

```
virtual void visit(Factor &Node) override {
    if (Node.getKind() == Factor::Ident) {
        V = nameMap[Node.getVal()];
    } else {
        int intval;
        Node.getVal().getAsInteger(10, intval);
        V = ConstantInt::get(Int32Ty, intval, true);
    }
};
```

17. 对于 BinaryOp 节点，必须使用正确的计算操作:

```
virtual void visit(BinaryOp &Node) override {
    Node.getLeft()->accept(*this);
    Value *Left = V;
    Node.getRight()->accept(*this);
```

```
    Value *Right = V;
    switch (Node.getOperator()) {
        case BinaryOp::Plus:
        V = Builder.CreateNSWAdd(Left, Right); break;
        case BinaryOp::Minus:
        V = Builder.CreateNSWSub(Left, Right); break;
        case BinaryOp::Mul:
        V = Builder.CreateNSWMul(Left, Right); break;
        case BinaryOp::Div:
        V = Builder.CreateSDiv(Left, Right); break;
    }
};
};
}
```

18. 至此，访问者类就完成了。compile() 方法创建全局上下文和模块，运行树遍历，并将生成的 IR 转储到控制台:

```
void CodeGen::compile(AST *Tree) {
    LLVMContext Ctx;
    Module *M = new Module("calc.expr", Ctx);
    ToIRVisitor ToIR(M);
    ToIR.run(Tree);
    M->print(outs(), nullptr);
}
```

现在，已经实现了编译器的前端，从读取源代码到生成 IR。所有这些组件必须在用户输入时一起工作，这是编译器驱动程序的任务，还需要实现运行时所需的函数。这两个都是下一节的主题。

2.6.3. 缺失的部分——驱动程序和运行时库

前几节中的所有阶段都由 Calc.cpp 驱动程序粘合在一起: 声明输入表达式的参数，初始化 LLVM，并调用前几节中的所有代码:

1. 首先，包含所需的头文件:

```
#include "CodeGen.h"
#include "Parser.h"
#include "Sema.h"
#include "llvm/Support/CommandLine.h"
#include "llvm/Support/InitLLVM.h"
#include "llvm/Support/raw_ostream.h"
```

2. LLVM 自带声明命令行选项的系统，只需要为需要的每个选项声明一个静态变量。在此过程中，该选项会注册到一个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。我们为输入表达式声明一个选项:

```
static llvm::cl::opt<std::string>
    Input(llvm::cl::Positional,
```

```
        llvm::cl::desc("<input expression>"),
        llvm::cl::init(""));
```

3. main() 函数中，首先初始化 LLVM 库。需要 ParseCommandLineOptions() 来处理命令行上给出的选项。这还处理帮助信息的输出，若发生错误，使用此方法将结束应用程序的运行:

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

4. 接下来，调用词法分析器和解析器。在语法分析之后，检查是否发生了任何错误。若是这种情况，则退出编译器，并返回一个表示失败的错误码:

```
    Lexer Lex(Input);
    Parser Parser(Lex);
    AST *Tree = Parser.parse();
    if (!Tree || Parser.hasError()) {
        llvm::errs() << "Syntax errors occured\n";
        return 1;
    }
```

5. 若有语义错误，也会这样做:

```
    Sema Semantic;
    if (Semantic.semantic(Tree)) {
        llvm::errs() << "Semantic errors occured\n";
        return 1;
    }
```

6. 作为驱动程序的最后一步，将调用代码生成器:

```
    CodeGen CodeGenerator;
    CodeGenerator.compile(Tree);
    return 0;
}
```

现在，已经成功地为用户输入创建了一些 IR 代码。将目标代码生成委托给 LLVM llc 静态编译器，这样就完成了编译器的实现。我们将所有组件链接在一起以创建 calc 应用程序。

运行时库由一个文件 rtcalc.c 组成，实现了 calc_read() 和 calc_write() 函数，用 C 语言实现:

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

void calc_write(int v)
{
    printf("The result is: %d\n", v);
}
```

calc_write() 只将结果值写入终端:

```c
int calc_read(char *s)
{
    char buf[64];
    int val;
    printf("Enter a value for %s: ", s);
    fgets(buf, sizeof(buf), stdin);
    if (EOF == sscanf(buf, "%d", &val))
    {
        printf("Value %s is invalid\n", buf);
        exit(1);
    }
    return val;
}
```

calc_read() 从终端读取一个整数。不能阻止用户输入字母或其他字符，所以必须仔细检查输入。若输入不是数字，则退出应用程序。更复杂的方法是让用户意识到问题，并再次要求提供一个数字。

下一步是构建并试用我们的编译器 calc，这是一个从表达式创建 IR 的应用程序。

2.6.3.1 构建和测试 calc 应用程序

为了构建 calc，首先需要在原始 src 目录之外创建一个新的 CMakeLists.txt 文件，其中包含所有源文件实现:

1. 首先，将最低所需的 CMake 版本设置为 LLVM 所需的值，并将项目名称命名为 calc:

```
cmake_minimum_required (VERSION 3.20.0)
project ("calc")
```

2. 接下来，需要加载 LLVM 包，将 LLVM 提供的 CMake 模块目录添加到搜索路径中:

```
find_package(LLVM REQUIRED CONFIG)
message("Found LLVM ${LLVM_PACKAGE_VERSION}, build type ${LLVM_BUILD_TYPE}")
list(APPEND CMAKE_MODULE_PATH ${LLVM_DIR})
```

3. 还需要从 LLVM 中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:

```
separate_arguments(LLVM_DEFINITIONS_LIST NATIVE_COMMAND ${LLVM_DEFINITIONS})
add_definitions(${LLVM_DEFINITIONS_LIST})
include_directories(SYSTEM ${LLVM_INCLUDE_DIRS})
llvm_map_components_to_libnames(llvm_libs Core)
```

4. 最后，还需要在构建中包含 src 子目录，这是本章中所有 C++ 的实现:

```
add_subdirectory ("src")
```

还需要在 src 子目录中添加一个新的 CMakeLists.txt 文件。src 目录中的 CMake 描述如下所示，只需定义可执行文件的名称 calc，然后列出要编译的源文件和要链接的库:

```
add_executable (calc
    Calc.cpp CodeGen.cpp Lexer.cpp Parser.cpp Sema.cpp)
target_link_libraries(calc PRIVATE ${llvm_libs})
```

最后，可以开始构建 calc 应用程序。在 src 目录之外，创建一个新的构建目录并对其进行修改，就可以运行 CMake 配置和构建调用:

```
$ cmake -GNinja -DCMAKE_C_COMPILER=clang -DCMAKE_CXX_COMPILER=clang++
-DLLVM_DIR=<path to llvm installation configuration> ../
$ ninja
```

我们现在有了一个新构建的、功能性的 calc 应用程序，可以生成 LLVM IR 代码。这可以进一步与 llc(LLVM 静态后端编译器) 一起使用，将 IR 代码编译成目标文件。

然后，可以使用喜欢的 C 编译器来链接小型运行时库。在 Unix-x86 上，可以键入以下命令:

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

在其他 Unix 平台 (如 AArch64 或 PowerPC) 上，需要删除-relocationmodel=pic 选项。

在 Windows 上，需要使用 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
```

现在，已经创建了第一个基于 llvm 的编译器! 请花点时间练习不同的表达方式。特别要检查乘法运算符是否在加法运算符之前求值，以及使用括号是否改变了求值顺序 (例如四则运算)。

2.7. 总结

在本章中，了解了编译器的典型组件，使用算术表达式语言介绍编程语言的语法。了解了如何为这种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，并使用 LLVM llc 静态编译器从中创建目标文件。现在我们已经开发了第一个基于 llvm 的编译器!

下一章中，我们将深化这方面的知识，为编程语言构建前端。

第二部分　从源码到机器码

继续了解如何开发自己的编译器。将从构建前端开始，读取源文件并创建源文件的抽象语法树，如何从源文件生成 LLVM IR。使用 LLVM 的优化功能，创建优化的机器码。此外，还会探索几个高级主题，包括为面向对象语言构造生成 LLVM IR 和添加调试元数据。

- 第 3 章，将源码文件转换为抽象语法树
- 第 4 章，生成 IR 代码的基础知识
- 第 5 章，高级语言结构生成的 IR
- 第 6 章，生成 IR 代码的进阶知识
- 第 7 章，优化 IR

第 3 章 将源码文件转换为抽象语法树

正如在前一章了解到的，编译器通常分为两部分——前端和后端。本章中，将实现一种编程语言的前端——主要处理源语言的部分。我们将学习现实世界的编译器使用的技术，并将其应用到我们的编程语言中。

旅程将从定义编程语言的语法开始，并以抽象语法树 (AST) 结束，其将成为代码生成的基础。对于想要实现编译器的每种编程语言，都可以使用这种方法。

本章中，将了解以下内容:

- 定义一个真正的编程语言，将了解 tinylang 语言，它是实际的编程语言的一个子集，将为它实现一个编译器前端
- 组织编译器项目的目录结构
- 了解如何为编译器处理多个输入文件
- 处理用户信息并以令人愉快的方式通知他们问题的技巧
- 使用模块化部件构建词法分析器
- 根据从语法导出的规则构造递归下降解析器来执行语法分析
- 执行语义分析，通过创建 AST 并分析其特点

有了本章的技能，读者们将能够为任何编程语言构建一个编译器前端。

3.1. 定义一种编程语言

与前一章中简单的 calc 语言相比，真正的编程面临更多挑战。为了了解其中的细节，我们将在本章和后续章节中使用 Modula-2 的一小部分。Modula-2 设计良好，可选支持泛型和面向对象编程 (OOP)。但在本书中，我们并不打算创建一个完整的 Modula-2 编译器，所以我们将这个子集称为 tinylang。

让我们从一个例子开始，看看 tinylang 编写的程序是什么样的。下面的函数使用欧几里得算法计算最大公约数:

```
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;
```

```
16
17 END Gcd.
```

现在我们对使用这种语言编写的程序有了大致的了解，下面来快速了解一下 tinylang 子集的语法。在接下来的几节中，我们将使用这种语法来派生词法分析器和解析器:

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

Modula-2 中的编译单元以 MODULE 关键字开头，后面跟着模块的名称。模块的内容可以包含一个导入模块的列表、声明，以及一个包含在初始化时运行语句的块:

```
1  declaration
2      : "CONST" ( constantDeclaration ";" )*
3      | "VAR" ( variableDeclaration ";" )*
4      | procedureDeclaration ";" ;
```

声明引入了常量、变量和过程。常量的声明以 CONST 关键字作为前缀。类似地，变量声明以 VAR 关键字开头。常量的声明非常简单:

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

标识符的名称不变。该值是从一个表达式派生的，该表达式必须在编译时可计算。变量的声明会有点复杂:

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

为了能够一次声明多个变量，需要使用标识符列表。类型名可能来自另一个模块，本例中以模块名作为前缀，这称为限定标识符 (qualified identifier):

```
1  procedureDeclaration
2      : "PROCEDURE" identifier ( formalParameters )? ";"
3          block identifier ;
4  formalParameters
5      : "(" ( formalParameterList )? ")" ( ":" qualident )? ;
6  formalParameterList
7      : formalParameter (";" formalParameter )* ;
8  formalParameter : ( "VAR" )? identList ":" qualident ;
```

前面的代码展示了如何常量、变量声明和过程。过程可以有参数和返回类型，普通参数以值的形式传递，VAR 参数以引用的形式传递。块规则中缺少的另一部分是 statementSequence，是一个单条语句的列表:

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

若语句后面跟着另一个语句，则用分号分隔，所以只支持 Modula-2 语句的子集:

```
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。限定标识符后面跟:= 是赋值操作。若后面跟着(，则是一个过程调用。其他语句是常见的控制语句:

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

IF 语句也有简化的语法，只能有一个 ELSE 块。使用该语句，可以有条件地保护语句:

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

WHILE 语句描述了一个由条件保护的循环。与 IF 语句一起，能够用 tinylang 编写简单的算法。最后，还缺少了表达式的定义:

```
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中的 calc 非常相似，只支持 INTEGER 和 BOOLEAN 数据类型。

此外，还使用了标识符和 integer_literal 标记。标识符是以字母或下划线开头，后面跟着字母、数字和下划线的名称。整数字面值是一个十进制数字序列或后跟字母 H 的十六进制数字序列。

这些规则已经有很多了，我们只介绍 Modula-2 的一部分! 不过，也可以在这个子集中编写小型应用程序。来实现一个 tinylang 编译器吧!

3.2. 项目的目录结构

tinylang 的目录结构遵循我们在第 1 章安装 LLVM 中列出的方式。每个组件的源代码位于 lib 目录的子目录中，头文件位于 include/tinylang 的子目录中。子目录以组件命名。在第 1 章安装 LLVM 中，我们只创建了基本组件。

前一章中，我们知道我们需要实现词法分析器、解析器、AST 和语义分析器。每个都是自己的组件，分别称为 Lexer、Parser、AST 和 Sema。本章将要使用的目录布局是这样的:

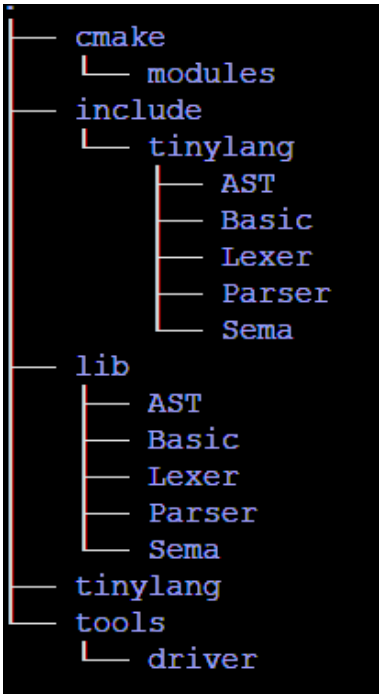


图 3.1 - tinylang 项目的目录布局

组件具有明确定义的依赖关系。Lexer 只依赖于 Basic，解析器依赖于 Basic、Lexer、AST 和 Sema，Sema 只依赖于 Basic 和 AST。定义良好的依赖关系有助于我们重用组件。

来仔细看看实现吧!

3.3. 管理编译器的输入文件

真正的编译器必须处理许多文件，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，以此类推。所有这些文件必须加载到内存中，并通过编译器的分析阶段运行。开发过程中，开发人员可能会犯语法或语义错误。当检测到错误时，应该打印一条错误消息，包括源代码行和一个标记，这个基本组成部分很重要。

幸运的是，LLVM 提供了一个解决方案:LLVM::SourceMgr 类。通过调用 AddNewSourceBuffer() 方法，一个新的源文件添加到 SourceMgr 中。或者，通过调用 AddIncludeFile() 方法来加载文件。这两种方法都返回一个标识缓冲区的 ID。可以使用这个 ID 来取得一个指向相关文件的内存缓冲区的指针。要在文件中定义位置，可以使用 llvm::SMLoc 类。该类封装了一个指向缓冲区的指针，各种 PrintMessage() 方法允许向用户发送错误和其他信息消息。

3.4. 处理用户信息

只是缺少消息的集中定义。在大型软件 (如编译器) 中，应该不希望到处散布消息字符串。若有修改消息或将其翻译成另一种语言的请求，最好将它们放在一个中心位置! 一种简单的方法是，每个消息都有一个 ID(枚举成员)、一个严重级别 (如 Error 或 Warning) 和一个包含消息的字符串。在代码中，只需要引用消息 ID 即可。

严重性级别和消息字符串仅在消息输出时使用，必须一致地管理这三项 (ID、安全级别和消息)。LLVM 库使用预处理器来解决这个问题，数据存储在一个后缀为.def 的文件中，并包装在一个宏名称中。该文件通常包含多次，并对该宏有不同的定义。该定义位于 include/tinylang/Basic/Diagnostic.def 文件路径中:

```
#ifndef DIAG
#define DIAG(ID, Level, Msg)
#endif

DIAG(err_sym_declared, Error, "symbol {0} already declared")

#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。有了这个定义，就可以定义一个 diagnosticengine 类来发出错误消息。接口在 include/tinylang/Basic/Diagnostic.h 文件中:

```
#ifndef TINYLANG_BASIC_DIAGNOSTIC_H
#define TINYLANG_BASIC_DIAGNOSTIC_H

#include "tinylang/Basic/LLVM.h"
#include "llvm/ADT/StringRef.h"
#include "llvm/Support/FormatVariadic.h"
#include "llvm/Support/SMLoc.h"
#include "llvm/Support/SourceMgr.h"
#include "llvm/Support/raw_ostream.h"
#include <utility>

namespace tinylang {
```

在包含必要的头文件之后，可以使用 Diagnostic.def 定义枚举。为了不污染全局命名空间，我们使用了名为 diag 的嵌套命名空间:

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

DiagnosticsEngine 类使用 SourceMgr 实例通过 report() 方法发送消息。消息可以有参数。为了实现这个功能，我们使用了 LLVM 提供的可变格式支持。消息文本和严重性级别在静态方法的帮助下检索，还会统计发出的错误消息的数量:

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

消息字符串由 getDiagnosticText() 返回，而级别由 getDiagnosticKind() 返回。这两个方法稍后会在.cpp 文件中进行实现:

```
SourceMgr &SrcMgr;
unsigned NumErrors;

public:
    DiagnosticsEngine(SourceMgr &SrcMgr)
        : SrcMgr(SrcMgr), NumErrors(0) {}
    unsigned nunErrors() { return NumErrors; }
```

由于消息可以具有可变数量的参数，所以 C++ 中的解决方案是使用可变模板。当然，LLVM 提供的 formatv() 函数也会使用这种方法。要获得格式化的消息，只需要转发模板参数即可:

```
    template <typename... Args>
    void report(SMLoc Loc, unsigned DiagID,
                Args &&... Arguments) {
        std::string Msg =
            llvm::formatv(getDiagnosticText(DiagID),
        std::forward<Args>(Arguments)...)
        .str();
        SourceMgr::DiagKind Kind = getDiagnosticKind(DiagID);
        SrcMgr.PrintMessage(Loc, Kind, Msg);
        NumErrors += (Kind == SourceMgr::DK_Error);
    }
};

} // namespace tinylang
#endif
```

至此，我们已经实现了类的大部分内容，只缺失了 getDiagnosticText() 和 getDiagnosticKind()。它们在 lib/Basic/Diagnostic.cpp 文件中定义，并使用了 Diagnostic.def 文件:

```
#include "tinylang/Basic/Diagnostic.h"

using namespace tinylang;

namespace {
```

```
const char *DiagnosticText[] = {
    #define DIAG(ID, Level, Msg) Msg,
    #include "tinylang/Basic/Diagnostic.def"
};
```

与头文件中一样，定义 DIAG 宏来检索所需的部件。这里，定义了一个保存文本消息的数组。因此，DIAG 宏只返回 Msg 部分。我们对于 level 中使用了相同的方法:

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

毫不奇怪，这两个函数都只是对数组进行索引，并返回所需的数据:

```
const char *
DiagnosticsEngine::getDiagnosticText(unsigned DiagID) {
    return DiagnosticText[DiagID];
}

SourceMgr::DiagKind
DiagnosticsEngine::getDiagnosticKind(unsigned DiagID) {
    return DiagnosticKind[DiagID];
}
```

SourceMgr 和 DiagnosticsEngine 类的组合为其他组件提供了良好的基础。我们将首先在 lexer 中使用它们!

3.5. 构造词法分析器

正如前一章所知，我们需要一个 Token 类和一个 Lexer 类，所以需要 TokenKind 枚举来为每个令牌类提供唯一的编号。拥有一个集所有功能于一体的头文件和实现文件并不能扩展，TokenKind 可以普遍使用，并放置在 Basic 组件中。Token 和 Lexer 类属于 Lexer 组件，但位于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、标点符号和标记，它们表示许多值的集合。例如 CONST 关键字，; 分隔符和标识标记，它们分别表示源中的标识符。每个标记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自然显示名称。

与许多编程语言一样，关键字是标识符的子集。要将标记分类为关键字，我们需要一个关键字过滤器，检查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。编程语言不断发展，可能会引入新的关键字。例如，最初的 K&R C 语言没有用 enum 关键字定义枚举，所以应该出现一个标志来指示关键字的语言级别。

我们收集了几条信息，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。对于诊断消息，我们将信息集中存储在一个名为 include/tinylang/Basic/tokenkind.def 的.def 文件中，该文件如下所示。需要注意的一点是，其关键字的前缀是 kw_:

```
#ifndef TOK
#define TOK(ID)
#endif
#ifndef PUNCTUATOR
#define PUNCTUATOR(ID, SP) TOK(ID)
#endif
#ifndef KEYWORD
#define KEYWORD(ID, FLAG) TOK(kw_ ## ID)
#endif

TOK(unknown)
TOK(eof)
TOK(identifier)
TOK(integer_literal)

PUNCTUATOR(plus, "+")
PUNCTUATOR(minus, "-")
// ···

KEYWORD(BEGIN , KEYALL)
KEYWORD(CONST , KEYALL)
// ···

#undef KEYWORD
#undef PUNCTUATOR
#undef TOK
```

有了这些集中的定义，就可以很容易地在 include/tinylang/Basic/TokenKind.h 文件中创建 TokenKind 枚举，枚举也会有自己的命名空间，例如:

```
#ifndef TINYLANG_BASIC_TOKENKINDS_H
#define TINYLANG_BASIC_TOKENKINDS_H
namespace tinylang {
    namespace tok {
        enum TokenKind : unsigned short {
#define TOK(ID) ID,
#include "TokenKinds.def"
        NUM_TOKENS
    };
```

现在应该熟悉填充数组的模式了，TOK 宏定义为只返回 ID。另外，还将 NUM_TOKENS 定义为枚举的最后一个成员，用于表示已定义的标记的数量:

```
        const char *getTokenName(TokenKind Kind);
        const char *getPunctuatorSpelling(TokenKind Kind);
        const char *getKeywordSpelling(TokenKind Kind);
    }
```

```
}

#endif
```

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

```
#include "tinylang/Basic/TokenKinds.h"
#include "llvm/Support/ErrorHandling.h"

using namespace tinylang;

static const char * const TokNames[] = {
    #define TOK(ID) #ID,
    #define KEYWORD(ID, FLAG) #ID,
    #include "tinylang/Basic/TokenKinds.def"
    nullptr
};
```

标记的文本名称源自其枚举标签 ID。有两个特点:

- 首先，一些操作符共享相同的前缀，例如 < 和 <=。若当前查看的字符是 <，则必须先检查下一个字符，再决定找到哪个标记。请记住，输入需要以空字节结束，若当前字符有效，则可以使用下一个字符:

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

- 另一个是，现在有更多的关键字。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散列表，这些关键字都存储在 tokenkind.def 文件中，可以在 Lexer 类的实例化期间完成。使用这种方法，可以使用附加标志过滤关键字，还可以支持不同级别的语言。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 llvm::StringMap 实例作为哈希表:

```
    class KeywordFilter {
        llvm::StringMap<tok::TokenKind> HashTable;
        void addKeyword(StringRef Keyword,
                        tok::TokenKind TokenCode);
    public:
        void addKeywords();
```

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

```
    tok::TokenKind getKeyword(
            StringRef Name,
            tok::TokenKind DefaultTokenCode = tok::unknown) {
```

```
        auto Result = HashTable.find(Name);
        if (Result != HashTable.end())
            return Result->second;
        return DefaultTokenCode;
    }
};
```

实现文件中，填充关键字表:

```
void KeywordFilter::addKeyword(StringRef Keyword,
                               tok::TokenKind TokenCode) {
    HashTable.insert(std::make_pair(Keyword, TokenCode));
}

void KeywordFilter::addKeywords() {
#define KEYWORD(NAME, FLAGS) \
    addKeyword(StringRef(#NAME), tok::kw_##NAME);
#include "tinylang/Basic/TokenKinds.def"
}
```

使用刚学到的技术，编写一个高效的词法分析器类并不困难。由于编译速度很重要，许多编译器使用手写词法分析器，其中一个例子就是 clang。

3.6. 构建递归下降语法分析器

如前一章所示，解析器是从语法派生出来的。回顾一下所有的构造规则，对于语法的每个规则，创建一个以规则左侧的非终结符命名的方法来解析规则的右侧。根据右边的定义，可以有以下操作:

- 对于每个非终结符，调用相应的方法
- 每个标记都要进行处理
- 对于备选项和可选的或重复的组，将检查超前标记 (下一个未使用的令牌)，以决定在何处继续

让我们将这些结构规则应用于以下语法规则:

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

可以很容易地将其转换为下面的 C++ 代码:

```
void Parser::parseIfStatement() {
    consume(tok::kw_IF);
    parseExpression();
    consume(tok::kw_THEN);
    parseStatementSequence();
    if (Tok.is(tok::kw_ELSE)) {
        advance();
```

```
        parseStatementSequence();
    }
    consume(tok::kw_END);
}
```

tinylang 的整个语法可以通过这种方式转换成 C++。因为在互联网上找到的大多数语法都不适合这种结构，所以必须小心避免一些陷阱。

语法和解析器

有两种不同类型的解析器: 自顶向下解析器和自底向上解析器。其名称来源于解析过程中处理规则的顺序，解析器的输入是词法分析器生成的标记序列。

自顶向下解析器展开规则中最左边的符号，直到匹配到一个标记。若使用了所有标记并展开了所有符号，则解析成功。这正是 tinylang 解析器的工作方式。

自底而上的解析器则相反: 查看标记序列，并尝试用语法符号替换这些标记。例如，接下来的标记是 if、3、+ 和 4，自下而上的解析器将用表达式符号替换 3 + 4 标记，从而产生 if 表达式序列。当所有属于 IF 语句的标记进行处理时，这个标记和符号序列将使用 ifStatement 符号替换。

若使用了所有标记，并且只剩下开始符号，则解析成功。虽然自顶向下的解析器可以很容易地手工构造，但自底向上的解析器却不是这样。

描述这两种解析器的另一种方法是首先展开符号。两者都从左到右读取输入，但是自顶向下的解析器首先展开最左边的符号，而自底向上的解析器首先展开最右边的符号。因此，自顶向下的解析器也称为 LL 解析器，而自底向上的解析器称为 LR 解析器。

语法必须具有某些属性，以便可以从中派生出 LL 或 LR 解析器。语法是相应地命名的: 需要一个 LL 语法来构造 LL 解析器。

可以在大学教科书中找到有关编译器构造的更多细节，例如 Wilhelm, Seidl 和 Hack 合著的《Compiler Design. Syntactic and Semantic Analysis》，Springer 2013，以及 Grune 和 Jacobs 合著的《Parsing Techniques, A practical guide》，Springer 2008。

需要查找的一个问题是左递归规则。若规则的右侧以与左侧相同的终端开始，则称为左递归规则。典型的例子可以在表达式语法中看到:

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

若从语法上看还不清楚，翻译成的 C++ 会很明显地有无限递归:

```
void Parser::parseExpression() {
    parseExpression();
    consume(tok::plus);
    parseTerm();
}
```

左递归也可以间接发生，并且涉及更多规则，这更难以发现。这就是为什么会存在一种可以检测和消除左递归的算法。

Note

左递归规则只是 LL 解析器的问题，比如 tinylang 的递归下降解析器。原因是这些解析器首先展开最左边的符号。相反，若使用解析器生成器生成 LR 解析器，首先展开最右边的符号，则应该避免右递归规则。

每一步中，解析器决定如何仅通过使用预查标记继续执行。若不能确定地做出这个决定，则代表语法有冲突。为了说明这一点，看一下 C# 中的 using 语句。与 C++ 一样，using 语句可用于使符号在命名空间中可见，例如使用 Math;。也可以使用 M = Math; 为导入的符号定义别名。在语法中，这可以表示为:

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

这里有一个问题: 在解析器使用 using 关键字之后，预检标记是标识的，但这些信息不足以让我们决定是否必须跳过或解析可选组。若可选组开始使用的标记集与可选组后面的标记集重叠，就会出现这种情况。

让我们用替代，而非可选组来重写这个规则:

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

现在，有一个不同的冲突: 两种选择都以相同的标记开始。解析器只查看预检标记，无法确定哪个选项是正确的。

这些冲突很常见，知道如何处理它们就好。一种方法是以消除冲突的方式重写语法。在前面的示例中，两种备选方案都以相同的标记开头。这可以分解出来，得到以下规则:

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

这种表述没有冲突，但也应该注意到它的表达能力较弱。在另外两个公式中，很明显哪个标识是别名，哪个标识是命名空间名。在无冲突规则中，最左边的标识改变了它的角色。首先，它是命名空间的名称，但若后面跟着等号，就变成了别名。

第二种方法是添加一个谓词来区分这两种情况。该谓词通常称为解析器，可以使用上下文信息进行决策 (例如在符号表中查找名称)，也可以查看多个标记。假设词法分析器有一个名为 Token &peek(int n) 的方法，该方法在当前的超前标记之后返回第 n 个标记。这里，等号的存在性可以作为判断过程中的额外谓词:

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

第三种方法是使用回溯。为此，需要保存当前状态，并且尝试解析冲突的组。若这没有成功，需要返回到保存的状态并尝试另一个路径。这里搜索的是要应用的正确规则，效率不如其他方法。因此，只能在万不得已的情况下才会使用这种方法。

现在，让我们加入错误恢复。上一章中，介绍了一种称为恐慌模式 (panic mode) 的错误恢复技术。基本思想是跳过标记，直到找到一个适合继续解析的标记。例如，在 tinylang 中，语句后跟分号 (:)。

若 If 语句中存在语法问题，则跳过所有标记，直到找到分号为止，再继续执行下一个语句。与其使用临时定义的标记集，不如使用系统的方法。

对于每个非终结符，计算可以跟在该非终结符后面的标记集合 (称为跟随集)。对于非终结符语句，后面可以是;、ELSE 和 END 标记，所以必须在 parseStatement() 的错误恢复部分使用这个集。这个方法假定语法错误可以在本地处理，但这是不可能的。解析器会跳过标记，可以跳过很多标记，直到到达输入的结尾，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 来实现。若返回 true，则表示错误恢复尚未完成，而 false 则表示解析 (包括可能的错误恢复) 成功。

有许多方法可以扩展此错误恢复方案。使用活动调用者的 FOLLOW 集合是一种流行的方法。作为一个简单的例子，假设 parseStatement() 由 parseStatementSequence() 调用，而 parseBlock() 本身又由 parseModule() 调用。

这里，每个相应的非终结符都有一个 FOLLOW 集合。若解析器在 parseStatement() 中检测到语法错误，则跳过标记，直到该标记至少出现在后续的活动调用者集合中的一个。若标记位于语句的 FOLLOW 集合中，则在本地恢复错误，并向调用者返回一个假值；否则，返回 true，必须继续进行错误恢复。这个扩展的一种可能的实现策略是将 std::bitset 或 std::tuple 传递给调用方，以表示当前 FOLLOW 集合的并集。

最后一个问题仍然没有解决: 我们如何调用错误恢复? 上一章中，goto 用于跳转到错误恢复块。这是可行的，但不够好。根据前面讨论的内容，我们可以在单独的方法中跳过标记。Clang 有一个用于此目的的方法 skipUntil()，tinylang 也可以用。

因为下一步是向解析器添加语义操作，所以若有必要的话，最好有一个中央位置放置清理代码。嵌套函数将是理想的选择。C++ 没有嵌套函数，而 Lambda 函数可以达到类似的目的。我们最初看到的 parseIfStatement() 方法在添加完整的错误恢复代码时，类似如下实现:

```
bool Parser::parseIfStatement() {
    auto _errorhandler = [this] {
        return skipUntil(tok::semi, tok::kw_ELSE, tok::kw_END);
    };
    if (consume(tok::kw_IF))
        return _errorhandler();
    if (parseExpression(E))
        return _errorhandler();
    if (consume(tok::kw_THEN))
        return _errorhandler();
    if (parseStatementSequence(IfStmts))
        return _errorhandler();
    if (Tok.is(tok::kw_ELSE)) {
        advance();
        if (parseStatementSequence(ElseStmts))
        return _errorhandler();
    }
```

```
18      if (expect(tok::kw_END))
19          return _errorhandler();
20      return false;
21  }
```

解析器和词法分析器生成器

手动构造解析器和词法分析器可能是一项乏味的任务，特别是在尝试发明一种新的编程语言，并经常更改语法的情况下。幸运的是，有些工具可以自动完成这项任务。

经典的 Linux 工具是 flex(https://github.com/westes/flex) 和 bison (https://www.gnu.org/software/bison/)。flex 从一组正则表达式生成词法分析器，而 bison 从语法描述生成 LALR(1) 解析器。这两个工具都可以生成 C/C++ 源代码，并且可以一起使用。

另一个流行的工具是 AntLR(https://www.antlr.org/)。AntLR 可以从语法描述生成词法分析器、解析器和 AST。生成的解析器属于 LL(*) 类，所以它是一个自顶向下的解析器，使用可变数量的查找头来解决冲突。该工具是用 Java 编写的，但可以生成许多流行语言的源代码，包括 C/C++。

所有这些工具都需要一些库支持。若正在寻找生成自包含的词法分析器和解析器的工具，那么 Coco/R(https://ssw.jku.at/Research/Projects/Coco/) 可能是适合您的工具。Coco/R 根据 LL(1) 语法描述生成词法分析器和递归下降解析器，类似于本书中使用的语法描述。生成的文件基于模板文件，可以根据需要更改模板文件。该工具用 C# 编写的，可以移植到 C++、Java 和其他语言。

还有许多其他可用的工具，其在特性和支持的输出语言方面差异很大。在选择工具时，也需要考虑权衡。LALR(1) 解析器生成器 (如 bison) 可以使用范围广泛的语法，可以在互联网上找到的免费语法通常是 LALR(1) 语法。

缺点是，这些生成器生成需要在运行时解释的状态机，这可能比递归下降解析器慢，错误处理也更加复杂。Bison 对处理语法错误提供了基本支持，但要正确使用它，需要对解析器的工作原理有深入的了解。与此相比，AntLR 消耗的语法类略小，但会自动生成错误处理，并且还可以生成 AST，所以重写语法以便与 AntLR 一起使用可能会加快以后的开发进度。

3.7. 执行语义分析

我们在上一节中构建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。上一章的 calc 示例中，解析器构建了一个 AST。在另一个单独的阶段中，语义分析器在这个树上工作。这种方法总是可以使用的。本节中，我们将使用一种稍微不同的方法，并将解析器和语义分析器更多地交织在一起。

语义分析器需要做什么？一起来看看:

- 对于每个声明，必须检查变量、对象等的名称，以确保它们没有在其他地方声明过。
- 对于表达式或语句中每次出现的名称，必须检查是否声明了该名称，以及所需的用途是否符合声明。
- 对于每个表达式，必须计算结果类型。还需要计算表达式是否为常量，若是常量，具为哪个值。

- 对于赋值和形参传递，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 BOOLEAN 类型。

对于这样一个编程语言的小子集来说，要检查的东西已经很多了！

3.7.1. 处理名称作用域

先来了解一下名称作用域，名称作用域是该名称可见的范围。与 C 语言一样，tinylang 使用了使用前声明 (declare-before-use) 模型。例如，B 和 X 变量在模块级别可声明为 INTEGER 类型:

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

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。在一个过程中，可以声明更多的变量:

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

注释所在的位置，使用 B 指向 B 局部变量，而使用 X 指向 X 全局变量。局部变量 B 的作用域是 Proc。若在当前作用域中找不到名称，则在封闭作用域中继续搜索，所以 X 变量可以在过程中使用。在 tinylang 中，只有模块和过程打开一个新的作用域。其他语言结构，如结构和类，通常也会打开作用域。预定义实体，如 INTEGER 类型和 TRUE 在全局作用域中声明，包含模块的作用域。

在 tinylang 中，只有名字才是关键。可以将作用域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该名称。对于查找，还必须知道封闭或父作用域，接口 (在 include/tinylang/Sema/Scope.h 文件中) 如下所示:

```
#ifndef TINYLANG_SEMA_SCOPE_H
#define TINYLANG_SEMA_SCOPE_H

#include "tinylang/Basic/LLVM.h"
#include "llvm/ADT/StringMap.h"
#include "llvm/ADT/StringRef.h"

namespace tinylang {

class Decl;

class Scope {
    Scope *Parent;
    StringMap<Decl *> Symbols;

    public:
    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  }
```

注意，StringMap::insert() 方法不会覆盖现有条目。结果 std::pair 的第二个成员表示表是否更新，此信息将返回给调用者。

要实现对符号声明的搜索，lookup() 方法在当前作用域内搜索，若没有找到，则搜索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:

- 当前的作用域是模块作用域。
- 查找 INTEGER 类型声明。若没有找到声明或者不是类型声明，则会出现错误。
- 实例化一个名为 VariableDeclaration 的新 AST 节点，其中重要的属性是名称 B 和类型。
- 名称 B 插入到当前作用域，映射到声明实例。若该名称已经存在于作用域中，则这是一个错误，所以当前作用域的内容不会改变。

- 对于 X 变量也是如此。

这里执行两个任务。与 calc 示例一样，构造 AST 节点。同时，计算节点的属性，如类型。为什么这是可能的呢?

语义分析器可以依赖于两组不同的属性。作用域继承自调用者，可以通过计算类型声明的名称来计算 (或合成) 类型声明。该语言的设计方式，使得这两组属性足以计算 AST 节点的所有属性。

这里，需要先声明再使用模型。若一种语言允许在声明之前使用名称，例如 C++ 中类中的成员，不可能一次计算一个 AST 节点的所有属性，AST 节点必须仅使用部分计算的属性或仅使用普通信息 (例如在 calc 示例中) 来构造。

然后必须访问 AST 一次或多次以确定丢失的信息。在 tinylang(和 Modula-2) 的情况下，可以省去 AST 构造——AST 是通过 parseXXX() 方法的调用层次结构间接表示。使用 AST 生成代码更为常见，所以我们在这里也构造一个 AST。

在把这些部分放在一起之前，我们需要了解 LLVM 使用运行时类型信息 (RTTI) 的风格。

3.7.2. AST 使用 LLVM 风格的 RTTI

当然，AST 节点是类层次结构的一部分。声明总是有一个名称，其他属性取决于所声明的内容。若声明了变量，则需要指定类型。声明常量需要类型、值等。在运行时，需要找出使用的是哪种类型的声明，dynamic_cast<> 的 C++ 操作符可用于此。问题是，只有当 C++ 类附加了虚函数表时，所需的 RTTI 才可用——使用了虚函数。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，这种风格在整个 LLVM 库中使用。

我们的层次结构的 (抽象) 基类是 Decl，要实现 llvm 风格的 RTTI，必须添加一个公共枚举，其中包含每个子类的标签。此外，还需要该类型的私有成员和公共 getter。私有成员通常为 Kind。具体的实现，如下所示:

```
class Decl {
public:
    enum DeclKind { DK_Module, DK_Const, DK_Type,
        DK_Var, DK_Param, DK_Proc };
private:
    const DeclKind Kind;
public:
    DeclKind getKind() const { return Kind; }
};
```

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

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

现在，可以使用特殊模板 llvm::isa<> 来检查对象是否为所请求的类型，并使用 llvm::dyn_cast<> 来动态转换对象。存在更多模板，但这两个是最常用的模板。有关其他模板，请参阅https://llvm

.org/docs/ProgrammersManual.html#the-isa-cast-and-dyn-cast-templates; 有关 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。

3.7.3. 创建语义分析器

有了这些，现在可以实现所有的部分。首先，必须为存储在 include/llvm/tinylang/AST/AST.h 文件中的变量创建 AST 节点的定义。除了支持 LLVM 风格的 RTTI，基类存储声明的名称、名称的位置和一个指向外层声明的指针，后者在嵌套过程的代码生成过程中是必需的。Decl 基类声明如下:

```
class Decl {
public:
    enum DeclKind { DK_Module, DK_Const, DK_Type,
                    DK_Var, DK_Param, DK_Proc };

private:
    const DeclKind Kind;

protected:
    Decl *EnclosingDecL;
    SMLoc Loc;
    StringRef Name;

public:
    Decl(DeclKind Kind, Decl *EnclosingDecL, SMLoc Loc,
        StringRef Name)
        : Kind(Kind), EnclosingDecL(EnclosingDecL), Loc(Loc),
        Name(Name) {}

    DeclKind getKind() const { return Kind; }
    SMLoc getLocation() { return Loc; }
    StringRef getName() { return Name; }
    Decl *getEnclosingDecl() { return EnclosingDecL; }
};
```

变量的声明只向类型声明添加一个指针:

```
class TypeDeclaration;

class VariableDeclaration : public Decl {
    TypeDeclaration *Ty;

public:
    VariableDeclaration(Decl *EnclosingDecL, SMLoc Loc,
                        StringRef Name, TypeDeclaration *Ty)
        : Decl(DK_Var, EnclosingDecL, Loc, Name), Ty(Ty) {}

```

```
    TypeDeclaration *getType() { return Ty; }

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

解析器中的方法需要扩展语义动作和收集信息的变量:

```
bool Parser::parseVariableDeclaration(DeclList &Decls) {
    auto _errorhandler = [this] {
        while (!Tok.is(tok::semi)) {
            advance();
            if (Tok.is(tok::eof)) return true;
        }
        return false;
    };

    Decl *D = nullptr; IdentList Ids;
    if (parseIdentList(Ids)) return _errorhandler();
    if (consume(tok::colon)) return _errorhandler();
    if (parseQualident(D)) return _errorhandler();
    Actions.actOnVariableDeclaration(Decls, Ids, D);
    return false;
}
```

DeclList 是声明列表，std::vector<Decl*>，IdentList 是位置和标识符列表，std::vector<std::pair<SMLoc, StringRef>>。

parseQualident() 方法返回一个声明，在本例中应该是一个类型声明。

解析器类知道存储在 Actions 成员中的语义分析器类 Sema 的实例。对 actOnVariableDeclaration() 的调用运行语义分析器和 AST 构造，实现在 lib/Sema/Sema.cpp 文件中:

```
void Sema::actOnVariableDeclaration(DeclList &Decls,
IdentList &Ids,
Decl *D) {
    if (TypeDeclaration *Ty = dyn_cast<TypeDeclaration>(D)) {
        for (auto &[Loc, Name] : Ids) {
            auto *Decl = new VariableDeclaration(CurrentDecl, Loc,
            Name, Ty);
            if (CurrentScope->insert(Decl))
                Decls.push_back(Decl);
            else
                Diags.report(Loc, diag::err_symbold_declared, Name);
        }
    } else if (!Ids.empty()) {
        SMLoc Loc = Ids.front().first;
        Diags.report(Loc, diag::err_vardecl_requires_type);
```

```
    }
}
```

使用 llvm::dyn_cast<TypeDeclaration> 检查类型声明。若不是类型声明，则打印错误消息；否则，对于 Ids 列表中的每个名称，将实例化 VariableDeclaration 并添加到声明列表中。若将变量添加到当前作用域中失败，因为变量名已经声明，也会输出一条错误消息。

大多数其他实体以相同的方式构建——语义分析的复杂性是唯一的区别。模块和过程需要做更多的工作，打开了一个新的作用域。打开一个新的作用域很容易: 只需要实例化一个新的作用域对象。在解析模块或过程之后，必须删除作用域。

这必须可靠，我们不想在出现语法错误时将名称添加到错误的作用域中，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身，所以在使用过程之前，必须将其名称添加到当前作用域。语义分析器有两个方法来进入和离开作用域。作用域与声明相关联:

```
void Sema::enterScope(Decl *D) {
    CurrentScope = new Scope(CurrentScope);
    CurrentDecl = D;
}

void Sema::leaveScope() {
    Scope *Parent = CurrentScope->getParent();
    delete CurrentScope;
    CurrentScope = Parent;
    CurrentDecl = CurrentDecl->getEnclosingDecl();
}
```

一个简单的辅助类用于实现 RAII 惯用法:

```
class EnterDeclScope {
    Sema &Semantics;

public:
    EnterDeclScope(Sema &Semantics, Decl *D)
    : Semantics(Semantics) {
        Semantics.enterScope(D);
    }
    ~EnterDeclScope() { Semantics.leaveScope(); }
};
```

解析模块或过程时，语义分析器会进行两次交互。第一个是在名字被解析之后，构造了 (几乎是空的)AST 节点，并建立了一个新的作用域:

```
bool Parser::parseProcedureDeclaration(/* ··· */) {
    /* ··· */
    if (consume(tok::kw_PROCEDURE)) return _errorhandler();
```

```
    if (expect(tok::identifier)) return _errorhandler();
    ProcedureDeclaration *D =
        Actions.actOnProcedureDeclaration(
            Tok.getLocation(), Tok.getIdentifier());
    EnterDeclScope S(Actions, D);
    /* ... */
}
```

语义分析器检查当前范围中的名称，返回 AST 节点:

```
ProcedureDeclaration *
Sema::actOnProcedureDeclaration(SMLoc Loc, StringRef Name) {
    ProcedureDeclaration *P =
    new ProcedureDeclaration(CurrentDecl, Loc, Name);
    if (!CurrentScope->insert(P))
        Diags.report(Loc, diag::err_symbold_declared, Name);
    return P;
}
```

真正的工作是在所有声明和过程解析之后完成的。只需要检查过程声明末尾的名称是否与过程名称相等，以及用于返回类型的声明是否为类型声明:

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

    if (Name != ProcDecl->getName()) {
        Diags.report(Loc, diag::err_proc_identifier_not_equal);
        Diags.report(ProcDecl->getLocation(),
                     diag::note_proc_identifier_declaration);
    }
    ProcDecl->setDecls(Decls);
    ProcDecl->setStmts(Stmts);

    auto *RetTypeDecl =
    dyn_cast_or_null<TypeDeclaration>(RetType);
    if (!RetTypeDecl && RetType)
        Diags.report(Loc, diag::err_returntype_must_be_type, Name);
    else
        ProcDecl->setRetType(RetTypeDecl);
}
```

有些声明本身就存在，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，必须以编程方式添加。Modula-2 还预定义了一些程序，如 INC 或 DEC，可以添加到全局作用域。对于我们的类，初始化全局作用域很简单:

```
void Sema::initialize() {
    CurrentScope = new Scope();
    CurrentDecl = nullptr;
    IntegerType =
        new TypeDeclaration(CurrentDecl, SMLoc(), "INTEGER");
    BooleanType =
        new TypeDeclaration(CurrentDecl, SMLoc(), "BOOLEAN");
    TrueLiteral = new BooleanLiteral(true, BooleanType);
    FalseLiteral = new BooleanLiteral(false, BooleanType);
    TrueConst = new ConstantDeclaration(CurrentDecl, SMLoc(),
                                        "TRUE", TrueLiteral);
    FalseConst = new ConstantDeclaration(
        CurrentDecl, SMLoc(), "FALSE", FalseLiteral);
    CurrentScope->insert(IntegerType);
    CurrentScope->insert(BooleanType);
    CurrentScope->insert(TrueConst);
    CurrentScope->insert(FalseConst);
}
```

使用该方案，tinylang 所需的所有计算都可以完成。例如，下面来看看如何计算表达式的结果是否为常量:

- 必须确保常量声明的字面量或引用是常量
- 表达式的两边都是常量，运算符也会得到一个常量

为表达式创建 AST 节点时，这些规则被嵌入到语义分析器中，也可以计算类型和常数值。

但并不是所有种类的计算都可以用这种方法进行。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。该方法需要一个特殊的遍历 AST 的顺序，在构造期间不可能获取。好消息是，所提出的方法创建了一个完全修饰过的 AST，可以为代码生成做好准备。这个 AST 可以用于进一步的分析，所以昂贵的分析可以根据需要打开或关闭。

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 程序不会产生任何输出。尽管如此，其仍可以用于探索错误恢复和引发语义错误:

```
#include "tinylang/Basic/Diagnostic.h"
#include "tinylang/Basic/Version.h"
#include "tinylang/Parser/Parser.h"
#include "llvm/Support/InitLLVM.h"
#include "llvm/Support/raw_ostream.h"

using namespace tinylang;

int main(int argc_, const char **argv_) {
    llvm::InitLLVM X(argc_, argv_);

    llvm::SmallVector<const char *, 256> argv(argv_ + 1,
                                              argv_ + argc_);
    llvm::outs() << "Tinylang "
```

```
                << tinylang::getTinylangVersion() << "\n";

    for (const char *F : argv) {
        llvm::ErrorOr<std::unique_ptr<llvm::MemoryBuffer>>
            FileOrErr = llvm::MemoryBuffer::getFile(F);
        if (std::error_code BufferError =
                FileOrErr.getError()) {
            llvm::errs() << "Error reading " << F << ": "
                         << BufferError.message() << "\n";
            continue;
        }

        llvm::SourceMgr SrcMgr;
        DiagnosticsEngine Diags(SrcMgr);
        SrcMgr.AddNewSourceBuffer(std::move(*FileOrErr),
                                  llvm::SMLoc());
        auto TheLexer = Lexer(SrcMgr, Diags);
        auto TheSema = Sema(Diags);
        auto TheParser = Parser(TheLexer, TheSema);
        TheParser.parse();
    }
}
```

恭喜! 已经完成了 tinylang 的前端实现! 可以使用示例程序 Gcd.mod(在定义真正的编程语言一节中提供)，以运行前端:

```
$ tinylang Gcd.mod
```

这是一个有效的程序，看起来好像什么也没发生。一定要修改文件，并输出一些错误消息。我们将在下一章中添加代码生成的过程。

3.8. 总结

本章中，了解了实际编译器在前端使用的技术。从项目布局开始，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，扩展了一个现有的 LLVM 类，允许集中存储消息，词法分析器现在已经分成几个接口。

然后，了解了如何根据语法描述构造递归下降解析器，了解了要避免哪些陷阱，如何使用生成器来完成这项工作。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完全修饰的 AST，将在下一章中使用它来生成 IR 代码，最后是汇编代码。

第 4 章 生成 IR 代码的基础知识

为编程语言创建了修饰抽象语法树 (AST) 之后，下一个任务是从中生成 LLVM IR 代码。LLVM IR 代码类似于具有人类可读表示的三地址码，所以需要一种系统的方法来将语言概念 (如控制结构) 翻译成低层 LLVM IR。

本章中，将了解 LLVM IR 的基础知识，如何从 AST 为控制流结构生成 IR，如何使用现代算法为静态单赋值 (SSA) 形式的表达式生成 LLVM IR，如何生成汇编文本和目标代码。

本章中，将了解以下内容:

- 使用 AST 生成 IR
- 使用 AST 编号生成 SSA 格式的 IR 代码
- 设置模块和驱动程序

本章结束时，将了解如何为编程语言创建代码生成器，以及如何将其集成到编译器中。

4.1. AST 生成 IR

LLVM 代码生成器将 LLVM IR 中的一个模块作为输入，并将其转换为目标代码或汇编文本，需要将 AST 表示转换为 IR。为了实现一个 IR 代码生成器，首先看一个简单的例子，然后开发代码生成器所需的类。完整的实现将分为三类:

- CodeGenerator
- CGModule
- CGProcedure

CodeGenerator 类是编译器驱动程序使用的通用接口，CGModule 和 CGProcedure 类保存为编译单元和单个函数生成 IR 代码所需的状态。

我们将从 clang 生成的 IR 开始。

4.1.1. 理解 IR 代码

生成 IR 代码之前，最好先了解一下 IR 语言的主要元素。在第 2 章中，简要介绍了 IR。获得更多 IR 知识的一个简单方法是研究 clang 的输出。例如，这个 C 代码 (gcd.c)，其实现了计算两个数的最大公约数的欧几里得算法:

```
unsigned gcd(unsigned a, unsigned b) {
    if (b == 0)
    return a;
    while (b != 0) {
        unsigned t = a % b;
        a = b;
        b = t;
    }
    return a;
}
```

使用 clang 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，说明该命令用于编译 Linux 下 ARM 64 位 CPU 的源文件。-S 选项指示 clang 输出一个程序集文件，通过设置-emit-llvm 选项，创建一个 IR 文件。优化级别-O1，用于获得一个易于阅读的 IR 代码。Clang 有更多的选项，所有这些选项都在https://clang.llvm.org/docs/ClangCommandLineReference.html的命令行参数引用中进行了记录。来看一下生成的文件，并理解 C 代码是如何映射到 LLVM IR 的。

一个 C 文件翻译成一个模块，其中包含函数和数据对象。一个函数至少有一个基本块，一个基本块包含指令，这种分层结构也反映在 C++ API 中。所有数据元素都有类型，整数类型由字母 i 和位数表示。例如，64 位整数类型写为 i64。最基本的浮点类型是 float 和 double，分别表示 32 位和 64 位 IEEE 浮点类型。也可以创建聚合类型，如 vector、数组和结构体。

以下是 LLVM IR，在文件的顶部，确定了一些基本属性。

```
; 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"
```

第一行是一个注释，表面使用了哪个模块标识符。下一行中，将命名源文件的文件名。对于 clang，两者一样。

目标数据布局字符串建立一些基本属性。不同的部分用-号隔开。包括以下信息:

- 小写 e 表示内存中的字节使用小端模式存储。要指定大端序，必须使用大写 E。
- M: 指定应用于符号的名称改写。这里，m:e 表示使用 ELF 名称改写。
- iN:A:P 形式，例如 i8:8:32，指定了数据的对齐方式，以位表示。第一个数字是 ABI 要求的对齐方式，第二个数字是首选对齐方式。对于字节 (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 表示原生支持 32 位和 64 位宽的整数。
- S 指定了栈的对齐方式，同样以位为单位。S128 表示栈保持 16 字节对齐。

Note

所提供的目标数据布局必须与后端期望的匹配，将捕获的信息传递给与目标无关的优化过程。例如，优化过程可以查询数据布局以获得指针的大小和对齐方式，但改变数据布局中指针的大小，并不会改变后端代码的生成。

目标数据布局提供了更多的信息，可以在https://llvm.org/docs/LangRef.html#data-layout的参考手册中找到更多信息。

最后，目标三重字符串指定了我们要编译的架构，在命令行上给出的信息。三元组是一个配置字符串，通常由 CPU 架构、厂商和操作系统组成，通常还会添加更多关于环境的信息。例如，x86_64-pc-win32 三元组用于运行在 64 位 x86 CPU 上的 Windows 系统。x86_64 是 CPU 架构，pc 是

通用的供应商，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 作为它的三元组。aarch64 是 CPU 架构，而操作系统是运行 gnu 环境的 linux。基于 linux 的系统没有真正的供应商，所以这部分未知。对于特定目的而言，不知道或不重要的部分通常忽略: 所以 aarch64-linux-gnu 和 aarch64-unknown-linux-gnu 三元组描述了同一个 Linux 系统。

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。无符号数据类型被转换成 32 位整数类型 i32。函数名以 @ 为前缀，参数名以% 为前缀。函数体用大括号括起来:

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

IR 代码组织成基本块，格式良好的基本块是一个线性指令序列，以一个可选的标签开始，以一个终止指令结束，所以每个基本块都有一个入口点和一个出口点。LLVM 允许在构造时使用畸形基本块，第一个基本块的标签是 entry。代码块中的代码很简单: 第一条指令将%b 参数与 0 进行比较。第二条指令在条件为 true 时跳转到 return 标签，然后跳转到 while 语句。若条件为 false，则使用 body 标签。

IR 代码的另一个特点是采用静态单一赋值 (SSA) 形式。代码使用无限量的虚拟寄存器，但每个寄存器只写入一次。比较的结果会赋值给指定的虚拟寄存器%cmp，使用这个寄存器，但不再写入。诸如常量传播和公共子表达式消除之类的优化，在 SSA 形式下工作得非常好，所有现代编译器都在使用。

SSA

SSA 是在 20 世纪 80 年代后期发展起来的，因其简化了数据流分析和优化，所以广泛应用于编译器中。例如，R 是 SSA 形式，循环内部公共子表达式的识别就会容易得多。SSA 的一个基本属性是建立了 def-use 和 use-def 链: 对于单个定义，可知道所有的用法 (def-use)，对于每个用法，知道唯一的定义 (use-def)。这个信息广泛使用，例如在常量传播中: 若一个定义确定为常量，则所有对该值的使用都可以很容易地替换为该常量值。

Cytron 等人 (1989) 提出的构造 SSA 形式的算法非常流行，也用于 LLVM 的实现。早期的观察发现，若语言没有 goto 语句，这些算法会变得更简单。

对 SSA 的深入研究可以在 F. rastello 和 F. B. Tichadou 的《基于 SSA 的编译器设计》一书中找到，Springer 2022。

下一个基本块是 while 循环的主体:

```
while.body:
    %b.loop = phi i32 [ %rem, %while.body ],
                      [ %b, %entry ]
```

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

在 gcd 的循环中，参数 a 和 b 会赋新值。若一个寄存器只能写入一次，则是不可能的。解决方案是使用特殊的 phi 指令，phi 指令有一个基本块和值的列表作为参数。基本块表示来自该基本块的传入边，值是来自该基本块的值。运行时，phi 指令将之前执行的基本块的标签与参数列表中的标签进行比较。

指令的值就是与标签相关联的值。对于第一条 phi 指令，若先前执行的基本块是 while.body，则该值为%rem 寄存器。若先前执行的基本块，则值为%b，这些值是位于基本块开头的值。%b 循环寄存器从第一个 phi 指令获取一个值。第二个 phi 指令的参数列表中使用了相同的寄存器，但假定该值是通过第一个 phi 指令改变之前的值。

循环体之后，必须选择返回值:

```
return:
    %retval = phi i32 [ %a, %entry ],
                      [ %b.loop, %while.body ]
    ret i32 %retval
}
```

同样，phi 指令用于选择所需的值。ret 指令不仅结束了这个基本块，运行时表示这个函数的结束。它有一个返回值作为参数。

使用 phi 指令有一些限制，必须是基本块的第一个指令。第一个基本块比较特殊: 没有之前执行过的块，所以不能以 phi 指令开始。

LLVM IR 参考

我们只触及了 LLVM IR 的皮毛，访问 LLVM 语言参考手册https://llvm.org/docs/LangRef.html，可了解更多细节。

IR 代码本身看起来很像 C 语言和汇编语言的混合体。尽管有这种熟悉的风格，但还是不清楚如何使用 AST 生成 IR 代码，尤其是 phi 指令看起来很难生成。下一节中，将为此实现一个算法!

4.1.2. 了解加载和存储的方法

LLVM 中的所有局部优化都是基于这里所示的 SSA，使用全局变量的内存引用。IR 语言有 load 和 store 指令，用于获取和存储这些值，也可以将此用于局部变量。这些指令不属于 SSA 形式，LLVM 知道如何将其转换为所需的 SSA，可以为每个局部变量分配内存槽，并使用 load 和 store 指令来更改它们的值，只需要记住指向存储变量的内存槽的指针。clang 编译器使用这种方法。

来看一下 load 和 store 的 IR 代码。再次编译 gcd.c，但这次没有启用优化:

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

gcd 函数现在看起来有所不同。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, ptr %4, align 4
    store i32 %1, ptr %5, align 4
    %7 = load i32, ptr %5, align 4
    %8 = icmp eq i32 %7, 0
    br i1 %8, label %9, label %11
```

IR 代码现在依赖于寄存器和标签的自动编号，未指定参数的名称，隐式地为%0 和%1。基本块没有标签，因此赋值为 2，前几条指令为 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 指向的内存槽中。为了比较%1 和 0，这个值显式地从内存槽加载。使用这种方法，不需要使用 phi 指令! 相反，可以从内存槽中加载一个值进行计算，然后将新值存储回内存槽中。下次读取内存槽时，就会得到最后一次计算的值。gcd 函数的所有其他基本块都遵循这种模式。

以这种方式使用加载和存储指令的优点是，生成 IR 代码相当容易；缺点是，在将基本块转换为 SSA 形式后，在第一个优化步骤中，LLVM 将使用 mem2reg 时，会删除大量的 IR 指令。

因此，我们直接生成 SSA 形式的 IR 代码，通过将控制流映射到基本块开始生成 IR 代码。

4.1.3. 控制流映射到基本块

基本块的概念，是按该顺序执行的指令的线性序列。基本块在开始时只有一个条目，以终止指令结束，终止指令是将控制流转移到另一个基本块的指令，例如：分支指令、切换指令或返回指令。请参阅https://llvm.org/docs/LangRef.html#terminator-instructions获取终止器说明的完整列表。一个基本块可以以 phi 指令开始，但在一个基本块内，既不允许 phi 指令，也不允许分支指令。换句话说，只能在第一个指令中输入一个基本块，并且只能在最后一个指令中留下一个基本块，即结束指令。不可能在基本块内分支到指令，也不可能从基本块中间分支到另一个基本块。请注意，带有 call 指令的简单函数调用可以发生在基本块中。每个基本块只有一个标签，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，从而生成控制流图 (CFG)。一个基本块可以有前身和后继，函数的第一个基本块是特殊的 (因为它不允许有前块)。

由于这些限制，源语言中的控制语句，如 WHILE 和 IF，会产生几个基本块。来看一下 WHILE 语句。WHILE 语句的条件控制是执行循环体还是执行下一个语句，该条件语句必须在一个单独的基本块中生成，因为其两个前块:

- 由 WHILE 之前的语句产生的基本块
- 从循环体的末端返回到条件的分支

还有两个后块:

- 循环体的开始部分

- 由 WHILE 后面的语句产生的基本块

循环体本身至少有一个基本块:

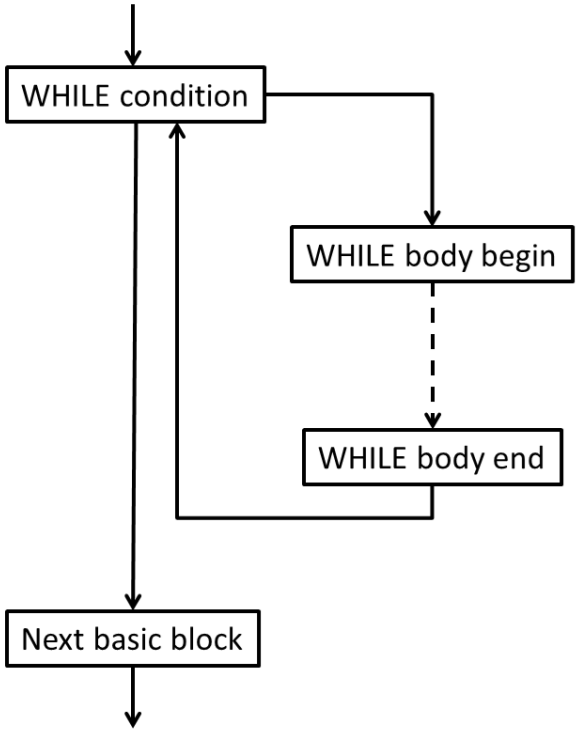


图 4.1 - WHILE 语句的基本块

IR 代码生成遵循这种结构。在 CGProcedure 类中存储一个指向当前基本块的指针，并使用 llvm::IRBuilder<> 的实例将指令插入基本块。创建基本块:

```
void emitStmt(WhileStatement *Stmt) {
    llvm::BasicBlock *WhileCondBB = llvm::BasicBlock::Create(
        CGM.getLLVMCtx(), "while.cond", Fn);
    llvm::BasicBlock *WhileBodyBB = llvm::BasicBlock::Create(
        CGM.getLLVMCtx(), "while.body", Fn);
    llvm::BasicBlock *AfterWhileBB = llvm::BasicBlock::Create(
        CGM.getLLVMCtx(), "after.while", Fn);
```

Fn 变量表示当前函数，getLLVMCtx() 返回 LLVM 上下文，两者都会在稍后设定。我们用一个基本块的分支来结束当前的基本块，该分支将保存以下条件:

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

该条件的基本块成为新的当前基本块。生成条件语句，并以一个条件分支结束代码块:

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

接下来，生成循环体，在条件语句的基本块中添加一个分支:

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

这样，就生成了 WHILE 语句。现在已经生成了 WhileCondBB 和 Curr 块，可以对其进行密封:

```
    sealBlock(WhileCondBB);
    sealBlock(Curr);
```

WHILE 语句后面的空基本块变成了当前基本块:

```
    setCurr(AfterWhileBB);
}
```

按照这个模式，可以为语言的每个语句创建 emit() 方法。

4.2. 使用 AST 编号生成 SSA 格式的 IR 代码

为了使用 AST 生成 SSA 形式的 IR 代码，可以使用一种称为 AST 编号的方法。基本思想是，对于每个基本块，我们存储写入该基本块中局部变量的当前值。

Note

实现基于 Braun et al. 的论文《Simple and Efficient Construction of Static Single Assignment Form》，International Conference on CompilerConstruction 2013 (CC 2013)，Springer(见http://individual.utoronto.ca/dfr/ece467/braun13.pdf)。在其呈现的形式中，仅适用于具有结构化受控流的 IR 代码。若需要支持任意控制流，本文还描述了必要的扩展——例如：goto。

虽然很简单，但仍然需要几个步骤。首先介绍所需的数据结构，然后了解如何在基本块的局部读写值。然后，将处理几个基本块中使用的值，并通过优化创建的 phi 指令来结束。

4.2.1. 定义数据结构保存值

使用 BasicBlockDef struct 来保存单个块的信息:

```
struct BasicBlockDef {
    llvm::DenseMap<Decl *, llvm::TrackingVH<llvm::Value>> Defs;
    // ...
};
```

llvm::Value 类表示 SSA 格式的值，其作用类似于计算结果的标签。通过 IR 指令创建，然后使用。优化过程中可能会发生各种变化，若优化器检测到%1 和%2 的值始终相同，可将%2 替换为%1。这会改变标签，但不会改变计算。

为了了解这些变化，不能直接使用 Value 类。相反，这时需要一个值句柄。Value 的处理有不同的功能。为了跟踪替换，需要使用 llvm::TrackingVH<> 类，所以 Defs 成员将 AST 的声明 (变量或正式参数) 映射到其当前值，需要为每个基本块存储这些信息:

```
llvm::DenseMap<llvm::BasicBlock *, BasicBlockDef> CurrentDef;
```

有了这个数据结构，就可以处理局部值了。

4.2.2. 定义保存值的数据结构

为了在基本块中存储局部变量的当前值，将在 map 中创建一个条目:

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

因为值可能不在基本块中，所以变量值的查找有点复杂，所以需要使用可能的递归搜索将搜索扩展到前面的节点:

```
llvm::Value *
readLocalVariable(llvm::BasicBlock *BB, Decl *Decl) {
    auto Val = CurrentDef[BB].Defs.find(Decl);
    if (Val != CurrentDef[BB].Defs.end())
        return Val->second;
    return readLocalVariableRecursive(BB, Decl);
}
```

实际的工作是搜索前面的块，将在下一节中实现。

4.2.3. 前块中搜索值

若正在查看的当前基本块只有一个前块，则直接在那里搜索变量值。若基本块有多个前块，需要在所有这些块中搜索值并组合结果。为了说明这种情况，可以看看前一节中使用 WHILE 条件语句的基本块。

这个基本块有两个前块——一个来自 WHILE 语句之前的语句，另一个来自 WHILE 循环主体末尾的分支。在条件中使用的变量应该有一些初始值，并且最有可能在循环体中更改，所以需要收集这些定义并创建 phi 指令。从 WHILE 语句创建一个包含循环的基本块。

因为需要递归地搜索前块，所以必须打破这种循环。可以使用一个简单的技巧: 插入一个空的 phi 指令，并将其记录为变量的当前值。若在搜索中再次看到这个基本块，则看到该变量有一个可以使用的值，搜索到这里就停止了。当收集了所有的值，就必须更新 phi 指令。

然而，我们仍然会面临一个问题。在查找的时候，并不是一个基本块的所有前块都已知。来看看为 WHILE 语句创建的基本块，首先生成循环条件的 IR，但只有在主体的 IR 生成之后，才能添

加从主体末尾返回到包含条件基本块的分支 (这是因为这个基本块之前是未知的)。若需要在条件中读取变量的值，那就卡住了，因为不是所有的前块都已知。

为了解决这个问题，必须做点什么:

1. 首先，必须在基本块上添加一个密封 (Sealed) 标志。
2. 然后，若知道基本块的所有前块，必须将基本块定义为密封。若基本块没有密封，并且需要查找这个基本块中尚未定义变量的值，必须插入一个空的 phi 指令，并将其作为值。
3. 也需要记住这个指令。若块稍后密封，则需要用实际值更新指令。为了实现这一点，必须在结构体 BasicBlockDef 中添加两个成员:IncompletePhis 映射，记录了稍后需要更新的 phi 指令，以及 Sealed 标志，这表示基本块是否密封:

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

4. 该方法的实现，如本节开头所讨论:

```
llvm::Value *CGProcedure::readLocalVariableRecursive(
        llvm::BasicBlock *BB, Decl *Decl) {
    llvm::Value *Val = nullptr;
    if (!CurrentDef[BB].Sealed) {
        llvm::PHINode *Phi = addEmptyPhi(BB, Decl);
        CurrentDef[BB].IncompletePhis[Phi] = Decl;
        Val = Phi;
    } else if (auto *PredBB = BB->getSinglePredecessor()) {
        Val = readLocalVariable(PredBB, Decl);
    } else {
        llvm::PHINode *Phi = addEmptyPhi(BB, Decl);
        writeLocalVariable(BB, Decl, Phi);
        Val = addPhiOperands(BB, Decl, Phi);
    }
    writeLocalVariable(BB, Decl, Val);
    return Val;
}
```

5. addEmptyPhi() 方法在基本块的开头插入一个空的 phi 指令:

```
llvm::PHINode *
CGProcedure::addEmptyPhi(llvm::BasicBlock *BB,
        Decl *Decl) {
    return BB->empty()
        ? llvm::PHINode::Create(mapType(Decl), 0,
                                "", BB)
        : llvm::PHINode::Create(mapType(Decl), 0,
                                "", &BB->front());
}
```

6. 要将缺失的操作数添加到 phi 指令中，必须首先搜索基本块的所有前块，并将操作数值对和基本块添加到 phi 指令中，需要尝试优化指令:

```
llvm::Value *
CGProcedure::addPhiOperands(llvm::BasicBlock *BB,
                            Decl *Decl,
                            llvm::PHINode *Phi) {
    for (auto *PredBB : llvm::predecessors(BB))
        Phi->addIncoming(readLocalVariable(PredBB, Decl),
                         PredBB);
    return optimizePhi(Phi);
}
```

该算法可以产生不必要的 phi 指令，有种方法可以用来进行优化，这将在下一节中实现。

4.2.4. 优化生成的 phi 指令

如何优化指令，为什么要这样做? 尽管 SSA 形式对许多优化都有利，但算法通常不能解析 phi 指令，从而阻碍了优化，所以生成的 phi 指令越少越好:

1. 若指令只有一个操作数，或者所有操作数的值都相同，则用这个值替换指令。若该指令没有操作数，则用特殊的 Undef 值替换该指令。只有当指令有两个或更多不同的操作数时，才必须保留这条指令:

```
llvm::Value *
CGProcedure::optimizePhi(llvm::PHINode *Phi) {
    llvm::Value *Same = nullptr;
    for (llvm::Value *V : Phi->incoming_values()) {
        if (V == Same || V == Phi)
            continue;
        if (Same && V != Same)
            return Phi;
        Same = V;
    }
    if (Same == nullptr)
        Same = llvm::UndefValue::get(Phi->getType());
```

2. 删除一条 phi 指令可能会给其他 phi 指令带来优化机会，LLVM 会跟踪用户和值的使用 (即 SSA 定义中提到的 use-def 链)，必须在其他 phi 指令中搜索该值使用情况，并尝试优化这些指令:

```
    llvm::SmallVector<llvm::PHINode *, 8> CandidatePhis;
    for (llvm::Use &U : Phi->uses()) {
        if (auto *P =
                llvm::dyn_cast<llvm::PHINode>(U.getUser()))
        if (P != Phi)
            CandidatePhis.push_back(P);
    }
    Phi->replaceAllUsesWith(Same);
    Phi->eraseFromParent();
    for (auto *P : CandidatePhis)
```

```
11          optimizePhi(P);
12      return Same;
13  }
```

还可以进一步改进这个算法，可以选择并记住两个不同的值，而不是总是迭代每个 phi 指令的值列表。在 optimizePhi 函数中，可以检查这两个值是否仍然在 phi 指令的列表中，则没必要优化。但即使没有这个优化，这个算法也可以运行得也很快，所以现在不进行实现。

现在，唯一没有做的是实现密封基本块的操作。

4.2.5. 密封块

了解了一个区块的所有前块，就可以密封这个块了。若语言只包含结构化语句 (如 tinylang)，那么很容易确定块的位置。再看一下为 WHILE 语句生成的基本块。

包含条件的基本块可以在主体的末端添加分支后进行密封，这是缺少的最后一个前块。要密封一个块，可以简单地将缺失的操作数，添加到不完整的 phi 指令中并设置标志:

```
1  void CGProcedure::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 了。

4.2.6. 生成表达式的 IR

要翻译表达式，如第 2 章所示。唯一有趣的部分是如何访问变量。上一节讨论了局部变量，但还要考虑其他类型的变量。这里，就讨论一下接下来需要些做什么:

- 对于过程的局部变量，使用上一节中的 readLocalVariable() 和 writeLocalVariable() 方法。
- 对于一个封闭过程中的局部变量，需要一个指向封闭过程框架的指针。这将在本章稍后的章节中进行处理。
- 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)。通过值传递的参数会视为局部变量，通过引用传递的参数会当作全局变量。

综上所述，得到如下用于读取变量或形参的代码:

```
1  llvm::Value *CGProcedure::readVariable(llvm::BasicBlock *BB,
2  Decl *D) {
3      if (auto *V = llvm::dyn_cast<VariableDeclaration>(D)) {
4          if (V->getEnclosingDecl() == Proc)
5              return readLocalVariable(BB, D);
```

```
        else if (V->getEnclosingDecl() == CGM.getModuleDeclaration()) {
            return Builder.CreateLoad(mapType(D),
            CGM.getGlobal(D));
        } else
        llvm::report_fatal_error("Nested procedures not yet supported");
    } else if (auto *FP = llvm::dyn_cast<FormalParameterDeclaration>(D)) {
        if (FP->isVar()) {
            return Builder.CreateLoad(mapType(FP, false),
            FormalParams[FP]);
        } else
        return readLocalVariable(BB, D);
    } else
        llvm::report_fatal_error("Unsupported declaration");
}
```

对变量或形式参数的写入是对称的——需要将要读的方法与要写的方法交换，并使用存储指令，而非加载指令。

接下来，在为这些函数生成 IR 代码时，应用这些函数。

4.2.7. 生成函数的 IR

大多数 IR 代码都存在于函数中，IR 代码中的函数类似于 C 语言中的函数，通过名称、参数类型、返回值和其他属性指定。要调用不同编译单元中的函数，需要声明该函数，这与 C 语言中的原型类似。若向函数中添加基本块，就可以定义函数。接下来的几节会介绍这些内容，但首先要讨论的是，符号名称的可见性。

4.2.8. 通过链接和名称修改控制可见性

函数 (以及全局变量) 具有附加的链接样式。使用链接方式，我们定义了符号名称的可见性，若多个符号具有相同的名称会发生什么。最基本的链接样式是私有的和外部的，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全局可见。

对于没有模块概念的语言 (如 C 语言)，这是足够的。对于模块，需要做更多的事情。假设有一个模块 Square，提供了一个 Root() 函数，还有一个模块 Cube，也提供了一个 Root() 函数。若函数私有，那么没什么问题，该函数获取名称 Root 和私有链接。若导出函数，以便可以从其他模块调用它，情况就不同了。因为函数名不唯一，所以仅使用函数名就不够了。

解决方案是调整名称使其全局唯一，这称为名称修改，如何做到取决于语言的要求和特点。在我们的示例中，基本思想是使用模块名和函数名的组合来创建全局唯一的名称。使用 Square.Root 作为名称看起来是一个显而易见的解决方案，但这可能会导致汇编程序的问题，因为点可能具有特殊的含义。不需要在名称组件之间使用分隔符，可以通过在名称组件前加上长度:6Square4Root 来获得类似的效果。单这不是 LLVM 的合法标识符，我们可以通过在整个名称前加上_t (t 表示 tinylang) 来修复这个问题:_t6Square4Root。通过这种方式，可以为导出符号创建唯一名称:

```
std::string CGModule::mangleName(Decl *D) {
    std::string Mangled("_t");
```

```
    llvm::SmallVector<llvm::StringRef, 4> List;
    for (; D; D = D->getEnclosingDecl())
        List.push_back(D->getName());
    while (!List.empty()) {
        llvm::StringRef Name = List.pop_back_val();
        Mangled.append(llvm::Twine(Name.size()).concat(Name).str());
    }
    return Mangled;
}
```

若语言支持类型重载，则需要使用类型名扩展此方案。例如，C++ 中中要区分 int root(int) 和 double root(double) 函数，必须将形参类型和返回值添加到函数名中。还需要考虑生成的名称的长度，因为某些链接器对长度进行了限制。在 C++ 中使用嵌套的命名空间和类，修改后的名称可能相当长。

这里，C++ 定义了一个压缩方案，以避免一遍又一遍地重复名称组件。

接下来，就来看看如何处理函数参数。

4.2.9. AST 描述类型转换为 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。由于 tinylang 目前只有两种类型，所以很容易:

```
llvm::Type *CGModule::convertType(TypeDeclaration *Ty) {
    if (Ty->getName() == "INTEGER")
        return Int64Ty;
    if (Ty->getName() == "BOOLEAN")
        return Int1Ty;
    llvm::report_fatal_error("Unsupported type");
}
```

Int64Ty、Int1Ty 和 VoidTy 是类成员，包含 i64、i1 和 void LLVM 类型的类型表示。

对于通过引用传递的形参，这是不够的。该参数的 LLVM 类型为指针型，当我们想要使用形参的值时，需要知道底层类型。这由 HonorReference 标志控制，其默认值为 true。我们对函数进行推广，并考虑形参:

```
llvm::Type *CGProcedure::mapType(Decl *Decl,
                                 bool HonorReference) {
    if (auto *FP = llvm::dyn_cast<FormalParameterDeclaration>(
    Decl)) {
        if (FP->isVar() && HonorReference)
        return llvm::PointerType::get(CGM.getLLVMCtx(),
        /*AddressSpace=*/0);
        return CGM.convertType(FP->getType());
    }
    if (auto *V = llvm::dyn_cast<VariableDeclaration>(Decl))
```

```
        return CGM.convertType(V->getType());
    return CGM.convertType(llvm::cast<TypeDeclaration>(Decl));
}
```

有了这些辅助程序，我们就可以创建 LLVM IR 函数了。

4.2.10. 创建 LLVM IR 函数

要在 LLVM IR 中生成一个函数，需要一个函数类型，这类似于 C 中的原型。创建函数类型包括映射类型，然后使用工厂方法来创建函数类型:

```
llvm::FunctionType *CGProcedure::createFunctionType(
ProcedureDeclaration *Proc) {
    llvm::Type *ResultTy = CGM.VoidTy;
    if (Proc->getRetType()) {
        ResultTy = mapType(Proc->getRetType());
    }
    auto FormalParams = Proc->getFormalParams();
    llvm::SmallVector<llvm::Type *, 8> ParamTypes;
    for (auto FP : FormalParams) {
        llvm::Type *Ty = mapType(FP);
        ParamTypes.push_back(Ty);
    }
    return llvm::FunctionType::get(ResultTy, ParamTypes,
                                   /*IsVarArgs=*/false);
}
```

根据函数类型，我们还创建了 LLVM 函数。这将函数类型与链接修改后的名称进行了关联:

```
llvm::Function *
CGProcedure::createFunction(ProcedureDeclaration *Proc,
                            llvm::FunctionType *FTy) {
    llvm::Function *Fn = llvm::Function::Create(
        Fty, llvm::GlobalValue::ExternalLinkage,
        CGM.mangleName(Proc), CGM.getModule());
```

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

创建了函数后，可以添加更多关于它的信息:

- 首先，可以给出参数的名称，可使得 IR 更具可读性。
- 其次，可以给函数和参数添加属性来指定一些特征。例如，通过引用传递的参数进行此操作。

在 LLVM 级别，这些参数是指针。但是从源语言设计来看，这些是非常受限的指针。与 C++ 中的引用类似，总是需要为 VAR 参数指定一个变量。根据设计，我们知道这个指针永远不会为空，并且始终可解引用，所以可以读取指向的值，而不会有风险。而且，该指针不能传递——特别是，因为没有比函数调用更长的指针副本，所以不能捕获指针。

llvm::AttributeBuilder 类用于为形式参数构建属性集。要获取参数类型的存储大小，可以简单地查询数据布局对象:

```
    for (auto [Idx, Arg] : llvm::enumerate(Fn->args())) {
        FormalParameterDeclaration *FP =
            Proc->getFormalParams()[Idx];
        if (FP->isVar()) {
            llvm::AttrBuilder Attr(CGM.getLLVMCtx());
            llvm::TypeSize Sz =
                CGM.getModule()->getDataLayout().getTypeStoreSize(
                    CGM.convertType(FP->getType()));
            Attr.addDereferenceableAttr(Sz);
            Attr.addAttribute(llvm::Attribute::NoCapture);
            Arg.addAttrs(Attr);
        }
        Arg.setName(FP->getName());
    }
    return Fn;
}
```

这样，就创建了 IR 函数。下一节中，我们将把函数体的基本块添加到函数中。

4.2.11. 生成函数体

我们几乎完成了为函数生成 IR 代码，只需要把这些片段组合在一起就可以生成函数了，包括它的函数体:

1. 给定一个来自 tinylang 的过程声明。首先，将创建函数类型和函数:

```
void CGProcedure::run(ProcedureDeclaration *Proc) {
    this->Proc = Proc;
    Fty = createFunctionType(Proc);
    Fn = createFunction(Proc, Fty);
```

2. 接下来，将创建函数的第一个基本块，并使其成为当前块:

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

3. 然后，必须逐一检查所有的形参。为了正确处理 VAR 参数，需要初始化 FormalParams 成员(在 readVariable() 中使用)。与局部变量相比，形参在第一个基本块中有一个值，所以些值必须已知:

```
    for (auto [Idx, Arg] : llvm::enumerate(Fn->args())) {
        FormalParameterDeclaration *FP =
            Proc->getFormalParams()[Idx];
        FormalParams[FP] = &Arg;
```

```
        writeLocalVariable(Curr, FP, &Arg);
    }
```

4. 设置之后，可以调用 emit() 方法来生成 IR 代码:

```
    auto Block = Proc->getStmts();
    emit(Proc->getStmts());
```

5. 生成 IR 代码后的最后一个块可能还没有密封，所以现在必须调用 sealBlock()。tinylang 中的过程可能有隐式返回，因此必须检查最后一个基本块是否有正确的结束符；否则，需要添加一个结束符:

```
    if (!Curr->getTerminator()) {
        Builder.CreateRetVoid();
    }
    sealBlock(Curr);
}
```

至此，我们已经完成了为函数生成 IR 代码的工作。但仍然需要创建 LLVM 模块，它将所有 IR 代码整合在一起。我们将在下一节中进行此操作。

4.3. 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。为了简化 IR 生成过程，可以将前几节中的所有函数包装到代码生成器类中。为了获得一个工作的编译器，还需要定义想要为其生成代码的目标架构，并添加生成代码的通道。我们将在本章和接下来的几章中实现它，先从代码生成器开始。

4.3.1. 代码生成器

IR 模块包含为编译单元生成大括号内的所有元素。我们在全局级别遍历模块级别的声明，创建全局变量，并调用过程的代码生成。tinylang 中的全局变量会映射到 llvm::GobalValue 类的实例。这种映射保存在全局变量中，可用于过程代码的生成:

```
void CGModule::run(ModuleDeclaration *Mod) {
    for (auto *Decl : Mod->getDecls()) {
        if (auto *Var =
                llvm::dyn_cast<VariableDeclaration>(Decl)) {
            // Create global variables
            auto *V = new llvm::GlobalVariable(
                *M, convertType(Var->getType()),
                /*isConstant=*/false,
                llvm::GlobalValue::PrivateLinkage, nullptr,
                mangleName(Var));
            Globals[Var] = V;
        } else if (auto *Proc =
            llvm::dyn_cast<ProcedureDeclaration>(Decl)) {
```

```
            CGProcedure CGP(*this);
            CGP.run(Proc);
        }
    }
}
```

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

```
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，这个类必须了解要为哪个目标架构生成代码。该信息在驱动程序中的 llvm::TargetMachine 类中:

```
std::unique_ptr<llvm::Module>
CodeGenerator::run(ModuleDeclaration *Mod,
                   std::string FileName) {
    std::unique_ptr<llvm::Module> M =
        std::make_unique<llvm::Module>(FileName, Ctx);
    M->setTargetTriple(TM->getTargetTriple().getTriple());
    M->setDataLayout(TM->createDataLayout());
    CGModule CGM(M.get());
    CGM.run(Mod);
    return M;
}
```

为了方便使用，必须为代码生成器引入一个工厂方法:

```
CodeGenerator *
CodeGenerator::create(llvm::LLVMContext &Ctx,
                      llvm::TargetMachine *TM) {
    return new CodeGenerator(Ctx, TM);
}
```

CodeGenerator 类提供了一个用于创建 IR 代码的接口，适合在编译器驱动程序中使用。在集成它之前，需要实现对机器代码生成的支持。

4.3.2. 初始化目标机型类

现在，只有目标机型未知。在目标机型上，需要定义为其生成代码的 CPU 架构。对于每个 CPU，都有一些特性会影响代码的生成过程。例如，CPU 架构族中的一个较新的 CPU 可以支持向量指令。通过这些特性，我们可以切换使用向量指令的标志。支持从命令行设置这些选项，LLVM 提供了一些支持代码。在 Driver 类中，可以包含以下头文件:

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

这个 include 将常见的命令行选项添加到编译器驱动程序中。许多 LLVM 工具也使用这些命令行选项，其好处是为用户提供了一个通用的接口，不过只缺少指定目标三元组的选项:

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

让我们定义目标机型:

1. 要显示错误消息，必须将应用程序的名称传递给该函数:

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

2. 首先，收集命令行提供的所有信息。这些是代码生成器的选项——CPU 的名称和应该激活或停用的功能，以及目标的三个值:

```
    llvm::Triple Triple = llvm::Triple(
        !MTriple.empty()
            ? llvm::Triple::normalize(MTriple)
            : llvm::sys::getDefaultTargetTriple());

    llvm::TargetOptions TargetOptions =
        codegen::InitTargetOptionsFromCodeGenFlags(Triple);
    std::string CPUStr = codegen::getCPUStr();
    std::string FeatureStr = codegen::getFeaturesStr();
```

3. 然后，必须在目标注册表中查找目标。若发生错误，将显示错误消息并退出。一个可能的错误是，用户指定的目标不支持:

```
    std::string Error;
    const llvm::Target *Target =
        llvm::TargetRegistry::lookupTarget(
         codegen::getMArch(), Triple, Error);

    if (!Target) {
        llvm::WithColor::error(llvm::errs(), Argv0) << Error;
        return nullptr;
    }
```

4. 在 Target 类的帮助下，可以使用用户请求的所有已知选项来配置目标机型:

```
    llvm::TargetMachine *TM = Target->createTargetMachine(
        Triple.getTriple(), CPUStr, FeatureStr,
        TargetOptions, std::optional<llvm::Reloc::Model>(
            codegen::getRelocModel()));
    return TM;
}
```

有了目标机型实例，就可以生成相应 CPU 架构的 IR 代码。目前缺少的是向汇编文本的转换或目标代码文件的生成，我们将在下一节添加这种支持。

4.3.3. 生成汇编程文本和目标代码

在 LLVM 中，IR 代码通过通道运行。每次遍历都只执行一项任务，比如移除无效代码。我们将在第 7 章中了解更多关于通道的信息。输出汇编代码或目标文件也作为通道实现。让我们为它添加基本支持吧!

需要包含更多的 LLVM 头文件。首先，需要 llvm::legacy::PassManager 类来保存向文件发出代码的通道。还希望能够输出 LLVM IR 代码，因此还需要一个通道来生成它。最后，使用 llvm::ToolOutputFile 类进行文件操作:

```
#include "llvm/IR/IRPrintingPasses.h"
#include "llvm/IR/LegacyPassManager.h"
#include "llvm/MC/TargetRegistry.h"
#include "llvm/Pass.h"
#include "llvm/Support/ToolOutputFile.h"
```

还需要另一个用于输出 LLVM IR 的命令行选项:

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

最后，给输出文件一个名字:

```
static llvm::cl::opt<std::string>
    OutputFilename("o",
                   llvm::cl::desc("Output filename"),
                   llvm::cl::value_desc("filename"));
```

新 emit() 方法中的第一个任务是，处理用户在命令行中没有给出的输出文件名。若从标准输入中读取输入，使用减号表示，则将结果输出到标准输出。ToolOutputFile 类了解如何处理特殊的文件名:

```
bool emit(StringRef Argv0, llvm::Module *M,
          llvm::TargetMachine *TM,
          StringRef InputFilename) {
    CodeGenFileType FileType = codegen::getFileType();
    if (OutputFilename.empty()) {
        if (InputFilename == "-") {
            OutputFilename = "-";
        }
```

否则，删除输入文件名的可能扩展名，并根据用户给出的命令行选项追加.ll、.s 或.o 作为扩展名。FileType 选项在 llvm/CodeGen/CommandFlags.inc 头文件中定义，在前面包含了它。这个选项不支持发出 IR 代码，所以添加了 new-emit-llvm 选项，只有在与汇编文件类型一起使用时才有效:

```
    else {
        if (InputFilename.endswith(".mod"))
            OutputFilename =
                InputFilename.drop_back(4).str();
        else
            OutputFilename = InputFilename.str();
        switch (FileType) {
            case CGFT_AssemblyFile:
                OutputFilename.append(EmitLLVM ? ".ll" : ".s");
                break;
            case CGFT_ObjectFile:
                OutputFilename.append(".o");
                break;
            case CGFT_Null:
                OutputFilename.append(".null");
                break;
        }
    }
}
```

一些平台区分文本和二进制文件，所以必须在打开输出文件时提供正确的标志:

```
std::error_code EC;
sys::fs::OpenFlags OpenFlags = sys::fs::OF_None;
if (FileType == CGFT_AssemblyFile)
    OpenFlags |= sys::fs::OF_TextWithCRLF;
auto Out = std::make_unique<llvm::ToolOutputFile>(
    OutputFilename, EC, OpenFlags);
if (EC) {
    WithColor::error(llvm::errs(), Argv0)
        << EC.message() << '\n';
    return false;
}
```

现在，可以将所需的通道添加到 PassManager。TargetMachine 类有一个工具方法，用于添加所请求的类，所以只需要检查用户是否请求输出 LLVM IR 代码即可:

```
    legacy::PassManager PM;
    if (FileType == CGFT_AssemblyFile && EmitLLVM) {
        PM.add(createPrintModulePass(Out->os()));
    } else {
        if (TM->addPassesToEmitFile(PM, Out->os(), nullptr,
                                    FileType)) {
            WithColor::error(llvm::errs(), Argv0)
                << "No support for file type\n";
            return false;
        }
    }
```

所有这些准备工作完成后，生成文件可以归结为一个函数调用:

```
PM.run(*M);
```

若没有显式地请求保留该文件，ToolOutputFile 类会自动删除该文件。这使得错误处理更容易，因为可能有许多地方需要处理错误。我们成功地生成了代码，所以想保留这个文件:

```
Out->keep();
```

最后，必须向调用者报告成功与否:

```
    return true;
}
```

使用 llvm::Module 调用 emit() 方法 (通过调用 CodeGenerator 类创建了该方法)，将按要求发出代码。假设将 tinylang 中的最大公约数算法存储在 Gcd.mod 文件中:

```
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.
```

为了把这个翻译成 Gcd.o 目标文件，需要输入以下内容:

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

若想直接在屏幕上检查生成的 IR 代码，请输入以下命令:

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

以 tinylang 的当前实现状态，无法在 tinylang 中创建完整的程序，可以使用一个名为 callgcd.c 的 C 程序来测试生成的目标文件。注意，调用 GCD 函数时已经修改了名称:

```c
#include <stdio.h>

extern long _t3Gcd3GCD(long, long);

int main(int argc, char *argv[]) {
    printf("gcd(25, 20) = %ld\n", _t3Gcd3GCD(25, 20));
    printf("gcd(3, 5) = %ld\n", _t3Gcd3GCD(3, 5));
    printf("gcd(21, 28) = %ld\n", _t3Gcd3GCD(21, 28));
    return 0;
}
```

要使用 clang 编译并运行整个应用程序，需要输入以下命令:

```
$ tinylang --filetype=obj Gcd.mod
$ clang callgcd.c Gcd.o -o gcd
$ gcd
```

让我们好好庆祝一下吧! 至此，通过读取语言源码，并生成了汇编代码或目标文件，我们已经创建了一个完整的编译器。

4.4. 总结

本章中，了解了如何实现 LLVM IR 代码的代码生成器，基本块是保存所有指令和表示分支的重要数据结构。了解了如何为源语言的控制语句创建基本块，以及如何向基本块中添加指令。使用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管道。有了这些知识，读者们将能够为自己的语言编译器生成 LLVM IR 代码和汇编程序文本或目标代码。

下一章中，将了解如何处理聚合数据结构，以及如何确保函数调用符合平台的规则。

第 5 章 高级语言结构生成的 IR

现在的高级语言通常使用聚合数据类型和面向对象编程 (OOP) 结构。LLVM IR 对聚合数据类型有一定的支持，而 OOP 结构 (如类) 必须自己实现。添加聚合类型会引起传递聚合类型参数的问题。不同的平台有不同的规则 (这也会体现在 IR 上)，遵循调用约定还能够调用系统函数。

本章中，将了解如何将聚合数据类型和指针转换为 LLVM IR，以及如何以符合系统的方式将参数传递给函数。还将了解如何在 LLVM IR 中实现类和虚函数。

本章中，将了解以下内容:

- 处理数组、结构体和指针
- 正确获取应用程序的二进制接口 (ABI)
- 为类和虚函数创建 IR 代码

本章结束时，将了解为聚合数据类型和 OOP 构造创建 LLVM IR 的知识。还将了解如何根据平台的规则，传递聚合数据类型。

5.1. 环境要求

本章使用的代码在这里，https://github.com/PacktPublishing/Learn-LLVM-17/tree/main/Chapter05。

5.2. 处理数组、结构体和指针

对于应用程序来说，只有 INTEGER 这样的基本类型是不够的。要表示矩阵或复数等数学对象，必须在已有数据类型的基础上构造新的数据类型。这些新数据类型通常称为聚合 (aggregate) 或组合 (composite)。

数组是由相同类型的元素组成的序列。在 LLVM 中，数组总是静态的，所以元素的数量恒定。tinylang 类型 ARRAY [10] OF INTEGER 或 C 类型 long[10] 在 IR 中表示如下:

```
[10 x i64]
```

结构是不同类型的组合物。在编程语言中，通常用命名成员表示。例如，在 tinylang 中，结构可写成 RECORD x: REAL; color: INTEGER; y: REAL; END; C 语言中同样的结构是 struct { float x; long color; float y; };。在 LLVM IR 中，只列出类型名:

```
{ float, i64, float }
```

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为 0。

该结构的成员按照数据布局字符串中的规范在内存中排列。有关 LLVM 中数据布局字符串的更多信息，详见第 4 章。

若有必要，还会插入未使用的填充字节。若需要控制内存布局，可以使用打包结构，其中所有元素都具有 1 字节对齐方式。在 C 语言中，可以以下方式利用结构体的__packed__属性:

```
struct __attribute__((__packed__)) { float x; long long color; float y; }
```

LLVM IR 中的语法略有不同:

```
<{ float, i64, float }>
```

加载到寄存器中，数组和结构体视为一个单元。例如，不可能将数组值寄存器%x 的单个元素引用为%x[3]。这是由于 SSA 形式，不能判断%x[i] 和%x[j] 是否指的是同一个元素。相反，需要特殊的指令来提取和插入单元素值到数组中。要读取第二个元素，需要使用以下指令:

```
%el2 = extractvalue [10 x i64] %x, 1
```

也可以更新一个元素，比如第一个元素:

```
%xnew = insertvalue [10 x i64] %x, i64 %el2, 0
```

这两个指令也适用于结构体。例如，使用寄存器%pt 访问 color 成员:

```
%color = extractvalue { float, float, i64 } %pt, 2
```

这两条指令都有一个重要的限制: 索引必须是一个常量。对于结构体来说，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。对于数组来说，只是无法高效地实现。这两条指令在元素数量较少，且已知的特定情况下都有价值。例如，复数可以建模为两个浮点数组成的数组。传递这个数组是合理的，而且很清楚在计算过程中必须访问数组的哪一部分。

对于前端的一般使用，必须借助于指向内存的指针。LLVM 中的所有全局值都用指针表示。下面将全局变量 @arr 声明为包含 8 个 i64 元素的数组，这等价于 C 语言中的 long arr[8] 声明:

```
define i64 @second() {
    %1 = load i64, ptr getelementptr inbounds ([8 x i64], ptr @arr, i64
    0, i64 1)
    ret i64 %1
}
```

getelementptr 指令是地址计算的主力，需要更详细的解释。第一个操作数 [8 x i64] 是指令所操作的基类型，第二个操作数 ptr @arr 指定基指针。请注意这里的差别: 声明了一个包含八个元素的数组，由于所有全局值都视为指针，所以有一个指向数组的指针。在 C 的语法中，实际上使用 long (*arr)[8]! 其结果是，在对元素进行索引之前，必须先对指针解引用，如 C 语言中的 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 是元素索引，计算的结果是索引元素的地址。还需要注意的是，此指令不涉及内存存取。

除了结构体，索引参数不需要是常量，可以在循环中使用 getelementptr 指令来检索数组的元素。这里对结构体的处理有所不同: 只能使用常量，且类型必须为 i32。

有了这些知识，数组可以很容易地集成到第 4 章的代码生成器中，要创建这个类型，必须扩展 convertType() 方法。若 Arr 变量保存数组的类型标识符，并且假设数组中的元素数量为整数字面值，则可以向 convertType() 方法添加以下代码来处理数组:

```
if (auto *ArrayTy =
            llvm::dyn_cast<ArrayTypeDeclaration>(Ty)) {
    llvm::Type *Component =
        convertType(ArrayTy->getType());
    Expr *Nums = ArrayTy->getNums();
    uint64_t NumElements =
        llvm::cast<IntegerLiteral>(Nums)
            ->getValue()
            .getZExtValue();
    llvm::Type *T =
        llvm::ArrayType::get(Component, NumElements);
    // TypeCache is a mapping between the original
    // TypeDeclaration (Ty) and the current Type (T).
    return TypeCache[Ty] = T;
}
```

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。我们在过程的第一个基本块中可以这样做:

```
for (auto *D : Proc->getDecls()) {
    if (auto *Var =
            llvm::dyn_cast<VariableDeclaration>(D)) {
        llvm::Type *Ty = mapType(Var);
        if (Ty->isAggregateType()) {
            llvm::Value *Val = Builder.CreateAlloca(Ty);
            // The following method requires a BasicBlock (Curr),
            // a VariableDeclation (Var), and an llvm::Value (Val)
            writeLocalVariable(Curr, Var, Val);
        }
    }
}
```

为了读写一个元素,必须生成 getelementptr 指令,这会添加到 emitExpr()(读取值) 和 emitStmt()(写入值) 方法中。要读取数组的元素，首先读取变量的值，再处理变量的选择器。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:

```
auto &Selectors = Var->getSelectors();
for (auto I = Selectors.begin(), E = Selectors.end();
        I != E; ) {
    if (auto *IdxSel =
            llvm::dyn_cast<IndexSelector>(*I)) {
        llvm::SmallVector<llvm::Value *, 4> IdxList;
        while (I != E) {
            if (auto *Sel =
                    llvm::dyn_cast<IndexSelector>(*I)) {
                IdxList.push_back(emitExpr(Sel->getIndex()));
                ++I;
```

```
            } else
                break;
        }
        Val = Builder.CreateInBoundsGEP(Val->getType(), Val, IdxList);
        Val = Builder.CreateLoad(
            Val->getType(), Val);
    }
    // . . . Check for additional selectors and handle
    // appropriately by generating getelementptr and load.
    else {
        llvm::report_fatal_error("Unsupported selector");
    }
}
```

写入数组元素使用相同的代码，但不生成加载指令，可以使用指针作为存储指令中的目标。对于记录，可以使用类似的方法。记录成员的选择器包含名为 Idx 的常量字段索引，可把这个常量转换成一个 LLVM 常量值:

```
llvm::Value *FieldIdx = llvm::ConstantInt::get(Int32Ty, Idx);
```

然后，在 Builder.CreateGEP() 方法中使用 value，就像在数组中一样。

现在，应该知道如何将聚合数据类型转换为 LLVM IR。以符合系统的方式传递这些类型值要非常谨慎，下一节将介绍如何正确地实现。

5.3. 获得应用程序二进制接口

通过向代码生成器添加数组和记录，有时生成的代码不会按预期执行，原因是我们忽略了平台的调用约定。对于同一个程序或库中的函数如何调用另一个函数，每个平台都定义了自己的规则。ABI 文档中总结了这些规则，常见信息包括以下内容:

- 机器寄存器用于参数传递吗? 如果是，是哪些?
- 数组和结构这样的聚合类型如何传递给函数?
- 如何处理返回值?

某些平台上，聚合类型间接传递，所以类型的副本会放在栈上，只有指向副本的指针作为参数传递。其他平台上，在寄存器中传递一个小的聚集 (例如 128 或 256 位宽)，只有超过该阈值时才使用间接参数传递。有些平台还使用浮点寄存器和向量寄存器来传递参数，而另一些平台则要求浮点值使用整数寄存器传递。

当然，这些都是些底层特性，但它会影响 LLVM IR，明明已经在 LLVM IR 中定义了函数的所有参数类型？！事实证明，这还不够。为了理解这一点，来看一下复数。有些语言内置了复数的数据类型。例如，C99 有 float _Complex(以及其他)，旧版本的 C 没有复数类型，但可以定义 struct complex {float re, im;} 并在此类型上创建算术运算。这两种类型都可以映射到 LLVM 的 {float, float } IR 类型。

若 ABI 现在声明内置复数类型的值在两个浮点寄存器中传递，但用户定义的聚合总是间接传递，该函数给出的信息，不足以让 LLVM 决定如何传递这个特定的参数。但需要向 LLVM 提供更多的信息，而这些信息等级特定于 ABI。

有两种方法可以将这些信息指定给 LLVM: 参数属性和类型重写，这取决于目标平台和代码生成器。最常用的参数属性如下所示:

- inreg 指定参数在寄存器中传递
- byval 指定参数按值传递，该参数必须是指针类型。将指向的数据生成一个隐藏副本，并将该指针传递给调用函数。
- zeroext 和 signext 指定传递的整数值是零或符号扩展。
- sret 指定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类型。

虽然所有代码生成器都支持 zeroext、signext 和 sret 属性，但只有一些支持 inreg 和 byval。可以使用 addAttr() 将属性添加到函数的参数中。例如，要在参数 Arg 上设置 inreg 属性，可以使用以下方式:

```
Arg->addAttr(llvm::Attribute::InReg);
```

若需要设置多个属性，可以使用 llvm::AttrBuilder 类。

提供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:

1. 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。
2. 将参数转换为不同的表示形式，例如：通过整数寄存器传递浮点值。

要在不改变值位的情况下强制转换类型，可以使用 bitcast 指令。位转换指令可以操作简单的数据类型，如整数和浮点值。当通过整数寄存器传递浮点值时，必须将该浮点值强制转换为整数。在 LLVM 中，32 位的浮点数表示为 float，32 位的整数表示为 i32。可以通过以下方式将浮点值转换为整数:

```
%intconv = bitcast float %fp to i32
```

此外，位转换指令要求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 针对每种支持的 CPU 平台都有文档，可以查找文档并熟悉相关信息。

还有关于 LLVM 代码生成器需求的文档。信息的来源是 clang 实现，可以在https://github.com/llvm/llvm-project/blob/main/clang/lib/CodeGen/TargetInfo.cpp上找到。这个文件包含针对所有受支持平台的 ABI 操作，也是收集信息的地方。

本节中，了解了如何为函数调用生成与平台的 ABI 兼容的 IR。下一节将为生成类和虚函数的 IR 创建不同的方法。

5.4. 为类和虚函数创建 IR

许多现代编程语言使用类支持面向对象，类是高级语言结构。本节中，我们将探讨如何将类结构映射到 LLVM IR 中。

5.4.1. 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，或者覆盖现有的虚函数。我们用 Oberon-2 中的类来说明这一点，Oberon-2 也是一个很好的 tinylang 模型。Shape 类定义了一个具有颜色和面积的抽象类型:

```
TYPE Shape = RECORD
                color: INTEGER;
                PROCEDURE (VAR s: Shape) GetColor(): INTEGER;
                PROCEDURE (VAR s: Shape) Area(): REAL;
             END;
```

GetColor 方法只返回颜号:

```
PROCEDURE (VAR s: Shape) GetColor(): INTEGER;
BEGIN RETURN s.color; END GetColor;
```

抽象形状的面积无法计算，所以这是一种抽象的方法:

```
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;
```

对于一个圆，面积则可以计算:

```
PROCEDURE (VAR s: Circle) Area(): REAL;
BEGIN RETURN 2 * radius * radius; END;
```

该类型也可以在运行时查询。若形状是 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，其工作原理与 C++ 非常相似。但显著区别是 Oberon-2 语法显式声明了 this 指针，将其称为方法的接收方。

要解决的基本问题是如何在内存中布局类，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，Shape 类只有一个数据成员，可以将其映射到相应的 LLVM 结构类型:

```
@Shape = type { i64 }
```

Circle 类添加了另一个数据成员，解决方案是在末尾附加新的数据成员:

```
@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始终具有相同的内存偏移量，并且使用相同的索引通过 getelementptr 指令访问相应的字段。

为了实现方法的动态调用，必须进一步扩展 LLVM 结构。若在 Shape 对象上调用 Area() 函数，则调用抽象方法，从而导致应用程序停止。若在 Circle 对象上调用，则调用相应的方法来计算圆的面积。另一方面，可以为这两个类的对象调用 GetColor() 函数。

实现这一点的基本思想是，将一个表与每个对象的函数指针关联起来。这里，表有两个信息: 一个用于 GetColor() 方法，另一个用于 Area() 函数。Shape 类和 Circle 类都有这样一个表。这些表在 Area() 函数的条目中有所不同，该函数根据对象的类型调用不同的代码。这个表称为虚函数表，通常缩写为 vtable。

虚函数表本身没有用，必须把它和一个对象关联起来，总是添加指向虚函数表的指针，作为该结构的第一个数据成员。在 LLVM 级别，@Shape 类型是这样的:

```
@Shape = type { ptr, i64 }
```

@Circle 类型也进行了类似的扩展。

得到的内存结构如图 5.1 所示:

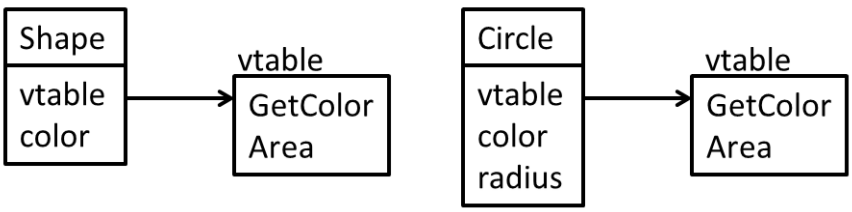


图 5.1 - 类和虚函数表的内存布局

LLVM IR 中，Shape 类的虚函数表可以可视化为如下表示，其中两个指针对应于 GetColor() 和 GetArea() 方法，如图 5.1 所示:

```
@ShapeVTable = constant { ptr, ptr } { GetColor(), Area() }
```

此外，LLVM 没有 void 指针，而是使用指向字节的指针。随着隐藏虚值表字段的引入，现在还需要有一种方法来初始化它。C++ 中，这是调用构造函数的一部分。在 Oberon-2 中，该字段在分配内存时自动初始化。

然后，通过以下步骤执行对方法的动态调用:

1. 通过 getelementptr 指令计算虚函数表指针的偏移量。
2. 加载指向虚参表的指针。
3. 计算函数在虚函数表中的偏移量。
4. 加载函数指针。

5. 使用调用指令通过指针间接调用函数。

还可以在 LLVM IR 中可视化对虚拟方法 (如 Area()) 的动态调用。首先，从 Shape 类的相应指定位置加载一个指针。下面的 load 表示将指针加载到实际的虚函数表中，以便形成函数:

```
// Load a pointer from the corresponding location.
%ptrToShapeObj = load ptr, ...
// Load the first element of the Shape class.
%vtable = load ptr, ptr %ptrToShapeObj, align 8
```

之后，getelementptr 获取偏移量以调用 Area():

```
%offsetToArea = getelementptr inbounds ptr, ptr %vtable, i64 1
```

然后，加载指向 Area() 的函数指针:

```
%ptrToAreaFunction = load ptr, ptr %offsetToArea, align 8
```

最后，通过指针调用 Area() 函数：

```
%funcCall = call noundef float %ptrToAreaFunction(ptr noundef
  nonnull align 8 dereferenceable(12) %ptrToShapeObj)
```

即使在单继承的情况下，生成的 LLVM IR 也可能看起来非常冗长。尽管生成对方法的动态调用过程看起来效率不高，但大多数 CPU 架构只需两条指令就可以执行该动态调用。

此外，要将函数转换为方法，还需要对对象数据的引用，需要通过将指向数据的指针作为方法的第一个参数来实现的。在 Oberon-2 中，这是明确的接收者。在类似 C++ 的语言中，是隐式 this 指针。

使用虚函数表，每个类在内存中都有唯一地址。这对运行时类型测试也有帮助吗? 是的，但帮助有限。为了说明这个问题，用一个继承自 Circle 类的 Ellipse 类来扩展类层次，这不是数学意义上的 is-a 关系。

若有一个 Shape 类型的 shape 变量，则可以实现 shape IS Circle 类型的测试，将 shape 变量中存储的虚表指针与 Circle 类的虚表指针进行比较。只有当 shape 具有确切的 Circle 类型时，这种比较才会为 true。但若 shape 是 Ellipse 类型，则比较返回 false。但在需要 Circle 类型对象的所有地方，都可以使用 Ellipse 类型的对象。

解决方案是使用运行时类型信息扩展虚函数表，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，存储一个指向基类虚函数表的指针就足够了，如图 5.2 所示:

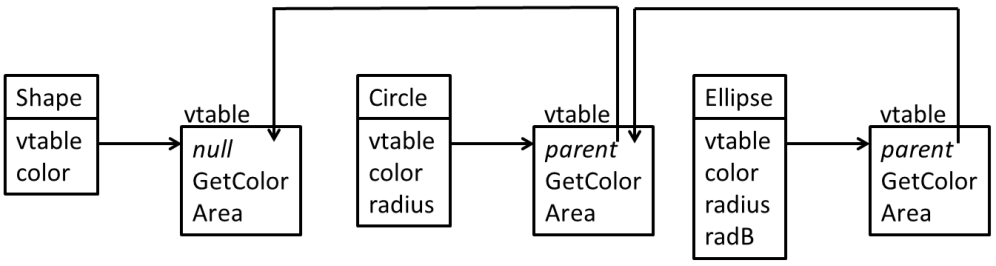


图 5.2 - 支持简单类型测试的类和虚函数表布局

若测试失败，则使用指向基类的虚函数表的指针重复测试。重复此操作，直到测试结果为 true；若没有基类，则为 false。与调用动态函数相比，类型测试是一个代价高昂的操作，最坏的情况下，继承层次结构可以向上回溯到根类。

若了解整个类层次结构，就有一种有效的方法: 按照深度优先的顺序，对类层次结构的每个成员进行编号。类型测试变成了对一个数字或一个间隔进行比较，这可以在恒定的时间内完成。事实上，这就是 LLVM 自己的运行时类型测试的方法，我们在前一章中已经了解过了。

将运行时类型信息与虚函数表耦合是一种设计决策，要么是源语言强制要求的，要么只是作为实现细节。例如，语言在运行时支持反射，所以需要详细的运行时类型信息，并且数据类型没有 vtable，将两者耦合就不是一个好主意。在 C++ 中，耦合会导致具有虚函数 (因此没有虚函数表) 的类，没有运行时的类型数据。

通常，编程语言支持的接口是虚拟方法的集合。接口很重要，提供了有用的抽象。我们将在下一节中查看接口的可能实现方式。

5.4.2. 使用接口扩展单继承

Java 等语言支持接口，接口是抽象方法的集合，类似于没有数据成员、只定义了抽象方法的基类。接口带来了一个有趣的问题，实现接口的每个类，都可以在虚函数表中的不同位置拥有相应的方法。原因很简单，虚函数表中函数指针的顺序，是从语言源码中类定义中的函数顺序派生出来的。接口的定义与此无关，不同的顺序是标准。

接口中定义的方法可以有不同的顺序，可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，这个表可以指定该方法在虚函数表中的索引，也可以指定存储在虚函数表中的函数指针的副本。若在接口上调用方法，则搜索接口对应的虚函数表，获取指向该函数的指针，并调用该方法。在 Shape 类中添加两个 I1 和 I2 接口会产生以下布局:

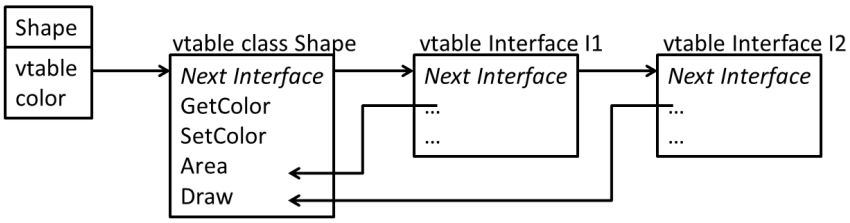


图 5.3 - 接口虚函数表的布局

需要注意的是，必须找到合适的表。可以使用类似于运行时类型测试的方法: 在接口虚函数表中执行线性搜索。可以为每个接口分配唯一编号 (例如，一个内存地址)，并使用这个编号来标识这个虚函数表。这种方案的缺点是显而易见的: 通过接口调用方法比在类上调用相同的方法要花费更多的时间，但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，类实现的接口是已知的，以可以构造一个完美的哈希函数，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标识接口的唯一编号，还有其他方法可以计算唯一编号。若源码中的符号名称唯一，则可以计算该符号的加密哈希 (例如 MD5)，并使用该哈希作为数字。计算在编译时进行，没有运行时成本。

结果比线性搜索快得多，只需要常数时间。不过，会涉及数的几个算术操作，比类类型方法调用要慢。

通常，接口也参与运行时类型测试，使列表搜索更长。若实现了哈希表方法，也可以用于运行时类型测试。

有些语言允许有多个父类。这对实现有一些有趣的挑战，我们将在下一节中了解这些。

5.4.3. 增加对多重继承的支持

多重继承增加了另一个挑战。若一个类继承自两个或多个基类，则需要以一种仍然可以从方法访问的方式组合数据成员。与单继承情况类似，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，而且是一个图形对象。为了对此建模，让 Circle 类继承 Shape 类和 GraphicObj 类。类布局中，Shape 类中的字段首先出现，再追加 GraphicObj 类的所有字段，包括隐藏的虚函数表指针。之后，添加 Circle 类的新数据成员，得到如图 5.4 所示的整体结构:

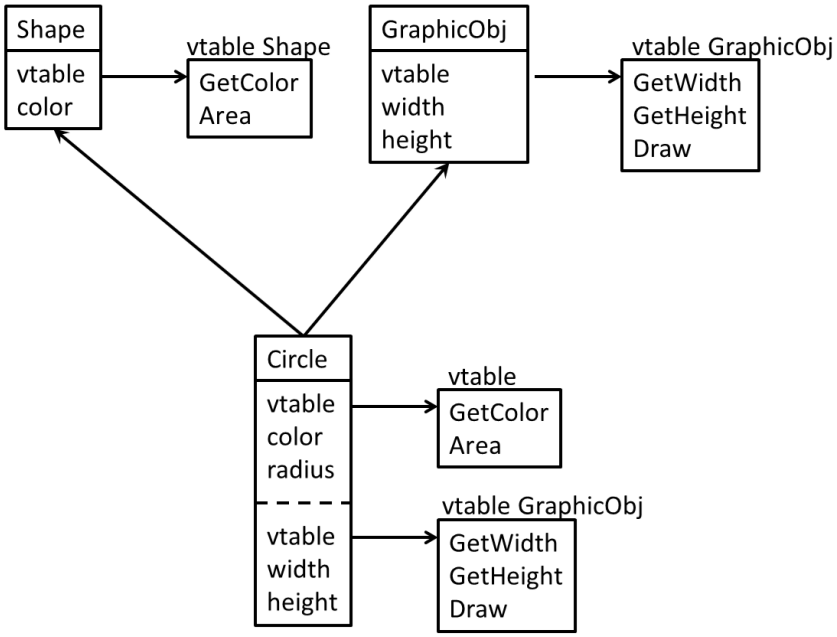


图 5.4 - 具有多重继承的类和变量的布局

这种方法有几个含义，可以有多个指向对象的指针。指向 Shape 或 Circle 类的指针指向对象的顶部，而指向 GraphicObj 类的指针指向该对象内部，即嵌入的 GraphicObj 对象的开始，在比较指针时必须考虑到这一点。

调用虚拟方法也会受到影响，若在 GraphicObj 类中定义了一个方法，则这个方法需要 GraphicObj 类的类布局。若在 Circle 类中没有重写此方法，则有两种可能。若方法调用通过指向 GraphicObj 实例的指针完成，可以在 GraphicObj 类的 vtable 中查找方法的地址并调用该函数。更复杂的情况是，若使用指向 Circle 类的指针调用该方法，可以在 Circle 类的虚函数表中查找方法的地址。调用的方法期望 this 指针是 GraphicObj 类的一个实例，所以也必须调整那个指针。因为我们知道 GraphicObj 类在 Circle 类中的偏移量，所以可以这样做。

若在 Circle 类中重写了 GrapicObj 方法，则通过指向 Circle 类的指针调用该方法，不需要做什么特殊操作。但若该方法是通过指向 GraphicObj 实例的指针调用，因为该方法需要一个指向 Circle 实例的 this 指针，就需要做一些调整。在编译时，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，所以无法了解需要调整的部分。为了解决这个问题，我们将再调用方法之前，对 this 指针进行调整，并将每个函数指针一起存储在虚函数表中，如图 5.5 所示:

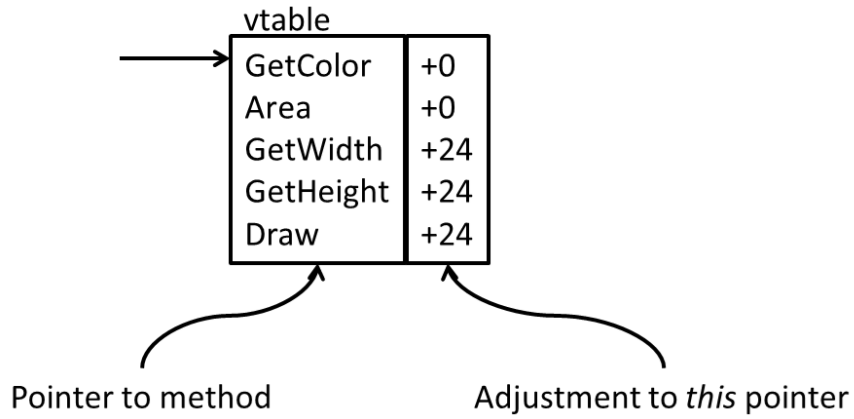


图 5.5 - vtable 与 this 指针的调整

一个方法调用现在的过程:

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。

这种方法也可用于实现接口。由于接口只有方法，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，而且很可能更快，但每个对象实例增加了开销。

最坏的情况下，若类有一个 64 位的数据字段，但是实现了 10 个接口，则对象需要 96 字节的内存:8 字节用于类本身的虚表指针，8 字节用于数据成员，10 * 8 字节用于每个接口的虚表指针。

为了支持对对象进行有意义的比较并执行运行时类型测试，需要首先规范指向对象的指针。若在虚函数表中添加一个字段，包含对象顶部的偏移量，则可以调整指针以指向实际对象。在 Circle 类的实值表中，该偏移量为 0，但在嵌入 GraphicObj 类的实值表中则不是，这是否需要实现取决于源码语言的语义。

LLVM 本身并不支持面向对象特性的特殊实现。如本节所示，可以使用可用的 LLVM 数据类型实现所有方法。正如具有单继承的 LLVM IR 示例一样，当涉及多继承时，IR 可能会变得更加冗长。若想尝试一种新方法，一个好方法就是先用 C 语言做一个原型。所需的指针操作很快翻译成 LLVM IR，但在高级语言中对功能的推理会更容易。

利用本节中获得的知识，可以在自己的代码生成器中实现将编程语言中常见的所有 OOP 结构简化为 LLVM IR。现在，已经掌握了如何表示单继承、使用接口的单继承或内存中的多重继承，以及如何实现类型测试和如何查找虚拟函数，这些都是 OOP 语言的核心概念。

5.5. 总结

在本章中，了解了如何将聚合数据类型和指针转换为 LLVM IR 代码，应用程序二进制接口的复杂性，将类和虚函数转换为 LLVM IR 的不同方法。有了本章的知识，您将能够为编程语言创建一个 LLVM IR 代码生成器。

下一章中，将了解一些有关 IR 生成的高级技术。异常处理在现代编程语言中相当常见，LLVM 对此提供了一些支持。将类型信息添加到指针可以进行某些优化。最后，对许多开发人员来说，调试应用程序的能力必不可少，因此还需要在代码生成器中添加功能，以生成调试元数据。

第 6 章 生成 IR 代码的进阶知识

通过前面章节中介绍的 IR 生成，已经可以实现编译器中所需的大部分功能。本章中，将研究一些编译器中经常出现的高级主题。例如，许多现代语言都使用异常处理，将研究如何将其转换为 LLVM IR。

为了支持 LLVM 优化器，使其能够在某些情况下生成更好的代码，必须向 IR 代码添加类型的元数据。添加的调试元数据，可使编译器用户能够使用源代码级调试工具进行调试。

本章中，将了解以下内容:

- 抛出和捕获异常: 如何在编译器中实现异常处理
- 为基于类型的别名分析生成元数据: 将更多的元数据到 LLVM IR，有助于 LLVM 更好地优化代码
- 添加调试元数据: 将实现向生成的 IR 代码添加调试信息所需的类

本章结束时，将了解异常处理，以及用于基于类型的别名分析和调试信息的元数据。

6.1. 抛出和捕获异常

LLVM IR 中的异常处理与平台支持密切相关，将使用 libunwind 来了解最常见的异常处理类型。C++ 可以充分发挥 libunwind 的潜力，先来看一个 C++ 示例，其中 bar() 函数可以抛出一个 int 或 double 值：

```
int bar(int x) {
    if (x == 1) throw 1;
    if (x == 2) throw 42.0;
    return x;
}
```

foo() 函数调用 bar()，但只处理抛出的 int 值，并且还声明只抛出 int 值：

```
int foo(int x) {
    int y = 0;
    try {
        y = bar(x);
    }
    catch (int e) {
        y = e;
    }
    return y;
}
```

抛出异常需要两次调用运行时库，可以在 bar() 函数中看到。__cxa_allocate_exception() 为异常分配内存，这个函数接受要分配的字节数作为参数。异常负载 (本例中的 int 或 double 值) 被复制到分配的内存中，再通过__cxa_throw() 引发异常。这个函数接受三个参数: 指向已分配异常的指针、有关有效负载的类型信息，以及指向析构函数的指针 (若异常有有效负载有的话)。__cxa_throw() 函数启动堆栈展开过程，并且永远不会返回。LLVM IR 中，对 int 值执行:

```
%eh = call ptr @__cxa_allocate_exception(i64 4)
store i32 1, ptr %eh
call void @__cxa_throw(ptr %eh, ptr @_ZTIi, ptr null)
unreachable
```

_ZTIi 是描述 int 类型的类型信息。对于 double 类型，则为_ZTId。

目前，还没有做特定于 LLVM 的工作。这一点在 foo() 函数中发生了变化，对 bar() 的调用可能引发异常。若是 int 类型异常，控制流必须转移到 catch 的 IR 代码。所以，必须使用 invoke 指令，而非 call 指令:

```
%y = invoke i32 @_Z3bari(i32 %x) to label %next
                                 unwind label %lpad
```

这两条指令的区别在于，invoke 有两个相关联的标签。在函数正常结束时，第一个标号表示执行继续，通常以 ret 指令结束。示例代码中，这个标签名为%next。若发生异常，则在标号为%lpad 的“着陆垫”上继续执行。

著陆台是一个基本的块，必须以 landingpad 指令开始。landingpad 指令向 LLVM 提供有关处理过的异常类型的信息。例如，一个着陆台可能是这样的:

```
lpad:
%exc = landingpad { ptr, i32 }
            cleanup
            catch ptr @_ZTIi
            filter [1 x ptr] [ptr @_ZTIi]
```

这里有三种可能的操作类型：

- cleanup: 表示存在用于清理当前状态的代码，用于调用局部对象的析构函数。若此标记存在，则在堆栈展开期间调用着陆垫指令。
- catch: 是类型-值对的列表，表示可以处理的异常类型。若在此列表中找到抛出的异常类型，则调用着陆垫指令。在 foo() 函数的情况下，该值是指向 int 类型的 C++ 运行时类型信息的指针，类似于__cxa_throw() 函数的参数。
- filter: 指定一个异常类型数组。若在数组中找不到当前异常的异常类型，则调用着陆垫指令，用于实现 throw() 规范。对于 foo() 函数，数组只有一个成员——int 类型的类型信息。

着陆平垫指令的结果类型是 {ptr, i32 } 结构。第一个元素是指向抛出异常的指针，第二个元素是类型选择器。从结构体中进行提取:

```
%exc.ptr = extractvalue { ptr, i32 } %exc, 0
%exc.sel = extractvalue { ptr, i32 } %exc, 1
```

类型选择器是一个数字，可以帮助确定为什么要调用着陆垫指令。若当前异常类型与 landingpad 指令的 catch 部分给出的异常类型匹配，则该值为正值。若当前异常类型不匹配过滤部分给出的值，则值为负。若应该调用清理代码，则为 0。

类型选择器是类型信息表的偏移量，由着陆垫指令的 catch 和 filter 部分给出的值构造而成。优化过程中，多个着陆垫指令可以合并为一个，所以在 IR 级别上该表的结构是未知的。要检索给定类型的类型选择器，需要调用内部的 @llvm.eh.typeid.for 的函数。需要它来检查类型选择器值是否与 int 的类型信息相对应，以便执行 catch (int e) {} 块中的代码:

```
%tid.int = call i32 @llvm.eh.typeid.for(ptr @_ZTIi)
%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() 实现的。__cxa_begin_catch() 函数需要一个参数——当前异常——是着陆垫指令返回的值之一，返回一个指向异常负载的指针——本例中是 int 值。

__cxa_end_catch() 函数标志着异常处理的结束，并释放用__cxa_allocate_exception() 分配的内存。注意，若在 catch 块中抛出另一个异常，则运行时行为会复杂得多。异常处理方法如下:

```
catchint:
%payload = call ptr @__cxa_begin_catch(ptr %exc.ptr)
%retval = load i32, ptr %payload
call void @__cxa_end_catch()
br label %return
```

若当前异常的类型与 throws() 声明中的列表不匹配，则调用意外异常处理程序，需要再次检查类型选择器:

```
filterorcleanup:
%tst.blzero = icmp slt i32 %exc.sel, 0
br i1 %tst.blzero, label %filter, label %cleanup
```

若类型选择器的值小于 0，则调用该处理程序:

```
filter:
call void @__cxa_call_unexpected(ptr %exc.ptr) #4
unreachable
```

同样，该处理程序也不会返回。

这种情况下不需要清理，所有清理代码要做的就是恢复堆栈展开的执行:

```
cleanup:
resume { ptr, i32 } %exc
```

libunwind 驱动堆栈展开过程，但没有绑定到语言，依赖于语言的处理在个性函数中完成。对于 Linux 上的 C++，个性函数 (personality function) 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。每个需要参与堆栈展开的函数都有一个个性函数。此个性函数分析函数是否捕获异常、是否具有不匹配的筛选器列表或是否需要清理调用，将这些信息返回给解卷器，解卷器采取相应的行动。在 LLVM IR 中，个性函数的指针是作为函数定义的一部分给出的:

```
define i32 @_Z3fooi(i32) personality ptr @__gxx_personality_v0
```

至此，异常处理工具就完成了。

要在编译器中为编程语言使用异常处理，最简单的策略是利用现有的 C++ 运行时函数。这还有一个好处，就是异常可以与 C++ 互操作。缺点是将一些 C++ 运行时绑定到语言的运行时中，尤其是内存管理。若想避免这种情况，则需要创建自己的_cxa_函数。不过，想必还是会想要使用 libunwind，因为它提供了堆栈解旋机制:

1. 来看看如何创建这个 IR，在第 2 章中创建了 calc 表达式编译器。将扩展表达式编译器的代码生成器，以便在执行除零操作时引发并处理异常。生成的 IR 将检查除法的除数是否为 0。若为 true，则会引发异常。还将在函数中添加一个着陆垫指令，用来捕获异常并打印 Divide by 0 ! 到控制台，并结束计算。这个简单的例子中，不需要使用异常处理，但可以让专注于代码生成过程。必须将所有代码添加到 CodeGen.cpp 文件中，首先添加所需的新字段和一些辅助方法，需要存储__cxa_allocate_exception() 和__cxa_throw() 函数的 LLVM 声明，其由函数类型和函数本身组成。需要一个 GlobalVariable 实例来保存类型信息，还需要引用控制着陆垫指令的基本块和包含不可达指令的基本块:

```
GlobalVariable *TypeInfo = nullptr;
FunctionType *AllocEHFty = nullptr;
Function *AllocEHFn = nullptr;
FunctionType *ThrowEHFty = nullptr;
Function *ThrowEHFn = nullptr;
BasicBlock *LPadBB = nullptr;
BasicBlock *UnreachableBB = nullptr;
```

2. 还要添加一个新辅助函数，来比较两个值的 IR。createICmpEq() 函数接受 Left 和 Right 值作为参数进行比较，创建一个比较指令来测试值是否相等，并为两个基本块创建一个分支指令，用于相等和不相等的情况，这两个基本块通过 TrueDest 和 FalseDest 参数中的引用返回。此外，新基本块的标签可以在 TrueLabel 和 FalseLabel 参数中给出:

```
void createICmpEq(Value *Left, Value *Right,
                  BasicBlock *&TrueDest,
                  BasicBlock *&FalseDest,
                  const Twine &TrueLabel = "",
                  const Twine &FalseLabel = "") {
    Function *Fn =
        Builder.GetInsertBlock()->getParent();
    TrueDest = BasicBlock::Create(M->getContext(),
                                  TrueLabel, Fn);
    FalseDest = BasicBlock::Create(M->getContext(),
                                   FalseLabel, Fn);
    Value *Cmp = Builder.CreateCmp(CmpInst::ICMP_EQ,
                                   Left, Right);
    Builder.CreateCondBr(Cmp, TrueDest, FalseDest);
}
```

3. 要使用运行时中的函数，需要创建几个函数声明。在 LLVM 中，函数类型给出了签名，并且必须构造函数本身，使用 createFunc() 方法创建这两个对象。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，变量参数列表标志设置为 false，表示参数列表中没有变量部分:

```
void createFunc(FunctionType *&Fty, Function *&Fn,
                const Twine &N, Type *Result,
                ArrayRef<Type *> Params = None,
                bool IsVarArgs = false) {
    Fty = FunctionType::get(Result, Params, IsVarArgs);
    Fn = Function::Create(
        Fty, GlobalValue::ExternalLinkage, N, M);
}
```

完成这些准备工作后，就可以生成抛出异常的 IR 了。

6.1.1. 抛出异常

为了生成引发异常的 IR 代码，将添加 addThrow() 方法。这个新方法需要初始化新字段，然后通过__cxa_throw() 函数生成引发异常的 IR。所引发异常的有效负载为 int 类型，可以设置为任意值。下面是需要编写的代码:

1. 新的 addThrow() 方法首先检查 TypeInfo 字段是否已初始化。若还没有初始化，则会创建一个 i8 指针类型的全局外部常量_ZTIi。这代表了描述 C++ int 类型的元数据:

```
void addThrow(int PayloadVal) {
    if (!TypeInfo) {
        TypeInfo = new GlobalVariable(
        *M, Int8PtrTy,
        /*isConstant=*/true,
        GlobalValue::ExternalLinkage,
        /*Initializer=*/nullptr, "_ZTIi");
```

2. 初始化继续使用辅助 createFunc() 方法为__cxa_allocate_exception() 和__cxa_throw() 函数创建 IR 声明:

```
        createFunc(AllocEHFty, AllocEHFn,
                "__cxa_allocate_exception", Int8PtrTy,
                {Int64Ty});
        createFunc(ThrowEHFty, ThrowEHFn, "__cxa_throw",
                VoidTy,
                {Int8PtrTy, Int8PtrTy, Int8PtrTy});
```

3. 使用异常处理的函数需要一个个性函数，有助于堆栈展开。添加了 IR 代码来声明 C++ 库中的__gxx_personality_v0() 个性函数，并将其设置为当前函数的个性例程。当前函数没有存储为一个字段，可以使用 Builder 实例来查询当前的基本块，其将函数存储为 Parent 字段:

```
        FunctionType *PersFty;
        Function *PersFn;
```

```
        createFunc(PersFty, PersFn,
                   "__gxx_personality_v0", Int32Ty, std::nulopt,
                   true);
        Function *Fn =
        Builder.GetInsertBlock()->getParent();
        Fn->setPersonalityFn(PersFn);
```

4. 接下来，必须创建并填充着陆台指令的基本块。首先，需要保存指向当前基本块的指针，再创建一个新的基本块，在构建器中进行设置，以便可以用作插入指令的基本块，并调用 addLandingPad() 方法。该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。这段代码填充了着陆垫指令的基本块:

```
        BasicBlock *SaveBB = Builder.GetInsertBlock();
        LPadBB = BasicBlock::Create(M->getContext(),
                                    "lpad", Fn);
        Builder.SetInsertPoint(LPadBB);
        addLandingPad();
```

5. 初始化部分通过创建保存不可达指令的基本块来完成，并创建基本块并将其设置为构建器中的插入点。然后，可以将不可达指令添加到其中，再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:

```
        UnreachableBB = BasicBlock::Create(
        M->getContext(), "unreachable", Fn);
        Builder.SetInsertPoint(UnreachableBB);
        Builder.CreateUnreachable();
        Builder.SetInsertPoint(SaveBB);
    }
```

6. 要抛出异常，通过调用__cxa_allocate_exception() 函数为异常和负载分配内存。我们的有效负载是 C++ int 类型，通常大小为 4 字节，再为 size 创建一个常量无符号值，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，所以只需要创建调用指令:

```
    Constant *PayloadSz =
        ConstantInt::get(Int64Ty, 4, false);
    CallInst *EH = Builder.CreateCall(
        AllocEHFty, AllocEHFn, {PayloadSz});
```

7. 接下来，将 PayloadVal 值存储在分配的内存中，需要通过调用 ConstantInt::get() 函数创建一个 LLVM IR 常量。所述指向所分配内存的指针为 i8 指针类型; 为了存储 i32 类型的值，需要创建一个 bitcast 指令来强制转换该类型:

```
    Value *PayloadPtr =
        Builder.CreateBitCast(EH, Int32PtrTy);
    Builder.CreateStore(
        ConstantInt::get(Int32Ty, PayloadVal, true),
        PayloadPtr);
```

8. 最后，必须通过调用__cxa_throw() 函数来抛出异常。由于这个函数引发了一个异常，该异常也在同一个函数中处理，所以需要使用 invoke 指令，而非 call 指令。与调用指令不同，因为它有两个后继基本块，所以调用指令会结束一个基本块，这些是 UnreachableBB 和 LPadBB 基本块。若函数没有引发异常，控制流将转移到 UnreachableBB 基本块。由于__cxa_throw() 函数的设计，因为控制流会转移到 LPadBB 基本块来处理异常，所以这种情况永远不会发生。这样就完成了 addThrow() 方法的实现:

```
    Builder.CreateInvoke(
    ThrowEHFty, ThrowEHFn, UnreachableBB, LPadBB,
    {EH,
     ConstantExpr::getBitCast(TypeInfo, Int8PtrTy),
     ConstantPointerNull::get(Int8PtrTy)});
}
```

接下来，将添加代码来生成处理异常的 IR。

6.1.2. 捕捉异常

要生成捕获异常的 IR 代码，必须添加 addLandingPad() 方法。生成的 IR 从异常中提取类型信息。若匹配 C++ int 类型，则通过打印 Divide by 0! 到控制台来处理异常并从函数中返回。若类型不匹配，只需执行 resume 指令，该指令将控制转移回运行时。由于调用层次结构中没有其他函数来处理此异常，因此将终止应用程序的运行。以下步骤描述了生成用于捕获异常的 IR:

1. 生成的 IR 中，需要从 C++ 运行时库中调用__cxa_begin_catch() 和__cxa_end_catch() 函数。为了输出错误消息，将从 C 运行时库生成对 puts() 函数的调用。此外，为了从异常中获取类型信息，必须生成对 llvm.eh.typeid.for 指令的调用。还需要 FunctionType 和 Function 实例，我们将利用 createFunc() 方法来创建它们:

```
void addLandingPad() {
    FunctionType *TypeIdFty; Function *TypeIdFn;
    createFunc(TypeIdFty, TypeIdFn,
                "llvm.eh.typeid.for", Int32Ty,
                {Int8PtrTy});
    FunctionType *BeginCatchFty; Function *BeginCatchFn;
    createFunc(BeginCatchFty, BeginCatchFn,
                "__cxa_begin_catch", Int8PtrTy,
                {Int8PtrTy});
    FunctionType *EndCatchFty; Function *EndCatchFn;
    createFunc(EndCatchFty, EndCatchFn,
                "__cxa_end_catch", VoidTy);
    FunctionType *PutsFty; Function *PutsFn;
    createFunc(PutsFty, PutsFn, "puts", Int32Ty,
                {Int8PtrTy});
```

2. 着陆台指令是我们生成的第一条指令，结果类型是一个包含 i8 指针和 i32 类型字段的结构体。这个结构是通过调用 StructType::get() 函数生成的。此外，由于需要处理 C++ int 类型的异常，还需要将 this 作为子句添加到 landingpad 指令中，该指令必须是 i8 指针类型的常量，所以需

要生成一个位转换指令来将 TypeInfo 值转换为这种类型，再将指令返回的值存储在 Exc 变量中，供后续使用:

```
    LandingPadInst *Exc = Builder.CreateLandingPad(
        StructType::get(Int8PtrTy, Int32Ty), 1, "exc");
    Exc->addClause(
        ConstantExpr::getBitCast(TypeInfo, Int8PtrTy));
```

3. 接下来，从返回值中提取类型选择器。调用 llvm.eh.typeid.for 指令，检索 TypeInfo 字段的类型 ID，表示 C++ int 类型。使用这个 IR，已经生成了两个值。我们需要进行比较，以决定是否可以处理异常:

```
    Value *Sel =
        Builder.CreateExtractValue(Exc, {1}, "exc.sel");
    CallInst *Id =
        Builder.CreateCall(TypeIdFty, TypeIdFn,
                           {ConstantExpr::getBitCast(
                               TypeInfo, Int8PtrTy)});
```

4. 要生成用于比较的 IR，必须调用 createICmpEq() 函数。这个函数还生成了两个基本块，将它们存储在 TrueDest 和 FalseDest 变量中:

```
    BasicBlock *TrueDest, *FalseDest;
    createICmpEq(Sel, Id, TrueDest, FalseDest, "match",
                 "resume");
```

5. 若这两个值不匹配，控制流将在 false 基本块处继续。这个基本块只包含一个 resume 指令，将控制权交还给 C++ 运行时:

```
    Builder.SetInsertPoint(FalseDest);
    Builder.CreateResume(Exc);
```

6. 若这两个值相等，则控制流在 TrueDest 基本块处继续。生成 IR 代码，以便从存储在 Exc 变量中的着陆平台指令的返回值中，提取指向异常的指针。生成对__cxa_begin_catch() 函数的调用，将指向异常的指针作为参数传递。这表示在运行时开始处理异常:

```
    Builder.SetInsertPoint(TrueDest);
    Value *Ptr =
        Builder.CreateExtractValue(Exc, {0}, "exc.ptr");
    Builder.CreateCall(BeginCatchFty, BeginCatchFn,
                       {Ptr});
```

7. 再用 puts() 函数来处理异常，将消息输出到控制台，这里通过调用 CreateGlobalStringPtr() 函数生成一个指向该字符串的指针，在生成的调用中将该指针作为参数传递给 puts() 函数:

```
    Value *MsgPtr = Builder.CreateGlobalStringPtr(
        "Divide by zero!", "msg", 0, M);
    Builder.CreateCall(PutsFty, PutsFn, {MsgPtr});
```

8. 现在已经处理了异常，必须生成对__cxa_end_catch() 函数的调用，以通知运行时有关它的信息。最后，从函数返回一个 aret 指令:

```
    Builder.CreateCall(EndCatchFty, EndCatchFn);
    Builder.CreateRet(Int32Zero);
}
```

使用 addThrow() 和 addLandingPad() 函数，可以生成引发异常和处理异常的 IR，但仍需要添加 IR 来检查除数是否为 0。

我们将在下一节中讨论这一点。

6.1.3. 将异常处理代码集成到应用程序中

除法的 IR 是在 visit(BinaryOp &) 方法中生成的，必须生成一个 IR 来将除数与 0 进行比较，而不是仅仅生成一个 sdiv 指令。若除数为 0，则控制流继续在基本块中运行，从而引发异常；否则，控制流将在带有 sdiv 指令的基本块中继续。在 createICmpEq() 和 addThrow() 函数的帮助下，可以编写如下代码:

```
    case BinaryOp::Div:
        BasicBlock *TrueDest, *FalseDest;
        createICmpEq(Right, Int32Zero, TrueDest,
                     FalseDest, "divbyzero", "notzero");
        Builder.SetInsertPoint(TrueDest);
        addThrow(42); // Arbitrary payload value.
        Builder.SetInsertPoint(FalseDest);
        V = Builder.CreateSDiv(Left, Right);
        break;
```

代码生成部分现在已经完成。要构建应用程序，必须切换到构建目录并运行 ninja 工具:

```
$ ninja
```

构建完成后，可以用 with a: 3/a 表达式检查生成的 IR:

```
$ src/calc "with a: 3/a"
```

将看到抛出和捕获异常所需的 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。使用表达式计算器的运行时函数将 rtcalc.c 文件重命名为 rtcalc.cpp，并在文件中的每个函数前面添加 extern ”C”。然后，使用 llc 工具将生成的 IR 转换为目标文件，并使用 clang++ 编译器创建可执行文件:

```
$ src/calc "with a: 3/a" | llc -filetype obj -o exp.o
$ clang++ -o exp exp.o ../rtcalc.cpp
```

现在，可以用不同的值对程序进行测试:

```
$ ./exp
Enter a value for a: 1
The result is: 3
$ ./exp
Enter a value for a: 0
Divide by zero!
```

第二次运行时，输入为 0，这会引发异常——像预期的那样工作了!

本节中，了解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，所使用的类型信息和 catch 子句的数量取决于编译器的输入，但需要生成的 IR 仍然遵循本节中提供的模式。

添加元数据是向 LLVM 提供进一步信息的另一种方式。下一节中，将添加类型元数据，以便在某些情况下支持 LLVM 优化器。

6.2. 为基于类型的别名分析生成元数据

两个指针可能指向同一个存储单元，在这一点上它们彼此别名。内存在 LLVM 模型中没有类型，这使得优化器很难确定两个指针是否相互别名。若编译器可以证明两个指针不会相互别名，就可以进行更多的优化。在下一节中，我们将更深入地研究这个问题，并研究在实现这种方法之前添加元数据将有什么好处。

6.2.1. 理解对元数据的需求

为了演示这个问题，来看看下面的函数:

```
void doSomething(int *p, float *q) {
    *p = 42;
    *q = 3.1425;
}
```

优化器不能决定指针 p 和 q 是否指向相同的内存单元。优化过程中，可以执行一个重要的分析，称为别名分析。若 p 和 q 指向相同的存储单元，其互为别名。此外，若优化器可以证明两个指针永远不会相互别名，这就提供了优化机会。例如，在 doSomething() 函数中，存储可以在不改变结果的情况下重新排序。

此外，若一种类型的变量可以是另一种不同类型变量的别名，则取决于源语言的定义。注意，语言也可能包含不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强制转换。

LLVM 开发人员选择的解决方案是在加载和存储指令中添加元数据。添加的元数据有两个目的:

- 首先，定义了类型层次结构，在此基础上类型可以别名另一个类型
- 其次，描述了加载或存储指令中的内存访问

来看一下 C 中的类型层次结构。每种类型的层次结构都从一个根节点开始，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，可以在相同的 LLVM 模块中使

用不同的类型层次结构，并且 LLVM 可以安全地假设这些类型可以别名。在根节点下面是标量类型的节点，聚合类型的节点不添加到根节点，但会引用标量类型和其他聚合类型。Clang 定义 C 语言的层次结构如下:

- 根节点称为简单的 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，所有指针都可以转换为指向 char 的指针。
- char 节点下面是其他标量类型的节点和所有指针的类型，称为 any 指针。

除此之外，聚合类型可定义为成员类型和偏移量的序列。

这些元数据定义会添加到加载和存储指令的访问标记中。访问标记由三部分组成: 基本类型、访问类型和偏移量。根据基本类型的不同，访问标签描述内存访问有两种可能的方式:

1. 若基类型是聚合类型，则访问标记描述具有必要访问类型的 struct 成员的内存访问，并位于给定的偏移量。
2. 若基类型是标量类型，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，就可以在访问标记上定义一个关系，用于计算两个指针是否可以相互别名。来仔细看看 (基类型，偏移量) 元组的直接父对象的选项:

1. 若基类型是标量类型且偏移量为 0，则直接父类型为 (父类型，0)，父类型为父节点的类型，如类型层次结构中定义的那样。若偏移量不为 0，则父级未定义。
2. 若基类型是聚合类型，则 (基类型，偏移量) 元组的直接父级是 (新类型，新偏移量) 元组，新类型是偏移量处成员的类型。新偏移量是新类型的偏移量，调整到其新起点。

这个关系的传递闭包就是父关系。两个内存访问 (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)，若 (基类型 1，偏移量 1) 和 (基类型 2，偏移量 2) 在父关系中相关，则相互别名，反之亦然。

用一个例子来说明这一点:

```
struct Point { float x, y; }
void func(struct Point *p, float *x, int *i, char *c) {
    p->x = 0; p->y = 0; *x = 0.0; *i = 0; *c = 0;
}
```

当使用标量类型的内存访问标记定义时，i 参数的访问标记是 (int, int, 0)，而 c 参数的访问标记是 (char, char, 0)。类型层次结构中，int 类型的节点的父节点是 char 节点，所以 (int, 0) 的直接父指针是 (char, 0)，并且两个指针都可以别名。对于 x 和 c 参数也是如此，但 x 和 i 参数是不相关的，所以它们不会相互别名。对结构体 Point 的 y 成员的访问是 (Point, float, 4)，其中 4 是该结构体中 y 成员的偏移量。(Point, 4) 的直接父变量是 (float, 0)，所以对 p->y 和 x 的访问可以别名；同理，参数 c 也可以使用别名。

6.2.2. LLVM 中创建 TBAA 元数据

创建元数据，必须使用 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 builder 类可以避免我们了解构造的内部

细节。

通过 createTBAARoot() 方法创建根节点，该方法需要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() 方法创建一个匿名的唯一根节点。

使用 createTBAAScalarTypeNode() 方法将标量类型添加到层次结构中，该方法将类型的名称和父节点作为参数。

另一方面，为聚合类型添加类型节点稍微复杂一些。createTBAAStructTypeNode() 方法接受类型的名称和字段列表作为参数。具体来说，字段以 std::pair<llvm::MDNode*, uint64_t> 实例的形式给出，其中第一个元素表示成员的类型，第二个元素表示 struct 中的偏移量。

使用 createTBAAStructTagNode() 方法创建访问标记，该方法将基类型、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。Instruction 类包含一个名为 setMetadata() 的方法，该方法用于添加各种基于类型的别名分析元数据。第一个参数必须是 llvm::LLVMContext::MD_tbaa 类型，第二个参数必须是访问标签。

有了这些知识，就可以向 tinylang 添加用于基于类型的别名分析 (TBAA) 的元数据了。

6.2.3. tinylang 中添加 TBAA 元数据

为了支持 TBAA，必须添加一个新的 CGTBAA 类，该类负责生成元数据节点。此外，我们使 CGTBAA 类成为 CGModule 类的成员，称其为 TBAA。

每条加载和存储指令都必须加注解，为此在 CGModule 类中创建了一个名为 decorateInst() 的新函数，这个函数尝试创建标记访问信息。若成功，则元数据将添加到相应的加载或存储指令。此外，这种设计还允许在不需要的情况下关闭元数据生成过程，例如在构建中关闭优化:

```
void CGModule::decorateInst(llvm::Instruction *Inst,
                            TypeDeclaration *Type) {
    if (auto *N = TBAA.getAccessTagInfo(Type))
        Inst->setMetadata(llvm::LLVMContext::MD_tbaa, N);
}
```

我们将新 CGTBAA 类的声明放在 include/tinylang/CodeGen/CGTBAA.h 头文件中，并将定义放在 lib/CodeGen/CGTBAA.cpp 文件中。除了 AST 定义，头文件还需要包含定义元数据节点和构建器的文件:

```
#include "tinylang/AST/AST.h"
#include "llvm/IR/MDBuilder.h"
#include "llvm/IR/Metadata.h"
```

CGTBAA 类需要存储一些数据成员:

1. 首先，需要缓存类型层次结构的根:

```
class CGTBAA {
    llvm::MDNode *Root;
```

2. 为了构造元数据节点，需要一个 MDBuilder 类的实例:

```
    llvm::MDBuilder MDHelper;
```

3. 最后，必须存储为一个类型生成的元数据以供重用:

```
    llvm::DenseMap<TypeDenoter *, llvm::MDNode *> MetadataCache;
    // ···
};
```

现在已经定义了构造所需的变量，必须添加创建元数据所需的方法:

1. 构造函数初始化数据成员:

```
CGTBAA::CGTBAA(CGModule &CGM)
    : CGM(CGM),
    MDHelper(llvm::MDBuilder(CGM.getLLVMCtx())),
    Root(nullptr) {}
```

2. 必须惰性地实例化类型层次结构的根，将其命名为 Simple tinylang TBAA:

```
llvm::MDNode *CGTBAA::getRoot() {
    if (!Root)
        Root = MDHelper.createTBAARoot("Simple tinylang TBAA");
    return Root;
}
```

3. 对于标量类型，必须在 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:

```
llvm::MDNode *
CGTBAA::createScalarTypeNode(TypeDeclaration *Ty,
                             StringRef Name,
                             llvm::MDNode *Parent) {
    llvm::MDNode *N =
        MDHelper.createTBAAScalarTypeNode(Name, Parent);
    return MetadataCache[Ty] = N;
}
```

4. 为记录创建元数据的方法更加复杂，所以必须枚举记录的所有字段。与标量类型类似，新的元数据节点存储在缓存中:

```
llvm::MDNode *CGTBAA::createStructTypeNode(
        TypeDeclaration *Ty, StringRef Name,
        llvm::ArrayRef<std::pair<llvm::MDNode *, uint64_t>> Fields) {
    llvm::MDNode *N =
        MDHelper.createTBAAStructTypeNode(Name, Fields);
    return MetadataCache[Ty] = N;
}
```

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。每个标量类型都是映射到根节点的唯一类型，并且将所有指针映

射到单个类型，再结构化类型引用这些节点。若不能映射类型，则返回 nullptr:

```
llvm::MDNode *CGTBAA::getTypeInfo(TypeDeclaration *Ty) {
    if (llvm::MDNode *N = MetadataCache[Ty])
        return N;

    if (auto *Pervasive =
            llvm::dyn_cast<PervasiveTypeDeclaration>(Ty)) {
        StringRef Name = Pervasive->getName();
        return createScalarTypeNode(Pervasive, Name, getRoot());
    }
    if (auto *Pointer =
            llvm::dyn_cast<PointerTypeDeclaration>(Ty)) {
        StringRef Name = "any pointer";
        return createScalarTypeNode(Pointer, Name, getRoot());
    }
    if (auto *Array =
            llvm::dyn_cast<ArrayTypeDeclaration>(Ty)) {
        StringRef Name = Array->getType()->getName();
        return createScalarTypeNode(Array, Name, getRoot());
    }
    if (auto *Record =
            llvm::dyn_cast<RecordTypeDeclaration>(Ty)) {
        llvm::SmallVector<std::pair<llvm::MDNode *, uint64_t>,
        4> Fields;
        auto *Rec =
            llvm::cast<llvm::StructType>(CGM.convertType(Record));
        const llvm::StructLayout *Layout =
        CGM.getModule()->getDataLayout().getStructLayout(Rec);

        unsigned Idx = 0;
        for (const auto &F : Record->getFields()) {
            uint64_t Offset = Layout->getElementOffset(Idx);
            Fields.emplace_back(getTypeInfo(F.getType()), Offset);
            ++Idx;
        }
        StringRef Name = CGM.mangleName(Record);
        return createStructTypeNode(Record, Name, Fields);
    }
    return nullptr;
}
```

6. 获取元数据的一般方法是 getAccessTagInfo()。要获取 TBAA 访问标记信息，必须添加对 getTypeInfo() 函数的调用。这个函数期望 TypeDeclaration 作为它的参数，可从生成元数据的指令中检索:

```
llvm::MDNode *CGTBAA::getAccessTagInfo(TypeDeclaration *Ty) {
    return getTypeInfo(Ty);
}
```

最后，为了生成 TBAA 元数据，只需要将元数据添加到 tinylang 中成的所有加载和存储指令上即可。

例如，在 CGProcedure:: writvariable() 中，存储一个全局变量使用一个存储指令:

```
    Builder.CreateStore(Val, CGM.getGlobal(D));
```

为了修饰这个特定的指令，需要用以下几行替换这一行，其中 decorateInst() 将 TBAA 元数据添加到这个存储指令中:

```
    auto *Inst = Builder.CreateStore(Val, CGM.getGlobal(D));
    // NOTE: V is of the VariableDeclaration class, and
    // the getType() method in this class retrieves the
    // TypeDeclaration that is needed for decorateInst().
    CGM.decorateInst(Inst, V->getType());
```

有了这些更改，就完成了 TBAA 元数据的生成。

现在，可以将示例 tinylang 文件编译为 LLVM 中间表示，以查看新实现的 TBAA 元数据。例如，以下文件 Person.mod:

```
MODULE Person;

TYPE
    Person = RECORD
                Height: INTEGER;
                Age: INTEGER
             END;

PROCEDURE Set(VAR p: Person);
BEGIN
    p.Age := 18;
END Set;

END Person.
```

本章的构建目录中构建的 tinylang 编译器可以用来生成这个文件的中间表示:

```
$ tools/driver/tinylang -emit-llvm ../examples/Person.mod
```

新生成的 Person.ll 文件中，可以看到存储指令是用本章中生成的 TBAA 元数据，其中元数据反映了最初声明的记录类型的字段:

```
; ModuleID = '../examples/Person.mod'
source_filename = "../examples/Person.mod"
target datalayout = "e-m:o-i64:64-i128:128-n32:64-S128"
target triple = "arm64-apple-darwin22.6.0"

```

```
define void @_t6Person3Set(ptr nocapture dereferenceable(16) %p) {
entry:
    %0 = getelementptr inbounds ptr, ptr %p, i32 0, i32 1
    store i64 18, ptr %0, align 8, !tbaa !0
    ret void
}

!0 = !{!"_t6Person6Person", !1, i64 0, !1, i64 8}
!1 = !{!"INTEGER", !2, i64 0}
!2 = !{!"Simple tinylang TBAA"}
```

既然了解了如何生成 TBAA 元数据，我们将在下一节中探索一个非常相似的主题: 生成调试元数据。

6.3. 生成调试元数据

为了允许源代码级调试，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，并使用 intrinsic 来跟踪变量值。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 格式生成调试信息。我们将在下一节中查看其总体结构。

6.3.1. 理解调试元数据的一般结构

为了描述通用结构，LLVM 使用的元数据类似于用于基于类型分析的元数据。静态结构描述了文件、编译单元、函数和词法块，以及使用的数据类型。

使用的主要类是 llvm::DIBuilder，使用 llvm/IR/DIBuilder 目录下的头文件来获取类声明。此构建器类提供了一个易于使用的接口来创建调试元数据。之后，元数据将添加到 LLVM 对象 (如全局变量) 中，或者在调试内部函数的调用中使用。下面是一些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描述一个文件。可以使用 createFile() 方法来创建，文件可以包含主编译单元，包含导入的声明。
- llvm::DICompileUnit: 用于描述当前编译单元。除其他事项外，还指定源语言、特定于编译器的生产者字符串、是否启用优化，当然还有 DIFile(编译单元驻留在其中)。可以通过 createCompileUnit() 来创建。
- llvm:: dissubprogram: 描述一个函数。这里最重要的信息是作用域 (通常是嵌套函数的 DICompileUnit 或 dissubprogram)、函数名、修改后的函数名和函数类型。可以通过调用 createFunction() 来创建。
- llvm::DILexicalBlock: 描述了一个词法块，并对许多高级语言中的块范围进行了建模。可以通过调用 createLexicalBlock() 来创建。

LLVM 对编译器翻译的语言不做任何假设，所以没有关于该语言的数据类型的信息。要支持源代码级调试，特别是在调试器中显示变量值，还必须添加类型信息。下面是一些重要的构念:

- 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，例如 tinylang 中的 INTEGER 或 C++ 中的 int。除了类型的名称之外，所需的参数还包括以位为单位的大小和编码——例如，若是有符号类型或无符号类型。
- 有几种方法可以构造复合数据类型的元数据，如 llvm::DIComposite 类所示。可以使用 createArrayType()、createStructType()、createUnionType() 和 createVectorType() 函数分别实例化数组、结构、联合和矢量数据类型的元数据。这些函数需要期望的参数，例如基类型和数组类型的订阅数量或结构类型的字段成员列表。
- 还有支持枚举、模板、类等的方法。

函数列表显示必须将源语言的每个细节添加到调试信息中，假设 llvm::DIBuilder 类的实例称为 DBuilder，还假设在一个名为 file 的文件中有一些 tinylang 源文件。/home/llvmuser 目录下的 File.mod 文件中，其第 5 行是 Func():INTEGER 函数，第 7 行包含一个局部 VAR i:INTEGER 声明。为此创建元数据，从文件的信息开始。这里，需要指定文件名和文件所在文件夹的绝对路径:

```
llvm::DIFile *DbgFile = DBuilder.createFile("File.mod",
                                            "/home/llvmuser");
```

该文件在 tinylang 中是一个模块，这使得它成为 LLVM 的编译单元。这包含了很多信息:

```
bool IsOptimized = false;
llvm::StringRef CUFlags;
unsigned ObjCRunTimeVersion = 0;
llvm::StringRef SplitName;
llvm::DICompileUnit::DebugEmissionKind EmissionKind =
    llvm::DICompileUnit::DebugEmissionKind::FullDebug;
llvm::DICompileUnit *DbgCU = DBuilder.createCompileUnit(
    llvm::dwarf::DW_LANG_Modula2, DbgFile, "tinylang",
    IsOptimized, CUFlags, ObjCRunTimeVersion, SplitName,
    EmissionKind);
```

此外，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，这样做的一个缺点是不能简单地添加新的源语言。要做到这一点，必须在 DWARF 委员会创建一个请求。请注意，调试器和其他调试工具也需要支持一种新语言——仅仅向枚举中添加一个新成员远远不够。

许多情况下，选择一种接近源语言的语言就足够了。在 tinylang 的例子中，这是 Modula-2，使用 DW_LANG_Modula2 作为语言标识符。编译单元保存在文件中，该文件由我们前面创建的 DbgFile 变量表示。

此外，调试信息可以携带有关生成器的信息，这些信息可以是编译器的名称和版本信息。这里，只是传递了 tinylang 字符串。若不想添加这些信息，则可以简单地使用一个空字符串作为参数。

下一组信息包括 IsOptimized 标志，指示编译器是否开启了优化。通常，这个标志来自于-O 命令行开关，可以使用 CUFlags 参数将其他参数设置传递给调试器。这里没有使用这个，直接传递一个空字符串。我们也不使用 Objective-C，所以将 0 作为 Objective-C 运行时参数。

通常，调试信息嵌入在我们正在创建的目标文件中。若想要将调试信息写入一个单独的文件，则 SplitName 参数必须包含这个文件的名称；否则，只需传递一个空字符串就足够了。最后，可以

定义应该发出的调试信息的级别。默认值是完整的调试信息，正如 FullDebug 枚举值的使用所表明的那样，但若只想生成行号，也可以选择 LineTablesOnly 值，或者根本不生成调试信息的 NoDebug 值。对于后者，最好一开始就不要创建调试信息。

我们的极简源码只使用 INTEGER 数据类型，这是一个带符号的 32 位值。为这种类型创建元数据很简单:

```
llvm::DIBasicType *DbgIntTy =
    DBuilder.createBasicType("INTEGER", 32,
        llvm::dwarf::DW_ATE_signed);
```

要为函数创建调试元数据，必须首先为签名创建类型，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，其中所有类型的参数按源顺序排列，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。在本例中，还可以静态地构造元数据。这对于内部函数很有用，比如模块初始化。通常，这些函数的参数是已知的，编译器编写器可以硬编码它们:

```
llvm::Metadata *DbgSigTy = {DbgIntTy};
llvm::DITypeRefArray DbgParamsTy =
            DBuilder.getOrCreateTypeArray(DbgSigTy);
llvm::DISubroutineType *DbgFuncTy =
            DBuilder.createSubroutineType(DbgParamsTy);
```

我们的函数有 INTEGER 返回类型，没有其他参数，因此 DbgSigTy 数组只包含指向该类型元数据的指针。该静态数组被转换为类型数组，然后用于创建函数的类型。

函数本身需要更多的数据:

```
unsigned LineNo = 5;
unsigned ScopeLine = 5;
llvm::DISubprogram *DbgFunc = DBuilder.createFunction(
    DbgCU, "Func", "_t4File4Func", DbgFile, LineNo,
    DbgFuncTy, ScopeLine, llvm::DISubprogram::FlagPrivate,
    llvm::DISubprogram::SPFlagLocalToUnit);
```

函数属于编译单元，在示例中，编译单元存储在 DbgCU 变量中。需要在源文件中指定函数的名称，即 Func，并且修改后的名称存储在目标文件中。这些信息有助于调试器定位函数的机器码。根据 tinylang 规则，修改后的名称为_t4File4Func，还必须指定包含函数的文件。

乍一看，这可能令人惊讶，但想想 C 和 C++ 中的包含机制: 函数可以存储在不同的文件中，然后在主编译单元中使用 #include 包含该文件。这里的情况并非如此，我们使用与编译单元使用的文件相同的文件。接下来，传递函数的行号和函数类型。函数的行号不能是函数的词法作用域开始的行号，可以指定不同的 ScopeLine。函数也有保护，在这里用 FlagPrivate 值指定它来指示 private 函数。函数保护的其他可能值是 FlagPublic 和 FlagProtected，分别用于 public 和 protected 域内的函数。

除了保护级别，还可以指定其他标志。例如，FlagVirtual 表示虚函数，FlagNoReturn 表示该函数不返回给调用者。可以在 LLVM include 文件中找到可能值的完整列表，即 llvm/include/llvm/IR/DebugInfoFlags.def。

最后，可以指定特定于函数的标志。最常用的标志是 SPFlagLocalToUnit 值，表明该函数是编译单元的本地函数。MainSubprogram 值也经常使用，表明这个函数是应用程序的主要函数。前面提到的 LLVM 包含文件，还列出了与特定于函数的标志相关的所有可能值。

目前，我们只创建了引用静态数据的元数据。变量是动态的，因此将在下一节中探讨如何将静态元数据添加到 IR 代码中以访问变量。

6.3.2. 跟踪变量及其值

前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。llvm::DIBuilder 类的 createGlobalVariableExpression() 函数创建描述全局变量的元数据。这包括源文件中变量的名称、修改过的名称、源文件等。LLVM IR 中的全局变量由 GlobalVariable 类的实例表示。这个类有一个名为 addDebugInfo() 的方法，可将从 createGlobalVariableExpression() 返回的元数据节点与全局变量关联起来。

对于局部变量，需要采取另一种方法。因为只知道值，所以 LLVM IR 不知道表示局部变量的类。LLVM 社区开发的解决方案是将对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，所以可以用它做一些神奇的事情。大多数情况下，内在函数不会导致机器级别的子例程调用。这里，函数调用可将元数据与值关联起来，函数内最重要的调试元数据是 llvm.dbg.declare 和 llvm.dbg.value。

llvm.dbg.declare 内部变量提供信息，由前端生成，用于声明局部变量。本质上，this 描述了局部变量的地址。优化过程中，通道可以用 (可能多次) 调用 llvm.dbg 来替换这个内在的调用。值来保存调试信息并跟踪本地源变量。优化之后，llvm.dbg.declare 用于描述局部变量在内存中的程序点，所以可能会出现多次调用。

另一方面，只要将局部变量设置为新值，就会调用 llvm.dbg.value。这个内部变量描述的是局部变量的值，而不是它的地址。

这一切是如何运作的呢?LLVM IR 表示和通过 LLVM::DIBuilder 类进行的编程创建略有不同，因此将对两者进行研究。

继续上一节的示例，将使用 alloca 指令为 function 函数中的 I 变量分配本地存储:

```
@i = alloca i32
```

之后，必须添加对 llvm.dbg.declare 的调用:

```
call void @llvm.dbg.declare(metadata ptr %i,
                metadata !1, metadata !DIExpression())
```

第一个参数是局部变量的地址。第二个参数是描述局部变量的元数据，通过调用本地变量 createAutoVariable() 或调用 llvm::DIBuilder 类参数 createParameterVariable() 来创建。最后，第三个参数描述一个地址表达式，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() 方法来为本地 @i 变量分配存储空间:

```
llvm::Type *IntTy = llvm::Type::getInt32Ty(LLVMCtx);
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。

局部变量也需要通过元数据来描述:

```
llvm::DILocalVariable *DbgLocalVar =
    Dbuilder.createAutoVariable(DbgFunc, "i", DbgFile,
                                7, DbgIntTy);
```

使用上一节中的值，可以指定该变量是 DbgFunc 函数的一部分，称为 i，在 DbgFile 文件的第 7 行中定义，并且是 DbgIntTy 类型。

最后，使用 llvm.dbg.declare 内部变量将调试元数据与变量的地址关联起来。使用 llvm::DIBuilder 避免添加调用的所有细节:

```
llvm::DILocation *DbgLoc =
                llvm::DILocation::get(LLVMCtx, 7, 5, DbgFunc);
DBuilder.insertDeclare(Val, DbgLocalVar,
                       DBuilder.createExpression(), DbgLoc,
                       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，保存与作用域相关联的位置的行和列。此外，insertDeclare() 方法将调用添加到 LLVM IR 的内部函数。就这个函数的参数而言，需要存储在 Val 中的变量的地址，以及存储在 DbgValVar 中的变量的调试元数据。还传递了一个空地址表达式和前面创建的调试位置，与普通指令一样，需要指定将调用插入到哪个基本块中。若指定一个基本块，则在末尾插入调用。或者，可以指定一条指令，然后在该指令之前插入调用。还有一个指向 alloca 指令的指针，这是插入到底层基本块中的最后一条指令。因此，可以使用这个基本块，并且调用会添加在 alloca 指令之后。

若局部变量的值发生了变化，必须向 IR 添加对 llvm.dbg.value 的调用，以设置局部变量的新值。可以使用 llvm::DIBuilder 类的 insertValue() 方法来实现。

当实现函数的 IR 生成时，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。需要添加调试信息，所以使用 llvm.dbg.value 的频率比在 clang 生成的 IR 中看到的要高得多。

若变量没有专用存储空间，而是更大的聚合类型的一部分，该怎么办? 可能出现这种情况的一种情况是使用嵌套函数。要实现对调用者堆栈帧的访问，必须收集结构中所有使用的变量，并将指向该记录的指针传递给调用函数。调用的函数内部，可以引用调用方的变量，就好像是函数的局部变量一样。不同的是，这些变量现在是总量的一部分。

对 llvm.dbg.declare 的调用中，若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。但若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。

嵌套帧的情况下，需要计算帧中的偏移量。需要访问 DataLayout 实例，可以从创建 IR 代码的 LLVM 模块获得该实例。若 llvm::Module 实例命名为 Mod，保存嵌套帧结构的变量命名为 frame，并且是 llvm::StructType 类型，可以通过以下方式访问帧的第三个成员。这个访问会提供成员的偏移量:

```
const llvm::DataLayout &DL = Mod->getDataLayout();
uint64_t Ofs = DL.getStructLayout(Frame)->getElementOffset(3);
```

此外，表达式是由一系列操作创建的。要访问帧的第三个成员，调试器需要向基指针添加偏移量。作为一个例子，需要创建一个数组和相关信息:

```
llvm::SmallVector<int64_t, 2> AddrOps;
AddrOps.push_back(llvm::dwarf::DW_OP_plus_uconst);
AddrOps.push_back(Offset);
```

从这个数组中，可以创建表达式，然后传递给 llvm.dbg.declare，而不是空表达式:

```
llvm::DIExpression *Expr = DBuilder.createExpression(AddrOps);
```

用户并不限于使用这种偏移操作。DWARF 知道许多不同的运算符，从而可以创建相当复杂的表达式。可以在 LLVM 包含文件 (llvm/include/llvm/BinaryFormat/Dwarf.def) 中找到完整的操作符列表。

此时，可以为变量创建调试信息。为了使调试器能够遵循源代码中的控制流，还需要提供行号信息。这是下一节的主题。

6.3.3. 添加行号

调试器允许程序员逐行调试应用程序，所以调试器需要知道哪些机器指令属于源代码中的哪一行，LLVM 允许为每条指令添加一个源位置。在前一节中，创建了 llvm::DILocation 类型的位置信息，调试位置提供的信息不仅仅是行、列和范围。若需要，可以指定该行内联到的作用域。还可以指出此调试位置属于隐式代码——即前端生成，但不在源码中的代码。

此信息可以添加到指令之前，必须将调试位置包装在 llvm::DebugLoc 对象中。为此，必须简单地将从 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不会改变，但可以在优化期间删除为源码级语句或表达式生成的机器码，这种封装有助于处理这些变化。

添加行号信息主要归结为从 AST 检索行号信息，并将其添加到生成的指令中。指令类具有 setDebugLoc() 方法，该方法将位置信息添加到指令上。

在下一节中，将学习如何生成调试信息，并将其添加到 tinylang 编译器中。

6.3.4. 使 tinylang 支持调试

我们将调试元数据的生成封装在新的 CGDebugInfo 类中。此外，将声明放在 tinylang/CodeGen/CGDebugInfo.h 头文件中，并将定义放在 tinylang/CodeGen/CGDebugInfo.cpp 文件中。

CGDebugInfo 类有五个重要的成员，需要一个对模块代码生成器 CGM 的引用，所以需要将类型从 AST 表示转换为 LLVM 类型。当然，需要一个名为 Dbuilder 的 llvm::DIBuilder 类的实例，还需要一个指向编译单元实例的指针，将其存储在成员 CU 中。

为了避免再次为类型创建调试元数据，必须添加一个映射来缓存此信息，这个成员叫做 TypeCache。最后，需要一种方法来管理范围信息，为此必须基于 llvm::SmallVector<> 类 ScopeStack 创建一个堆栈:

```
CGModule &CGM;
llvm::DIBuilder DBuilder;
llvm::DICompileUnit *CU;
llvm::DenseMap<TypeDeclaration *, llvm::DIType *>
    TypeCache;
llvm::SmallVector<llvm::DIScope *, 4> ScopeStack;
```

CGDebugInfo 类的以下方法利用了这些成员:

1. 首先，需要在构造函数中创建编译单元，还在这里创建了包含编译单元的文件。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:

```
CGDebugInfo::CGDebugInfo(CGModule &CGM)
        : CGM(CGM), DBuilder(*CGM.getModule()) {
    llvm::SmallString<128> Path(
        CGM.getASTCtx().getFilename());
    llvm::sys::fs::make_absolute(Path);

    llvm::DIFile *File = DBuilder.createFile(
        llvm::sys::path::filename(Path),
        llvm::sys::path::parent_path(Path));

    bool IsOptimized = false;
    llvm::StringRef CUFlags;
    unsigned ObjCRunTimeVersion = 0;
    llvm::StringRef SplitName;
    llvm::DICompileUnit::DebugEmissionKind EmissionKind =
        llvm::DICompileUnit::DebugEmissionKind::FullDebug;
    CU = DBuilder.createCompileUnit(
        llvm::dwarf::DW_LANG_Modula2, File, "tinylang",
        IsOptimized, CUFlags, ObjCRunTimeVersion,
        SplitName, EmissionKind);
}
```

2. 通常，我们需要提供行号。行号可以从源管理器位置派生，这在大多数 AST 节点中都是可用的。源代码管理器可以将其转换为行号:

```
    CGDebugInfo::CGDebugInfo(CGModule &CGM)
unsigned CGDebugInfo::getLineNumber(SMLoc Loc) {
    return CGM.getASTCtx().getSourceMgr().FindLineNumber(
        Loc);
}
```

3. 关于作用域的信息保存在堆栈中，需要打开和关闭作用域以及检索当前作用域的方法。编译单元是全局作用域，则会自动添加:

```
llvm::DIScope *CGDebugInfo::getScope() {
    if (ScopeStack.empty())
    openScope(CU->getFile());
    return ScopeStack.back();
}

void CGDebugInfo::openScope(llvm::DIScope *Scope) {
    ScopeStack.push_back(Scope);
}

void CGDebugInfo::closeScope() {
    ScopeStack.pop_back();
}
```

4. 接下来，必须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 encoding 参数的使用，将 INTEGER 类型声明为有符号类型，并将 BOOLEAN 类型编码为布尔值:

```
llvm::DIType *
CGDebugInfo::getPervasiveType(TypeDeclaration *Ty) {
    if (Ty->getName() == "INTEGER") {
        return DBuilder.createBasicType(
            Ty->getName(), 64, llvm::dwarf::DW_ATE_signed);
    }
    if (Ty->getName() == "BOOLEAN") {
        return DBuilder.createBasicType(
            Ty->getName(), 1, llvm::dwarf::DW_ATE_boolean);
    }
    llvm::report_fatal_error(
        "Unsupported pervasive type");
}
```

5. 若只是重命名类型名，必须将其映射到类型定义，需要利用作用域和行号信息:

```
llvm::DIType *
CGDebugInfo::getAliasType(AliasTypeDeclaration *Ty) {
    return DBuilder.createTypedef(
        getType(Ty->getType()), Ty->getName(),
        CU->getFile(), getLineNumber(Ty->getLocation()),
        getScope());
}
```

6. 为数组创建调试信息需要指定数组大小和对齐方式，可以从 DataLayout 类中检索这些数据，还需要指定数组的索引范围:

```
llvm::DIType *
CGDebugInfo::getArrayType(ArrayTypeDeclaration *Ty) {
    auto *ATy =
        llvm::cast<llvm::ArrayType>(CGM.convertType(Ty));
```

```
    const llvm::DataLayout &DL =
        CGM.getModule()->getDataLayout();

    Expr *Nums = Ty->getNums();
    uint64_t NumElements =
        llvm::cast<IntegerLiteral>(Nums)
            ->getValue()
            .getZExtValue();
    llvm::SmallVector<llvm::Metadata *, 4> Subscripts;
    Subscripts.push_back(
        DBuilder.getOrCreateSubrange(0, NumElements));
    return DBuilder.createArrayType(
        DL.getTypeSizeInBits(ATy) * 8,
        1 << Log2(DL.getABITypeAlign(ATy)),
        getType(Ty->getType()),
        DBuilder.getOrCreateArray(Subscripts));
}
```

7. 使用所有这些单一方法，创建一个中心方法来为类型创建元数据，这些元数据还负责缓存数据:

```
llvm::DIType *
CGDebugInfo::getType(TypeDeclaration *Ty) {
    if (llvm::DIType *T = TypeCache[Ty])
        return T;
    if (llvm::isa<PervasiveTypeDeclaration>(Ty))
        return TypeCache[Ty] = getPervasiveType(Ty);
    else if (auto *AliasTy =
        llvm::dyn_cast<AliasTypeDeclaration>(Ty))
            return TypeCache[Ty] = getAliasType(AliasTy);
    else if (auto *ArrayTy =
        llvm::dyn_cast<ArrayTypeDeclaration>(Ty))
            return TypeCache[Ty] = getArrayType(ArrayTy);
    else if (auto *RecordTy =
        llvm ::dyn_cast<RecordTypeDeclaration>(Ty))
        r   eturn TypeCache[Ty] = getRecordType(RecordTy);
    llvm::report_fatal_error("Unsupported type");
    return nullptr;
}
```

8. 还需要添加一个方法来为全局变量生成元数据:

```
void CGDebugInfo::emitGlobalVariable(
VariableDeclaration *Decl,
llvm::GlobalVariable *V) {
    llvm::DIGlobalVariableExpression *GV =
        DBuilder.createGlobalVariableExpression(
            getScope(), Decl->getName(), V->getName(),
            CU->getFile(),
            getLineNumber(Decl->getLocation()),
```

```
            getType(Decl->getType()), false);
    V->addDebugInfo(GV);
}
```

9. 要为过程发送调试信息，需要为过程类型创建元数据。为此，需要一个参数类型的列表，第一个条目的返回类型。若过程没有返回类型，则必须使用未指定类型; 这称为 void，与 C 语言类似。若形参是引用，则需要添加引用类型; 否则，必须将类型添加到列表中:

```
llvm::DISubroutineType *
CGDebugInfo::getType(ProcedureDeclaration *P) {
    llvm::SmallVector<llvm::Metadata *, 4> Types;
    const llvm::DataLayout &DL =
        CGM.getModule()->getDataLayout();
    // Return type at index 0
    if (P->getRetType())
        Types.push_back(getType(P->getRetType()));
    else
        Types.push_back(
            DBuilder.createUnspecifiedType("void"));
    for (const auto *FP : P->getFormalParams()) {
        llvm::DIType *PT = getType(FP->getType());
        if (FP->isVar()) {
            llvm::Type *PTy = CGM.convertType(FP->getType());
            PT = DBuilder.createReferenceType(
                llvm::dwarf::DW_TAG_reference_type, PT,
                DL.getTypeSizeInBits(PTy) * 8,
                1 << Log2(DL.getABITypeAlign(PTy)));
        }
        Types.push_back(PT);
    }
    return DBuilder.createSubroutineType(
        DBuilder.getOrCreateTypeArray(Types));
}
```

10. 对于过程本身，现在可以使用在前一步中创建的过程类型创建调试信息。过程也会打开一个新的作用域，所以必须将过程压入作用域堆栈，还要将 LLVM 函数对象与新的调试信息关联起来:

```
void CGDebugInfo::emitProcedure(
        ProcedureDeclaration *Decl, llvm::Function *Fn) {
    llvm::DISubroutineType *SubT = getType(Decl);
    llvm::DISubprogram *Sub = DBuilder.createFunction(
        getScope(), Decl->getName(), Fn->getName(),
        CU->getFile(), getLineNumber(Decl->getLocation()),
        SubT, getLineNumber(Decl->getLocation()),
        llvm::DINode::FlagPrototyped,
        llvm::DISubprogram::SPFlagDefinition);
    openScope(Sub);
    Fn->setSubprogram(Sub);
```

```
}
```

11. 当到达过程的末尾时，必须通知构建器完成该过程的调试信息的构造，还需要从作用域堆栈中删除这个过程:

```
void CGDebugInfo::emitProcedureEnd(
ProcedureDeclaration *Decl, llvm::Function *Fn) {
    if (Fn && Fn->getSubprogram())
        DBuilder.finalizeSubprogram(Fn->getSubprogram());
    closeScope();
}
```

12. 最后，当添加完调试信息后，需要在构建器上实现 finalize() 方法，再验证生成的调试信息。这是开发过程中的一个重要步骤，可以帮助找到错生成的元数据:

```
void CGDebugInfo::finalize() { DBuilder.finalize(); }
```

调试信息应该只在用户请求时生成，所以需要一个新的命令行开关。我们将把它添加到 CGModule 类的文件中，也将在这个类中使用它:

```
static llvm::cl::opt<bool>
    Debug("g", llvm::cl::desc("Generate debug information"),
        llvm::cl::init(false));
```

-g 选项可以与 tinylang 编译器一起使用，以生成调试元数据。

此外，CGModule 类持有 std::unique_ptr<CGDebugInfo> 类的实例。指针在用于设置命令行开关的构造函数中初始化:

```
    if (Debug)
        DebugInfo.reset(new CGDebugInfo(*this));
```

在 CGModule.h 中定义的 getter 方法中，我们只返回指针:

```
CGDebugInfo *getDbgInfo() {
    return DebugInfo.get();
}
```

生成调试元数据的常用模式是检索指针，并检查它是否有效。例如，在创建一个全局变量之后，可以像这样添加调试信息:

```
VariableDeclaration *Var = ...;
llvm::GlobalVariable *V = ...;
if (CGDebugInfo *Dbg = getDbgInfo())
    Dbg->emitGlobalVariable(Var, V);
```

为了添加行号信息，需要在 CGDebugInfo 类中使用一个名为 getDebugLoc() 的转换方法，将 AST 中的位置信息转换为调试元数据:

```
llvm::DebugLoc CGDebugInfo::getDebugLoc(SMLoc Loc) {
    std::pair<unsigned, unsigned> LineAndCol =
        CGM.getASTCtx().getSourceMgr().getLineAndColumn(Loc);
    llvm::DILocation *DILoc = llvm::DILocation::get(
        CGM.getLLVMCtx(), LineAndCol.first, LineAndCol.second,
        getScope());
    return llvm::DebugLoc(DILoc);
}
```

此外，可以调用 CGModule 类中的实用函数来将行号信息添加到指令中:

```
void CGModule::applyLocation(llvm::Instruction *Inst,
                             llvm::SMLoc Loc) {
    if (CGDebugInfo *Dbg = getDbgInfo())
        Inst->setDebugLoc(Dbg->getDebugLoc(Loc));
}
```

通过这种方式，可以添加编译器的调试信息。

6.4. 总结

本章中，了解了如何在 LLVM 中抛出和捕获异常，以及如何生成 IR 来利用此功能。为了增强 IR 的范围，了解了如何将各种元数据添加到指令。用于基于类型的别名分析的元数据为 LLVM 优化器提供信息，并帮助进行某些优化以生成更好的机器代码。用户总是喜欢使用源代码级调试器，并且通过向 IR 代码中添加调试信息，可以实现编译器的这一重要特性。

优化 IR 代码是 LLVM 的核心任务。下一章中，我们将学习通道管理器是如何工作的，以及如何影响通道管理器所管理的优化流水线。

第 7 章 优化 IR

LLVM 使用一系列的通道来优化 IR，一个通道在 IR 的一个单元上运行，例如一个函数或一个模块。操作可以是转换，以定义的方式更改 IR，也可以是分析，其收集诸如依赖关系之类的信息。这一系列的通道称为通道流水线，通道管理器在编译器生成的 IR 上执行通道流水线，所以需要了解通道管理器的作用以及如何构造一个通道流水线。编程语言的语义可能需要开发新的通道，我们必须将这些通道添加到流水线中。

本章中，将了解以下内容:

- 如何利用 LLVM 通道管理器在 LLVM 内实现通道
- 如何在 LLVM 项目中实现一个 instrumentation 通道，以及一个单独的插件
- 使用 LLVM 工具的 ppprofiler 通道时，如何使用 opt 和 clang 的通道插件
- 向编译器中添加优化流水时，将使用基于新通道管理器的优化流水扩展 tinylang 编译器

本章结束时，将了解如何开发新的通道，以及如何将其添加到通道流水线中，还可以在编译器中设置通道流水线。

7.1. 环境要求

本章使用的代码在这里https://github.com/PacktPublishing/Learn-LLVM-17/tree/main/Chapter07。

7.2. LLVM 通道管理器

LLVM 核心库优化编译器创建的 IR 并将其转换为目标代码。这项艰巨的任务需要分解成一个个独立的步骤，称为“通道”(passes)。这些通道需要以正确的顺序执行，这是通道管理器的目标。

为什么不硬编码通道的顺序呢? 编译器的用户通常希望编译器提供不同级别的优化。开发人员更喜欢快速的编译速度，而不是在开发期间进行优化。最终的应用程序应该运行得尽可能快，并且编译器应该能够执行复杂的优化，并且可以接受较长的编译时间。不同级别的优化需要执行不同数量的优化，所以编译器作者可能希望提供自己的通道，以利用自己对语言的了解。例如，可能希望用内联 IR 甚至是预先计算的结果，来替换众所周知的库函数。对于 C 语言，这样的通道是 LLVM 库的一部分，但对于其他语言，需要自行提供。了解了通道后，可能需要重新定制或添加一些通道。例如，若知道通行证的操作使一些 IR 代码无法访问，则希望在通道之后运行固定代码删除通道，通道管理器可以协助完成这些需求。

通道通常根据其工作的范围进行分类:

- 模块传递将整个模块作为输入。这样的传递在给定的模块上执行其工作，并可用于该模块内的过程内操作。
- 调用图通道对调用图的强连接组件 (scc) 进行操作，自下而上的顺序遍历组件。
- 函数通道将单个函数作为输入，并仅在该函数上执行其工作。
- 循环通道作用于函数内部的循环。

除了 IR 代码之外，通道还可能需要、更新或使某些分析结果无效。执行了许多不同的分析，例如，别名分析或支配树的构造。若一个通道需要这样的分析，则可以向分析管理器请求。若已经计算了信息，将返回缓存的结果；否则，将计算该信息。若一个通道更改了 IR 代码，则需要宣布保留哪些分析结果，以便可以在必要时使缓存的分析信息无效。

在底层，通道管理器有以下作用:

- 分析结果在各通道之间共享，这需要跟踪哪个通道，需要哪个分析，以及每个分析的状态。目标是避免不必要的分析预计算，并尽快释放分析结果占用的内存。
- 这些通道以流水线方式执行。例如，若几个函数通道应该依次执行，通道管理器将在第一个函数上运行这些函数中的每个函数，再将运行第二个函数通道的所有函数，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一组数据 (一个 IR 函数) 执行转换，然后切换到下一个有限的数据集。

让我们实现一个新的 IR 转换通道，并探索如何将其添加到优化流水线中。

7.3. 实现一个新的通道

一个通道可以在 LLVM IR 上执行任意复杂度的转换。为了说明添加新通道的机制，我们添加一个执行简单检测的通道。

为了研究程序的性能，了解函数被调用的频率和运行的时间是很有趣的。收集这些数据的一种方法是在每个函数中插入计数器，这个过程称为插装。我们将编写一个简单的插装通道，在每个函数的入口和每个退出点插入一个特殊的函数调用，这些函数收集计时信息并将其写入文件。因此，可以创建一个非常基本的分析器，将其命名为“穷人”(poor people) 分析器，或者简而言之——ppprofiler。我们将开发新通道，以便其可以作为一个独立的插件使用，或者作为一个插件添加到 LLVM 源代码树中。最后，我们将了解如何将 LLVM 自带的通道集成到框架中。

7.3.1. 将 ppprofiler 作为插件进行开发

本节中，将了解如何在 LLVM 树中创建一个新通道作为插件。新通道的目标是在函数的入口处插入对__ppp_enter() 函数的调用，并在每个返回指令之前插入对__ppp_exit() 函数的调用。只有当前函数的名称作为参数传递，这些函数的实现可以计算调用的次数，并测量所使用的时间。我们将在本章的最后实现这个运行时支持，并将研究如何开发通道。

我们将源代码存储在 ppprofile.cpp 文件中。遵循以下步骤:

1. 首先，包含一些文件:

```
#include "llvm/ADT/Statistic.h"
#include "llvm/IR/Function.h"
#include "llvm/IR/PassManager.h"
#include "llvm/Passes/PassBuilder.h"
#include "llvm/Passes/PassPlugin.h"
#include "llvm/Support/Debug.h"
```

2. 为了缩短源代码，告诉编译器正在使用 llvm 命名空间:

```
using namespace llvm;
```

3. LLVM 的内置调试基础结构要求定义一个调试类型，它是一个字符串。这个字符串随后显示在打印的统计信息中:

```
#define DEBUG_TYPE "ppprofiler"
```

4. 接下来，将使用 ALWAYS_ENABLED_STATISTIC 宏定义一个计数器变量。第一个参数是计数器变量的名称，而第二个参数是将在统计中打印的文本:

```
ALWAYS_ENABLED_STATISTIC(
    NumOfFunc, "Number of instrumented functions.");
```

> **Note**
>
> 可以使用两个宏来定义计数器变量。若使用 STATISTIC 宏，则只有在启用断言的情况下，或者在 CMake 命令行中将 LLVM_FORCE_ENABLE_STATS 设置为 ON 时，才会在调试构建中收集统计值。若使用 ALWAYS_ENABLED_STATISTIC 宏，则总是收集统计值。但使用-stats 命令行选项输出统计信息只适用于前两种方法。若需要，可以通过调用 llvm::PrintStatistics(llvm::raw_ostream) 函数打印收集的统计信息。

5. 接下来，必须在匿名命名空间中声明通道类，这个类继承自 PassInfoMixin 模板。这个模板只添加了一些样板代码，比如 name() 方法，不用于确定传递的类型，run() 方法在执行传递时由 LLVM 调用，还需要一个名为 instrument() 的辅助方法:

```
namespace {
class PPProfilerIRPass
: public llvm::PassInfoMixin<PPProfilerIRPass> {
public:
    llvm::PreservedAnalyses
    run(llvm::Module &M, llvm::ModuleAnalysisManager &AM);

private:
    void instrument(llvm::Function &F,
                    llvm::Function *EnterFn,
                    llvm::Function *ExitFn);
};
}
```

6. 现在，来定义如何插装函数。除了要检测的函数外，还要通道要调用的函数:

```
void PPProfilerIRPass::instrument(llvm::Function &F,
                                  Function *EnterFn,
                                  Function *ExitFn) {
```

7. 在函数内部，更新统计计数器:

```
    ++NumOfFunc;
```

8. 为了方便地插入 IR 代码，需要 IRBuilder 类的一个实例。我们把它设置为第一个基本块，这是函数的入口块:

```
    IRBuilder<> Builder(&*F.getEntryBlock().begin());
```

9. 现在有了构建器，可以插入一个全局常量来保存检测函数的名称:

```
    GlobalVariable *FnName =
        Builder.CreateGlobalString(F.getName());
```

10. 接下来，将插入对__ppp_enter() 函数的调用，并将名称作为参数传递:

```
    Builder.CreateCall(EnterFn->getFunctionType(), EnterFn,
                       {FnName});
```

11. 要调用__ppp_exit() 函数，必须定位所有返回指令。SetInsertionPoint() 函数设置的插入点在作为参数传递的指令之前，所以可以在该点插入调用:

```
    for (BasicBlock &BB : F) {
        for (Instruction &Inst : BB) {
            if (Inst.getOpcode() == Instruction::Ret) {
                Builder.SetInsertPoint(&Inst);
                Builder.CreateCall(ExitFn->getFunctionType(),
                                   ExitFn, {FnName});
            }
        }
    }
}
```

12. 接下来，将实现 run() 方法。LLVM 通道所工作的模块和一个分析管理器，若需要，可以从中请求分析结果:

```
PreservedAnalyses
PPProfilerIRPass::run(Module &M,
                      ModuleAnalysisManager &AM) {
```

13. 这里有一个小麻烦: 若在实现的运行时模块检测到__ppp_enter() 和__ppp_exit() 函数，则创建了一个无限递归，就会遇到麻烦。为了避免这种情况，若定义了其中一个函数，必须什么都不做:

```
    if (M.getFunction("__ppp_enter") ||
        M.getFunction("__ppp_exit")) {
            return PreservedAnalyses::all();
    }
```

14. 现在，我们准备声明函数。首先，创建函数类型，然后是函数体:

```
    Type *VoidTy = Type::getVoidTy(M.getContext());
    PointerType *PtrTy =
        PointerType::getUnqual(M.getContext());
    FunctionType *EnterExitFty =
        FunctionType::get(VoidTy, {PtrTy}, false);
```

```
    Function *EnterFn = Function::Create(
        EnterExitFty, GlobalValue::ExternalLinkage,
        "__ppp_enter", M);
    Function *ExitFn = Function::Create(
        EnterExitFty, GlobalValue::ExternalLinkage,
        "__ppp_exit", M);
```

15. 现在需要做的就是循环遍历模块的所有函数，并通过调用 instrument() 方法检测找到的函数。需要忽略函数声明，它们只是原型。也可能存在没有名称的函数，这与我们的方法不兼容，就会过滤掉这些函数:

```
    for (auto &F : M.functions()) {
        if (!F.isDeclaration() && F.hasName())
            instrument(F, EnterFn, ExitFn);
    }
```

16. 最后，必须声明，我们没有保留任何分析。这很可能过于悲观，但这样做是出于安全考虑:

```
    return PreservedAnalyses::none();
}
```

新通道的功能现在已经实现了。为了能够使用我们的通道，需要用 PassBuilder 对象注册。这可以通过两种方式实现: 静态或动态。若插件静态链接，则需要提供一个名为 get<PluginName>PluginInfo() 的函数。要使用动态链接，需要提供 llvmGetPassPluginInfo() 函数。这两种情况下，都会返回一个 PassPluginLibraryInfo 结构体的实例，提供了关于插件的一些基本信息。最重要的是，结构体包含一个指向注册通道的函数指针。现在，让我们把它添加到源文件中。

17. RegisterCB() 函数中，注册了一个 Lambda 函数，该函数将在解析通道流水线字符串时调用。若通道的名称是 ppprofiler，将通道添加到模块通道管理器中。这些回调将在下一节中展开:

```
void RegisterCB(PassBuilder &PB) {
    PB.registerPipelineParsingCallback(
        [](StringRef Name, ModulePassManager &MPM,
           ArrayRef<PassBuilder::PipelineElement>) {
            if (Name == "ppprofiler") {
                MPM.addPass(PPProfilerIRPass());
                return true;
            }
            return false;
        });
}
```

18. getPPProfilerPluginInfo() 函数在静态链接插件时调用，会返回一些关于插件的基本信息:

```
llvm::PassPluginLibraryInfo getPPProfilerPluginInfo() {
    return {LLVM_PLUGIN_API_VERSION, "PPProfiler", "v0.1",
        RegisterCB};
}
```

19. 若插件动态链接，则在加载插件时调用 llvmGetPassPluginInfo() 函数。但当将此代码静态地链接到工具中时，可能会出现链接器错误，因为该函数可能在多个源文件中定义。解决方案是使用宏来保护函数:

```
#ifndef LLVM_PPPROFILER_LINK_INTO_TOOLS
extern "C" LLVM_ATTRIBUTE_WEAK ::llvm::PassPluginLibraryInfo
llvmGetPassPluginInfo() {
    return getPPProfilerPluginInfo();
}
#endif
```

这样，我们就实现了通道插件。在了解如何使用新插件之前，让我们检查一下，若想要将通道插件添加到 LLVM 源代码树中，需要更改哪些内容。

7.3.2. 将通道添加到 LLVM 源代码树中

例如，若计划将新通道与预编译的 clang 一起使用，则将其实现为插件是有用的。另一方面，若编写自己的编译器，就有很好的理由将新通道直接添加到 LLVM 源代码树中。有两种不同的方式可以做到这一点——作为一个插件和作为一个集成通道。插件方法的修改会更少一些。

利用 LLVM 源代码树中的插件机制

LLVM IR 上执行转换的传递源位于 llvm-project/llvm/lib/Transforms 目录中。这个目录中，创建一个名为 PPProfiler 的新目录，并将源文件 PPProfiler.cpp 复制到其中，不需要对源代码进行任何更改!

为了将新插件集成到构建系统中，创建一个名为 CMakeLists.txt 的文件，内容如下:

```
add_llvm_pass_plugin(PPProfiler PPProfiler.cpp)
```

最后，在父目录下的 CMakeLists.txt 文件中，需要通过添加以下行来包含新的源目录:

```
add_subdirectory(PPProfiler)
```

现在，已经准备好构建添加了 PPProfiler 的 LLVM。进入 LLVM 的构建目录，手动运行 Ninja:

```
$ ninja install
```

CMake 将检测到构建描述中的变化，并重新运行配置步骤。这里，会看到一行输出:

```
-- Registering PPProfiler as a pass plugin (static build: OFF)
```

这说明该插件已检测到并已构建为动态库。安装完成后，在 <install directory>/lib 目录下，将找到动态库 PPProfiler.so。

目前为止，与前一节的通道插件的唯一区别是，动态库是作为 LLVM 的一部分安装的，也可以静态地将新插件链接到 LLVM 工具。要做到这一点，需要重新运行 CMake 配置并在命令行中添

加-DLLVM_PPPROFILER_LINK_INTO_TOOLS=ON 选项。从 CMake 中查找以下信息以确认已更改的构建选项:

```
-- Registering PPProfiler as a pass plugin (static build: ON)
```

重新编译安装 LLVM 后，发生如下变化:

- 该插件编译到静态库 libPPProfiler.a 中，并且该库安装在 <install directory>/lib 目录下。
- LLVM 工具 (如 opt) 会链接到该库。
- 该插件注册为扩展，可以检查 <install directory>/include/llvm/Support/Extension.def 文件现在包含以下行:

```
HANDLE_EXTENSION(PPProfiler)
```

此外，所有支持此扩展机制的工具都将获得新的通道。创建优化流水线一节中，将了解如何在编译器中执行此操作。

因为新的源文件位于单独的目录中，所以这种方法工作得很好，并且只修改一个现有文件。若试图保持修改后的 LLVM 源树与主存储库同步，这将最小化合并冲突。

某些情况下，将新通道添加为插件并不是最好的方法，LLVM 提供的通道使用不同的方式进行注册。若开发了一个新通道，并将其添加到 LLVM，并且 LLVM 社区接受了您的贡献，可以使用注册机制。

将通道集成到注册表中

为了将新通道完全集成到 LLVM 中，需要对插件的源代码进行稍微不同的结构调整。这样做的主要原因是从通道注册表调用通道类的构造函数，这需要将类接口放入头文件中。

与之前一样，必须将新通道放入 LLVM 的 Transforms 组件中。通过创建 lvm-project/llvm/include/llvm/Transforms/PPProfiler/PPProfiler.h 头文件开始实现。该文件的内容是类定义，将其放入 LLVM 的命名空间中。无需其他更改:

```
#ifndef LLVM_TRANSFORMS_PPPROFILER_PPPROFILER_H
#define LLVM_TRANSFORMS_PPPROFILER_PPPROFILER_H

#include "llvm/IR/PassManager.h"

namespace llvm {
class PPProfilerIRPass
    : public llvm::PassInfoMixin<PPProfilerIRPass> {
public:
    llvm::PreservedAnalyses
    run(llvm::Module &M, llvm::ModuleAnalysisManager &AM);
private:
    void instrument(llvm::Function &F,
                    llvm::Function *EnterFn,
                    llvm::Function *ExitFn);
};
```

```
} // namespace llvm
#endif
```

接下来，将通道插件的源文件 PPProfiler.cpp 复制到新目录 llvmproject/llvm/lib/Transforms/PPProfiler 中。该文件需要按照如下方式更新:

1. 由于类定义现在位于头文件中，因此必须从该文件中删除类定义。在顶部，添加 #include:

```
#include "llvm/Transforms/PPProfiler/PPProfiler.h"
```

2. 必须删除 llvmGetPassPluginInfo() 函数，因为通道没有内置到动态库中。

和前面一样，还需要为构建提供一个 CMakeLists.txt 文件，必须将新通道声明为一个新组件:

```
add_llvm_component_library(LLVMPPProfiler
    PPProfiler.cpp

    LINK_COMPONENTS
    Core
    Support
)
```

之后，与前一节一样，需要通过向父目录下的 CMakeLists.txt 添加以下行来包含新的源目录:

```
add_subdirectory(PPProfiler)
```

LLVM 内部，可用的通道保存在 llvm/lib/Passes/PassRegistry.def 数据库文件中，需要更新这个文件。新通道是一个模块通道，因此需要在文件中搜索定义模块通道部分，例如，搜索 MODULE_PASS 宏。在这个部分中，添加以下行:

```
MODULE_PASS("ppprofiler", PPProfilerIRPass())
```

该数据库文件在 llvm/lib/Passes/PassBuilder.cpp 类中使用。这个文件需要包含新的头文件:

```
#include "llvm/Transforms/PPProfiler/PPProfiler.h"
```

这些都是基于新通道的插件版本所必需的源码更改。

由于创建了一个新的 LLVM 组件，还需要在 llvm/lib/Passes/CMakeLists.txt 文件中添加链接依赖项。在 LINK_COMPONENTS 关键字下，需要添加一行新组件的名称:

```
PPProfiler
```

现在，可以开始构建和安装 LLVM 了，所有 LLVM 工具都可以使用新的通道 ppprofiler。它已经编译到 libLLVMPPProfiler.a 库中，并且可以作为 PPProfiler 组件在构建系统中使用。

我们已经讨论了如何创建一个新的通道。在下一节中，将研究如何使用 ppprofiler 通道。

7.4. 使用 ppprofiler

回想一下我们在 ppprofiler 通道开发为插件一节中，从 LLVM 源码树中作为插件开发的 ppprofiler 通道。这里，因为 LLVM 工具可以加载插件，将学习如何将此通道与 LLVM 工具 (如 opt 和 clang) 一起使用。

我们先看一下 opt。

opt 中运行通道插件

要使用这个新插件，需要一个包含 LLVM IR 的文件。要做到这一点，最简单的方法是翻译一个 C 程序，例如一个“Hello World”风格的程序:

```c
#include <stdio.h>

int main(int argc, char *argv[]) {
    puts("Hello");
    return 0;
}
```

使用 clang 编译 hello.c 文件:

```
$ clang -S -emit-llvm -O1 hello.c
```

会得到一个非常简单的 IR 文件，hello.ll。其中包含以下代码:

```
$ cat hello.ll
@.str = private unnamed_addr constant [6 x i8] c"Hello\00",
        align 1

define dso_local i32 @main(
            i32 noundef %0, ptr nocapture noundef readnone %1) {
    %3 = tail call i32 @puts(
                    ptr noundef nonnull dereferenceable(1) @.str)
    ret i32 0
}
```

这就足够测试通道了。

要运行该通道，必须提供几个参数。首先，需要通过--load-pass-plugin 选项告诉 opt 加载动态库。要运行单个通道，必须指定--passes 选项。使用 hello.ll 文件作为输入，可以运行如下命令:

```
$ opt --load-pass-plugin=./PPProfile.so \
      --passes="ppprofiler" --stats hello.ll -o hello_inst.bc
```

若启用了统计信息生成功能，将看到如下输出:

```
===-------------------------------------------------------------===
... Statistics Collected ...
===-------------------------------------------------------------===

1 ppprofiler - Number of instrumented functions.
```

否则，将提示未启用统计:

```
Statistics are disabled. Build with asserts or with
-DLLVM_FORCE_ENABLE_STATS
```

文件 hello_inst.bc 是结果，可以使用 llvm-dis 工具将该文件转换为可读的 IR。正如预期的那样，将会看到对__ppp_enter() 和__ppp_exit() 函数的调用，以及一个新的函数名常量:

```
$ llvm-dis hello_inst.bc -o -
@.str = private unnamed_addr constant [6 x i8] c"Hello\00",
        align 1
@0 = private unnamed_addr constant [5 x i8] c"main\00",
     align 1
define dso_local i32 @main(i32 noundef %0,
                           ptr nocapture noundef readnone %1) {
    call void @__ppp_enter(ptr @0)
    %3 = tail call i32 @puts(
                    ptr noundef nonnull dereferenceable(1) @.str)
    call void @__ppp_exit(ptr @0)
    ret i32 0
}
```

这看起来已经很不错了! 若能将此 IR 转换为可执行文件并运行，，需要为调用函数提供实现。

通常，运行时对特性的支持比将该特性添加到编译器本身要复杂得多。当调用__ppp_enter() 和__ppp_exit() 时，可以将其视为事件。为了以后分析数据，有必要保存事件。可获得的基本数据是该类型的事件、函数的名称及其地址以及时间戳。没有点技巧，处理起来并不是那么容易。

创建一个名为 runtime.c 的文件，内容如下:

1. 需要文件 I/O、标准函数和时间支持。这是由以下头文件提供的:

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

2. 对于该文件，需要一个文件描述符。当程序结束时，该文件描述符应该正确关闭:

```c
static FILE *FileFD = NULL;

static void cleanup() {
    if (FileFD == NULL) {
        fclose(FileFD);
```

```
        FileFD = NULL;
    }
}
```

3. 为了简化运行时，只对输出使用固定的名称。若文件未打开，则打开文件并注册清理功能:

```
static void init() {
    if (FileFD == NULL) {
        FileFD = fopen("ppprofile.csv", "w");
        atexit(&cleanup);
    }
}
```

4. 可以调用 clock_gettime() 函数来获取时间戳，CLOCK_PROCESS_CPUTIME_ID 参数返回该进程消耗的时间。注意，并非所有系统都支持此参数。若有必要，可以使用其他时钟之一，比如 CLOCK_REALTIME:

```
typedef unsigned long long Time;

static Time get_time() {
    struct timespec ts;
    clock_gettime(CLOCK_PROCESS_CPUTIME_ID, &ts);
    return 1000000000L * ts.tv_sec + ts.tv_nsec;
}
```

5. 现在，很容易定义__ppp_enter() 函数。只需确保文件是打开的，获取时间戳，并写入事件:

```
void __ppp_enter(const char *FnName) {
    init();
    Time T = get_time();
    void *Frame = __builtin_frame_address(1);
    fprintf(FileFD,
            // "enter|name|clock|frame"
            "enter|%s|%llu|%p\n", FnName, T, Frame);
}
```

6. __ppp_exit() 函数仅在事件类型方面有所不同:

```
void __ppp_exit(const char *FnName) {
    init();
    Time T = get_time();
    void *Frame = __builtin_frame_address(1);
    fprintf(FileFD,
    // "exit|name|clock|frame"
    "exit|%s|%llu|%p\n", FnName, T, Frame);
}
```

这是一个非常简单的运行时支持实现。在我们尝试之前，应该对实现做一些注释，因为有几个部分明显存在问题。

首先，因为只有一个文件描述符，对它的访问不受保护，所以该实现线程不安全。尝试在多线程程序中使用此运行时实现，很可能导致输出文件中的数据受到干扰。

此外，省略了检查与 I/O 相关的函数的返回值，这可能导致数据丢失。

最重要的是，事件的时间戳并不精确。调用函数已经增加了开销，但在该函数中执行 I/O 操作会使情况变得更糟。原则上，可以匹配函数的进入和退出事件，并计算函数的运行时。但这个值本身就有缺陷，因为可能包含 I/O 所需的时间，所以不要相信这里记录的时间。

尽管有这些缺陷，这个小的运行时文件允许产生一些输出。将检测文件的位码与包含运行时代码的文件一起编译，并运行生成的可执行文件:

```
$ clang hello_inst.bc runtime.c
$ ./a.out
```

这将在目录中生成一个名为 ppprofile.csv 的新文件，其中包含以下内容:

```
$ cat ppprofile.csv
enter|main|3300868|0x1
exit|main|3760638|0x1
```

Cool -新通道和运行时似乎可以工作了!

指定通道流水线

使用--passes 选项，不仅可以命名单个通道，还可以描述整个流水线。例如，优化级别 2 的默认流水线名为 default<O2>。可以使用--passes="ppprofile,default<O2>" 参数在默认管道之前运行 ppprofile 通道。注意，这种流水线描述中的通道名称，类型必须相同。

现在，让我们在 clang 中使用新通道。

将新通道插入 clang

上一节中，了解了如何使用 opt 运行单个通道。若需要调试通道，对于真正的编译器，步骤不应该那么复杂。

为了获得最佳结果，编译器需要按照一定的顺序运行优化通道。LLVM 通道管理器有默认的执行顺序，这也称为默认传递管道。使用 opt 时，可以使用-passes 选项指定不同的通道流水线。这很灵活，但对用户来说也很复杂。大多数情况下，只想在特定的点添加一个新的通道，例如在优化通道运行之前或在循环优化过程结束时，这些点称为扩展点。PassBuilder 类允许在扩展点注册通道。可以调用 registerPipelineStartEPCallback() 方法，在优化通道前添加一个通道，这正是我们 ppprofiler 通道的位置。优化过程中，函数可能内联，通道将错过这些内联函数。相反，在优化通道之前运行通道，可以保证所有函数都检测到。

要使用这种方法，需要在通道插件中扩展 RegisterCB() 函数。将以下代码添加到函数中:

```
PB.registerPipelineStartEPCallback(
    [](ModulePassManager &PM, OptimizationLevel Level) {
```

```
        PM.addPass(PPProfilerIRPass());
    });
```

每当通道管理器填充默认通道流水线时，都会调用扩展点的所有回调，只需在这里添加新的通道即可。

要将插件加载到 clang 中，可以使用-fpass-plugin 选项。创建 hello.c 文件的可执行文件就很简单:

```
$ clang -fpass-plugin=./PPProfiler.so hello.c runtime.c
```

请运行可执行文件，并验证运行是否创建了 ppprofile .csv 文件。

> **Note**
>
> 因为通过检查特殊函数是否尚未在模块中声明，所以没有检测 runtime.c 文件。

但能扩展到更大的程序吗? 假设为第 5 章构建 tinylang 编译器的二进制文件。需要怎么做呢? 可以在 CMake 命令行中传递编译器和链接器标志，这正是我们所需要的。

C++ 编译器的标志在 CMAKE_CXX_FLAGS 变量中给出，所以在 CMake 命令行中指定以下命令会将新的通道添加到所有编译器运行中:

```
-DCMAKE_CXX_FLAGS="-fpass-plugin=<PluginPath>/PPProfiler.so"
```

请将 <PluginPath> 替换为动态库的绝对路径。

类似地，指定下列语句将添加 runtime.o 文件到每个链接器调用，请将 <RuntimePath> 替换为 runtime.c 编译版本的绝对路径:

```
-DCMAKE_EXE_LINKER_FLAGS="<RuntimePath>/runtime.o"
```

当然，这需要 clang 作为构建编译器。确保 clang 用作构建编译器的最快方法是设置 CC 和 CXX 环境变量:

```
export CC=clang
export CXX=clang++
```

有了这些选项，第 5 章中的 CMake 配置应该像往常一样运行。

构建了 tinylang 可执行文件之后，可以使用示例 Gcd.mod 文件运行。还需要创建 ppprofile.csv 文件，这次有超过 44,000 行! 当然，拥有这样一个数据集会提出一个问题，即是否可以从中获得有用的东西。例如，获得 10 个最常调用的函数的列表，以及调用计数和在函数中花费的时间，将是有用的信息。幸运的是，在 Unix 系统上，有一些工具可以提供帮助。让我们构建一个简短的管道来匹配进入事件和退出事件，对函数进行计数，并显示前 10 个函数。awk Unix 工具可以帮助完成这些步骤中的大部分。

要将进入事件与退出事件匹配，必须将进入事件存储在记录关联映射中。匹配退出事件时，查找存储的进入事件，并写入新记录。发出的行包含进入事件的时间戳、退出事件的时间戳以及两者之间的差异。必须把这个放进 join.awk 文件中:

```
BEGIN { FS = "|"; OFS = "|" }
/enter/ { record[$2] = $0 }
/exit/ { split(record[$2],val,"|")
         print val[2], val[3], $3, $3-val[3], val[4] }
```

为了计算函数调用和执行，使用了两个关联映射，count 和 sum。在 count 中，函数调用时计数，而在 sum 中，增加执行时间。最后，映射转储，可以将其放入 avg.awk 文件中:

```
BEGIN { FS = "|"; count[""] = 0; sum[""] = 0 }
{ count[$1]++; sum[$1] += $4 }
END { for (i in count) {
        if (i != "") {
            print count[i], sum[i], sum[i]/count[i], I }
} }
```

运行这两个脚本之后，可以按降序对结果进行排序，然后可以从文件中取出前 10 行。仍然可以改进函数名，__ppp_enter() 和__ppp_exit()，这时依旧很难阅读。使用 llvm-cxxfilt 工具，可以修改这些名称。demangle.awk 脚本如下所示:

```
{ cmd = "llvm-cxxfilt " $4
    (cmd) | getline name
    close(cmd); $4 = name; print }
```

要获得前 10 个函数调用，可以运行以下命令:

```
$ cat ppprofile.csv | awk -f join.awk | awk -f avg.awk |\
  sort -nr | head -15 | awk -f demangle.awk
```

下面是输出中的一些示例行:

```
446 1545581 3465.43 charinfo::isASCII(char)
409 826261 2020.2 llvm::StringRef::StringRef()
382 899471 2354.64
            tinylang::Token::is(tinylang::tok::TokenKind) const
171 1561532 9131.77 charinfo::isIdentifierHead(char)
```

第一个数字是函数的调用次数，第二个数字是累计执行时间，第三个数字是平均执行时间。如前所述，不要相信时间值，但调用计数应该是准确的。

目前，我们已经实现了一个新的检测通道，或者作为一个插件，或者作为 LLVM 的补充，并在一些实际场景中使用了它。下一节中，将探讨如何在编译器中设置一个优化流水线。

7.5. 向编译器添加优化管道

我们在前面章节中开发的 tinylang 编译器，但没有对 IR 代码进行优化。接下来的小节中，我们将在编译器中添加一个优化流水线来实现这一目标。

7.5.1. 创建优化流水线

PassBuilder 类是设置优化流水线的核心。这个类知道所有已注册的通道，并且可以从文本描述构造一个通道流水线。可以使用这个类根据命令行给出的描述创建通道流水线，或者根据所请求的优化级别使用默认流水线。还支持使用通道插件，例如：在前一节中讨论的 ppprofiler 通道插件，就可以模拟 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 ModulePassManager 类的实例，ModulePassManager 类是通道管理器，保存构造的通道流水线并运行，代码生成通道仍然使用旧的通道管理器，所以必须保留旧的通道管理器。

为了实现它，我们将扩展 tinylang 编译器中的 tools/driver/Driver.cpp 文件:

1. 这里将使用新的类，所以将从添加新的包含文件开始。llvm/Passes/PassBuilder.h 文件定义了 PassBuilder 类，llvm/Passes/PassPlugin.h 文件是插件支持所必需的。最后，llvm/Analysis/targettransformminfo.h 文件提供了一个 IR 级转换，并与目标特定信息连接起来的通道:

```
#include "llvm/Passes/PassBuilder.h"
#include "llvm/Passes/PassPlugin.h"
#include "llvm/Analysis/TargetTransformInfo.h"
```

2. 要使用新通道管理器的某些特性，必须添加三个命令行选项，使用与 opt 工具相同的名称。--passes 选项允许通过管道的文本规范，而--load-pass-plugin 选项允许使用通道插件。若给出了--debug-pass-manager 选项，则通道管理器会输出执行通道的信息:

```
static cl::opt<bool>
    DebugPM("debug-pass-manager", cl::Hidden,
        cl::desc("Print PM debugging information"));
static cl::opt<std::string> PassPipeline(
    "passes",
    cl::desc("A description of the pass pipeline"));
static cl::list<std::string> PassPlugins(
    "load-pass-plugin",
    cl::desc("Load passes from plugin library"));
```

3. 用户以优化级别影响通道流水线的构造。PassBuilder 类支持 6 个不同的优化级别: 无优化、3 个优化速度级别和 2 个减小文件大小级别。我们可以在一个命令行选项中获取所有级别:

```
static cl::opt<signed char> OptLevel(
    cl::desc("Setting the optimization level:"),
    cl::ZeroOrMore,
    cl::values(
        clEnumValN(3, "O", "Equivalent to -O3"),
        clEnumValN(0, "O0", "Optimization level 0"),
        clEnumValN(1, "O1", "Optimization level 1"),
        clEnumValN(2, "O2", "Optimization level 2"),
        clEnumValN(3, "O3", "Optimization level 3"),
        clEnumValN(-1, "Os", "Like -O2 with extra optimizations for size"),
        clEnumValN(-2, "Oz", "Like -Os but reduces code size further")),
        cl::init(0));
```

4. LLVM 的插件机制支持静态链接插件的插件注册表，是在项目配置期间创建的。为了利用这个注册表，必须包含 llvm/Support/Extension.def 数据库文件，以创建返回插件信息的函数原型:

```
#define HANDLE_EXTENSION(Ext) \
    llvm::PassPluginLibraryInfo get##Ext##PluginInfo();
#include "llvm/Support/Extension.def"
```

5. 现在，必须用一个新版本替换现有的 emit() 函数，必须在函数的顶部声明所需的 PassBuilder 实例:

```
bool emit(StringRef Argv0, llvm::Module *M,
          llvm::TargetMachine *TM,
          StringRef InputFilename) {
    PassBuilder PB(TM);
```

6. 为了实现对命令行中给出的传递插件的支持，必须循环遍历用户给出的插件库列表，并尝试加载插件。若此操作失败，将输出一个错误消息; 否则，将注册通道:

```
for (auto &PluginFN : PassPlugins) {
    auto PassPlugin = PassPlugin::Load(PluginFN);
    if (!PassPlugin) {
        WithColor::error(errs(), Argv0)
            << "Failed to load passes from '" << PluginFN
            << "'. Request ignored.\n";
        continue;
    }

    PassPlugin->registerPassBuilderCallbacks(PB);
}
```

7. 来自静态插件注册表的信息，以类似的方式用于向 PassBuilder 实例注册这些插件:

```
#define HANDLE_EXTENSION(Ext) \
    get##Ext##PluginInfo().RegisterPassBuilderCallbacks(PB);
#include "llvm/Support/Extension.def"
```

8. 现在，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:

```
LoopAnalysisManager LAM(DebugPM);
FunctionAnalysisManager FAM(DebugPM);
CGSCCAnalysisManager CGAM(DebugPM);
ModuleAnalysisManager MAM(DebugPM);
```

9. 接下来，必须使用对 PassBuilder 实例上各自的注册方法的调用来填充分析管理器。分析管理器将使用默认分析通道填充，并运行注册回调。还必须确保功能分析管理器使用默认别名分析流水线，并且所有的分析管理器都知道彼此的存在:

```
FAM.registerPass(
    [&] { return PB.buildDefaultAAPipeline(); });
PB.registerModuleAnalyses(MAM);
PB.registerCGSCCAnalyses(CGAM);
```

```
PB.registerFunctionAnalyses(FAM);
PB.registerLoopAnalyses(LAM);
PB.crossRegisterProxies(LAM, FAM, CGAM, MAM);
```

10. MPM 模块通道管理器保存构造的通道流水线，实例用 debug 标志初始化:

```
ModulePassManager MPM(DebugPM);
```

11. 现在，需要实现两种不同的方法，用通道流水线填充模块通道管理器。若用户在命令行上提供了一个通道流水线——使用了--passes 选项——则使用这个作为通道流水线:

```
if (!PassPipeline.empty()) {
    if (auto Err = PB.parsePassPipeline(
    MPM, PassPipeline)) {
        WithColor::error(errs(), Argv0)
        << toString(std::move(Err)) << "\n";
        return false;
    }
}
```

12. 否则，使用所选择的优化级别来确定要构建的通道流水线。默认的通道流水线名称为 default，将优化级别作为参数:

```
else {
    StringRef DefaultPass;
    switch (OptLevel) {
        case 0: DefaultPass = "default<O0>"; break;
        case 1: DefaultPass = "default<O1>"; break;
        case 2: DefaultPass = "default<O2>"; break;
        case 3: DefaultPass = "default<O3>"; break;
        case -1: DefaultPass = "default<Os>"; break;
        case -2: DefaultPass = "default<Oz>"; break;
    }
    if (auto Err = PB.parsePassPipeline(
            MPM, DefaultPass)) {
        WithColor::error(errs(), Argv0)
        << toString(std::move(Err)) << "\n";
        return false;
    }
}
```

13. IR 代码上运行转换的通道流水线就已经设置好了。之后，需要一个打开的文件来写入结果。系统汇编器和 LLVM IR 输出是基于文本的，所以应该为其设置 OF_Text 标志:

```
std::error_code EC;
sys::fs::OpenFlags OpenFlags = sys::fs::OF_None;
CodeGenFileType FileType = codegen::getFileType();
if (FileType == CGFT_AssemblyFile)
    OpenFlags |= sys::fs::OF_Text;
auto Out = std::make_unique<llvm::ToolOutputFile>(
```

```
    outputFilename(InputFilename), EC, OpenFlags);
if (EC) {
    WithColor::error(errs(), Argv0)
        << EC.message() << '\n';
    return false;
}
```

14. 对于代码生成过程，必须使用旧的通道管理器。必须简单地声明 CodeGenPM 实例并添加通道，这使得特定于目标的信息在进行 IR 级别的转换时可用:

```
legacy::PassManager CodeGenPM;
CodeGenPM.add(createTargetTransformInfoWrapperPass(
    TM->getTargetIRAnalysis()));
```

15. 要输出 LLVM IR，必须添加一个将 IR 输出到流中的通道:

```
if (FileType == CGFT_AssemblyFile && EmitLLVM) {
    CodeGenPM.add(createPrintModulePass(Out->os()));
}
```

16. 否则，必须让 TargetMachine 实例添加所需的代码生成通道，由作为参数传递的 FileType 值作为指示:

```
else {
    if (TM->addPassesToEmitFile(CodeGenPM, Out->os(),
                                nullptr, FileType)) {
        WithColor::error()
        << "No support for file type\n";
        return false;
    }
}
```

17. 所有这些准备工作之后，准备执行通道。必须在 IR 模块上运行优化流水线，再运行代码生成通道。在所有这些工作之后，保留输出文件:

```
    MPM.run(*M, MAM);
    CodeGenPM.run(*M);
    Out->keep();
    return true;
}
```

18. 虽然代码很多，但过程很简单。我们也必须更新 tools/driver/CMakeLists.txt 构建文件中的依赖项，除了添加目标组件，还必须添加来自 LLVM 的所有转换和代码生成组件，这些名称大致类似于源代码所在的目录名称。在配置过程中，组件名会转换为链接库名:

```
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. 编译器驱动支持插件，必须声明这个支持:

```
target_link_libraries(tinylang
    PRIVATE tinylangBasic tinylangCodeGen
    tinylangLexer tinylangParser tinylangSema)
```

这些都是对源代码和构建系统的必要补充。

20. 要构建扩展编译器，必须切换到构建目录并键入以下命令:

```
$ ninja
```

对构建系统文件的更改会自动检测，并且在编译和链接更改的源代码前运行 cmake。若需要重新运行配置步骤，请按照第 1 章，编译 tinylang 应用程序部分中的说明进行操作。

我们已经使用了 opt 工具的选项作为蓝图，所以应该尝试运行 tinylang，使用这些选项来加载一个通道插件并运行通道。

当前的实现中，既可以运行默认的传递管道，也可以自己构造一个。后者是非常灵活的，但它几乎是多余的。默认管道在类 C 语言中运行得非常好，但缺少的是一种扩展通道流水线的方法。我们将在下一节中讨论如何实现它。

7.5.2. 扩展通道流水线

上一节中，使用 PassBuilder 类从用户提供的描述或预定义的名称创建了一个通道流水线。现在，让我们看看定制通道流水线的另一种方法: 扩展点。

通道流水线的构造过程中，通道构建器允许添加用户提供的通道，这些点称为扩展点。一些扩展点的特特征，如下所示:

- 流水线起始扩展点，允许在流水线的开头添加通道
- 窥视孔扩展点，允许在每个指令组合器实例通道之后添加通道

还存在其他扩展点。要使用扩展点，必须注册回调。通道流水线的构造过程中，回调在定义的扩展点上运行，并且可以向给定的通道管理器添加通道。

要为流水线的起始扩展点注册回调函数，必须调用 PassBuilder 类的 registerPipelineStartEPCallback() 方法。例如，要在流水线的开头添加 PPProfiler 通道，需要调用 createModuleToFunctionPassAdaptor() 模板函数，将该通道修改为模块通道，然后将其添加到模块通道管理器:

```
PB.registerPipelineStartEPCallback(
    [](ModulePassManager &MPM) {
        MPM.addPass(PPProfilerIRPass());
    });
```

可以在流水线创建之前的任何地方——也就是 parsepaspipeline() 方法调用之前，在通道流水线设置代码中添加这段代码。

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添加管道扩展点的流水线描述。opt 工具也可以这样做，让我们对管道开始扩展点执行此操作。在 tools/driver/Driver.cpp 文件中添加如下代码:

1. 首先，必须为用户创建一个新的命令行来指定管道描述，从 opt 工具中获取选项名:

```
static cl::opt<std::string> PipelineStartEPPipeline(
    "passes-ep-pipeline-start",
    cl::desc("Pipeline start extension point));
```

2. 使用 Lambda 函数作为回调是最方便的方式。为了解析管道描述，必须调用 PassBuilder 实例的 parsepaspipeline() 方法。这些通道会添加到 PM 通道管理器中，并作为 Lambda 函数的参数。若发生错误，只打印错误消息，而不停止应用程序。在调用 crossRegisterProxies() 方法后，添加以下代码:

```
PB.registerPipelineStartEPCallback(
[&PB, Argv0](ModulePassManager &PM) {
    if (auto Err = PB.parsePassPipeline(
            PM, PipelineStartEPPipeline)) {
        WithColor::error(errs(), Argv0)
            << "Could not parse pipeline "
            << PipelineStartEPPipeline.ArgStr << ": "
            << toString(std::move(Err)) << "\n";
    }
});
```

Tip

要允许用户在每个扩展点添加通道，需要为每个扩展点添加上述代码片段。

3. 是时候尝试不同的传递管理器选项了。使用--debugpass-manager 选项，可以了解哪些通道以何种顺序执行。还可以在每个通道执行前后打印 IR，通过--print-before-all 和--print-after-all 选项完成。若创建了自己的通道流水线，就可以在感兴趣的地方插入打印通道。例如，可以尝试--passes=”print,inline,print” 选项。此外，为了识别哪一个通道改变了 IR 代码，可以使用--printchanged 选项，只会在 IR 代码与之前通道的结果相比发生了变化时打印出来，大大减少的输出使得跟踪 IR 修改变得更加容易。

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<O?” >” 流水描述作为 parsePassPipeline() 方法的参数，也可以调用 buildPerModuleDefaultPipeline() 方法，为请求级别构建默认的优化管道——除了级别 0，此优化级别不执行任何优化。

没有通道添加到通道管理器中，若仍然想要运行某个通道，则可以手动将其添加到通道管理器中。这个级别上运行的一个简单通道是 AlwaysInliner 通道，将一个带有 always_inline 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 类的相应成员之后，可以这样实现:

```
PassBuilder::OptimizationLevel Olevel = ···;
if (OLevel == PassBuilder::OptimizationLevel::O0)
    MPM.addPass(AlwaysInlinerPass());
else
    MPM = PB.buildPerModuleDefaultPipeline(OLevel, DebugPM);
```

当然，可以以这种方式向通道管理器添加多个通道。构造通道流水线时，PassBuilder 也使用 addPass() 方法。

运行扩展点的回调

因为没有为优化级别 O0 填充通道流水线，所以注册的扩展点不会调用。使用扩展点来注册应该在 O0 级别运行的通道会有问题，可以使用 runRegisteredEPCallbacks() 来运行注册的扩展点回调，从而产生一个仅由通过扩展点注册通道填充的通道管理器。

通过将优化流水线添加到 tinylang 中，既创建了一个类似于 clang 的优化编译器。LLVM 社区致力于在每个版本中改进优化和优化管道，所以很少不使用默认管道。大多数情况下，添加新通道是为了实现编程语言的某些语义。

7.6. 总结

本章中，了解了如何为 LLVM 创建一个新的通道，使用通道流水线描述和扩展点运行通道。通过构造和执行类似于 clang 的通道流水线扩展了编译器，将 tinylang 变成了一个优化编译器。通道流水线允许在扩展点添加通道，并且了解了如何在这些点注册通道，可以使用开发的通道或现有的通道扩展优化流水线。

下一章中，将解了 TableGen 语言的基础知识，该语言在 LLVM 和 clang 中广泛使用，以减少手动编程。

第三部分 LLVM 的进阶

深入研究 LLVM 的各种底层细节，探索 TableGen 语言，这是 LLVM 的特定于领域的语言，并了解如何在后端使用它。LLVM 还有一个即时 (JIT) 编译器，将探索如何使用，并根据需求进行定制。此外，还将尝试用于识别应用程序中的错误的各种工具和库。了解了这些后，就可以去支持 LLVM 尚未支持的新架构了。

- 第 8 章，TableGen 语言
- 第 9 章，JIT 编译
- 第 10 章，使用 LLVM 工具进行调试

第 8 章 TableGen 语言

LLVM 中的大部分后端都用 TableGen 语言编写，TableGen 是一种特殊的语言，用于生成 C++ 源代码的片段，以避免每个后端都实现类似的代码，并减少源码量，所以了解 TableGen 很重要。

本章中，将了解以下内容:

- 了解 TableGen 语言时，将了解 TableGen 背后的主要思想
- TableGen 语言的实验中，将定义自己的 TableGen 类和记录，并学习 TableGen 语言的语法
- 使用 TableGen 文件生成 C++ 代码中，可以开发自己的 TableGen 后端
- TableGen 的缺点

本章结束时，能够使用现有的 TableGen 类来定义自己的记录，了解如何从头创建 TableGen 类和记录，以及如何开发 TableGen 后端以生成源码。

8.1. 环境要求

本章使用的代码在这里，https://github.com/PacktPublishing/Learn-LLVM-17/tree/main/Chapter08。

8.2. 了解 TableGen 语言

LLVM 有自己的领域特定语言 (DSL)，称为 TableGen，用于为很多用例生成 C++ 代码，减少开发人员手写的代码量。TableGen 语言不是一种编程语言，只用于定义记录。对于名称和值的集合来说，这是一个花哨的词。为了理解为什么使用这种语言，先来看两个例子。

定义 CPU 的一条机器指令需要的典型数据为:

- 指令的助记符
- 位模式
- 操作数的数量和类型
- 可能的限制或副作用

这些数据可以表示为一条记录。例如，一个名为 asmstring 的字段可以保存助记符的值; 比如，“添加”。另外，一个名为 opcode 的字段可以保存指令的二进制表示形式。记录将描述一个指令，每个 LLVM 后端都以这种方式描述指令集。

记录是一个通用的概念，可用来描述各种各样的数据。另一个例子是命令行选项的定义，命令行选项:

- 有名字
- 有可选的参数
- 有帮助文本
- 可能属于一组选项

同样，可以将这些数据视为记录，Clang 将这种方法用于 Clang 驱动程序的命令行选项。

TableGen 语言

LLVM 中，TableGen 语言用于各种任务。后端的大部分是用 TableGen 语言编写的; 例如，寄存器文件的定义，所有带有助记符和二进制编码的指令，调用约定，指令选择的模式，指令调度的调度模型。LLVM 的其他用途包括定义内部函数、定义属性和定义命令行选项。
可以在https://llvm.org/docs/TableGen/ProgRef.html找到开发者参考，在https://llvm.org/docs/TableGen/BackGuide.html找到后端开发者指南。

为了实现这种灵活性，TableGen 语言的解析和语义在库中实现。为了使用记录中生成的 C++ 代码，需要创建一个工具来获取解析后的记录，并生成 C++ 代码。在 LLVM 中，这个工具称为 llvm-tblgen，在 Clang 中，称为 clang-tblgen。这些工具包含项目所需的代码生成器，也可以用来学习 TableGen 语言，这是下一节要做的。

8.3. 实验 TableGen 语言

初学者对 TableGen 语言感到不知所措，开始尝试这门语言后，就会发现没那么难。

8.3.1. 定义记录和类

让我们为指令定义一条简单的记录:

```
def ADD {
    string Mnemonic = "add";
    int Opcode = 0xA0;
}
```

def 关键字表示定义了一条记录，后面跟着记录的名称。记录主体由花括号包围，主体由字段定义组成，类似于 C++ 中的结构体。

可以使用 llvm-tblgen 工具查看生成的记录，将上述源代码保存在 inst.td 文件中，然后运行如下命令:

```
$ llvm-tblgen --print-records inst.td
------------- Classes -----------------
------------- Defs -----------------
def ADD {
    string Mnemonic = "add";
    int Opcode = 160;
}
```

这只表明正确的解析了定义的记录。

定义指令使用单一的记录不是很舒服。现代的 CPU 有数百条指令，有了这么多的记录，很容易在字段名中引入输入错误。若决定重命名一个字段或添加一个新字段，则要更改的记录数量将会非常的大，所以蓝图必不可少。C++ 中，类也有类似的用途；TableGen 中，也被称为类。下面是一个 Inst 类的定义，以及基于该类的两条记录:

```
class Inst<string mnemonic, int opcode> {
    string Mnemonic = mnemonic;
    int Opcode = opcode;
}

def ADD : Inst<"add", 0xA0>;
def SUB : Inst<"sub", 0xB0>;
```

类的语法类似于记录的语法。class 关键字表示定义了一个类，后跟类名。类可以有一个参数列表，Inst 类有两个参数，助记符和操作码，用于初始化记录的字段。这些字段的值在类实例化时给出。ADD 和 SUB 记录显示了类的两个实例化，先使用 llvm-tblgen 来查看记录:

```
$ llvm-tblgen --print-records inst.td
------------- Classes -----------------
class Inst<string Inst:mnemonic = ?, int Inst:opcode = ?> {
    string Mnemonic = Inst:mnemonic;
    int Opcode = Inst:opcode;
}
------------- Defs -----------------
def ADD { // Inst
    string Mnemonic = "add";
    int Opcode = 160;
}
def SUB { // Inst
    string Mnemonic = "sub";
    int Opcode = 176;
}
```

现在，有了一个类定义和两条记录，用于定义记录的类的名称显示为注释。注意，该类的参数具有默认值“?”，这表示 int 未初始化。

调试技巧

要获得更详细的记录转储，可以使用--print-detailed-records 选项。输出包括记录和类定义的行号，以及初始化记录字段的位置。可用来跟踪一个记录字段如何分配了某个值。

ADD 和 SUB 指令有很多共同之处，但加法是可交换运算，而减法不是。这里的一个小小的挑战是，TableGen 只支持有限的数据类型集。已经在示例中使用了 string 和 int，其他可用的数据类型有 bit、bits<n>、list<type> 和 dag。位类型表示单个位; 也就是 0 或 1。若需要固定数量的位，则使用 bits<n> 类型。例如，bits<5> 是一个宽为 5 位的整数类型。基于另一种类型定义列表，可以使用 list<type> 类型。例如，list<int> 是一个整数列表，而 list<Inst> 是示例中 Inst 类的记录列表。dag 类型表示有向无环图 (dag) 节点，对于定义模式和操作很有用，并且在 LLVM 后端中广泛使用。

使用单个比特表示一个标志就足够了，因此可以使用一个比特将指令标记为可交换的。大多数指令都不可交换，所以可以使用默认值:

```
class Inst<string mnemonic, int opcode, bit commutable = 0> {
    string Mnemonic = mnemonic;
    int Opcode = opcode;
    bit Commutable = commutable;
}

def ADD : Inst<"add", 0xA0, 1>;
def SUB : Inst<"sub", 0xB0>;
```

应该运行 llvm-tblgen 来验证记录是否按预期定义。

类不需要有参数，也可以稍后赋值。可以定义所有指令都不可交换:

```
class Inst<string mnemonic, int opcode> {
    string Mnemonic = mnemonic;
    int Opcode = opcode;
    bit Commutable = 0;
}

def SUB : Inst<"sub", 0xB0>;
```

使用 let 语句，可以覆盖该值:

```
let Commutable = 1 in
    def ADD : Inst<"add", 0xA0>;
```

或者，打开记录主体来覆盖该值:

```
def ADD : Inst<"add", 0xA0> {
    let Commutable = 1;
}
```

同样，请使用 llvm-tblgen 来验证在这两种情况下，是否将可交换标志设置为 1。

类和记录可以从多个类继承，并且可以添加新字段或覆盖现有字段的值。可以使用继承来引入一个新的可交换类:

```
class Inst<string mnemonic, int opcode> {
    string Mnemonic = mnemonic;
    int Opcode = opcode;
    bit Commutable = 0;
}

class CommutableInst<string mnemonic, int opcode>
    : Inst<mnemonic, opcode> {
    let Commutable = 1;
}

```

```
def SUB : Inst<"sub", 0xB0>;
def ADD : CommutableInst<"add", 0xA0>;
```

生成的记录总相同，但是该语言允许以不同的方式定义记录。注意，在后一个示例中，可交换标志可能是多余的: 代码生成器可以查询它所基于的类的记录，若该列表包含可交换类，则可以在内部设置该标志。

8.3.2. 使用多个类一次创建多个记录

另一个常用语句是 multiclass，multiclass 允许一次定义多个记录，让我们扩展这个示例。

add 指令的定义非常简单，一个 CPU 通常有几个 add 指令。一种常见的变体是一条指令有两个寄存器操作数，而另一条指令有一个寄存器操作数和一个直接操作数，这是一个小数字。假设对于具有直接操作数的指令，指令集的设计者决定用 i 作为后缀来标记，所以以 add 和 addi 指令结束。此外，假设操作码相差 1。许多算术和逻辑指令遵循这个方案，所以定义需要尽可能紧凑。

第一个挑战是需要操作值，可用于修改值的操作符数量有限。例如，要生成 1 和字段操作码的值的和:

```
!add(opcode, 1)
```

这样的表达式最好用作类的参数。测试一个字段值，需要不可用的动态语句，所以不可能根据找到的值更改。记住，所有的计算都是在构造记录时完成的!

类似地，字符串也可以连接起来:

```
!strconcat(mnemonic,"i")
```

因为所有操作符都以感叹号 (!) 开头，所以也称为 bang 操作符。可以在开发者参考:https://llvm.org/docs/TableGen/ProgRef.html#appendix-a-bang-operators中找到 bang 操作符的完整列表。

现在，可以定义一个多类。Inst 类再次作为基类:

```
class Inst<string mnemonic, int opcode> {
    string Mnemonic = mnemonic;
    int Opcode = opcode;
}
```

多类的定义有点复杂，分步骤来做:

1. 多类的定义使用与类相似的语法。新的多类名为 InstWithImm，有两个参数，助记符和操作码:

```
multiclass InstWithImm<string mnemonic, int opcode> {
```

2. 首先，定义一条带有两个寄存器操作数的指令。与普通的记录定义一样，使用 def 关键字定义记录，并使用 Inst 类创建记录内容。还需要定义一个空名称，将在后面解释为什么这么做:

```
    def "": Inst<mnemonic, opcode>;
```

3. 接下来，用直接操作数定义一条指令。可以使用 bang 操作符从多类的参数中派生助记符和操作码的值，记录命名为 I:

```
    def I: Inst<!strconcat(mnemonic,"i"), !add(opcode, 1)>;
```

4. 这就是全部了，类主体已经完成了:

```
}
```

要实例化记录，必须使用 defm 关键字:

```
defm ADD : InstWithImm<"add", 0xA0>;
```

语句的结果如下所示:

1. Inst<”add”, 0xA0> 记录实例化，记录的名称由 multiclass 语句中 defm 关键字后面的名称和 def 后面的名称拼接而成，因此名称为 ADD。
2. 为基于类型的别名分析生成元数据: 将向 LLVM IR 添加的元数据，这将帮助 LLVM 更好地优化代码
3. Inst<”addi”, 0xA1> 记录实例化，并按照相同的模式命名为 ADDI。

用 llvm-tblgen 验证这个声明:

```
$ llvm-tblgen -print-records inst.td
------------- Classes -----------------
class Inst<string Inst:mnemonic = ?, int Inst:opcode = ?> {
    string Mnemonic = Inst:mnemonic;
    int Opcode = Inst:opcode;
}
------------- Defs -----------------
def ADD { // Inst
    string Mnemonic = "add";
    int Opcode = 160;
    }
def ADDI { // Inst
    string Mnemonic = "addi";
    int Opcode = 161;
}
```

使用 multiclass，可以一次生成多个记录。这个功能会经常使用!

记录不需要有名称，匿名记录完全没问题，省略名称是定义匿名记录所需要做的全部工作。由多类生成的记录的名称由两个名称组成，并且必须给出两个名称才能创建命名记录。若省略 def 后面的名称，则只创建匿名记录。若多类中的 def 后面没有名称，则会创建一个匿名记录。这就是多类示例中的第一个定义使用空名称”” 的原因: 没有它，记录是匿名的。

8.3.3. 模拟函数调用

某些情况下，使用前面例子中的多分类可能会导致重复。假定 CPU 也支持内存操作数，类似于直接操作数。可以通过在多类中添加一个新的记录来定义:

```
multiclass InstWithOps<string mnemonic, int opcode> {
    def "": Inst<mnemonic, opcode>;
    def "I": Inst<!strconcat(mnemonic,"i"), !add(opcode, 1)>;
    def "M": Inst<!strconcat(mnemonic,"m"), !add(opcode, 2)>;
}
```

这完全没问题，但假设要定义的记录不是 3 条，而是 16 条，并且需要多次这样做。出现这种情况的场景是，CPU 支持许多向量类型，并且向量指令根据所使用的类型略有不同。

请注意，带有 def 语句的所有三行都具有相同的结构。变化只存在于名称和助记符的后缀中，增量值会添加到操作码中。C 语言中，可以将数据放入一个数组中，并实现一个函数，该函数根据索引值返回数据。然后，可以在数据上创建一个循环，而不是手动重复书写语句。

令人惊讶的是，可以在 TableGen 语言中做类似的事情! 下面是如何转换示例:

1. 要存储数据，需要定义一个包含所有必需字段的类。这个类称为 InstDesc，其描述了指令的一些属性:

```
class InstDesc<string name, string suffix, int delta> {
    string Name = name;
    string Suffix = suffix;
    int Delta = delta;
}
```

2. 现在，可以为每个操作数类型定义记录。注意，其准确地捕获了数据中观察到的差异:

```
def RegOp : InstDesc<"", "", 0>;
def ImmOp : InstDesc<"I", """, 1>;
def MemOp : InstDesc""""","""", 2>;
```

3. 假设有一个枚举数字 0、1 和 2 的循环，并且希望根据索引选择先前定义的记录之一，需要怎么做呢? 解决方案是创建一个以索引作为参数的 getDesc 类。它有一个字段 ret，可以将其解释为返回值。要给这个字段赋正确的值，使用第二个操作符:

```
class getDesc<int n> {
    InstDesc ret = !cond(!eq(n, 0) : RegOp,
                         !eq(n, 1) : ImmOp,
                         !eq(n, 2) : MemOp);
}
```

此操作符的工作方式类似于 C 中的 switch/case 语句。

4. 现在，可以定义多类了。TableGen 语言有一个循环语句，允许定义变量，但没有动态执行! 因此，循环范围是静态定义的，可以为变量赋值，但不能更改该值，但这也足以检索数据。注意 getDesc 类的使用类似于函数调用，但是没有函数调用! 相反，将创建一个匿名记录，并从该记录中获取值。最后，past 操作符 (#) 执行字符串连接，类似于前面使用的!strconcat 操作符:

```
multiclass InstWithOps<string mnemonic, int opcode> {
    foreach I = 0-2 in {
        defvar Name = getDesc<I>.ret.Name;
```

```
        defvar Suffix = getDesc<I>.ret.Suffix;
        defvar Delta = getDesc<I>.ret.Delta;
        def Name: Inst<mnemonic # Suffix,
                       !add(opcode, Delta)>;
    }
}
```

5. 现在，可以像以前一样使用 multiclass 来定义记录:

```
defm ADD : InstWithOps<"add", 0xA0>;
```

请运行 llvm-tblgen 查看记录。除了各种 ADD 记录之外，还将看到使用 getDesc 类生成的一些匿名记录。

几个 LLVM 后端的指令定义中使用了这种技术。根据现在所掌握的知识，阅读这些文件应该没有问题。

foreach 语句使用语法 0-2 来表示范围的边界，这叫做值域。另一种语法是使用三个点 (0…3)，这在数字为负数时很有用。最后，不局限于数值范围，还可以遍历元素列表，可以使用字符串或先前定义的记录。例如，可能喜欢使用 foreach 语句，但觉得使用 getDesc 类过于复杂。这种情况下，循环 InstDesc 记录是解决方案:

```
multiclass InstWithOps<string mnemonic, int opcode> {
    foreach I = [RegOp, ImmOp, MemOp] in {
        defvar Name = I.Name;
        defvar Suffix = I.Suffix;
        defvar Delta = I.Delta;
        def Name: Inst<mnemonic # Suffix, !add(opcode, Delta)>;
    }
}
```

目前，只使用最常用的语句在 TableGen 语言中定义了记录。下一节中，将了解如何从 TableGen 语言中定义的记录生成 C++ 源代码。

8.4. 使用 TableGen 文件生成 C++ 代码

上一节中，用 TableGen 语言定义了记录。使用这些记录，需要编写自己的 TableGen 后端，该后端可以生成 C++ 源代码或使用记录作为输入执行其他操作。

第 3 章中，Lexer 类的实现使用数据库文件来定义标记和关键字，各种查询函数使用该数据库文件。除此之外，数据库文件还用于实现关键字过滤器。关键字过滤器是一个哈希映射，使用 llvm::StringMap 类实现。当找到一个标识符时，就调用关键字过滤器来确定该标识符是否实际上是一个关键字。若仔细看看 ppprofiler 通道的实现，则将看到该函数经常调用，所以实现该功能很有用。

然而，这并不像看起来那么容易。例如，可以尝试用二进制搜索替换散列映射中的查找。这需要对数据库文件中的关键字进行排序。目前，在开发过程中，一个新的关键字可能会添加到错误的地方。确保关键字按正确顺序排列的唯一方法是，添加一些在运行时检查顺序的代码。

可以通过更改内存布局来加快标准二进制搜索。可以使用 Eytzinger 布局，而不是对关键字进行排序，其以宽度优先的顺序枚举搜索树。这种布局增加了数据的缓存局部性，加快了搜索速度。就我个人而言，不可能用手动的方式，在数据库文件中以宽度优先的顺序维护关键字。

另一种流行的搜索方法是生成最小完美哈希函数。若将新键插入动态哈希表 (如 llvm::StringMap)，则该键可能会映射到已占用的槽，这称为键碰撞。键冲突是不可避免的，已经开发了许多策略来缓解这个问题。但若知道所有的键，就可以构建没有键冲突的哈希函数，这样的哈希函数堪称完美。若不需要比键更多的槽，则称为最小键。可以高效地生成完美的哈希函数——例如，使用 gperf GNU 工具。

总而言之，使用关键字生成查找函数一定有原因，让我们将数据库文件转成到 TableGen!

8.4.1. 用 TableGen 语言定义数据

TokenKinds.def 数据库文件定义了三个不同的宏。TOK 宏用于没有固定拼写的令牌——用于整数字面值。PUNCTUATOR 宏用于所有类型的标点符号，并包含首选拼写。KEYWORD 宏定义了一个由字面量和一个标志组成的关键字，这个标志用于指示这个字面量在哪个语言级别上是关键字。例如，C++11 中就添加了 thread_local 关键字。

TableGen 语言中用来表达关键字的方法是，创建一个保存所有数据的 Token 类。然后，可以添加该类的子类，还需要一个 Flag 类，用于与关键字一起定义的标志，还需要一个类来定义关键字过滤器。这些类定义了基本的数据结构，可以在其他项目中重用，可以创建了一个 Keyword.td 文件:

1. 标志为名称和相关联的值建模，使用这些数据生成一个枚举很容易:

```
class Flag<string name, int val> {
    string Name = name;
    int Val = val;
}
```

2. Token 类用作基类，只是有一个名字。注意，这个类没有参数:

```
class Token {
    string Name;
}
```

3. Tok 类与数据库文件中相应的 Tok 宏具有相同的功能，表示没有固定拼写的标记。它派生自基类 Token，只是为名称添加了初始化:

```
class Tok<string name> : Token {
    let Name = name;
}
```

4. 以同样的方式，标点器类类似于 PUNCTUATOR 宏，为标记的拼写添加了一个字段:

```
class Punctuator<string name, string spelling> : Token {
    let Name = name;
    string Spelling = spelling;
}
```

5. 最后，Keyword 类需要一个标志列表:

```
class Keyword<string name, list<Flag> flags> : Token {
    let Name = name;
    list<Flag> Flags = flags;
}
```

6. 有了这些定义，现在可以为关键字过滤器定义一个类，称为 TokenFilter。其接受标记列表作为参数:

```
class TokenFilter<list<Token> tokens> {
    string FunctionName;
    list<Token> Tokens = tokens;
}
```

使用这些类定义，就能够从 TokenKinds.def 数据库文件中捕获所有数据。TinyLang 语言不使用标志，但实际使用的语言，如 C 和 C++ 经历了几次修订，通常需要标记。因此，我们使用 C 和 C++ 中的关键字作为示例，来创建一个 KeywordC.td 文件:

1. 首先，包含前面创建的类定义:

```
Include "Keyword.td"
```

2. 接下来，定义标志，该值为该标志的二进制值。注意，如何使用!or 操作符来为 KEYALL 标志创建值:

```
def KEYC99 : Flag<"KEYC99", 0x1>;
def KEYCXX : Flag<"KEYCXX", 0x2>;
def KEYCXX11: Flag<"KEYCXX11", 0x4>;
def KEYGNU : Flag<"KEYGNU", 0x8>;
def KEYALL : Flag<"KEYALL",
                    !or(KEYC99.Val, KEYCXX.Val,
                        KEYCXX11.Val , KEYGNU.Val)>;
```

3. 有些符号没有固定的拼写，例如注释:

```
def : Tok<"comment">;
```

4. 操作符是使用 Punctuator 类定义:

```
def : Punctuator<"plus", "+">;
def : Punctuator<"minus", "-">;
```

5. 关键字需要使用不同的标志:

```
def kw_auto: Keyword<"auto", [KEYALL]>;
def kw_inline: Keyword<"inline", [KEYC99,KEYCXX,KEYGNU]>;
def kw_restrict: Keyword<"restrict", [KEYC99]>;
```

6. 最后，是关键字过滤器的定义:

```
def : TokenFilter<[kw_auto, kw_inline, kw_restrict]>;
```

当然，该文件不包括 C 和 C++ 中的所有标记，其演示了定义的 TableGen 类的所有可能用法。基于这些 TableGen 文件，我们将在下一节中实现 TableGen 后端。

8.4.2. 实现 TableGen 后端

由于解析和创建记录是通过 LLVM 库完成的，只需要关心后端实现，主要由基于记录中的信息生成 C++ 源代码片段组成。需要清楚要生成什么源代码，才能将其放入后端。

绘制要生成的源码的草图

TableGen 工具的输出是一个包含 C++ 片段的文件，这些片段由宏保护。目标是替换 TokenKinds.def 数据库文件。根据 TableGen 文件中的信息，可以生成以下内容:

1. 用于定义标志的枚举成员。开发者可以自由命名类型，但应该基于无符号类型。若生成的文件名为 TokenKinds.inc，预期的使用方式为:

```
enum Flags : unsigned {
    #define GET_TOKEN_FLAGS
    #include "TokenKinds.inc"
}
```

2. TokenKind 枚举，以及 getTokenName()、getPunctuatorSpelling() 和 getKeywordSpelling() 函数的原型和定义。这段代码替换了 TokenKinds.def 数据库文件，大部分 TokenKinds.h 包含文件和 TokenKinds.cpp 源文件。
3. 新的 lookupKeyword() 函数，可以用来代替使用 llvm::StringMap 类型的当前实现，这就是要优化的函数。

了解了想要生成的内容，就可以开始实现后端了。

创建一个新的 TableGen 工具

新工具使用一个驱动程序，该驱动计算命令行选项，并在不同的文件中调用生成函数和实际的生成器函数。我们将驱动文件命名为 TableGen.cpp，将包含生成器的文件命名为 TokenEmitter.cpp，还需要 TableGenBackends.h 头文件。让我们从生成 TokenEmitter.cpp 文件中的 C++ 代码开始实现:

1. 像往常一样，文件以包含所需的头文件开始。最重要的是 llvm/TableGen/Record.h，定义了 Record 类，用于保存解析.td 文件生成的记录:

```
#include "TableGenBackends.h"
#include "llvm/Support/Format.h"
#include "llvm/TableGen/Record.h"
#include "llvm/TableGen/TableGenBackend.h"
#include <algorithm>
```

2. 为了简化编码，导入了 llvm 命名空间:

```
using namespace llvm;
```

3. TokenAndKeywordFilterEmitter 类负责生成 C++ 源代码。如前一节所述，emitFlagsFragment()、emitTokenKind() 和 emitKeywordFilter() 方法会生成源代码，其中包含要生成的源代码的草图。唯一的公共方法 run() 调用了所有编写代码的方法，记录保存在 RecordKeeper 的一个实例中，并作为参数传递给构造函数。这个类位于匿名命名空间中:

```
namespace {
class TokenAndKeywordFilterEmitter {
    RecordKeeper &Records;

public:
    explicit TokenAndKeywordFilterEmitter(RecordKeeper &R)
        : Records(R) {}

    void run(raw_ostream &OS);

private:
    void emitFlagsFragment(raw_ostream &OS);
    void emitTokenKind(raw_ostream &OS);
    void emitKeywordFilter(raw_ostream &OS);
};
} // End anonymous namespace
```

4. run() 方法调用所有发出方法，也乘以每个相位的长度。可用--time-phases 选项设置，再在所有代码生成后显示时间:

```
void TokenAndKeywordFilterEmitter::run(raw_ostream &OS) {
    // Emit Flag fragments.
    Records.startTimer("Emit flags");
    emitFlagsFragment(OS);

    // Emit token kind enum and functions.
    Records.startTimer("Emit token kind");
    emitTokenKind(OS);

    // Emit keyword filter code.
    Records.startTimer("Emit keyword filter");
    emitKeywordFilter(OS);
    Records.stopTimer();
}
```

5. emitFlagsFragment() 方法显示了生成 C++ 源代码的函数的典型结构，生成的代码由 GET_TOKEN_FLAGS 宏保护。要生成 C++ 源片段，需要循环遍历 TableGen 文件中从 Flag 类派生的所有记录。有了这样的记录，就很容易查询记录的名称和值。注意，名称 Flag、Name 和 Val 必须与 TableGen 文件中的完全相同。若在 TableGen 文件中将 Val 重命名为 Value，则还需要更改此函数中的字符串。所有生成的源代码都写入提供的流，OS:

```
void TokenAndKeywordFilterEmitter::emitFlagsFragment(
raw_ostream &OS) {
    OS << "#ifdef GET_TOKEN_FLAGS\n";
```

```
    OS << "#undef GET_TOKEN_FLAGS\n";
    for (Record *CC :
            Records.getAllDerivedDefinitions("Flag")) {
        StringRef Name = CC->getValueAsString("Name");
        int64_t Val = CC->getValueAsInt("Val");
        OS << Name << " = " << format_hex(Val, 2) << ",\n";
    }
    OS << "#endif\n";
}
```

6. emitTokenKind() 方法发出标记分类函数的声明和定义。先看一下如何发出声明，整体结构与前一种方法相同——只是要生成更多的 C++ 源码，生成的源码由 GET_TOKEN_KIND_DECLARATION 宏保护。注意，此方法尝试生成格式良好的 C++ 代码，使用新行和缩进，就像人类开发者一样。若生成的源码不正确，并且需要检查它以找到错误，这将非常有帮助。也很容易犯这样的错误: 毕竟，正在编写一个生成 C++ 源代码的 C++ 函数。

   首先，生成 TokenKind 枚举。关键字的名称应该以 kw_ 字符串作为前缀。循环遍历 Token 类的所有记录，若也是 Keyword 类的子类，则可以查询这些记录，生成的前缀为:

```
    OS << "#ifdef GET_TOKEN_KIND_DECLARATION\n"
        << "#undef GET_TOKEN_KIND_DECLARATION\n"
        << "namespace tok {\n"
        << " enum TokenKind : unsigned short {\n";
    for (Record *CC :
            Records.getAllDerivedDefinitions("Token")) {
        StringRef Name = CC->getValueAsString("Name");
        OS << " ";
        if (CC->isSubClassOf("Keyword"))
        OS << "kw_";
        OS << Name << ",\n";
    }
    OS << " NUM_TOKENS\n"
       << " };\n";
```

7. 接下来，生成函数声明。这只是一个常量字符串，这将完成声明生成:

```
    OS << " const char *getTokenName(TokenKind Kind) "
          "LLVM_READNONE;\n"
        << " const char *getPunctuatorSpelling(TokenKind "
          "Kind) LLVM_READNONE;\n"
        << " const char *getKeywordSpelling(TokenKind "
          "Kind) "
          "LLVM_READNONE;\n"
        << "}\n"
        << "#endif\n";
```

8. 现在，转向生成定义。同样，生成的代码由一个名为 GET_TOKEN_KIND_DEFINITION 的宏保护。首先，将标记名称发送到 TokNames 数组中，getTokenName() 函数使用该数组检索名

称。注意，当在字符串中使用引号符号时，必须转义为\”:

```
    OS << "#ifdef GET_TOKEN_KIND_DEFINITION\n";
    OS << "#undef GET_TOKEN_KIND_DEFINITION\n";
    OS << "static const char * const TokNames[] = {\n";
    for (Record *CC :
        Records.getAllDerivedDefinitions("Token")) {
        OS << " \"" << CC->getValueAsString("Name")
           << "\",\n";
    }
    OS << "};\n\n";
    OS << "const char *tok::getTokenName(TokenKind Kind) "
          "{\n"
       << " if (Kind <= tok::NUM_TOKENS)\n"
       << " return TokNames[Kind];\n"
       << " llvm_unreachable(\"unknown TokenKind\");\n"
       << " return nullptr;\n"
       << "};\n\n";
```

9. 接下来，生成 getPunctuatorSpelling() 函数。与其他部分的区别是，循环遍历从 punctuation 类派生的所有记录。还会生成一个 switch 语句，而不是一个数组:

```
    OS << "const char "
          "*tok::getPunctuatorSpelling(TokenKind "
          "Kind) {\n"
       << " switch (Kind) {\n";
    for (Record *CC :
            Records.getAllDerivedDefinitions("Punctuator")) {
        OS << " " << CC->getValueAsString("Name")
           << ": return \""
           << CC->getValueAsString("Spelling") << "\";\n";
    }
    OS << " default: break;\n"
       << " }\n"
       << " return nullptr;\n"
       << "};\n\n";
```

10. 最后，生成 getKeywordSpelling() 函数。代码类似于发出 getPunctuatorSpelling()，这次循环遍历 Keyword 类的所有记录，并且该名称以 kw_为前缀:

```
    OS << "const char *tok::getKeywordSpelling(TokenKind "
          "Kind) {\n"
       << " switch (Kind) {\n";
    for (Record *CC :
         Records.getAllDerivedDefinitions("Keyword")) {
        OS << " kw_" << CC->getValueAsString("Name")
           << ": return \"" << CC->getValueAsString("Name")
           << "\";\n";
    }
    OS << " default: break;\n"
```

```
        << " }\n"
        << " return nullptr;\n"
        << «};\n\n»;
    OS << «#endif\n»;
}
```

11. 因为生成过滤器需要从记录中收集一些数据，所以 emitKeywordFilter() 方法比前面的方法更复杂。生成的源代码使用 std::lower_bound() 函数，从而实现二分查找。
TableGen 文件中可以定义 TokenFilter 类的多个记录。出于演示目的，最多只生成一个标记过滤器方法:

```
    std::vector<Record *> AllTokenFilter =
        Records.getAllDerivedDefinitionsIfDefined(
            "TokenFilter");
    if (AllTokenFilter.empty())
        return;
```

12. 用于筛选器的关键字位于名为 Tokens 的列表中。要访问该列表，首先需要查找记录中的 Tokens 字段。这将返回一个指向 RecordVal 类实例的指针，可以通过调用方法 getValue() 从中检索 Initializer 实例。Tokens 字段可定义为一个列表，可将初始化器实例强制转换为 ListInit。若失败，则退出该函数:

```
    ListInit *TokenFilter = dyn_cast_or_null<ListInit>(
        AllTokenFilter[0]
            ->getValue("Tokens")
            ->getValue());
    if (!TokenFilter)
        return;
```

13. 现在，可以构造一个过滤器表了。对于 TokenFilter 列表中存储的每个关键字，需要 Flag 字段的名称和值。该字段再次定义为列表，因此需要遍历这些元素以计算最终值。结果名称/标志值对存储在 Table vector 中:

```
    using KeyFlag = std::pair<StringRef, uint64_t>;
    std::vector<KeyFlag> Table;
    for (size_t I = 0, E = TokenFilter->size(); I < E;
            ++I) {
        Record *CC = TokenFilter->getElementAsRecord(I);
        StringRef Name = CC->getValueAsString("Name");
        uint64_t Val = 0;
        ListInit *Flags = nullptr;
        if (RecordVal *F = CC->getValue("Flags"))
            Flags = dyn_cast_or_null<ListInit>(F->getValue());
        if (Flags) {
            for (size_t I = 0, E = Flags->size(); I < E; ++I) {
                Val |=
                Flags->getElementAsRecord(I)->getValueAsInt(
                "Val");
            }
```

```
        }
        Table.emplace_back(Name, Val);
    }
```

14. 为了能够执行二分查找，需要对表进行排序。比较函数由 Lambda 函数提供:

```
    llvm::sort(Table.begin(), Table.end(),
               [](const KeyFlag A, const KeyFlag B) {
                   return A.first < B.first;
               });
```

15. 现在，可以生成 C++ 源代码了。首先，生成包含关键字名称和相关标志值的排序表:

```
    OS << "#ifdef GET_KEYWORD_FILTER\n"
       << "#undef GET_KEYWORD_FILTER\n";
    OS << "bool lookupKeyword(llvm::StringRef Keyword, "
          "unsigned &Value) {\n";
    OS << " struct Entry {\n"
       << " unsigned Value;\n"
       << " llvm::StringRef Keyword;\n"
       << " };\n"
       << "static const Entry Table[" << Table.size()
       << "] = {\n";
    for (const auto &[Keyword, Value] : Table) {
        OS << " { " << Value << ", llvm::StringRef(\""
           << Keyword << "\", " << Keyword.size()
           << ") },\n";
    }
    OS << " };\n\n";
```

16. 接下来，使用标准 C++ 函数 std::lower_bound() 在排序表中查找关键字。若关键字在表中，则 Value 参数接收与关键字关联的标志的值，函数返回 true。否则，函数返回 false:

```
    OS << " const Entry *E = "
          "std::lower_bound(&Table[0], "
          "&Table["
       << Table.size()
       << "], Keyword, [](const Entry &A, const "
          "StringRef "
          "&B) {\n";
    OS << " return A.Keyword < B;\n";
    OS << " });\n";
    OS << " if (E != &Table[" << Table.size()
       << "]) {\n";
    OS << " Value = E->Value;\n";
    OS << " return true;\n";
    OS << " }\n";
    OS << " return false;\n";
    OS << "}\n";
    OS << "#endif\n";
```

```
}
```

17. 现在唯一缺少的部分是调用该实现的方法，为此定义了一个全局函数 EmitTokensAndKeywordFilter()。在 llvm/TableGen/TableGenBackend.h 头文件中声明的 emitSourceFileHeader() 函数，在生成的文件的顶部生成注释:

```
void EmitTokensAndKeywordFilter(RecordKeeper &RK,
                                raw_ostream &OS) {
    emitSourceFileHeader("Token Kind and Keyword Filter "
                         "Implementation Fragment",
                         OS);
    TokenAndKeywordFilterEmitter(RK).run(OS);
}
```

这样，就在 TokenEmitter.cpp 文件中完成了源生成器的实现，代码不算太复杂。

TableGenBackends.h 头文件只包含 EmitTokensAndKeywordFilter() 函数的声明。为了避免包含其他文件，可以对 raw_ostream 和 RecordKeeper 类使用前向声明:

```
#ifndef TABLEGENBACKENDS_H
#define TABLEGENBACKENDS_H

namespace llvm {
    class raw_ostream;
    class RecordKeeper;
} // namespace llvm

void EmitTokensAndKeywordFilter(llvm::RecordKeeper &RK,
                                llvm::raw_ostream &OS);
#endif
```

缺少的部分是驱动程序的实现，其任务是解析 TableGen 文件并根据命令行选项发出记录。实现在 TableGen.cpp 文件中:

- 与往常一样，实现从包含所需的头文件开始。最重要的一个是 llvm/TableGen/Main.h，这个头文件声明了 TableGen 的前端:

```
#include "TableGenBackends.h"
#include "llvm/Support/CommandLine.h"
#include "llvm/Support/PrettyStackTrace.h"
#include "llvm/Support/Signals.h"
#include "llvm/TableGen/Main.h"
#include "llvm/TableGen/Record.h"
```

- 为了简化编码，导入了 llvm 命名空间:

```
using namespace llvm;
```

- 用户可以选择一个操作，ActionType 枚举包含所有可能的操作:

```
enum ActionType {
    PrintRecords,
    DumpJSON,
    GenTokens,
};
```

- 使用一个名为 Action 的命令行选项对象。用户需要指定--gen-tokens 选项，以生成实现的令牌过滤器。另外两个选项--print-records 和--dump-json 是转储读记录的标准选项。注意，该对象位于匿名命名空间中:

```
namespace {
cl::opt<ActionType> Action(
    cl::desc("Action to perform:"),
    cl::values(
        clEnumValN(
            PrintRecords, "print-records",
            "Print all records to stdout (default)"),
        clEnumValN(DumpJSON, "dump-json",
            "Dump all records as "
            "machine-readable JSON"),
        clEnumValN(GenTokens, "gen-tokens",
            "Generate token kinds and keyword "
            "filter")));
```

- Main() 函数基于 action 的值执行请求的操作，若在命令行上指定了--gen-tokens，则调用 EmitTokensAndKeywordFilter() 函数。函数结束后，匿名命名空间关闭:

```
bool Main(raw_ostream &OS, RecordKeeper &Records) {
    switch (Action) {
    case PrintRecords:
        OS << Records; // No argument, dump all contents
        break;
    case DumpJSON:
        EmitJSON(Records, OS);
        break;
    case GenTokens:
        EmitTokensAndKeywordFilter(Records, OS);
        break;
    }
    return false;
}
} // namespace
```

- 最后，定义一个 main() 函数。设置堆栈跟踪处理程序并解析命令行选项之后，将调用 TableGenMain() 函数来解析 TableGen 文件并创建记录。若没有错误，该函数也会调用 Main() 函数:

```
int main(int argc, char **argv) {
    sys::PrintStackTraceOnErrorSignal(argv[0]);
```

```
    PrettyStackTraceProgram X(argc, argv);
    cl::ParseCommandLineOptions(argc, argv);

    llvm_shutdown_obj Y;

    return TableGenMain(argv[0], &Main);
}
```

现在实现了 TableGen 工具。编译后，就可以使用 KeywordC 运行它。示例输入文件如下:

```
$ tinylang-tblgen --gen-tokens -o TokenFilter.inc KeywordC.td
```

生成的 C++ 源代码会写入 TokenFilter.inc 文件中。

标记过滤器的性能

对关键字过滤器使用纯二分查找，并不比基于 llvm::StringMap 类型的实现提供更好的性能。为了超越当前实现的性能，需要生成一个完美的哈希函数。

来自 Czech、Havas 和 Majewski 的经典算法可以很容易地实现，并且提供了非常好的性能。它描述在生成最小完美哈希函数的最优算法，信息处理通讯，第 43 卷，第 5 期，1992 年。参见https://www.sciencedirect.com/science/article/abs/pii/002001909290220P。

最先进的算法是 Pibiri 和 Trani 的《PTHash: Revisiting FCH Minimal Perfect Hashing, SIGIR '21》中进行了描述。参见https://arxiv.org/pdf/2104.10402.pdf。

这两种算法都是生成标记过滤器的不错选择，会比 llvm::StringMap 更快。

8.5. TableGen 的缺点

以下是 TableGen 的一些缺点:

- TableGen 语言建立在一个简单的概念之上，所以不具有与其 DSL 相同的计算能力。显然，一些程序员希望用一种不同的、更强大的语言来取代 TableGen，这个话题在 LLVM 讨论论坛中不时会看见。
- 由于可以实现自己的后端，TableGen 语言非常灵活，所以定义的语义隐藏在后端内部。因此，可以创建其他开发人员基本上无法理解的 TableGen 文件。
- 最后，若试图解决一个重要的任务，后端实现可能会非常复杂。我们有理由认为，若 TableGen 语言更强大，这种苦力活将会减少。

即使不是所有开发者都对 TableGen 的功能感到满意，但该工具在 LLVM 中广泛使用，对于开发者来说，理解它非常重要。

8.6. 总结

本章中，首先了解了 TableGen 背后的主要思想，并用 TableGen 语言定义了第一个类和记录，了解了 TableGen 语法的知识。最后，基于所定义的 TableGen 类，开发了用于生成 C++ 源码的 TableGen 后端。

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，也称为即时 (JIT) 编译。

第 9 章 JIT 编译

LLVM 核心库拥有 ExecutionEngine 组件，该组件允许在内存中编译和执行中间表示 (IR) 代码。使用这个组件，可以构建即时 (JIT) 编译器，其允许直接执行 IR 代码。因为不需要将目标代码存储在辅助存储器上，所以 JIT 编译器的工作方式更像解释器。

本章中，将了解 JIT 编译器的应用，以及 LLVM JIT 编译器的工作原理。将探索 LLVM 动态编译器和解释器，如何自己实现 JIT 编译器工具，如何将 JIT 编译器作为静态编译器的一部分使用，以及相关的挑战。

本章中，将了解以下内容:

- 获得 LLVM 的 JIT 实现和用例的概述
- 使用 JIT 编译直接执行
- 根据现有类实现自己的 JIT 编译器
- 从头开始实现自己的 JIT 编译器

本章结束时，将了解如何开发 JIT 编译器，可以使用预配置的类，也可以使用适合自己的版本。

9.1. 环境要求

本章使用的代码在这里，https://github.com/PacktPublishing/Learn-LLVM-17/tree/main/Chapter09。

9.2. LLVM 的整体 JIT 的实现和用例

目前，我们只讨论了提前 (AOT) 编译器，这些编译器编译整个应用程序。应用程序只能在编译完成后运行，若编译是在应用程序的运行时执行的，编译器就是 JIT 编译器。JIT 编译器有一些有趣的用例:

- 实现虚拟机: 编程语言可以用 AOT 编译器翻译成字节码。运行时，使用 JIT 编译器将字节码编译为机器码。这种方法的优点是字节码与硬件无关，并且由于 JIT 编译器，与 AOT 编译器相比没有性能损失。Java 和 C# 现在使用这个模型，但这并不是一个新想法:1977 年的 USCD Pascal 编译器已经使用了类似的方法。
- 表达式求值: 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。例如，这可以加快金融模拟。lldb LLVM 调试器使用这种方法在调试时计算表达式。
- 数据库查询: 数据库从数据库查询创建一个执行计划。执行计划描述了对表和列的操作，执行时产生查询答案。可以使用 JIT 编译器将执行计划转换为机器码，从而加快查询的执行速度。

LLVM 的静态编译模型与 JIT 模型的距离并不像人们想象的那么遥远，LLVM 静态编译器将 LLVM IR 编译成机器码，并将结果保存为磁盘上的目标文件。若目标文件不是存储在磁盘上，而是存储在内存中，代码是可执行的吗? 不是直接的，对全局函数和全局数据的引用使用重定位而不是绝对地址。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。若像链接器和动态加载器将重定位解析为地址，就可以执行目标代码了。运行静态编译器将 IR 代码编译成

内存中的对象文件，在内存中的对象文件上执行链接步骤，然后运行代码，就得到了一个 JIT 编译器。

LLVM 核心库中的 JIT 实现就是基于这个思想。LLVM 的开发历史中，有几个 JIT 实现，具有不同的功能集。最新的 JIT API 是按请求编译 (ORC) 引擎。若对这个缩略词感到好奇，在可执行和链接格式 (ELF) 和调试标准 (DWARF) 出现之后，首席开发人员打算根据托尔金的世界创造另一个缩略词。

ORC 引擎基于并扩展了在内存中，对象文件上使用静态编译器和动态链接器的思想。实现使用分层的方法。两个基本层次是编译层和链接层，在此之上有一层提供延迟编译支持。转换层可以堆叠在惰性编译层之上或之下，允许开发人员添加转换或简单地收到某些事件的通知。此外，这种分层的方法还有一个优点，即 JIT 引擎可以针对不同的需求进行定制。例如，高性能的虚拟机可能选择提前编译所有内容，而不使用延迟编译层。另一方面，其他虚拟机将强调启动时间和对用户的响应，并将在惰性编译层的帮助下实现这一点。

旧的 MCJIT 引擎仍然可用，其 API 来自一个更旧的、已经删除的 JIT 引擎。随着时间的推移，这个 API 逐渐变得臃肿，缺乏 ORC API 的灵活性。

我们的目标是删除这个实现，因为 ORC 引擎现在提供了 MCJIT 引擎的所有功能，新的开发应该使用 ORC API。下一节中，深入实现 JIT 编译器之前，我们将研究 LLVM 解释器和动态编译器。

9.3. 使用 JIT 直接执行

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讨 lli 工具。

9.3.1. 探索 lli 工具

让我们用一个非常简单的例子来试试 lli 工具。下面的 LLVM IR 可以存储为一个名为 hello.ll 的文件，相当于 C 语言中的 hello world 应用程序。该文件声明了来自 C 库的 printf() 函数的原型，hellostr 常量包含要打印的消息。在 main() 函数内部，生成对 printf() 函数的调用，该函数包含将打印的 hellostr 消息，应用程序总是返回 0。

完整的代码如下所示:

```
declare i32 @printf(ptr, ...)

@hellostr = private unnamed_addr constant [13 x i8] c"Hello
world\0A\00"

define dso_local i32 @main(i32 %argc, ptr %argv) {
    %res = call i32 (ptr, ...) @printf(ptr @hellostr)
    ret i32 0
}
```

这个 LLVM IR 文件是通用的，对所有平台都有效。可以使用 cli 工具直接执行 IR，命令如下:

```
$ lli hello.ll
Hello world
```

这里有趣的一点是如何找到 printf() 函数。IR 代码会编译为机器码，并触发 printf 符号的查找。这个符号在 IR 中找不到，所以在当前进程中搜索它，lli 工具会动态链接到 C 库，在那里可以找到该符号。

当然，lli 工具不会链接到创建的库。为了使用这些函数，lli 工具支持加载动态库和对象。下面的 C 代码只输出消息:

```
#include <stdio.h>

void greetings() {
    puts("Hi!");
}
```

存储在 greetings.c 中，使用它来探索用 lli 加载对象。下面的命令将把这个源代码编译成一个动态库，-fPIC 选项指示 clang 生成与位置无关的代码，这是动态库所必需的。此外，编译器还会用-shared 创建一个 greetings.so 动态库:

```
$ clang greetings.c -fPIC -shared -o greetings.so
```

我们还将该文件编译为 greetings.o 对象文件:

```
$ clang greetings.c -c -o greetings.o
```

现在有两个文件，greetings.so 动态库和 greetings.o 对象文件，加载到 lli 工具中。

还需要一个调用 greetings() 函数的 LLVM IR 文件，还要创建一个 main.ll 文件，包含单个函数调用:

```
declare void @greetings(...)

define dso_local i32 @main(i32 %argc, i8** %argv) {
    call void (...) @greetings()
    ret i32 0
}
```

执行时，因为 lli 无法定位 greetings 符号，所以前面的 IR 崩溃了，:

```
$ lli main.ll
JIT session error: Symbols not found: [ _greetings ]
lli: Failed to materialize symbols: { (main, { _main }) }
```

greetings() 函数是在一个外部文件中定义的，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，该选项将动态库的路径作为参数:

```
$ lli -load ./greetings.so main.ll
Hi!
```

若包含动态库的目录不在动态加载器的搜索路径中，则指定动态库的路径非常重要。若省略，则将找不到库。

或者，用-extra-object 命令 lli 加载目标文件:

```
$ lli -extra-object greetings.o main.ll
Hi!
```

其他支持的选项有-extra-archive(加载存档文件) 和-extramodule(加载另一个位码文件)。这两个选项都需要文件的路径作为参数。

现在，了解了如何使用 lli 工具直接执行 LLVM IR。下一节中，我们将实现自己的 JIT 工具。

9.4. 用 LLJIT 实现 JIT 编译器

lli 工具只不过是 LLVM API 的一个包装。第一部分中，我们了解了 ORC 引擎使用分层方法。ExecutionSession 类表示一个正在运行的 JIT 程序。除了其他项之外，这个类还保存诸如使用过的 JITDylib 实例之类的信息。JITDylib 实例是一个符号表，将符号名映射到地址。例如，这些符号可以是在 LLVM IR 文件中定义的符号，也可以加载动态库的符号。

为了执行 LLVM IR，LLJIT 类提供了这个功能，不需要自己创建 JIT 堆栈。这个类提供了相同的功能，所以从旧的 MCJIT 实现迁移时，也可以使用这个类。

为了说明 LLJIT 工具的功能，我们将在合并 JIT 功能的同时创建一个交互式计算器应用程序。我们的 JIT 计算器源码，将使用第 2 章中的 calc 示例进行扩展。

交互式 JIT 计算器的主要思想:

1. 允许用户输入函数定义，例如定义 $f(x) = x * 2$.
2. 然后，用户输入的函数由 LLJIT 实用程序编译成一个函数——本例中为 f。
3. 允许用户用一个数值来调用定义的函数:$f(3)$.
4. 使用提供的参数计算函数，并将结果输出到控制台: 6

讨论将 JIT 功能整合到计算器源代码之前，与原始计算器示例相比，其有几个区别:

- 首先，之前只输入和解析了以 with 关键字开头的函数，而不是前面描述的 def 关键字。本章中，只接受以 def 开头的函数定义，这在抽象语法树 (AST) 类中表示为一个特定的节点，称为 DefDecl。DefDecl 类知道定义它的参数及其名称，当然函数名称也存储在这个类中。
- 其次，还需要 AST 能够识别函数调用，以表示 LLJIT 实用程序使用或 JIT 处理的函数。无论何时用户输入函数名，后面跟着括号内的参数，AST 都会将其识别为 FuncCallFromDef 节点。这个类本质上了解与 DefDecl 类相同的信息。

由于添加了这两个 AST 类，语义分析、解析器和代码生成类将相应地进行调整，以处理 AST 中的更改。另外添加了一个新的数据结构，称为 JITtedFunctions。这个数据结构是一个映射，将定

义的函数名作为键，函数定义的参数数量作为值存储在映射中，稍后我们将了解如何在 JIT 计算器中使用此数据结构。

有关我们对 calc 示例所做更改的更多详细信息，可以在 lljit 源目录中找到包含 calc 更改和本节 JIT 实现的完整源码。

9.4.1. 将 LLJIT 引擎集成到计算器中

首先，讨论如何在交互式计算器中设置 JIT 引擎。所有与 JIT 引擎相关的实现都存在于 Calc.cpp 中，并且该文件有一个用于程序执行的 main() 循环:

1. 除了包含代码生成、语义分析器和解析器实现的头文件外，还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。InitLLVM.h 头文件用于工具的基本初始化，TargetSelect.h 头文件用于本机目标的初始化，还包含了 C++ 标准头文件 <iostream>，允许用户进行输入:

```
#include "CodeGen.h"
#include "Parser.h"
#include "Sema.h"
#include "llvm/ExecutionEngine/Orc/LLJIT.h"
#include "llvm/Support/InitLLVM.h"
#include "llvm/Support/TargetSelect.h"
#include <iostream>
```

2. 接下来，将 llvm 和 llvm::orc 命名空间添加到当前作用域:

```
using namespace llvm;
using namespace llvm::orc;
```

3. 将创建的 LLJIT 实例中的许多调用，都会返回一个错误类型 error。ExitOnError 类允许丢弃在日志记录的 stderr，并退出应用程序时由 LLJIT 实例调用返回的错误值。我们声明一个全局的 ExitOnError 变量如下:

```
ExitOnError ExitOnErr;
```

4. 然后，添加 main() 函数，初始化工具和本机目标:

```
int main(int argc, const char **argv{
    InitLLVM X(argc, argv);
    InitializeNativeTarget();
    InitializeNativeTargetAsmPrinter();
    InitializeNativeTargetAsmParser();
```

5. 使用 LLJITBuilder 类来创建一个 LLJIT 实例，其封装在前面声明的 ExitOnErr 变量中，以免发生错误。一个可能的错误是，平台还不支持 JIT 编译:

```
auto JIT = ExitOnErr(LLJITBuilder().create());
```

6. 接下来，声明 JITtedFunctions 映射，该映射跟踪函数定义:

```
StringMap<size_t> JITtedFunctions;
```

7. 为了创造一个等待用户输入的环境，添加了一个 while() 循环，允许用户输入一个表达式，将用户输入的那一行保存在一个名为 calcExp 的字符串中:

```
    while (true) {
        outs() << "JIT calc > ";
        std::string calcExp;
        std::getline(std::cin, calcExp);
```

8. 初始化 LLVM 上下文类，以及一个新的 LLVM 模块。模块的数据布局也相应地设置，还声明了一个代码生成器，将用于为用户在命令行上定义的函数生成 IR:

```
    std::unique_ptr<LLVMContext> Ctx = std::make_unique<LLVMContext>();
    std::unique_ptr<Module> M = std::make_unique<Module>("JIT calc.expr", *Ctx);
    M->setDataLayout(JIT->getDataLayout());
    CodeGen CodeGenerator;
```

9. 必须解释用户输入的代码行，以确定用户是在定义一个新函数，还是在调用之前用参数定义的函数。Lexer 类在接收用户输入行时定义，可了解词法分析器主要关心的两种情况:

```
    Lexer Lex(calcExp);
    Token::TokenKind CalcTok = Lex.peek();
```

10. 词法分析器可以检查用户输入的第一个标记，若用户正在定义一个新函数 (由 def 关键字或 Token::KW_def 表示)，则解析它并检查其语义。若解析器或语义分析器检测到用户定义函数的问题，将相应地发出错误，计算器程序将停止运行。若解析器或语义分析器都没有检测到错误，就有一个有效的 AST 数据结构——DefDecl:

```
    if (CalcTok == Token::KW_def) {
        Parser Parser(Lex);
        AST *Tree = Parser.parse();
        if (!Tree || Parser.hasError()) {
            llvm::errs() << "Syntax errors occured\n";
            return 1;
        }
        Sema Semantic;
        if (Semantic.semantic(Tree, JITtedFunctions)) {
            llvm::errs() << "Semantic errors occured\n";
            return 1;
        }
```

11. 然后，可以将新构造的 AST 传递到代码生成器中，为用户定义的函数编译 IR。IR 生成的细节将在后面讨论，但这个编译成 IR 的函数需要知道模块和 JITtedFunctions 的映射关系。生成 IR 之后，可以调用 addIRModule() 将这些信息添加到我们的 LLJIT 实例中，并将我们的模块和上下文包装在 ThreadSafeModule 类中，以防止其他线程并发访问:

```
        CodeGenerator.compileToIR(Tree, M.get(), JITtedFunctions); ExitOnErr(
        ↪  JIT->addIRModule(ThreadSafeModule(std::move(M), std::move(Ctx))));
```

12. 相反，若用户调用一个带参数的函数，由 Token::ident 表示，还需要在将用户输入转换为有效

AST 之前解析和语义检查用户输入是否有效。这里，解析和检查与之前略有不同，因为其可以包括检查，例如：确保用户提供给函数调用的参数数量与函数最初定义的参数数量匹配:

```
    } else if (CalcTok == Token::ident) {
        outs() << "Attempting to evaluate expression:\n";
        Parser Parser(Lex);
        AST *Tree = Parser.parse();
        if (!Tree || Parser.hasError()) {
            llvm::errs() << "Syntax errors occured\n";
            return 1;
        }
        Sema Semantic;
        if (Semantic.semantic(Tree, JITtedFunctions)) {
            llvm::errs() << "Semantic errors occured\n";
            return 1;
        }
```

13. 为函数调用构造了一个有效的 AST, FuncCallFromDef，可以从 AST 中获取函数的名称，然后代码生成器准备生成对先前添加到 LLJIT 实例的函数的调用。背后的过程是，用户定义的函数重新生成为一个单独的函数中的 LLVM 调用，将创建该函数，用于实际评估原始函数。这一步需要 AST、模块、函数调用名和函数定义的映射:

```
        llvm::StringRef FuncCallName = Tree->getFnName();
        CodeGenerator.prepareCalculationCallFunc(Tree, M.get(), FuncCallName,
        ↪  JITtedFunctions);
```

14. 代码生成器完成了重新生成原始函数和创建单独的求值函数的工作之后，必须将这些信息添加到 LLJIT 实例中。创建 ResourceTracker 实例来跟踪分配给已添加到 LLJIT 的函数的内存，以及模块和上下文的另一个 ThreadSafeModule 实例，再将这两个实例作为 IR 模块添加到 JIT 中:

```
        auto RT = JIT->getMainJITDylib().createResourceTracker();
        auto TSM = ThreadSafeModule(std::move(M), std::move(Ctx));
        ExitOnErr(JIT->addIRModule(RT, std::move(TSM)));
```

15. 然后，通过 lookup() 方法在 LLJIT 实例中查询单独的求值函数，方法是将求值函数的名称 calc_expr_func 提供给该函数。若查询成功，则将 calc_expr_func 函数的地址强制转换为适当的类型，该函数不接受参数并返回单个整数。获取函数的地址后，调用该函数以使用用户定义函数提供的参数生成结果，在将结果输出到控制台:

```
        auto CalcExprCall = ExitOnErr(JIT->lookup("calc_expr_func"));
        int (*UserFnCall)() = CalcExprCall.toPtr<int (*)()>();
        outs() << "User defined function evaluated to:" << UserFnCall() << "\n";
```

16. 函数调用完成后，之前与函数关联的内存使用 ResourceTracker 释放:

```
ExitOnErr(RT->remove());
```

9.4.2. 修改代码生成——支持通过 LLJIT 进行 JIT 编译

现在，简要地看一下我们在 CodeGen.cpp 中所做的一些更改，以支持 JIT 版计算器:

- 代码生成类有两个重要的方法: 一个用于将用户定义函数编译成 LLVM IR 并将 IR 输出到控制台，另一个用于准备计算求值函数 calc_expr_func，该函数包含对原始用户定义函数的调用以进行求值。第二个函数也将 IR 输出给用户:

```
void CodeGen::compileToIR(AST *Tree, Module *M,
                    StringMap<size_t> &JITtedFunctions) {
    ToIRVisitor ToIR(M, JITtedFunctions);
    ToIR.run(Tree);
    M->print(outs(), nullptr);
}

void CodeGen::prepareCalculationCallFunc(AST *FuncCall,
        Module *M, llvm::StringRef FnName,
        StringMap<size_t> &JITtedFunctions) {
    ToIRVisitor ToIR(M, JITtedFunctions);
    ToIR.genFuncEvaluationCall(FuncCall);
    M->print(outs(), nullptr);
}
```

- 如上面的源代码所述，这些代码生成函数定义了一个 ToIRVisitor 实例，其接受我们的模块，并在初始化时在其构造函数中使用 JITtedFunctions 映射:

```
class ToIRVisitor : public ASTVisitor {
    Module *M;
    IRBuilder<> Builder;
    StringMap<size_t> &JITtedFunctionsMap;
. . .

public:
    ToIRVisitor(Module *M,
                StringMap<size_t> &JITtedFunctions)
        : M(M), Builder(M->getContext()),
        JITtedFunctionsMap(JITtedFunctions) {
```

- 最终，该信息用于生成 IR 或计算先前生成 IR 的函数。生成 IR 时，代码生成器希望看到 DefDecl 节点，其表示定义一个新函数。函数名及其定义的参数数量，存储在函数定义映射中:

```
virtual void visit(DefDecl &Node) override {
    llvm::StringRef FnName = Node.getFnName();
    llvm::SmallVector<llvm::StringRef, 8> FunctionVars =
    Node.getVars();
    (JITtedFunctionsMap)[FnName] = FunctionVars.size();
```

- 之后，实际的函数定义由 genUserDefinedFunction() 调用创建:

```
    Function *DefFunc = genUserDefinedFunction(FnName);
```

- genUserDefinedFunction() 中，第一步是检查函数是否存在于模块中。若没有，则确保函数原型存在于 map 数据结构中，再使用名称和实参数来构造一个函数，该函数具有用户定义的实参数，并使该函数返回单个整数值:

```
Function *genUserDefinedFunction(llvm::StringRef Name) {
    if (Function *F = M->getFunction(Name))
        return F;

    Function *UserDefinedFunction = nullptr;
    auto FnNameToArgCount = JITtedFunctionsMap.find(Name);
    if (FnNameToArgCount != JITtedFunctionsMap.end()) {
        std::vector<Type *> IntArgs(FnNameToArgCount->second,
        Int32Ty);
        FunctionType *FuncType = FunctionType::get(Int32Ty,
        IntArgs, false);
        UserDefinedFunction =
            Function::Create(FuncType,
            GlobalValue::ExternalLinkage, Name, M);
    }
    return UserDefinedFunction;
}
```

- 生成用户定义的函数之后，创建一个新的基本块，并将函数插入到基本块中。每个函数实参还与用户定义的名称相关联，因此可为所有函数实参设置名称，并生成对函数内实参进行操作的数学运算:

```
    BasicBlock *BB = BasicBlock::Create(M->getContext(),
    "entry", DefFunc);
    Builder.SetInsertPoint(BB);
    unsigned FIdx = 0;
    for (auto &FArg : DefFunc->args()) {
        nameMap[FunctionVars[FIdx]] = &FArg;
        FArg.setName(FunctionVars[FIdx++]);
    }
    Node.getExpr()->accept(*this);
};
```

- 计算用户定义函数时，示例中期望的 AST 称为 FuncCallFromDef 节点。首先，我们定义求值函数，并将其命名为 calc_expr_func(接受零参数并返回一个结果):

```
virtual void visit(FuncCallFromDef &Node) override {
    llvm::StringRef CalcExprFunName = "calc_expr_func";
    FunctionType *CalcExprFunTy = FunctionType::get(Int32Ty, {},
    false);
    Function *CalcExprFun = Function::Create(
        CalcExprFunTy, GlobalValue::ExternalLinkage,
        CalcExprFunName, M);
```

- 接下来，创建一个新的基本块，将 calc_expr_func 插入其中:

```
    BasicBlock *BB = BasicBlock::Create(M->getContext(),
    "entry", CalcExprFun);
    Builder.SetInsertPoint(BB);
```

- 与前面类似，用户定义函数由 genUserDefinedFunction() 检索，将函数调用的数值参数传递给重新生成的函数:

```
    llvm::StringRef CalleeFnName = Node.getFnName();
    Function *CalleeFn = genUserDefinedFunction(CalleeFnName);
```

- 当有了实际的 llvm::Function 实例，就可以利用 IRBuilder 创建一个对定义函数的调用，并返回结果，这样的输出结果，用户可以访问:

```
    auto CalleeFnVars = Node.getArgs();
    llvm::SmallVector<Value *> IntParams;
    for (unsigned i = 0, end = CalleeFnVars.size(); i != end;
    ++i) {
        int ArgsToIntType;
        CalleeFnVars[i].getAsInteger(10, ArgsToIntType);
        Value *IntParam = ConstantInt::get(Int32Ty, ArgsToIntType,
        true);
        IntParams.push_back(IntParam);
    }
    Builder.CreateRet(Builder.CreateCall(CalleeFn, IntParams,
    "calc_expr_res"));
};
```

9.4.3. 构建基于 LLJIT 的计算器

最后，为了编译 JIT 计算器源代码，还需要创建一个包含构建描述的 CMakeLists.txt 文件，保存在 Calc.cpp 和其他源文件附近:

1. 将最低所需的 CMake 版本设置为 LLVM 所需的版本，并给项目命名:

```
cmake_minimum_required (VERSION 3.20.0)
project ("jit")
```

2. 需要加载 LLVM 包，可将 LLVM 提供的 CMake 模块目录添加到搜索路径中，再包括 DetermineGCCCompatible 和 ChooseMSVCCRT 模块，其分别检查编译器是否具有 gcc 兼容的命令行语法，并确保使用与 LLVM 相同的 C 运行时:

```
find_package(LLVM REQUIRED CONFIG)
list(APPEND CMAKE_MODULE_PATH ${LLVM_DIR})
include(DetermineGCCCompatible)
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. 之后，若确定编译器具有 gcc 兼容的命令行语法，检查是否启用了运行时类型信息和异常处理。若未启用，则在编译中相应地添加用于关闭这些特性的 C++ 标志:

```
if(LLVM_COMPILER_IS_GCC_COMPATIBLE)
    if(NOT LLVM_ENABLE_RTTI)
        set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fno-rtti")
    endif()
    if(NOT LLVM_ENABLE_EH)
        set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fno-exceptions")
    endif()
endif()
```

5. 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 (calc
    Calc.cpp CodeGen.cpp Lexer.cpp Parser.cpp Sema.cpp)
target_link_libraries(calc PRIVATE ${llvm_libs})
```

前面的步骤是基于 JIT 的交互式计算器工具所需要的全部内容。接下来，创建并更改为构建目录，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
```

这将编译计算工具，然后可以启动计算器，开始定义函数，看看计算器是如何计算我们定义的函数。

下面的示例调用显示了首先定义的函数的 IR，然后创建 calc_expr_func 函数来生成对最初定义函数的调用，以便使用传递给的参数求值:

```
$ ./calc
JIT calc > def f(x) = x*2
define i32 @f(i32 %x) {
    entry:
    %0 = mul nsw i32 %x, 2
    ret i32 %0
}

JIT calc > f(20)
Attempting to evaluate expression:
define i32 @calc_expr_func() {
entry:
    %calc_expr_res = call i32 @f(i32 20)
    ret i32 %calc_expr_res
```

```
}

declare i32 @f(i32)
User defined function evaluated to: 40

JIT calc > def g(x,y) = x*y+100
define i32 @g(i32 %x, i32 %y) {
entry:
    %0 = mul nsw i32 %x, %y
    %1 = add nsw i32 %0, 100
    ret i32 %1
}

JIT calc > g(8,9)
Attempting to evaluate expression:
define i32 @calc_expr_func() {
entry:
    %calc_expr_res = call i32 @g(i32 8, i32 9)
    ret i32 %calc_expr_res
}

declare i32 @g(i32, i32)
User defined function evaluated to: 172
```

就是这样! 我们刚刚创建了一个基于 JIT 的计算器应用程序!

由于 JIT 计算器是一个简单的例子，描述了如何将 LLJIT 合并到我们的项目中，但它有一些限制:

- 计算器不支持十进制的负数
- 不能多次重新定义同名函数

第二个限制是设计上的，因此是 ORC API 本身所期望和强制的:

```
$ ./calc
JIT calc > def f(x) = x*2
define i32 @f(i32 %x) {
    entry:
    %0 = mul nsw i32 %x, 2
    ret i32 %0
}
JIT calc > def f(x,y) = x+y
define i32 @f(i32 %x, i32 %y) {
    entry:
    %0 = add nsw i32 %x, %y
    ret i32 %0
}
Duplicate definition of symbol '_f'
```

记住，除了为当前进程或动态库公开符号外，还有许多其他可能公开名称的方法。例如，StaticLibraryDefinitionGenerator 类公开在静态存档中找到的符号，并且可以在 DynamicLibrarySearchGenerator 类中使用。

此外，LLJIT 类还有一个 addObjectFile() 方法来公开对象文件的符号。若现有的实现不能满足需求，还可以提供自己的 DefinitionGenerator 实现。

使用预定义的 LLJIT 类很方便，但其会限制我们的灵活性。下一节中，我们将研究如何使用 ORC API 提供的层来实现 JIT 编译器。

9.5. 从头开始构建 JIT 编译器类

使用 ORC 的分层方法，很容易构建针对需求定制的 JIT 编译器。没有放之四海而皆准的 JIT 编译器，本章的第一节给出了一些示例。让我们看一下如何从头开始设置 JIT 编译器。

ORC API 使用堆叠在一起的层。最低层是对象链接层，由 llvm::orc::RTDyldObjectLinkingLayer 类表示，负责链接内存中的对象并将其转换为可执行代码。此任务所需的内存由 MemoryManager 接口的实例管理，有一个默认的实现，也可以使用自定义版本。

对象链接层之上是编译层，负责创建内存中的对象文件。llvm::orc::IRCompileLayer 类接受 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 类是接受 LLVM IR 的层实现的通用类。

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，该模块在内存中进行编译和链接。为了增加功能，可以在两层之上合并更多的层。

例如，CompileOnDemandLayer 类拆分一个模块，以便只编译请求的函数，这可以用来实现延迟编译，CompileOnDemandLayer 类也是 IRLayer 类的一个子类。IRTransformLayer 类，也是 IRLayer 类的一个子类，以一种非常通用的方式，允许对模块应用转换。

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。这个类管理 JITDylib 符号表，提供符号查找功能，并跟踪使用的资源管理器。

JIT 编译器的通用方式:

1. 初始化 ExecutionSession 类的实例。
2. 初始化层，至少由 RTDyldObjectLinkingLayer 类和 IRCompileLayer 类组成。
3. 创建第一个 JITDylib 符号表，通常使用 main 或类似的名称。

JIT 编译器的一般用法也非常简单:

1. 在符号表中添加一个 IR 模块。
2. 查找一个符号，触发相关函数的编译，也可能触发整个模块的编译。
3. 执行函数。

下一小节中，我们将按照通用方式实现 JIT 编译器类。

9.5.1. 创建 JIT 编译器类

为了保持 JIT 编译器类的实现简单，所有内容都放在 JIT.h 中，可以放在名为 JIT 的目录中。与使用 LLJIT 相比，类的初始化要复杂一些。由于要处理可能的错误，调用构造函数之前，需要一个

工厂方法来预先创建一些对象。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可多次包含:

```
#ifndef JIT_H
#define JIT_H
```

2. 首先，需要一些包含文件，其中的大多数都提供了一个与头文件同名的类。Core.h 头文件提供了几个基本类，包括 ExecutionSession 类。ExecutionUtils.h 头文件提供了 DynamicLibrarySearchGenerator 类来搜索库中的符号，CompileUtils.h 头文件提供了 ConcurrentIRCompiler 类:

```
#include "llvm/Analysis/AliasAnalysis.h"
#include "llvm/ExecutionEngine/JITSymbol.h"
#include "llvm/ExecutionEngine/Orc/CompileUtils.h"
#include "llvm/ExecutionEngine/Orc/Core.h"
#include "llvm/ExecutionEngine/Orc/ExecutionUtils.h"
#include "llvm/ExecutionEngine/Orc/IRCompileLayer.h"
#include "llvm/ExecutionEngine/Orc/IRTransformLayer.h"
#include "llvm/ExecutionEngine/Orc/JITTargetMachineBuilder.h"
#include "llvm/ExecutionEngine/Orc/Mangling.h"
#include "llvm/ExecutionEngine/Orc/RTDyldObjectLinkingLayer.h"
#include "llvm/ExecutionEngine/Orc/TargetProcessControl.h"
#include "llvm/ExecutionEngine/SectionMemoryManager.h"
#include "llvm/Passes/PassBuilder.h"
#include "llvm/Support/Error.h"
```

3. 声明一个新类。我们的新类称为 JIT:

```
class JIT {
```

4. 私有数据成员反映了 ORC 层和一些辅助类。ExecutionSession、ObjectLinkingLayer、CompileLayer、OptIRLayer 和 MainJITDylib 实例表示正在运行的 JIT 程序、层和符号表，TargetProcessControl 实例用于与 JIT 目标进程进行交互。这可以是同一进程，同一机器上的另一个进程，或者不同机器上的远程进程，可能具有不同的架构。DataLayout 和 MangleAndInterner 类需要以正确的方式修改符号的名称。符号名称可以内化，所以所有相等的名称具有相同的地址。所以检查两个符号名是否相等，比较地址就足够了，这个操作非常快:

```
    std::unique_ptr<llvm::orc::TargetProcessControl> TPC;
    std::unique_ptr<llvm::orc::ExecutionSession> ES;
    llvm::DataLayout DL;
    llvm::orc::MangleAndInterner Mangle;
    std::unique_ptr<llvm::orc::RTDyldObjectLinkingLayer>
        ObjectLinkingLayer;
    std::unique_ptr<llvm::orc::IRCompileLayer>
        CompileLayer;
    std::unique_ptr<llvm::orc::IRTransformLayer>
```

```
        OptIRLayer;
    llvm::orc::JITDylib &MainJITDylib;
```

5. 初始化分为三个部分。C++ 中，构造函数不能返回错误。简单且推荐的解决方案是创建一个静态工厂方法，可以在构造对象之前进行错误处理。层的初始化更为复杂，所以也为它们引入了工厂方法。

   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，并由多个类共享。为了控制当前流程，创建了一个 SelfTargetProcessControl 实例。若想要针对不同的流程，则需要更改此实例。

   接下来，构造一个 JITTargetMachineBuilder 实例，为此需要知道 JIT 进程的目标三元组，再向目标机器生成器查询数据布局。若构建器无法基于提供的三元组实例化目标机器，则此步骤可能会失败，例如：因为对此目标的支持没有编译到 LLVM 库中:

```
public:
    static llvm::Expected<std::unique_ptr<JIT>> create() {
        auto SSP =
            std::make_shared<llvm::orc::SymbolStringPool>();
        auto TPC =
            llvm::orc::SelfTargetProcessControl::Create(SSP);
        if (!TPC)
            return TPC.takeError();
        llvm::orc::JITTargetMachineBuilder JTMB(
            (*TPC)->getTargetTriple());
        auto DL = JTMB.getDefaultDataLayoutForTarget();
        if (!DL)
            return DL.takeError();
```

6. 此时，已经处理了所有可能失败的调用，现在可以初始化 ExecutionSession 实例了。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:

```
        auto ES =
            std::make_unique<llvm::orc::ExecutionSession>(
                std::move(SSP));

        return std::make_unique<JIT>(
            std::move(*TPC), std::move(ES), std::move(*DL),
            std::move(JTMB));
    }
```

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。此外，每个层工厂方法都需要对 ExecutionSession 实例的引用，该实例将层连接到正在运行的 JIT 会话。除了位于层堆栈底部的对象链接层外，每一层都需要引用前一层，说明了堆叠顺序:

```
JIT(std::unique_ptr<llvm::orc::ExecutorProcessControl>
            EPCtrl,
    std::unique_ptr<llvm::orc::ExecutionSession>
        ExeS,
```

```
    llvm::DataLayout DataL,
    llvm::orc::JITTargetMachineBuilder JTMB)
    : EPC(std::move(EPCtrl)), ES(std::move(ExeS)),
        DL(std::move(DataL)), Mangle(*ES, DL),
        ObjectLinkingLayer(std::move(
            createObjectLinkingLayer(*ES, JTMB))),
        CompileLayer(std::move(createCompileLayer(
            *ES, *ObjectLinkingLayer,
            std::move(JTMB)))),
        OptIRLayer(std::move(
            createOptIRLayer(*ES, *CompileLayer))),
        MainJITDylib(
            ES->createBareJITDylib("<main>")) {
```

8. 构造函数中，添加了一个生成器，用于在当前进程中搜索符号。GetForCurrentProcess() 方法很特殊，返回值包装在一个 Expected<> 模板中，表明也可以返回一个 Error 对象。由于我们知道不会发生错误，所以当前进程最终将运行! 使用 cantFail() 函数展开结果，若发生错误，该函数将终止应用程序:

```
    MainJITDylib.addGenerator(llvm::cantFail(
        llvm::orc::DynamicLibrarySearchGenerator::
            GetForCurrentProcess(DL.getGlobalPrefix())));
}
```

9. 要创建一个对象链接层，需要提供一个内存管理器。这里，我们坚持使用默认的 SectionMemoryManager 类，也可以提供不同的实现:

```
    static std::unique_ptr<
        llvm::orc::RTDyldObjectLinkingLayer>
    createObjectLinkingLayer(
        llvm::orc::ExecutionSession &ES,
        llvm::orc::JITTargetMachineBuilder &JTMB) {
    auto GetMemoryManager = []() {
        return std::make_unique<
            llvm::SectionMemoryManager>();
    };
    auto OLLayer = std::make_unique<
        llvm::orc::RTDyldObjectLinkingLayer>(
        ES, GetMemoryManager);
```

10. Windows 上使用的通用对象文件格式 (COFF) 对象文件格式存在轻微的复杂性，此文件格式不允许将函数标记为导出。这随后导致对象链接层内部检查失败：将符号中存储的标志与 IR 中的标志进行比较，由于缺少导出标记，导致不匹配，解决方案是只覆盖这种文件格式的标志。这就完成了对象层的构造，对象会返回给调用者:

```
  if (JTMB.getTargetTriple().isOSBinFormatCOFF()) {
      OLLayer
          ->setOverrideObjectFlagsWithResponsibilityFlags(
              true);
```

```
        OLLayer
            ->setAutoClaimResponsibilityForObjectSymbols(
                true);
    }
    return OLLayer;
}
```

11. 要初始化编译器层，需要一个 IRCompiler 实例，IRCompiler 实例负责将一个 IR 模块编译成一个 object 文件。若 JIT 编译器不使用线程，则可以使用 SimpleCompiler 类，使用给定的目标机器编译 IR 模块。TargetMachine 类线程不安全，因此 SimpleCompiler 类也不是。为了支持多线程编译，使用 ConcurrentIRCompiler 类，为每个要编译的模块创建一个新的 TargetMachine 实例。这种方法解决了多线程的问题:

```
    static std::unique_ptr<llvm::orc::IRCompileLayer>
    createCompileLayer(
            llvm::orc::ExecutionSession &ES,
            llvm::orc::RTDyldObjectLinkingLayer &OLLayer,
            llvm::orc::JITTargetMachineBuilder JTMB) {
        auto IRCompiler = std::make_unique<
            llvm::orc::ConcurrentIRCompiler>(
            std::move(JTMB));
        auto IRCLayer =
            std::make_unique<llvm::orc::IRCompileLayer>(
                ES, OLLayer, std::move(IRCompiler));
        return IRCLayer;
    }
```

12. 我们没有将 IR 模块直接编译为机器码，而是安装了一个层来首先优化 IR。这是一个深思熟虑的设计决策: 将 JIT 编译器转换为优化的 JIT 编译器，生成更快的代码，而生成代码需要更长的时间，所以用户需要等待。我们没有添加延迟编译，所以当只查找一个符号时，整个模块都会编译。用户看到代码执行之前，可能会等待很久。

> **Note**
>
> 所有情况下，引入惰性编译并不是一个合适的解决方案。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。防止了诸如内联之类的过程间优化，因为内联传递需要访问调用函数的体才能内联。用户看到延迟编译的启动速度更快，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，所以优化层本质上是一个转换层。

IRTransformLayer 类将转换委托给函数——我们的例子中，委托给了 optimizeModule 函数:

```
    static std::unique_ptr<llvm::orc::IRTransformLayer>
    createOptIRLayer(
            llvm::orc::ExecutionSession &ES,
            llvm::orc::IRCompileLayer &CompileLayer) {
        auto OptIRLayer =
        std::make_unique<llvm::orc::IRTransformLayer>(
```

```
            ES, CompileLayer,
            optimizeModule);
        return OptIRLayer;
    }
```

13. optimizeModule() 函数是 IR 模块转换的一个示例。该函数获取要转换的模块作为参数，并返回转换后的 IR 模块版本。由于 JIT 编译器可能运行多个线程，所以 IR 模块可封装在 ThreadSafeModule 实例中:

```
    static llvm::Expected<llvm::orc::ThreadSafeModule>
    optimizeModule(
        llvm::orc::ThreadSafeModule TSM,
        const llvm::orc::MaterializationResponsibility
        &R) {
```

14. 为了优化 IR，回顾以下第 7 章，需要 PassBuilder 实例来创建优化流水线。首先，定义两个分析管理器，然后在通道构建器中注册，再用 O2 级别的默认优化流水线填充 ModulePassManager 实例。这又是一个设计决策:O2 级别生成的机器码已经很快了，但 O3 级别生成的代码甚至更快。接下来，我们在模块上运行流水线。最后，优化后的模块返回给调用者:

```
    TSM.withModuleDo([](llvm::Module &M) {
        bool DebugPM = false;
        llvm::PassBuilder PB(DebugPM);
        llvm::LoopAnalysisManager LAM(DebugPM);
        llvm::FunctionAnalysisManager FAM(DebugPM);
        llvm::CGSCCAnalysisManager CGAM(DebugPM);
        llvm::ModuleAnalysisManager MAM(DebugPM);
        FAM.registerPass(
            [&] { return PB.buildDefaultAAPipeline(); });
        PB.registerModuleAnalyses(MAM);
        PB.registerCGSCCAnalyses(CGAM);
        PB.registerFunctionAnalyses(FAM);
        PB.registerLoopAnalyses(LAM);
        PB.crossRegisterProxies(LAM, FAM, CGAM, MAM);
        llvm::ModulePassManager MPM =
            PB.buildPerModuleDefaultPipeline(
                llvm::PassBuilder::OptimizationLevel::O2,
                DebugPM);
        MPM.run(M, MAM);
    });

    return TSM;
}
```

15. JIT 类的客户机需要一种添加 IR 模块的方法，通过 addIRModule() 函数提供了这种方法。回想一下我们创建的层堆栈: 必须将 IR 模块添加到顶层; 否则，会不绕过一些层。这是一个不容易发现的编程错误: 若 OptIRLayer 成员用 CompileLayer 成员取代，JIT 类仍然工作，但不是作为优化 JIT，因为已经绕过了这一层。对于这个小实现来说无关紧要，但是在一个大 JIT 优化中，

我们会引入一个函数来返回顶层:

```
    llvm::Error addIRModule(
            llvm::orc::ThreadSafeModule TSM,
            llvm::orc::ResourceTrackerSP RT = nullptr) {
        if (!RT)
            RT = MainJITDylib.getDefaultResourceTracker();
        return OptIRLayer->add(RT, std::move(TSM));
    }
```

16. 同样，JIT 类的客户机需要一种查找符号的方法。我们把它委托给 ExecutionSession 实例，传递一个对主符号表的引用和请求符号的内部名称:

```
    llvm::Expected<llvm::orc::ExecutorSymbolDef>
    lookup(llvm::StringRef Name) {
        return ES->lookup({&MainJITDylib},
                          Mangle(Name.str()));
    }
```

这个 JIT 类的初始化可能很棘手，它涉及到 JIT 类的工厂方法和构造函数调用，以及每个层的工厂方法。尽管这个发行版是由 C++ 的限制造成的，但代码本身很简单。

接下来，将使用新的 JIT 编译器类来实现一个简单的命令行实用程序，该实用程序将 LLVM IR 文件作为输入。

9.5.2. 使用新的 JIT 编译器类

首先在与 JIT.h 文件相同的目录下创建一个名为 JIT.cpp 的文件，并在此源文件中添加以下内容:

1. 首先，包含几个头文件。必须包含 JIT.h 来使用我们的新类，还必须包含 IRReader.h 头文件，定义了一个读取 LLVM IR 文件的函数，CommandLine.h 头文件允许我们以 LLVM 风格解析命令行选项，还需要 InitLLVM.h 进行工具的基本初始化。最后，TargetSelect.h 需要用于初始化本机目标:

```
#include "JIT.h"
#include "llvm/IRReader/IRReader.h"
#include "llvm/Support/CommandLine.h"
#include "llvm/Support/InitLLVM.h"
#include "llvm/Support/TargetSelect.h"
```

2. 接下来，我们将 llvm 命名空间添加到当前作用域:

```
using namespace llvm;
```

3. JIT 工具在命令行上只需要一个输入文件，用 cl::opt<> 类声明它:

```
static cl::opt<std::string>
    InputFile(cl::Positional, cl::Required,
        cl::desc("<input-file>"));
```

4. 为了读取 IR 文件，调用 parseIRFile() 函数，该文件可以是文本 IR 表示或位码文件。该函数返回一个指向所创建模块的指针。因为可以解析文本 IR 文件，错误处理有点不同，这在语法上不一定正确。最后，SMDiagnostic 实例保存语法错误时的错误信息。若发生错误，则输出错误消息，并退出应用程序:

```
std::unique_ptr<Module>
loadModule(StringRef Filename, LLVMContext &Ctx,
           const char *ProgName) {
    SMDiagnostic Err;
    std::unique_ptr<Module> Mod =
        parseIRFile(Filename, Err, Ctx);
    if (!Mod.get()) {
        Err.print(ProgName, errs());
        exit(-1);
    }
    return Mod;
}
```

5. jitmain() 函数放在 loadModule() 方法之后，函数设置我们的 JIT 引擎并编译一个 LLVM IR 模块。该函数需要带有 IR 的 LLVM 模块来执行。这个模块还需要 LLVM 上下文类，上下文类包含重要的类型信息。因为我们的目标是调用 main() 函数，所以也传递了常用的 argc 和 argv 参数:

```
Error jitmain(std::unique_ptr<Module> M,
    std::unique_ptr<LLVMContext> Ctx,
    int argc, char *argv[]) {
```

6. 接下来，创建前面构造的 JIT 类的一个实例。若发生错误，则返回相应的错误消息:

```
auto JIT = JIT::create();
if (!JIT)
    return JIT.takeError();
```

7. 将模块添加到主 JITDylib 实例中，再次将模块和上下文包装在 ThreadSafeModule 实例中。若发生错误，则返回一条错误消息:

```
if (auto Err = (*JIT)->addIRModule(
        orc::ThreadSafeModule(std::move(M),
                              std::move(Ctx))))
    return Err;
```

8. 查找主符号，此符号必须在命令行上给出的 IR 模块中。查找触发该 IR 模块的编译。若在 IR 模块中引用了其他符号，则使用上一步中添加的生成器进行解析。结果是 ExecutorAddr 类，表示执行进程的地址:

```
    llvm::orc::ExecutorAddr MainExecutorAddr = MainSym->getAddress();
    auto *Main = MainExecutorAddr.toPtr<int(int, char**)>();
```

9. 可以调用 IR 模块中的 main() 函数，并传递函数期望的 argc 和 argv 参数:

```
    (void)Main(argc, argv);
```

10. 函数执行成功后进行报告:

```
  return Error::success();
}
```

11. 实现了 jitmain() 函数之后，添加了 main() 函数，其初始化工具和本机目标并解析命令行:

```
int main(int argc, char *argv[]) {
    InitLLVM X(argc, argv);
    InitializeNativeTarget();
    InitializeNativeTargetAsmPrinter();
    InitializeNativeTargetAsmParser();

    cl::ParseCommandLineOptions(argc, argv, "JIT\n");
```

12. 初始化 LLVM 上下文类，并加载命令行中指定的 IR 模块:

```
    auto Ctx = std::make_unique<LLVMContext>();
    std::unique_ptr<Module> M =
        loadModule(InputFile, *Ctx, argv[0]);
```

13. 加载 IR 模块后，可以调用 jitmain() 函数。为了处理错误，使用 ExitOnError 实用程序类来打印错误消息，并在遇到错误时退出应用程序。我们还设置了一个带有应用程序名称，其输出在错误消息之前:

```
    ExitOnError ExitOnErr(std::string(argv[0]) + ": ");
    ExitOnErr(jitmain(std::move(M), std::move(Ctx),
                      argc, argv));
```

14. 若控制流达到这一点，则 IR 已成功执行。返回 0 表示成功:

```
    return 0;
}
```

现在，可以通过编译一个简单的示例来测试新实现的 JIT 编译器，该示例输出 Hello World! 到控制台。底层新类使用一个固定的优化级别，因此对于足够大的模块，可以注意到启动和运行时的差异。

要构建 JIT 编译器，可以按照在用 LLJIT 实现自己的 JIT 编译器一节末尾所做的相同的 CMake 步骤，只需要确保 JIT.cpp 源文件正在用正确的库进行编译。

```
add_executable(JIT JIT.cpp)
include_directories(${CMAKE_SOURCE_DIR})
target_link_libraries(JIT ${llvm_libs})
```

再切换到 build 目录并编译应用程序:

```
$ cmake -G Ninja <path to jit source directory>
$ ninja
```

我们的 JIT 工具现在可以使用了。一个简单的 Hello World! 程序可以用 C 编写，如下所示:

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

int main(int argc, char** argv) {
    printf("Hello world!\n");
    return 0;
}
```

可以用下面的命令将 Hello World C 源代码编译成 LLVM IR:

```
$ clang -S -emit-llvm main.c
```

记住——我们将 C 源代码编译成 LLVM IR，因为 JIT 编译器接受 IR 文件作为输入。最后，可以用 IR 示例调用 JIT 编译器:

```
$ JIT main.ll
Hello world!
```

9.6. 总结

本章中，了解了如何开发 JIT 编译器。首先了解了 JIT 编译器的可能应用程序，探索了 LLVM 动态编译器和解释器。使用预定义的 LLJIT 类，构建了一个交互式的基于 jit 的计算器工具，并了解了如何查找符号和向 LLJIT 添加 IR 模块。为了能够利用 ORC API 的分层结构，还实现了一个优化的 JIT 类。

下一章中，将了解如何利用 LLVM 工具进行调试。

第 10 章 使用 LLVM 工具进行调试

LLVM 有一组工具，可以发现应用程序中的某些错误，所有这些工具都使用了 LLVM 和 clang 库。

本章中，料及如何使用消毒工具来检查应用程序，如何使用最常见的消毒工具来识别各种各样的 bug，之后为应用程序实现模糊测试，这将检查通常在单元测试中找不到的 bug。还将了解如何识别应用程序中的性能瓶颈，运行静态分析器来识别编译器通常无法发现的问题，并创建基于 clang 的工具，在该工具中可以使用新功能扩展 clang。

本章中，将了解以下内容:

- 用消毒程序检测应用程序
- 使用 libFuzzer 查找 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，将了解如何使用各种 LLVM 和 clang 工具来识别应用程序中的大量错误。还将获得使用新功能扩展 clang 的知识，例如：加强命名约定或添加新的源代码分析。

10.1. 环境要求

要在 XRay 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，比如 Fedora 和 FreeBSD，为这些脚本提供了一个包，也可以使用。

若要在同一区域查看 Chrome 浏览器的可视化效果，则需要安装 Chrome 浏览器。Chrome 浏览器可以从https://www.google.com/chrome/网站下载，也可以使用系统自带的软件包管理器安装。

此外，要通过扫描-构建脚本运行静态分析器，需要在 Fedora 和 Ubuntu 上安装 perlcore 包。

10.2. 用消毒器检测应用程序

LLVM 具有几个消毒器，这些通道使用中间表示 (IR) 来检查应用程序的某些错误行为。通常，它们需要库支持，这是 compiler-rt 项目的一部分。消毒器可以在 clang 中启用，要构建编译器-rt 项目，可以简单地在构建 LLVM 时将 CMake 变量-DLLVM_ENABLE_RUNTIMES=compiler-rt 添加到 CMake 配置步骤中。

下面几节中，将研究地址、内存和线程消毒器。首先，来看看地址消毒器。

10.2.1. 使用地址消毒器检测内存访问问题

可以使用地址杀毒器，来检测应用程序中不同类型的内存访问错误。这包括常见的错误，例如在释放动态分配的内存后使用，或者在已分配内存的边界外写入动态分配的内存。

启用后，地址消毒器会替换对 malloc() 和 free() 函数的调用，并使用检查保护来检测所有内存访问。这会给应用程序增加很多开销，而且只会在应用程序的测试阶段使用地址消毒器。若对实

现细节感兴趣，可以在 llvm/lib/Transforms/Instrumentation/AddressSanitzer.cpp 中为通道的实现源码，实现算法的描述见https://github.com/google/sanitizers/wiki/AddressSanitizerAlgorithm。

让我们运行一个简短的示例来展示地址消毒器的功能!

下面的示例应用程序 outfbounds.c 分配了 12 个字节的内存，但初始化了 14 个字节:

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

int main(int argc, char *argv[]) {
    char *p = malloc(12);
    memset(p, 0, 14);
    return (int)*p;
}
```

可以编译并运行此应用程序而不会发现任何问题，因为这种行为是此类错误的典型表现。即使在较大的应用程序中，这类错误也可能在很长一段时间内被忽视。但若使用-fsanitize=address 选项启用地址消毒器，则应用程序在检测到错误后停止。

使用-g 选项启用调试符号也很有用，它有助于识别源代码中错误的位置。以下代码是如何在启用地址消毒器和调试符号的情况下编译源文件的示例:

```
$ clang -fsanitize=address -g outofbounds.c -o outofbounds
```

现在，当运行应用程序时，将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 0x7fffffffeb10 sp
0x7fffffffe2d8
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
```

该报告还包含有关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) 和出错的源代码行。要找到源代码行，必须查看位置 #1 处的堆栈跟踪，这是地址消毒器拦截应用程序执行之前的最后一个位置——outfbounds.c 文件中的第 6 行，这是包含对 memset() 调用的行，这就是缓冲区溢出发生的确切位置。

若替换包含 memset(p, 0, 14); 在带有以下代码的 outfbounds.c 文件中，当释放了内存，就可以引入对内存的访问，需要将源代码存储在 useafterfree.c 文件中:

```
memset(p, 0, 12);
free(p);
```

同样，若编译并运行它，消毒器会在释放内存后检测指针的使用:

```
$ clang -fsanitize=address -g useafterfree.c -o useafterfree
$ ./useafterfree
=================================================================
==1118==ERROR: AddressSanitizer: heap-use-after-free on address
0x602000000010 at pc 0x0000002b2a5c bp 0x7fffffffeb00 sp
0x7fffffffeaf8
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
```

这次，报告指向第 8 行，其中包含对 p 指针的解引用。

x86_64 Linux 和 macOS 上，还可以启用泄漏检测，若将 ASAN_OPTIONS 环境变量设置为在运行应用程序之前 detect_leaks=1，还将获得关于内存泄漏的报告。

命令行中，可以这样做:

```
$ ASAN_OPTIONS=detect_leaks=1 ./useafterfree
```

地址消毒器非常有用，它捕获了其他方法很难检测到的一类错误。内存消毒器执行类似的任务，我们将在下一节中研究它的用例。

10.2.2. 使用内存消毒器查找未初始化的内存访问

使用未初始化的内存是另一类很难发现的 bug。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，对于堆栈上的自动变量也是如此。

有很多出错的机会，内存消毒器可以帮助找到这些错误。若对实现细节感兴趣，可以在 llvm/lib/Transforms/Instrumentation/MemorySanitizer.cpp 文件中找到内存消毒通道的源码。文件顶部的注释解释了实现背后的思想。

让我们运行一个小示例，并将以下源代码保存为 memory.c 文件。注意，变量 x 没有初始化，而是用作返回值:

```
int main(int argc, char *argv[]) {
    int x;
    return x;
}
```

没启用杀菌器时，应用程序将运行良好。若使用-fsanitize=memory 选项，将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
```

与地址消毒器一样，内存消毒器在发现第一个错误时停止应用程序。如图所示，内存杀毒器提供了一个使用初始化值的警告。

下一节中，我们将研究如何使用线程消毒器来检测多线程应用程序中的数据争用。

10.2.3. 使用线程消毒器发现数据竞争

为了利用现代 CPU 的强大功能，应用程序现在使用多线程。这是一种强大的技术，但也引入了新的错误。多线程应用程序中一个非常常见的问题是，对全局数据的访问没有受到保护，例如：使用互斥锁或信号量。

这就是数据竞争。线程消毒器可以检测基于 pthread 的应用程序和使用 LLVM lib++ 实现的应用程序中的数据竞争，可以在 llvm/lib/Transforms/Instrumentation/ThreadSanitizer.cpp 文件中找到实现。为了演示线程消毒器的功能，将创建一个非常简单的生产者-消费者风格的应用程序。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，因此这是一种数据竞争。

需要在 thread.c 文件中书写以下源代码:

```c
#include <pthread.h>

int data = 0;

void *producer(void *x) {
    for (int i = 0; i < 10000; ++i) ++data;
    return x;
}

void *consumer(void *x) {
    for (int i = 0; i < 10000; ++i) --data;
    return x;
}

int main() {
    pthread_t t1, t2;
    pthread_create(&t1, NULL, producer, NULL);
    pthread_create(&t2, NULL, consumer, NULL);
    pthread_join(t1, NULL);
    pthread_join(t2, NULL);
    return data;
}
```

前面的代码中，data 变量在两个线程之间共享。为了使示例简单，它是 int 类型，通常会使用 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() 函数则减少数据变量。没有实现访问保护，因此这构成了数据竞争。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 data 变量的当前值。

若编译并运行此应用程序，则不会注意到错误——返回值始终为零。若执行的循环次数增加 100 倍，将出现一个错误——在本例中，返回值不等于 0。此时，需要注意出现的其他值。

可以使用线程消毒器来识别程序中的数据争用。要在启用线程消毒器的情况下进行编译，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显示行号，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，在这里访问全局变量。它还显示了名为 T1 和 T2 的两个线程，访问了变量以及分别调用 pthread_create() 函数的文件和行号。

在此基础上，我们了解了如何使用三种不同类型的消毒器来识别应用程序中的常见问题。地址消毒器可识别常见的内存访问错误，例如：越界访问或在内存被释放后使用内存。使用内存消毒器，可以找到对未初始化内存的访问，线程消毒器可检查数据竞争。

下一节中，我们将尝试通过在随机数据上运行应用程序来触发消毒器，这个过程称为模糊测试。

10.3. 使用 libFuzzer 查找 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，由于可能输入的指数数量，可能会错过某些奇怪的输入，以及一些错误。

模糊测试在这方面可以提供帮助。其思想是为应用程序提供随机生成的数据，或者基于有效输入但随机更改的数据。这是重复进行的，因此应用程序将使用大量输入进行测试，这就是为什么模糊测试是一种强大的测试方法。人们注意到，模糊测试已经帮助发现了网页浏览器和其他软件中的数百个 bug。

有趣的是，LLVM 自带了自己的模糊测试库。libFuzzer 最初是 LLVM 核心库的一部分，最终移到了 compiler-rt 中，所以该库设计用于测试小而快速的函数。

运行一个小示例，看看 libFuzzer 是如何工作的。首先，需要提供 LLVMFuzzerTestOneInput() 函数。该函数由 fuzzer 驱动程序调用，并提供一些输入。下面的函数对输入中的连续 ASCII 数字进行计数。但它完成了，就会给它随机输入。

将示例保存在 fuzzer.c 文件中:

```
#include <stdint.h>
#include <stdlib.h>
int count(const uint8_t *Data, size_t Size) {
    int cnt = 0;
    if (Size)
        while (Data[cnt] >= '0' && Data[cnt] <= '9') ++cnt;
    return cnt;
}

int LLVMFuzzerTestOneInput(const uint8_t *Data, size_t Size) {
    count(Data, Size);
    return 0;
}
```

前面的代码中，count() 函数对 Data 变量所指向的内存中的位数进行计数，检查数据的大小只是为了确定是否有可用的字节。在 while 循环中，不检查大小。

与普通的 C 字符串一起使用，不会有错误，因为 C 字符串总是以 0 字节结束。LLVMFuzzerTestOneInput() 函数是所谓的模糊目标，是 libFuzzer 调用的函数。会调用要测试的函数并返回 0，这是当前唯一允许的值。

要用 libFuzzer 编译文件，必须添加-fsanitize=fuzzer 选项。建议还启用地址消毒器和生成调试符号。我们可以使用下面的命令来编译 fuzzer.c 文件:

```
$ clang -fsanitize=fuzzer,address -g fuzzer.c -o fuzzer
```

当运行测试时，就会发出一个冗长的报告。该报告包含比堆栈跟踪更多的信息，所以需要仔细看一下:

- 第一行说明用于初始化随机数生成器的种子，可以使用-seed= 选项来重复执行:

```
INFO: Seed: 1297394926
```

- 默认情况下，libFuzzer 将输入限制为最多 4096 字节，可以使用-max_len= 选项来改变默认值:

```
INFO: -max_len is not provided; libFuzzer will not generate
inputs larger than 4096 bytes
```

- 现在，可以在不提供样本输入的情况下运行测试。所有样本输入的集合称为 corpus，在这次运行时它是空的:

```
INFO: A corpus is not provided, starting from an empty corpus
```

- 下面是关于生成的测试数据的一些信息：尝试了 28 个输入，找到了 6 个输入，它们加在一起长度为 19 字节，覆盖了 6 个覆盖点或基本块:

```
#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"-
```

- 在此之后，检测到缓冲区溢出，并遵循来自地址消毒程序的信息。最后，报告说明导致缓冲区溢出的输入保存在哪里:

```
artifact_prefix='./'; Test unit written to
↪  ./crash-17ba0791499db908433b80f37c5fbc89b870084b
```

有了保存的输入，测试用例可以用同样的崩溃输入再次执行:

```
$ ./fuzzer crash-17ba0791499db908433b80f37c5fbc89b870084b
```

这有助于识别问题，可以使用保存的输入作来复现和修复可能出现的问题。然而，仅使用随机数据并不是在所有情况下都很有帮助。若尝试对 tinylang 词法分析器或解析器进行模糊测试，因为找不到有效的标记，将导致纯随机数据会立即拒绝。

提供一小部分有效的输入 (称为语料库) 更有用，语料库的文件可随机选作为输入。可以认为输入大部分是有效的，只是翻转了几个比特。这也适用于其他必须具有特定格式的输入。例如，对于一个处理 JPEG 和 PNG 文件的库，可以提供一些小的 JPEG 和 PNG 文件作为语料库。

提供语料库的示例如下所示。可以将语料库文件保存在一个或多个目录中，可以在 printf 命令的帮助下为模糊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
```

运行测试时，必须在命令行上提供目录:

```
$ ./fuzzer corpus/
```

然后，语料库用作生成随机输入的基础，正如报告所示的那样:

```
INFO: seed corpus: files: 2 min: 4b max: 7b total: 11b rss: 29Mb
```

此外，若正在测试一个处理标记或其他神奇值 (如编程语言) 的函数，则可以通过提供一个带有标记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"
```

这个键可选，可以忽略它。现在，可以使用-dict = 选项在命令行上指定字典文件。

既然已经介绍了使用 libFuzzer 查找错误，来看看 libFuzzer 的限制和替代方案。

10.3.1. 限制和替代方案

libFuzzer 实现速度很快，但对测试目标提出了一些限制:

- 测试函数必须在内存中接受作为数组的输入。有些库函数需要数据的文件路径，无法使用 libFuzzer 进行测试。
- 不能调用 exit() 函数。
- 全球状态不应改变。
- 不应使用硬件随机数生成器。

前两个限制是 libFuzzer 作为库实现的暗示。为了避免评估算法中的混淆，需要后两个限制。若不满足这些限制中的一个，会对模糊目标的两个相同调用可能会产生不同的结果。

最著名的模糊测试替代工具是 AFL，可以在https://github.com/google/AFL上找到。AFL 需要一个仪表化的二进制文件 (提供了用于仪表化的 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相同的字典文件，所以可以使用这两种工具测试应用程序。此外，在 libFuzzer 不适用的地方，AFL 可能是一个很好的选择。

还有很多方法可以影响 libFuzzer 的工作方式，可以阅读参考页面https://llvm.org/docs/LibFuzzer.html了解更多细节。

下一节中，我们将研究应用程序可能遇到的另一个问题——使用 XRay 工具检测性能瓶颈。

10.4. 使用 XRay 进行性能分析

若应用程序似乎运行缓慢，可能想知道代码中的时间花在了哪里，可以用 XRay 检测代码可以帮助完成这项任务。每个函数进入和退出时，都会向运行时库插入一个特殊的调用。允许计算函数调用的频率，以及在函数中花费的时间。可以在 llvm/lib/XRay/目录中找到检测通道的实现，其运行时是 compiler-rt 的一部分。

下面的示例源代码中，使用 ussleep() 函数模拟实际工作。func1() 函数休眠 10µs，func2() 函数调用 func1() 或休眠 100µs，取决于 n 参数是奇数还是偶数。在 main() 函数中，两个函数都在循环中调用。需要在 xraydemo.c 文件中保存以下源码:

```
#include <unistd.h>

void func1() { usleep(10); }

```

```
void func2(int n) {
    if (n % 2) func1();
    else usleep(100);
}

int main(int argc, char *argv[]) {
    for (int i = 0; i < 100; i++) { func1(); func2(i); }
    return 0;
}
```

要在编译期间启用 XRay 检测，需要指定-fxrayinstrument 选项，指令少于 200 条的函数不会检测。这是因为这是开发人员定义的任意阈值。我们的例子中，函数不会检测。该阈值可以通过-fxrayinstruction-threshold= 选项指定。

或者，可以添加一个 function 属性来控制是否应该检测一个函数，添加以下原型将总是检测函数:

```
void func1() __attribute__((xray_always_instrument));
```

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并编译 xraydemo.c 文件:

```
$ clang -fxray-instrument -fxray-instruction-threshold=1 -g xraydemo.c -o xraydemo
```

生成的二进制文件中，检测在默认情况下关闭。若运行二进制文件，将注意到与未检测的二进制文件相比没有区别。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，可以运行如下应用程序:

```
$ 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。这种模式下，将收集所有运行时数据，这可能导致生成较大的日志文件。当给出 patch_premain=true 选项时，还会检测在 main() 函数之前运行的函数。

执行该命令后，采集数据所在目录下将创建一个新文件。需要使用 llvm-xray 工具从该文件中提取可读信息。

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，要获得调用次数最多的前 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，还可以使用-sortorder= 选项影响排序顺序。

可以运行以下命令来从程序中获取统计信息:

```
$ 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
```

func1() 函数调用的次数最多，还可以看到在此函数中花费的累积时间。这个例子只有三个函数，所以-top= 选项在这里没有明显的效果，但是对于实际的应用程序，是非常有用的。

从收集的数据中，可以重建运行时期间发生的所有堆栈帧，使用 stack 子命令查看 top 10 的堆叠信息。为了简洁起见，这里简化了显示的输出:

```
$ 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
```

堆栈帧是函数如何调用的序列。func2() 函数由 main() 函数调用，这是累积时间最长的堆栈帧。深度取决于调用了多少函数，堆栈帧通常很大。

此子命令还可以用于从堆栈帧创建火焰图，可以轻松地确定哪些函数具有较大的累积运行时。输出是带有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伸缩的矢量图形 (SVG) 文件，并可以在浏览器中查看该文件。

使用以下命令，指示 llvm-xray 使用-all-stacks 选项输出所有堆栈帧。使用-stack-format =flame 选项，输出将是 flamegraph.pl 脚本所期望的格式。使用-aggregate-type 选项，可以选择是按总时间统计堆栈帧，还是按调用次数统计堆栈帧。llvm-xray 的输出通过管道输入到 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
```

运行命令并生成新的火焰图后，可以在浏览器中打开生成的 flame.svg 文件。图表如下:

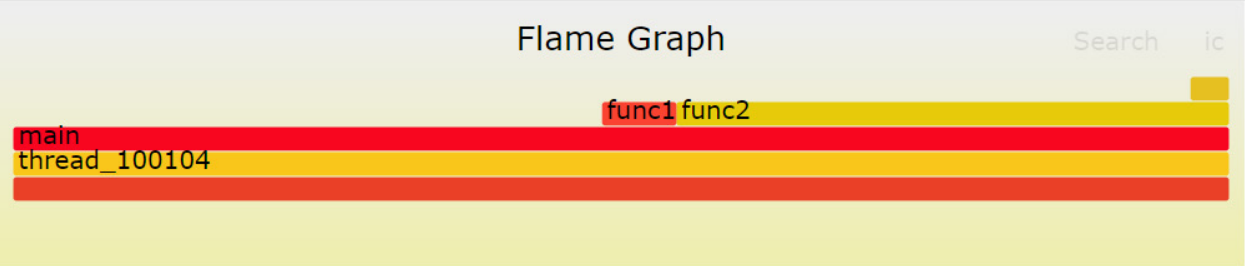


图 10.1 - llvm-xray 生成的火焰图

火焰图乍一看可能会令人困惑，因为 x 轴没有通常的时间流逝含义。相反，函数只是按名称的字母顺序排序。此外，火焰图的 y 轴显示堆栈深度，底部从零开始计数。颜色的选择要有很好的对比，没有别的意思。从前面的图中，可以很容易地确定调用层次结构和在函数中花费的时间。

只有将鼠标移动到表示堆叠帧的矩形上时，才会显示堆叠帧的相关信息。通过单击该框架，可以放大该堆栈框架。若想确定值得优化的函数，火焰图会有很大的帮助。要了解更多关于火焰图的知识，请访问火焰图的发明者 Brendan Gregg 的主页:http://www.brendangregg.com/flamegraphs.html。

此外，可以使用转换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 文件后，执行如下命令:

```
$ llvm-xray convert --output-format=trace_event\
    --output=xray.evt --symbolize --sort\
    --instr_map=./xraydemo xray-log.xraydemo.xVsWiE
```

若不指定-symbol 选项，则结果图中不会显示函数名。

完成后，打开 Chrome 并输入 chrome:///tracing。接下来，单击 Load 按钮来加载 xray.evt 文件。将看到以下可视化数据:

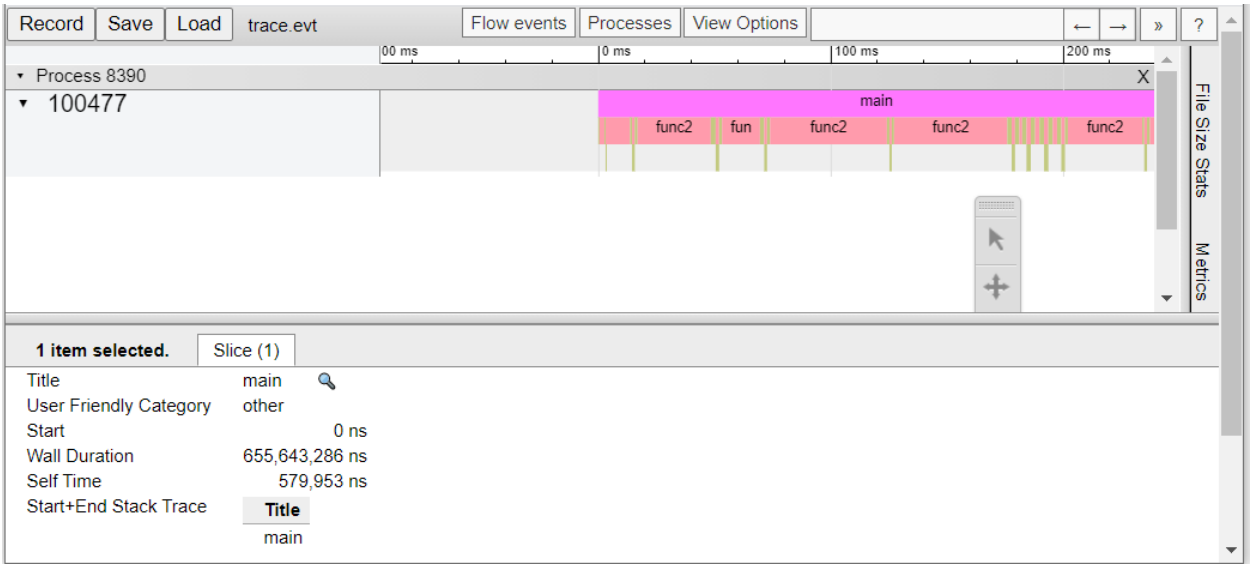


图 10.2 - 由 llvm-xray 生成的可视化 Chrome 跟踪查看器

在此视图中，堆栈帧按函数调用发生的时间排序。要进一步了解可视化，请阅读https://www.chromium.org/developers/how-tos/trace-event-profiling-tool上的教程。

Tip

llvm-xray 工具具有更多适用于性能分析的功能。可以在 LLVM 网站https://llvm.org/docs/XRay.html和https://llvm.org/docs/XRayExample.html上阅读相关内容。

本节中，了解如何使用 XRay 检测应用程序，如何收集运行时信息，以及如何可视化该数据。可以利用这些知识，找到应用程序中的性能瓶颈。

识别应用程序中错误的另一种方法是分析源代码，可以通过 clang 静态分析器完成。

10.5. 使用 clang 静态分析器检查源代码

clang 静态分析器是一个对 C、C++ 和 Objective C 源代码执行额外检查的工具。静态分析器执行的检查比编译器执行的检查更彻底。在时间和所需资源方面，成本也更高。静态分析器有一组检查器，用于检查某些错误。

该工具执行源代码的符号解释，通过应用程序查看所有代码路径，并从中派生出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：用于识别常数值。静态分析器的上下文中，检查器应用于派生值。

例如，若除法的除数为零，则静态分析器就会发出警告。可以在 div.c 文件中存储下面的例子来进行检查:

```
int divbyzero(int a, int b) { return a / b; }

int bug() { return divbyzero(5, 0); }
```

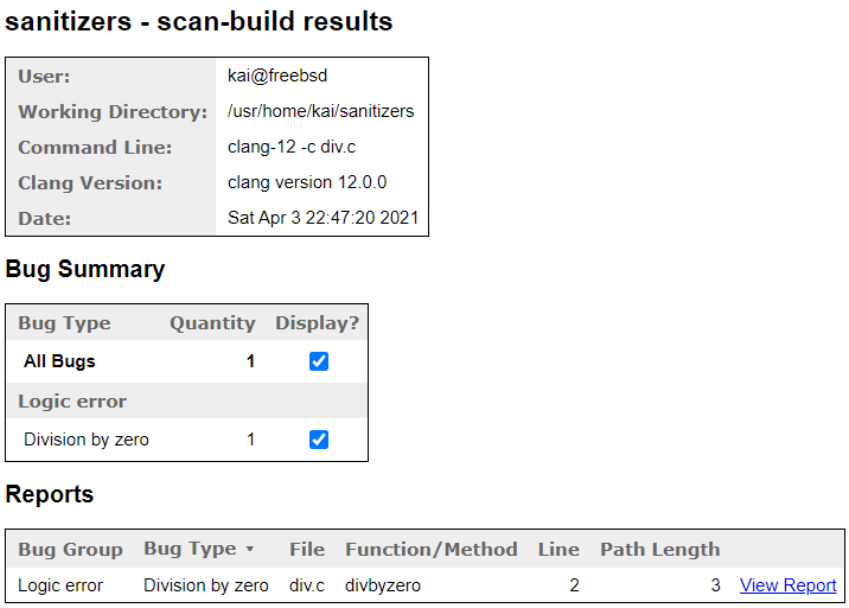
本例中，静态分析器将对除 0 发出警告。但编译时，使用 clang -Wall -c div.c 命令编译该文件时，将不会显示任何警告。

有两种方法可以从命令行调用静态分析器。旧的工具是 scan-build，包含在 LLVM 中，可以用于简单的场景。新的工具是 CodeChecker，可在https://github.com/Ericsson/codechecker/上获得。要检查单个文件，扫描构建工具是最简单的解决方案。只需将编译命令传递给工具，其他的工作都是会自动完成:

```
$ scan-build clang -c div.c
scan-build: Using '/usr/home/kai/LLVM/llvm-17/bin/clang-17' 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/scan-build-2021-03-01-023401-8721-1'
to examine bug reports.
```

屏幕上的输出已经说明发现了一个问题，触发 DivideZero 检查器。在/tmp 目录的上述子目录中，可以找到 HTML 格式的完整报告。可以使用 scan-view 命令查看报告，或者打开在浏览器的子目录中找到的 index.html 文件。

报告的第一页展示了发现的漏洞摘要:



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	☑
Logic error		
Division by zero	1	☑

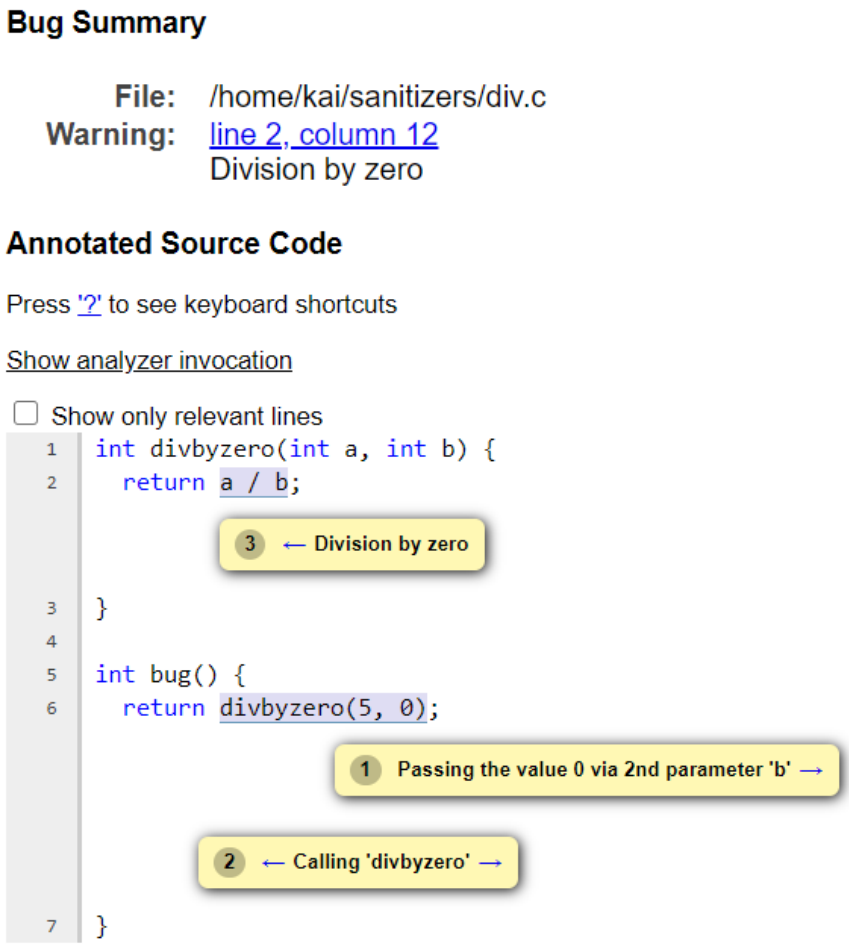
Reports

Bug Group	Bug Type ▾	File	Function/Method	Line	Path Length	
Logic error	Division by zero	div.c	divbyzero	2	3	View Report

图 10.3 - 摘要页面

对于发现的每个错误，摘要页面显示了错误的类型、源码中的位置，以及分析器发现错误之后的路径长度。还提供了指向该错误的详细报告的链接。

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 ← Division by zero
3  }
4
5  int bug() {
6    return divbyzero(5, 0);
                 1 Passing the value 0 via 2nd parameter 'b' →
               2 ← Calling 'divbyzero' →
7  }
```

图 10.4 -详细报告

有了这个详细的报告，就可以通过跟踪编号的气泡来验证错误。我们的简单示例展示了传递 0 作为参数值，如何导致除零错误。

因此，需要人工验证。若派生的约束对于某个检查器来说不够精确，则可能出现误报。

不仅限于该工具提供的检查程序——还可以添加新的检查程序。

10.5.1. 向 clang 静态分析器添加新的检查器

许多 C 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() 函数释放一次。不调用 free() 函数，或者多次调用是一个编程错误。这种编码模式还有很多实例，静态分析器为其中一些实例提供了检查器。

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，必须成对使用，类似于内存管理函数。没有为这些函数实现检查器，让我们实现一个。

要向 clang 静态分析器添加一个新的检查器，必须创建一个新的 checker 类的子类。静态分析器通过代码尝试所有可能的路径。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。若需要处理这些事件，对应的类必须提供回调，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。不知道分析器引擎当前尝试的代码路径，所以检查器无法管理状态。因此，跟踪的状态必须向引擎注册，并且只能使用 ProgramStateRef 实例进行更改。

为了检测错误，检查器需要跟踪从 iconv_open() 函数返回的描述符。分析器引擎为 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 iconv_close() 是否调用。对于状态，我们创建了 IconvState 类，其封装了一个 bool 值。

新的 IconvChecker 类需要处理四种类型的事件:

- PostCall，发生在函数调用之后。调用 iconv_open() 函数之后，检索返回值的符号，并记住其处于“打开”状态。
- PreCall，发生在函数调用之前。调用 iconv_close() 函数之前，检查描述符的符号是否处于“打开”状态。若没有，则已经为描述符调用了 iconv_close() 函数，并且已经检测到对该函数的两次调用。
- DeadSymbols，在清理未使用的符号时发生。我们检查描述符中未使用的符号是否仍处于“打开”状态。是的话，则检测到缺少对 iconv_close() 的调用，这是一个资源泄漏。
- PointerEscape，当分析器无法再跟踪符号时调用该函数。这种情况下，因为不能再推断描述符是否已关闭，所以需要从状态中删除符号。

可以创建一个新目录来实现新的检查器作为 clang 插件，并在 IconvChecker.cpp 文件中添加实现:

1. 对于实现，需要包含几个头文件。发布报告需要包含文件 BugType.h，头文件 Checker.h 提供了 Checker 类的声明和事件的回调，它们在 CallEvent.h 中声明。此外，CallDescription.h 文件

有助于匹配函数和方法。最后，需要使用 CheckerContext.h 文件来声明 CheckerContext 类，该类是提供对分析器状态访问的中心类:

```
#include "clang/StaticAnalyzer/Core/BugReporter/BugType.h"
#include "clang/StaticAnalyzer/Core/Checker.h"
#include "clang/StaticAnalyzer/Core/PathSensitive/CallDescription.h"
#include "clang/StaticAnalyzer/Core/PathSensitive/CallEvent.h"
#include "clang/StaticAnalyzer/Core/PathSensitive/CheckerContext.h"
#include "clang/StaticAnalyzer/Frontend/CheckerRegistry.h"
#include <optional>
```

2. 为了避免输入命名空间的名称，可以使用 clang 和 ento 命名空间:

```
using namespace clang;
using namespace ento;
```

3. 我们将一个状态与代表图标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，使用 bool 类型的变量，打开状态的值为 true。状态值封装在 IconvState 结构体中，该结构体与 FoldingSet 数据结构一起使用，FoldingSet 是一个过滤重复条目的哈希集。为了使用此数据结构实现，这里添加了 Profile() 方法，该方法设置该结构的唯一位。将结构体放入匿名命名空间中，以避免污染全局命名空间。这个类提供了 getOpened() 和 getClosed() 工厂方法以及 isOpen() 查询方法，而非对外暴露 bool 值:

```
namespace {
class IconvState {
    const bool IsOpen;

    IconvState(bool IsOpen) : IsOpen(IsOpen) {}

public:
    bool isOpen() const { return IsOpen; }

    static IconvState getOpened() {
        return IconvState(true);
    }

    static IconvState getClosed() {
        return IconvState(false);
    }

    bool operator==(const IconvState &O) const {
        return IsOpen == O.IsOpen;
    }

    void Profile(llvm::FoldingSetNodeID &ID) const {
        ID.AddInteger(IsOpen);
    }
};
} // namespace
```

4. IconvState 结构体表示 iconv 描述符的状态，其由 SymbolRef 类的符号表示。这最好使用映射来完成，符号作为键，将状态作为值。检查器不能保持状态，儿必须用全局程序状态注册状态，通过 REGISTER_MAP_WITH_PROGRAMSTATE 宏完成。这个宏引入了 IconvStateMap 名称，稍后将使用它来访问映射:

```
REGISTER_MAP_WITH_PROGRAMSTATE(IconvStateMap, SymbolRef,
                               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:

```
namespace {
    class IconvChecker
        : public Checker<check::PostCall, check::PreCall,
                         check::DeadSymbols,
                         check::PointerEscape> {
```

6. IconvChecker 类具有 CallDescription 类型的字段，用于识别程序中对 iconv_open()，iconv() 和 iconv_close() 的函数调用:

```
    CallDescription IconvOpenFn, IconvFn, IconvCloseFn;
```

7. 该类还包含对检测到的错误类型的引用:

```
    std::unique_ptr<BugType> DoubleCloseBugType;
    std::unique_ptr<BugType> LeakBugType;
```

8. 最后，这个类有几个方法。除了构造函数和调用事件的方法之外，还需要一个方法来生成错误报告:

```
    void report(ArrayRef<SymbolRef> Syms,
                const BugType &Bug, StringRef Desc,
                CheckerContext &C, ExplodedNode *ErrNode,
                std::optional<SourceRange> Range =
                    std::nullopt) const;

public:
    IconvChecker();
    void checkPostCall(const CallEvent &Call,
                       CheckerContext &C) const;
    void checkPreCall(const CallEvent &Call,
                      CheckerContext &C) const;
    void checkDeadSymbols(SymbolReaper &SymReaper,
                          CheckerContext &C) const;
    ProgramStateRef
    checkPointerEscape(ProgramStateRef State,
                       const InvalidatedSymbols &Escaped,
                       const CallEvent *Call,
                       PointerEscapeKind Kind) const;
```

```
};
} // namespace
```

9. IconvChecker 类的构造函数的实现，使用函数的名称初始化 CallDescription 字段，并创建表示错误类型的对象:

```
IconvChecker::IconvChecker()
    : IconvOpenFn({"iconv_open"}), IconvFn({"iconv"}),
        IconvCloseFn({"iconv_close"}, 1) {
    DoubleCloseBugType.reset(new BugType(
        this, "Double iconv_close", "Iconv API Error"));
    LeakBugType.reset(new BugType(
        this, "Resource Leak", "Iconv API Error",
        /*SuppressOnSink=*/true));
}
```

10. 现在，可以实现第一个调用事件方法 checkPostCall()，此方法在分析器执行函数调用后调用。若执行的函数不是一个全局 C 函数，也没有命名为 iconv_open:

```
void IconvChecker::checkPostCall(
        const CallEvent &Call, CheckerContext &C) const {
    if (!Call.isGlobalCFunction())
        return;
    if (!IconvOpenFn.matches(Call))
        return;
```

11. 否则，可以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，需要从 CheckerContext 实例中获取一个 ProgramStateRef 实例。状态是不可变的，将符号添加到状态会产生一个新状态。最后，分析器引擎通过调用 addTransition() 方法获知新状态:

```
    if (SymbolRef Handle =
            Call.getReturnValue().getAsSymbol()) {
        ProgramStateRef State = C.getState();
        State = State->set<IconvStateMap>(
            Handle, IconvState::getOpened());
        C.addTransition(State);
    }
}
```

12. 同样，在分析器执行函数之前调用 checkPreCall() 方法。只对一个名为 iconv_close 的全局 C 函数感兴趣:

```
void IconvChecker::checkPreCall(
        const CallEvent &Call, CheckerContext &C) const {
    if (!Call.isGlobalCFunction()) {
        return;
    }
    if (!IconvCloseFn.matches(Call)) {
```

```
        return;
    }
```

13. 若函数的第一个参数的符号 (即 iconv 描述符) 已知，则可以从程序状态中检索该符号的状态:

```
    if (SymbolRef Handle =
            Call.getArgSVal(0).getAsSymbol()) {
        ProgramStateRef State = C.getState();
        if (const IconvState *St =
                State->get<IconvStateMap>(Handle)) {
```

14. 若状态表示关闭状态，则已经检测到双重关闭错误，并且可以生成错误报告。若已经为这个路径生成了错误报告，则调用 generateErrorNode() 返回一个 nullptr 值，所以必须检查这种情况:

```
            if (!St->isOpen()) {
                if (ExplodedNode *N = C.generateErrorNode()) {
                    report(Handle, *DoubleCloseBugType,
                           "Closing a previous closed iconv "
                           "descriptor",
                           C, N, Call.getSourceRange());
                }
                return;
            }
        }
```

15. 否则，必须将符号的状态设置为“关闭”状态:

```
        State = State->set<IconvStateMap>(
            Handle, IconvState::getClosed());
        C.addTransition(State);
    }
}
```

16. 调用 checkDeadSymbols() 方法来清理未使用的符号。我们循环遍历跟踪的所有符号，并询问 SymbolReaper 实例当前符号是否已无效:

```
void IconvChecker::checkDeadSymbols(
        SymbolReaper &SymReaper, CheckerContext &C) const {
    ProgramStateRef State = C.getState();
    SmallVector<SymbolRef, 8> LeakedSyms;
    for (auto [Sym, St] : State->get<IconvStateMap>()) {
        if (SymReaper.isDead(Sym)) {
```

17. 若符号已无效，则需要检查状态。若状态仍然是打开的，这是潜在的资源泄漏。有一个例外:iconv_open() 在出现错误时返回-1。若分析器位于处理此错误的代码路径中，则由于函数调用失败而假定资源泄漏是错误的。可以尝试从 ConstraintManager 实例中获取符号的值，若该值为-1，则不认为符号是资源泄漏。可向 SmallVector 实例添加一个泄漏符号，以便稍后生成错误报告。最后，从程序状态中移除无效符号:

```
            if (St.isOpen()) {
                bool IsLeaked = true;
                if (const llvm::APSInt *Val =
                State->getConstraintManager().getSymVal(
                State, Sym))
                IsLeaked = Val->getExtValue() != -1;
                if (IsLeaked)
                LeakedSyms.push_back(Sym);
                }
                State = State->remove<IconvStateMap>(Sym);
            }
        }
```

18. 循环之后，调用 generateNonFatalErrorNode()，此方法转换到新的程序状态，若此路径还没有错误节点，则返回一个错误节点。LeakedSyms 容器保存了泄漏符号的列表 (可能为空)，调用 report() 方法来生成错误报告:

```
        if (ExplodedNode *N =
                C.generateNonFatalErrorNode(State)) {
            report(LeakedSyms, *LeakBugType,
                "Opened iconv descriptor not closed", C, N);
        }
    }
```

19. 当分析器检测到无法跟踪参数的函数调用时，调用 checkPointerEscape() 函数，必须假设我们不知道 iconv 描述符是否会在函数内部关闭。异常是对 iconv() 的调用，执行转换并且已知不调用 iconv_close() 函数，以及 iconv_close() 函数本身，会在 checkPreCall() 方法中处理。若调用是在系统头文件中，并且知道在调用的函数中参数不会转义，则不会改变状态。其他情况下，都会从状态中删除符号:

```
ProgramStateRef IconvChecker::checkPointerEscape(
        ProgramStateRef State,
        const InvalidatedSymbols &Escaped,
        const CallEvent *Call,
        PointerEscapeKind Kind) const {
    if (Kind == PSK_DirectEscapeOnCall) {
        if (IconvFn.matches(*Call) ||
            IconvCloseFn.matches(*Call))
            return State;
        if (Call->isInSystemHeader() ||
            !Call->argumentsMayEscape())
        return State;
    }
    for (SymbolRef Sym : Escaped)
        State = State->remove<IconvStateMap>(Sym);
    return State;
}
```

20. report() 方法生成一个错误报告。该方法的重要参数是符号数组、错误类型和错误描述。该方法内部，为每个符号创建一个错误报告，并将该符号标记为对该错误感兴趣的符号。若源范围作为参数提供，则会添加到报告中，并生成报告:

```
void IconvChecker::report(
        ArrayRef<SymbolRef> Syms, const BugType &Bug,
        StringRef Desc, CheckerContext &C,
        ExplodedNode *ErrNode,
        std::optional<SourceRange> Range) const {
    for (SymbolRef Sym : Syms) {
        auto R = std::make_unique<PathSensitiveBugReport>(
            Bug, Desc, ErrNode);
        R->markInteresting(Sym);
        if (Range)
            R->addRange(*Range);
        C.emitReport(std::move(R));
    }
}
```

21. 现在，需要在 CheckerRegistry 实例中注册新的检查器。当插件加载时，使用 clang_registerCheckers()，我们在其中执行注册。每个检查器都有一个名称，并且属于一个包。因为 iconv 库是一个标准的 POSIX 接口，所以可以将 IconvChecker 称为检查器并将其放入 unix 打包程序中。这是 addChecker() 方法的第一个参数，第二个参数是功能的简要文档，第三个参数可以是一个文档的 URI，该文档提供有关检查器的更多信息:

```
extern "C" void
clang_registerCheckers(CheckerRegistry &registry) {
    registry.addChecker<IconvChecker>(
        "unix.IconvChecker",
        "Check handling of iconv functions", "");
}
```

22. 最后，需要声明我们正在使用的静态分析器 API 的版本，使系统能够确定插件是否兼容:

```
extern "C" const char clang_analyzerAPIVersionString[] =
    CLANG_ANALYZER_API_VERSION_STRING;
```

这就完成了新检查器的实现。为了构建插件，还需要在 CMakeLists.txt 文件中创建一个构建描述，该文件与 IconvChecker.cpp 位于同一目录下:

23. 首先定义所需的 CMake 版本和项目名称:

```
cmake_minimum_required(VERSION 3.20.0)
project(iconvchecker)
```

24. 接下来，包括 LLVM 文件。若 CMake 不能自动找到文件，则必须设置 LLVM_DIR 变量，指向包含 CMake 文件的 LLVM 目录:

```
find_package(LLVM REQUIRED CONFIG)
```

25. 将包含 CMake 文件的 LLVM 目录附加到搜索路径中，并从 LLVM 中包含所需的模块:

```
list(APPEND CMAKE_MODULE_PATH ${LLVM_DIR})
include(AddLLVM)
include(HandleLLVMOptions)
```

26. 为 clang 加载 CMake 定义。若 CMake 不能自动找到文件，必须指定 Clang_DIR 变量，使其指向包含 CMake 文件的 Clang 目录:

```
find_package(Clang REQUIRED)
```

27. 接下来，将包含 CMake 文件的 Clang 目录附加到搜索路径中，并从 Clang 中包含所需的模块:

```
list(APPEND CMAKE_MODULE_PATH ${Clang_DIR})
include(AddClang)
```

28. 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}")
```

29. 前面的定义设置了构建环境。插入以下命令，定义了插件名称和源文件，并且它是一个 clang 插件:

```
add_llvm_library(IconvChecker MODULE IconvChecker.cpp
                 PLUGIN_TOOL clang)
```

30. Windows 上，插件支持与 Unix 不同，必需的 LLVM 和 clang 库必须链接进来。下面的代码确保了这一点:

```
if(WIN32 OR CYGWIN)
    set(LLVM_LINK_COMPONENTS Support)
    clang_target_link_libraries(IconvChecker PRIVATE
        clangAnalysis
        clangAST
        clangStaticAnalyzerCore
        clangStaticAnalyzerFrontend)
endif()
```

现在，可以配置和构建插件，假设设置了 CMAKE_GENERATOR 和 CMAKE_BUILD_TYPE 环境变量:

```
$ cmake -DLLVM_DIR=~/LLVM/llvm-17/lib/cmake/llvm \
        -DClang_DIR=~/LLVM/llvm-17/lib/cmake/clang \
        -B build
$ cmake --build build
```

可以使用保存在 conf.c 文件中的以下源代码来测试新的检查器，该文件对 iconv_close() 函数有两次调用:

```c
#include <iconv.h>

void doconv() {
    iconv_t id = iconv_open("Latin1", "UTF-16");
    iconv_close(id);
    iconv_close(id);
}
```

要将插件与扫描-构建脚本一起使用，需要通过-load-plugin 选项指定插件的路径。使用 conf.c 文件运行如下:

```
$ scan-build -load-plugin build/IconvChecker.so clang-17 -c conv.c
scan-build: Using '/home/kai/LLVM/llvm-17/bin/clang-17' for static
analysis
conv.c:6:3: warning: Closing a previous closed iconv descriptor [unix.
IconvChecker]
    6 | iconv_close(id);
      | ^~~~~~~~~~~~~~~
1 warning generated.
scan-build: Analysis run complete.
scan-build: 1 bug found.
scan-build: Run 'scan-view /tmp/scan-build-2023-08-08-114154-12451-1' to examine bug
↪  reports.
```

至此，已经了解了如何使用自己的检查器扩展 clang 静态分析器，可以使用这些知识来创建新的通用检查器并将其贡献给社区，或者创建需求专用的构建的检查器，以提高产品的质量。

静态分析器是通过利用 clang 基础设施构建的。下一节将介绍如何构建自己的 clang 插件扩展。

10.6. 创建基于 clang 的工具

静态分析器是一个令人印象深刻的例子，说明了可以使用 clang 基础设施做些什么。也可以使用插件扩展 clang，这样就可以将自己的功能添加到 clang 中。该技术非常类似于在 LLVM 中添加一个通道插件。

我们用一个简单的插件来探索这个功能，LLVM 编码标准要求函数名以小写字母开头。然而，编码标准已经发生了变化，在许多情况下，函数以大写字母开头。一个警告违反命名规则的插件可以帮助解决这个问题，所以来试一试。

因为要在 AST 上运行用户定义的操作，所以需要定义 PluginASTAction 类的子类。若使用 clang 库编写自己的工具，则可以为操作定义 ASTFrontendAction 类的子类。PluginASTAction 类是 ASTFrontendAction 类的子类，具有解析命令行选项的能力。

需要的另一个类是 ASTConsumer 类的子类。ASTConsumer 是一个类，可以使用它在 AST 上运行一个动作，而不管 AST 的来源是什么。可以在 NamingPlugin.cpp 文件中创建实现:

1. 首先包括所需的头文件。除了上面提到的 ASTConsumer 类，还需要一个编译器和插件注册表的实例:

```
#include "clang/AST/ASTConsumer.h"
#include "clang/Frontend/CompilerInstance.h"
#include "clang/Frontend/FrontendPluginRegistry.h"
```

2. 使用 clang 命名空间，并将实现放在一个匿名命名空间中，以避免名称冲突:

```
using namespace clang;
namespace {
```

3. 接下来，定义 ASTConsumer 类的子类。稍后，将希望在检测到违反命名规则的情况下发出警告，所以需要一个对 DiagnosticsEngine 实例的引用。
4. 需要在类中存储一个 CompilerInstance 实例，之后可以请求一个 DiagnosticsEngine 实例:

```
class NamingASTConsumer : public ASTConsumer {
    CompilerInstance &CI;

public:
    NamingASTConsumer(CompilerInstance &CI) : CI(CI) {}
```

5. ASTConsumer 实例有几个入口方法，HandleTopLevelDecl() 方法符合我们的目的，顶层为每个声明调用该方法。这不仅包括函数，还包括变量，所以必须使用 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，可以包含多个声明。这需要对声明进行循环。下面的代码显示了 HandleTopLevelDecl() 方法:

```
    bool HandleTopLevelDecl(DeclGroupRef DG) override {
        for (DeclGroupRef::iterator I = DG.begin(),
                E = DG.end();
        I != E; ++I) {
            const Decl *D = *I;
            if (const FunctionDecl *FD =
                dyn_cast<FunctionDecl>(D)) {
```

6. 找到函数声明后，需要检索函数的名称。还需要确保名称不为空:

```
                std::string Name =
                    FD->getNameInfo().getName().getAsString();
                assert(Name.length() > 0 &&
                    "Unexpected empty identifier");
```

若函数名不是以小写字母开头，就违反了命名规则:

```
                char &First = Name.at(0);
                if (!(First >= 'a' && First <= 'z')) {
```

7. 要发出警告，需要一个 DiagnosticsEngine 实例，所以还需要一个消息 ID。在 clang 内部，消息 ID 定义为枚举。因为插件不是 clang 的一部分，所以需要创建一个自定义 ID，然后可以用它来发出警告:

```
                    DiagnosticsEngine &Diag = CI.getDiagnostics();
                    unsigned ID = Diag.getCustomDiagID(
```

```
                            DiagnosticsEngine::Warning,
                            "Function name should start with "
                            "lowercase letter");
                        Diag.Report(FD->getLocation(), ID);
```

8. 除了关闭所有的大括号，需要从这个函数返回 true 来表示处理可以继续:

```
                    }
                }
            }
            return true;
        }
};
```

9. 接下来，需要创建 PluginASTAction 子类，它实现 clang 调用的接口:

```
class PluginNamingAction : public PluginASTAction {
public:
```

必须实现的第一个方法是 CreateASTConsumer() 方法，其返回 NamingASTConsumer 类的一个实例。该方法由 clang 调用，传递的 CompilerInstance 实例，可以访问编译器的所有重要类:

```
    std::unique_ptr<ASTConsumer>
    CreateASTConsumer(CompilerInstance &CI,
                      StringRef file) override {
        return std::make_unique<NamingASTConsumer>(CI);
    }
```

10. 插件还可以访问命令行选项。目前，插件没有命令行参数，只会返回 true 来表示成功:

```
    bool ParseArgs(const CompilerInstance &CI,
                   const std::vector<std::string> &args)
                                                  override {
        return true;
    }
```

11. 插件的操作类型描述了何时调用该操作。默认值是 Cmdline，则必须在要调用的命令行上命名插件。要重写该方法并将其值更改为 AddAfterMainAction，它会自动运行该操作:

```
    PluginASTAction::ActionType getActionType() override {
        return AddAfterMainAction;
    }
```

12. PluginNamingAction 类的实现完成了，只有类和匿名命名空间的右大括号不见了。将它们添加到代码中:

```
};
}
```

13. 最后，需要注册插件。第一个参数是插件的名称，第二个参数是帮助信息:

```
static FrontendPluginRegistry::Add<PluginNamingAction> X("naming-plugin",
↪  "naming plugin");
```

这就完成了插件的实现。要编译这个插件，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，需要设置一个完整的项目，可以按照以下步骤来做:

1. 首先定义所需的 CMake 版本和项目名称:

```
cmake_minimum_required(VERSION 3.20.0)
project(naminglugin)
```

2. 接下来，包括 LLVM 文件。若 CMake 不能自动找到文件，必须设置 LLVM_DIR 变量，使它指向包含 CMake 文件的 LLVM 目录:

```
find_package(LLVM REQUIRED CONFIG)
```

3. 将包含 CMake 文件的 LLVM 目录附加到搜索路径中，并包含一些必需的模块:

```
list(APPEND CMAKE_MODULE_PATH ${LLVM_DIR})
include(AddLLVM)
include(HandleLLVMOptions)
```

4. 然后，为 clang 加载 CMake 定义。若 CMake 不能自动找到文件，必须设置 Clang_DIR 变量，使其指向包含 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，定义了插件的名称，插件的源文件，并且是一个 clang 插件:

```
add_llvm_library(NamingPlugin MODULE NamingPlugin.cpp
                 PLUGIN_TOOL clang)
```

Windows 上，插件支持与 Unix 不同，必需的 LLVM 和 clang 库必须链接进来:

```
if(WIN32 OR CYGWIN)
    set(LLVM_LINK_COMPONENTS Support)
    clang_target_link_libraries(NamingPlugin PRIVATE
        clangAST clangBasic clangFrontend clangLex)
endif()
```

现在，可以配置和构建插件，假设设置了 CMAKE_GENERATOR 和 CMAKE_BUILD_TYPE 环境变量:

```
$ cmake -DLLVM_DIR=~/LLVM/llvm-17/lib/cmake/llvm \
        -DClang_DIR=~/LLVM/llvm-17/lib/cmake/clang \
        -B build
$ cmake --build build
```

这些步骤创建 NamingPlugin，所以构建目录中的动态库。

要测试这个插件，将下面的源代码保存为 named.c 文件。函数名 Func1 违反了命名规则，但主函数不违反规则:

```
int Func1() { return 0; }
int main() { return Func1(); }
```

要调用这个插件，需要指定-fplugin= 选项:

```
$ clang -fplugin=build/NamingPlugin.so naming.c
naming.c:1:5: warning: Function name should start with lowercase
letter
int Func1() { return 0; }
    ^
1 warning generated.
```

这种调用需要重写 PluginASTAction 类的 getActionType() 方法，并且返回一个不同于 Cmdline 默认值的值。

若没有这样做——例如，想对插件动作的调用有更多的控制——则可以从编译器命令行运行插件:

```
$ clang -cc1 -load ./NamingPlugin.so -plugin naming-plugin naming.c
```

恭喜你——已经构建了你的第一个 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，都是粗粒度的。这可以通过使用 RecursiveASTVisitor 类来解决。该类遍历所有 AST 节点，可以覆盖感兴趣的 VisitXXX() 方法。使用 Visitor，可以按照以下步骤重写这个插件:

1. 需要一个附加的包含来定义 RecursiveASTVisitor 类。插入如下代码:

```
#include "clang/AST/RecursiveASTVisitor.h"
```

2. 然后，将访问者定义为匿名命名空间中的第一个类。将只存储对 AST 上下文的引用，能够访问用于 AST 操作的所有重要方法，包括发出警告所需的 DiagnosticsEngine 实例:

```
class NamingVisitor
    : public RecursiveASTVisitor<NamingVisitor> {
private:
    ASTContext &ASTCtx;
public:
    explicit NamingVisitor(CompilerInstance &CI)
        : ASTCtx(CI.getASTContext()) {}
```

3. 遍历期间，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:

```
    virtual bool VisitFunctionDecl(FunctionDecl *FD) {
        std::string Name =
            FD->getNameInfo().getName().getAsString();
        assert(Name.length() > 0 &&
                "Unexpected empty identifier");
        char &First = Name.at(0);
        if (!(First >= 'a' && First <= 'z')) {
            DiagnosticsEngine &Diag = ASTCtx.getDiagnostics();
            unsigned ID = Diag.getCustomDiagID(
                DiagnosticsEngine::Warning,
                "Function name should start with "
                "lowercase letter");
            Diag.Report(FD->getLocation(), ID);
        }
        return true;
    }
};
```

4. 这就完成了访问者的实现。NamingASTConsumer 类中，只存储一个 Visitor 实例:

```
    std::unique_ptr<NamingVisitor> Visitor;

public:
    NamingASTConsumer(CompilerInstance &CI)
        : Visitor(std::make_unique<NamingVisitor>(CI)) {}
```

5. 删除 HandleTopLevelDecl() 方法——该功能现在位于访问者类中，因此需要重写 HandleTranslationUnit() 方法。这个类对每个翻译单元调用一次，将从这里开始 AST 遍历:

```
    void
    HandleTranslationUnit(ASTContext &ASTCtx) override {
        Visitor->TraverseDecl(
            ASTCtx.getTranslationUnitDecl());
    }
```

这个新实现具有相同的功能，其优点是更容易扩展。例如，若想检查变量声明，必须实现 VisitVarDecl() 方法。若希望使用语句，则必须实现 VisitStmt() 方法。使用这种方法，可以为 C、C++ 和 Objective C 语言的每个实体提供一个 visitor 方法。

通过访问 AST，可以构建执行复杂任务的插件。如本节所述，强制命名约定是对 clang 的一个有用的补充。可以作为插件实现的另一个有用的附加功能是计算软件度量，例如圈复杂度。

还可以添加或替换 AST 节点，例如，添加运行时检测。添加插件可以以所需的方式扩展 clang。

10.7. 总结

本章中，了解了如何使用各种消毒器。使用地址消毒器检测指针错误，使用内存消毒器检测未初始化的内存访问，并使用线程消毒器执行数据争用。应用程序错误通常是由格式不正确的输入触发的，所以实现了模糊测试，用随机数据测试应用程序。

还使用 XRay 对应用程序进行了检测，以确定性能瓶颈，还了解了可视化数据的各种方法。本章还了解了如何利用 clang 静态分析器，通过解释源代码来识别潜在的错误，以及如何创建自己的 clang 插件。

这些技能将提高构建的应用程序的质量，可在应用程序用户抱怨运行时错误前发现它们。运用本章的知识，不仅可以发现大量的常见错误，还可以用新的功能扩展 clang。

下一章中，将了解如何为 LLVM 添加一个新后端。

第四部分 创建自定义后端

了解如何使用 TableGen 语言为 LLVM 不支持的 CPU 架构添加新的后端目标，还将探索 LLVM 中的各种指令选择框架，并了解如何实现。最后，将深入研究 LLVM 中指令选择框架之外的概念，这些概念对于优化后端很有价值。

- 第 11 章，目标描述
- 第 12 章，指令选择
- 第 13 章，超越指令选择

第 11 章　目标描述

LLVM 具有非常灵活的架构，可以添加一个新的目标后端。后端的核心是目标描述，大部分代码都是从目标描述中生成的。本章中，将学习如何添加对历史 CPU 的支持。

本章中，将学习以下主题:

- 为创建一个新的后端做准备，介绍 M88k CPU 架构，了解在哪里可以找到所需的信息
- 将新架构添加到 Triple 类中，了解如何使 LLVM 意识到新的 CPU 架构
- LLVM 中扩展 ELF 文件格式定义，向处理 ELF 对象文件的库和工具中添加对特定于 M88k 的支持
- 了解创建目标描述将应用 TableGen 语言的知识，对目标描述中的注册文件和指令进行建模
- 将 M88k 后端添加到 LLVM，了解 LLVM 后端所需的最小基础结构
- 实现汇编器解析器，了解如何开发汇编器
- 创建反汇编器，了解如何创建反汇编器

本章结束时，将了解如何为 LLVM 添加一个新的后端。将了解在目标描述中开发寄存器文件定义和指令定义的知识，并了解如何根据描述创建汇编程序和反汇编程序。

11.1. 为新后端做准备

无论是商业上需要支持一个新的 CPU,还是一个爱好项目来增加对一些旧架构的支持,向 LLVM 添加新的后端都是一项主要任务。本章和接下来的两章概述了为新后端开发所需的内容，我们将为摩托罗拉 M88k 架构添加一个后端，其为 20 世纪 80 年代的 RISC 架构。

参考

可以在维基百科 (https://en.wikipedia.org/wiki/Motorola_88000) 上阅读有关摩托罗拉架构的信息，关于这种架构的信息仍然可以在互联网上获得。可以在http://www.bitsavers.org/components/motorola/88000/上找到包含指令集和计时信息的 CPU 手册，在https://archive.org/details/bitsavers_attunixSysa0138776555SystemVRelease488000ABI1990_8011463上找到包含 ELF 格式定义和调用约定的 System V ABI M88k 处理器补充。

OpenBSD(可从https://www.openbsd.org/获得) 仍然支持 LUNA-88k 系统。OpenBSD 系统上，很容易为 M88k 创建一个 GCC 交叉编译器。通过 GXemul(可在http://gavare.se/gxemul/上获得)，可获得了一个能够运行 M88k 架构的某些 OpenBSD 版本的模拟器。

M88k 架构已经停产很久了，但我们找到了足够的信息和工具，为它添加 LLVM 后端。我们将从扩展 Triple 类开始。

11.2. 将新架构添加到 Triple 类中

Triple 类的一个实例表示 LLVM 为之生成代码的目标平台。为了支持新架构，第一个任务是扩展 Triple 类。llvm/include/llvm/TargetParser/Triple.h 文件中，添加一个成员到 ArchType 枚举和一个

新谓词:

```
class Triple {
public:
    enum ArchType {
        // Many more members
        m88k, // M88000 (big endian): m88k
    };

    /// Tests whether the target is M88k.
    bool isM88k() const {
        return getArch() == Triple::m88k;
    }
    // Many more methods
};
```

llvm/lib/TargetParser/Triple.cpp 文件中，有许多方法使用 ArchType 枚举。需要扩展它们; 例如，getArchTypeName() 方法中，需要添加一个新 case:

```
switch (Kind) {
    // Many more cases
    case m88k: return "m88k";
}
```

若忘记在其中一个函数中处理新的 m88k 枚举成员，编译器会发出警告。接下来，我们将展开可执行和可链接格式 (ELF)。

11.3. 扩展 LLVM 中的 ELF 文件格式定义

ELF 文件格式是 LLVM 支持的二进制对象文件格式之一。ELF 本身为许多 CPU 架构定义，M88k 架构也有定义。我们所需要做的就是，添加重定位的定义和一些标志。重新定位在《System V ABI M88k 处理器补充》的第 4 章，IR 代码生成基础 (见本章开头设置新后端部分中的链接) 中给出:

1. 我们需要在 llvm/include/llvm/BinaryFormat/ELFRelocs/M88k.def 文件中进行如下修改:

```
#ifndef ELF_RELOC
#error "ELF_RELOC must be defined"
#endif
ELF_RELOC(R_88K_NONE, 0)
ELF_RELOC(R_88K_COPY, 1)
// Many more…
```

2. llvm/include/llvm/BinaryFormat/ELF.h 文件中添加了以下标志，以及重定位的定义:

```
// M88k Specific e_flags
enum : unsigned {
    EF_88K_NABI = 0x80000000, // Not ABI compliant
    EF_88K_M88110 = 0x00000004 // File uses 88110-specific
```

```
    features
};

// M88k relocations.
enum {
    #include "ELFRelocs/M88k.def"
};
```

代码可以添加到文件中的任何位置，但最好保持文件的结构，并在 MIPS 架构代码之前插入。

3. 还需要扩展一些其他的方法。在 llvm/include/llvm/Object/ELFObjectFile.h 文件中，有一些在枚举成员和字符串之间进行转换的方法，必须在 getFileFormatName() 方法中添加新的 case 语句:

```
switch (EF.getHeader()->e_ident[ELF::EI_CLASS]) {
    // Many more cases
    case ELF::EM_88K:
    return "elf32-m88k";
}
```

4. 类似地，可以扩展 getArch() 方法:

```
switch (EF.getHeader().e_machine) {
    // Many more cases
    case ELF::EM_88K:
    return Triple::m88k;
```

5. 最后，在 getELFRelocationTypeName() 方法中，使用 llvm/lib/Object/ELF.cpp 文件中的重定位定义:

```
switch (Machine) {
    // Many more cases
    case ELF::EM_88K:
    switch (Type) {
        #include "llvm/BinaryFormat/ELFRelocs/M88k.def"
        default:
        break;
    }
    break;
}
```

6. 为了完成支持，还可以扩展 llvm/lib/ObjectYAML/ELFYAML.cpp 文件。该文件由 yaml2obj 和 obj2yaml 工具使用，根据 YAML 描述创建一个 ELF 文件。第一个添加需要在 ScalarEnumerationTraits<ELFYAML::ELF_EM>::enumeration() 方法中完成，该方法列出了所有 ELF 支持的架构:

```
ECase(EM_88K);
```

7. ScalarEnumerationTraits<ELFYAML::ELF_REL>::enumeration() 方法中，需要再次包含重定位的定义:

```
    case ELF::EM_88K:
#include "llvm/BinaryFormat/ELFRelocs/M88k.def"
    break;
```

至此，已经完成了对 ELF 文件格式的 m88k 架构的支持。可以使用 llvm-readobj 工具检查 ELF 对象文件，由 OpenBSD 上的交叉编译器创建的对象文件，可以使用 yaml2obj 工具为 m88k 架构创建一个 ELF 对象文件。

必须添加对目标文件格式的支持?

对架构的支持集成到 ELF 文件格式实现中只需要几行代码，若要为其创建的 LLVM 后端使用 ELF 格式，则该采用这种方式。另一方面，添加对全新二进制文件格式的支持是一项复杂的任务。若需要这样做，常用的方法是只输出汇编程序文件，并使用外部汇编程序来创建目标文件。

有了这些补充，ELF 文件格式的 LLVM 实现现在支持 M88k 架构。下一节中，将创建 M88k 架构的目标描述，其中描述了该架构的指令、寄存器和更多细节。

11.4. 创建目标描述

目标描述是后端实现的核心，使用 TableGen 语言编写，并定义了架构的基本属性，例如：寄存器和指令格式以及用于指令选择的模式。若不熟悉 TableGen 语言，建议先阅读本书第 8 章。基本定义在 llvm/include/llvm/Target/Target 文件中，该文件可在https://github.com/llvm/llvm-project/blob/main/llvm/include/llvm/Target/Target.td上找到。该文件有大量注释，都是关于定义使用的信息来源。

我们将根据目标描述生成整个后端。这个目标还没有实现，稍后需要扩展生成的代码。由于其大小，目标描述分割成几个文件。顶级文件是 M88k.td，llvm/lib/Target/M88k 目录下，该目录还包括其他文件。我们从寄存器定义开始。

11.4.1. 添加寄存器定义

CPU 架构通常定义一组寄存器，这些寄存器的特性可以有所不同，一些架构允许访问子寄存器。例如，x86 架构有特殊的寄存器名来访问寄存器值的一部分，其他架构没有实现这个。除了通用寄存器、浮点寄存器和向量寄存器外，架构还可以有用于状态码或浮点操作配置的特殊寄存器。我们需要为 LLVM 定义所有这些信息，寄存器定义存储在 M88kRegisterInfo 中，在 llvm/lib/Target/M88k 目录中也可以找到。

M88k 架构定义了通用寄存器、用于浮点操作的扩展寄存器和控制寄存器。为了保持示例较小，只定义通用寄存器。首先为寄存器定义一个超类，寄存器有名称和编码。该名称用于指令的文本表示。类似地，编码用作指令二进制表示的一部分。该架构定义了 32 个寄存器，寄存器的编码使用 5 位，限制了保存编码的字段。还定义了所有生成的 C++ 代码，应该放在在 M88k 命名空间中:

```
class M88kReg<bits<5> Enc, string n> : Register<n> {
```

```
    let HWEncoding{15-5} = 0;
    let HWEncoding{4-0} = Enc;
    let Namespace = "M88k";
}
```

接下来，可以定义所有 32 个通用寄存器。r0 寄存器很特殊，在读取时总是返回常量 0，所以将该寄存器的 isConstant 标志设置为 true:

```
foreach I = 0-31 in {
    let isConstant = !eq(I, 0) in
        def R#I : M88kReg<I, "r"#I>;
}
```

对于寄存器分配器，需要将单个寄存器分组到寄存器类中。寄存器的顺序，定义了分配顺序。寄存器分配器还需要有关寄存器的其他信息，可以存储在寄存器中的值类型、寄存器的溢出大小(以位为单位)，以及在内存中所需的对齐方式。没有直接使用 RegisterClass 基类，而是创建了一个新的 M88kRegisterClass 类。根据需要更改参数列表，还避免了用于生成代码的 C++ 命名空间名称的重复，这是 RegisterClass 类的第一个参数:

```
class M88kRegisterClass<list<ValueType> types, int size,
                        int alignment, dag regList,
                        int copycost = 1>
    : RegisterClass<"M88k", types, alignment, regList> {
    let Size = size;
    let CopyCost = copycost;
}
```

此外，还定义了一个寄存器操作数类。操作数描述指令的输入和输出，指令的组装和分解过程中使用，也在指令选择阶段使用的模式中使用。使用自己的类，我们可以给用于解码寄存器操作数函数，一个符合 LLVM 编码准则的名称:

```
class M88kRegisterOperand<RegisterClass RC>
: RegisterOperand<RC> {
    let DecoderMethod = "decode"#RC#"RegisterClass";
}
```

基于这些定义，现在定义通用寄存器。注意，m88k 架构的通用寄存器是 32 位宽的，可以保存整数和浮点值。为了避免写入所有寄存器名，使用序列生成器，基于模板字符串生成字符串列表:

```
def GPR : M88kRegisterClass<[i32, f32], 32, 32,
                            (add (sequence "R%u", 0, 31))>;
```

同样，我们定义了寄存器操作数。r0 寄存器很特殊，它只包含常数 0。全局指令选择框架可以使用这一事实，因此将该信息添加到寄存器操作数上:

```
def GPROpnd : M88kRegisterOperand<GPR> {
    let GIZeroRegister = R0;
}
```

m88k 架构有一个扩展，为浮点值定义了一个扩展的寄存器文件，可与通用寄存器相同的方式定义这些寄存器。

通用寄存器也成对使用，主要用于 64 位浮点运算，需要对它们进行建模。使用 sub_hi 和 sub_lo 子寄存器索引来描述高 32 位和低 32 位，还需要为生成的代码设置 C++ 命名空间:

```
let Namespace = "M88k" in {
    def sub_hi : SubRegIndex<32, 0>;
    def sub_lo : SubRegIndex<32, 32>;
}
```

使用 RegisterTuples 类定义寄存器对，该类将子寄存器索引列表作为第一个参数，并将寄存器列表作为第二个参数。只需要偶数/奇数对，就可通过 sequence 的第四个可选参数来实现这一点，这是在生成序列时使用的步幅:

```
def GRPair : RegisterTuples<[sub_hi, sub_lo],
                            [(add (sequence "R%u", 0, 30, 2)),
                            (add (sequence "R%u", 1, 31, 2))]>;
```

要使用寄存器对，需要定义一个寄存器类和一个寄存器操作数:

```
def GPR64 : M88kRegisterClass<[i64, f64], 64, 32,
                              (add GRPair), /*copycost=*/ 2>;
def GPR64Opnd : M88kRegisterOperand<GPR64>;
```

注意，我们将 copycost 参数设置为 2，因为需要两个指令而不是一个指令来将一个寄存器对复制到另一个寄存器对。

这就完成了对寄存器的定义。下一节中，我们将定义指令格式。

11.4.2. 定义指令格式和指令信息

指令是使用 TableGen 指令类定义的。定义一条指令是一项复杂的任务，必须考虑许多细节，指令具有供汇编程序和反汇编程序使用的文本表示形式。有一个名称，并且可以有操作数。汇编程序将文本表示转换为二进制格式，必须定义该格式的布局。对于指令选择，需要给指令附加一个模式。为了管理这种复杂性，需要定义一个类层次结构。基类将描述各种指令格式，并存储在 M88kIntrFormats.td 文件中，指令本身和指令选择所需的其他定义存储在 M88kInstrInfo.td 文件中。

首先为 m88k 架构指令定义一个名为 M88kInst 的类，可从预定义的指令类派生出这个类。新类有几个参数，outs 和 ins 参数使用特殊的 dag 类型将输出和输入操作数描述为一个列表。指令的文本表示分为 asm 参数中给出的助记符和操作数，pattern 参数可以保存用于指令选择的模式。

还要定义两个新字段:

- Inst 字段用于保存指令的位模式。因为指令的大小取决于平台，所以这个字段不能预先定义。m88k 架构的所有指令都是 32 位宽的，所以该字段为 bits<32> 类型。
- 另一个字段称为 SoftFail，与 Inst 具有相同的类型。保存一个位掩码，用于指令的实际编码可以与 Inst 字段中的位不同，但仍然有效。唯一需要这个的平台是 ARM，所以可以简单地将这个字段设置为 0。

其他字段在超类中定义，只设置值。TableGen 语言中可以进行简单的计算，为 AsmString 字段创建值时使用，AsmString 字段包含完整的汇编程序表示。若操作数的操作数字符串为空，AsmString 字段将只包含 asm 参数的值；否则，将是两个字符串的连接，它们之间有一个空格:

```
class InstM88k<dag outs, dag ins, string asm, string operands,
               list<dag> pattern = []>
    : Instruction {
    bits<32> Inst;
    bits<32> SoftFail = 0;
    let Namespace = "M88k";
    let Size = 4;
    dag OutOperandList = outs;
    dag InOperandList = ins;
    let AsmString = !if(!eq(operands, ""), asm,
                        !strconcat(asm, " ", operands));
    let Pattern = pattern;
    let DecoderNamespace = "M88k";
}
```

对于指令编码，制造商通常将指令分组在一起，一组指令具有相似的编码。可以使用这些组系统地创建定义指令格式的类。例如，m88k 架构的所有逻辑操作都将目标寄存器编码为 21 到 25 位，将第一个源寄存器编码为 16 到 20 位。请注意这里的实现模式: 为值声明了 rd 和 rs1 字段，并将这些值赋给了之前在超类中定义的 Inst 字段的正确位:

```
class F_L<dag outs, dag ins, string asm, string operands,
          list<dag> pattern = []>
        : InstM88k<outs, ins, asm, operands, pattern> {
    bits<5> rd;
    bits<5> rs1;
    let Inst{25-21} = rd;
    let Inst{20-16} = rs1;
}
```

有几组基于这种格式的逻辑操作。其中一种是使用三个寄存器的指令组，在手册中称为三元寻址模式:

```
class F_LR<bits<5> func, bits<1> comp, string asm,
           list<dag> pattern = []>
    : F_L<(outs GPROpnd:$rd), (ins GPROpnd:$rs1, GPROpnd:$rs2),
           !if(comp, !strconcat(asm, ".c"), asm),
           "$rd, $rs1, $rs2", pattern> {
```

```
    bits<5> rs2;
    let Inst{31-26} = 0b111101;
    let Inst{15-11} = func;
    let Inst{10} = comp;
    let Inst{9-5} = 0b00000;
    let Inst{4-0} = rs2;
}
```

让我们更详细地研究一下这个类提供的功能。函数形参指定操作，作为一个特殊的特性，第二个操作数可以在操作之前进行补充，可以通过将标志 comp 设置为 1 来表示。助记符在 asm 参数中给出，并可以传递指令选择模式。

通过初始化超类，可以提供更多信息。and 指令的完整汇编文本模板是 and $rd，$rs1，$rs2。对于该组的所有指令，操作数字符串是固定的，所以可以在这里定义。助记符由该类的用户提供，可以在这里连接.c 后缀，这表示应该首先补充第二个操作数。最后，可以定义输出和输入操作数，这些操作数表示为有向无环图或简称为 dag。dag 有一个操作和一个参数列表。参数也可以是 dag，允许构造复杂的图。例如，输出操作数为 (outs GPROpnd:$rd)。

outs 操作将此 dag 表示为输出操作数列表。唯一的参数 GPROpnd:$rd 由一个类型和一个名称组成，连接了之前看到的几个部分。类型是 GPROnd，在前一节中定义的寄存器操作数的名称。名称 $rd 指的是目标寄存器。前面的操作数字符串中使用了这个名称，并且在 F_L 超类中也使用了这个名称，输入操作数的定义类似，类的其余部分初始化 Inst 字段的其他部分。请花点时间检查所有 32 位是否已经分配。

将最后的指令定义放在 M88kInstrInfo.td 文件中。由于每个逻辑指令都有两个变体，因此使用一个多类来同时定义这两个指令。这里也将指令选择的模式定义为有向无环图，设置模式中的操作，第一个参数是目标寄存器。第二个参数是一个嵌套图，这是实际的模式。同样，操作的名称是第一个 OpNode 元素。

LLVM 有许多预定义的操作，这些操作可以在 llvm/include/llvm/Target/TargetSelectionDAG.td 文件中找到件 (https://github.com/llvm/llvm-project/blob/main/llvm/include/llvm/Target/TargetSelectionDAG.td)。例如，有 and 操作，表示位与操作。参数是两个源寄存器，$rs1 和 $rs2。大致可以这样阅读该模式: 若指令选择的输入包含使用两个寄存器的 OpNode 操作，则该操作的结果赋值给 $rd 寄存器，并生成该指令。利用图结构，可以定义更复杂的模式。例如，第二个模式使用非操作数将补码集成到模式中。

需要指出的一个小细节是，逻辑运算是可交换的。这对指令选择很有帮助，所以可以将这些指令的 isCommutable 标志设置为 1:

```
multiclass Logic<bits<5> Fun, string OpcStr, SDNode OpNode> {
    let isCommutable = 1 in
    def rr : F_LR<Fun, /*comp=*/0b0, OpcStr,
                    [(set i32:$rd,
                    (OpNode GPROpnd:$rs1, GPROpnd:$rs2))]>;
    def rrc : F_LR<Fun, /*comp=*/0b1, OpcStr,
                    [(set i32:$rd,
                    (OpNode GPROpnd:$rs1, (not GPROpnd:$rs2)))]>;
}
```

最后，定义指令的记录:

```
defm AND : Logic<0b01000, "and", and>;
defm XOR : Logic<0b01010, "xor", xor>;
defm OR : Logic<0b01011, "or", or>;
```

第一个参数是函数的位模式，第二个是助记符，第三个参数是模式中使用的 dag 操作。

要完全理解类层次结构，重新查看类定义。指导设计原则是避免信息的重复。例如，0b01000 函数位模式只使用一次。若没有 Logic 多类，需要输入此位模式两次，并重复该模式几次，这很容易出错。

注意，最好为说明建立一个命名方案。例如，and 指令的记录命名为 ANDrr，而带有补充寄存器的变体命名为 ANDrrc。这些名称最终会出现在生成的 C++ 源码，使用命名方案有助于理解该名称指的是哪条汇编指令。

目前，我们对 m88k 架构的寄存器文件进行了建模，并定义了一些指令。下一节中，我们将创建顶层文件。

11.4.3. 为目标描述创建顶层文件

我们创建了 M88kRegisterInfo.td，M88kInstrFormats.td 和 M88kInstrInfo.td 的文件。目标描述是一个名为 M88k.td 的文件，该文件首先包含 LLVM 定义:

```
include "llvm/Target/Target.td"

include "M88kRegisterInfo.td"
include "M88kInstrFormats.td"
include "M88kInstrInfo.td"
```

我们将在稍后添加更多后端功能时扩展这个 include 部分。

顶层文件还定义了一些全局实例，第一个名为 M88kInstrInfo 的记录保存了所有指令的信息:

```
def M88kInstrInfo : InstrInfo;
```

我们将汇编类称为 M88kAsmParser。为了使 TableGen 能够识别硬编码寄存器，指定寄存器名，以百分号为前缀，需要定义一个汇编解析器来指定:

```
def M88kAsmParser : AsmParser;
def M88kAsmParserVariant : AsmParserVariant {
    let RegisterPrefix = "%";
}
```

最后，需要定义目标:

```
def M88k : Target {
    let InstructionSet = M88kInstrInfo;
```

```
    let AssemblyParsers = [M88kAsmParser];
    let AssemblyParserVariants = [M88kAsmParserVariant];
}
```

现在，已经定义了足够的目标，可以编写第一个实用程序了。下一节中，我们将向 LLVM 添加 M88k 后端。

11.5. 为 LLVM 添加 M88k 后端

我们还没有讨论在哪里放置目标描述文件，LLVM 中的每个后端在 llvm/lib/Target 中都有一个子目录。这里创建 M88k 目录，并将目标描述文件复制到其中。

当然，仅添加 TableGen 文件是不够的。LLVM 使用注册表查找目标实现的实例，并期望某些全局函数注册这些实例。由于已经生成了一些部分，所以已经可以提供一个实现。

关于目标的所有信息，如目标机器、汇编器、反汇编器等的目标三元组和工厂函数，都存储在 target 类的实例中。每个目标都持有该类的一个静态实例，这个实例在中心注册表中注册:

1. 实现在目标的 TargetInfo 子目录下的 M88kTargetInfo.cpp 文件中。Target 类的单个实例保存在 getTheM88kTarget() 函数中:

```
using namespace llvm;
Target &llvm::getTheM88kTarget() {
    static Target TheM88kTarget;
    return TheM88kTarget;
}
```

2. LLVM 要求每个目标提供一个 LLVMInitialize<Target Name> TargetInfo() 函数来注册目标实例。这个函数必须有一个 C 链接，可以在 LLVM 的 C API 中使用:

```
extern "C" LLVM_EXTERNAL_VISIBILITY void
LLVMInitializeM88kTargetInfo() {
    RegisterTarget<Triple::m88k, /*HasJIT=*/false> X(
        getTheM88kTarget(), "m88k", "M88k", "M88k");
}
```

3. 还需要在同一目录下创建一个 M88kTargetInfo.h 头文件，只包含一个声明:

```
namespace llvm {
class Target;
Target &getTheM88kTarget();
}
```

4. 最后，添加一个 CMakeLists.txt 文件用于构建:

```
add_llvm_component_library(LLVMM88kInfo
    M88kTargetInfo.cpp
    LINK_COMPONENTS Support
    ADD_TO_COMPONENT M88k)
```

接下来，用机器代码 (MC) 级别使用的信息部分填充目标实例:

1. 实现在 MCTargetDesc 子目录下的 M88kMCTargetDesc.cpp 文件中，TableGen 将前一节中创建的目标描述转换为 C++ 源码段。这里，我们包括寄存器信息、指令信息和子目标信息的部分:

```
using namespace llvm;

#define GET_INSTRINFO_MC_DESC
#include "M88kGenInstrInfo.inc"

#define GET_SUBTARGETINFO_MC_DESC
#include "M88kGenSubtargetInfo.inc"

#define GET_REGINFO_MC_DESC
#include "M88kGenRegisterInfo.inc"
```

2. 目标注册中心期望这里的每个类都有一个工厂方法。从指令信息开始，分配一个 MCInstrInfo 类的实例，并调用 InitM88kMCInstrInfo() 生成的函数来填充对象:

```
static MCInstrInfo *createM88kMCInstrInfo() {
    MCInstrInfo *X = new MCInstrInfo();
    InitM88kMCInstrInfo(X);
    return X;
}
```

3. 接下来，分配 MCRegisterInfo 类的对象，并调用生成的函数来填充。M88k::R1 参数值告诉 LLVM R1 寄存器保存着返回地址:

```
static MCRegisterInfo *
createM88kMCRegisterInfo(const Triple &TT) {
    MCRegisterInfo *X = new MCRegisterInfo();
    InitM88kMCRegisterInfo(X, M88k::R1);
    return X;
}
```

4. 最后，需要一个用于子目标信息的工厂方法。这个方法接受一个目标三元组、一个 CPU 名称和一个特征字符串作为参数，并将它们转发给生成函数:

```
static MCSubtargetInfo *
createM88kMCSubtargetInfo(const Triple &TT,
StringRef CPU, StringRef FS) {
    return createM88kMCSubtargetInfoImpl(TT, CPU,
                                         /*TuneCPU*/ CPU,
                                         FS);
}
```

5. 定义了工厂方法之后，现在可以注册它们了。与目标注册类似，LLVM 期望一个名为 LLVMInitialize<Target Name> TargetMC() 的全局函数:

```
extern "C" LLVM_EXTERNAL_VISIBILITY void
LLVMInitializeM88kTargetMC() {
    TargetRegistry::RegisterMCInstrInfo(
        getTheM88kTarget(), createM88kMCInstrInfo);
```

```
    TargetRegistry::RegisterMCRegInfo(
        getTheM88kTarget(), createM88kMCRegisterInfo);
    TargetRegistry::RegisterMCSubtargetInfo(
        getTheM88kTarget(), createM88kMCSubtargetInfo);
}
```

6. M88kMCTargetDesc.h 头文件只是使一些生成的代码可用:

```
#define GET_REGINFO_ENUM
#include "M88kGenRegisterInfo.inc"

#define GET_INSTRINFO_ENUM
#include "M88kGenInstrInfo.inc"

#define GET_SUBTARGETINFO_ENUM
#include "M88kGenSubtargetInfo.inc"
```

实现差不多完成了。为了避免链接器错误，需要提供另一个函数，该函数为 TargetMachine 类的对象注册一个工厂方法。这个类是代码生成所必需的，将在第 12 章中实现。这里，只是在 M88kTargetMachine.cpp 文件中定义了一个空函数:

```
#include "TargetInfo/M88kTargetInfo.h"
#include "llvm/MC/TargetRegistry.h"

extern "C" LLVM_EXTERNAL_VISIBILITY void
LLVMInitializeM88kTarget() {
    // TODO Register the target machine. See chapter 12.
}
```

这就是我们的第一个实现，但 LLVM 还不知道添加了新后端。为了集成它，打开 llvm/CMakeLists.txt 文件，找到定义所有实验目标的部分，将 M88k 目标添加到列表中:

```
set(LLVM_ALL_EXPERIMENTAL_TARGETS ARC ··· M88k ···)
```

假设新后端 LLVM 源代码在目录中，可以通过输入以下命令来配置构建:

```
$ mkdir build
$ cd build
$ cmake -DLLVM_EXPERIMENTAL_TARGETS_TO_BUILD=M88k \
../llvm-m88k/llvm
···
-- Targeting M88k
···
```

构建 LLVM 之后，可以验证工具已经知道我们的新目标:

```
$ bin/llc -version
LLVM (http://llvm.org/):
```

```
  LLVM version 17.0.2
  Registered Targets:
      m88k - M88k
```

到达这里还是有一些困难的，现在可以庆祝一下了!

修复可能的编译错误

LLVM 17.0.2 中有一个问题，会导致编译错误。在代码中的一个地方，子目标信息的 TableGen 发射器使用已删除的值 llvm::None，而非 std::nullopt，从而在编译 M88kMCTargetDesc.cpp 时导致错误。修复此问题的最简单方法是从 LLVM 18 开发分支中挑选修复该问题的提交:git cherry-pick -x a587f429。

下一节中，将实现汇编解析器，这将是我们第一个实现的 LLVM 工具。

11.6. 实现汇编解析器

汇编器解析器很容易实现，LLVM 为它提供了一个框架，而且大部分都是从目标描述生成的。

当框架检测到需要解析指令时，就会调用类中的 ParseInstruction() 方法，通过提供的词法分析器解析输入，并构造一个所谓的操作数向量。操作数可以是标记，如指令助记符、寄存器名或直接对象，也可以是特定于目标的类别。例如，从 jmp %r2 输入构造两个操作数: 一个用于助记符的标记操作数和一个寄存器操作数。

然后，生成的匹配器尝试将操作数向量与指令进行匹配。若找到匹配项，则创建 MCInst 类的实例，其中保存解析后的指令；否则，会发出错误消息。这种方法的优点是，可自动地从目标描述中派生出匹配器，不需要处理所有的语法问题。

但我们需要添加更多的支持类，来使汇编解析器工作，这些类都放在 MCTargetDesc 目录中。

实现 M88k 目标的 MCAsmInfo 支持类

本节中，将探讨实现汇编解析器配置所需的第一个类:MCAsmInfo 类:

1. 需要为汇编解析器设置一些自定义参数，MCAsmInfo 基类 (https://github.com/llvm/llvm-project/blob/main/llvm/include/llvm/MC/MCAsmInfo.h) 包含常用参数。此外，为每种支持的对象文件格式创建一个子类：例如，MCAsmInfoELF 类 (https://github.com/llvm/llvm-project/blob/main/llvm/include/llvm/MC/MCAsmInfoELF.h)。原因是，因为它们必须支持相似的特性，所以使用相同对象文件格式的系统上的系统汇编程序共享相同的特征。我们的目标操作系统是 OpenBSD，使用 ELF 文件格式，因此从 MCAsmInfoELF 类派生出我们的 m88kmccasminfo 类。M88kMCAsmInfo.h 文件中的声明如下:

```
namespace llvm {
class Triple;

class M88kMCAsmInfo : public MCAsmInfoELF {
```

```
public:
    explicit M88kMCAsmInfo(const Triple &TT);
};
```

2. M88kMCAsmInfo.cpp 文件中的实现只设置两个默认值。目前的两个关键设置是系统使用大端模式和使用 | 符号进行注释，其他设置用于生成之后的代码:

```
using namespace llvm;

M88kMCAsmInfo::M88kMCAsmInfo(const Triple &TT) {
    IsLittleEndian = false;
    UseDotAlignForAlignment = true;
    MinInstAlignment = 4;
    CommentString = "|"; // # as comment delimiter is only
                         // allowed at first column
    ZeroDirective = "\t.space\t";
    Data64bitsDirective = "\t.quad\t";
    UsesELFSectionDirectiveForBSS = true;
    SupportsDebugInformation = false;
    ExceptionsType = ExceptionHandling::SjLj;
}
```

现在已经完成了 MCAsmInfo 类的实现。我们将学习实现的下一个类，可在 LLVM 中创建指令的二进制表示。

为 M88k 目标实现 MCCodeEmitter 支持类

LLVM 内部，一条指令由 MCInst 类的实例表示，指令可以作为汇编文本或以二进制形式发送到目标文件中。M88kMCCodeEmitter 类创建指令的二进制表示，而 M88kInstPrinter 类可生成其文本表示。

首先，将实现 M88kMCCodeEmitter 类，存储在 M88kMCCodeEmitter.cpp 文件中:

1. 类的大部分由 TableGen 生成，所以只需要添加一些样板代码。注意，没有相应的头文件; 工厂函数的原型将添加到 M88kMCTargetDesc.h 文件中。首先，为生成的指令数量设置一个统计计数器:

```
using namespace llvm;
#define DEBUG_TYPE "mccodeemitter"
STATISTIC(MCNumEmitted,
          "Number of MC instructions emitted");
```

2. M88kMCCodeEmitter 类位于匿名命名空间中，只需要实现在基类中声明的 encodeInstruction() 方法和 getMachineOpValue() 辅助方法。另一个 getBinaryCodeForInstr() 方法由 TableGen 根据目标描述生成:

```
namespace {
class M88kMCCodeEmitter : public MCCodeEmitter {
    const MCInstrInfo &MCII;
```

```
    MCContext &Ctx;

public:
    M88kMCCodeEmitter(const MCInstrInfo &MCII,
                      MCContext &Ctx)
    : MCII(MCII), Ctx(Ctx) {}

    ~M88kMCCodeEmitter() override = default;

    void encodeInstruction(
        const MCInst &MI, raw_ostream &OS,
        SmallVectorImpl<MCFixup> &Fixups,
        const MCSubtargetInfo &STI) const override;

    uint64_t getBinaryCodeForInstr(
        const MCInst &MI,
        SmallVectorImpl<MCFixup> &Fixups,
        const MCSubtargetInfo &STI) const;

    unsigned
    getMachineOpValue(const MCInst &MI,
                      const MCOperand &MO,
                      SmallVectorImpl<MCFixup> &Fixups,
                      const MCSubtargetInfo &STI) const;
};
} // end anonymous namespace
```

3. encodeInstruction() 方法只是查找指令的二进制表示，将统计计数器加 1，然后以大端格式写出字节。指令有一个固定的 4 字节大小，所以在端流上使用 uint32_t 类型:

```
void M88kMCCodeEmitter::encodeInstruction(
        const MCInst &MI, raw_ostream &OS,
        SmallVectorImpl<MCFixup> &Fixups,
        const MCSubtargetInfo &STI) const {
    uint64_t Bits =
        getBinaryCodeForInstr(MI, Fixups, STI);
    ++MCNumEmitted;
    support::endian::write<uint32_t>(OS, Bits,
                                     support::big);
}
```

4. getMachineOpValue() 方法的任务是返回操作数的二进制表示形式，我们定义了用于存储在指令中的寄存器的位范围，计算存储在这些地方的值。从生成的代码调用该方法，只支持两种情况。对于寄存器，将返回目标描述中定义的寄存器编码。对于 immediate，返回的是立即数:

```
unsigned M88kMCCodeEmitter::getMachineOpValue(
        const MCInst &MI, const MCOperand &MO,
        SmallVectorImpl<MCFixup> &Fixups,
        const MCSubtargetInfo &STI) const {
```

```
    if (MO.isReg())
        return Ctx.getRegisterInfo()->getEncodingValue(
            MO.getReg());
    if (MO.isImm())
        return static_cast<uint64_t>(MO.getImm());
    return 0;
}
```

5. 最后，生成的文件并为类创建一个工厂方法:

```
#include "M88kGenMCCodeEmitter.inc"

MCCodeEmitter *
llvm::createM88kMCCodeEmitter(const MCInstrInfo &MCII,
                              MCContext &Ctx) {
    return new M88kMCCodeEmitter(MCII, Ctx);
}
```

实现了 M88k 目标的指令输出器支持类

M88kInstPrinter 类具有与 M88kMCCodeEmitter 类相似的结构，InstPrinter 类负责生成 LLVM 指令的文本表示。类的大部分是由 TableGen 生成的，但必须添加对输出操作数的支持。该类在 M88kInstPrinter.h 头文件中声明，实现在 M88kInstPrinter.cpp 文件中:

1. 头文件开始。在包含必需的头文件和声明 llvm 命名空间之后，声明两个前向引用以减少必需的 include 的数量:

```
namespace llvm {
class MCAsmInfo;
class MCOperand;
```

2. 除了构造函数，只需要实现 printOperand() 和 printInst() 方法。其他方法由 TableGen 生成:

```
class M88kInstPrinter : public MCInstPrinter {
public:
    M88kInstPrinter(const MCAsmInfo &MAI,
                    const MCInstrInfo &MII,
                    const MCRegisterInfo &MRI)
        : MCInstPrinter(MAI, MII, MRI) {}

    std::pair<const char *, uint64_t>
    getMnemonic(const MCInst *MI) override;
    void printInstruction(const MCInst *MI,
                          uint64_t Address,
                          const MCSubtargetInfo &STI,
                          raw_ostream &O);

    static const char *getRegisterName(MCRegister RegNo);

```

```
    void printOperand(const MCInst *MI, int OpNum,
                      const MCSubtargetInfo &STI,
                      raw_ostream &O);

    void printInst(const MCInst *MI, uint64_t Address,
                   StringRef Annot,
                   const MCSubtargetInfo &STI,
                   raw_ostream &O) override;
};
} // end namespace llvm
```

3. 该实现位于 M88kInstPrint.cpp 文件中。包含所需的头文件并使用 llvm 命名空间之后，生成的 C++ 的文件应包含在内:

```
using namespace llvm;

#define DEBUG_TYPE "asm-printer"

#include "M88kGenAsmWriter.inc"
```

4. printOperand() 方法会检查操作数的类型，并生成一个寄存器名或立即数。通过生成的 getRegisterName() 方法查找寄存器名:

```
void M88kInstPrinter::printOperand(
        const MCInst *MI, int OpNum,
        const MCSubtargetInfo &STI, raw_ostream &O) {
    const MCOperand &MO = MI->getOperand(OpNum);
    if (MO.isReg()) {
        if (!MO.getReg())
            O << '0';
        else
            O << '%' << getRegisterName(MO.getReg());
    } else if (MO.isImm())
        O << MO.getImm();
    else
        llvm_unreachable("Invalid operand");
}
```

5. printInst() 方法只调用 printInstruction() 生成的方法来输出指令，然后使用 printAnnotation() 方法来输出可能需要的注释:

```
void M88kInstPrinter::printInst(
        const MCInst *MI, uint64_t Address, StringRef Annot,
        const MCSubtargetInfo &STI, raw_ostream &O) {
    printInstruction(MI, Address, STI, O);
    printAnnotation(O, Annot);
}
```

实现特定 M88k 的目标描述

M88kMCTargetDesc.cpp 文件，可以需要添加一些内容:

1. 首先，为 MCInstPrinter 类和 MCAsmInfo 类创建一个新的工厂方法:

```
static MCInstPrinter *createM88kMCInstPrinter(
        const Triple &T, unsigned SyntaxVariant,
        const MCAsmInfo &MAI, const MCInstrInfo &MII,
        const MCRegisterInfo &MRI) {
    return new M88kInstPrinter(MAI, MII, MRI);
}

static MCAsmInfo *
        createM88kMCAsmInfo(const MCRegisterInfo &MRI,
        const Triple &TT,
        const MCTargetOptions &Options) {
    return new M88kMCAsmInfo(TT);
}
```

2. 最后，LLVMInitializeM88kTargetMC() 函数中，需要添加工厂方法的注册:

```
extern "C" LLVM_EXTERNAL_VISIBILITY void
LLVMInitializeM88kTargetMC() {
    // ···
    TargetRegistry::RegisterMCAsmInfo(
        getTheM88kTarget(), createM88kMCAsmInfo);
    TargetRegistry::RegisterMCCodeEmitter(
        getTheM88kTarget(), createM88kMCCodeEmitter);
    TargetRegistry::RegisterMCInstPrinter(
        getTheM88kTarget(), createM88kMCInstPrinter);
}
```

已经实现了所有必需的支持类，最后可以添加汇编解析器了。

创建 M88k 汇编解析器类

AsmParser 目录中只有一个 M88kAsmParser.cpp 实现文件。M88kOperand 类表示已解析的操作数，由生成的源代码和 M88kAssembler 类中的汇编解析器实现使用。两个类都在一个匿名命名空间中，只有工厂方法是全局可见的。先来看看 M88kOperand 类:

1. 操作数可以是标记、寄存器或立即数。我们定义了 OperandKind 枚举来区分这些情况，当前类型存储在 kind 成员中。还存储了操作数的起始和结束位置，用于输出错误信息:

```
class M88kOperand : public MCParsedAsmOperand {
    enum OperandKind { OpKind_Token, OpKind_Reg,
                       OpKind_Imm };
    OperandKind Kind;
    SMLoc StartLoc, EndLoc;
```

2. 为了存储该值，定义了一个联合。标记存储为 StringRef，寄存器由其编号标识。直接对象由 MCExpr 类表示:

```
    union {
        StringRef Token;
        unsigned RegNo;
        const MCExpr *Imm;
    };
```

3. 构造函数初始化除了联合之外的所有字段，还定义了返回起始和结束位置值的方法:

```
public:
    M88kOperand(OperandKind Kind, SMLoc StartLoc,
                SMLoc EndLoc)
        : Kind(Kind), StartLoc(StartLoc), EndLoc(EndLoc) {}
    SMLoc getStartLoc() const override { return StartLoc; }
    SMLoc getEndLoc() const override { return EndLoc; }
```

4. 对于每个操作数类型，必须定义四个方法。对于寄存器，方法是 isReg() 检查操作数是否为寄存器，getReg() 返回值，createReg() 创建寄存器操作数，以及 addRegOperands() 为指令添加操作数。后一个函数由生成的源代码在构造指令时调用。标记和立即数的方法类似:

```
bool isReg() const override {
    return Kind == OpKind_Reg;
}

unsigned getReg() const override { return RegNo; }

static std::unique_ptr<M88kOperand>
createReg(unsigned Num, SMLoc StartLoc,
          SMLoc EndLoc) {
    auto Op = std::make_unique<M88kOperand>(
        OpKind_Reg, StartLoc, EndLoc);
    Op->RegNo = Num;
    return Op;
}

void addRegOperands(MCInst &Inst, unsigned N) const {
    assert(N == 1 && "Invalid number of operands");
    Inst.addOperand(MCOperand::createReg(getReg()));
}
```

5. 最后，超类定义了一个需要实现的 print() 虚函数。这只用于调试目的:

```
void print(raw_ostream &OS) const override {
    switch (Kind) {
        case OpKind_Imm:
            OS << "Imm: " << getImm() << "\n"; break;
        case OpKind_Token:
            OS << "Token: " << getToken() << "\n"; break;
```

```
        case OpKind_Reg:
            OS << "Reg: "
               << M88kInstPrinter::getRegisterName(getReg())
               << "\n"; break;
        }
    }
};
```

接下来，声明 M88kAsmParser 类。匿名命名空间将在声明后结束:

1. 类的开始，将包含生成的代码段:

```
class M88kAsmParser : public MCTargetAsmParser {
#define GET_ASSEMBLER_HEADER
#include "M88kGenAsmMatcher.inc"
```

2. 接下来，定义必需的字段。需要一个对实际解析器的引用，其属于 MCAsmParser 类，以及对子目标信息的引用:

```
    MCAsmParser &Parser;
    const MCSubtargetInfo &SubtargetInfo;
```

3. 为了实现汇编器，我们覆写了 MCTargetAsmParser 超类中定义的两个方法。MatchAndEmitInstruction() 尝试匹配一条指令，并发出由 MCInst 类的实例表示的指令。解析指令是在 ParseInstruction() 方法中完成的，而 parseRegister() 和 tryParseRegister() 方法负责解析寄存器:

```
    bool
    ParseInstruction(ParseInstructionInfo &Info,
                     StringRef Name, SMLoc NameLoc,
                     OperandVector &Operands) override;
    bool parseRegister(MCRegister &RegNo, SMLoc &StartLoc,
                       SMLoc &EndLoc) override;
    OperandMatchResultTy
    tryParseRegister(MCRegister &RegNo, SMLoc &StartLoc,
                     SMLoc &EndLoc) override;
    bool parseRegister(MCRegister &RegNo, SMLoc &StartLoc,
                       SMLoc &EndLoc,
                       bool RestoreOnFailure);
    bool parseOperand(OperandVector &Operands,
                      StringRef Mnemonic);
    bool MatchAndEmitInstruction(
        SMLoc IdLoc, unsigned &Opcode,
        OperandVector &Operands, MCStreamer &Out,
        uint64_t &ErrorInfo,
        bool MatchingInlineAsm) override;
```

4. 内联构造函数，主要初始化所有字段。这样就完成了类声明，之后匿名命名空间也结束了:

```
public:
    M88kAsmParser(const MCSubtargetInfo &STI,
```

```
                    MCAsmParser &Parser,
                    const MCInstrInfo &MII,
                    const MCTargetOptions &Options)
        : MCTargetAsmParser(Options, STI, MII),
            Parser(Parser), SubtargetInfo(STI) {
        setAvailableFeatures(ComputeAvailableFeatures(
            SubtargetInfo.getFeatureBits()));
    }
};
```

5. 包含汇编程序的生成部分:

```
#define GET_REGISTER_MATCHER
#define GET_MATCHER_IMPLEMENTATION
#include "M88kGenAsmMatcher.inc"
```

6. 每当需要一条指令时，就调用 ParseInstruction() 方法，必须能够解析指令的所有语法形式。目前，只有接受三个操作数的指令，用逗号分隔，解析起来很简单。注意，若出现错误，返回值为 true !

```
bool M88kAsmParser::ParseInstruction(
        ParseInstructionInfo &Info, StringRef Name,
        SMLoc NameLoc, OperandVector &Operands) {
    Operands.push_back(
        M88kOperand::createToken(Name, NameLoc));
    if (getLexer().isNot(AsmToken::EndOfStatement)) {
        if (parseOperand(Operands, Name)) {
            return Error(getLexer().getLoc(),
                         "expected operand");
        }
        while (getLexer().is(AsmToken::Comma)) {
            Parser.Lex();
            if (parseOperand(Operands, Name)) {
                return Error(getLexer().getLoc(),
                "expected operand");
            }
        }
        if (getLexer().isNot(AsmToken::EndOfStatement))
            return Error(getLexer().getLoc(),
                "unexpected token in argument list");
    }
    Parser.Lex();
    return false;
}
```

7. 操作数可以是寄存器或立即寄存器。泛化一个位并解析一个表达式，而不仅仅是一个整数。这有助于以后添加地址模式。若成功，解析后的操作数可添加到操作数列表中:

```
bool M88kAsmParser::parseOperand(
        OperandVector &Operands, StringRef Mnemonic) {
```

```
    if (Parser.getTok().is(AsmToken::Percent)) {
        MCRegister RegNo;
        SMLoc StartLoc, EndLoc;
        if (parseRegister(RegNo, StartLoc, EndLoc,
                          /*RestoreOnFailure=*/false))
            return true;
        Operands.push_back(M88kOperand::createReg(
            RegNo, StartLoc, EndLoc));
        return false;
    }

    if (Parser.getTok().is(AsmToken::Integer)) {
        SMLoc StartLoc = Parser.getTok().getLoc();
        const MCExpr *Expr;
        if (Parser.parseExpression(Expr))
            return true;
        SMLoc EndLoc = Parser.getTok().getLoc();
        Operands.push_back(
            M88kOperand::createImm(Expr, StartLoc, EndLoc));
        return false;
    }
    return true;
}
```

8. parseRegister() 方法尝试解析一个寄存器。首先，检查百分号%。若后面跟着一个与寄存器名匹配的标识符，就成功地解析了一个寄存器，并在 RegNo 参数中返回寄存器号。若不能识别一个寄存器，若 RestoreOnFailure 参数为 true，就可能要撤销词法分析:

```
bool M88kAsmParser::parseRegister(
        MCRegister &RegNo, SMLoc &StartLoc, SMLoc &EndLoc,
        bool RestoreOnFailure) {
    StartLoc = Parser.getTok().getLoc();

    if (Parser.getTok().isNot(AsmToken::Percent))
        return true;
    const AsmToken &PercentTok = Parser.getTok();
    Parser.Lex();

    if (Parser.getTok().isNot(AsmToken::Identifier) ||
        (RegNo = MatchRegisterName(
            Parser.getTok().getIdentifier())) == 0) {
        if (RestoreOnFailure)
            Parser.getLexer().UnLex(PercentTok);
        return Error(StartLoc, "invalid register");
    }
    Parser.Lex();
    EndLoc = Parser.getTok().getLoc();
    return false;
```

```
}
```

9. parseRegister() 和 tryparseRegister() 覆写的方法只是之前定义的方法的包装。后一种方法，还将布尔值返回值转换为 OperandMatchResultTy 枚举的枚举成员:

```
bool M88kAsmParser::parseRegister(MCRegister &RegNo,
                                  SMLoc &StartLoc,
                                  SMLoc &EndLoc) {
    return parseRegister(RegNo, StartLoc, EndLoc,
                         /*RestoreOnFailure=*/false);
}

OperandMatchResultTy M88kAsmParser::tryParseRegister(
        MCRegister &RegNo, SMLoc &StartLoc, SMLoc &EndLoc) {
    bool Result =
        parseRegister(RegNo, StartLoc, EndLoc,

    /*RestoreOnFailure=*/true);
    bool PendingErrors = getParser().hasPendingError();
    getParser().clearPendingErrors();
    if (PendingErrors)
        return MatchOperand_ParseFail;
    if (Result)
        return MatchOperand_NoMatch;
    return MatchOperand_Success;
}
```

10. 最后，MatchAndEmitInstruction() 方法驱动解析，该方法的大部分用于发出错误消息。为了识别指令，调用 MatchInstructionImpl() 生成的方法:

```
bool M88kAsmParser::MatchAndEmitInstruction(
        SMLoc IdLoc, unsigned &Opcode,
        OperandVector &Operands, MCStreamer &Out,
        uint64_t &ErrorInfo, bool MatchingInlineAsm) {
    MCInst Inst;
    SMLoc ErrorLoc;

    switch (MatchInstructionImpl(
            Operands, Inst, ErrorInfo, MatchingInlineAsm)) {
    case Match_Success:
        Out.emitInstruction(Inst, SubtargetInfo);
        Opcode = Inst.getOpcode();
        return false;
    case Match_MissingFeature:
        return Error(IdLoc, "Instruction use requires "
                     "option to be enabled");
    case Match_MnemonicFail:
        return Error(IdLoc,
                     "Unrecognized instruction mnemonic");
```

```
    case Match_InvalidOperand: {
        ErrorLoc = IdLoc;
        if (ErrorInfo != ~0U) {
            if (ErrorInfo >= Operands.size())
                return Error(
                    IdLoc, "Too few operands for instruction");
            ErrorLoc = ((M88kOperand &)*Operands[ErrorInfo])
                    .getStartLoc();
            if (ErrorLoc == SMLoc())
                ErrorLoc = IdLoc;
        }
        return Error(ErrorLoc,
                     "Invalid operand for instruction");
    }
    default:
        break;
    }
    llvm_unreachable("Unknown match type detected!");
}
```

11. 和其他类一样，汇编解析器也有自己的工厂方法:

```
extern "C" LLVM_EXTERNAL_VISIBILITY void
LLVMInitializeM88kAsmParser() {
    RegisterMCAsmParser<M88kAsmParser> X(
        getTheM88kTarget());
}
```

这就完成了汇编解析器的实现。构建 LLVM 之后，可以使用 llvm-mc 机器码工具组装一条汇编指令:

```
$ echo 'and %r1,%r2,%r3' | \
    bin/llvm-mc --triple m88k-openbsd --show-encoding
        .text
        and %r1, %r2, %r3 | encoding: [0xf4,0x22,0x40,0x03]
```

注意，使用竖条 | 作为注释符号。这是我们在 M88kMCAsmInfo 类中配置的值。

调试汇编匹配器

要调试汇编器匹配器，可以指定--debug-only=asm-matcher 命令行选项。这有助于理解，为什么解析的指令不能与目标描述中定义的指令匹配。

下一节中，我们将向 llvm-mc 工具添加反汇编器特性。

11.7. 创建反汇编器

实现反汇编器是可选的，但实现并不需要太多的努力，并且生成反汇编表可能会捕获其他生成器未检查的编码错误。反汇编程序位于 m88kdisassemer.cpp 文件中，位于 Disassembler 子目录中:

1. 通过定义调试类型和 DecodeStatus 类型开始实现，两个都是生成代码所必需的:

```
using namespace llvm;
#define DEBUG_TYPE "m88k-disassembler"
using DecodeStatus = MCDisassembler::DecodeStatus;
```

2. M88kDisassmbler 类位于匿名命名空间中，只需要实现 getInstruction() 方法:

```
namespace {
class M88kDisassembler : public MCDisassembler {
public:
    M88kDisassembler(const MCSubtargetInfo &STI,
                     MCContext &Ctx)
        : MCDisassembler(STI, Ctx) {}
    ~M88kDisassembler() override = default;

    DecodeStatus
    getInstruction(MCInst &instr, uint64_t &Size,
                   ArrayRef<uint8_t> Bytes,
                   uint64_t Address,
                   raw_ostream &CStream) const override;
};
} // end anonymous namespace
```

3. 还需要提供一个工厂方法，在目标注册中心中注册:

```
static MCDisassembler *
createM88kDisassembler(const Target &T,
                       const MCSubtargetInfo &STI,
                       MCContext &Ctx) {
    return new M88kDisassembler(STI, Ctx);
}

extern "C" LLVM_EXTERNAL_VISIBILITY void
LLVMInitializeM88kDisassembler() {
    TargetRegistry::RegisterMCDisassembler(
        getTheM88kTarget(), createM88kDisassembler);
}
```

4. decodeGPRRegisterClass() 函数将寄存器号转换为 TableGen 生成的寄存器枚举成员。这是 M88kInstPrinter::getMachineOpValue() 方法的逆操作。注意，M88kRegisterOperand 类的 DecoderMethod 字段中指定了这个函数的名称:

```
static const uint16_t GPRDecoderTable[] = {
    M88k::R0, M88k::R1, M88k::R2, M88k::R3,
    // ...
};

static DecodeStatus
decodeGPRRegisterClass(MCInst &Inst, uint64_t RegNo,
                       uint64_t Address,
```

```
                            const void *Decoder) {
    if (RegNo > 31)
        return MCDisassembler::Fail;

    unsigned Register = GPRDecoderTable[RegNo];
    Inst.addOperand(MCOperand::createReg(Register));
    return MCDisassembler::Success;
}
```

5. 然后，包括生成的反汇编表:

```
#include "M88kGenDisassemblerTables.inc"
```

6. 最后，解码指令，需要获取 bytes 数组中接下来的四个字节，从中创建指令编码，并调用 decodeInstruction() 生成的函数:

```
DecodeStatus M88kDisassembler::getInstruction(
        MCInst &MI, uint64_t &Size, ArrayRef<uint8_t> Bytes,
        uint64_t Address, raw_ostream &CS) const {
    if (Bytes.size() < 4) {
        Size = 0;
        return MCDisassembler::Fail;
    }
    Size = 4;

    uint32_t Inst = 0;
    for (uint32_t I = 0; I < Size; ++I)
        Inst = (Inst << 8) | Bytes[I];

    if (decodeInstruction(DecoderTableM88k32, MI, Inst,
                          Address, this, STI) !=
        MCDisassembler::Success) {
        return MCDisassembler::Fail;
    }
    return MCDisassembler::Success;
}
```

这就是反汇编程序需要做的全部工作。编译 LLVM 后，可以使用 llvm-mc 工具再次测试功能:

```
$ echo "0xf4,0x22,0x40,0x03" | \
    bin/llvm-mc --triple m88k-openbsd -disassemble
          .text
          and %r1, %r2, %r3
```

此外，现在可以使用 llvm-objdump 工具来反汇编 ELF 文件。然而，为了使它真正有用，需要将所有指令添加到目标描述中。

11.8. 总结

本章中，了解了如何创建 LLVM 目标描述，并开发了一个简单的后端目标，该目标支持 LLVM 指令的汇编和反汇编。首先收集了所需的文档，并通过增强 Triple 类使 LLVM 了解新的架构。该文档还包括 ELF 文件格式的重定位定义，并将对它们的支持添加到 LLVM 中。

然后，了解了目标描述中的寄存器定义和指令定义，并使用生成的 C++ 源代码来实现指令汇编和反汇编程序。

下一章，我们将在后端添加代码生成功能。

第 12 章 指令选择

任何后端的核心都是指令选择，LLVM 实现了几种方法。本章中，我们将通过选择有向无环图 (DAG) 和全局指令选择来实现指令选择。

本章中，将学习以下主题:

- 定义调用约定的规则: 如何在目标描述中，描述调用规则
- 通过选择 DAG 进行指令选择: 如何使用图数据结构实现指令选择
- 添加寄存器和指令信息: 如何访问目标描述中的信息，以及需要提供哪些信息
- 将空框架放置到位: 介绍堆栈布局和函数的序言
- 生成机器指令: 如何将机器指令最终写入目标文件或汇编文本
- 创建目标计算机和子目标: 如何配置后端
- 全局指令选择: 指令选择的另一种方法
- 进一步扩展后端: 提供了一些关于后续步骤的指导

本章结束时，将了解如何创建一个 LLVM 后端来翻译简单的指令。还将获得通过选择 DAG 和全局指令选择开发指令选择的知识，并且将熟悉为使指令选择工作而必须实现的支持类。

12.1. 定义调用约定规则

实现调用约定的规则是，将 LLVM 中间表示 (IR) 降为机器码的重要组成部分，基本规则可以在目标描述中定义。

大多数调用约定遵循一个基本模式: 定义一个寄存器子集用于参数传递。若这个子集没有耗尽，则在下一个空闲寄存器中传递下一个参数。若没有空闲的寄存器，则将值传递到堆栈上。这可以通过循环遍历参数，决定如何将每个参数传递给被调用函数来实现，同时跟踪使用的寄存器。LLVM 中，这个循环是在框架内实现的，状态保存在一个名为 CCState 的类中，所以可在目标描述中定义规则。

这些规则是作为一系列条件给出的。若条件成立，则执行一个操作。根据该操作的结果，要么为参数找到一个位置，要么计算下一个条件。例如，32 位整数在寄存器中传递。条件是类型检查，动作是将寄存器赋值给该参数。目标描述如下所示:

```
1 CCIfType<[i32],
2         CCAssignToReg<[R2, R3, R4, R5, R6, R7, R8, R9]>>,
```

当然，若调用的函数有超过 8 个参数，寄存器就会耗尽，操作就会失败。剩下的参数在堆栈上传递，可以将其指定为下一个动作:

```
1 CCAssignToStack<4, 4>
```

第一个参数是以字节为单位的堆栈槽的大小，而第二个参数是对齐方式。它是一个包罗万象的规则，所以没有任何限制条件。

12.1.1. 执行调用约定规则

对于调用约定，需要注意更多预定义的条件和操作。例如，CCIfInReg 检查参数是否标记为 inreg 属性，若函数具有可变参数列表，CCIfVarArg 计算结果为 true。CCPromoteToType 动作将参数的类型提升为更大的类型，CCPassIndirect 动作表明参数值应该存储在堆栈中，并且指向该存储的指针作为普通参数传递。所有预定义的条件和动作都可以在 llvm/include/llvm/Target/TargetCallingConv.td 中引用。

参数和返回值都以这种方式定义，将把定义放入 M88kCallingConv.td 文件中:

1. 首先，必须为参数定义规则。为了简化编码，只考虑 32 位值:

```
def CC_M88k : CallingConv<[
    CCIfType<[i8, i16], CCPromoteToType<i32>>,
    CCIfType<[i32,f32],
        CCAssignToReg<[R2, R3, R4, R5, R6, R7, R8, R9]>>,
    CCAssignToStack<4, 4>
]>;
```

2. 之后，必须定义返回值的规则:

```
def RetCC_M88k : CallingConv<[
    CCIfType<[i32], CCAssignToReg<[R2]>>
]>;
```

3. 最后，必须定义调用者保存的寄存器序列。使用序列操作符来生成寄存器序列，而不是将它们写下来:

```
def CSR_M88k :
    CalleeSavedRegs<(add R1, R30,
        (sequence "R%d", 25, 14))>;
```

目标描述中为调用约定定义规则的好处是，可以使用各种指令选择方法重用。接下来，通过选择 DAG 查看指令的选择。

12.2. 通过 DAG 进行指令选择

后端从 IR 创建机器指令是一项非常重要的任务。实现它的一种常见方法是利用 DAG:

1. 首先，必须从 IR 创建 DAG。DAG 的一个节点表示一个操作，边缘表示控制和数据流依赖关系。
2. 接下来，必须遍历 DAG 并使类型和操作合法化，只使用硬件支持的类型和操作。这要求我们创建一个配置，告诉框架如何处理非合法类型和操作。例如，一个 64 位值可以拆分为两个 32 位值，两个 64 位值的乘法可以更改为库调用，计数填充等复杂操作可以扩展为计算该值的一系列更简单的操作。
3. 然后，利用模式匹配对 DAG 中的节点进行匹配，并用机器指令替换节点。我们在前一章中看到了这种模式。
4. 最后，指令调度程序将机器指令重新排序为更高效的顺序。

通过选择 DAG 对指令选择过程的高级描述若对更多细节感兴趣，可以在https://llvm.org/docs/CodeGenerator.html#selectiondaginstruction-selection-process的LLVM 目标无关代码生成器用户指南中找到。

此外，LLVM 中的所有后端都实现了选择 DAG，其主要优点是可以生成高性能的代码，但这是有代价的: 创建 DAG 很昂贵，并且减慢了编译速度此，所以这促使 LLVM 开发人员寻找其他更理想的方法。一些目标通过 FastISel 实现指令选择，这只用于未优化的代码。可以快速生成代码，但生成的代码不如选择 DAG 方法生成的代码。还增加了一种全新的指令选择方法，使测试工作量增加了一倍。指令选择还使用另一种方法，称为全局指令选择，我们将在稍后的全局指令选择一节中对其进行研究。

本章中，目标是实现足够的后端，来实现一个简单的 IR 功能:

```
define i32 @f1(i32 %a, i32 %b) {
    %res = and i32 %a, %b
    ret i32 %res
}
```

此外，对于一个真正的后端，需要更多的代码，必须添加什么来实现更大的功能。

为了通过选择 DAG 实现指令选择，需要创建两个新类:M88kISelLowering 和 M88kDAGToDAGISel。前一个类用于定制 DAG，例如：通过定义哪些类型是合法的，包含支持降低函数和函数调用的代码。后一个类执行 DAG 转换，实现主要是从目标描述生成的。

我们将在后端添加几个类的实现，图 12.1 描述了将进一步开发的主类之间的高层关系:

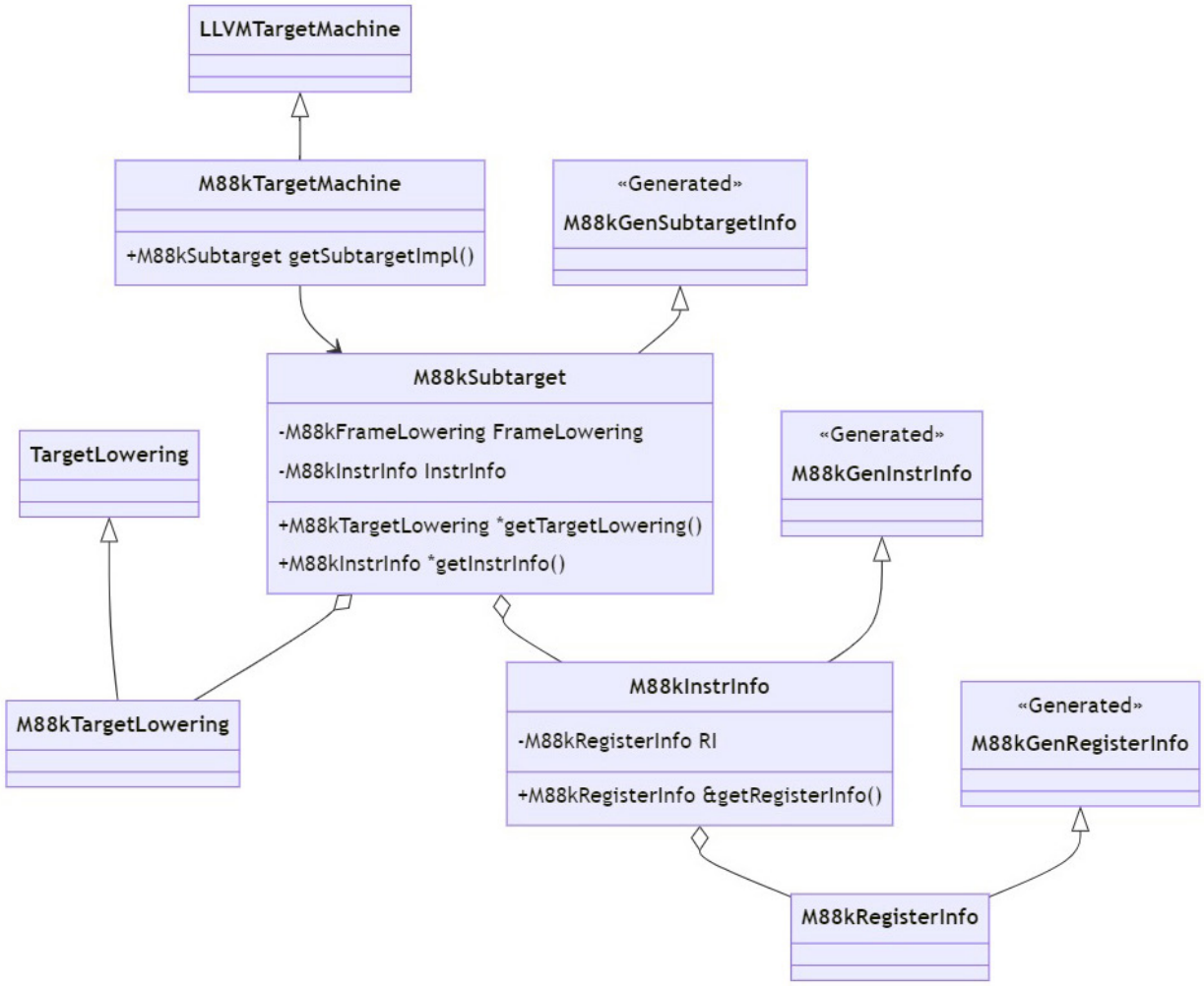


图 12.1 - 主类之间的关系

12.2.1. 简化 DAG——处理合法类型和设置操作

首先，实现 M88kISelLowering，该类存储在 M88kISelLowering.cpp 文件中。构造函数配置合法的类型和操作:

1. 构造函数将对 TargetMachine 和 M88kSubtarget 类的引用作为参数。TargetMachine 类负责目标的一般配置，需要运行。LLVM 后端通常针对一个 CPU 系列，M88kSubtarget 类描述所选 CPU 的特性。我们将在本章后面讨论这两个类:

```
M88kTargetLowering::M88kTargetLowering(
    const TargetMachine &TM, const M88kSubtarget &STI)
    : TargetLowering(TM), Subtarget(STI) {
```

2. 第一个动作是声明哪个机器值类型使用哪个寄存器类，寄存器类从目标描述中生成。这里，只处理 32 位的值:

```
    addRegisterClass(MVT::i32, &M88k::GPRRegClass);
```

3. 添加了所有的寄存器类之后，必须计算这些寄存器类的派生属性。需要查询子目标以获取寄

存器信息，这些信息大多由目标描述生成:

```
computeRegisterProperties(Subtarget.getRegisterInfo());
```

4. 接下来，必须声明哪个寄存器包含堆栈指针:

```
setStackPointerRegisterToSaveRestore(M88k::R31);
```

5. 布尔值在不同平台上的表示方式不同。对于我们的目标，声明一个布尔值存储在 0 位，其他位将清除:

```
setBooleanContents(ZeroOrOneBooleanContent);
```

6. 之后，设置函数的对齐方式。最小函数对齐是正确执行所需的对齐，给出了首选的对齐方式:

```
setMinFunctionAlignment(Align(4));
setPrefFunctionAlignment(Align(4));
```

7. 最后，声明哪些操作是合法的。前一章中，只定义了三个逻辑指令，对 32 位值是合法的:

```
setOperationAction(ISD::AND, MVT::i32, Legal);
setOperationAction(ISD::OR, MVT::i32, Legal);
setOperationAction(ISD::XOR, MVT::i32, Legal);
```

8. 除了 Legal 之外，还可以使用其他几个动作。Promote 扩大类型，Expand 用其他操作替换操作，LibCall 将操作降低为库调用，而 Custom 调用 LowerOperation() 钩子方法，就可以实现自定义处理。例如，在 M88k 架构中，没有计数指令，因此要求将此操作扩展为其他操作:

```
    setOperationAction(ISD::CTPOP, MVT::i32, Expand);
}
```

现在，在 M88kInstrInfo.td 文件中提到的目标描述中，我们用 and 助记符定义了一条机器指令，并为其添加了一个模式。人若扩展 AND 多类记录，并且只查看使用三个寄存器的指令，可以得到 TableGen 的定义:

```
let isCommutable = 1 in
    def ANDrr : F_LR<0b01000, Func, /*comp=*/0b0, "and",
                     [(set i32:$rd,
                         (and GPROpnd:$rs1, GPROpnd:$rs2))]>;
```

“and”字符串是指令的助记符。C++ 源代码中，使用 M88k::ANDrr 来引用这个指令。模式内部，使用 DAG 和节点类型。C++ 中，会命名为 ISD::AND，在 setOperationAction() 方法中使用。指令选择期间，若模式匹配，则和类型的 DAG 节点可使用 M88k::ANDrr 指令替换，其中包括输入操作数。进行指令选择时，最重要的任务是定义正确的合法化行为，并将模式附加到指令定义中。

12.2.2. 向下转译 DAG——处理形参

转到 M88kISelLowering 类执行的另一个重要任务。我们在前一节中定义了调用约定的规则，但是还需要将物理寄存器和内存位置映射到 DAG 中使用的虚拟寄存器。对于参数，这在

LowerFormalArguments() 方法中完成，返回值在 LowerReturn() 方法中处理。首先，我们必须处理实参:

1. 将从包含生成的源代码开始:

```
#include "M88kGenCallingConv.inc"
```

2. LowerFormalArguments() 方法接受几个参数。SDValue 类表示与 DAG 节点关联的值，处理 DAG 时经常使用。第一个参数 Chain 表示控制流，可能更新的 Chain 也是该方法的返回值。CallConv 参数标识所使用的调用约定，若变量参数列表是参数的一部分，则 IsVarArg 设置为 true。需要处理的参数在 Ins 参数中传递，以及在 DL 参数中的位置，DAG 参数能够访问 SelectionDAG 类。最后，映射的结果将存储在 InVals vector 参数中:

```
SDValue M88kTargetLowering::LowerFormalArguments(
    SDValue Chain, CallingConv::ID CallConv,
    bool IsVarArg,
    const SmallVectorImpl<ISD::InputArg> &Ins,
    const SDLoc &DL, SelectionDAG &DAG,
    SmallVectorImpl<SDValue> &InVals) const {
```

3. 我们的第一个动作是检索对 machine 函数和 machine 寄存器信息的引用:

```
    MachineFunction &MF = DAG.getMachineFunction();
    MachineRegisterInfo &MRI = MF.getRegInfo();
```

4. 接下来，必须调用生成的代码。需要实例化 CCState 类的对象，对 AnalyzeFormalArguments() 方法的调用中，使用的 CC_M88k 参数值是在目标描述中使用的约定名称。结果存储在 arglos vector 中:

```
    SmallVector<CCValAssign, 16> ArgLocs;
    CCState CCInfo(CallConv, IsVarArg, MF, ArgLocs,
        *DAG.getContext());
    CCInfo.AnalyzeFormalArguments(Ins, CC_M88k);
```

5. 当确定了参数的位置，需要将它们映射到 DAG，必须遍历所有位置:

```
    for (unsigned I = 0, E = ArgLocs.size(); I != E; ++I) {
        SDValue ArgValue;
        CCValAssign &VA = ArgLocs[I];
        EVT LocVT = VA.getLocVT();
```

6. 映射取决于确定的位置，处理分配给寄存器的参数。目标是将物理寄存器复制到虚拟寄存器，需要确定正确的寄存器类。因为只处理 32 位的值，所以这样做很容易:

```
        if (VA.isRegLoc()) {
            const TargetRegisterClass *RC;
            switch (LocVT.getSimpleVT().SimpleTy) {
            default:
                llvm_unreachable("Unexpected argument type");
            case MVT::i32:
                RC = &M88k::GPRRegClass;
```

```
                break;
            }
```

7. 使用存储在 RC 变量中的寄存器类，可以创建虚拟寄存器并复制值。还需要将物理寄存器声明为“常驻”(live-in):

```
            Register VReg = MRI.createVirtualRegister(RC);
            MRI.addLiveIn(VA.getLocReg(), VReg);
            ArgValue =
                DAG.getCopyFromReg(Chain, DL, VReg, LocVT);
```

8. 调用约定的定义中，增加了 8 位和 16 位值提升为 32 位的规则，需要确保这里的提升。必须插入 DAG 节点，以确保提升该值，该值会截断为正确的大小。注意，将 ArgValue 的值作为操作数传递给 DAG 节点，并将结果存储在同一个变量中:

```
            if (VA.getLocInfo() == CCValAssign::SExt)
                ArgValue = DAG.getNode(
                    ISD::AssertSext, DL, LocVT, ArgValue,
                    DAG.getValueType(VA.getValVT()));
            else if (VA.getLocInfo() == CCValAssign::ZExt)
                ArgValue = DAG.getNode(
                    ISD::AssertZext, DL, LocVT, ArgValue,
                    DAG.getValueType(VA.getValVT()));
            if (VA.getLocInfo() != CCValAssign::Full)
                ArgValue = DAG.getNode(ISD::TRUNCATE, DL,
                                       VA.getValVT(), ArgValue);
```

9. 最后，通过将 DAG 节点添加到结果 vector 来完成对寄存器参数的处理:

```
            InVals.push_back(ArgValue);
        }
```

10. 参数的另一个可能位置是在堆栈上，但没有定义加载和存储指令，还不能处理这种情况。这将结束所有参数位置的循环:

```
        } else {
            llvm_unreachable("Not implemented");
        }
    }
```

11. 之后，可能需要添加代码来处理变量参数列表。再一次，添加了一些代码，来提醒我们没有实现它:

```
    assert(!IsVarArg && "Not implemented");
```

12. 最后，必须返回 Chain 参数:

```
    return Chain;
}
```

12.2.3. 向下转译 DAG——处理返回值

返回值的处理方式类似，但必须扩展目标描述。首先，需要定义一个名为 RET_GLUE 的新 DAG 节点类型。这种 DAG 节点类型用于将返回值粘合在一起，从而防止重新排列，例如由指令调度程序重新排列。M88kInstrInfo.td 中的定义如下所示:

```
def retglue : SDNode<"M88kISD::RET_GLUE", SDTNone,
                [SDNPHasChain, SDNPOptInGlue, SDNPVariadic]>;
```

同一个文件中，还定义了一个伪指令来表示函数调用的返回，将用于 RET_GLUE 节点:

```
let isReturn = 1, isTerminator = 1, isBarrier = 1,
        AsmString = "RET" in
    def RET : Pseudo<(outs), (ins), [(retglue)]>;
```

我们将在生成输出时展开这个伪指令。

有了这些定义，就可以实现 LowerReturn() 方法了:

1. 参数与 LowerFormalArguments() 相同，只是顺序略有不同:

```
SDValue M88kTargetLowering::LowerReturn(
    SDValue Chain, CallingConv::ID CallConv,
    bool IsVarArg,
    const SmallVectorImpl<ISD::OutputArg> &Outs,
    const SmallVectorImpl<SDValue> &OutVals,
    const SDLoc &DL, SelectionDAG &DAG) const {
```

2. 首先，调用生成的代码，这次使用 RetCC_M88k 的调用约定:

```
    SmallVector<CCValAssign, 16> RetLocs;
    CCState RetCCInfo(CallConv, IsVarArg,
                      DAG.getMachineFunction(), RetLocs,
                      *DAG.getContext());
    RetCCInfo.AnalyzeReturn(Outs, RetCC_M88k);
```

3. 然后，再次遍历这些位置。根据当前调用约定的简单定义，该循环最多执行一次。若增加对返回 64 位值的支持，这将会改变，这需要在两个寄存器中返回:

```
    SDValue Glue;
    SmallVector<SDValue, 4> RetOps(1, Chain);
    for (unsigned I = 0, E = RetLocs.size(); I != E; ++I) {
        CCValAssign &VA = RetLocs[I];
```

4. 之后，将返回值复制到分配给返回值的物理寄存器中。这与处理参数非常相似，除了使用 Glue 变量将值粘合在一起:

```
    Register Reg = VA.getLocReg();
    Chain = DAG.getCopyToReg(Chain, DL, Reg, OutVals[I], Glue);
    Glue = Chain.getValue(1);
    RetOps.push_back(
```

```
        DAG.getRegister(Reg, VA.getLocVT()));
}
```

5. 返回值是链和胶合寄存器的复制操作，后者只有在有值返回时才会返回:

```
    RetOps[0] = Chain;
    if (Glue.getNode())
        RetOps.push_back(Glue);
```

6. 最后，构造一个 RET_GLUE 类型的 DAG 节点，传递必要的值:

```
    return DAG.getNode(M88kISD::RET_GLUE, DL, MVT::Other,
                       RetOps);
}
```

恭喜! 有了这些定义，就为指令选择奠定了基础。

12.2.4. 指令选择中实现 DAG 到 DAG 的转换

目前，仍然缺少一个关键部分: 需要定义执行目标描述中定义的 DAG 转换的传递。该类名为 M88kDAGToDAGISel，存储在 M88kISelDAGToDAG.cpp 文件中。类的大部分都生成了，但仍然需要添加一些代码:

1. 我们将首先定义调试类型并为该传递提供描述性名称:

```
#define DEBUG_TYPE "m88k-isel"
#define PASS_NAME
            "M88k DAG->DAG Pattern Instruction Selection"
```

2. 然后，必须在匿名命名空间中声明该类。只重写 Select() 方法，其他代码会生成并包含在类的主体中:

```
class M88kDAGToDAGISel : public SelectionDAGISel {
public:
    static char ID;

    M88kDAGToDAGISel(M88kTargetMachine &TM,
                     CodeGenOpt::Level OptLevel)
        : SelectionDAGISel(ID, TM, OptLevel) {}

    void Select(SDNode *Node) override;

#include "M88kGenDAGISel.inc"
};
} // end anonymous namespace
```

3. 之后，必须添加初始化传递的代码。LLVM 后端仍然使用遗留的通道管理器，其设置与用于 IR 转换的通道管理器不同，静态成员 ID 值用于标识通道。初始化通道可以使用 INITIALIZE_PASS 宏来实现，该宏扩展为 C++ 代码。还必须添加一个工厂方法来创建实例:

```
char M88kDAGToDAGISel::ID = 0;

INITIALIZE_PASS(M88kDAGToDAGISel, DEBUG_TYPE, PASS_NAME,
                false, false)

FunctionPass *
llvm::createM88kISelDag(M88kTargetMachine &TM,
                        CodeGenOpt::Level OptLevel) {
    return new M88kDAGToDAGISel(TM, OptLevel);
}
```

4. 最后，必须实现 Select() 方法。只调用生成的代码，但遇到了一个复杂的转换，就不能用 DAG 模式来表达，则可以在调用生成的代码之前，添加自己的代码来执行转换:

```
void M88kDAGToDAGISel::Select(SDNode *Node) {
    SelectCode(Node);
}
```

这样，就实现了指令选择。进行第一次测试之前，仍然需要添加一些支持类。我们将在接下来的几节中研究这些类。

12.3. 添加寄存器和指令信息

目标描述捕获有关寄存器和指令的大部分信息。要访问该信息，必须实现 M88kRegisterInfo 和 M88kInstrInfo 类。这些类还包含钩子，可以覆盖这些钩子来完成那些太复杂，而无法在目标描述中表达的任务。从 M88kRegisterInfo 类开始，在 M88kRegisterInfo.h 文件中声明:

1. 头文件首先包含从目标描述生成的代码:

```
#define GET_REGINFO_HEADER
#include "M88kGenRegisterInfo.inc"
```

2. 之后，必须在 llvm 命名空间中声明 M88kRegisterInfo 类。我们只重写了几个方法:

```
namespace llvm {
struct M88kRegisterInfo : public M88kGenRegisterInfo {
    M88kRegisterInfo();

    const MCPhysReg *getCalleeSavedRegs(
        const MachineFunction *MF) const override;

    BitVector getReservedRegs(
        const MachineFunction &MF) const override;

    bool eliminateFrameIndex(
        MachineBasicBlock::iterator II, int SPAdj,
        unsigned FIOperandNum,
        RegScavenger *RS = nullptr) const override;
```

```
15
16      Register getFrameRegister(
17          const MachineFunction &MF) const override;
18  };
19  } // end namespace llvm
```

类的定义存储在 M88kRegisterInfo.cpp 文件中:

1. 同样，该文件以包含从目标描述生成的代码开始:

```
1  #define GET_REGINFO_TARGET_DESC
2  #include "M88kGenRegisterInfo.inc"
```

2. 构造函数初始化超类，将保存返回地址的寄存器作为参数传递:

```
1  M88kRegisterInfo::M88kRegisterInfo()
2      : M88kGenRegisterInfo(M88k::R1) {}
```

3. 然后，调用者保存实现返回的寄存器列表的方法。我们在目标描述中定义了列表，并且只返回该列表:

```
1  const MCPhysReg *M88kRegisterInfo::getCalleeSavedRegs(
2          const MachineFunction *MF) const {
3      return CSR_M88k_SaveList;
4  }
```

4. 之后，处理保留寄存器，保留的寄存器取决于平台和硬件。r0 寄存器包含一个常量值 0，因此将其视为保留寄存器，r28 和 r29 寄存器始终保留供链接器使用。最后，r31 寄存器用作堆栈指针。这个列表可能取决于函数，由于这个动态行为无法生成:

```
1  BitVector M88kRegisterInfo::getReservedRegs(
2          const MachineFunction &MF) const {
3      BitVector Reserved(getNumRegs());
4      Reserved.set(M88k::R0);
5      Reserved.set(M88k::R28);
6      Reserved.set(M88k::R29);
7      Reserved.set(M88k::R31);
8      return Reserved;
9  }
```

5. 若果需要帧寄存器，则使用 r30。注意，代码还不支持创建帧。若函数需要一个帧，那么 r30 也必须在 getReservedRegs() 方法中标记为保留。因为它在超类中声明为纯虚，所以必须实现这个方法:

```
1  Register M88kRegisterInfo::getFrameRegister(
2          const MachineFunction &MF) const {
3      return M88k::R30;
4  }
```

6. 类似地，因为其为纯虚函数，所以需要实现 eliminateFrameIndex() 方法。调用它是为了将操作数中的帧索引替换为正确的值，以便对堆栈上的值进行寻址:

```
bool M88kRegisterInfo::eliminateFrameIndex(
        MachineBasicBlock::iterator MI, int SPAdj,
        unsigned FIOperandNum, RegScavenger *RS) const {
    return false;
}
```

M88kInstrInfo 类有许多钩子方法，可以覆盖它们来完成特殊任务。例如，用于分支分析和重新具体化。现在，只重写 expandPostRAPseudo() 方法，这个方法中扩展了伪指令 RET。

1. 头文件首先包含生成的代码:

```
#define GET_INSTRINFO_HEADER
#include "M88kGenInstrInfo.inc"
```

2. M88kInstrInfo 类派生自生成的 m88kgenstrinfo 类。除了覆写 expandPostRAPseudo() 方法之外，唯一的其他添加是该类拥有先前定义的类 M88kRegisterInfo 的实例:

```
namespace llvm {
class M88kInstrInfo : public M88kGenInstrInfo {
    const M88kRegisterInfo RI;
    [[maybe_unused]] M88kSubtarget &STI;

    virtual void anchor();

public:
    explicit M88kInstrInfo(M88kSubtarget &STI);

    const M88kRegisterInfo &getRegisterInfo() const {
        return RI;
    }

    bool
    expandPostRAPseudo(MachineInstr &MI) const override;
} // end namespace llvm
```

实现存储在 M88kInstrInfo.cpp 类中:

1. 与头文件一样，实现从包含生成的代码开始:

```
#define GET_INSTRINFO_CTOR_DTOR
#define GET_INSTRMAP_INFO
#include "M88kGenInstrInfo.inc"
```

2. 然后，定义 anchor() 方法，用于将虚参表固定到该文件:

```
void M88kInstrInfo::anchor() {}
```

3. 最后，在 expandPostRAPseudo() 方法中展开 RET。这个方法是在寄存器分配器运行后调用的，目的是扩展伪指令，伪指令可能仍然与机器码混合在一起。若机器指令 MI 的操作码是伪指令 RET，则必须插入 jmp %r1 跳转指令，该指令是退出函数的指令。复制所有表示要返回值的隐

式操作数，并删除伪指令。若在代码生成过程中需要其他伪指令，则可以扩展这个函数来扩展:

```
bool M88kInstrInfo::expandPostRAPseudo(
        MachineInstr &MI) const {
    MachineBasicBlock &MBB = *MI.getParent();

    switch (MI.getOpcode()) {
        default:
            return false;
        case M88k::RET: {
            MachineInstrBuilder MIB =
            BuildMI(MBB, &MI, MI.getDebugLoc(),
                    get(M88k::JMP))
                .addReg(M88k::R1, RegState::Undef);

            for (auto &MO : MI.operands()) {
                if (MO.isImplicit())
                    MIB.add(MO);
            }
            break;
    }
    }
    MBB.erase(MI);
    return true;
}
```

这两个类都有最小的实现。若继续开发目标，需要覆写更多的方法。TargetInstrInfo 和 TargetRegisterInfo 基类中的注释值得一读，可以在 llvm/include/llvm/CodeGen 目录中找到。

还需要更多的类来运行指令选择。接下来，我们将了解帧的向下转译。

12.4. 向下转译空帧

平台的二进制接口不仅定义了参数的传递方式，还包括如何布局堆栈帧: 哪里存储局部变量，将寄存器溢出到哪里等。函数的开始和结束处需要一个特殊的指令序列，称为序言和尾声。当前的开发状态下，目标不支持创建序言和尾部声明的机器指令，但用于指令选择的帧代码要求 TargetFrameLowering 的子类可用。简单的解决方案是为 M88kFrameLowering 类提供一个空的实现。

该类的声明在 M88kFrameLowering.h 文件中，我们要做的就是重写纯虚函数:

```
namespace llvm {
class M88kFrameLowering : public TargetFrameLowering {
public:
    M88kFrameLowering();

    void
    emitPrologue(MachineFunction &MF,
    MachineBasicBlock &MBB) const override;
```

```
9
10      void
11      emitEpilogue(MachineFunction &MF,
12                   MachineBasicBlock &MBB) const override;
13      bool hasFP(const MachineFunction &MF) const override;
14  };
15  }
```

该实现存储在 M88kFrameLowering.cpp 文件中，提供了关于构造函数中堆栈帧的一些基本细节。堆栈向下扩展到更小的地址，并以 8 字节的边界对齐。当调用函数时，局部变量直接存储在调用函数的堆栈指针下面，因此局部区域的偏移量为 0。即使在函数调用期间，堆栈也应该保持在 8 字节的边界上对齐。最后一个参数表示不能重新排列堆栈。其他函数只有一个空实现:

```
1   M88kFrameLowering::M88kFrameLowering()
2       : TargetFrameLowering(
3           TargetFrameLowering::StackGrowsDown, Align(8),
4           0, Align(8), false /* StackRealignable */) {}
5
6   void M88kFrameLowering::emitPrologue(
7       MachineFunction &MF, MachineBasicBlock &MBB) const {}
8
9   void M88kFrameLowering::emitEpilogue(
10      MachineFunction &MF, MachineBasicBlock &MBB) const {}
11
12  bool M88kFrameLowering::hasFP(
13      const MachineFunction &MF) const { return false; }
```

当然，随着我们的实现的增长，这个类将是第一个需要完全实现的类。

将所有部件组装在一起之前，需要实现汇编输出器，用于生成机器指令。

12.5. 发出机器指令

指令选择从 LLVM IR 中创建机器指令，由 MachineInstr 类表示，但这并不是结束。MachineInstr 类的实例仍然携带额外的信息，比如标签或标志。为了通过机器码组件发出一条指令，需要将 MachineInstr 的实例降低为 MCInst 的实例。机器代码组件，提供了将指令写入目标文件或将其作为汇编文本输出的功能。M88kAsmPrinter 类负责发出整个编译单元，向下转译指令委托给 M88kMCInstLower 类。

汇编输出器是后端中运行的最后一个通道，其实现存储在 M88kAsmPrinter.cpp 文件中:

1. M88kAsmPrinter 类的声明位于匿名命名空间中。除了构造函数，只重写 getPassName() 函数和 emitInstruction() 函数，getPassName() 函数以人类可读的字符串形式返回通道的名称:

```
1   namespace {
2   class M88kAsmPrinter : public AsmPrinter {
3   public:
4       explicit M88kAsmPrinter(
```

```
        TargetMachine &TM,
        std::unique_ptr<MCStreamer> Streamer)
        : AsmPrinter(TM, std::move(Streamer)) {}

    StringRef getPassName() const override {
        return "M88k Assembly Printer";
    }

    void emitInstruction(const MachineInstr *MI) override;
};
} // end of anonymous namespace
```

2. 像许多其他类一样，必须在目标注册表中注册汇编输出器:

```
extern "C" LLVM_EXTERNAL_VISIBILITY void
LLVMInitializeM88kAsmPrinter() {
    RegisterAsmPrinter<M88kAsmPrinter> X(
        getTheM88kTarget());
}
```

3. emitInstruction() 方法负责将机器指令 (MI) 发送到输出流，将指令的向下转译委托给 M88kMCInstLower 类:

```
void M88kAsmPrinter::emitInstruction(
        const MachineInstr *MI) {
    MCInst LoweredMI;
    M88kMCInstLower Lower(MF->getContext(), *this);
    Lower.lower(MI, LoweredMI);
    EmitToStreamer(*OutStreamer, LoweredMI);
}
```

基类 AsmPrinter 提供了许多可以覆盖的有用钩子，emitStartOfAsmFile() 方法在事件触发之前调用，而 emitEndOfAsmFile() 在所有事件触发之后调用，这些方法可以生成有针对性的数据或代码文件的开始和结束。类似地，emitFunctionBodyStart() 和 emitFunctionBodyEnd() 方法也会在函数体发出前后调用。请阅读 llvm/include/llvm/CodeGen/AsmPrinter.h 文件中的注释，了解可以自定义哪些内容。

M88kMCInstLower 类向下转译操作数和指令，实现包含两个用于此目的的方法。声明在 M88kMCInstLower.h 文件中:

```
class LLVM_LIBRARY_VISIBILITY M88kMCInstLower {
public:
    void lower(const MachineInstr *MI, MCInst &OutMI) const;
    MCOperand lowerOperand(const MachineOperand &MO) const;
};
```

该定义在 M88kMCInstLower.cpp 文件中:

1. 为了将 MachineOperand 降格为 MCOperand，需要检查操作数类型。这里，只通过提供原始的

MachineOperand 值来创建 mcoperand 等效寄存器和立即值，从而处理寄存器和立即值。当表达式作为操作数引入，这个方法就需要增强了:

```
MCOperand M88kMCInstLower::lowerOperand(
        const MachineOperand &MO) const {
    switch (MO.getType()) {
    case MachineOperand::MO_Register:
        return MCOperand::createReg(MO.getReg());

    case MachineOperand::MO_Immediate:
        return MCOperand::createImm(MO.getImm());

    default:
        llvm_unreachable("Operand type not handled");
    }
}
```

2. 指令的向下转译也是类似的。首先，复制操作码，然后处理操作数。MachineInstr 的实例可以添加隐式操作数，这些操作数不会向下转译，需要过滤:

```
void M88kMCInstLower::lower(const MachineInstr *MI,
MCInst &OutMI) const {
    OutMI.setOpcode(MI->getOpcode());
    for (auto &MO : MI->operands()) {
        if (!MO.isReg() || !MO.isImplicit())
            OutMI.addOperand(lowerOperand(MO));
    }
}
```

这样，就实现了汇编输出器。现在，需要把所有的碎片拼凑起来。我们将在下一节中进行此操作。

12.6. 创建目标机器和子目标

我们已经实现了指令选择类和一些其他类。现在，需要设置后端如何工作。与优化流水线一样，后端也分为几个通道。配置这些通道是 M88kTargetMachine 类的主要任务，所以需要指定哪些特性可用于指令选择。通常，平台是一系列 CPU，它们都有一组通用的指令，但因特定的扩展而有所不同。例如，一些 CPU 有向量指令，而另一些没有。在 LLVM IR 中，函数可以附加属性，这些属性指定应该为哪个 CPU 编译该函数，或者可以使用哪些特性。换句话说，每个函数可以有不同的配置，这是在 M88kSubTarget 类中捕获的。

12.6.1. 实现 M88kSubtarget

首先实现 M88kSubtarget 类，声明存储在 M88kSubtarget.h 类中:

1. 子目标的部分代码由目标描述生成，首先包含这些代码:

```
#define GET_SUBTARGETINFO_HEADER
#include "M88kGenSubtargetInfo.inc"
```

2. 然后，声明这个类，从生成的 M88kGenSubtargetInfo 类派生。类拥有两个先前定义的类——指令信息、目标向下转译类和帧向下转译类:

```
namespace llvm {
class StringRef;
class TargetMachine;
class M88kSubtarget : public M88kGenSubtargetInfo {
    virtual void anchor();

    Triple TargetTriple;
    M88kInstrInfo InstrInfo;
    M88kTargetLowering TLInfo;
    M88kFrameLowering FrameLowering;
```

3. 子目标是用目标三元组、CPU 名称和一个特征字符串，以及目标机器初始化的。所有这些参数描述了后端为硬件生成的代码:

```
public:
    M88kSubtarget(const Triple &TT,
                  const std::string &CPU,
                  const std::string &FS,
                  const TargetMachine &TM);
```

4. 接下来，会再次包含生成的文件，这次是为了为目标描述中定义的特性自动定义 getter 方法:

```
#define GET_SUBTARGETINFO_MACRO(ATTRIBUTE, DEFAULT, \
                                GETTER) \
    bool GETTER() const { return ATTRIBUTE; }
#include "M88kGenSubtargetInfo.inc"
```

5. 此外，需要声明 ParseSubtargetFeatures() 方法。方法本身是从目标描述生成的:

```
    void ParseSubtargetFeatures(StringRef CPU,
                                StringRef TuneCPU,
                                StringRef FS);
```

6. 接下来，必须为成员变量添加 getter 方法:

```
    const TargetFrameLowering *
    getFrameLowering() const override {
        return &FrameLowering;
    }
    const M88kInstrInfo *getInstrInfo() const override {
        return &InstrInfo;
    }
    const M88kTargetLowering *
    getTargetLowering() const override {
```

```
        return &TLInfo;
    }
```

7. 最后，必须为寄存器信息添加 getter 方法，该方法属于指令信息类。这就结束了声明:

```
    const M88kRegisterInfo *
    getRegisterInfo() const override {
        return &InstrInfo.getRegisterInfo();
    }
};
} // end namespace llvm
```

接下来，必须实现实际的子目标类。实现存储在 M88kSubtarget.cpp 文件中:

1. 同样，通过包含生成的源代码来开始文件:

```
#define GET_SUBTARGETINFO_TARGET_DESC
#define GET_SUBTARGETINFO_CTOR
#include "M88kGenSubtargetInfo.inc"
```

2. 然后，定义 anchor 方法，将虚函数表固定在这个文件上:

```
void M88kSubtarget::anchor() {}
```

3. 最后，定义构造函数，生成的类需要两个 CPU 参数: 第一个用于指令集，第二个用于调度。这里的用例是，希望针对最新的 CPU 优化代码，但仍然能够在较旧的 CPU 上运行代码。我们不支持这个特性，两个参数使用相同的 CPU 名称:

```
M88kSubtarget::M88kSubtarget(const Triple &TT,
                             const std::string &CPU,
                             const std::string &FS,
                             const TargetMachine &TM)
    : M88kGenSubtargetInfo(TT, CPU, /*TuneCPU*/ CPU, FS),
      TargetTriple(TT), InstrInfo(*this),
      TLInfo(TM, *this), FrameLowering() {}
```

12.6.2. 实现 M88kTargetMachine——定义

最后，可以实现 M88kTargetMachine 类。该类保存所有使用的子目标实例，还拥有 TargetLoweringObjectFile 的子类，该子类为向下转译过程提供了诸如节名之类的详细信息。最后，创建在此后端中运行通道的配置。

M88kTargetMachine.h 文件中的声明如下:

1. M88kTargetMachine 类派生自 LLVMTargetMachine 类，唯一的成员是 TargetLoweringObjectFile 的一个实例和子目标映射:

```
namespace llvm {
class M88kTargetMachine : public LLVMTargetMachine {
    std::unique_ptr<TargetLoweringObjectFile> TLOF;
```

```
    mutable StringMap<std::unique_ptr<M88kSubtarget>>
        SubtargetMap;
```

2. 构造函数的参数完全描述了，将为其生成代码的目标配置。使用 TargetOptions 类，可以控制代码生成的许多细节——例如，是否可以使用浮点乘法和加法指令。此外，重定位模型、代码模型和优化级别可传递给构造函数。若目标机器用于实时编译，则 JIT 参数设置为 true。

```
public:
    M88kTargetMachine(const Target &T, const Triple &TT,
                      StringRef CPU, StringRef FS,
                      const TargetOptions &Options,
                      std::optional<Reloc::Model> RM,
                      std::optional<CodeModel::Model> CM,
                      CodeGenOpt::Level OL, bool JIT);
```

3. 我们还需要重写一些方法。getSubtargetImpl() 方法返回要用于给定函数的子目标实例，而 getObjFileLowering() 方法只返回成员变量。另外，覆写了 createPassConfig() 方法，该方法返回后端传递的配置:

```
    ~M88kTargetMachine() override;

    const M88kSubtarget *
    getSubtargetImpl(const Function &) const override;

    TargetPassConfig *
    createPassConfig(PassManagerBase &PM) override;

    TargetLoweringObjectFile *
    getObjFileLowering() const override {
        return TLOF.get();
    }
};
} // end namespace llvm
```

12.6.3. 实现 M88kTargetMachine——添加实现

该类的实现存储在 M88kTargetMachine.cpp 文件中。注意，在第 11 章中创建了这个文件。现在，我们将用一个完整的实现来替换这个文件:

1. 首先，必须注册目标机器，必须通过前面定义的初始化函数初始化 DAG 到 DAG 的通道:

```
extern "C" LLVM_EXTERNAL_VISIBILITY void
LLVMInitializeM88kTarget() {
    RegisterTargetMachine<M88kTargetMachine> X(
        getTheM88kTarget());
    auto &PR = *PassRegistry::getPassRegistry();
    initializeM88kDAGToDAGISelPass(PR);
}
```

2. 接下来，必须定义支持函数 computeDataLayout()。这个函数中，数据布局作为后端，期望定义。由于数据布局取决于硬件特性，因此三元组、CPU 名称和特性集字符串将传递给该函数，使用以下组件创建数据布局字符串。目标是大端 (E)，并使用 ELF 符号识别。
   指针是 32 位宽且 32 位对齐的，所有标量类型都是自然对齐的。MC88110 CPU 具有扩展寄存器集，并支持 80 位宽的浮点数。若要支持这个特殊的特性，则需要添加对 CPU 名称的检查，并用相应的浮点值扩展字符串。接下来，必须声明所有全局变量都有 16 位的首选对齐方式，并且硬件只有 32 位寄存器:

```
namespace {
std::string computeDataLayout(const Triple &TT,
                              StringRef CPU,
                              StringRef FS) {
    std::string Ret;
    Ret += "E";
    Ret += DataLayout::getManglingComponent(TT);
    Ret += "-p:32:32:32";
    Ret += "-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64";
    Ret += "-f32:32:32-f64:64:64";
    Ret += "-a:8:16";
    Ret += "-n32";
    return Ret;
}
} // namespace
```

3. 现在，可以定义构造函数和析构函数，许多参数只是传递给超类构造函数。注意，这里调用了 computeDataLayout() 函数。此外，TLOF 成员使用 TargetLoweringObjectFileELF 实例初始化，所以使用的是 ELF 文件格式。在构造函数体中，必须调用 initAsmInfo() 方法，该方法初始化超类的许多数据成员:

```
M88kTargetMachine::M88kTargetMachine(
    const Target &T, const Triple &TT, StringRef CPU,
    StringRef FS, const TargetOptions &Options,
    std::optional<Reloc::Model> RM,
    std::optional<CodeModel::Model> CM,
    CodeGenOpt::Level OL, bool JIT)
    : LLVMTargetMachine(
        T, computeDataLayout(TT, CPU, FS), TT, CPU,
        FS, Options, !RM ? Reloc::Static : *RM,
        getEffectiveCodeModel(CM, CodeModel::Medium),
        OL),
    TLOF(std::make_unique<
        TargetLoweringObjectFileELF>()) {
    initAsmInfo();
}

M88kTargetMachine::~M88kTargetMachine() {}
```

4. 之后，定义 getSubtargetImpl() 方法。要使用的子目标实例取决于 target-cpu 和 target-features 函

数属性。例如，可以将 target-cpu 属性设置为 MC88110，从而针对第二代 CPU。属性目标特性可以描述，不应该使用该 CPU 的图形指令。但还没有在目标描述中定义 CPU 和它们的特性，所以在这里做了一些不必要的事情，但实现非常简单: 查询函数属性并使用返回的字符串或默认值。有了这些信息，可以查询 SubtargetMap 成员，若没有找到，则创建子目标:

```
const M88kSubtarget *
M88kTargetMachine::getSubtargetImpl(
        const Function &F) const {
    Attribute CPUAttr = F.getFnAttribute("target-cpu");
    Attribute FSAttr =
        F.getFnAttribute("target-features");

    std::string CPU =
        !CPUAttr.hasAttribute(Attribute::None)
            ? CPUAttr.getValueAsString().str()
            : TargetCPU;
    std::string FS = !FSAttr.hasAttribute(Attribute::None)
                         ? FSAttr.getValueAsString().str()
                         : TargetFS;

    auto &I = SubtargetMap[CPU + FS];
    if (!I) {
        resetTargetOptions(F);
        I = std::make_unique<M88kSubtarget>(TargetTriple,
                                            CPU, FS, *this);
    }
    return I.get();
}
```

5. 最后，创建通道配置，需要自己的类 M88kPassConfig，其派生自 TargetPassConfig 类，只需要重写 addInstSelector 方法即可:

```
namespace {
class M88kPassConfig : public TargetPassConfig {
    public:
    M88kPassConfig(M88kTargetMachine &TM,
                   PassManagerBase &PM)
        : TargetPassConfig(TM, PM) {}
    bool addInstSelector() override;
};
} // namespace
```

6. 有了这个定义，就可以实现 createPassConfig 工厂方法:

```
TargetPassConfig *M88kTargetMachine::createPassConfig(
        PassManagerBase &PM) {
    return new M88kPassConfig(*this, PM);
}
```

7. 最后，必须在 addInstSelector() 方法中将指令选择类添加到通道流水线中。返回值 false 表示添加了一个将 LLVM IR 转换为机器指令的通道:

```
bool M88kPassConfig::addInstSelector() {
        addPass(createM88kISelDag(getTM<M88kTargetMachine>(),
                                  getOptLevel()));
    return false;
}
```

完成实现是一段漫长的旅程! 现在，我们可以构建 llc 工具了，可以运行一个示例，将下面的简单 IR 保存在 and.ll 文件中:

```
define i32 @f1(i32 %a, i32 %b) {
    %res = and i32 %a, %b
    ret i32 %res
}
```

现在，可以运行 llc 并验证生成的程序集看起来是合理的:

```
$ llc -mtriple m88k-openbsd < and.ll
        .text
        .file "<stdin>"
        .globl f1                               | -- Begin function f1
        .align 2
        .type f1,@function
f1:                                             | @f1
| %bb.0:
        and %r2, %r2, %r3
        jmp %r1
.Lfunc_end0:
        .size   f1, .Lfunc_end0-f1
                                                | -- End function
        .section        ".note.GNU-stack","",@progbits
```

要针对 m88k 目标进行编译，必须在命令行或 IR 文件中指定这个三元组，如本例中所示。

了解全局指令选择之前，先享受一下成功吧！

12.7. 全局指令的选择

通过选择 DAG 进行指令选择可以生成快速的代码，但是这样做需要时间。对于开发人员来说，编译器的速度至关重要，他们想要快速地尝试他们所做的更改。通常，编译器在优化级别 0 时应该非常快，但是随着优化级别的增加，可能会花费更多的时间。但构造选择 DAG 需要花费大量时间，这种方法无法按需要进行扩展。第一个解决方案是创建另一个名为 FastISel 的指令选择算法，该算法速度很快。但不能生成好的代码，也不与选择 DAG 实现共享代码，这是一个明显的问题，所以并非所有目标都支持 FastISel。

选择 DAG 方法不能扩展，它是一个大型的单片算法。若可以避免创建新的数据结构，如选择 DAG，则应该能够使用小组件执行指令选择。后端已经有了一个通道流水线，所以使用通道是很自然的选择。基于这些想法，GlobalISel 执行以下步骤:

1. 首先，将 LLVM IR 向下转译为通用机器指令，通用机器指令代表了实际硬件中最常见的操作。注意，此转换使用机器函数和机器基本块，所以直接转换为后端其他部分使用的数据结构。
2. 然后，将通用机器指令合法化。
3. 然后，将通用机器指令的操作数映射到注册库。
4. 最后，使用目标描述中定义的模式，将通用指令替换为真正的机器指令。

因为这些都是通道，所以可以在中间插入任意多的通道。例如，组合通道可用于用另一个通用机器指令或真正的机器指令替换通用机器指令序列。关闭这些通道可以提高编译速度，打开它们可以提高生成代码的质量，所以可以根据需要进行扩展。

这种方法还有另一个优点。选择 DAG 逐个基本块转换基本块，但是机器传递对机器函数工作，这使我们能够在指令选择期间考虑函数的所有基本块，所以这种指令选择方法称为全局指令选择(GlobalISel)。让我们从调用的转换开始，看看这种方法是如何工作的。

12.7.1. 向下转译参数和返回值

为了将 LLVM IR 转换为通用机器指令，只需要实现如何处理参数和返回值。同样，可以通过使用从目标描述生成的代码来向下转译实现。我们要创建的类名为 m88kcalllowervm，声明在 GISel/M88kcalllowervm.h 头文件中:

```
class M88kCallLowering : public CallLowering {
public:
    M88kCallLowering(const M88kTargetLowering &TLI);

    bool
    lowerReturn(MachineIRBuilder &MIRBuilder,
                const Value *Val,
                ArrayRef<Register> VRegs,
                FunctionLoweringInfo &FLI,
                Register SwiftErrorVReg) const override;

    bool lowerFormalArguments(
        MachineIRBuilder &MIRBuilder, const Function &F,
        ArrayRef<ArrayRef<Register>> VRegs,
        FunctionLoweringInfo &FLI) const override;
    bool enableBigEndian() const override { return true; }
};
```

GlobalISel 框架在转换函数时会调用 lowerReturn() 和 lowerFormalArguments() 方法。要转换函数调用，还需要覆盖并实现 lowerCall() 方法。还需要覆写 enableBigEndian()，否则就会生成错误的机器码。

对于 GISel/M88kCallLowering.cpp 文件中的实现，需要定义支持类。从目标描述生成的代码告诉我们参数是如何传递的——例如，在寄存器中。需要创建 ValueHandler 的一个子类来生成机器指令。对于传入参数，需要从 IncomingValueHandler 派生类，以及从 OutgoingValueHandler 派生返回值。两者非常相似，所以只查看传入参数的处理程序:

```
namespace {
struct FormalArgHandler
        : public CallLowering::IncomingValueHandler {
    FormalArgHandler(MachineIRBuilder &MIRBuilder,
                     MachineRegisterInfo &MRI)
        : CallLowering::IncomingValueHandler(MIRBuilder,
                                             MRI) {}

    void assignValueToReg(Register ValVReg,
                          Register PhysReg,
                          CCValAssign VA) override;

    void assignValueToAddress(Register ValVReg,
                              Register Addr, LLT MemTy,
                              MachinePointerInfo &MPO,
                              CCValAssign &VA) override{};

    Register
    getStackAddress(uint64_t Size, int64_t Offset,
                    MachinePointerInfo &MPO,
                    ISD::ArgFlagsTy Flags) override {
        return Register();
    };
};
} // namespace
```

目前，只能处理在寄存器中传递的参数，因此必须为其他方法提供一个虚拟实现。assignValueToReg() 方法将传入的物理寄存器的值复制到虚拟寄存器，必要时进行截断。这里所要做的，就是将物理寄存器标记为函数的“常驻”(live-in)，并调用超类实现:

```
void FormalArgHandler::assignValueToReg(
        Register ValVReg, Register PhysReg,
        CCValAssign VA) {
    MIRBuilder.getMRI()->addLiveIn(PhysReg);
    MIRBuilder.getMBB().addLiveIn(PhysReg);
    CallLowering::IncomingValueHandler::assignValueToReg(
        ValVReg, PhysReg, VA);
}
```

现在，我们可以实现 lowerFormalArgument() 方法:

1. 首先，将 IR 函数的参数转换为 ArgInfo 类的实例。setArgFlags() 和 splitToValueTypes() 有助于在传入参数需要多个虚拟寄存器的情况下，复制参数属性和分割值类型:

```
bool M88kCallLowering::lowerFormalArguments(
        MachineIRBuilder &MIRBuilder, const Function &F,
        ArrayRef<ArrayRef<Register>> VRegs,
        FunctionLoweringInfo &FLI) const {
    MachineFunction &MF = MIRBuilder.getMF();
    MachineRegisterInfo &MRI = MF.getRegInfo();
    const auto &DL = F.getParent()->getDataLayout();

    SmallVector<ArgInfo, 8> SplitArgs;
    for (const auto &[I, Arg] :
            llvm::enumerate(F.args())) {
        ArgInfo OrigArg{VRegs[I], Arg.getType(),
                        static_cast<unsigned>(I)};
        setArgFlags(OrigArg,
                    I + AttributeList::FirstArgIndex, DL,
                    F);
        splitToValueTypes(OrigArg, SplitArgs, DL,
                          F.getCallingConv());
    }
```

2. 有了 SplitArgs 变量中准备的参数，就可以生成机器代码了。生成的调用约定 CC_M88k 和辅助类 FormalArghandler 帮助下，这一切都由框架代码自行完成:

```
    IncomingValueAssigner ArgAssigner(CC_M88k);
    FormalArgHandler ArgHandler(MIRBuilder, MRI);
    return determineAndHandleAssignments(
        ArgHandler, ArgAssigner, SplitArgs, MIRBuilder,
        F.getCallingConv(), F.isVarArg());
}
```

返回值的处理类似，主要区别是最多返回一个值。下一个任务是使通用机器指令合法化。

12.7.2. 通用机器指令合法化

从 LLVM IR 到通用机器码的转换大部分是固定的，所以会生成使用不受支持的数据类型的指令，以及其他挑战，合法化通道任务，从而定义哪些操作和指令合法。有了这些信息，GlobalISel 框架试图将指令转化为合法形式。例如，m88k 架构只有 32 位寄存器，因此使用 64 位值的按位和操作则不合法。但若将 64 位值拆分为两个 32 位值，并使用两个按位和操作，就有了合法的代码。这可以转化为一个合法化规则:

```
getActionDefinitionsBuilder({G_AND, G_OR, G_XOR})
    .legalFor({S32})
    .clampScalar(0, S32, S32);
```

无论何时，当合法化程序处理一条 G_AND 指令时，若所有操作数都是 32 位宽的，就是合法的。否则，操作数将被限制为 32 位，有效地将较大的值拆分为多个 32 位值，并再次应用该规则。

若一条指令不能合法化，那么后端就会以一条错误消息终止。

所有的合法化规则都在 M88kLegalizerInfo 类的构造函数中定义，这使得该类非常简单。

合法化是什么意思?

在 GlobalISel 中，若通用指令可以让指令选择器翻译，就是合法的。这给了我们在实现上更多的自由。例如，只要指令选择器能够正确处理该类型，就可以声明一条指令处理位值，即使硬件只处理 32 位值。

我们需要看的下一关是寄存器库的选择器。

12.7.3. 为操作数选择一个寄存器库

许多架构定义了多个寄存器库，寄存器库是一组寄存器，典型的寄存器库有通用寄存器库和浮点寄存器库。为什么这些信息很重要? 寄存器库中，将值从一个寄存器移动到另一个寄存器通常很简单，但将值复制到另一个寄存器库可能代价高昂，甚至不可能，所以必须为每个操作数选择一个合适的寄存器库。

该类的实现涉及到对目标描述的添加。在 GISel/M88lRegisterbanks.td 文件中，定义了一个寄存器库，引用了我们定义的寄存器类:

```
def GRRegBank : RegisterBank<"GRRB", [GPR, GPR64]>;
```

这一行，生成了一些支持代码，但仍然需要添加一些可能生成的代码。需要定义部分映射，这告诉框架一个值从哪个位索引开始，它有多宽，以及它映射到哪个寄存器库。我们有两个条目，每个寄存器类一个:

```
RegisterBankInfo::PartialMapping
    M88kGenRegisterBankInfo::PartMappings[]{
        {0, 32, M88k::GRRegBank},
        {0, 64, M88k::GRRegBank},
    };
```

要索引这个数组，必须定义一个枚举:

```
enum PartialMappingIdx { PMI_GR32 = 0, PMI_GR64, };
```

因为只有三个地址指令，所以需要三个部分映射，每个操作数对应一个。必须用所有这些指针创建一个数组，其中第一个表项表示无效映射:

```
RegisterBankInfo::ValueMapping
    M88kGenRegisterBankInfo::ValMappings[]{
        {nullptr, 0},
        {&M88kGenRegisterBankInfo::PartMappings[PMI_GR32], 1},
        {&M88kGenRegisterBankInfo::PartMappings[PMI_GR32], 1},
        {&M88kGenRegisterBankInfo::PartMappings[PMI_GR32], 1},
```

```
        {&M88kGenRegisterBankInfo::PartMappings[PMI_GR64], 1},
        {&M88kGenRegisterBankInfo::PartMappings[PMI_GR64], 1},
        {&M88kGenRegisterBankInfo::PartMappings[PMI_GR64], 1},
    };
```

要访问该数组，必须定义一个函数:

```
const RegisterBankInfo::ValueMapping *
M88kGenRegisterBankInfo::getValueMapping(
        PartialMappingIdx RBIdx) {
    return &ValMappings[1 + 3*RBIdx];
}
```

创建这些表时，很容易出错。所有这些信息都可以从目标描述中获得，并且源代码中的注释指出该代码应该由 TableGen! 然而，这还没有实现，所以必须手动书写代码。

必须在 M88kRegisterBankInfo 类中实现的最重要的函数是 getInstrMapping()，为指令的每个操作数返回映射的寄存器库。现在这变得很简单，可以查找部分映射数组，然后可以将其传递给 getInstructionMapping()，该方法构造完整的指令映射:

```
const RegisterBankInfo::InstructionMapping &
M88kRegisterBankInfo::getInstrMapping(
        const MachineInstr &MI) const {
    const ValueMapping *OperandsMapping = nullptr;
    switch (MI.getOpcode()) {
    case TargetOpcode::G_AND:
    case TargetOpcode::G_OR:
    case TargetOpcode::G_XOR:
        OperandsMapping = getValueMapping(PMI_GR32);
        break;
    default:
#if !defined(NDEBUG) || defined(LLVM_ENABLE_DUMP)
        MI.dump();
#endif
        return getInvalidInstructionMapping();
    }

    return getInstructionMapping(DefaultMappingID, /*Cost=*/1,
                                 OperandsMapping,
                                 MI.getNumOperands());
}
```

开发过程中，通常会忘记通用指令的寄存器库映射。

不幸的是，运行时生成的错误消息没有提及映射失败的指令。简单的修复方法是在返回无效映射之前转储指令。但这里需要小心，dump() 方法并非在所有构建类型中都可用。

映射寄存器库之后，必须将通用机器指令转换成真正的机器指令。

12.7.4. 翻译通用机器指令

对于通过选择 DAG 进行指令选择，目标描述中添加了使用 DAG 操作和操作数的模式。为了重用这些模式，引入了从 DAG 节点类型到通用机器指令的映射。例如，DAG 和操作映射到通用的 G_AND 机器指令，并非所有 DAG 操作都具有等效的通用机器指令，但涵盖了最常见的情况，所以有利于在目标描述中定义所有的代码选择模式。

M88kInstructionSelector 类的大部分实现 (可以在 GISel/M88kInstructionSelector.cpp 文件中找到) 是从目标描述生成的。然而，需要重写 select() 方法，可以翻译目标描述中模式未涵盖的通用机器指令。由于只支持非常小的通用指令子集，所以可以简单地调用生成的模式匹配器:

```
bool M88kInstructionSelector::select(MachineInstr &I) {
    if (selectImpl(I, *CoverageInfo))
        return true;
    return false;
}
```

随着指令选择的实现，可以使用 GlobalISel 翻译 LLVM IR !

12.7.5. 运行一个示例

要使用 GlobalISel 翻译 LLVM IR，需要在 llc 的命令行中添加-global-isel 选项。例如，可以使用前面定义的 IR 文件——and.ll:

```
$ llc -mtriple m88k-openbsd -global-isel < and.ll
```

输出的汇编文本相同。为了让自己相信翻译使用了 GlobalISel，必须利用这样一个事实，即可以在使用-stop-after= 选项运行指定的传递后停止翻译。例如，要查看合法化后的通用指令，可以执行以下命令:

```
$ llc -mtriple m88k-openbsd -global-isel < and.ll -stop-after=legalizer
```

因为 GlobalISel 可使调试和测试实现变得容易，其另一个优点是能够在运行一次通道之后 (或之前) 停止。

现在，我们有了一个工作后端，可以将一些 LLVM IR 转换为 m88k 架构的机器码。让我们考虑一下如何基于此，实现一个更完整的后端。

12.8. 进化后端

使用本章和前一章的代码，创建了一个后端，可以将一些 LLVM IR 转换为机器码。看到后端能够正常工作是非常令人满意的，但不能用于实际任务。需要编写更多的代码。以下是如何进一步发展后端的秘诀:

- 第一个决定是，是使用 GlobalISel，还是选择 DAG。根据经验，GlobalISel 更容易理解和开发，但是 LLVM 源代码树中的所有目标都实现了选择 DAG，并且开发者可能已经有了使用经验。

- 接下来，应该定义用于加减整数值的指令，这与按位和指令类似。
- 之后，应该实现加载和存储指令。由于需要转换不同的寻址模式，这就更复杂了。最可能的情况是，需要处理索引，对数组中的一个元素进行寻址，这很可能需要前面定义的加法指令。
- 最后，可以完全实现帧和调用的向下转译，可以翻译一个简单的“Hello, world!”样式的应用程序变成可运行的程序。
- 下一个逻辑步骤是实现分支指令，支持循环的转换。为了生成最优的代码，需要在指令信息类中实现分支分析方法。

这时，自定义后端已经可以翻译简单的算法。还应该获得足够的经验，可以根据优先级开发缺失的部分。

12.9. 总结

本章中，后端添加了两种不同的指令选择: 通过选择 DAG 进行指令选择和全局指令选择，所以必须在目标描述中定义调用约定。此外，需要实现寄存器和指令信息类，能够访问从目标描述生成的信息，但还需要使用其他信息对其进行增强，了解到堆栈帧布局和脚本生成稍后才需要。为了转换示例，添加了一个类来发出机器指令，并创建了后端配置，还了解了全局指令选择的工作原理。最后，获得了一些关于如何自己开发后端的指导。

下一章中，将介绍一些在指令选择之后可以完成的任务——将在后端流水线中添加一个新通道，看看如何将后端集成到 clang 编译器中，以及如何交叉编译到不同的架构中。

第 13 章 超越指令选择

现在已经在前面的章节中，了解了使用 SelectionDAG 和 GlobalISel 基于 LLVM 的框架进行指令选择，可以探索指令选择之外的其他有趣概念。本章封装了后端之外的一些更高级的主题，这些主题对于高度优化的编译器来说可能会很有趣。例如，一些通道超出了指令选择的范畴，可以对不同的指令执行不同的优化，所以开发人员可以在编译器中引入自己的通道来执行特定于目标的任务。

最后，本章中，将学习以下主题:

- 向 LLVM 添加一个新的机器函数通道
- 将新目标集成到 clang 前端
- 如何针对不同的 CPU 架构

13.1. 为 LLVM 添加新机器功能通道

本节中，我探讨如何在 LLVM 中实现一个在指令选择后运行的新机器函数通道。将创建一个 MachineFunctionPass 类，它是 LLVM 中原始 FunctionPass 类的一个子集，可以与 opt 一起运行。该类调整了原始基础设施，以通过 llc 在后端对 MachineFunction 表示进行操作的通道实现。

后端中的通道实现利用遗留的通道管理器接口，而不是新的通道管理器。因为 LLVM 目前在后端，没有一个完整的新通道管理器的工作实现，所以本章将遵循在遗留通道管理器流水线中添加新通道的方法。

实现方面，机器函数通道一次优化单个 (机器) 函数，但不是覆写 runOnFunction() 方法，而是覆写 runOnMachineFunction() 方法。本节将实现的机器函数通道是一个通道，用于检查除零行为，特别是插入后端捕获的代码。由于 MC88100 上的硬件限制，因为该 CPU 不能可靠地检测除零情况，所以这种类型的通道对于 M88k 目标很重要。

继续上一章的后端实现，来看看后端机器函数通道是如何实现的!

13.1.1. 实现了 M88k 目标的顶层接口

首先，在 llvm/lib/Target/M88k/M88k.h 中，在 llvm 命名空间声明中添加两个原型，以便稍后使用:

1. 将要实现的机器函数通道称为 M88kDivInstrPass，将添加一个函数声明来初始化这个通道，并接受通道注册表，这是一个管理所有通道的注册和初始化的类:

```
void initializeM88kDivInstrPass(PassRegistry &);
```

2. 接下来，声明创建 M88kDivInstr 通道的实际函数，将 M88k 目标机器信息作为其参数:

```
FunctionPass *createM88kDivInstr(const M88kTargetMachine &);
```

13.1.2. 为机器函数通道添加 TargetMachine 实现

接下来，我们将分析 llvm/lib/Target/M88k/M88kTargetMachine.cpp 中需要进行的一些更改:

1. LLVM 中，通常为用户提供打开或关闭通道选项，所以为用户提供与机器功能通道相同的灵活性。首先声明一个名为 m88k-no-check-zero-division 的命令行选项，并将其初始化为 false，所以除非用户显式地将其关闭，否则将始终检查是否有零除法。我们把它添加到 llvm 命名空间声明中，并且是 llc 的一个选项:

```
using namespace llvm;
static cl::opt<bool>
    NoZeroDivCheck("m88k-no-check-zero-division", cl::Hidden,
                   cl::desc("M88k: Don't trap on integer division by zero."),
                   cl::init(false));
```

2. 还创建一个返回命令行值的正式方法，以便查询它以确定是否将运行该通道。我们最初的命令行选项将封装在 noZeroDivCheck() 方法中，以便稍后可以使用命令行结果:

```
M88kTargetMachine::~M88kTargetMachine() {}
bool M88kTargetMachine::noZeroDivCheck() const { return NoZeroDivCheck; }
```

3. 接下来，在 LLVMInitializeM88kTarget() 中，M88k 目标和通道注册和初始化，将插入对 initializeM88kDivInstrPass() 的调用，该方法先前在 llvm/lib/target/M88k/M88k.h 中声明:

```
extern "C" LLVM_EXTERNAL_VISIBILITY void
LLVMInitializeM88kTarget() {
    RegisterTargetMachine<M88kTargetMachine> X(getTheM88kTarget());
    auto &PR = *PassRegistry::getPassRegistry();
    initializeM88kDAGToDAGISelPass(PR);
    initializeM88kDivInstrPass(PR);
}
```

4. M88k 目标还需要覆盖 addMachineSSAOptimization()，这是一个方法，当机器指令是 SSA 形式时，添加通道来优化机器指令。我们的机器函数通道是作为一种机器 SSA 优化添加的，此方法声明为一个将覆写的函数。将在 M88kTargetMachine.cpp 的末尾添加完整的实现:

```
    bool addInstSelector() override;
    void addPreEmitPass() override;
    void addMachineSSAOptimization() override;

. . .
void M88kPassConfig::addMachineSSAOptimization() {
    addPass(createM88kDivInstr(getTM<M88kTargetMachine>()));
    TargetPassConfig::addMachineSSAOptimization();
}
```

5. 我们的方法返回命令行选项来切换机器函数传递 (noZeroDivCheck() 方法) 也在 M88kTargetMachine.h 中声明:

```
    ~M88kTargetMachine() override;
    bool noZeroDivCheck() const;
```

13.1.3. 开发具体的机器功能通道

现在，完成了在 M88k 目标机上的实现，下一步将是开发机器功能通道本身。实现包含在新文件 llvm/lib/Target/M88k/m88kdivinstrument.cpp 中:

1. 首先，添加机器函数通道所需的头文件。这包括访问 M88k 目标信息的头文件和允许操作机器函数和机器指令的头文件:

```
#include "M88k.h"
#include "M88kInstrInfo.h"
#include "M88kTargetMachine.h"
#include "MCTargetDesc/M88kMCTargetDesc.h"
#include "llvm/ADT/Statistic.h"
#include "llvm/CodeGen/MachineFunction.h"
#include "llvm/CodeGen/MachineFunctionPass.h"
#include "llvm/CodeGen/MachineInstrBuilder.h"
#include "llvm/CodeGen/MachineRegisterInfo.h"
#include "llvm/IR/Instructions.h"
#include "llvm/Support/Debug.h"
```

2. 之后，将添加一些代码来准备我们的机器函数通道。第一个是名为 m88k-div-instr 的 DEBUG_TYPE 定义，用于调试时的细粒度控制。定义这个 DEBUG_TYPE 允许用户指定机器函数的通道名称，并在调试信息启用时查看与该通道相关的调试信息:

```
#define DEBUG_TYPE "m88k-div-instr"
```

3. 还指定使用 llvm 名称空间，并为机器函数声明一个 STATISTIC 值。这个统计数据称为 InsertedChecks，其跟踪编译器插入了多少个除零检查。最后，声明一个匿名命名空间来封装后续的机器函数通道实现:

```
using namespace llvm;
STATISTIC(InsertedChecks, "Number of inserted checks for
division by zero");
namespace {
```

4. 这个机器函数通道的目的是检查除零的情况，并插入会导致 CPU 陷入陷阱的指令。这些指令需要条件码，所以们称之为 CC0 的枚举值定义了对 M88k 目标有效的条件码，以及其编码:

```
enum class CC0 : unsigned {
    EQ0 = 0x2,
    NE0 = 0xd,
    GT0 = 0x1,
    LT0 = 0xc,
    GE0 = 0x3,
    LE0 = 0xe
};
```

5. 接下来，让为机器函数通道创建实际的类，称为 M88kDivInstr。首先，将其创建为继承 MachineFunctionPass 类型的实例，再声明 M88kDivInstr 传递将需要的各种必要实例。这包括 M88kBuilder(将在后面创建并详细介绍) 和 M88kTargetMachine(包含目标指令和寄存器信

息)。生成指令时，还需要寄存器库信息和机器寄存器信息。还添加了一个 AddZeroDivCheck 布尔值来表示前面的命令行选项，可以打开或关闭通道:

```
class M88kDivInstr : public MachineFunctionPass {
    friend class M88kBuilder;
    const M88kTargetMachine *TM;
    const TargetInstrInfo *TII;
    const TargetRegisterInfo *TRI;
    const RegisterBankInfo *RBI;
    MachineRegisterInfo *MRI;
    bool AddZeroDivCheck;
```

6. 对于 M88kDivInstr 类的公共变量和方法，我们声明了一个标识号，LLVM 将使用它来标识我们的通道，以及 M88kDivInstr 构造函数，接受 M88kTargetMachine。接下来，覆写 getRequiredProperties() 方法，该方法表示 MachineFunction 在优化过程中可能具有的属性，还要覆写 runOnMachineFunction() 方法，这是通道检查除零时运行的主要方法之一。第二个公开声明的重要函数是 runOnMachineBasicBlock() 函数，将从 runOnMachineFunction() 内部执行:

```
public:
    static char ID;
    M88kDivInstr(const M88kTargetMachine *TM = nullptr);
    MachineFunctionProperties getRequiredProperties() const override;
    bool runOnMachineFunction(MachineFunction &MF) override;
    bool runOnMachineBasicBlock(MachineBasicBlock &MBB);
```

7. 最后，最后一部分是声明私有方法和关闭类。在 M88kDivInstr 类中声明的唯一私有方法是 addZeroDivCheck() 方法，在除法指令之后插入对除零的检查。正如稍后将看到的，MachineInstr 将需要指向 M88k 目标上的特定分割指令:

```
private:
    void addZeroDivCheck(MachineBasicBlock &MBB, MachineInstr *DivInst);
};
```

8. 接下来创建一个 M88kBuilder 类，是一个构建器实例，用于创建特定于 M88k 的指令。这个类保存了 MachineBasicBlock 的一个实例 (和一个相应的迭代器)，以及 DebugLoc 来跟踪这个构建器类的调试位置。其他必要的实例包括 M88k 目标机的目标指令信息、目标寄存器信息和寄存器库信息：

```
class M88kBuilder {
    MachineBasicBlock *MBB;
    MachineBasicBlock::iterator I;
    const DebugLoc &DL;
    const TargetInstrInfo &TII;
    const TargetRegisterInfo &TRI;
    const RegisterBankInfo &RBI;
```

9. 对于 M88kBuilder 类的公共方法，必须实现该构造器的构造函数。初始化时，构建器需要一个 M88kDivInstr 传递的实例来初始化目标指令、寄存器信息和寄存器库信息，以及 MachineBasicBlock 和调试位置:

```
public:
    M88kBuilder(M88kDivInstr &Pass, MachineBasicBlock *MBB, const DebugLoc &DL)
        : MBB(MBB), I(MBB->end()), DL(DL), TII(*Pass.TII), TRI(*Pass.TRI),
        ↪  RBI(*Pass.RBI) {}
```

10. 接下来，在 M88k 构造器中创建了一个方法来设置 MachineBasicBlock，并且相应地设置了 MachineBasicBlock 迭代器:

```
void setMBB(MachineBasicBlock *NewMBB) {
    MBB = NewMBB;
    I = MBB->end();
}
```

11. 接下来需要实现 constrainstt() 函数，并使用它处理 MachineInstr 实例。对于给定的 MachineInstr，检查 MachineInstr 实例的操作数的寄存器类，是否可以通过已有的函数 constrainSelectedInstRegOperands() 来约束。如下所示，这个机器函数通道可以约束机器指令的寄存器操作数:

```
void constrainInst(MachineInstr *MI) {
    if (!constrainSelectedInstRegOperands(*MI, TII, TRI, RBI))
    llvm_unreachable("Could not constrain register operands");
}
```

12. 该通道插入的指令之一是 BCND 指令，如 M88kInstrInfo 中定义的那样，是 M88k 目标上的一个条件分支。为了创建这个指令，需要一个条件代码，是在 m88kdivinstrument.cpp 开头实现的 CC0 枚举——一个寄存器和 MachineBasicBlock。BCND 指令只是在创建和检查新创建的指令是否可以约束后返回:

```
MachineInstr *bcnd(CC0 Cc, Register Reg, MachineBasicBlock
    *TargetMBB) {
    MachineInstr *MI = BuildMI(*MBB, I, DL, TII.get(M88k::BCND))
                           .addImm(static_cast<int64_t>(Cc))
                           .addReg(Reg)
                           .addMBB(TargetMBB);
    constrainInst(MI);
    return MI;
}
```

13. 类似地，也需要一个 trap 指令用于我们的机器函数通道，这是一个 TRAP503 指令。该指令需要一个寄存器，若寄存器的第 0 位没有设置，则会引发 TRAP503 陷阱，该陷阱将在除零后引发。创建 TRAP503 指令时，在返回 TRAP503 之前会检查是否存在约束。此外，这结束了 M88kBuilder 类的类实现，并完成了前面声明的匿名命名空间:

```
    MachineInstr *trap503(Register Reg) {
        MachineInstr *MI = BuildMI(*MBB, I, DL, TII.
            get(M88k::TRAP503)).addReg(Reg);
        constrainInst(MI);
        return MI;
```

```
    }
};
} // end anonymous namespace
```

14. 现在可以开始实现在机器函数通道中，执行实际检查的函数。首先，探索一下addZeroDivCheck() 是如何实现的。该函数只是在当前机器指令之间插入一个除零检查，该指令应该指向 DIVSrr 或 DIVUrr，这些分别是有符号除法和无符号除法的助记符。插入 BCND 和 TRAP503 指令，InsertedChecks 统计量加 1 表示添加了两条指令:

```
void M88kDivInstr::addZeroDivCheck(MachineBasicBlock &MBB,
                                   MachineInstr *DivInst) {
    assert(DivInst->getOpcode() == M88k::DIVSrr ||
           DivInst->getOpcode() == M88k::DIVUrr && "Unexpected
           opcode");
    MachineBasicBlock *TailBB = MBB.splitAt(*DivInst);
    M88kBuilder B(*this, &MBB, DivInst->getDebugLoc());
    B.bcnd(CC0::NE0, DivInst->getOperand(2).getReg(), TailBB);
    B.trap503(DivInst->getOperand(2).getReg());
    ++InsertedChecks;
}
```

15. 接下来实现 runOnMachineFunction()，在 LLVM 中创建函数传递类型时要覆写的重要函数之一。此函数返回 true 或 false，取决于在机器函数传递期间是否进行了任何更改。对于给定的机器功能，收集所有相关的 M88k 子目标信息，包括目标指令、目标寄存器、寄存器库和机器寄存器信息。关于用户是否打开或关闭 M88kDivInstr 机器功能传递的详细信息也会查询并存储在 AddZeroDivCheck 变量中，需要对机器功能中的所有机器基本块进行除零分析。执行机器基本块分析的函数是 runOnMachineBasicBlock()，接下来将实现这一点。最后，若机器函数发生了变化，则由返回的 changed 变量表示:

```
bool M88kDivInstr::runOnMachineFunction(MachineFunction &MF) {
    const M88kSubtarget &Subtarget =
    MF.getSubtarget<M88kSubtarget>();
    TII = Subtarget.getInstrInfo();
    TRI = Subtarget.getRegisterInfo();
    RBI = Subtarget.getRegBankInfo();
    MRI = &MF.getRegInfo();
    AddZeroDivCheck = !TM->noZeroDivCheck();
    bool Changed = false;
    for (MachineBasicBlock &MBB : reverse(MF))
        Changed |= runOnMachineBasicBlock(MBB);
    return Changed;
}
```

16. 对于 runOnMachineBasicBlock() 函数，还返回一个 Changed 布尔标志来指示机器基本块是否已更改，但最初设置为 false。此外，在一个机器基本块中，需要分析所有的机器指令，并检查这些指令是否分别是 DIVUrr 或 DIVSrr 操作码。除了检查操作码是否为分割指令外，还需要检查用户是否打开或关闭了机器功能通道。若满足所有这些条件，则用条件分支进行除零

检查，并通过前面实现的 addZeroDivCheck() 函数相应地添加陷阱指令。

```
bool M88kDivInstr::runOnMachineBasicBlock(MachineBasicBlock
&MBB) {
    bool Changed = false;
    for (MachineBasicBlock::reverse_instr_iterator I =
    MBB.instr_rbegin();
            I != MBB.instr_rend(); ++I) {
        unsigned Opc = I->getOpcode();
        if ((Opc == M88k::DIVUrr || Opc == M88k::DIVSrr) &&
        AddZeroDivCheck) {
            addZeroDivCheck(MBB, &*I);
            Changed = true;
        }
    }
    return Changed;
}
```

17. 之后，需要实现构造函数来初始化函数通道，并设置适当的机器函数属性。这可以通过调用 initializem88kdivinstpass() 函数来实现，在 M88kDivInstr 类的构造函数中使用 PassRegistry 实例，也可以通过设置机器函数属性来说明通道需要机器函数是 SSA 形式:

```
M88kDivInstr::M88kDivInstr(const M88kTargetMachine *TM)
: MachineFunctionPass(ID), TM(TM) {
    initializeM88kDivInstrPass(*PassRegistry::getPassRegistry());
}
MachineFunctionProperties M88kDivInstr::getRequiredProperties()
const {
    return MachineFunctionProperties().set(
        MachineFunctionProperties::Property::IsSSA);
}
```

18. 下一步是初始化机器函数传递的 ID，并使用机器函数通道的详细信息实例化 INITIALIZE_PASS 宏。这需要通道实例、命名信息和两个布尔参数，用于指示通道是否只检查 CFG，以及通道是否为分析通道。由于 M88kDivInstr 不执行这两种操作，因此为通道初始化宏指定了两个 false 参数:

```
char M88kDivInstr::ID = 0;
INITIALIZE_PASS(M88kDivInstr, DEBUG_TYPE, "Handle div instructions", false,
↪  false)
```

19. 最后，createM88kDivInstr() 函数使用 M88kTargetMachine 实例创建 M88kDivInstr 通道的新实例，将封装到 llvm 命名空间中，并在完成此函数后结束命名空间:

```
namespace llvm {
FunctionPass *createM88kDivInstr(const M88kTargetMachine &TM) {
    return new M88kDivInstr(&TM);
}
} // end namespace llvm
```

13.1.4. 构建新实现的机器函数通道

我们几乎完成了实现新的机器函数通道! 现在，需要确保 CMake 意识到 m88kdivinst.cpp 中的新机器功能通道。然后，将该文件添加到 llvm/lib/Target/M88k/CMakeLists.txt 中:

```
add_llvm_target(M88kCodeGen
    M88kAsmPrinter.cpp
    M88kDivInstr.cpp
    M88kFrameLowering.cpp
    M88kInstrInfo.cpp
    M88kISelDAGToDAG.cpp
```

最后一步是用以下命令用我们的新机器函数通道实现构建 LLVM，需要 CMake 选项-DLLVM_EXPERIMENTAL_TARGETS_TO_BUILD=M88k 来构建 M88k 目标:

```
$ cmake -G Ninja ../llvm-project/llvm -DLLVM_EXPERIMENTAL_TARGETS_TO_
BUILD=M88k -DCMAKE_BUILD_TYPE=Release -DLLVM_ENABLE_PROJECTS="llvm"
$ ninja
```

这样，就实现了机器函数通道，但看它是如何工作的不是很有趣吗? 我们可以通过 llc 传递 LLVM IR 来演示此通道的结果。

13.1.5. 使用 llc 运行机器功能通道

我们有下面的 IR，包含一个除以 0 的除法:

```
$ cat m88k-divzero.ll
target datalayout = "E-m:e-p:32:32:32-i1:8:8-i8:8:8-i16:16:16-
i32:32:32-i64:64:64-f32:32:32-f64:64:64-a:8:16-n32"
target triple = "m88k-unknown-openbsd"

@dividend = dso_local global i32 5, align 4
define dso_local i32 @testDivZero() #0 {
    %1 = load i32, ptr @dividend, align 4
    %2 = sdiv i32 %1, 0
    ret i32 %2
}
```

把它输入 llc:

```
$ llc m88k-divzero.ll
```

结果汇编中，默认情况下，除零检查，用 bnd.n(BCND) 和 t0(TRAP503) 表示，由我们的新机函数通道插入:

```
| %bb.1:
    subu %r2, %r0, %r2
    bcnd.n ne0, %r0, .LBB0_2
    divu %r2, %r2, 0
    tb0 0, %r3, 503
. . .
.LBB0_3:
    bcnd.n ne0, %r0, .LBB0_4
    divu %r2, %r2, 0
    tb0 0, %r3, 503
```

然而，来看看当为 llc 指定--m88k-no-check-zero 除法时会发生什么:

```
$ llc m88k-divzero.ll -m88k-no-check-zero-division
```

这个后端选项指示 llc 不运行检查除零的传递，生成的汇编将不包含任何 bnd 或 TRAP503 指令。这里有一个例子:

```
| %bb.1:
    subu %r2, %r0, %r2
    divu %r2, %r2, 0
    jmp.n %r1
    subu %r2, %r0, %r2
```

实现一个机器函数通道需要几个步骤，但是这些步骤可以作为指导方针，有助于实现任何类型的机器函数通道。我们已经在本节中广泛地探讨了后端，所以切换一下方向，看看前端是如何了解 M88k 目标的。

13.2. 将新目标集成到 clang 前端

前面的章节中，我们在 LLVM 中开发了 M88k 目标的后端实现。为了完成 M88k 目标的编译器实现，通过为 M88k 目标添加 clang 实现来研究将新目标连接到前端。

13.2.1. clang 中实现驱动程序的集成

让我们从添加驱动程序集成到 M88k 的 clang 开始:

1. 要做的第一个更改是在 clang/include/clang/Basic/TargetInfo.h 文件中。BuiltinVaListKind 枚举列出了每个目标的不同类型的__ builtin_va_list 类型，用于支持可变函数，因此为 M88k 添加了相应的类型:

```
enum BuiltinVaListKind {
    . . .
    // typedef struct __va_list_tag {
        // int __va_arg;
```

```
        // int *__va_stk;
        // int *__va_reg;
        //} va_list;
    M88kBuiltinVaList
};
```

2. 接下来，必须添加一个新的头文件 clang/lib/Basic/Targets/M88k.h。该文件是前端中 M88k 目标特性支持的头文件。第一步是定义一个新的宏，以防止多个包含相同的头文件、类型、变量等，还必须包括实现所需的各种头文件:

```
#ifndef LLVM_CLANG_LIB_BASIC_TARGETS_M88K_H
#define LLVM_CLANG_LIB_BASIC_TARGETS_M88K_H
#include "OSTargets.h"
#include "clang/Basic/TargetInfo.h"
#include "clang/Basic/TargetOptions.h"
#include "llvm/Support/Compiler.h"
#include "llvm/TargetParser/Triple.h"
```

3. 将声明的方法将相应地添加到 clang 和 targets 命名空间中，就像 llvm-project 中的其他目标一样:

```
namespace clang {
namespace targets {
```

4. 现在，声明实际的 M88kTargetInfo 类，并让它扩展原来的 TargetInfo 类。若这个类链接到动态库，这个类标记为 LLVM_LIBRARY_VISIBILITY，这个属性允许 M88kTargetInfo 类只在库中可见，而在外部不可访问:

```
class LLVM_LIBRARY_VISIBILITY M88kTargetInfo: public TargetInfo
{
```

5. 另外，必须声明两个变量——一个表示寄存器名的字符数组和一个枚举值，其中包含 M88k 目标中可选择的可用 CPU 类型。我们设置的默认 CPU 是 CK_Unknown CPU。稍后，将看到这可以用户如何通过选项对此进行修改:

```
    static const char *const GCCRegNames[];
    enum CPUKind { CK_Unknown, CK_88000, CK_88100, CK_88110 } CPU = CK_Unknown;
```

6. 之后，首先声明类实现中需要的公共方法。除了类的构造函数之外，还定义了各种 getter 方法。这包括获得目标特定的 #define 值的方法，获得目标支持的内置列表的方法，返回 GCC 寄存器名及其别名的方法，最后，一个返回之前添加到 clang/include/clang/Basic/TargetInfo.h 的 M88k BuiltinVaListKind 方法。

```
public:
    M88kTargetInfo(const llvm::Triple &Triple, const TargetOptions&);
    void getTargetDefines(const LangOptions &Opts,
    MacroBuilder &Builder) const override;
    ArrayRef<Builtin::Info> getTargetBuiltins() const override;
    ArrayRef<const char *> getGCCRegNames() const override;
    ArrayRef<TargetInfo::GCCRegAlias> getGCCRegAliases() const override;
```

```
    BuiltinVaListKind getBuiltinVaListKind() const override {
        return TargetInfo::M88kBuiltinVaList;
    }
```

7. getter 方法之后，还必须定义对 M88k 目标执行各种检查的方法。第一个检查 M88k 目标是否具有特定的目标特性，以字符串的形式提供。当使用内联汇编时，添加一个函数来验证约束。最后，们有一个函数来检查一个特定的 CPU 是否对 M88k 目标有效，也以字符串的形式提供:

```
    bool hasFeature(StringRef Feature) const override;
    bool validateAsmConstraint(const char *&Name,
                               TargetInfo::ConstraintInfo &info)
                               const override;
    bool isValidCPUName(StringRef Name) const override;
```

8. 接下来，为 M88kTargetInfo 类声明 setter 方法。第一个方法简单地设置了我们想要瞄准的特定 M88k CPU，而第二个方法设置了一个 vector 来包含 M88k 的所有有效支持的 CPU:

```
    bool setCPU(const std::string &Name) override;
    void fillValidCPUList(SmallVectorImpl<StringRef> &Values) const override;
};
```

9. 为了完成驱动程序实现的头文件，结束开头添加的名称空间和宏定义:

```
} // namespace targets
} // namespace clang
#endif // LLVM_CLANG_LIB_BASIC_TARGETS_M88K_H
```

10. 现在，已经在 clang/lib/Basic/Targets 中完成了 M88k 头文件，必须在 clang/lib/Basic/Targets/M88k.cpp 中添加相应的 TargetInfo C++ 实现。我们将从包含所需的头文件开始，特别是刚刚创建的新 M88k.h 头文件:

```
#include "M88k.h"
#include "clang/Basic/Builtins.h"
#include "clang/Basic/Diagnostic.h"
#include "clang/Basic/TargetBuiltins.h"
#include "llvm/ADT/StringExtras.h"
#include "llvm/ADT/StringRef.h"
#include "llvm/ADT/StringSwitch.h"
#include "llvm/TargetParser/TargetParser.h"
#include <cstring>
```

11. 正如之前在头文件中所做的那样，从 clang 和目标命名空间开始，然后实现 M88kTargetInfo 类的构造函数:

```
namespace clang {
namespace targets {
M88kTargetInfo::M88kTargetInfo(const llvm::Triple &Triple,
                               const TargetOptions &)
    : TargetInfo(Triple) {
```

12. 构造函数中，为 M88k 目标设置了数据布局字符串。这个数据布局字符串位于发出的 LLVM IR 文件的顶部，数据布局字符串的每个部分的解释如下:

```
    std::string Layout = "";
    Layout += "E"; // M68k is Big Endian
    Layout += "-m:e";
    Layout += "-p:32:32:32"; // Pointers are 32 bit.
    // All scalar types are naturally aligned.
    Layout += "-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64";

    // Floats and doubles are also naturally aligned.
    Layout += "-f32:32:32-f64:64:64";
    // We prefer 16 bits of aligned for all globals; see above.
    Layout += "-a:8:16";

    Layout += "-n32"; // Integer registers are 32bits.
    resetDataLayout(Layout);
```

13. M88kTargetInfo 类的构造函数通过将各种变量类型设置为 signed long long、unsigned long 或 signed int 来结束:

```
    IntMaxType = SignedLongLong;
    Int64Type = SignedLongLong;
    SizeType = UnsignedLong;
    PtrDiffType = SignedInt;
    IntPtrType = SignedInt;
}
```

14. 完成目标器的 CPU 设置功能。这个函数接受一个字符串，并将 CPU 设置为用户在 llvm::StringSwitch 中提供的特定 CPU 字符串，其本质上只是一个常规的开关，但专用于使用 llvm 的字符串。可以看到 M88k 目标上有三种支持的 CPU 类型，提供的字符串不匹配任何预期的类型，则有 CK_Unknown 类型:

```
bool M88kTargetInfo::setCPU(const std::string &Name) {
    StringRef N = Name;
    CPU = llvm::StringSwitch<CPUKind>(N)
                .Case("generic", CK_88000)
                .Case("mc88000", CK_88000)
                .Case("mc88100", CK_88100)
                .Case("mc88110", CK_88110)
                .Default(CK_Unknown);
    return CPU != CK_Unknown;
}
```

15. 前面说过，在 M88k 目标上有三种支持的有效 CPU 类型:mc88000、mc88100 和 mc88110，其中通用类型只是 mc88000 CPU。必须实现以下函数，在 clang 中强制执行这些有效的 CPU。首先，必须声明一个字符串数组 ValidCPUNames[]，以表示 M88k 上的有效 CPU 名称。其次，fillValidCPUList() 方法将有效 CPU 名称数组填充到一个 vector 中。这个 vector 会在 isValidCPUName() 中使用，以检查提供的特定 CPU 名称是否确实对我们的 M88k 目标有效:

```
static constexpr llvm::StringLiteral ValidCPUNames[] = {
    {"generic"}, {"mc88000"}, {"mc88100"}, {"mc88110"}};

void M88kTargetInfo::fillValidCPUList(
        SmallVectorImpl<StringRef> &Values) const {
    Values.append(std::begin(ValidCPUNames),
    std::end(ValidCPUNames));
}
bool M88kTargetInfo::isValidCPUName(StringRef Name) const {
    return llvm::is_contained(ValidCPUNames, Name);
}
```

16. 接下来，实现 gettargetdefinitions() 方法。这个函数定义了前端所必需的宏，比如有效的 CPU 类型。除了__m88k__和__m88k 宏，还必须为有效的 CPU 定义相应的 CPU 宏:

```
void M88kTargetInfo::getTargetDefines(const LangOptions &Opts,
                                      MacroBuilder &Builder)
const {
    using llvm::Twine;
    Builder.defineMacro("__m88k__");
    Builder.defineMacro("__m88k");
    switch (CPU) { // For sub-architecture
    case CK_88000:
        Builder.defineMacro("__mc88000__");
        break;
    case CK_88100:
    Builder.defineMacro("__mc88100__");
    break;
    case CK_88110:
        Builder.defineMacro("__mc88110__");
        break;
    default:
        break;
    }
}
```

17. 接下来的几个函数是存根函数，但对于前端的基本支持是必需的。这包括从目标获取内置函数的函数，以及在支持目标的特定特性时查询目标的函数。现在，保留它们，并为这些函数设置默认返回值，以便稍后实现:

```
ArrayRef<Builtin::Info> M88kTargetInfo::getTargetBuiltins()
const {
    return std::nullopt;
}
bool M88kTargetInfo::hasFeature(StringRef Feature) const {
    return Feature == "M88000";
}
```

18. 这些函数之后，将在 M88k 上添加寄存器名的实现。通常，支持的寄存器名列表及其用途可以在感兴趣的特定平台的 ABI 上找到。这个实现中，将实现从 0 到 31 的主要通用寄存器，并创建一个数组来存储这些信息。寄存器别名方面，目前实现的寄存器没有别名:

```
const char *const M88kTargetInfo::GCCRegNames[] = {
    "r0", "r1", "r2", "r3", "r4", "r5", "r6", "r7",
    "r8", "r9", "r10", "r11", "r12", "r13", "r14", "r15",
    "r16", "r17", "r18", "r19", "r20", "r21", "r22", "r23",
    "r24", "r25", "r26", "r27", "r28", "r29", "r39", "r31"};

ArrayRef<const char *> M88kTargetInfo::getGCCRegNames() const {
    return llvm::makeArrayRef(GCCRegNames);
}
ArrayRef<TargetInfo::GCCRegAlias>
M88kTargetInfo::getGCCRegAliases() const {
    return std::nullopt; // No aliases.
}
```

19. 我们要实现的最后一个函数是，验证目标上的内联汇编约束的函数。此函数仅接受一个字符(表示内联程序集约束)，并相应地处理该约束。实现了一些内联汇编寄存器约束，例如：地址寄存器、数据寄存器和浮点寄存器，并且还考虑了一些常量:

```
bool M88kTargetInfo::validateAsmConstraint(
const char *&Name, TargetInfo::ConstraintInfo &info) const {
    switch (*Name) {
    case 'a': // address register
    case 'd': // data register
    case 'f': // floating point register
        info.setAllowsRegister();
        return true;
    case 'K': // the constant 1
    case 'L': // constant -1^20 .. 1^19
    case 'M': // constant 1-4:
        return true;
    }
    return false;
}
```

20. 我们通过关闭在文件开头启动的 clang 和 targets 命名空间来结束该文件:

```
} // namespace targets
} // namespace clang
```

完成 clang/lib/Basic/Targets/M88k.cpp 的实现后，必需在 clang/include/clang/Driver/Options.td 中添加 M88k 特性组和有效 CPU 类型的实现。

回想一下，之前为 M88k 目标定义了三种有效的 CPU 类型:mc88000、mc88100 和 mc88110。这些 CPU 类型也需要在 Options 中定义。因为这个文件是中心位置，定义了 clang 将接受的所有选项和标志:

13.2.2. clang 中实现对 M88k 的 ABI 支持

现在，需要在 clang 的前端中添加 ABI 支持，可以从前端生成特定于 M88k 目标的代码:

1. 从添加以下 clang/lib/CodeGen/TargetInfo.h 开始，这是一个为 M88k 目标创建代码生成信息的原型:

```
std::unique_ptr<TargetCodeGenInfo>
createM88kTargetCodeGenInfo(CodeGenModule &CGM);
```

2. 还需要将以下代码添加到 clang/lib/Basic/Targets.cpp 中，这将帮助 clang 了解 M88k 可接受的目标三元组。对于 M88k 目标，可接受的操作系统是 OpenBSD，所以 clang 接受 m88k-openbsd 作为目标三元组:

```
#include "Targets/M88k.h"
#include "Targets/MSP430.h"
...
    case llvm::Triple::m88k:
        switch (os) {
            case llvm::Triple::OpenBSD:
            return std::make_unique<OpenBSDTargetInfo<M88kTargetInfo>>(Triple,
            ↪  Opts);
        default:
            return std::make_unique<M88kTargetInfo>(Triple, Opts);
        }
    case llvm::Triple::le32:
...
```

现在，需要创建一个名为 clang/lib/CodeGen/Targets/M88k.cpp 的文件，以便可以继续 M88k 的代码生成信息和 ABI 实现。

3. clang/lib/CodeGen/Targets/M88k.cpp 中，必须添加以下必要的头文件，其中之一就是刚刚修改的 TargetInfo.h 头文件，必须指定使用 clang 和 clang::codegen 命名空间:

```
#include "ABIInfoImpl.h"
#include "TargetInfo.h"
using namespace clang;
using namespace clang::CodeGen;
```

4. 接下来，必须声明一个新的匿名空间，并将 M88kABIInfo 放入其中。M88kABIInfo 继承自 clang 中现有的 ABIInfo，并在其中包含 DefaultABIInfo。对于我们的目标，很大程度上依赖于现有的 ABIInfo 和 DefaultABIInfo，这大大简化了 M88kABIInfo 类:

```
namespace {
class M88kABIInfo final : public ABIInfo {
    DefaultABIInfo defaultInfo;
```

5. 此外，除了为 M88kABIInfo 类添加构造函数外，还添加了两个方法。computeInfo() 实现默认的 clang::CodeGen::ABIInfo 类。还有 EmitVAArg() 函数，生成代码，从传入的指针中检索参数(这是更新后的)。这主要用于可变函数支持:

```
public:
    explicit M88kABIInfo(CodeGen::CodeGenTypes &CGT)
        : ABIInfo(CGT), defaultInfo(CGT) {}
    void computeInfo(CodeGen::CGFunctionInfo &FI) const override
    {}
    CodeGen::Address EmitVAArg(CodeGen::CodeGenFunction &CGF,
                               CodeGen::Address VAListAddr,
                               QualType Ty) const override {
        return VAListAddr;
    }
};
```

6. 接下来添加 M88kTargetCodeGenInfo 类的类构造函数，从原始的 TargetCodeGenInfo 扩展而来。之后，必须关闭新创建的匿名命名空间:

```
class M88kTargetCodeGenInfo final : public TargetCodeGenInfo {
public:
    explicit M88kTargetCodeGenInfo(CodeGen::CodeGenTypes &CGT)
        : TargetCodeGenInfo(std::make_unique<DefaultABIInfo>(CGT))
{} };
}
```

7. 最后，必须添加实现来创建实际的 M88kTargetCodeGenInfo 类 std::unique_ptr，其接受一个生成 LLVM IR 代码的 CodeGenModule。这直接对应于最初添加到 TargetInfo.h 中的内容:

```
std::unique_ptr<TargetCodeGenInfo>
CodeGen::createM88kTargetCodeGenInfo(CodeGenModule &CGM) {
    return std::make_unique<M88kTargetCodeGenInfo>(CGM.getTypes());
}
```

以上就是 ABI 对前端 M88k 的支持。

13.2.3. clang 中实现对 M88k 工具链的支持

clang 中 M88k 目标集成的最后一部分将是，为我们的目标实现工具链支持。和前面一样，需要为工具链支持创建一个头文件，称这个头文件为 clang/lib/Driver/ToolChains/Arch/M88k.h:

1. 首先，必须定义 LLVM_CLANG_LIB_DRIVER_TOOLCHAINS_ARCH_M88K_H，以防止以后的多次包含，并添加任何必要的头文件供以后使用。必须声明 clang、driver、tools 和 m88k 命名空间，每个命名空间都嵌套在另一个命名空间内:

```
#ifndef LLVM_CLANG_LIB_DRIVER_TOOLCHAINS_ARCH_M88K_H
#define LLVM_CLANG_LIB_DRIVER_TOOLCHAINS_ARCH_M88K_H
#include "clang/Driver/Driver.h"
#include "llvm/ADT/StringRef.h"
#include "llvm/Option/Option.h"
#include <string>
#include <vector>
```

```
namespace clang {
namespace driver {
namespace tools {
namespace m88k {
```

2. 接下来，必须声明一个描述浮点 ABI 的枚举值，用于软浮点和硬浮点。浮点计算既可以由浮点硬件本身完成，很快；也可以通过软件模拟完成，很慢:

```
enum class FloatABI { Invalid, Soft, Hard, };
```

3. 在此之后，必须添加定义，以便通过驱动程序获得浮点 ABI，并通过 clang 的-mcpu= 和-mtune= 选项获得 CPU。还必须声明一个从驱动程序中检索目标特性的函数:

```
FloatABI getM88kFloatABI(const Driver &D, const
llvm::opt::ArgList &Args);
StringRef getM88kTargetCPU(const llvm::opt::ArgList &Args);
StringRef getM88kTuneCPU(const llvm::opt::ArgList &Args);
void getM88kTargetFeatures(const Driver &D, const
llvm::Triple &Triple, const llvm::opt::ArgList &Args,
std::vector<llvm::StringRef> &Features);
```

4. 最后，通过结束最初定义的名称空间和宏来结束头文件:

```
} // end namespace m88k
} // end namespace tools
} // end namespace driver
} // end namespace clang
#endif // LLVM_CLANG_LIB_DRIVER_TOOLCHAINS_ARCH_M88K_H
```

将实现的最后一个文件是工具链支持的 C++ 实现，位于 clang/lib/Driver/ToolChains/Arch/M88k.cpp 中:

1. 同样，将通过包含稍后将使用的必要的头文件和命名空间来开始实现，还必须包括之前创建的 M88k.h 头文件:

```
#include "M88k.h"
#include "ToolChains/CommonArgs.h"
#include "clang/Driver/Driver.h"
#include "clang/Driver/DriverDiagnostic.h"
#include "clang/Driver/Options.h"
#include "llvm/ADT/SmallVector.h"
#include "llvm/ADT/StringSwitch.h"
#include "llvm/Option/ArgList.h"
#include "llvm/Support/Host.h"
#include "llvm/Support/Regex.h"
#include <sstream>
using namespace clang::driver;
using namespace clang::driver::tools;
using namespace clang;
using namespace llvm::opt;
```

2. 接下来实现 normalizeCPU() 函数，该函数将 CPU 名称处理为 clang 中的-mcpu= 选项，每个 CPU 名称都有几个可接受的变体。此外，当用户指定-mcpu=native 时，可以针对当前主机的 CPU 类型进行编译:

```
static StringRef normalizeCPU(StringRef CPUName) {
    if (CPUName == "native") {
        StringRef CPU = std::string(llvm::sys::getHostCPUName());
        if (!CPU.empty() && CPU != "generic")
        return CPU;
    }
    return llvm::StringSwitch<StringRef>(CPUName)
    .Cases("mc88000", "m88000", "88000", "generic", "mc88000")
    .Cases("mc88100", "m88100", "88100", "mc88100")
    .Cases("mc88110", "m88110", "88110", "mc88110")
    .Default(CPUName);
}
```

3. 接下来，必须实现 getM88kTargetCPU() 函数。该函数中，给定之前在 clang/include/clang/Driver/Options.td 文件中实现的 clang CPU 名称。可以得到相应的 LLVM 名称的 M88k CPU，就是我们的目标:

```
StringRef m88k::getM88kTargetCPU(const ArgList &Args) {
    Arg *A = Args.getLastArg(options::OPT_m88000, options::OPT_
        m88100, options::OPT_m88110, options::OPT_mcpu_EQ);
    if (!A)
        return StringRef();
    switch (A->getOption().getID()) {
    case options::OPT_m88000:
        return "mc88000";
    case options::OPT_m88100:
        return "mc88100";
    case options::OPT_m88110:
        return "mc88110";
    case options::OPT_mcpu_EQ:
        return normalizeCPU(A->getValue());
    default:
        llvm_unreachable("Impossible option ID");
    }
}
```

4. 之后实现 getM88kTuneCPU() 函数。这是 clang 选项-mtune= 的行为，更改指令调度模型，以使用来自 M88k 的给定 CPU 的数据。我们只需针对当前目标 CPU 进行调优:

```
StringRef m88k::getM88kTuneCPU(const ArgList &Args) {
    if (const Arg *A = Args.getLastArg(options::OPT_mtune_EQ))
        return normalizeCPU(A->getValue());
    return StringRef();
}
```

5. 还将实现 getM88kFloatABI() 方法，该方法获取浮点 ABI。最初，将 ABI 设置为 m88k::FloatABI::Invalid 作为默认值。接下来，必须检查是否有-msoft-float 或-mhard-float 选项在命令行中设置。若指定了-msoft-float，则 ABI 设置为 m88k::FloatABI::Soft。同样，当-mhard-float 指定为 clang 时，设置 m88k::FloatABI::Hard。最后，若没有指定这些选项，则选择当前平台上的默认值，即 M88k 的硬浮点值:

```
m88k::FloatABI m88k::getM88kFloatABI(const Driver &D, const
ArgList &Args) {
    m88k::FloatABI ABI = m88k::FloatABI::Invalid;

    if (Arg *A =
        Args.getLastArg(options::OPT_msoft_float,
    options::OPT_mhard_float)) {
        if (A->getOption().matches(options::OPT_msoft_float))
            ABI = m88k::FloatABI::Soft;
        else if (A->getOption().matches(options::OPT_mhard_float))
            ABI = m88k::FloatABI::Hard;
    }
    if (ABI == m88k::FloatABI::Invalid)
        ABI = m88k::FloatABI::Hard;
    return ABI;
}
```

6. 接下来将添加 getM88kTargetFeatures() 的实现，这个函数的重要部分是作为参数传递的特征向量，所处理的唯一目标特性是浮点 ABI。从驱动程序和传递给它的参数中，将获得在前一步中实现的适当的浮点 ABI。注意，在软浮点 ABI 的特征向量中也添加了-hard-float 目标特征，多以 M88k 目前只支持硬浮点:

```
void m88k::getM88kTargetFeatures(const Driver &D,
                                 const llvm::Triple &Triple,
                                 const ArgList &Args,
                                 std::vector<StringRef> &Features) {
    m88k::FloatABI FloatABI = m88k::getM88kFloatABI(D, Args);
    if (FloatABI == m88k::FloatABI::Soft)
        Features.push_back("-hard-float");
}
```

13.3.4. 构建具有 clang 集成的 M88k 目标

我们几乎完成了将 M88k 集成到 clang 中的实现。最后一步是将新的 clang 文件添加到相应的 CMakeLists.txt 文件中，可以用 M88k 目标实现构建 clang 项目:

1. 首先，将 Targets/M88k.cpp 添加到 clang/lib/Basic/CMakeLists.txt 中。
2. 接下来，在 clang/lib/CodeGen/CMakeLists.txt 中添加 Targets/M88k.cpp。
3. 最后，将 ToolChains/Arch/M88k.cpp 添加到 clang/lib/Driver/CMakeLists.txt 中。

好了! 这就是对 M88k 目标的工具链支持的工具链实现，我们已经完成了对 M88k 的 clang 的集

成!

需要做的最后一步是用 M88k 目标构建 clang。下面的命令将构建 clang 和 LLVM 项目。对于 clang，要注意 M88k 目标。必须设置 CMake 选项-DLLVM_EXPERIMENTAL_TARGETS_TO_BUILD=M88k:

```
$ cmake -G Ninja ../llvm-project/llvm -DLLVM_EXPERIMENTAL_
TARGETS_TO_BUILD=M88k -DCMAKE_BUILD_TYPE=Release -DLLVM_ENABLE_
PROJECTS="clang;llvm"
$ ninja
```

现在应该有一个可以识别 M88k 目标的 clang 版本了! 可以通过-print-targets 选项检查 clang 支持的目标列表来确认这一点:

```
$ clang --print-targets | grep M88k
    m88k        - M88k
```

本节中，深入研究了将新后端目标集成到 clang 中并使其被识别的技术细节。下一节中，我们将探讨交叉编译的概念，详细介绍针对不同于当前主机 CPU 架构的过程。

13.3. 针对不同的 CPU 架构

现如今，许多小型计算机 (如树莓派)，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，或者需要花费太多时间。因此，编译器的一个常见需求是为不同的 CPU 架构生成代码。让主机为不同的目标编译可执行文件的整个过程，称为交叉编译。

交叉编译中，涉及到两个系统: 主机系统和目标系统。编译器在主机系统上运行，并为目标系统生成代码。为了表示系统，使用了所谓的三元组。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。此外，有关环境的附加信息经常被添加到配置字符串中。例如，x86_64-pc-win32 三元组用于在 64 位 x86 CPU 上运行的 Windows 系统。CPU 架构是 x86_64, pc 是通用厂商，win32 是操作系统，所有这些部分都用连字符连接起来。在 ARMv8 CPU 上运行的 Linux 系统使用 aarch64-unknown-linux-gnu 作为三元组，使用 aarch64 作为 CPU 架构。此外，操作系统是 linux，运行 gnu 环境。没有真正的基于 linux 的系统供应商，所以这部分是未知的。此外，对于特定目的不知道或不重要的部分经常省略:aarch64-linux-gnu 三元组描述相同的 Linux 系统。

假设开发机器在 x86-64 位 CPU 上运行 Linux，并且希望交叉编译到运行 Linux 的 ARMv8 CPU 系统。主机三元组为 x86_64-linux-gnu，目标三元组为 aarch64-linux-gnu。不同的系统有不同的特点，所以应用程序必须以可移植的方式编写; 否则，可能会出现一些问题。一些常见的问题如下:

- 字节顺序: 多字节值在内存中的存储顺序可以不同。
- 指针大小: 指针大小随 CPU 架构不同而不同 (通常为 16 位、32 位或 64 位)。C int 类型可能不够大，无法容纳指针。
- 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 double-double 算法，通过使用两个 64 位双精度值的组合来提供更高的精度。

若不注意这些问题，则即使应用程序在您的主机系统上完美运行，也可能在目标平台上异常运行或崩溃。LLVM 库在不同的平台上进行了测试，并且还包含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
- 能够为目标生成二进制文件的链接器
- 目标的头文件和库

幸运的是，Ubuntu 和 Debian 发行版都有支持交叉编译的包。我们将在下面的设置中利用这一点。gcc 和 g++ 编译器、链接器、ld 和库都可以作为预编译的二进制文件使用，它们可以生成 ARMv8 代码和可执行文件。下面的命令安装所有这些包:

```
$ sudo apt -y install gcc-12-aarch64-linux-gnu \
    g++-12-aarch64-linux-gnu binutils-aarch64-linux-gnu \
    libstdc++-12-dev-arm64-cross
```

新文件安装在/usr/aarch64-linux-gnu 目录下。该目录是目标系统的 (逻辑) 根目录，包含通常的 bin、lib 和 include 目录，交叉编译器 (aarch64-linux-gnu-gcc-8 和 aarch64-linux-gnu-g++-8) 知晓这个目录。

在其他系统上交叉编译

有些发行版 (如 Fedora) 只提供对裸机目标 (如 Linux 内核) 的交叉编译支持，但不提供用户应用程序所需的头文件和库文件，可以从目标系统复制缺失的文件。

若发行版没有附带所需的工具链，可以从源代码构建它。对于编译器，可以使用 clang 或 gcc/g++。gcc 和 g++ 编译器必须配置为为目标系统生成代码，binutils 工具需要为目标系统处理文件。此外，C 和 C++ 库需要使用这个工具链进行编译。步骤因操作系统、主机和目标架构而异。在 web 上，搜索 gcc cross-compile <architecture>，可以找到相关说明。

有了这些准备，除了一个小细节之外，可以交叉编译示例应用程序 (包括 LLVM 库) 了。LLVM 在构建过程中使用 TableGen 工具。交叉编译期间，将为目标架构编译所有内容，包括此工具。可以使用第 1 章中构建的 llvm-tblgen，也可以只编译这个工具。假设在本书的 GitHub 代码库克隆的目录中，输入以下命令:

```
$ 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。创建并输入一个构建目录。cmake 命令仅为 X86 目标创建 LLVM

的构建文件。为了节省空间和时间，完成了发布构建，启用了断言来捕获可能的错误。只有 llvm-tblgen 工具是用 ninja 编译的。

有了 llvm-tblgen 工具，就可以开始交叉编译过程了。CMake 命令行非常长，开发者可能希望将命令存储在脚本文件中。与以前版本的不同之处在于必须提供更多信息:

```
$ 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-12 \
    -DCMAKE_CXX_COMPILER=aarch64-linux-gnu-g++-12 \
    ../llvm-project/llvm
$ ninja
```

同样，需要创建一个构建目录并在运行 CMake 命令之前输入它。这些 CMake 参数中有一些以前没有使用过，需要一些解释:

- CMAKE_CROSSCOMPILING 设置为 ON，说明 CMake 正在交叉编译。
- LLVM_TABLEGEN 指定要使用的 llvm-tblgen 工具的路径。
- LLVM_DEFAULT_TARGET_TRIPLE 是目标架构的三元组。
- LLVM_TARGET_ARCH 用于 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 LLVM 应该包含代码生成器的目标列表，该列表至少应该包括目标架构。
- CMAKE_C_COMPILER 和 CMAKE_CXX_COMPILER 分别指定用于构建的 C 和 C++ 编译器。交叉编译器的二进制文件以目标三元组为前缀，并且不会让 CMake 自动找到。

使用其他参数，将请求启用断言的版本构建，并将的 tinylang 应用程序构建为 LLVM 的一部分。编译过程完成后，file 命令可以证明我们已经为 ARMv8 创建了一个二进制文件。具体来说，我们可以运行 $ file bin/ tinylang 并检查输出是否为用于 ARM aarch64 架构的 ELF 64 位对象。

使用 clang 进行交叉编译

由于 LLVM 为不同的架构生成代码，使用 clang 进行交叉编译似乎显而易见。这里的难点是 LLVM 不提供所有必需的部分——例如，缺少 C 库。正因为如此，必须混合使用 LLVM 和 GNU 工具，所以需要告诉 CMake 更多关于正在使用的环境的信息。至少，需要为 clang 和 clang++

指定以下选项:--target=<targettriple>(为不同的目标启用代码生成)，--sysroot=<path>(目标的根目录路径)，-I(搜索头文件的路径) 和-L(搜索库的路径)。
CMake 运行期间，编译一个小的应用程序时，若设置有问题，CMake 会报错。这一步足以检查你是否有一个工作环境。常见的问题是由于不同的库名称或错误的搜索路径，导致选择错误的头文件或链接，从而编译失败。

交叉编译非常复杂。根据本节的说明，将能够为您选择的架构结构交叉编译应用程序。

13.4. 总结

本章中，了解了如何创建超越指令选择的通道，如何在后端创建机器函数通道! 还了解了如何向 clang 中添加新的实验目标，以及需要对驱动程序、ABI 和工具链进行的一些更改。最后，在考虑编译器构造的最高原则时，了解了如何为另一个目标架构交叉编译应用程序。

现在我们已经学完了 LLVM 17，您已经具备了在项目中以创造性方式使用 LLVM 的知识，并探索了许多有趣的主题。LLVM 生态系统非常活跃，新功能一直在添加，所以一定要关注它的发展!

作为编译器开发人员，我们很高兴能够撰写有关 LLVM 的文章，并在此过程中发现一些新特性。也祝各位读者玩得开心!