LLVM Techniques, Tips, and Best Practices

Clang and Middle-End Libraries

Design powerful and reliable compilers using the latest libraries and tools from LLVM

Min-Yih Hsu

LLVM Techniques, Tips, and Best Practices

Clang and Middle-End Libraries

Design powerful and reliable compilers using the latest libraries and tools from LLVM

(Clang 和中端库 - 使用最新的 LLVM 库和工具设计强大且可靠的编译器)

作者: Min-Yih Hsu 許民易

译者；陈晓伟

本书概述

编译器将高级编程语言转换为低层机器可执行的代码，所以每个程序员或工程师，在职业生涯的某个时刻，都会与编译器一起优化应用程序。LLVM 为开发者提供了基础设施、库和 (开发人员构建自己的) 编译器所需的工具。使用 LLVM 的工具集，可以有效地为不同的后端生成代码，并进行优化。

本书将探索 LLVM 编译器的基础结构，并介绍如何来解决问题。我们从查看 LLVM 重要组件的结构和设计理念开始，逐步使用 Clang 库来构建分析高级源代码的工具。随着了解的深入，本书将向介绍如何处理 LLVM IR——用以转换和优化源码。了解了这些，就将能够利用 LLVM 和 Clang 创建编程语言工具，包括编译器、解释器、IDE 和源代码分析程序。

本书的最后，可以使用 LLVM 框架创建强大的工具技能，以应对现实中的各种挑战。

关键特性

- (以务实的方式) 探索 Clang，LLVM 的中端和后端
- 点亮 LLVM 的各个技能点，并掌握各种常见用例
- 通过示例应对实际的 LLVM 开发

内容纲要

- 了解 LLVM 的构建系统是如何工作的，以及如何减少构建资源
- 掌握使用 LLVM 的 LIT 框架运行自定义测试的方法
- 为 Clang 构建不同类型的插件和扩展
- 基于 Clang 自定义工具链和编译器标志
- 为 PassManager 写 LLVM Pass
- 了解如何检查和修改 LLVM IR
- 了解如何使用 LLVM 的配置文件引导优化 (PGO) 框架
- 创建自定义 (编译器) 消毒器

本书适用于所有具有 LLVM 工作经验的软件工程师。如果你是一名学术研究者，这本书将助你在短时间内学习有用的 LLVM 技能，使你能够快速构建项目原型。编程语言爱好者也会发现这本书中的内容，在 LLVM 的帮助下构建一种新的编程语言也十分有趣。

作者简介

Min-Yih "Min" Hsu 是加州大学欧文分校计算机科学博士研究生。他的研究集中在编译器工程、代码优化、高级硬件架构和系统安全。2015 年起，他一直是 LLVM 社区的活跃成员，并贡献了许多补丁。他还致力于通过各种途径倡导 LLVM 和编译器工程，比如写博客和发表演讲。在业余时间，他喜欢了解各种不同的咖啡豆和煮咖啡的方法。

我要感谢所有支持过我的人，特别是家人和导师。
还要感谢 LLVM 社区不论出身的包容和善待每一位成员。

审评者介绍

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

他在浦那工程学院获得了技术学士学位。到目前为止，他的工作主要与编译器有关，并他对编译器的性能方面特别感兴趣。他曾致力于 DSP 图像处理语言的研究，使用 LLVM 的模块化特性可以根据编译器的需求快速实现。然而，LLVM 的文档比较分散，他希望这本书可以为 LLVM 编译器基础架构提供一种综合性的概述。

本书相关

- Github 翻译地址：
  https://github.com/xiaoweiChen/LLVM-Techniques-Tips-and-Best-Practies

前言

大多数程序员的开发流程都有编译器——或者某种形式的编译技术。现代编译器不仅将高级编程语言转换为低层机器码，而且在优化所编译速度、大小甚至内存占用方面也扮演着关键性角色。要有这些特性，构建可使用的编译器就是一项具有挑战性的任务。

LLVM 是一个编译器优化和代码生成的框架，提供了构建块，大大减少了开发人员创建高质量优化编译器和编程语言工具的工作。Clang 是该公司最著名的产品之一，是 C 族语言编译器，可构建数千个广泛使用的软件，包括谷歌 Chrome 浏览器和 iOS 应用程序。LLVM 也用于许多不同的编程语言的编译器中，比如 Apple 的 Swift。毫不夸张地说，可以用 LLVM 创建一种新的编程语言，就可以成为时下最热门的话题之一。

LLVM 提供了广泛的特性，有成百个库和成千个 API，包括优化关键功能到实际实用程序当中。本书中，为 LLVM 中两个最重要的子系统——Clang 和中端库提供了一个完整的开发指南。首先介绍几个组件和开发最佳实践，可以积累一些 LLVM 的开发经验，再了解如何使用 Clang 进行开发。更具体地说，我们将专注于增强和定制 Clang 中的功能。本书的最后一部分，将了解有关 LLVM IR 开发的关键知识。这包括如何用最新的语法编写 LLVM Pass，以及如何处理不同的 IR 结构。还会有几个实用例程，可以极大地提高 LLVM 开发的生产率。注意，在本书中不假设任何特定的 LLVM 版本——试图保持最新，并囊括来自 LLVM 源代码的最新特性。

本书每一章都提供了一些代码片段和项目示例。可以从本书的 GitHub 存储库中下载它们，并尝试进行修改。

适读人群

本书适用于所有具有 LLVM 工作经验的软件工程师，本书会提供了简明的开发指南和参考。如果你是一名学术研究者，这本书将助你在短时间内学习有用的 LLVM 技能，使你能够快速构建项目原型。编程语言爱好者也会发现这本书中的内容，在 LLVM 的帮助下构建一种新的编程语言也十分有趣。

本书内容

第 1 章，使用有限的资源构建 LLVM。简要介绍了 LLVM，并介绍如何在不过多消耗 CPU、内存资源和磁盘空间的情况下构建 LLVM。这为以后的开发周期和更流畅的体验奠平了道路。

第 2 章，探索 LLVM 的构建系统。介绍如何编写用于源码树内和树外 LLVM 开发的 CMake 构建脚本，将了解到如何利用 LLVM 的自定义构建系统来编写更具表现力和健壮性的构建脚本。

第 3 章，LLVM LIT 测试。介绍了如何使用 LLVM 的 LIT 基础架构运行测试。本章不仅可以更好地理解 LLVM 源码树中测试的工作原理，还能将这种直观的、可扩展的测试结构集成到其他项目中。

第 4 章，TableGen 开发。介绍了如何编写 TableGen——一种由 LLVM 发明的**特殊领域特定语言 (DSL)**。这里使用 TableGen 作为处理结构化数据的工具，可以了解在 LLVM 之外如何使用 TableGen。

第 5 章，探索 Clang 架构。标志着关于 Clang 主题的开始。本章为概述了 Clang，特别是编译流程，并介绍了各个组件在 Clang 的编译流程中的作用。

第 6 章，扩展预处理器。展示了 Clang 的预处理器架构，以及如何不需要修改 LLVM 源代码树中的任何代码，开发一个插件来扩展其功能。

第 7 章，处理 AST。介绍了如何在 Clang 中使用**抽象语法树 (AST)** 进行开发。包括了解使用 AST 的内存表示的主题，以及创建插件的教程，该插件将自定义的 AST 处理逻辑插入到编译流中。

第 8 章，使用编译器标志和工具链。涵盖了向 Clang 添加自定义编译器标志和工具链的步骤，以及如何让 Clang 支持新特性或新平台。

第 9 章，使用 PassManager 和 AnalysisManager。标志着对 LLVM 中端库讨论的开始。本章的重点是编写一个 LLVM Pass——使用 PassManager 语法，以及如何通过 AnalysisManager 访问程序分析数据。

第 10 章，处理 LLVM IR。这是一个很大的章节，包含了关于 LLVM IR 的各种核心知识，包括 LLVM IR 的内存表示结构和使用不同的 IR 单元 (如函数、指令和循环) 等技能。

第 11 章，准备相关的工具。介绍了一些实用程序，在使用 LLVM IR 时，用以提高工作效率 (如有较好的调试经验)。

第 12 章，学习 LLVM IR 表达式。展示表达式如何在 LLVM IR 上进行工作。它涵盖了两个主要的用例：消毒器和**数据引导优化 (PGO)**。对于前者，将了解如何创建自定义消毒器。对于后者，将了解如何在 LLVM Pass 中利用 PGO 数据。

编译环境

这本书旨在向您介绍 LLVM 的最新特性，因此我们鼓励您在 12.0 版本之后使用 LLVM，或使用开发分支 (即主分支)。

假设您正在 Linux 或 Unix 系统 (包括 macOS) 上工作。本书中的工具和示例命令大多是在命令行界面中运行的，您可以自由地使用任何代码编辑器或 IDE 来编写代码。

书中涉及的软件/硬件	操作系统
LLVM 版本高于或等于 12.x	macOS 或 Linux(任意衍生版)
GCC 5.1 或更高版本, 或者是支持 C++14 的任意编译器	
CMake 3.13.4 或更高版本	
Python 3.x	
Ninja Build 或 GNU Make	
Graphviz	

在第 1 章，使用有限的资源构建 LLVM 中，将详细介绍如何从源代码构建 LLVM。

如果你正在使用这本书的数字版本，我们建议自己输入代码或通过 GitHub 库访问代码 (链接在下一节中提供)。这样做将帮助您避免与复制和粘贴代码相关的任何潜在错误。

下载示例

您可以从 GitHub 网站https://github.com/PacktPublishing/LLVM-Techniques-Tips-and-Best-Practices-Clang-and-Middle-End-Libraries下载本书的示例代码文件。如果代码有更新，它将在现有的 GitHub 库中更新。

我们还有其他的代码包，还有丰富的书籍和视频目录，都在https://github.com/PacktPublishing/。去看看吧!

下载彩图

我们还提供了一个 PDF 文件，其中包含了本书中使用的屏幕截图/图表的彩色图像。可以在这里下载：https://static.packt-cdn.com/downloads/9781838824952_ColorImages.pdf。

联系方式

我们欢迎读者的反馈。

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

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

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

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

欢迎评论

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

想要了解 Packt 的更多信息，请访问packt.com。

目录

第一部分：构建系统和 LLVM 的工具 12
第 1 章　使用有限的资源构建 LLVM 13
1.1. 相关准备 13
1.2. 使用工具减少构建资源 14
1.2.1 使用 Ninja 替换 GNU Make 14
1.2.2 避免使用 BFD 连接器 15
1.3. 调整 CMake 参数 16
1.3.1 选择正确的构建类型 16
1.3.2 避免构建所有目标 17
1.3.3 构建动态库 17
1.3.4 拆解调试信息 18
1.3.5 构建优化版的 llvm-tblgen 18
1.3.6 使用新 PassManager 和 Clang 19
1.4. 使用 GN 获得更快的处理时间 19
1.5. 总结 21
1.6. 扩展阅读 21
第 2 章　探索 LLVM 的构建系统 22
2.1. 相关准备 22
2.2. LLVM 的 CMake 指令表 22
2.2.1 使用 CMake 添加新的库 22
2.2.2 使用 CMake 函数添加可执行文件和工具 24
2.2.3 使用 CMake 函数添加 Pass 插件 25
2.3. 如何使用 CMake 集成 LLVM 25
2.4. 总结 27
第 3 章　LLVM LIT 测试 28
3.1. 相关准备 28
3.2. 在源码树外项目中使用 LIT 29
3.2.1 准备示例项目 29
3.2.2 对 LIT 进行配置 31
3.2.3 LIT 的内部构件 33
3.3. FileCheck 技巧 34

3.3.1 准备示例项目 . . . 34
3.3.2 书写 FileCheck 指令 . . . 35
3.4. 探索 TestSuite 框架 . . . 39
3.4.1 准备示例项目 . . . 39
3.4.2 将代码导入 llvm-test-suite . . . 40
3.5. 总结 . . . 42
3.6. 扩展阅读 . . . 42
第 4 章 TableGen 开发 . . . 43
4.1. 相关准备 . . . 43
4.2. 介绍 TableGen 语法 . . . 43
4.2.1 布局和记录 . . . 44
4.2.1 叹号操作符 . . . 45
4.2.3 多记录 . . . 46
4.2.4 有向无环图 (DAG) 数据类型 . . . 47
4.3. 用 TableGen 做甜甜圈 . . . 47
4.4. 用 TableGen 后端打印食谱 . . . 51
4.4.1 TableGen 的高层工作流 . . . 52
4.4.2 编写 TableGen 后端 . . . 53
4.4.3 集成 RecipePrinter TableGen 后端 . . . 56
4.5. 总结 . . . 58
4.6. 扩展阅读 . . . 59
第二部分：LLVM 的前端开发 . . . 60
第 5 章 探索 Clang 架构 . . . 61
5.1. 相关准备 . . . 61
5.2. 了解 Clang 的子系统及其作用 . . . 62
5.2.1 驱动 . . . 63
5.2.2 前端 . . . 64
5.2.3 LLVM，组译器和连接器 . . . 65
5.3. 探索 Clang 工具的功能和扩展选项 . . . 66
5.3.1 FrontendAction 类 . . . 66
5.3.2 Clang 插件 . . . 67
5.3.3 libTooling 和 Clang 工具 . . . 68
5.4. 总结 . . . 69
5.5. 扩展阅读 . . . 69
第 6 章 扩展预处理器 . . . 70
6.1. 相关准备 . . . 70
6.2. 使用 SourceLocation 和 SourceManager . . . 71
6.2.1 SourceLocation . . . 71
6.2.2 SourceManager . . . 72

6.3. 解预处理器和词法分析器的基本知识 . . . 72
6.3.1 Clang 中预处理器和词法分析器的作用 . . . 72
6.3.2 Token 类 . . . 73
6.3.3 处理宏 . . . 76
6.4. 定制开发预处理器的插件和回调 . . . 78
6.4.1 项目的目标和准备工作 . . . 78
6.4.2 实现自定义 pragma . . . 80
6.4.3 实现自定义预处理器的回调 . . . 82
6.5. 总结 . . . 84
6.6. 练习 . . . 84
第 7 章 处理 AST . . . 86
7.1. 相关准备 . . . 86
7.2. Clang 中的 AST . . . 87
7.2.1 Clang 的 AST 内存表示 . . . 87
7.2.2 Clang AST 中的类型 . . . 90
7.2.3 ASTMatcher . . . 91
7.3. 编写 AST 插件 . . . 96
7.3.1 项目概述 . . . 97
7.3.2 打印诊断消息 . . . 98
7.3.3 创建 AST 插件 . . . 101
7.4. 总结 . . . 110
第 8 章 使用编译器标志和工具链 . . . 111
8.1. 相关准备 . . . 111
8.2. Clang 中的驱动程序和工具链 . . . 112
8.3. 添加自定义驱动标志 . . . 114
8.3.1 项目概述 . . . 114
8.3.2 声明自定义驱动标志 . . . 116
8.3.3 翻译自定义驱动标志 . . . 118
8.3.4 向前端传递标志 . . . 123
8.4. 添加自定义工具链 . . . 124
8.4.1 项目概述 . . . 125
8.4.2 创建工具链并添加自定义的包含路径 . . . 127
8.4.3 创建自定义汇编阶段 . . . 129
8.4.4 创建自定义链接阶段 . . . 131
8.4.5 验证自定义工具链 . . . 134
8.5. 总结 . . . 135
8.6. 练习 . . . 135
第三部分：LLVM 的中端开发 . . . 136
第 9 章 使用 PassManager 和 AnalysisManager . . . 137

9.1. 相关准备 . . . 137
9.2. 为新 PassManager 写一个 LLVM Pass . . . 138
9.2.1 项目概述 . . . 139
9.2.2 编写 StrictOpt Pass . . . 141
9.3. 使用新 AnalysisManager . . . 145
9.3.1 项目概述 . . . 145
9.3.2 编写 HaltAnalyzer Pass . . . 146
9.4. 新 PassManager 中的设施 . . . 149
9.4.1 打印通道流水细节 . . . 150
9.4.2 打印每次 Pass 对 IR 的修改 . . . 151
9.4.3 将 Pass 流水一分为二 . . . 153
9.5. 总结 . . . 154
9.6. 问题 . . . 154
第 10 章 处理 LLVM IR . . . 155
10.1. 相关准备 . . . 155
10.2. 学习 LLVM IR 基础知识 . . . 156
10.2.1 迭代不同的 IR 单元 . . . 158
10.2.2 迭代指令 . . . 158
10.2.3 迭代基本块 . . . 160
10.2.4 迭代调用图 . . . 165
10.2.5 了解 GraphTraits . . . 165
10.3. 值和指令 . . . 168
10.3.1 SSA . . . 168
10.3.2 值 . . . 170
10.3.3 指令 . . . 172
10.4. 循环 . . . 177
10.4.1 LLVM 中的循环表示 . . . 177
10.4.2 LLVM 中的循环设施 . . . 181
10.5. 总结 . . . 184
第 11 章 配套工具 . . . 185
11.1. 相关准备 . . . 185
11.2. 打印诊断消息 . . . 186
11.3. 收集统计信息 . . . 188
11.3.1 使用 `Statistic` 类 . . . 188
11.3.2 使用优化注释 . . . 191
11.4. 添加耗时测量 . . . 197
11.4.1 Timer 类 . . . 197
11.4.2 收集时间轨迹 . . . 199
11.5. LLVM 中的错误处理工具 . . . 202

11.5.1 Error 类 . . . 202
11.6. 了解 Expected 和 ErrorOr 类 . . . 206
11.6.1 Expected 类 . . . 206
11.6.2 ErrorOr 类 . . . 207
11.7. 总结 . . . 208
第 12 章 LLVM IR 表达式 . . . 209
12.1. 相关准备 . . . 209
12.2. 开发杀毒器 . . . 210
12.2.1 使用地址杀毒器的例子 . . . 210
12.2.2 创建循环计数器杀毒器 . . . 212
12.3. PGO . . . 226
12.3.1 基于检测的 PGO . . . 227
12.3.2 基于采样的 PGO . . . 227
12.3.3 使用性能数据进行分析 . . . 236
12.4. 总结 . . . 239
练习答案 . . . 240

第一部分：构建系统和 LLVM 的工具

您将学习在源码树内和树外开发 LLVM 构建系统的高级技能。

本节包括以下几章:

- 第 1 章，使用有限的资源构建 LLVM
- 第 2 章，探索 LLVM 的构建系统
- 第 3 章，LLVM LIT 测试
- 第 4 章，TableGen 开发

第 1 章 使用有限的资源构建 LLVM

LLVM 是非常受欢迎的编译器优化和代码生成框架，许多令人惊叹的工业和学术项目都会使用 LLVM，比如 JavaScript 引擎和机器学习 (ML) 框架中的即时 (JIT) 编译器。它是构建编程语言和二进制文件的非常好用的工具箱。然而，这个项目很健壮，但其学习资源非常零散，而且也没有形成很好的文档。即使是对于一些有 LLVM 经验的开发人员，学习曲线都相当陡峭。本书旨在通过实用的方式提供 LLVM 中常见和重要的知识来解决这些问题——展示一些有用的工程技巧，指出不太为人所知，但很使用的特性，并会举例说明。

作为一个 LLVM 开发人员，从源代码构建 LLVM 第一件事。考虑到现在 LLVM 的规模，这个任务可能需要几个小时才能完成。更糟糕的是，对源码进行修改，并重新构建项目可能也会花很长的时间。因此，为了节省资源，特别是时间，了解如何使用正确的工具，以及如何为项目找到最佳的构建配置至关重要。

本章中，我们将讨论以下主题:

- 使用工具减少构建资源
- 通过调整 CMake 参数节省构建资源
- GN(一种可选的 LLVM 构建系统) 的使用，以及其优缺点

1.1. 相关准备

在撰写本书时，构建 LLVM 有几个软件方面的需求:

- 支持 C++14 的编译器
- CMake
- CMake 支持的构建系统之一，如 GNU Make 或 Ninja
- Python(2.7 也不错，但强烈推荐使用 3.x)
- zlib

这些项目的确切版本不时发生变化。https://llvm.org/docs/GettingStarted.html#software上有更多的信息。

本章假定您以前构建过 LLVM。如果没构建过，可以参考以下步骤进行构建:

1. 从 GitHub 获取 LLVM 源码副本:

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

2. 默认分支的构建通常没有错误。如果您想使用更稳定的版本，比如版本 10.X，可以使用如下命令:

```
$ git clone -b release/10.x \
   https://github.com/llvm/llvmproject
```

3. 然后，创建一个构建文件夹，在那里使用 CMake 命令，所有的构建组件将放置在这个文件夹中。可以使用以下命令:

```
$ mkdir .my_build
$ cd .my_build
```

1.2. 使用工具减少构建资源

正如本章开头所说，如果用默认的 (CMake) 配置来构建 LLVM，可以通过 CMake 构建整个项目，整个过程可能需要几个小时才能完成:

```
$ cmake ../llvm
$ make all
```

可以通过使用一些工具和改变环境来避免这么久的耗时。本节中，将介绍一些指导原则，以帮助您选择正确的工具和配置，从而加快构建时间，以及改善内存占用。

1.2.1 使用 Ninja 替换 GNU Make

第一个改进是使用 Ninja(https://ninja-build.org) 而不是 GNU Make，后者是 CMake 在主要 Linux/Unix 平台上生成的默认构建系统。

以下是在系统上设置 Ninja 构建的步骤:

1. 在 Ubuntu 上，可以通过以下命令来安装 Ninja:

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

Ninja is also available in most Linux distributions.

2. 然后，为 LLVM 构建调用 CMake 时，需要额外添加一个参数:

```
$ cmake -G "Ninja" ../llvm
```

3. 最后，使用下面的命令进行构建:

```
$ ninja all
```

Ninja 在大型代码库 (如 LLVM) 上的运行速度明显快于 GNU Make。Ninja 的快速运行速度的秘密是，虽然像 Makefile 这样的大多数构建脚本都是手工编写的，但 Ninja 的构建脚本的语法是 build.ninja，更类似于汇编代码 (它不应该由开发人员编辑，而是由其他高级构建系统 (如 CMake) 生成)。Ninja 使用类似于程序集的构建脚本，这使得它能够在底层进行优化，并消除许多冗余，比如: 在调用构建时，解析速度会变慢。Ninja 在构建目标之间产生更好的依赖关系方面也能很好的完成。

Ninja 会根据其并行度做出决定，以智能的方式绝对最大程度并行执行多个作业。如果想显式地分配工作线程的数量，GNU Make 使用的命令行选项仍然可以在这里使用:

```
$ ninja -j8 all
```

接下来，让我们看看如何避免使用 BFD 链接器。

1.2.2 避免使用 BFD 连接器

第二个改进是使用 BFD 链接器以外的链接器，这是在大多数 Linux 系统中使用的默认链接器。BFD 连接器是 Unix/Linux 系统上最成熟的连接器，但在速度或内存消耗方面并没有得到优化。这将造成性能瓶颈，特别是对于像 LLVM 这样的大型项目。因为与编译阶段不同，链接阶段很难进行文件级的并行化。更不用说，在建立 LLVM 时，BFD 链路的峰值内存消耗通常需要 20GB 左右，这对内存较少的计算机造成了负担。幸运的是，目前至少有两个连接器提供了良好的单线程性能和较低的内存消耗:GNU gold 连接器和 LLVM 自己的连接器 LLD。

gold 链接器最初是由谷歌开发的，并捐赠给 GNU 的 binutils。在现代 Linux 发行版中，默认情况下应该将其放在 binutils 包中。LLD 是 LLVM 的子项目之一，具有更快的连接速度和实验性的并行连接技术。一些 Linux 发行版 (例如较新的 Ubuntu 版本) 已经在包存储库中包含了 LLD。您也可以从 LLVM 的官方网站下载预构建版本。

要使用 gold 链接器或 LLD 来构建 LLVM，添加一个额外的 CMake 参数，使用你想要使用的链接器的名称。

对于 gold 链接器，可以使用以下命令:

```
$ cmake -G "Ninja" -DLLVM_USE_LINKER=gold ../llvm
```

类似地，对于 LLD，使用如下命令:

```
$ cmake -G "Ninja" -DLLVM_USE_LINKER=lld ../llvm
```

限制用于链接的并行线程的数量

限制用于链接的并行线程的数量是减少 (峰值) 内存消耗的另一种方法。你可以在 cmake 过程中，通过指定 `LLVM_PARALLEL_LINK_JOBS=<N>` 变量来实现，其中 `N` 是期望的工作线程数。

因此，只要使用不同的工具，构建时间就可以显著减少。下一节中，我们将通过调整 LLVM 的 CMake 参数来提高构建速度。

1.3. 调整 CMake 参数

本节将向您展示 LLVM 构建系统中一些最常见的 CMake 参数，可以帮助您以最高效率实现自定义构建。

开始之前，您应该有一个配置了 cmake 的构建文件夹。以下大部分小节将修改构建文件夹中的 CMakeCache.txt。

1.3.1 选择正确的构建类型

LLVM 使用 CMake 提供的几种预定义构建类型。其中最常见的类型如下:

- `Release`: 如果没有指定任何类型，这就是默认的构建类型。它将采用最高的优化级别 (通常是-O3)，并消除大多数调试信息。通常，这种构建类型会使构建速度稍微慢一些。
- `Debug`: 此构建类型将在不应用任何优化 (即-O0) 的情况下进行编译，保存所有的调试信息。注意，这将生成大量的工件，通常占用约 20GB 的空间，所以在使用这种构建类型时，请确保有足够的存储空间。这通常会使构建速度稍微快一些，因为没有执行优化。
- `RelWithDebInfo`: 这种构建类型尽可能多地应用编译器优化 (通常是-O2)，并保留所有调试信息。这是一个在空间消耗、运行速度和可调试性之间进行平衡的选项。

可以选择其中一个使用 `CMAKE_BUILD_TYPE CMAKE` 变量。例如，要使用 `RelWithDebInfo` 类型，可以使用以下命令:

```
$ cmake -DCMAKE_BUILD_TYPE=RelWithDebInfo ···
```

建议首先使用 `RelWithDebInfo`(如果你稍后要调试 LLVM)。现代编译器在改进优化后的二进制文件中，提升调试信息的质量方面已经走了很长一段路。所以，可以先尝试一下，以避免不必要的存储空间浪费。如果不太顺利，可以使用 `Debug` 方式。

除了配置构建类型，`LLVM_ENABLE_ASSERTIONS` 是另一个 CMake(布尔型) 参数，它控制断言 (也就是，`assert(bool predicate)` 函数，如果 `predicate` 参数不为真，将终止程序) 是否启用。默认情况下，该标志只有在构建类型为 `Debug` 时才为真，但可以手动打开它，即使是在其他构建类型中，也可以强制执行更严格的检查。

1.3.2 避免构建所有目标

LLVM 支持的 (硬件) 目标数量在过去几年中迅速增长。在写这本书的时候，有近 20 个官方支持的目标。它们中的每一个都处理重要的任务，如本机代码生成，需要大量的时间来构建。然而，在同一时间完成所有这些目标的机会很低。因此，可以通过 CMake 参数 `LLVM_TARGETS_TO_BUILD` 选择要构建的目标子集。例如，只构建 X86 目标，我们可以使用以下命令:

```
$ cmake -DLLVM_TARGETS_TO_BUILD="X86" ···
```

也可以使用分号分隔的列表指定多个目标，如下所示:

```
$ cmake -DLLVM_TARGETS_TO_BUILD="X86;AArch64;AMDGPU" ···
```

用双引号包围目标列表!

在某些 shell 中，例如 BASH，分号是命令的结束符号。因此，如果没有双引号的目标列表，CMake 命令的其余部分将被切断。

接下来，让我们看看如何调整 CMake 参数来构建动态库。

1.3.3 构建动态库

LLVM 最具标志性的特点之一是其**模块化设计**。每个组件、优化算法、代码生成和实用程序库都放入自己的库中，开发人员可以根据它们的使用情况链接各自的库。默认情况下，每个组件都构建为**静态库** (Unix/Linux 下是 `*.a`，Windows 下是 `*.lib`)。然而，静态库有以下缺点:

- 链接静态库通常比链接动态库花费更多的时间 (Unix/Linux 下是 `*.so`,Windows 下是 `*.dll`)。
- 如果多个可执行文件链接到同一组库上，就像许多 LLVM 工具那样，当使用静态库方法时，可执行文件的总大小将比动态库大得多 (因为每个可执行文件都有一个这些库的副本)。
- 当使用调试器 (例如 GDB) 调试 LLVM 程序时，通常会在一开始就花费相当多的时间来加载静态链接的可执行文件，从而降低了调试体验。

因此，建议在开发阶段通过将 CMake 参数 `BUILD_SHARED_LIBS` 将每个 LLVM 组件构建为动态库:

```
$ cmake -DBUILD_SHARED_LIBS=ON ···
```

这将为您节省大量的存储空间，并加快构建过程。

1.3.4 拆解调试信息

当您在调试模式下构建程序时——例如，在使用 GCC 和 Clang 时添加`-g` 标志——默认情况下，生成的二进制文件包含一个存储调试信息的部分。这些信息对于使用调试器 (例如，GDB) 调试该程序至关重要。LLVM 是一个庞大而复杂的项目，所以当你在调试模式下构建它时——使用`CMAKE_BUILD_TYPE=Debug`——编译的库和可执行文件会带来大量的调试信息，占用大量的磁盘空间。这会导致以下问题:

- 由于 C/C++ 的设计，相同调试信息的多个副本可能嵌入到不同的对象文件中 (例如，头文件的调试信息可能嵌入到包含它的每个库中)，这浪费了大量的磁盘空间。
- 链接阶段，链接器需要将对象文件及其相关的调试信息加载到内存中，这意味着如果对象文件包含大量的调试信息，内存压力将增加。

为了解决这些问题，LLVM 中的构建系统允许将调试信息从原始对象文件中分离到单独的文件中。通过将调试信息从目标文件中分离出来，同一个源文件的调试信息压缩到了另一个地方，从而避免了不必要的重复创建，节省了大量的磁盘空间。此外，由于调试信息不再是对象文件的一部分，连接器不再需要将它们加载到内存中，从而节省了大量的内存资源。最后，这个特性还可以提高增量构建的速度——也就是说，在 (小的) 代码更改之后重新构建项目——因为我们只需要在单个地方更新修改过的调试信息。

要使用这个特性，请使用 `LLVM_USE_SPLIT_DWARF`CMake 变量:

```
$ cmake -DCMAKE_BUILD_TYPE=Debug -DLLVM_USE_SPLIT_DWARF=ON ···
```

请注意，这个 CMake 变量只适用于使用 DWARF 调试格式的编译器，包括 GCC 和 Clang。

1.3.5 构建优化版的 llvm-tblgen

TableGen 是一种**领域特定语言 (DSL)**，用于描述结构化数据，这些数据将转换成相应的 C/C++ 代码,作为 LLVM 构建过程的一部分 (将在后面的章节中了解更多),转换工具为 `llvm-tblgen`。也就是说，`llvm-tblgen` 的运行时间会影响到 LLVM 本身的构建时间。因此，如果不开发 TableGen 部分,那么构建一个优化的 `llvm-tblgen` 版本是一个好主意,无论全局构建类型 (即 `CMAKE_BUILD_TYPE`)，使 `llvm-tblgen` 运行得更快，就可以缩短整体构建时间。

例如，下面的 CMake 命令将创建一个构建配置，将构建除 `llvm-tblgen` 可执行文件之外的所有内容的调试版本，而 `llvm-tblgen` 可执行文件将以优化版本进行构建:

```
$ cmake -DLLVM_OPTIMIZED_TABLEGEN=ON -DCMAKE_BUILD_TYPE=Debug ···
```

最后，将了解如何使用 Clang 和新 PassManager。

1.3.6 使用新 PassManager 和 Clang

Clang 是 LLVM 官方的 C 家族前端 (包括 C、C++ 和 Objective-C)，使用 LLVM 的库来生成机器代码，这些代码是由 LLVM 中最重要的子系统之一——PassManager 组织起来的。PassManager 将优化和代码生成所需的所有任务 (即 Pass) 放在一起。

在第 9 章中将引入 LLVM 的新 PassManager，它将在未来的某个时候取代旧 PassManager。与旧 PassManager 相比，新的 PassManager 的运行速度更快。这个优势间接地为 Clang 带来了更好的运行时性能。因此，如果我们使用 Clang 构建 LLVM 的源代码树，并且启用了新的 PassManager，编译速度将会更快。大多数主流的 Linux 发行包存储库已经包含了 Clang。如果你想要一个更稳定的 PassManager 实现，建议使用 Clang 6.0 或更高版本。可以通过设置 CMake 变量 `LLVM_USE_NEWPM` 用新 PassManager 构建 LLVM，如下所示:

```
$ env CC=`which clang` CXX=`which clang++` \
    cmake -DLLVM_USE_NEWPM=ON ···
```

LLVM 是一个庞大的项目，需要花费大量时间来构建，前两节介绍了一些提高构建速度的有用技巧和提示。下一节中，我们将介绍一个用于构建 LLVM 的替代构建系统。与默认的 CMake 构建系统相比，它有一些优势，这意味着它将更适合于某些场景。

1.4. 使用 GN 获得更快的周转时间

CMake 的可移植和灵活性非常好，已经经过了许多工业项目的实战测试。然而，当涉及到重新配置时，它就会有一些严重的问题。正如我们在前几节中看到的，已生成的构建文件，可以通过编辑 build 文件夹中的 CMakeCache.txt 文件来修改一些 CMake 参数。当你再次调用构建命令时，CMake 会重新配置构建文件。如果在源文件夹中编辑 CMakeLists.txt 文件，同样的重新配置也会出现。CMake 的重配置过程主要有两个缺点:

- 在某些系统中，CMake 的配置过程非常缓慢。即使是理论上只运行部分流程的重构，有时仍然需要很长时间。
- 有时 CMake 会无法解决不同变量和构建目标之间的依赖关系，所以你的更改不会反映出这一点。最糟糕的情况是，它会悄无声息地失败，并花费您很长时间来查找问题。

Ninja,更广为人知的名字是 GN,是谷歌的许多项目使用的一个构建文件生成器,比如 Chromium。GN 从它自己的描述语言生成 Ninja 文件。它具有快速配置时间和可靠的参数管理的良好声誉。LLVM 自 2018 年末 (大约版本 8.0.0) 以来，已经将 GN 支持作为一种可选的 (实验性的) 构建方法。如果您的开发对构建文件进行了更改，或者您想在短时间内尝试不同的构建选项，那么 GN 尤其有用。

使用 GN 构建 LLVM 的步骤如下:

1. LLVM 的 GN 支持位于 llvm/utils/gn 文件夹中。切换到该文件夹后，运行以下 get.py 脚本，在本地下载 GN 的可执行文件:

```
$ cd llvm/utils/gn
$ ./get.py
```

使用特定版本的 GN

如果希望使用自定义 GN 可执行文件，而不是 get.py 获取的可执行文件，只需将特定版本的 GN 放入系统的 PATH 中。如果您想知道还有哪些其他 GN 版本可用，您可以在https://dev.chromium.org/developers/how-tos/install-depot-tools查看关于安装 depot_tools 的信息。

2. 在同一个文件夹中使用 gn.py 生成构建文件 (本地版本的 gn.py 只是一个包装器，用于设置基本环境):

```
$ ./gn.py gen out/x64.release
```

out/x64.release 是构建文件夹的名称。通常，GN 用户的文件命名规则为 `<architecture>.<build type>.<other features>`。

3. 最后，可以切换到构建文件夹并启动 Ninja:

```
$ cd out/x64.release
$ ninja <build target>
```

4. 或者，使用`-C` 选项:

```
$ ninja -C out/x64.release <build target>
```

您可能已经知道初始构建文件生成过程非常快。现在，如果您想更改一些构建参数，请找到 args.gn 文件，在 build 文件夹下 (out/x64.release/args.gn)。如果想改变构建类型来调试和改变目标来构建 (修改 `LLVM_TARGETS_TO_BUILD` CMake 参数) 到 X86 和 AArch64。建议使用以下命令启动一个编辑器来编辑 args.gn:

```
$ ./gn.py args out/x64.release
```

在 args.gn 中，输入如下内容:

```
# Inside args.gn
is_debug = true
llvm_targets_to_build = ["X86", "AArch64"]
```

保存并退出编辑器后，GN 将执行一些语法检查并重新生成构建文件 (当然，您可以不使用 gn 命令编辑 args.gn，这样在调用 ninja 命令之前，构建文件不会重新生成)，这种重新生成/重新配置也会很快。最重要的是，不会有任何不确定的行为。由于 GN 的语言设计，可以很容易地分析不同构建参数之间的关系，几乎没有歧义。

GN 的构建参数列表可以通过以下命令找到:

```
$ ./gn.py args --list out/x64.release
```

不幸的是，在写这本书的时候，仍然有很多 CMake 参数没有移植到 GN 中。GN 不是 LLVM 现有的 CMake 构建系统的替代品，但它是一个替代方案。尽管如此，如果希望在涉及许多构建配置更改的开发中获得快速的处理时间，那么 GN 仍然是一个不错的构建方法。

1.5. 总结

在构建用于代码优化和代码生成的工具时，LLVM 是一个有用的框架。然而，其代码库的大小和复杂性导致了大量的构建时间。本章提供了一些加快 LLVM 构建时间的技巧，包括使用不同的构建工具，选择正确的 CMake 参数，甚至采用一个非 CMake 的构建系统。当使用 LLVM 进行开发时，这些技能减少了不必要的资源浪费并提高了您的生产力。

下一章，我们将深入研究 LLVM 基于 CMake 构建的基础设施，并向展示，如何构建在许多不同开发环境中至关重要的系统特性和指南。

1.6. 扩展阅读

查看 LLVM 使用的 CMake 变量的完整列表 https://llvm.org/docs/CMake.html#frequently-used-CMakevariables。

可以在以下网站了解更多关于 GN 的信息 https://gn.googlesource.com/gn。快速开始引导页地址为 https://gn.googlesource.com/gn/+/master/docs/quick_start.md。

第 2 章 探索 LLVM 的构建系统

在前一章中，我们知道了 LLVM 具有庞大的构建系统：包含数百个构建文件和数千个依赖项，其中还需要为异构源文件定制构建指令的目标。这些复杂性需要 LLVM 使用先进的构建系统，并采用更结构化的方式进行设计。本章中，会学习一些重要的指令，以便在进行源码树内和树外进行 LLVM 开发时，编写出更简洁和表达性更强的构建脚本。

本章中，我们将讨论以下主题:

- 了解 LLVM 重要的 CMake 指令的词汇表
- 其他项目如何通过 CMake 集成 LLVM

2.1. 相关准备

类似于第 1 章，你可能想从源码构建 LLVM。另外，由于本章将涉及很多 CMake 构建脚本，可能也希望为 CMakeLists.txt 准备一个语法高亮显示插件 (例如：VSCode 的 CMake Tools 插件)。目前，所有主要的 IDE 和编辑器都有现成的高亮显示。这一章会对熟悉 CMakeLists.txt 基本语法的读者更加友好一些。

本章中的所有代码示例都可以在本书的 GitHub 库中找到：https://github.com/PacktPublishing/LLVM-Techniques-Tips-and-Best-Practices/tree/main/Chapter02。

2.2. LLVM 的 CMake 指令表

因为对构建系统灵活性的要求，LLVM 的构建方式已经从 GNU autoconf 切换到 CMake。从那以后，LLVM 提出了许多定制的 CMake 函数、宏和规则来优化自己的使用。本节将为您概述其中最重要和最常用的几个。我们将学习如何以及何时使用它们。

2.2.1 使用 CMake 添加新的库

库是 LLVM 框架的基本构建块。要为一个新库编写 CMakeLists.txt 时，不要使用 `add_library` 指令:

```
# In an in-tree CMakeLists.txt file…
add_library(MyLLVMPass SHARED
  MyPass.cpp) # Do NOT do this to add a new LLVM library
```

使用 `add_library` 有几个缺点:

- 第 1 章在构建 LLVM 时，LLVM 更喜欢使用全局 CMake 参数 (即 `BUILD_SHARED_LIBS`) 来控制所有的组件库是静态构建还是动态构建。使用内置指令很难做到这一点的。
- 类似的，LLVM 更喜欢使用一个全局的 CMake 参数来控制一些编译标志，比如是否启用**运行时类型信息 (RTTI)** 和代码库中的 **C++ 异常处理**。
- 通过使用定制的 CMake 函数/宏，LLVM 可以创建自己的组件系统，这为开发人员提供了更高级别的抽象，可以以更容易的方式指定构建目标依赖项。

因此，可以使用 `add_llvm_component_library`:

```
# In a CMakeLists.txt
add_llvm_component_library(LLVMFancyOpt
  FancyOpt.cpp)
```

这里，LLVMFancyOpt 是库名，FancyOpt.cpp 是源文件。

常规的 CMake 脚本中，可以使用 target_link_libraries 来指定给定目标的库依赖关系，然后使用 add_dependencies 在不同的构建目标之间，分配依赖关系来创建显式的描绘构建顺序。当使用 LLVM 的自定义 CMake 函数来创建库目标时，有一个更简单的方法来完成这些任务。

通过使用 add_llvm_component_library 中的 LINK_COMPONENTS 参数 (或 add_llvm_library，是前一个的更底层实现)，可以为目标指定需要链接的组件:

```
add_llvm_component_library(LLVMFancyOpt
  FancyOpt.cpp
  LINK_COMPONENTS
  Analysis ScalarOpts)
```

或者，可以对 LLVM_LINK_COMPONENTS 变量做同样的事情，而且需要在函数调用之前定义:

```
set(LLVM_LINK_COMPONENTS
  Analysis ScalarOpts)
add_llvm_component_library(LLVMFancyOpt
  FancyOpt.cpp)
```

当需要使用的 LLVM 构建块时，组件库不过是具有特殊意义的普通库。如果选择构建它，其就会包含在巨大的 libLLVM 库中，组件名与真正的库名会略有不同。如果需要从组件名映射到库名，可以使用下面的 CMake 函数:

```
add_llvm_component_library(LLVMFancyOpt
  FancyOpt.cpp
  LINK_LIBS
  ${BOOST_LIBRARY})
```

如果想直接链接到一个普通的库 (非 LLVM 组件)，可以使用 LINK_LIBS 参数:

```
add_llvm_component_library(LLVMFancyOpt
  FancyOpt.cpp
  LINK_LIBS
  ${BOOST_LIBRARY})
```

要将常规构建目标依赖分配给库目标 (相当于 add_dependencies)，可以使用 DEPENDS 参数:

```
add_llvm_component_library(LLVMFancyOpt
  FancyOpt.cpp
  DEPENDS
  intrinsics_gen)
```

intrinsics_gen 是一个常见的目标，表示生成包含 LLVM intrinsics 的头文件。

为每个文件夹添加一个构建目标

许多 LLVM 定制的 CMake 函数都有涉及到源文件检测陷阱。假设有一个这样的目录结构:

```
/FancyOpt
   |___ FancyOpt.cpp
   |___ AggressiveFancyOpt.cpp
   |___ CMakeLists.txt
```

这里，有两个源文件：FancyOpt.cpp 和 AggressiveFancyOpt.cpp。FancyOpt.cpp 是这种优化的基线版本，而 AggressiveFancyOpt.cpp 是一个 (更激进的) 实现版本。通常，希望将它们独立成库，以便用户选择。所以，可以这样写一个 CMakeLists.txt 文件:

```
# In /FancyOpt/CMakeLists.txt
add_llvm_component_library(LLVMFancyOpt
  FancyOpt.cpp)
add_llvm_component_library(LLVMAggressiveFancyOpt
  AggressiveFancyOpt.cpp)
```

不幸的是，当处理第一个 `add_llvm_component_library` 语句时，会产生错误：`Found unknown source AggressiveFancyOpt.cpp …`

LLVM 的构建系统会执行更严格的规则，以确保同一文件夹中的所有 `C/C++` 源文件都能添加到相同的库、可执行文件或插件中。为了解决这个问题，需要将两个文件拆分到单独的文件夹中，如下所示:

```
/FancyOpt
   |___ FancyOpt.cpp
   |___ CMakeLists.txt
   |___ /AggressiveFancyOpt
      |___ AggressiveFancyOpt.cpp
      |___ CMakeLists.txt
```

/FancyOpt/CMakeLists.txt：

```
add_llvm_component_library(LLVMFancyOpt
  FancyOpt.cpp)
add_subdirectory(AggressiveFancyOpt)
```

/FancyOpt/AggressiveFancyOpt/CMakeLists.txt：

```
add_llvm_component_library(LLVMAggressiveFancyOpt
  AggressiveFancyOpt.cpp)
```

这些是使用 LLVM 的自定义 CMake 指令为 (组件) 库添加构建目标的重要方式。接下来的两节中，将展示如何使用不同的 (LLVM 的)CMake 指令来添加可执行文件和 Pass 插件构建目标。

2.2.2 使用 CMake 函数添加可执行文件和工具

与 `add_llvm_component_library` 类似，添加一个新的可执行目标，可以使用 `add_llvm_executable` 或 `add_llvm_tool`:

```
add_llvm_tool(myLittleTool
  MyLittleTool.cpp)
```

这两个函数语法相同，只有通过 `add_llvm_tool` 创建的目标将包含在安装过程中。还有一个全局的 CMake 变量 `LLVM_BUILD_TOOLS`，用来启用/禁用 LLVM 工具。

这两个函数也可以通过 `DEPENDS` 参数来设置依赖项，类似于前面的 `add_llvm_library`。这里，只能使用 `LLVM_LINK_COMPONENTS` 来指定链接组件。

2.2.3 使用 CMake 函数添加 Pass 插件

虽然本书的后面章节会讨论 Pass 插件的开发，但了解如何为 Pass 插件添加构建目标现最为合适 (早期的 LLVM 版本仍然使用 `add_llvm_library` 和一些特定的参数):

```
add_llvm_pass_plugin(MyPass
  HelloWorldPass.cpp)
```

`LINK_COMPONENTS`、`LINK_LIBS` 和 `DEPENDS` 参数也可以在这里使用，其用法和功能与 `add_llvm_component_library` 相同。

这是一些常见和重要的 (与 llvm 相关的)CMake 指令，使用这些指令不仅可以使 CMake 代码更简洁。如果想做一些源码树内的开发，还可以与 LLVM 的构建系统同步。下一节中，利用本章所了解到的知识，了解如何将 LLVM 集成到源码树外的 CMake 项目中。

LLVM 源码内/外的开发

本书中，源码树内开发是直接向 LLVM 项目贡献代码，例如：修复 LLVM Cug 或向现有的 LLVM 库添加新特性。另一方面，源码树外开发，要么为 LLVM 创建扩展 (例如，编写一个 LLVM pass)，要么在其他项目中使用 LLVM 库 (例如，使用 LLVM 的代码生成库来实现属于自己的编程语言)。

2.3. 如何使用 CMake 集成 LLVM

在源码树内项目中，因为大多数基础设施已经存在，可以实现编程语言的特性，有利于原型设计。此外，与创建源码树外项目，以及将其链接到 LLVM 库相比，将整个 LLVM 源码树拷贝到对应代码库中并不是个好主意。例如，只想使用 LLVM 的特性创建一个代码重构工具，并想 GitHub 上开源它，这样的话，其他 GitHub 上的开发人员需要下载数 GB 的 LLVM 源代码，从而才能使用你的小工具，这会让开发者的体验直线下降。

据我所知，至少有两种方式可以用来配置其他项目链接到 LLVM 源码:

- 使用 `llvm-config` 工具
- 使用 LLVM 的 CMake 模块

这两种方法可以获取相应的信息，包括头文件和库路径。然而，后者会创建了更简洁和可读的 CMake 脚本，这对于使用 CMake 的项目来说比较友好。本节将展示使用 LLVM 的 CMake 模块，并将其集成到其他 CMake 项目的基本步骤。

首先，需要准备一个源码树外 (C/C++) 的 CMake 项目。在前一节中讨论的核心 CMake 函数/宏将帮助我们解决这个问题，现在来看下如何使用它们:

1. 假设项目中已 CMakeLists.txt，并需要链接到 LLVM 库框架：

```
project(MagicCLITool)
set(SOURCE_FILES
  main.cpp)
add_executable(magic-cli
  ${SOURCE_FILES})
```

不管是在创建生成可执行文件 (就像我们在前面的代码块中看到的那样)，还是其他工件 (如库或甚至 LLVM Pass 插件)，现在最大的问题是如何获取包含路径，以及库路径。

2. 为了解析包含路径和库路径，LLVM 提供了 CMake 包接口，这样直接使用 `find_package` 指令导入各种配置即可:

```
project(MagicCLITool)
find_package(LLVM REQUIRED CONFIG)
include_directories(${LLVM_INCLUDE_DIRS})
link_directories(${LLVM_LIBRARY_DIRS})
...
```

为了让 `find_package` 起作用，需要在项目使用 CMake 命令时，设置 `LLVM_DIR`:

```
$ cmake -DLLVM_DIR=<LLVM install path>/lib/cmake/llvm ···
```

确保它指向 LLVM 安装路径下的 lib/cmake/llvm 子目录。

3. 解析了包含路径和库之后，就可以将主可执行文件链接到 LLVM 的库了。LLVM 的自定义 CMake 函数 (例如，`add_llvm_executable`) 在这里将非常有用，但需要 CMake 能够找到这些函数实现。

下面的代码导入了 LLVM 的 CMake 模块 (更具体地说，是 AddLLVM 的 CMake 模块)，它包含了前一节中介绍的那些与 LLVM 相关的函数/宏:

```
find_package(LLVM REQUIRED CONFIG)
...
list(APPEND CMAKE_MODULE_PATH ${LLVM_CMAKE_DIR})
include(AddLLVM)
```

4. 下面的代码使用了前一节中介绍的 CMake 函数，并添加了可执行的构建目标:

```
find_package(LLVM REQUIRED CONFIG)
...
include(AddLLVM)
set(LLVM_LINK_COMPONENTS
  Support
  Analysis)
add_llvm_executable(magic-cli
  main.cpp)
```

5. 同理，添加库目标:

```
find_package(LLVM REQUIRED CONFIG)
...
include(AddLLVM)
add_llvm_library(MyMagicLibrary
  lib.cpp
  LINK_COMPONENTS
  Support Analysis)
```

6. 最后，添加 LLVM Pass 插件:

```
find_package(LLVM REQUIRED CONFIG)
...
include(AddLLVM)
add_llvm_pass_plugin(MyMagicPass
  ThePass.cpp)
```

7. 实践中，还需要注意**特定于 LLVM 的宏定义**和 RTTI 设置:

```
find_package(LLVM REQUIRED CONFIG)
...
add_definitions(${LLVM_DEFINITIONS})
if(NOT ${LLVM_ENABLE_RTTI})
  # For non-MSVC compilers
  set(CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -fno-rtti")
endif()
add_llvm_xxx(source.cpp)
```

   对于 RTTI 部分更要谨慎，因为在默认情况下，LLVM 不支持 RTTI，但普通的 C++ 应用支持 RTTI。如果代码和 LLVM 的库之间的 RTTI 配置不同，编译时会出现错误。

尽管在 LLVM 的源码树中进行开发很方便，但有时将整个 LLVM 源代码封装在项目中可能不可行。因此，必须创建树外项目，并将 LLVM 合成为一个库。本节展示了如何将 LLVM 集成到基于 CMake 的其他项目中，并充分利用特性于 LLVM 的 CMake 指令。

2.4. 总结

本章深入探讨了 LLVM 的 CMake 构建系统。了解了如何使用 LLVM 自己的 CMake 指令来编写简洁而有效的构建脚本，用于源码树内开发和树外开发。学习这些 CMake 技能可以让你的 LLVM 开发之旅更高效，并使得 LLVM 特性与其他现有代码库或自定义逻辑相结合有了更多的选择。

下一章中，将介绍 LLVM 项目中另一个重要的基础架构，称为 LLVM LIT，是一个通用框架，可运行各种测试。

第 3 章 LLVM LIT 测试

前一章中，了解了如何利用 LLVM 的 CMake 来改善开发体验，并了解了如何将 LLVM 集成到其他源码树外项目中。本章中，将讨论如何动手使用 LLVM 的测试架构 LIT。

LIT 是一个测试基础设施，最初是为运行 LLVM 的回归测试而开发的。现在，它不仅是在 LLVM 中运行所有测试 (包括单元测试和回归测试) 的工具，而且还是一个可以在 LLVM 之外使用的通用测试框架。它还提供了广泛的测试格式来处理不同的场景。本章将全面介绍这个框架中的组件，并帮助您掌握 LIT 的使用方式。

本章中，我们将讨论以下主题:

- 在源码树外项目中使用 LIT
- 学习高级的 FileCheck 技巧
- 探索 TestSuite 框架

3.1. 相关准备

LIT 的核心使用 Python 编写,所以请确保开发环境中安装有 Python 2.7 或 Python 3.x(Python 3.x 更推荐，LLVM 正在逐渐弃用 Python 2.7)。

此外，还有一些支持实用程序，如 FileCheck。不幸的是，要构建这些实用程序，最快的方法是构建 check-XXX(伪) 目标。例如，可以构建 check-llvm-support:

```
$ ninja check-llvm-support
```

最后一节要求构建 llvm-test-suite，这是一个独立于 llvm-project 的代码库。我们可以克隆它:

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

配置构建最简单的方法是使用 CMake 的缓存进行配置。例如，要用优化 (O3) 构建测试套件:

```
$ mkdir .O3_build
$ cd .O3_build
$ cmake -G Ninja -DCMAKE_C_COMPILER=<desired Clang binary \
  path> -C ../cmake/caches/O3.cmake ../
```

然后，进行正常构建:

```
$ ninja all
```

3.2. 在源码树外项目中使用 LIT

编写一个源码树内 LLVM IR 回归测试非常简单，用测试指令注释 IR 文件：

```
; RUN: opt < %s -instcombine -S -o - | FileCheck %s
target triple = "x86_64-unknown-linux"
define i32 @foo(i32 %c) {
  entry:
  ; CHECK: [[RET:%.+]] = add nsw i32 %c, 3
  ; CHECK: ret i32 [[RET]]
  %add1 = add nsw i32 %c, 1
  %add2 = add nsw i32 %add1, 2
  ret i32 %add2
}
```

这个脚本检查 InstCombine(由前面代码中显示的`-instcombine` 命令行选项触发) 是否将两个后续的算术相加简化为一个指令。将该文件放入 llvm/test 下的任意文件夹后，当执行 llvm-lit 工具时，脚本将会自动选中，并作为回归测试的一部分运行。

尽管很方便，但几乎不能帮助您在树外项目中使用 LIT。当其他项目需要一些端到端测试工具 (比如：格式转换器、文本处理器、检测程序，还有编译器) 时，树外 LIT 尤其好用。本节将向您展示如何将 LIT 引入到树外项目中，然后提供 LIT 运行流程的完整描述。

3.2.1 准备示例项目

本节中，将使用源码树外的 CMake 项目。这个示例项目构建了一个工具 `js-minifier`，它用于简化 JavaScript 代码。我们将转换以下 JavaScript 代码:

```
const foo = (a, b) => {
  let c = a + b;
  console.log(`This is ${c}`);
}
```

可以将其转换成一些语义等价的代码，并尽可能短:

```
const foo = (a,b) => {let c = a + b; console.log(`This is ${c}`);}
```

本节的目标不是了解如何编写这个 `js-minifier`，而是如何创建一个 LIT 测试环境，从而对这个工具进行测试。

示例项目的文件夹结构如下所示:

```
/JSMinifier
  |___ CMakeLists.txt
  |___ /src
  |      |___ js-minifier.cpp
  |      |___ /test
  |              |___ test.js
  |              |___ CMakeLists.txt
  |___ /build
```

src 文件夹下的文件包含 js-minifier 的源代码 (这里不打算介绍)。这里我们关注的是用于测试 js-minifier 的文件，位于/test 文件夹下 (目前只有一个文件 test.js)。

本节中，我们将建立一个测试环境，当我们在 CMake /build 文件夹下运行 llvmlit(测试驱动程序和本节的主要字符) 时，就会打印测试结果，像这样:

```
$ cd build
$ llvm-lit -sv .
-- Testing: 1 tests, 1 workers --
PASS: JSMinifier Test :: test.js (1 of 1)
Testing Time: 0.03s
  Expected Passes : 1
```

这显示了多少测试用例通过了测试，以及具体是哪些测试用例。

下面是测试脚本 test.js：

```
// RUN: %jsm %s -o - | FileCheck

// CHECK: const foo = (a,b) =>
// CHECK-SAME: {let c = a + b; console.log(`This is ${c}`);}
const foo = (a, b) => {
  let c = a + b;
  console.log(`This is ${c}`);
}
```

可以看到，这是一个简单的测试过程，运行 js-minifier 工具——由`%jsm` 指令将被 js-minifier 可执行文件的实际路径所替代——并使用 CHECK 和 CHECK-SAME 的指令，用 FileCheck 检查运行结果。

这样，我们的示例项目就完成了。在结束准备工作之前，我们还需要创建最后一个工具。

因为试图减少对 LLVM 源代码树的依赖，所以要使用 PyPi 存储库中可用的 LIT 包 (即 pip 命令行工具) 重新创建 `llvm-lit` 工具。你所需要做的就是安装这个包:

```
$ pip install --user lit
```

最后，用下面的脚本包装这个包:

```
#!/usr/bin/env python
from lit.main import main
if __name__ == '__main__':
  main()
```

现在，我们无需构建 LLVM 树，就可以使用 LIT 了。接下来，我们将创建一些 LIT 配置脚本，这些脚本将驱动整个测试流。

3.2.2 对 LIT 进行配置

本小节中，将展示如何编写 LIT 配置脚本。这些脚本描述了测试过程——测试文件在哪里、如何配置测试环境 (例如，需要导入某些工具)、出现故障时的应对策略等。学习这些技能可以极大地改善在 LLVM 之外的地方使用 LIT 的体验。这就开始吧！

1. 在/JSMinifier/test 文件夹中，创建一个名为 lit.cfg.py 的文件:

```
import lit.formats

config.name = 'JSMinifier Test'
config.test_format = lit.formats.ShTest(True)
config.suffixes = ['.js']
```

代码为 LIT 提供了一些信息。这里的配置变量是一个 Python 对象，当这个脚本加载到 LIT 的运行时，将使用相应的信息进行替代。使用预定义的字段，本质上就是注册表配置值，可以使用 lit.*.py 脚本添加相应的定制字段。

config.test_format 字段表示 LIT 将在 shell 环境中运行每个测试 (以 ShTest 格式)，而 config.suffixes 字段表示只有文件名后缀为.js 的文件才会视为测试用例 (就是所有的 JavaScript 文件)。

2. 写完上一步的代码后，LIT 现在需要另外两个信息：测试文件的根路径和工作目录:

```
...
config.suffixes = ['.js']
config.test_source_root = os.path.dirname(__file__)
config.test_exec_root = os.path.join(config.my_obj_root,
'test')
```

config.test_source_root 指向/JSMinifier/test。另一方面，config.test_exec_root 表示工作目录，my_obj_root 是是一个自定义配置字段的值表示其父文件夹位置，它指向构建文件夹的路径。换句话说，config.test_exec_root 最终值为/JSMinifier/build/test。

3. 之前在 test.js 中，使用`%jsm` 指令作为占位符，其最终会被 js-minifier 可执行文件的实际/绝对路径所取代。以下几行将设置替换值:

```
...
config.test_exec_root = os.path.join(config.my_obj_root, 'test')

config.substitutions.append(('%jsm',
os.path.join(config.my_obj_root, 'js-minifier')))
```

这段代码向配置添加了一个 config.substitutions 字段，它使 LIT 用/JSMinifier/ build/js-minifier 值替换测试文件中出现的每一个`%jsm`。这将把所有内容打包到 lit.cfg.py 中。

4. 现在，创建一个名为 lit.site.cfg.py.in 的新文件，并将其放在/JSMinifier/test 文件夹下。这个文件的第一部分看起来是这样:

```
import os
config.my_src_root = r'@CMAKE_SOURCE_DIR@'
config.my_obj_root = r'@CMAKE_BINARY_DIR@'
```

神秘的 config.my_obj_root 字段在这里解析，但不指向正常的字符串，它分配到的值为 @CMAKE_BINARY_DIR@,这个字段将被 CMake 替换为实际路径,其值与 config.my_src_root 字段一致。

5. 最后，这些信息都放在了 lit.site.cfg.py.in 里面:

```
...
lit_config.load_configure(
  config, os.path.join(config.my_src_root, 'test/
    lit.cfg.py'))
```

尽管这段代码非常简单，但还是有点难以理解。简单地说，这个文件最终将具象化到另一个文件中，所有使用两个 @ 包围的变量都将进行解析，并复制到构建文件夹中。在那里，将回调前面步骤中的 lit.cfg.py，这将在本节的后面进行解释。

6. 最后，是时候使用 CMake 的 configure_file 函数将那些使用两个 @ 包围的字符串替换为实际值。在/JSMinifier/test/CMakeLists.txt 中，在文件中添加以下内容:

```
configure_file(lit.site.cfg.py.in
  lit.site.cfg.py @ONLY)
```

configure_file 函数将使用当前 CMake 中的对应变量的值替换输入文件 (lit.site.cfg.py.in) 中使用两个 @ 包围的字符串。

例如，假设有一个名为 demo.txt.in 的文件，其中包含以下内容:

```
name = "@FOO@"
age = @AGE@
```

现在，在 CMakeLists.txt 中使用 configure_file:

```
set(FOO "John Smith")
set(AGE 87)
configure_file(demo.txt.in
         demo.txt @ONLY)
```

前面提到的替换将启动，并生成一个输出文件 demo.txt，文件内容如下:

```
name = "John Smith"
age = 87
```

7. 回到 lit.site.cfg.py.in 中，由于 CMAKE_SOURCE_DIR 和 CMAKE_BINARY_DIR 分别指向根源文件夹和构建文件夹。/JSMinifier/build/test/lit.site.cfg.py 将包含以下内容:

```
import os
config.my_src_root = r'/absolute/path/to/JSMinifier'
config.my_obj_root = r'/absolute/path/to/JSMinifier/build'

lit_config.load_config(
  config, os.path.join(config.my_src_root, 'test/
    lit.cfg.py'))
```

至此，我们了解了如何为示例项目编写 LIT 配置脚本。现在，是时候解释一些关于 LIT 内部如何工作的细节，以及为什么我们需要这么多文件 (lit.cfg.py，lit.site.cfg.py 和 lit.site.cfg.py)。

3.2.3 LIT 的内部构件

让我们看看下面的图表，它展示了在我们刚刚创建的演示项目中，运行 LIT 测试的工作流程:

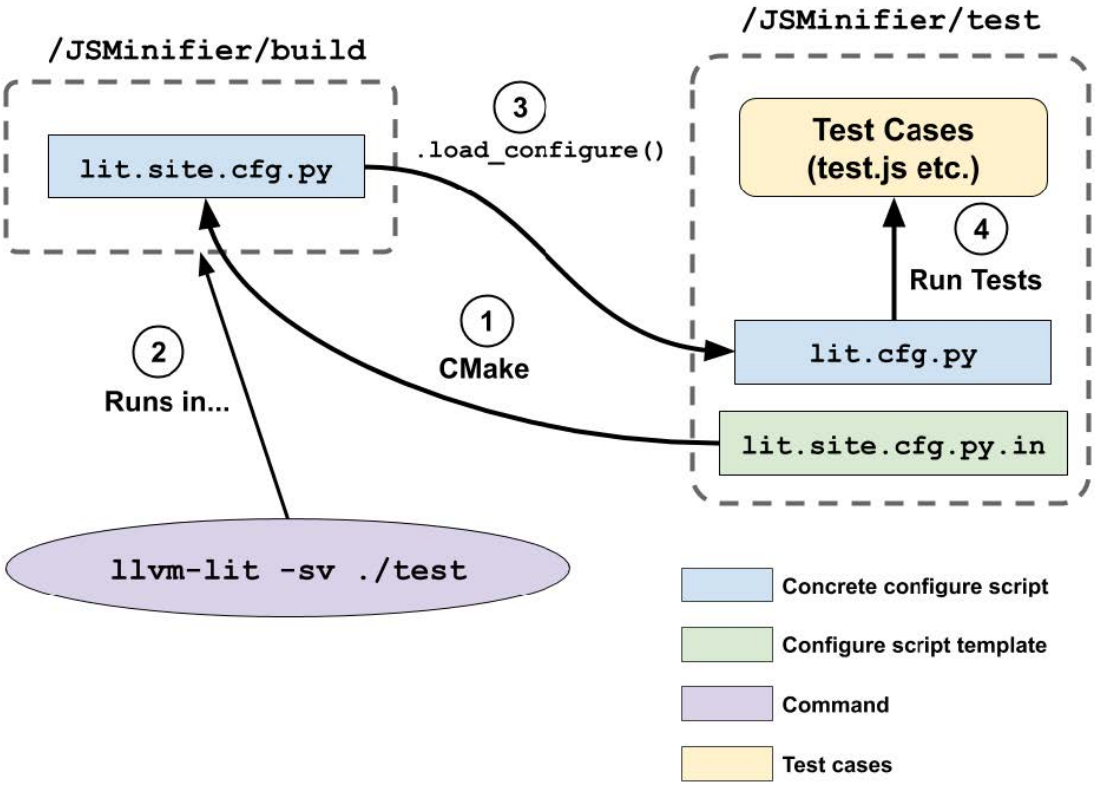

图 3.1 -示例项目中 LIT 的流程

让我们详细地看看这张图:

1. 将 lit.site.cfg.py.in 复制至/JSMinifier/build//JSMinifier/build/lit.site.cfg.py，其中包含一些 CMake 变量值。
2. `llvm-lit` 在/JSMinifier/build 中启动，将首先执行 lit.site.cfg.py。
3. 然后使用 `load_configure` Python 函数加载主要的 LIT 配置 (lit.cfg.py)，并运行所有的测试用例。

这个图中最关键的部分是解释 lit.site.cfg.py 和 lit.site.cfg.py.in 的角色: 它们有很多参数，比如构建目录的绝对路径，在 CMake 配置过程完成前都是未知的。因此，将一个弹性脚本 (即 lit.site.cfg.py) 放置在构建目录中，可以将该信息传递给实际的测试运行器。

在本节中，我们了解了如何为自己的树外示例项目编写 LIT 配置脚本。还了解了 LIT 在引擎盖下是如何工作的，这可以帮助您在各种各样的项目中使用 LIT(除了 LLVM)。

下一节中，我们将重点讨论 FileCheck，这是一个重要且常用的 LIT 程序，可以以更高级的模式执行检查。

3.3. FileCheck 技巧

FileCheck 是 LLVM 的高级模式检查器，与 Unix/Linux 系统中的 grep 类似，使用基于行的上下文，从而提供了更强大而简单的语法。此外，可以将 `FileCheck` 指令放在测试目标旁，可以让测试用例自包含，使测试更容易理解。

虽然基本的 `FileCheck` 语法很容易上手，但 `FileCheck` 还有许多其他功能，它们才能真正展示 `FileCheck` 的强大功能，并极大地改善了开发者的测试体验——例如创建更简洁的测试脚本和解析更复杂的程序输出，本节将向您展示其中的一些技巧。

3.3.1 准备示例项目

首先需要构建 `FileCheck` 命令行工具。与前一节类似，在 LLVM 树中构建一个 `check-XXX`(伪)目标是最简单的方法:

```
$ ninja check-llvm-support
```

在本节中，我们将使用一个假想的命令行工具 `js-obfuscator`，这是一个 JavaScript 混淆工具。**混淆**是一种常用的技术，用于隐藏知识产权或加强安全保护。例如，可以在以下 JavaScript 代码中使用 JavaScript 混淆器:

```
const onLoginPOST = (req, resp) => {
  if(req.name == 'admin')
    resp.send('OK');
  else
    resp.sendError(403);
}
myReset.post('/console', onLoginPOST);
```

将转换成以下代码:

```
const t = "nikfmnsdzaO";
const aaa = (a, b) => {
  if(a.z[0] == t[9] && a.z[1] == t[7] &&···)
    b.f0(t[10] + t[2].toUpperCase());
  else
    b.f1(0x193);
}
G.f4(YYY, aaa);
```

这个工具将尽量使原始脚本让人看不懂。测试部分面临的挑战是，在验证其正确性的同时仍然为随机性保留足够的空间。简单地说，`js-obfuscator` 只有 4 条混淆规则:

1. 只混淆局部变量名，包括形参。形式参数名以 $<$ *小写单词* $><$ *参数索引号* $>$ 格式进行混淆。局部变量名混淆成小写字母和大写字母的组合。
2. 如果用箭头语法来声明函数——例如，`foo = (arg1, arg2) => {···}`——箭头和左花括号 `(=> {)` 需要放在下一行。

3. 用不同表示形式的相同值替换文字数，例如：将 87 替换为 0x57 或 87.000。
4. 当使用`--shuffle-funcs` 选项时，会改变顶层函数的声明/出现顺序。

最后，下面的 JavaScript 代码是使用 `js-obfuscator` 的一个示例:

```
const square = x => x * x;
const cube = x => x * x * x;
const my_func1 = (input1, input2, input3) => {
  // TODO: Check if the arrow and curly brace are in the second
  // line
  // TODO: Check if local variable and parameter names are
  // obfuscated
  let intermediate = square(input3);
  let output = input1 + intermediate - input2;
  return output;
}
const my_func2 = (factor1, factor2) => {
  // TODO: Check if local variable and parameter names are
  // obfuscated
  let term2 = cube(factor1);
  // TODO: Check if literal numbers are obfuscated
  return my_func1(94, term2, factor2);
}
console.log(my_func2(1,2));
```

3.3.2 书写 FileCheck 指令

下面的步骤将填充前面代码中出现的 `TODO` 注释:

1. 根据行号，第一个任务是检查局部变量和参数是否正确地混淆。根据规范，形参有特殊的重命名规则 (即 *< 小写单词 >< 参数索引号 >*)，所以使用普通的 `CHECK` 指令和 FileCheck 自己的正则表达式是这里最合适的解决方案:

```
// CHECK: my_func1 = ({{[a-z]+0}}, {{[a-z]+1}},
// {{[a-z]+2}})
const my_func1 = (input1, input2, input3) => {
...
```

FileCheck 使用正则表达式的子集来进行模式匹配，使用 `{{…}}` 或 `[[…]]`。

2. 这段代码看起来非常简单。当进行了混淆，代码也需要正确的语义。因此，除了检查格式之外，对参数的后续引用也需要重构，这就是要使用 FileCheck 的模式绑定的原因：

```
// CHECK: my_func1 = ([[A0:[a-z]+0]],
// [[A1:[a-z]+1]], [[A2:[a-z]+2]])
const my_func1 = (input1, input2, input3) => {
  // CHECK: square([[A2]])
  let intermediate = square(input3);
...
```

该代码使用 [[…]] 语法将形参的模式绑定为名称为 A0 A2，其中绑定变量名和模式用冒号分开：[[< 绑定变量 >:< 模式 >]]。使用相同的 [[…]] 语法绑定变量的引用位置，不过没有了模式部分。

> **Note**
>
> 一个绑定变量可以有多个定义点，其参考点将读取最后的定义值。

3. 不要忘记第二条规则——函数头的箭头和左花括号需要放在第二行。要实现“后一行”的概念，可以使用 CHECK-NEXT 指令:

```
// CHECK: my_func1 = ([[A0:[a-z]+0]],
// [[A1:[a-z]+1]], [[A2:[a-z]+2]])
const my_func1 = (input1, input2, input3) => {
  // CHECK-NEXT: => {
```

与原来的 CHECK 指令相比，CHECK-NEXT 指令不仅检查模式是否存在，还确保模式在前一个指令的行后面。

4. 接下来，在 my_func1 中检查所有的局部变量和形参:

```
// CHECK: my_func1 = ([[A0:[a-z]+0]],
// [[A1:[a-z]+1]], [[A2:[a-z]+2]])
const my_func1 = (input1, input2, input3) => {
  // CHECK: let [[IM:[a-zA-Z]+]] = square([[A2]]);
  let intermediate = square(input3);
  // CHECK: let [[OUT:[a-zA-Z]+]] =
  // CHECK-SAME: [[A0]] + [[IM]] - [[A1]];
  let output = input1 + intermediate - input2;
  // CHECK: return [[OUT]];
  return output;
}
```

正如前面代码所示，CHECK-SAME 指令用于在同一行中匹配后续的模式。这背后的原理是 FileCheck 期望不同的 CHECK 指令在不同的行中进行匹配。那么，假设这段代码的一部分是这样的:

```
// CHECK: let [[OUT:[a-zA-Z]+]] =
// CHECK: [[A0]] + [[IM]] - [[A1]];
```

它将只匹配跨越两行或更多的代码，如下所示:

```
let BGHr =
  r0 + jkF + r1;
```

否则将抛出一个错误。如果想避免编写超长的检查语句行，可以使测试脚本更加简洁，并且可读性更强，这个指令就非常有用。

5. 进入 my_func2，现在检查文字数字是否正确混淆。这里的检查语句旨在接受除原始数字之外的任何实例/模式。因此，这里使用 CHECK-NOT 指令就足够了:

```
…
```

```
// CHECK: return my_func1(
// CHECK-NOT: 94
return my_func1(94,
        term2, factor2);
```

Note

因为 CHECK-NOT 在返回 my_func1(94，所以第一个 CHECK 指令是必需的。这里，CHECK-NOT 将给出假阴性结果，而没有 CHECK 指令将光标移动到正确的行。

此外，CHECK-NOT 在表示不 < 特定模式 >…但 < 正确模式 > 时，与 CHECK-SAME 一起使用会非常好用。

例如，如果混淆规则声明所有的文字数字都需要混淆成十六进制数，那可以用下面的代码来表达“不想看到 *94*…但是想看到 *0x5E* 或 *0x5e*”的断言:

```
...
// CHECK: return my_func1
// CHECK-NOT: 94,
// CHECK-SAME: {{0x5[eE]}}
return my_func1(94,
        term2, factor2);
```

6. 现在，只需要验证一个混淆规则：js-obfuscator 工具提供了命令行选项--shuffle-funcs，可以打乱所有顶级函数。这时，即使它们已经打乱了，也还需要检查顶级函数是否保持一定的顺序。在 JavaScript 中，函数在调用时解析。所以，cube、square、my_func1 和 my_func2 可以有任意的顺序，只要放在 console.log(…) 语句之前。使用 CHECK-DAG 指令表达这种灵活性非常好用。

   相邻的 CHECK-DAG 指令将以任意顺序匹配文本。例如，假设有以下指令:

```
// CHECK-DAG: 123
// CHECK-DAG: 456
```

这些指令将匹配以下内容:

```
123
456
```

还将匹配以下内容:

```
456
123
```

然而，这种排序自由在 CHECK 或 CHECK-NOT 指令中都不存在。例如，假设有这些指令:

```
// CHECK-DAG: 123
// CHECK-DAG: 456
// CHECK: 789
// CHECK-DAG: abc
// CHECK-DAG: def
```

这些指令将匹配以下文本:

```
456
123
789
def
abc
```

但是，不匹配以下文本:

```
456
789
123
def
abc
```

7. 回到我们的例子，可以使用下面的代码来检查混淆规则:

```
...
// CHECK-DAG: const square =
// CHECK-DAG: const cube =
// CHECK-DAG: const my_func1 =
// CHECK-DAG: const my_func2 =
// CHECK: console.log
console.log(my_func2(1,2));
```

但是，只有在向工具提供相应的命令行选项时，才会发生函数变换。不过，FileCheck 提供了一种将多个不同检查套件集成到单个文件中的方法，其中每个套件可以自定义运行方式，并将检查与其他套件分离。

8. FileCheck 中检查前缀的思想非常简单：可以创建一个独立运行的检查套件。前面提到的所有指令 (CHECK-NOT 和 CHECK-SAME) 中，每个套件将用相应字符串替换，包括 CHECK 本身，以区别于同一文件中的其他套件。例如，可以创建一个带有 YOLO 前缀的套件，这样这个例子 (部分) 现在看起来如下所示:

```
// YOLO: my_func2 = ([[A0:[a-z]+0]], [[A1:[a-z]+1]])
const my_func2 = (factor1, factor2) => {
...
// YOLO-NOT: return my_func1(94,
// YOLO-SAME: return my_func1({{0x5[eE]}},
return my_func1(94,
        term2, factor2);
...
```

要使用自定义前缀，需要在--check-prefix 选项中指定。这里，像这样使用 FileCheck 命令:

```
$ cat test.out.js | FileCheck --check-prefix=YOLO test.js
```

9. 最后，回到我们的例子。最后一个混淆规则可以通过 CHECK-DAG 指令使用另一个前缀来解决:

```
...
// CHECK-SHUFFLE-DAG: const square =
// CHECK-SHUFFLE-DAG: const cube =
// CHECK-SHUFFLE-DAG: const my_func1 =
// CHECK-SHUFFLE-DAG: const my_func2 =
// CHECK-SHUFFLE: console.log
console.log(my_func2(1,2));
```

这必须与默认的检查套件相结合。本节中提到的所有检查可以在两个独立的命令中运行，如下所示:

```
# Running the default check suite
$ js-obfuscator test.js | FileCheck test.js
# Running check suite for the function shuffling option
$ js-obfuscator --shuffle-funcs test.js | \
   FileCheck --check-prefix=CHECK-SHUFFLE test.js
```

本节中，我们通过示例项目展示了一些高级的 `FileCheck` 技能。这些技能提供了不同的方法来编写验证模式，并使 LIT 测试脚本更简洁。

目前为止，我们一直在讨论测试方法，以及在类 Shell 环境 (即以 `ShTest` LIT 格式) 中运行测试。下一节中，我们将介绍替代 LIT 的框架——`llvm-test-suite` 项目的 *TestSuite* 框架和测试格式——提供了一种与 LIT 不同的的测试方法。

3.4. 探索 TestSuite 框架

在前几节中，了解了如何在 LLVM 中执行回归测试，研究了 `ShTest` 的测试格式 (`config.test_format_format = lit.formats.ShTest(…)line`)，它以 shell 脚本的方式运行端到端测试。例如，在验证结果时，`ShTest` 格式提供了更多的灵活性，因为它可以使用 `FileCheck` 工具。

本节将介绍另一种测试格式:**TestSuite**。TestSuite 格式是 `llvm-test-suite` 的一部分，是为 LLVM 测试和基准测试创建的测试套件和基准测试的集合。与 `ShTest` 类似，这种 LIT 格式也用来运行端到端测试。当开发人员想要集成现有基于可执行的测试套件或基准代码库时，TestSuite 会使这种工作完成的更加轻松。例如，若想使用著名的 **SPEC 基准**作为测试套件，那么需要做的就是添加一个构建描述和期望的纯文本输出。当测试逻辑不能用文本测试脚本表达时，这也可以用 (正如在前面几节中了解的那样)。

本节中，将了解如何将一个现有的测试套件或基准代码库导入到 llvm-test-suite 项目中。

3.4.1 准备示例项目

首先，请按照本章开头的说明来构建 `llvm-test-suite`。

本节的其余部分将使用名为 *GeoDistance* 的伪测试套件项目，并使用 `C++` 和 Makefile 构建一个命令行工具 `geo-distance`，该工具计算由输入文件提供的每对经纬度的距离，并所有路径的距

离总和。

其有以下文件夹结构:

```
GeoDistance
  |___ helper.cpp
  |___ main.cpp
  |___ sample_input.txt
  |___Makefile
```

Makefile 长这样:

```
FLAGS := -DSMALL_INPUT -ffast-math
EXE := geo-distance
OBJS := helper.o main.o

%.o: %.cpp
  $(CXX) $(FLAGS) -c $^
$(EXE): $(OBJS)
  $(CXX) $(FLAGS) $< -o $@
```

运行 geo-distance 命令行工具:

```
$ geo-distance ./sample_input.txt
```

将总距离 (浮点) 输出到标准输出:

```
$ geo-distance ./sample_input.txt
94.873467
```

这里对浮点精度要求是小数点后三位 (0.001)。

3.4.2 将代码导入 llvm-test-suite

通常，只需要做两件事就可以将现有的测试套件或基准导入到 llvm-test-suite 中:

- 使用 CMake 作为构建系统
- 编写验证规则

使用 CMake 作为构建系统时，项目文件夹需要放在 llvm-test-suite 源代码树中的 MultiSource/Applications 子目录下，再更新相应地 CMakeLists.txt:

```
# Inside MultiSource/Applications/CMakeLists.txt
...
add_subdirectory(GeoDistance)
```

要从 Makefile 迁移到 CMakeLists.txt，但不需要使用 CMake 指令 `add_executable` 重写它，LLV 你提供了一些好用的函数和宏:

```
# Inside MultiSource/Applications/GeoDistance/CMakeLists.txt
# (Unfinished)
llvm_multisource(geo-distance)
llvm_test_data(geo-distance sample_input.txt)
```

这里有一些新的 CMake 指令。`llvm_multisource` 和 `llvm_singlesource`，分别从多个或单个源文件添加一个新的可执行构建目标，其内部实现是基于 `add_executable`，但正如前面的代码所示，可以选择将源文件列表置为空，这意味用当前目录中的所有 `C/C++` 源文件作为输入。

Note

如果有多个源文件，但使用的是 `llvm_singlesource`，那么每个源文件将视为一个独立的可执行文件。

`llvm_test_data` 将在运行时使用的资源/数据文件复制到适当的工作目录中。本例中，要复制的对象是 `sample_input.txt` 文件。

现在框架已经完成设置，可以使用以下代码配置编译标志了:

```
# Inside MultiSource/Applications/GeoDistance/CMakeLists.txt
# (Continue)
list(APPEND CPPFLAGS -DSMALL_INPUT)
list(APPEND CFLAGS -ffast-math)

llvm_multisource(geo-distance)
llvm_test_data(geo-distance sample_input.txt)
```

最后，TestSuite 需要知道如何运行测试，以及如何验证结果:

```
# Inside MultiSource/Applications/GeoDistance/CMakeLists.txt
# (Continue)
...
set(RUN_OPTIONS sample_input.txt)
set(FP_TOLERANCE 0.001)
llvm_multisource(geo-distance)
...
```

CMake 变量 `RUN_OPTIONS` 非常简单——为测试可执行文件提供了命令行选项。

对于验证部分，默认情况下，TestSuite 将使用一个增强版的 `diff` 以`.reference_output` 结尾的文件内容，与 `stdout` 的输出和退出码进行比较。

我们的例子中，预期内容和退出码在 GeoDistance/geo-distance.reference_output 中:

```
94.873
exit 0
```

聪明的读者可能会发现这里的预期答案与本节开始时的输出略有不同 (94.873467)，这是因为比较工具允许您指定所需的浮点精度，该精度由 CMake 变量 `FP_TOLERANCE` 控制。

本节中，我们学习了如何利用 `llvm-test-suite` 项目，及其 TestSuite 框架来测试来自现有代码库，或无法使用文本脚本表达测试逻辑的可执行文件。这将帮助您更有效地使用 LIT 测试不同类型的项目。

3.5. 总结

LIT 是一个通用的测试框架，它不仅可以在 LLVM 中使用，也可以在其他项目中使用。本章通过展示如何在不需要构建 LLVM 的情况下，将 LIT 集成到树外项目中来证明这一点。其次，我们了解了 FileCheck—LIT 测试脚本使用的强大模式检查器，这些技能可以增强测试脚本的表达能力。最后，展示了 TestSuite 框架，它适用于测试不同种类的程序，并可以作为 LIT 测试格式的补充。

下一章中，将讨论 LLVM 项目中的另一个框架：TableGen，其也是一个通用工具集，可以解决树外项目中的问题 (现在 LLVM 的后端开发基本上只使用这个工具集)。

3.6. 扩展阅读

目前，FileCheck 的源代码 (用 `C++` 编写) 仍在 LLVM 的源码树中。可以尝试用 Python 完成其功能 (https://github.com/mullproject/FileCheck.py)，将有效地使用 FileCheck 而无需构建 LLVM(就像 LIT 一样)!

第 4 章 TableGen 开发

TableGen 是一种**领域特定语言 (DSL)**，最初在**底层虚拟机 (LLVM)** 中开发，用来表达处理器的**指令集架构 (ISA)** 和其他硬件特性，类似于 **GNU 编译器集合 (GCC) 的机器描述 (MD)**。因此，许多人在学习 LLVM 后端开发时，便对对 TableGen 进行了学习。然而，TableGen 并不仅用于描述硬件规范：它是一种通用的 DSL，适用于涉及重要静态和结构化数据的任务。LLVM 在后台之外地方也使用了 TableGen。例如，Clang 使用 TableGen 进行命令行选项管理。社区也在探索在 TableGen 语法中实现 **InstCombine** 规则 (LLVM 的**窥视孔优化**) 的可能性。

尽管 TableGen 具有通用性，但核心语法从未被许多该领域的新开发人员所理解，因为不熟悉该语言本身，所以在 LLVM 的代码库中创建了大量复制粘贴的 TableGen 模板代码。本章试图说清楚这种情况，并展示如何将这种傲人的技术应用到实际应用中。

本章首先介绍了常见和重要的 TableGen 语法，这为您在 TableGen 中编写“美味的甜甜圈”做准备，在第二部分演示了 TableGen 的通用性。最后，本章将以开发自定义“发射器”或 **TableGen 后端**结束，本章会将 TableGen 中那些乏味的句子转换成厨房用语进行描述。

本章中，我们将讨论以下主题:

- 介绍 TableGen 的语法
- 用 TableGen 做甜甜圈
- 用 TableGen 后端打印食谱

4.1. 相关准备

本章主要关注 `utils` 文件夹中的一个工具:`llvm-tblgen`。使用以下命令对其进行构建:

```
$ ninja llvm-tblgen
```

Note

如果选择在 **Release 模式**下构建 `llvm-tblgen`，而不管全局构建类型如何，可以使用第 1 章中介绍的 CMake 变量 `LLVM_OPTIMIZED_TABLEGEN`。你可能想要更改该设置，因为在本章中使用 Debug 版本的 `llvm-tblgen` 会更好一些。

本章中所有的源代码都可以在 GitHub 库中找到: https://github.com/PacktPublishing/LLVM-Techniques-Tips-andBest-Practices-Clang-and-Middle-End-Libraries/tree/main/Chapter04.

4.2. 介绍 TableGen 语法

本节简要介绍了重要和常见的 TableGen 语法，提供了动手编写所需的所有基本知识，在下一节中会提供使用 TableGen 编写甜甜圈的食谱。

TableGen 是一种特定于领域的编程语言，可以对特制的数据布局进行建模。尽管是一种编程语言，但做的事情与传统语言完全不同。**传统编程语言**通常描述对 (输入) 数据执行的操作，如何与环境交互，以及它们如何生成结果，而不考虑采用的编程范式 (命令式、函数式、事件驱动……)。相比之下，TableGen 的行为几乎没有任何描述。

TableGen 的描述仅用于静态数据结构。首先，开发人员定义所需数据结构的布局 (本质上就是包含许多字段的表)。然后，填充/初始化字段时，需要立即将数据填充到布局中。后一部分可能是 TableGen 的独特之处：许多编程语言或框架提供了设计特定于领域的数据结构的方法 (例如，谷歌的 **Protocol Buffers**)，但在这些场景中，数据通常是**动态**填充的 (主要是在使用 DSL 的代码中)。

结构化查询语言 (SQL) 在许多方面与 TableGen 相同:SQL 和 TableGen(仅) 都处理结构化数据，并有定义布局的方法。在 SQL 中是 `TABLE`; 在 TableGen 中 `class`，这将在本节后面介绍。然而，除了精心设计布局之外，SQL 还提供了更多的功能, 还可以查询 (实际上，这就是它名字的由来: **结构化查询语言**) 和动态更新数据，这在 TableGen 中是不可以的。然而，在本章的后面，您将看到 TableGen 提供了一个很好的框架来灵活地处理和解释这个 (TableGen 定义的) 数据。

现在我们将介绍四种重要的 TableGen 结构:

- 布局和记录
- 叹号操作符
- 多记录
- **有向无环图 (DAG)** 数据类型

4.2.1 布局和记录

考虑到 TableGen 只是描述结构化数据的一种奇特、富有表现力的方式，数据布局和实例化数据都有一个原始的表示。布局通过类语法实现，如下面的代码所示:

```
class Person {
  string Name = "John Smith";
  int Age;
}
```

正如上所示，class 类似于 C 和许多其他编程语言中的结构体，只包含一组数据字段。每个字段都有一个类型，可以是任何基本类型 (`int`、`string`、`bit` 等) 或其他用户定义的类类型。字段还可以指定一个默认值，比如 `John Smith`。

了解了布局之后，创建一个实例 (对 TableGen 来说就是一个记录)，如下所示:

```
def john_smith : Person;
```

这里，`john_smith` 是一个使用 `Person` 作为模板的记录，因此也有两个字段- `Name` 和 `Age`——`Name` 字段用值 `john Smith` 填充。这看起来非常简单，但回想一下 TableGen 应该定义静态数据，并且大多数字段应该用值填充。现在，`Age` 字段还没有初始化。这里可以用括号闭包和语句覆盖它的值，如下所示:

```
def john_smith : Person {
  let Age = 87;
}
```

甚至可以为 john_smith 记录定义新的字段:

```
def john_smith : Person {
  let Age = 87;
  string Job = "Teacher";
}
```

注意，只能覆盖 (使用 let 关键字) 声明过的字段。

4.2.1 叹号操作符

叹号操作符是一组执行简单任务的函数，例如在 TableGen 中执行基本的算术或对值进行强制转换。下面是一个简单的把公斤换算成克的例子:

```
class Weight<int kilogram> {
  int Gram = !mul(kilogram, 1000);
}
```

常用的操作符包括算术操作符和位操作符 (仅举几个例子)，其中一些在这里概述:

- !add(a, b): 算术加法
- !sub(a, b): 算术减法
- !mul(a, b): 算术乘法
- !and(a, b): 逻辑和
- !or(a, b): 逻辑或
- !xor(a, b): 逻辑异或

使用条件运算符的情况，下面列出了一些:

- !ge(a, b): a 大于等于 b 时返回 1，否则返回 0
- !gt(a, b): a 大于 b 时返回 1，否则返回 0
- !le(a, b): a 小于等于 b 时返回 1，否则返回 0
- !lt(a, b): a 小于 b 时返回 1，否则返回 0
- !eq(a, b): a 等于 b 时返回 1，否则返回 0

其他有趣的操作符包括:

- !cast<type>(x): 该操作符根据类型参数对 x 操作数执行类型转换。在类型为数值类型的情况下，例如使用 int 或 bits，这将执行普通的算术类型转换。在一些特殊情况下，假设有以下场景:
  如果 type 是 string 而 x 是一个记录，则返回记录的名称。
  如果 x 是一个 string，将视为一个记录的名称。TableGen 将查找到目前为止的所有记录定义，并对名称为 x 的记录进行强制转换，并返回与类型参数匹配的类型。
- !if(pred, then, else): 如果 pred 为 1，该操作符返回 then 表达式，否则返回 else 表达式。
- !cond(cond1 : val1, cond2 : val2, …, condN : valN): 此操作符是!if 操作符的增强版本。在返回其关联的 val 表达式之前，将连续计算 cond1…condN，直到其中一个表达式返回 1。

Note

与运行时求值的函数不同，叹号操作符更像宏，在构建时求值——或者用 TableGen 的术语来说，当这些语法由 TableGen 后端处理时进行求值。

4.2.3 多记录

许多情况下，我们想要同时定义多个记录。例如，下面的代码片段尝试为多辆车创建汽车部件记录:

```
class AutoPart<int quantity> {…}

def car1_fuel_tank : AutoPart<1>;
def car1_engine : AutoPart<1>;
def car1_wheels : AutoPart<4>;
…
def car2_fuel_tank : AutoPart<1>;
def car2_engine : AutoPart<1>;
def car2_wheels : AutoPart<4>;
…
```

我们可以通过使用多记录语法进一步简化，如下所示:

```
class AutoPart<int quantity> {…}

multiclass Car<int quantity> {
  def _fuel_tank : AutoPart<quantity>;
  def _engine : AutoPart<quantity>;
  def _wheels : AutoPart<!mul(quantity, 4)>;
  …
}
```

创建记录实例时，使用 defm 语法而不是 def，如下所示:

```
defm car1 : Car<1>;
defm car2 : Car<1>;
```

multiclass 是对def的封装

因此，仍然会生成名称为 car1_fuel_tank、car1_engine、car2_fuel_tank 的记录。

尽管名称中有 class，但 multiclass 与 class 没有任何关系。multiclass 描述的不是记录的布局，而是作为模板来生成记录。在一个 multiclass 模板中是要创建的预期记录，以及在模板展开之后的记录名称后缀。例如，前面代码段中的 defm car1: Car<1> 指令最终扩展为三个 def 指令:

- def car1_fuel_tank : AutoPart<1>;
- def car1_engine : AutoPart<1>;
- def car1_wheels : AutoPart<!mul(1, 4)>;

正如前面的列表中看到的，会发现在 multiclass 中的名称后缀 (例如，_fuel_tank) 与本例中出现在发现 defm-car1 之后的名称连接在一起。此外，来自 multiclass 的 quantity 模板参数也会实例化到每个扩展的记录中。

简而言之，multiclass 会尝试从多个记录实例中提取公共参数，并使一次性创建这些参数成为可能。

4.2.4 有向无环图 (DAG) 数据类型

除了常规的数据类型之外，TableGen 还有一个非常独特的第一类类型:dag 类型，用于表示 DAG 实例。要创建一个 DAG 实例，你可以使用以下语法:

```
(operator operand1, operand2,···, operandN)
```

虽然操作符只能是一个记录实例，但操作数 (operand1···operandN) 可以是任意类型。下面是一个尝试建模算术表达式 x * 2 + y + 8 * z 的例子:

```
class Variable {···}
class Operator {···}
class Expression<dag expr> {···}

// define variables
def x : Variable;
def y : Variable;
def z : Variable;

// define operators
def mul : Operator;
def plus : Operator;

// define expression
def tmp1 : Expression<(mul x, 2)>;
def tmp2 : Expression<(mul 8, z)>;
def result : Expression<(plus tmp1, tmp2, y)>;
```

(可选) 可以将操作符和/或每个操作数与标记相关联，如下所示:

```
···
def tmp1 : Expression<(mul:$op x, 2)>;
def tmp2 : Expression<(mul:$op 8, z)>;
def result : Expression<(plus tmp1:$term1, tmp2:$term2,
y:$term3)>;
```

标签总是以美元符号 $ 开头，然后是用户定义的标签名。这些标记提供了每个 DAG 组件的逻辑描述，在 TableGen 后端处理 DAG 时非常有用。

本节中，介绍了 TableGen 语言的主要组件，并介绍了一些基本的语法。下一节中，将动手使用 TableGen 编写“美味的甜甜圈”。

4.3. 用 TableGen 做甜甜圈

有了前面部分的知识，是时候编写我们自己的甜甜圈了! 我们将按照以下步骤进行:

1. 要创建的第一个文件是 Kitchen.td。定义了烹饪环境，包括测量单位、设备和程序，仅举几个方面。我们将从测量单位开始:

```
class Unit {
  string Text;
  bit Imperial;
}
```

这里，Text 字段是菜谱上显示的文本格式，Imperial 只是一个布尔标志，用于标记这个单位是英制还是公制单位。每个重量或体积单元将是继承自这个类的一个记录——下面的代码是一个例子:

```
def gram_unit : Unit {
  let Imperial = false;
  let Text = "g";
}
def tbsp_unit : Unit {
  let Imperial = true;
  let Text = "tbsp";
}
```

我们需要创建许多度量单位，但是代码已经相当长了。简化和使它更可读的方法是使用类模板参数，如下所示:

```
class Unit<bit imperial, string text> {
  string Text = text;
  bit Imperial = imperial;
}
def gram_unit : Unit<false, "g">;
def tbsp_unit : Unit<true, "tbsp">;
```

与 C++ 的模板参数不同，TableGen 中的模板参数只接受具体的值，只是为字段赋值的另一种方法。

2. 由于 TableGen 不支持浮点数，需要定义一些方法来表示数字，例如 **1 和 $\frac{1}{4}$ 杯或 94.87g 的面粉**。一种解决办法是使用定点数，如下所示:

```
class FixedPoint<int integral, int decimal = 0> {
  int Integral = integral;
  int DecimalPoint = decimal;
}
def one_plus_one_quarter : FixedPoint<125, 2>; // Shown
as 1.25
```

提到 Integral 和 DecimalPoint 字段，FixedPoint 类表示的值等于下面的公式:

*Integral * 10(-DecimalPoint)*

$\frac{1}{4}$、$\frac{1}{2}$ 和 $\frac{3}{4}$ 显然是常用的测量单位 (特别是英制单位，如美制杯)，使用一个 helper 类来创建它们可能是个好主意:

```
class NplusQuarter<int n, bits<2> num_quarter> :
FixedPoint<?, 2> {···}
def one_plus_one_quarter : NplusQuarter<1,1>; // Shown as
1.25
```

这将使表示数量，如 N 和 $\frac{1}{4}$ 杯或 N 和 $\frac{1}{2}$ 杯变得更加容易。

TableGen 类也有继承——一个类可以继承一个或多个类。由于 TableGen 没有成员函数/方法的概念，继承类只是简单地集成字段。

3. 为了实现 `NplusQuarter`，特别是从 `NplusQuarter` 类模板参数到 `FixedPoint` 的转换，我们需要一些简单的算术计算，这就是使用 TableGen 的叹号操作符的位置:

```
class NplusQuarter<int n, bits<2> num_quarter> :
FixedPoint<?, 2> {
  int Part1 = !mul(n, 100);
  int Part2 = !mul(25, !cast<int>(num_quarter{1...0}));
  let Integral = !add(Part1, Part2);
}
```

另一个有趣的语法是 `num_quarter` 的位提取 (或切片)。通过 `num_quarter{1⋯0}`，这返回一个 `bits` 值，这个值的第 0 位和第 1 位与 `num_quarter` 第 0 位和第 1 位一样。这种技术还有其他一些变体。例如，可以对非连续范围的位进行切片:

```
num_quarter{8⋯6,4,2⋯0}
```

或者，它可以反向提取位，如下所示:

```
num_quarter{1⋯7}
```

Note

即使它已经声明 num_quarter 的宽度为 2 位 (bits<2> 类型)，读者们可能也想知道为什么代码需要显式地提取最小的 2 位。由于某些原因，TableGen 不会阻止将大于 3 的值赋给 `num_quarter`，例如:`def x: NplusQuarter<1,1999>`。

4. 有了计量单位和数字格式，就可以处理这个配方所需的配料了。首先，让我们使用一个文件 `Ingredients.td` 储存所有配料信息。为了使用前面提到的所有东西，可以使用包含语法导入 `Kitchen.td`:

```
// In Ingredients.td⋯
include "Kitchen.td"
```

然后，创建所有成分的基类 (有一些公共字段):

```
class IngredientBase<Unit unit> {
  Unit TheUnit = unit;
  FixedPoint Quantity = FixedPoint<0>;
}
```

每种食材都由 `IngredientBase` 派生的类表示，这些类都有参数来指定配方所需的量，以及计量食材的单位。以牛奶为例，如下所示:

```
class Milk<int integral, int num_quarter> :
IngredientBase<cup_unit> {
  let Quantity = NplusQuarter<integral, num_quarter>;
}
```

IngredientBased 的模板参数 cup_unit 表示，牛奶用美制杯为单位，数量将由 Milk 类模板参数确定。

编写一个配方时，每个成分都由一个记录表示，该记录由这些成分类创建:

```
def ingredient_milk : Milk<1,2>; // Need 1.5 cup of milk
```

5. 然而，有些食材总是混在一起——例如，柠檬皮和柠檬汁，蛋黄和蛋清。也就是说，如果你有两个蛋黄，就必须有两份蛋清。但是，如果需要创建一个记录，并为每个成分分配数，这样就会出现很多重复的代码。为了优雅的解决这个问题，这里使用 TableGen 的多类语法。

   以鸡蛋为例，假设想要同时创建相同数量的 WholeEgg、EggWhite 和 egggyolk 记录，首先需要定义 multiclass:

```
multiclass Egg<int num> {
  def _whole : WholeEgg {
    let Quantity = FixedPoint<num>;
  }
  def _yolk : EggYolk {
    let Quantity = FixedPoint<num>;
  }
  def _white : EggWhite {
    let Quantity = FixedPoint<num>;
  }
}
```

   编写配方时，使用 defm 语法创建 multiclass 记录:

```
defm egg_ingredient : Egg<3>;
```

   使用 defm 之后，将创建三个记录：egg_ingredient_whole、egg_ingredient_yolk 和 egg_ingredient_white，分别继承于 WholEgg、EggYolk 和 EggWhite。

6. 最后，我们需要一种方法来描述制作甜甜圈的步骤。许多食谱都有一些准备步骤，不需要按照特定的顺序来完成。以甜甜圈食谱为例：在油炸甜甜圈之前，随时都可以预热油。因此，用 dag 类型表示烘焙步骤可能是个好主意。

   先创建一个类来表示烘焙步骤:

```
class Step<dag action, Duration duration, string custom_
format> {
  dag Action = action;
  Duration TheDuration = duration;
  string CustomFormat = custom_format;
  string Note;
}
```

   Action 字段包含烘焙说明和有关所用配料的信息:

```
def mix : Action<"mix",···>;
def milk : Milk<···>;
def flour : Flour<···>;
def step_mixing : Step<(mix milk, flour), ···>;
```

`Action` 只是用来描述动作的类。下面的代码代表了 `step_mixing2` 使用 `step_mixing2` 的结果 (可能是生面团)，并将其与黄油混合的过程:

```
...
def step_mixing : Step<(mix milk, flour), ...>;
def step_mixing2 : Step<(mix step_mixing, butter), ...>;
```

最终，所有的 `Step` 记录将形成一个 DAG，其中的顶点是一个 `Step` 的记录，或是一个成分的记录。

这里，还用标签来注释 `dag` 操作符和操作数，如下所示:

```
def step_mixing2 : Step<(mix:$action step_mixing:$dough, butter)>
```

前一节 TableGen 语法介绍中，说过这些 `dag` 标签在 TableGen 代码中，除了影响 TableGen 后端处理当前记录的方式，并没有即时效果——例如，在 `Step` 类中有一个 `string` 类型字段 `CustomFormat`，如下所示:

```
def step_prep : Step<(heat:$action fry_oil:$oil, oil_
temp:$temp)> {
  let CustomFormat = "$action the $oil until $temp";
}
```

显示字段内容后，可以用记录的文本表示替换字符串中的 `$action`、`$oil` 和 `$temp`，生成一个字符串，例如: 花生油加热至 `300` 华氏度。

下一节的目标是开发一个自定义 TableGen 后端，将这里的 TableGen 版本的菜谱作为输入，并打印出纯文本的菜谱。

4.4. 用 TableGen 后端打印食谱

在上一节的最后一部分中，在用 TableGen 的语法编写完甜甜圈食谱之后，现在可以通过自定义的 TableGen 后端打印食谱了。

Note

请不要将 **TableGen 后端**与 **LLVM 后端**混淆: 前者将 TableGen 文件转换 (或转换) 为任意文本内容，C/C++ 头文件是最常见的形式。另一方面，LLVM 后端会将 LLVM **中间表示 (IR)** 转译为底层汇编代码。

本节中，我们将开发 TableGen 后端来打印前一节中甜甜圈食谱:

```
=======Ingredients=======
1. oil 500 ml
2. flour 300 g
3. milk 1.25 cup
4. whole egg 1
5. yeast 1.50 tsp
6. butter 3.50 tbsp
7. sugar 2.0 tbsp
8. salt 0.50 tsp
9. vanilla extract 1.0 tsp

=======Instructions=======
1. use deep fryer to heat oil until 160 C
2. use mixer to mix flour, milk, whole egg, yeast, butter,
sugar, salt, and vanilla extract. stir in low speed.
3. use mixer to mix outcome from (step 2). stir in medium
speed.
4. use bowl to ferment outcome from (step 3).
5. use rolling pin to flatten outcome from (step 4).
6. use cutter to cut outcome from (step 5).
7. use deep fryer to fry outcome from (step 1) and outcome from
(step 6).
```

首先，`llvm-tblgen` 是驱动 TableGen 翻译过程的程序。然后，再展示如何开发打印食谱的 TableGen 后端。最后，展示如何将后端集成到 `llvm-tblgen`。

4.4.1 TableGen 的高层工作流

TableGen 后端接受我们刚刚学习的 TableGen 代码的内存表示 (以 `C++` 对象形式)，并将其转换为任意文本内容。整个过程由 `llvm-tblgen` 可执行文件驱动，其工作流程如图所示:

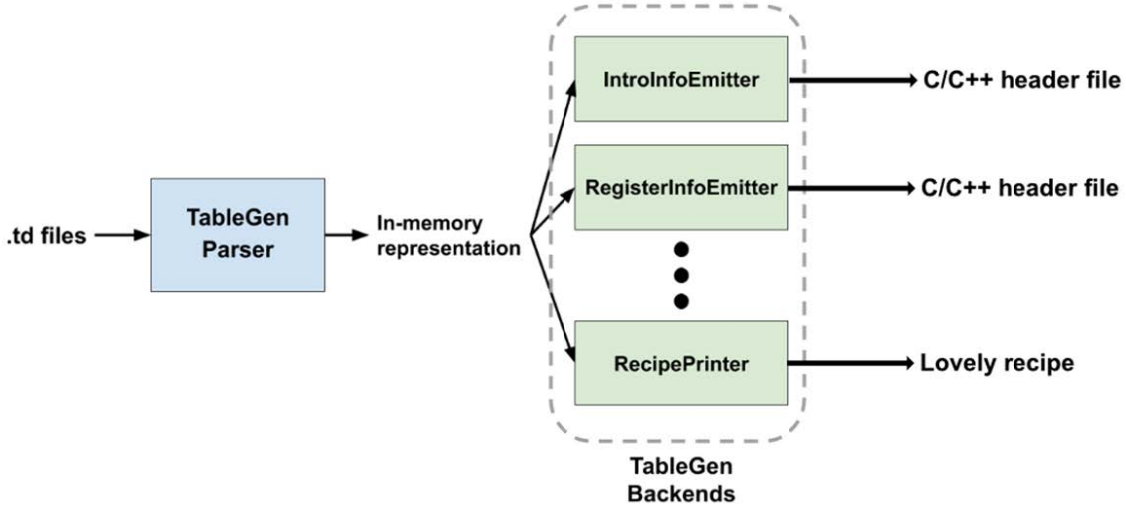

图 4.1 - llvm-tblgen 工作流程

TableGen 代码的内存表示 (由 C++ 类型和 API 组成) 在 TableGen 后端开发中扮演着重要的角色。与 LLVM IR 类似，是分层组织。从顶层开始，这里有层次结构的表，每个项目都是一个 C++ 类:

- `RecordKeeper`: 当前翻译单元中所有 Record 对象的集合 (和所有者)。
- `Record`: 表示一个记录或一个类。封闭字段由 `RecordVal` 表示。如果它是一个 `class`，也可以访问它的模板参数。
- `RecordVal`: 表示一对记录字段及其初始值，以及字段的类型和源位置等补充信息。
- `Init`: 表示字段的初始值。它是许多初始化值的父类，代表不同类型的初始化值——例如，`IntInit` 代表整数值，`DagInit` 代表 DAG 实例。

为了更了解 TableGen 后端实际完成的事情，一下是其框架结构:

```
class SampleEmitter {
  RecordKeeper &Records;
public:
  SampleEmitter(RecordKeeper &RK) : Records(RK) {}
  void run(raw_ostream &OS);
};
```

这个发射器接受一个 `RecordKeeper` 对象 (由构造函数传入) 作为输入，并将输出输出到 `raw_ostream` 流中——`SampleEmitter::run` 的函数参数。

下一节中，将展示如何设置开发环境，并动手编写 TableGen 后端。

4.4.2 编写 TableGen 后端

在本节中，将展示编写一个后端，用来打印使用 TableGen 编写菜谱的步骤。从设置开始。

项目设置

首先，LLVM 已经提供了一个编写 TableGen 后端的框架。请从 llvm 项目的源代码树中拷贝 llvm/lib/TableGen/TableGenBackendSkeleton.cpp 文件到 llvm/utils/TableGen 文件夹中，如下所示:

```
$ cd llvm
$ cp lib/TableGen/TableGenBackendSkeleton.cpp \
     utils/TableGen/RecipePrinter.cpp
```

然后，将 `cSkeletonEmitter` 类重构入 `RecipePrinter` 中。

`RecipePrinter` 有以下工作流程:

1. 收集所有烘焙步骤和配料记录。
2. 使用函数以文本格式打印计量单位、温度、设备等成分。
3. 将所有烘烤步骤的 DAG 线性化。
4. 打印每个线性化的烘烤步骤，并使用函数来自定义打印格式。

我们不打算涵盖所有的实现细节，因为许多后端代码实际上与 TableGen(例如：文本格式和字符串处理) 没有直接关系。因此，下面的小节只关注如何从 TableGen 的内存对象中检索信息。

完成所有的烘焙步骤

在 TableGen 后端,TableGen 记录由 C++ 类 `record` 表示。当我们想要检索从特定的 TableGen 类派生的所有记录时，可以使用 `RecordKeeper` 的函数 `getAllDerivedDefinitions`。例如，想要获取本例中从 `Step` 的 TableGen 类派生的所有烘焙步骤记录。下面是如何使用 `getAllDerivedDefinitions` 的例子:

```
// In RecipePrinter::run method···
std::vector<Record*> Steps = Records.
getAllDerivedDefinitions("Step");
```

这为我们提供了一个记录指针列表，表示所有的 `Step` 记录。

Note

本节的其余部分中，我们将使用这种格式的 `Record`(带有 Courier 字体) 来引用 `C++` 对应的 TableGen 记录。

检索字段值

从 `Record` 中检索字段值可能是最基本的操作。假设我们正在开发一个之前介绍的用于打印 `Unit` 记录对象的方法:

```
void RecipePrinter::printUnit(raw_ostream& OS, Record*
UnitRecord) {
  OS << UnitRecord->getValueAsString("Text");
}
```

`Record` 类提供了一些方便的函数，例如 `getValueAsString`，来检索字段的值，并尝试将其转换为特定类型。这样在获取实际值之前，就不需要检索特定字段的 `RecordVal` 值 (在本例中为 `Text` 字段)。类似的功能包括:

- `Record* getValueAsDef(StringRef FieldName)`
- `bool getValueAsBit(StringRef FieldName)`
- `int64_t getValueAsInt(StringRef FieldName)`
- `DagInit* getValueAsDag(StringRef FieldName)`

除了这些实用的函数外，有时只想检查记录中是否存在特定的字段。这种情况下，调用 `Record::getValue(StringRef FieldName)` 并检查返回值是否为空。但要注意，不是每个字段都需要初始化 (可能仍然需要检查字段是否存在，但未初始化)。当这种情况发生时，可以让 `Record::isValueUnset` 帮助你。

Note

TableGen 实际上使用了一个特殊的 `Init` 类——`UnsetInit`——来表示一个未初始化的值。

类型转换

`Init` 表示初始化值，但大多数时候我们不会直接使用，而是使用其子类。

例如，`StepOrIngredient` 是一个 `Init` 类型对象，可以表示 `Step` 记录或成分记录。由于 `DefInit` 提供了更丰富的功能，所以可以更容易地将其转换为底层的 `DefInit` 对象。可以使用以下代码将 `Init` 类型 `StepOrIngredient` 转换为 `DefInit` 对象:

```
const auto* SIDef = cast<const DefInit>(StepOrIngredient);
```

还可以使用 `isa<…>(…)` 首先检查底层类型，或者不想在转换失败时收到异常，可以使用 `dyn_cast<…>(…)`。

`Record` 表示 TableGen 记录，但如果能找到其父类就更好了，这将进一步获取相应字段的信息。

例如，在获取 `SIDef` 的基础 `Record` 对象后，可以使用 `isSubClassOf` 函数来判断 `Record` 是一个烘焙步骤，还是配料:

```
Record* SIRecord = SIDef->getDef();
if (SIRecord->isSubClassOf("Step")) {
  // This Record is a baking step!
} else if (SIRecord->isSubClassOf("IngredientBase")){
  // This Record is an ingredient!
}
```

了解底层的 TableGen 类，可以帮助我们以自己的方式打印记录。

处理 DAG 中的值

现在，开始打印 `Step` 记录。这里，使用 `dag` 类型来表示烘焙步骤所需的动作和配料:

```
def step_prep : Step<(heat:$action fry_oil:$oil, oil_
temp:$temp)> {
  let CustomFormat = "$action $oil until $temp";
}
```

这里，dag 存储在 TableGen 类 Step 的 Action 字段中。因此，可以使用 getValueAsDag 检索该字段是否为 DagInit 对象:

```
DagInit* DAG = StepRecord->getValueAsDag("Action");
```

DagInit 是从 Init 派生的另一个类，包含一些特定于 DAG 的 API。例如，可以遍历它的所有操作数，使用 getArg 函数获得相关联的 Init 对象，如下所示:

```
for(i = 0; i < DAG->arg_size; ++i) {
  Init* Arg = DAG->getArg(i);
}
```

此外，可以使用 getArgNameStr 函数来检索令牌 (如果有的话)，是在 TableGen 后端以字符串类型表示，与特定的操作数相关联:

```
for(i = 0; i < DAG->arg_size; ++i) {
  StringRef ArgTok = DAG->getArgNameStr(i);
}
```

如果 ArgTok 为空，意味着没有与该操作数相关联的令牌。要获得与操作符相关联的令牌，可以使用 getNameStrAPI。

Note

DagInit::getArgNameStr 和 DagInit::getNameStr 都返回不带美元符号的令牌字符串。

本节展示了使用 TableGen 指令的表示 C++ 的一些最重要技巧，这些在于编写 TableGen 后端的构建块。下一节中，将展示将所有内容放在一起，并运行自定义 TableGen 后端的最后一步。

4.4.3 集成 RecipePrinter TableGen 后端

完成 utils/TableGen/RecipePrinter.cpp 文件之后，是时候将所有内容放在一起了。

如前所述，TableGen 后端总是与 llvm-tblgen 工具相关联，这也是使用后端的唯一接口。llvm-tblgen 使用简单的命令行选项来选择要使用的后端。

下面的例子选择了一个后端：IntrInfoEmitter，从一个带有 X86 指令集信息的 TableGen 文件中生成一个 C/C++ 头文件:

```
$ llvm-tblgen -gen-instr-info /path/to/X86.td -o GenX86InstrInfo.inc
```

现在来了解一下如何将 RecipePrinter 源文件集成到 TableGen 后端:

- 为了将 RecipePrinter 源文件链接到 llvm-tblgen，并添加命令行选项来选择它，首先使用 utils/TableGen/TableGenBackends.h。这个文件只包含 TableGen 后端入口函数列表，这些函数以 raw_ostream 输出流和 RecordKeeper 对象作为参数。这里，也把我们的 EmitRecipe 函数放入列表中，如下所示:

```
...
void EmitX86FoldTables(RecordKeeper &RK, raw_ostream
&OS);
void EmitRecipe(RecordKeeper &RK, raw_ostream &OS);
void EmitRegisterBank(RecordKeeper &RK, raw_ostream &OS);
...
```

- 接下来，在 llvm/utils/TableGen/TableGen.cpp 中，首先添加一个新的 `ActionType` 枚举元素和所选的命令行选项:

```
enum Action Type {
  ...
  GenRecipe,
  ...
}
...
cl::opt<ActionType> Action(
  cl::desc("Action to perform:"),
  cl::values(
    ...
    clEnumValN(GenRecipe, "gen-recipe",
           "Print delicious recipes"),
    ...
  ));
```

- 然后进入 `LLVMTableGenMain` 函数，将函数调用插入到 `EmitRecipe` 中，如下所示:

```
bool LLVMTableGenMain(raw_ostream &OS, RecordKeeper
&Records) {
  switch (Action) {
    ...
    case GenRecipe:
    EmitRecipe(Records, OS);
    break;
  }
}
```

- 最后，不要忘记更新 utils/TableGen/CMakeLists.txt，如下所示:

```
add_tablegen(llvm-tblgen LLVM
  ...
  RecipePrinter.cpp
  ...)
```

- 所有要做的事情已经完成了! 现在可以执行以下命令:

```
$ llvm-tblgen -gen-recipe DonutRecipe.td
```

(可以使用-o 选项将输出重定向到文件。)

上面的命令会打印出甜甜圈的配方，就像这样:

```
=======Ingredients=======
1. oil 500 ml
2. flour 300 g
3. milk 1.25 cup
4. whole egg 1
5. yeast 1.50 tsp
6. butter 3.50 tbsp
7. sugar 2.0 tbsp
8. salt 0.50 tsp
9. vanilla extract 1.0 tsp

=======Instructions=======
1. use deep fryer to heat oil until 160 C
2. use mixer to mix flour, milk, whole egg, yeast,
butter, sugar, salt, and vanilla extract. stir in low
speed.
3. use mixer to mix outcome from (step 2). stir in medium
speed.
4. use bowl to ferment outcome from (step 3).
5. use rolling pin to flatten outcome from (step 4).
6. use cutter to cut outcome from (step 5).
7. use deep fryer to fry outcome from (step 1) and
outcome from (step 6).
```

本节中，学习了如何构建自定义 TableGen 后端，将用 TableGen 编写的菜谱转换成普通的文本格式。这里学到的内容包括，翻译 TableGen 代码的驱动程序 `llvm-tblgen` 是如何工作的，如何使用 TableGen 后端 `C++`API 来操作 TableGen 指令的内存表示，以及如何将自定义后端集成到 `llvm-tblgen` 并运行。结合本章和前一章中所学到的技能，现在可以使用 TableGen 作为解决方案，来创建完整且独立的工具链，从而实现相应的自定义逻辑。

4.5. 总结

本章中，介绍了 TableGen，是一种用于表示结构化数据的强大 DSL。本章已经展示了它在解决各种任务时的通用性 (最初是为编译器开发而创建的)。通过在 TableGen 中编写甜甜圈食谱，我们了解了它的核心语法。接着，了解了关于开发自定义 TableGen 后端，并了解了如何使用 `C++` API 与从源输入解析的内存中的 TableGen 指令直接交互，从而让读者能自行创建一个完整和独立的 TableGen 工具链来实现自己自定义逻辑。学习如何掌握 TableGen 不仅可以帮助开发与 llvm 相

关的项目，还可以为任意项目中解决结构化数据问题提供更多选择。

本节标志着第一部分的结束——介绍 LLVM 项目中常用的组件。从下一章开始，我们将进入 LLVM 的核心编译管道。我们将讨论的第一个主题就是 Clang——LLVM 的 C 族编程语言官方前端。

4.6. 扩展阅读

- LLVM 页面提供了 TableGen 语法的很好的参考: https://llvm.org/docs/TableGen/ProgRef.html
- 关于开发 TableGen 后端，LLVM 页面提供了一个很好的参考: https://llvm.org/docs/TableGen/BackGuide.html

第二部分：LLVM 的前端开发

在本节中，将介绍与前端相关的主题，包括 Clang 及其工具 (例如，语义推理)。我们将专注于插件开发，以及如何编写自定义工具链。

本节包括以下几章:

- 第 5 章，探索 Clang 架构
- 第 6 章，扩展预处理器
- 第 7 章，处理 AST
- 第 8 章，使用编译器标志和工具链

第 5 章 探索 Clang 架构

Clang 是 LLVM 的 C 族编程语言的官方前端，包括 C、C++ 和 Objective-C。处理输入源代码 (解析、类型检查和语义推理等)，并生成等效的 LLVM IR 代码，然后过其他 LLVM 子系统，执行优化和/或本地代码生成。许多类似 C 语言的方言或语言扩展也开始使用 Clang 托管它们的实现，例如：Clang 为 OpenCL、OpenMP 和 CUDA C/C++ 提供官方支持。除了常规的前端工作外，Clang 一直在发展，将其功能划分为库和模块，这样开发人员可以使用它们来创建各种与源代码处理相关的工具，例如：代码重构、代码格式化和语法高亮显示。学习 Clang 开发不仅可以让您更多地参与到 LLVM 项目中，还可以为创建功能强大的应用程序和工具提供了更多的可能性。

LLVM 将其大部分任务安排在管道 (即 PassManager) 中顺序执行。与 LLVM 不同，Clang 在组织子组件的方式上更加多样化。在本章中，将向您展示 Clang 的系统是如何组织的，角色分别是什么，以及其在源码的哪一部分。

术语

这一章到本书的其余部分，将使用 Clang(以大写 C 和 Minion Pro 字体开头) 作为整体来引用项目和使用到的技术。当使用 clang 时 (小写，Courier 字体)，则指的是可执行程序。

本章中，我们将讨论以下内容:

- 了解 Clange 的子系统及其作用
- 探索 Clang 工具的功能和扩展选项

本章结束时，您将对这个系统的路线图有了一定的了解，这样您就可以启动您感兴趣的项目，并为以后有关 Clang 开发的提供一些参考。

5.1. 相关准备

第 1 章中，了解了如何构建 LLVM。不过，没有构建 Clang。要将 Clang 包含在构建列表中，需要修改 CMake 变量 `LLVM_ENABLE_PROJECTS` 的值，如下所示:

```
$ cmake -G Ninja -DLLVM_ENABLE_PROJECTS="clang;clang-toolsextra" ···
```

该变量的值是以分号分隔的列表，其中每一项都是 LLVM 的子项目。本例包括了 Clang 和 `clang-tools-extra`，其中包含了一组基于 Clang 的有用工具，例如：`clang-format` 可以对编码风格进行统一，已经在无数的开源项目使用，特别是大型项目。

将 Clang 添加到现有的构建中

如果已经有了未启用 Clang 的 LLVM 构建，可以在 CMakeCache.txt 中编辑 CMake 参数 `LLVM_ENABLE_PROJECTS`，而不需要再次使用 CMake 命令。编辑了文件，并再次运行 Ninja(或你选择的构建系统)，CMake 就会重新配置。

可以使用以下命令来构建 `clang`, Clang 的驱动程序和主程序:

```
$ ninja clang
```

可以使用以下命令运行所有 Clang 测试:

```
$ ninja check-clang
```

现在，`clang` 可执行文件应该就在`/<your build directory>/bin` 文件夹中。

5.2. 了解 Clang 的子系统及其作用

本节中，我们将概述 Clang 的结构和组织架构。将简要介绍一些重要的组件或子系统，然后在本书的后面部分会有专门的章节来进行进一步的扩展。我们希望这将让您对 Clang 的内部结构有一些了解，以及它们如何在读者自己的项目进行使用。

首先，来鸟瞰一下结构。下图显示了 Clang 的高层结构:

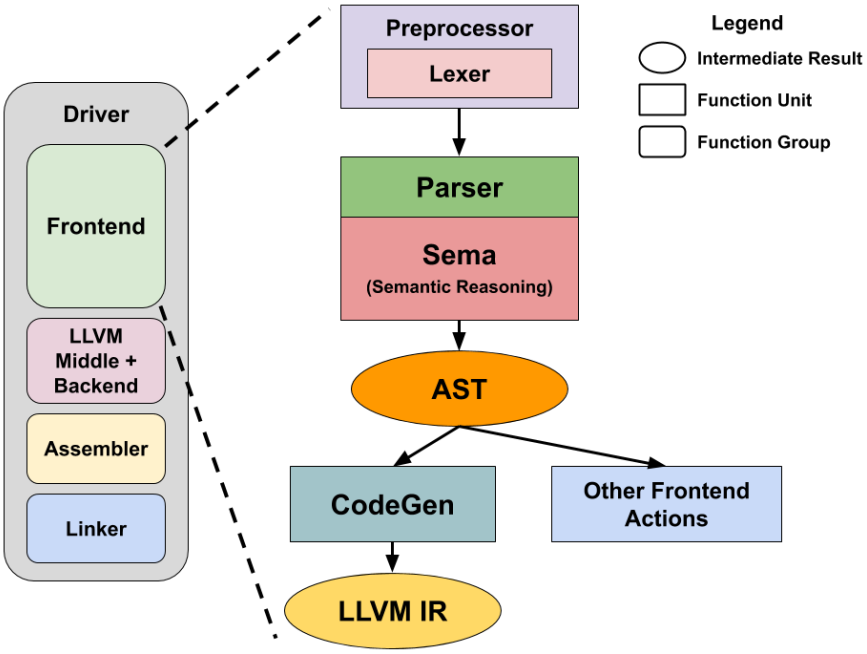

图 5.1 -Clang 的高层结构

如图例中所示，圆角矩形表示由多个具有类似功能的组件组成的子系统，例如：前端可以进一步分解为一些组件，如预处理器、解析器和代码生成逻辑等。此外，还有中间结果，在前面的图表中显示为椭圆。我们对其中两个特别感兴趣——**Clang AST** 和前者将在第 7 章中进行深入讨论，而后者则是本书第 3 部分“中端开发”的主要内容，会讨论可以应用于于 LLVM IR 的优化和分析。

让我们先看一下驱动程序的概述。下面的小节将简要介绍这些驱动程序组件。

5.2.1 驱动

常见的误解是，可执行文件 `clang` 是编译器的前端。虽然 `clang` 使用了 Clang 的前端组件，但可执行文件就是**编译器驱动**，或者简称为**驱动**。

编译源代码是一个复杂的过程。首先，它由多个阶段组成，包括以下几个阶段:

- **前端**: 解析和语义检查
- **中端**: 程序分析和优化
- **后端**: 原生代码生成
- **汇编**: 运行汇编器
- **链接**: 运行链接器

这些阶段及其包含的组件中，有无数的选项/参数和标志，例如: 告诉编译器在哪里搜索包含文件的选项 (即 GCC 和 Clang 中的`-I`)。此外，我们希望编译器能够计算出一些选项的值，例如：编译器可以在头文件搜索路径中默认包含一些 C/C++ 标准库的文件夹 (例如，Linux 系统中的`/include` 和`/usr/include`)，这样就不需要手动指定。继续这个例子，我们希望编译器能够跨不同的操作系统和平台进行移植，但是许多操作系统使用不同的 C/C++ 标准库路径。那么，编译器如何能够正确的选择路径呢?

这种情况下，需要驱动来协助编译器。驱动充当核心编译器组件的管家，为它们提供必要的信息 (例如，一个特定于操作系统的头文件包含路径)，以便用户只需要提供重要的命令行参数即可。观察驱动程序的方法是在正常使用 `clang` 时添加`-###` 命令行标志。例如，可以尝试用这个标志编译一个简单的 hello world 程序:

```
$ clang++ -### -std=c++11 -Wall ./hello_world.cpp -o hello_world
```

在 macOS 计算机上执行以上命令后，系统会显示如下信息:

```
"/path/to/clang" "-cc1" "-triple" "x86_64-apple-macosx11.0.0"
"-Wdeprecated-objc-isa-usage" "-Werror=deprecated-objcisa-usage"
"-Werror=implicit-function-declaration" "-emitobj" "-mrelax-all"
"-disable-free" "-disable-llvm-verifier" ⋯ "-fno-strict-return"
"-masm-verbose" "-munwind-tables"
"-target-sdk-version=11.0" ⋯ "-resource-dir"
"/Library/Developer/CommandLineTools/usr/lib/clang/12.0.0" "-isysroot"
"/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk" "-I/usr/local/include"
"-stdlib=libc++" ⋯ "-Wall" "-Wno-reorderinit-list"
```

```
 "-Wno-implicit-int-float-
conversion" "-Wno-c99-designator" ⋯ "-std=c++11" "-fdeprecated-macro"
"-fdebugcompilation-dir" "/Users/Rem" "-ferror-limit" "19" "-fmessage-length"
"87" "-stack-protector" "1" "-fstackcheck" "-mdarwin-stkchk-strong-link" ⋯
"-fexceptions" ⋯ "-fdiagnostics-show-option" "-fcolor-diagnostics" "-o"
"/path/to/temp/hello_world-dEadBeEf.o" "-x" "c++" "hello_world.cpp"⋯
```

这些信息，本质上是在驱动程序转译之后传递给真正 Clang 前端的编译标志。虽然不需要理解所有这些标志的意思，但即使对于一个简单的程序，编译流程中也包含大量的编译器选项和许多子组件的使用。

驱动程序的源代码可以在 `clang/lib/driver` 下找到。在第 8 章中，我们将更仔细进行了解。

5.2.2 前端

编译器教科书可能会说，编译器前端由词法**分析器**和**解析器**组成，它们生成**抽象语法树 (AST)**。Clang 的前端也使用了这个结构，但有些不同。首先，分析器通常与**预处理器**耦合，对源代码执行的语义分析将分离到名为 Sema 的子系统中，这将构建一个 AST，并执行所有类型的语义检查。

分析器和预编译器

由于编程语言标准的复杂性和实际源代码的规模，预处理变得非常重要，例如：当头文件层次结构有 10 层以上时，解析包含的文件就会变得棘手，这在大型项目中很常见。OpenMP 会使用 `#pragma` 来并行化循环，这使得像 `#pragma` 这样的高级指令可能会受到挑战。如何解决这些挑战，就需要预处理器和分析器之间的合作，分析器可以为所有预处理操作提供原语，其源代码可以在 `clang/lib/Lex` 下找到。第 6 章中，我们将了解如何对预处理器和分析器进行开发，并学习如何使用强大的扩展系统实现自定义逻辑。

解析器和 Sema

Clang 的解析器使用来自预处理器和词法分析器的标记流，并试图实现其语义结构。在 Sema 子系统在生成 AST 之前，从解析器的结果获取更多的语义检查和分析相应的指令 (变量名之类的标识符)。

Sema 就是解析器完成的一个操作。后来人们发现这个额外的抽象层并不是必须的，所以解析器现在只与 Sema 交互。尽管如此，Sema 仍然保留了这种回调风格的设计。例如，当解析 for 循环结构时，将调用 `clang::Sema::ActOnForStmt(⋯)` 函数 (定义在 `clang/lib/Sema/SemaStmt.cpp` 中)。然后，Sema 会做各种检查，以确保语法正确，并为 for 循环生成 AST 节点——这就是，`ForStmt` 对象。

AST

当使用自定义逻辑扩展 Clang 时，AST 是最重要的原语。我们将介绍的所有常用的 Clang 扩展/插件都是在 AST 上操作的。要体验 AST，可以使用下面的命令 (从源代码中) 打印出一个 AST:

```
$ clang -Xclang -ast-dump -fsyntax-only foo.c
```

例如，在我的电脑上，我使用了以下简单的代码，只包含一个函数:

```
1 int foo(int c) { return c + 1; }
```

将产生以下输出:

```
TranslationUnitDecl 0x560f3929f5a8 <<invalid sloc>> <invalid
sloc>
|…
`-FunctionDecl 0x560f392e1350 <./test.c:2:1, col:30> col:5 foo
'int (int)'
  |-ParmVarDecl 0x560f392e1280 <col:9, col:13> col:13 used c
'int'
  `-CompoundStmt 0x560f392e14c8 <col:16, col:30>
    `-ReturnStmt 0x560f392e14b8 <col:17, col:28>
      `-BinaryOperator 0x560f392e1498 <col:24, col:28> 'int' '+'
        |-ImplicitCastExpr 0x560f392e1480 <col:24> 'int' <LValueToRValue>
        |   `-DeclRefExpr 0x560f392e1440 <col:24> 'int' lvalue ParmVar 0x560f
            392e1280 'c' 'int'
        `-IntegerLiteral 0x560f392e1460 <col:28> 'int' 1
```

这个命令非常有用，因为 C++ AST 类代表特定的语言指令，这对于编写 AST 回调非常重要——而 AST 回调是许多 Clang 插件的核心。例如，从信息中可以知道变量的引用点 (`c + 1` 表达式中的 `c`) 由 `DeclRefExpr` 类表示。

与解析器的组织方式类似，可以注册不同类型的 ASTConsumer 实例来访问或操作 AST，稍后介绍的 **CodeGen** 就是其中之一。在第 7 章中，我们将展示如何使用插件实现自定义 AST 处理逻辑。

CodeGen

虽然没有规定应该如何处理 AST(例如，使用`-ast-dump` 命令行选项，前端将打印文本 AST 表示)，但 CodeGen 子系统执行的最常见的任务是发出 LLVM IR 代码，稍后将由 LLVM 编译成原生程序集或目标代码。

5.2.3 LLVM，组译器和连接器

当 CodeGen 子系统产生 LLVM IR 代码，其将有 LLVM 编译管道进行处理，并生成原生代码，或是汇编代码，再或是目标代码。LLVM 提供了一个称为 **MC 层**的框架，这个框架中体系结构可

以选择实现直接集成到 LLVM 流水线中的汇编程序。主流架构，如 x86 和 ARM 都使用这种方法。如果不这样做，在 LLVM 管道末端发出的任何文本汇编代码，都需要由驱动程序调用外部汇编程序进行处理。

尽管 LLVM 有自己的链接器 (即 LLD)，但集成的链接器仍不是一个成熟方案。因此，驱动程序总是调用外部链接器来链接目标文件，从而生成最终的二进制。

外部与集成

使用外部汇编程序或连接器意味着使用独立进程来运行程序。例如，要运行一个外部汇编程序，前端需要在启动汇编程序之前，将汇编代码放入一个临时文件，并将该文件路径作为命令行参数之一。另一方面，使用集成的汇编器/链接器会将汇编或链接的功能打包到库中，而不是打包到可执行文件中。因此，在编译管道的最后，LLVM 将调用 API 来处理程序集代码在内存中的实例，从而产生目标代码。当然，这种集成方法的优点是可以节省许多间接的操作 (写入临时文件，并立即读取它们)，也会让代码在某种程度上更加简洁。

至此，我们已经大致了解了从源代码到本机代码的普通编译流程。下一节中，我们将进阶对 `clang` 可执行文件的理解，并了解 Clang 提供的工具和扩展选项。这不仅增强了 `clang` 的功能，而且还提供了如何在 LLVM 之外的项目中，使用 Clang 的方法。

5.3. 探索 Clang 工具的功能和扩展选项

Clang 项目不仅包含 `clang` 可执行文件，还为开发人员提供接口来扩展其工具，并可以将其功能导出为库。本节中，我们将概述这些选项。其中的一些将在后面的章节中进行讨论。

Clang 中目前有三种工具和扩展选项:**Clang 插件**、**libTooling** 和 **Clang 工具**。在讨论 Clang 扩展时，为了解释它们之间的区别，并提供更多的背景知识，我们需要首先从一个重要的数据类型开始:`clang::FrontendAction` 类。

5.3.1 FrontendAction 类

了解 *Clang 的子系统及其扮演的角色*时，介绍了 Clang 的各种前端组件，例如：预处理器和 Sema 等。这些组件都封装在 `FrontendAction` 数据类型中。`FrontendAction` 实例可以视为在前端内部运行的单个任务，为任务提供了统一的接口与各种资源 (如输入源文件和 AST) 进行交互，从这个角度来看，其角色类似于 **LLVM Pass** (LLVM Pass 提供了统一的接口来处理 LLVM IR)。然而，这个数据类型与 LLVM Pass 也有一些显著的区别:

- 并非所有的前端组件都封装到 `FrontendAction` 中，比如：解析器和 Sema 都是独立的组件，为其他的 `ASTFrontendAction` 生成资料 (例如，AST)。
- 除了一些场景 (其中之一是 Clang 插件)，Clang 编译实例很少运行多个 FrontendAction。通常，只会执行一个 `FrontendAction`。

通常，`FrontendAction` 所要描述的是在前端的一个或两个重要位置要完成的任务。这解释了为什么其对工具或扩展开发非常重要——这就需要将我们的逻辑构建到一个 `FrontendAction`(更准确地说，是 `FrontendAction` 的派生类) 实例中来控制和定制一项 Clang 编译的行为。

为了让了解 `FrontendAction` 模块，以下是它的一些重要 API:

- `FrontendAction::BeginSourceFileAction(…)/EndSourceFileAction(…)`: 这些回调是派生类可以重写的，分别在处理源文件之前和处理完源文件之后需要执行的操作。
- `FrontendAction::ExecuteAction(…)`: 这个回调描述了 `FrontendAction` 的主要动作。注意，虽然不阻止你直接重写这个方法，但许多 `FrontendAction` 派生类已经提供了更简单的接口来描述一些常见的任务，例如：如果要处理一个 AST，应该继承 `ASTFrontendAction` 并利用其基础结构。
- `FrontendAction::CreateASTConsumer(…)`: 用于创建 `ASTConsumer` 实例的工厂函数，是一组回调函数，当其遍历 AST 的不同部分时，将在前端调用 (当前端遇到一组声明时，将调用回调函数)。注意，虽然大多数 `FrontendAction` 在生成 AST 之后工作，但 AST 可能不会生成，例如：如果用户只想运行预处理程序 (例如使用 Clang 的`-E` 命令行选项转储预处理后的内容)，可能会发生这种情况。因此，不一定要在自定义的 `FrontendAction` 中实现这个函数。

通常情况下，不会直接从 `FrontendAction` 派生，但是理解 `FrontendAction` 在 Clang 中的内部角色和接口，在使用工具或插件开发，给你带来更多的思考。

5.3.2 Clang 插件

Clang 插件可以动态注册新的 `FrontendAction`(更确切地说是 `ASTFrontendAction`)，可以在 `clang` 的主动作之前或之后处理 AST，甚至替换。在 **Chromium** 项目中可以找到实际的例子，他们使用 Clang 插件来强加一些特定于 Chromium 的规则，并确保他们的代码库没有任何非理想化的语法。例如，其中一个任务是检查 `virtual` 关键字是否放在了虚方法上。

通过使用简单的命令行选项，插件可以很容易地加载到 `clang` 中:

```
$ clang -fplugin=/path/to/MyPlugin.so … foo.cpp
```

如果想定制编译，但无法控制 `clang` 可执行文件 (也就是说，您不能使用修改过的 `clang` 版本)，那么这将非常有用。此外，使用 Clang 插件可以更紧密地与构建系统集成，例如：在修改了源文件或任意构建依赖项后，希望重新运行逻辑。因为 Clang 插件仍然使用 `clang` 作为驱动，而且现代的构建系统在解析普通的编译命令依赖关系方面做得很好，这可以通过调整一些编译标志来实现。

然而，使用 Clang 插件最大的缺点是它的 **API 问题**。理论上，可以在任何 `clang` 可执行文件中加载和运行插件，但前提是插件使用了 C++ API(和 ABI)，并且 `clang` 可执行文件与之匹配。不幸的是，到目前为止，Clang(以及整个 LLVM 项目) 并没有打算提供稳定的 C++ API。换句话说，为了选择最安全的方式，需要确保插件和 `clang` 使用的是完全相同的 LLVM(主) 版本。这个问题使得 Clang 插件很难独立发布。

我们将在第 7 章更详细地讨论这个问题。

5.3.3 libTooling 和 Clang 工具

LibTooling 是一个库，提供了在 Clang 技术之上构建独立工具的特性。可以像在项目中使用普通库一样使用它，而不需要依赖于 `clang` 可执行文件。此外，API 设计的更高层，这样就不需要处理很多 Clang 的内部细节，这使得它对非 Clang 开发人员更友好。

语言服务器是 libTooling 最著名的用例。语言服务器作为守护进程启动，并接受来自编辑器或 IDE 的请求。这些请求可以像检查代码片段的语法那样简单，也可以像代码完成那样复杂。虽然语言服务器不需要像普通编译器那样将传入的源代码编译成原生代码，但需要一种方法来解析和分析代码，这对于从头构建来说并不容易。libTooling 通过 Clang 现成的技术，为语言服务器开发人员提供了更简单的接口，从而避免了在这种情况下重新创建轮子的必要。

为了更具体地了解 libTooling 与 Clang 插件的区别，这里有一个 (简化的) 代码片段，用于执行自定义 `ASTFrontendAction` 类 `MyCustomAction`:

```
int main(int argc, char** argv) {
  CommonOptionsParser OptionsParser(argc, argv,…);
  ClangTool Tool(OptionsParser.getCompilations(), {"foo.cpp"});
  return Tool.run(newFrontendActionFactory<MyCustomAction>().
  get());
}
```

如前面代码所示，不能将此代码嵌入任何代码库。libTooling 还提供了许多不错的工具，如：`CommonOptionsParser` 用于解析文本命令行选项，并将其转换为 Clang 选项。

libTooling 的 API 稳定性

不幸的是，libTooling 也不提供稳定的 C++ API。然而，这并不是什么问题，因为这里可以完全控制所使用的 LLVM 版本。

最后但并非最不重要的是，**Clang 工具**是一组构建在 libTooling 上的实用程序。因为提供了一些常见的功能，所以可以看作为 libTooling 的命令行工具版本。例如，可以使用 `clang-refactor` 来重构代码。这包括重命名变量，如下代码所示:

```
// In foo.cpp…
struct Location {
  float Lat, Lng;
};
float foo(Location *loc) {
  auto Lat = loc->Lat + 1.0;
  return Lat;
}
```

如果想将 `Location` 结构体中的 `Lat` 成员变量重命名为 `Latitude`，可以使用以下命令:

```
$ clang-refactor --selection="foo.cpp:1:1-10:2" \
    --old-qualified-name="Location::Lat" \
    --new-qualified-name="Location::Latitude" \
    foo.cpp
```

构建 clang-refactor

要遵循本章开头的声明，在 LLVM_ENABLE_PROJECTS 的列表中包含 clang-tools-extra。这样，就能使用 ninja clang-refactor 命令来构建 clang-refactor 了。

会得到以下输出:

```
// In foo.cpp···
struct Location {
  float Latitude, Lng;
};
float foo(Location *loc) {
  auto Lat = loc->Latitude + 1.0;
  return Lat;
}
```

这是由 libTooling 内部构建的重构框架完成的，clang-refactor 仅提供了一个命令行接口。

5.4. 总结

本章中，了解了 Clang 的组织结构，以及一些重要子系统和组件的功能。然后，了解了 Clang 的主要扩展和工具选项之间的差异——Clang 插件、libTooling 和 Clang 工具——包括它们如何使用，以及优缺点。Clang 插件提供了一种简单的方法，通过动态加载的插件将自定义逻辑插入到 Clang 的编译管道中，但存在 API 稳定性问题；libTooling 与 Clang 插件有不同的关注点，其旨在为开发人员提供一个工具箱来创建一些独立的工具，并与 Clang 工具结合，为开发者提供各种实用的应用。

下一章中，将讨论预处理器的开发。我们将了解 Clang 中的预处理器和词法分析器是如何工作的，并展示如何编写插件来定制预处理逻辑。

5.5. 扩展阅读

- 以下是 Chromium 的 Clang 插件所做的检查列表: https://chromium.googlesource.com/chromium/src/tools/clang/+/refs/heads/master/plugins/FindBadConstructsAction.h。
- 可以在这里了解更多关于选择正确的 Clang 扩展接口: https://clang.llvm.org/docs/Tooling.html。
- LLVM 也有自己的基于 libtools 的语言服务器，叫做 clangd: http://clangd.llvm.org。

第 6 章 扩展预处理器

前一章中，我们讨论了 Clang 的结构——C 语言**底层虚拟机 (LLVM)** 的官方前端——以及它的一些最重要的组件。还介绍了各种 Clang 的工具和扩展选项。本章中，将深入到 Clang 前端管道的第一阶段: 预处理器。

预处理是 C 族编程语言的一个早期编译阶段，会替换了任何以哈希 (#) 字符开始的指令——`#include` 和 `#define`——只命名一个少数的文本内容 (或非文本标记)。例如，在解析头文件之前，预处理器基本上会将由 `#include` 指令指定的头文件的内容复制粘贴到当前编译单元中。这种技术的优点是可以提取通用代码进行重用。

本章中，将简要了解 Clang 的**预处理器/词法分析器**框架是如何工作的，以及一些重要的**应用程序编程接口 (API)**，这些接口可以帮助开发者进行开发。此外，Clang 还提供了一些方法，让开发者可以通过插件将自定义逻辑注入到预处理流程中，例如：允许以一种更简单的方式创建自定义的 `#pragma` 语法——OpenMP 使用的语法 (例如，`#pragma omp` 循环)。了解这些技术可以在解决不同抽象级别的问题时有更多的选择。以下是本章各小节的列表:

- 使用 `SourceLocation` 和 `SourceManager`
- 了解预处理器和词法分析器的基本知识
- 定制开发预处理器的插件和回调

6.1. 相关准备

本章希望您能够构建 Clang 可执行文件，可以通过以下命令获取:

```
$ ninja clang
```

下面的命令，可以在预处理后立即打印文本内容:

`clang` 的`-E` 选项对于在预处理之后立即打印文本内容非常有用。例如，`foo.c` 包含如下内容:

```
#define HELLO 4
int foo(int x) {
  return x + HELLO;
}
```

然后使用如下命令：

```
$ clang -E foo.c
```

上面的命令会给出这样的输出:

```
...
int foo(int x) {
  return x + 4;
}
```

如您所见，代码中的 HELLO 被 4 替换了。后面，可以在开发自定义扩展时使用此技巧进行调试。

本章的代码链接: https://github.com/PacktPublishing/LLVM-Techniques-Tips-and-Best-Practices-Clangand-Middle-End-Libraries/tree/main/Chapter06.

6.2. 使用 SourceLocation 和 SourceManager

当使用源码文件时，最基本的问题之一是编译器前端如何能够在文件中定位字符串片段。一方面，良好地打印格式消息 (例如，编译错误和警告消息) 是一项至关重要的工作，其中必须显示准确的行号和列号。另一方面，前端可能需要同时管理多个文件，并以一种有效的方式访问内存中的内容。在 Clang 中，这些问题主要由两个类处理:SourceLocation 和 SourceManager。我们将简要介绍它们，并展示如何在实践中进行使用。

6.2.1 SourceLocation

SourceLocation 类用于表示代码在文件中的位置。在实现时，使用行号和列号可能是最直观的方式。然而，在现实场景中，事情可能会变得复杂，例如：在内部不能天真地使用一对数字作为源代码位置的内存表示。主要原因是，Clang 的代码库中广泛使用了 SourceLocation 实例，并且基本上贯穿了整个前端编译管道。因此，重要的是使用一种简洁的方式来存储信息，而不是两个 32 位整数 (这可能还不够，因为还想知道具体的文件!)，这很容易使 Clang 的运行时占用过多的内存。

Clang 通过使用优雅设计，将 SourceLocation 作为一个大数据缓冲区的**指针** (或句柄) 来解决这个问题，该缓冲区存储了所有真正的源代码内容，例如：SourceLocation 只使用一个无符号整型，这也意味着其实例可以复制——这可以带来一些性能上的好处。由于 SourceLocation 是指针，所以只有当它与刚才提到的数据缓冲区一起使用才有意义，数据缓冲区是由本文中的第二个主要角色 SourceManager 管理的。

其他实用工具

SourceRange 是一对 SourceLocation 对象，表示源代码范围的开始和结束。FullSourceLocation 将普通的 SourceLocation 类和它关联的 SourceManager 类封装到一个类中，这样只需要一个 FullSourceLocation 实例即可，而不是两个对象 (一个 SourceLocation 对象和一个 SourceManager 对象)。

可复制的

除非有很好的理由，否则在编写 C++ 的正常情况下，应该避免通过对象的值传递对象 (例如，作为函数调用的参数)。因为涉及到对底层数据成员的复制，所以应该通过指针或引用来传递。但是，如果仔细设计，类型实例可以来回复制，而不需要做很多工作，例如：给没有成员变量或成员变量很少的类，加上一个默认的复制构造函数。如果实例可以复制，那么可以通过值来进行传递。

6.2.2 SourceManager

SourceManager 类管理存储在内存中的所有源文件，并提供访问它们的接口。还提供 API 来处理源代码位置，通过刚刚介绍的 SourceLocation 实例，例如：要从 SourceLocation 实例获取行号和列号，运行以下代码:

```
void foo(SourceManager &SM, SourceLocation SLoc) {
  auto Line = SM.getSpellingLineNumber(SLoc),
  Column = SM.getSpellingColumnNumber(SLoc);
  ...
}
```

前面代码片段中的 Line 和 Column 变量，分别是 SLoc 所指向源码位置的行号和列号。

想知道为什么在前面的代码片段中使用术语 spellingLineNumber，而不是使用 LineNumber 吗？在宏展开 (或任何在预处理过程中发生的展开) 的情况下，Clang 会在展开之前和之后跟踪宏内容的 SourceLocation 实例。拼写位置表示最初编写源代码的位置，而非宏扩展后的位置。

也可以使用以下 API 创建一个新的与拼写相关的扩展:

```
SourceLocation NewSLoc = SM.createExpansionLoc(
  SpellingLoc, // The original macro spelling location
  ExpansionStart, // Start of the location where macro is
  //expanded
  ExpansionEnd, // End of the location where macro is
  // expanded
  Len // Length of the content you want to expand
);
```

返回的 NewSLoc 可以通过 SourceManager 查询与拼写扩展关联的位置信息。

在后面的章节中，这些重要的概念和 API 将帮助您处理源代码位置——特别是在处理预处理程序时。下一节将介绍 Clang 中预处理器和词法分析器开发的一些背景知识，这些知识对于开发自定义预处理器插件和回调非常有用。

6.3. 了解预处理器和词法分析器的基本知识

前面的一节中，了解了如何在 Clang 中表示源位置，而源位置是预处理器的重要组成部分。本节中，将首先解释 Clang 的预处理器和词法分析器的原理，以及工作流程。然后，了解这个流程中的一些重要组件，并简要解释它们的用法。

6.3.1 Clang 中预处理器和词法分析器的作用

Clang 的预处理器和词法分析器执行的角色和主要操作，分别由 Preprocessor 和 Lexer 类表示，如下图所示:

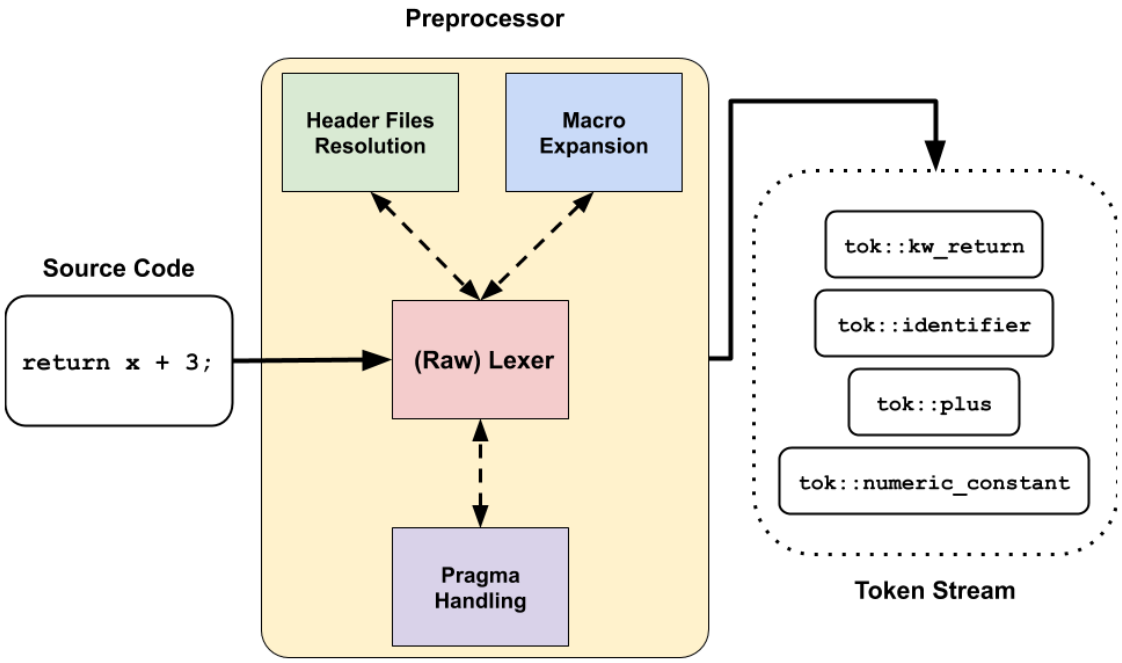

图 6.1 - Clang 预处理器和词法分析器的角色

相信大多数读者都熟悉词法分析器上下文中的**令牌**概念——源自原始源代码的子字符串，并充当语义推理的最小构建块。在一些传统的编译器中，分析器负责将输入的源代码切成一个令牌序列或**令牌流** (如上图所示)，这个令牌流稍后将被提供给解析器以构造语义结构。

实现方面，Clang 采取了与传统编译器 (或教科书上的编译器) 不同的路径:`Preprocessor` 使用的 `Lexer` 仍然是将源代码划分成令牌的主要执行者。然而，每当遇到预处理器指令 (即任何以 `#` 开头的指令) 或符号时，`Lexer` 都不会干涉，并将该任务转发给宏展开、头文件解析器或由预处理器组织的编译处理器。如果需要，这些辅助组件会将额外的令牌注入到主令牌流中，这些令牌流最终会返回给使用 `Preprocessor` 的用户。

换句话说，令牌流的大多数“消费者”并不直接与 `Lexer` 交互，而是与 `Preprocessor` 实例交互。因为 `Lexer` 本身只生成一个没有经过预处理的令牌流，这使得 `Lexer` 类被称为原始 `Lexer`(如前面的图所示)。为了更具体地了解如何使用因为 `Lexer` 本身只生成一个没有经过预处理的令牌流 `Preprocessor` 来检索令牌 (流)，我们提供了以下简单的代码片段，其会显示从当前处理的源代码中获取下一个令牌的方法:

```
Token GetNextToken(Preprocessor &PP) {
  Token Tok;
  PP.Lex(Tok);
  return Tok;
}
```

你可能已经猜到的，`Token` 是 Clang 中的单个令牌类，我们将在下一段中对其进行介绍。

6.3.2 Token 类

`Token` 类是单个令牌的表示，这些令牌可以来自源代码，也可以来自用于特殊目的的虚拟令牌。其也应用于预处理/词法分析框架，就像我们前面介绍的 `SourceLocation` 一样。因此，需要设计成内存简洁和可复制的类型。

对于 Token 类，在这里要强调两件事:

1. **令牌类型**描述了这个令牌是什么。
2. **标识符**表示语言关键字和任意前端标记 (例如函数名)。Clang 的预处理器使用了 IdentifierInfo 类来携带额外的标识符信息，将在本节的后面介绍这些信息。

令牌类型

令牌类型描述了这个 Token 是什么。Clang 的 Token 不仅用于表示具体的、物理语言结构，如: 关键字和符号，还用于表示由解析器插入的虚拟概念，以便使用单个 Token 编码具有尽可能多的信息。要查看令牌流的类型，可以使用以下命令行选项:

```
$ clang -fsyntax-only -Xclang -dump-tokens foo.cc
```

foo.cc 文件内容如下所示：

```
namespace foo {
  class MyClass {};
}
foo::MyClass Obj;
```

输出信息如下:

```
namespace 'namespace' [StartOfLine] Loc=<foo.cc:1:1>
identifier 'foo' [LeadingSpace] Loc=<foo.cc:1:11>
l_brace '{' [LeadingSpace] Loc=<foo.cc:1:15>
class 'class' [StartOfLine] [LeadingSpace] Loc=<foo.
cc:2:3>
identifier 'MyClass' [LeadingSpace] Loc=<foo.cc:2:9>
l_brace '{' [LeadingSpace] Loc=<foo.cc:2:17>
^^Ir_brace '}' Loc=<foo.cc:2:18>
semi ';' Loc=<foo.cc:2:19>
r_brace '}' [StartOfLine] Loc=<foo.cc:3:1>
identifier 'foo' [StartOfLine] Loc=<foo.cc:5:1>
coloncolon '::' Loc=<foo.cc:5:4>
identifier 'MyClass' Loc=<foo.cc:5:6>
identifier 'Obj' [LeadingSpace] Loc=<foo.cc:5:14>
semi ';' Loc=<foo.cc:5:17>
eof '' Loc=<foo.cc:5:18>
```

起始部分是每个令牌的令牌类型。完整的令牌类型列表可以在 clang/include/clang/Basic/

Tokenkings.def 中找到。这个文件是一个参考，可以了解语言结构 (例如，return 关键字) 与其对应的令牌类型 (kw_return) 的映射。

虽然无法可视化虚拟标记 (或者注释标记，因为它们在 Clang 的代码库中被调用)，但我们仍将使用与前面相同的示例来解释这些标记。在 C++ 中，::(前面指令中的冒号标记类型) 有几种不同的用法，例如：可以用于命名空间解析 (在 C++ 中更正式的说法是作用域解析)，如前面的代码片段所示，也可以 (可选地) 与 new 和 delete 操作符一起使用，如下面的代码所示:

```
int* foo(int N) {
  return ::new int[N]; // Equivalent to 'new int[N]'
}
```

为了提高解析处理的效率，解析器将首先尝试解析冒号标记是否为范围解析。如果是，该标记将替换为 annot_cxxscope 注释标记。

现在，让我们看看检索令牌类型的 API。Token 类提供了一个 getKind 函数来检索它的令牌类型，如下面的代码所示:

```
bool IsReturn(Token Tok) {
  return Tok.getKind() == tok::kw_return;
}
```

然而，如果只是做检查，就像在前面的代码中，有个更简洁的函数可用，如下所示:

```
bool IsReturn(Token Tok) {
  return Tok.is(tok::kw_return);
}
```

虽然很多时候，知道 Token 的标记类型就足以进行处理，但有些语言结构需要更多的方式来判断 (例如，表示函数名的标记，在这种情况下，标记类型、标识符并不像名称字符串那么重要)。Clang 使用一个专门化类 IdentifierInfo 来保存其他信息，比如：语言中任何标识符的符号名，这些会在下一段中讨论。

标识符

标准 C/C++ 使用单词**标识符**来表示各种各样的语言概念，从符号名 (如函数名或宏名) 到语言关键字，在标准中称为**保留标识符**。Clang 在实现端也遵循类似的路径: 用 IdentifierInfo 对象来装饰符合该语言标准标识符定义的 Token。此对象包含一些属性，例如：底层字符串内容或此标识符是否与宏函数关联。下面是如何从 Token 类型变量 Tok 中检索 IdentifierInfo 的实例:

```
IdentifierInfo *II = Tok.getIdentifierInfo();
```

如果 Tok 不代表语言标准定义的标识符，前面的 getIdentifierInfo 函数将返回 null。注意，如果两个标识符具有相同的文本内容，则它们由相同的 IdentifierInfo 对象表示。当想要比较不同的标识符标记是否具有相同的文本内容时，这就很方便了。

在各种令牌类型上使用专用的 IdentifierInfo 类型有以下优点:

- 对于 identifier 令牌类型的 Token，有时想知道它是否与宏相关联。这时，可以通过 IdentifierInfo::hasMacroDefinition 函数来查找。

- 对于 identifier 令牌类型的令牌,将底层字符串内容存储在辅助存储中 (即 IdentifierInfo 对象) 可以减少位于前端的热路径上的令牌对象的内存占用。可以使用 IdentifierInfo::getName 函数检索底层字符串内容。
- 对于表示语言关键字的 Token，尽管框架已经为这些类型的令牌提供了专用的令牌类型 (例如，kw_return 用于 return 关键字)，但其中一些令牌只在以后的语言标准中才成为语言关键字。例如，以下代码片段在 C++11 之前的标准中是合法的:

```
1 void foo(int auto) {}
```

- 使用下面的命令编译:

```
$ clang++ -std=c++03 -fsyntax-only ···
```

如果这样做，编译器不会给您任何抱怨，直到您将之前的-std=c++03 标准更改为-std=c++11 或更新的标准时。后一种情况下的错误消息将说，自 C++11 以来的语言关键字 auto 不能在那里使用。如果给定的标记在任何情况下都是关键字，那么前端则更容易判断，关键字标记上的 IdentifierInfo 对象是用于回答某个语言标准 (或语言特性) 下的关键字，比如：使用 IdentifierInfo::isKeyword(···) 函数，可以传递一个 LangOptions 类对象 (包含了诸如当前使用的语言标准和特性等信息的类) 作为这个函数的参数。

下一小节中，将介绍本节最后一个重要的预处理器概念: 预处理器如何处理 C 族语言中的宏。

6.3.3 处理宏

宏的实现不简单。除了前面介绍的源位置方面的挑战 (如何同时携带宏定义的源位置和扩展它们的位置) 外，重新定义和取消定义宏名的能力使整个过程变得复杂。以下面的代码作为一个例子:

```
1 #define FOO(X) (X + 1)
2 return FOO(3); // Equivalent to "return (3 + 1);"
3 #define FOO(X) (X - 100)
4 return FOO(3); // Now this is equivalent to "return (3 - 100);"
5 #undef FOO
6 return FOO(3); // "FOO(3)" here will not be expanded in preprocessor
```

前面的 C 代码显示了 FOO 的定义 (如果定义了 FOO) 在不同的词法位置 (不同的行) 上是不同的。

局部宏与模块宏

C++20 引入了一个新的语言概念，叫做**模块**。类似于许多其他面向对象语言 (如 Java 或 Python) 中的模块化机制。所以也可以在模块中定义宏，但其工作方式与传统的宏略有不同，传统的宏在 Clang 中称为局部宏。例如，可以通过使用关键字 (如 export) 来控制模块宏的可见性。在本书中，我们只讨论局部宏。

为了对这个概念进行建模，Clang 构建了一个系统来记录定义和未定义的链。在解释它是如何工作的之前，先来看看这个系统的三个最重要的组成部分:

1. MacroDirective: 该类是给定宏标识符的 #define 或 #undef 语句 j 进行逻辑表示。如前面的代码所示，在同一个宏标识符上可以有多个 #define(和 #undef) 语句，因此这些 MacroDirective 对象将根据它们出现的顺序形成一个链。更具体地说，#define 和 #undef 指令实际上分别由 MacroDirective、DefMacroDirective 和 UndefMacroDirective 的子类表示。
2. MacroDefinition: 该类表示当前时间点的宏标识符的定义。这个实例不包含完整的宏定义体，其更像一个指针，指向不同的宏定义体，这些宏定义体由 MacroInfo 类表示，稍后将在解析不同的 MacroDirective 类时引入这个类。这个类还可以告知定义了这个 MacroDefinition 类的 (最新的)DefMacroDirective 类。
3. MacroInfo: 该类包含宏体，包括宏体中的标记和宏定义的宏参数 (如果有的话)。

下面的图表说明了这些类与前面示例代码的关系:

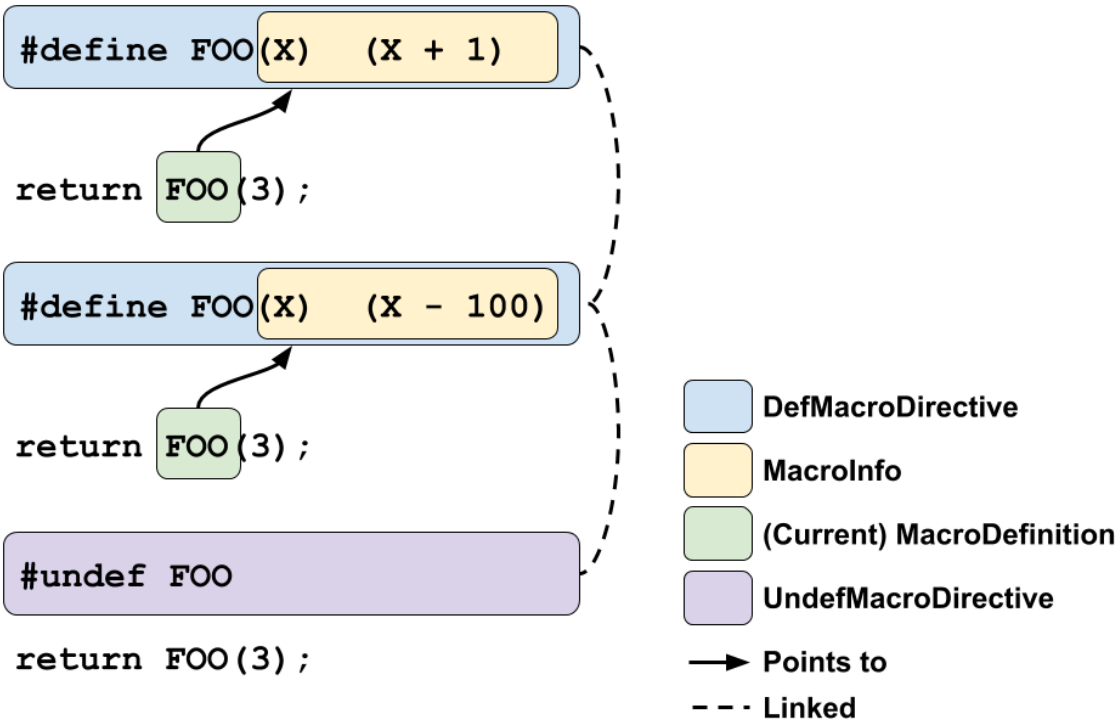

图 6.2 —C++ 中宏的类与前面的代码示例有何不同

要检索 MacroInfo 类和其 MacroDefinition 类，可以使用以下的预处理器 API，如下所示:

```
void printMacroBody(IdentifierInfo *MacroII, Preprocessor &PP)
{
  MacroDefinition Def = PP.getMacroDefinition(MacroII);
  MacroInfo *Info = Def.getMacroInfo();
  ...
}
```

前面的代码片段中显示的 IdentifierInfo 类型参数 MacroII 表示宏名。要进一步检查宏体，请运行以下代码:

```
void printMacroBody(IdentifierInfo *MacroII, Preprocessor &PP)
{
  ...
  MacroInfo *Info = Def.getMacroInfo();
  for(Token Tok : Info->tokens()) {
    std::cout << Tok.getName() << "\n";
  }
}
```

通过本节，了解了 Preprocessor 的工作流程，以及两个重要组件:Token 类和处理宏的子系统。了解这两种方法可以让您更好地了解 Clang 的预处理是如何工作的，并为下一节的 Preprocessor 插件和自定义回调开发做好准备。

6.4. 定制开发预处理器的插件和回调

Clang 的预处理框架也提供通过插件插入自定义逻辑的方法，与 LLVM 和 Clang 的其他部分一样灵活。更具体地说，允许开发人员编写插件来处理定制的 **pragma** 指令 (允许用户编写诸如 #pragma my_awesome_feature 之类的东西)。此外，Preprocessor 类还提供了一种更通用的方式来定义回调函数，以响应任意的**预处理事件**——例如，当宏展开或 #include 指令解析时。本节中，我们将使用一个简单的项目，利用这两种技术来展示它们的用法。

6.4.1 项目的目标和准备工作

C/C++ 中的宏以不卫生的设计而臭名昭著，如果使用不当，很容易导致编码错误。请看下面的代码片:

```
#define PRINT(val) \
printf("%d\n", val * 2)
void main() {
  PRINT(1 + 3);
}
```

前面代码中的 PRINT 看起来就像一个普通的函数，因此很容易相信这个程序将输出 8。然而这里，PRINT 是一个宏函数而不是普通函数，所以当它展开时，main 函数等价于:

```
void main() {
  printf("%d\n", 1 + 3 * 2);
}
```

因此，程序实际输出 7。这种歧义当然可以通过在宏体中每次出现 val 宏参数时都用括号括起来来解决，如下面的代码所示:

```
#define PRINT(val) \
printf("%d\n", (val) * 2)
```

现在，宏展开后，主函数将是这样的:

```
void main() {
```

```
  printf("%d\n", (1 + 3) * 2);
}
```

我们在这里要做的项目是开发一个自定义的 #pragma 语法来警告开发人员，如果某个由程序员指定的宏参数没有正确地括在括号中，以防止前面的卫生问题发生。下面是这个新语法的例子:

```
#pragma macro_arg_guard val
#define PRINT(val) \
printf("%d\n", val * 94 + (val) * 87);
void main() {
  PRINT(1 + 3);
}
```

与前面的例子类似，如果前面的 val 参数没有出现在括号中，这会引入 bug。

新的 macro_arg_guard pragma 语法中，pragma 后面的标记是宏参数名，以便在下一个宏函数中检入。由于前面代码片段中的 val * 94 表达式中的 val 没有括在括号中，编译时将输出以下警告消息:

```
$ clang ··· foo.c
[WARNING] In foo.c:3:18: macro argument 'val' is not enclosed by parenthesis
```

虽然只是一个简单的例子，但实际上在宏函数变得非常大或复杂时非常有用。这种情况下，手动为每个宏参数添加括号可能很容易出错。一个能够捕捉这种错误的工具绝对是有帮助的。

在我们深入编码部分之前，先建立项目文件夹。以下是文件夹结构:

```
MacroGuard
   |___ CMakeLists.txt
   |___ MacroGuardPragma.cpp
   |___ MacroGuardValidator.h
   |___ MacroGuardValidator.cpp
```

MacroGuardPragama.cpp 文件包括一个自定义的 PragmaHandler 函数，我们将在下一节实现这个函数时再讨论。对于 MacroGuardValidator.h/.cpp，包括自定义的 PPCallbacks 函数，用于检查指定的宏体和参数是否符合这里的规则。

因为这个项目是一个 LLVM 源码树之外的项目，如果不知道如何导入 LLVM 自己的 CMake 指令 (比如 add_llvm_library 和 add_llvm_executable)，请参考第 2 章中相关章节。在这里也要处理 Clang，所以需要使用类似的方式来导入 Clang 的构建配置，例如如下代码所示的包含文件夹路径:

```
# In MacroGuard/CmakeLists.txt
...
# (after importing LLVM's CMake directives)
find_package(Clang REQUIRED CONFIG)
include_directories(${CLANG_INCLUDE_DIRS}
```

这里不需要设置 Clang 库路径的原因是：通常情况下，插件会动态链接到加载程序提供的库的实现 (在我们的例子中，是 clang 可执行文件)，而不是在构建时显式链接那些库。

最后，添加插件的构建目标，如下所示:

```
set(_SOURCE_FILES
  MacroGuardPragma.cpp
  MacroGuardValidator.cpp
  )
add_llvm_library(MacroGuardPlugin MODULE
  ${_SOURCE_FILES}
  PLUGIN_TOOL clang)
```

PLUGIN_TOOL 的参数

在前面的代码片段中看到的 add_llvm_library 的 PLUGIN_TOOL 参数实际上是专为 Windows 平台设计的，因为动态链接库 (DLL) 文件——Windows 中的动态共享对象文件格式——有一个……有趣的规则，要求加载器可执行文件的名称显示在 DLL 文件头中。PLUGIN_TOOL 用于指定插件加载程序的名称。

建立了 CMake 脚本并构建了插件之后，可以使用以下命令运行插件:

```
$ clang ··· -fplugin=/path/to/MacroGuardPlugin.so foo.c
```

当然，目前还没有编写任何代码，所以什么也没有打印出来。下一节中，我们将开发一个自定义的 PragmaHandler 实例来实现我们新的 #pragma macro_arg_guard 语法。

6.4.2 实现自定义 pragma

实现上述特性的第一步是创建一个自定义的 #pragma 处理程序。为此，首先创建一个 MacroGuardHandler 类，其从 MacroGuardPragma.cpp 文件中的 PragmaHandler 类派生而来，如下所示:

```
1 struct MacroGuardHandler : public PragmaHandler {
2   MacroGuardHandler() : PragmaHandler("macro_arg_guard"){}
3   void HandlePragma(Preprocessor &PP, PragmaIntroducer
4                     Introducer, Token &PragmaTok) override;
5 };
```

每当预处理程序遇到非标准的编译指令时，将调用 HandlePragma 回调函数。我们将在这个函数中做两件事，如下所示:

1. 检索 pragma name 标记 (macro_arg_guard) 之后的任何补充标记 (作为 pragma 参数处理)。
2. 注册一个 PPCallbacks 实例，扫描下一个宏函数定义的主体，看看特定的宏参数是否正确地用括号括起来。下面将概述这项任务的细节。

对于第一个任务，我们利用 Preprocessor 来帮助我们解析 pragma 参数，这些参数是要封装的宏参数名。当调用 HandlePragma 时，Preprocessor 则停在 pragma name 标记后面的位置，如

下面的代码所示:

```
#pragma macro_arg_guard val
                        ^--Stop at here
```

因此，我们需要做的就是继续词法分析和存储这些令牌，直到到达这一行的末尾:

```
void MacroGuardHandler::HandlePragma(Preprocessor &PP,…) {
  Token Tok;
  PP.Lex(Tok);
  while (Tok.isNot(tok::eod)) {
    ArgsToEnclosed.push_back(Tok.getIdentifierInfo());
    PP.Lex(Tok);
  }
}
```

上述代码片段中的 eod 令牌类型表示**指令的结束**。它专门用于标记预处理指令的结束。

对于 ArgsToEscped 变量，下面的全局数组存储了指定宏参数的 IdentifierInfo 对象:

```
SmallVector<const IdentifierInfo*, 2> ArgsToEnclosed;
struct MacroGuardHandler: public PragmaHandler {
  …
};
```

我们将 ArgsToEnclosed 声明在全局作用域中的原因是,稍后要使用它与我们的 PPCallbacks 实例通信，后者将使用该数组的内容来执行验证。

尽管我们的 PPCallbacks 实例，即 MacroGuardValidator 类的实现细节要在下一节才讨论，但当 HandlePragma 函数第一次调用时，它需要在预处理器中注册，如下所示:

```
struct MacroGuardHandler : public PragmaHandler {
  bool IsValidatorRegistered;
  MacroGuardHandler() : PragmaHandler("macro_arg_guard"),
  IsValidatorRegistered(false) {}
  …
};
void MacroGuardHandler::HandlePragma(Preprocessor &PP,…) {
  …
  if (!IsValidatorRegistered) {
    auto Validator = std::make_unique<MacroGuardValidator>(…);
    PP.addCallbackPPCallbacks(std::move(Validator));
    IsValidatorRegistered = true;
  }
}
```

我们还使用一个标志来确保它只注册一次。在此之后，无论何时发生预处理事件，都将调用我们的 MacroGuardValidator 类来处理。我们的例子中，只对宏定义事件感兴趣，该事件向 MacroGuardValidator 发出信号，以验证刚刚定义的宏体。

在结束 PragmaHandler 之前，我们需要一些额外的代码将处理程序转换成一个插件，如下所示:

```
struct MacroGuardHandler : public PragmaHandler {
  ...
};
static PragmaHandlerRegistry::Add<MacroGuardHandler>
  X("macro_arg_guard", "Verify if designated macro args are
    enclosed");
```

声明了这个变量之后，当这个插件加载到 clang 时，一个 MacroGuardHandler 实例将插入到全局的 PragmaHandler 注册表中，当它遇到一个非标准的 #pragma 指令时，预处理器将查询注册表。当插件加载时，Clang 就能够识别我们自定义的 macro_arg_guard。

6.4.3 实现自定义预处理器的回调

Preprocessor 提供了一组回调函数，即 PPCallbacks 类，当某些预处理器事件 (比如：宏扩展) 发生时，这些回调函数将触发。上一节，展示了如何注册自己的 PPCallbacks 实现，即 MacroGuardValidator 与 Preprocessor。这里，展示了 MacroGuardValidator 如何在宏函数中验证宏参数-转义的规则。

首先，在 MacroGuardValidator.h/.cpp 中，放入以下框架:

```
// In MacroGuardValidator.h
extern SmallVector<const IdentifierInfo*, 2> ArgsToEnclosed;

class MacroGuardValidator : public PPCallbacks {
  SourceManager &SM;
public:
  explicit MacroGuardValidator(SourceManager &SM) : SM(SM) {}
  void MacroDefined(const Token &MacroNameToke,
  const MacroDirective *MD) override;
};

// In MacroGuardValidator.cpp
void MacroGuardValidator::MacroDefined(const Token
&MacroNameTok, const MacroDirective *MD) {
}
```

在 PPCallbacks 的所有回调函数中，我们只对 MacroDefined 感兴趣，它在宏定义处理时调用，由 MacroDirective 类型函数参数 (MD) 表示。需要显示一些警告消息时，SourceManager 类型成员变量 (SM) 用于打印 SourceLocation。

这里关注于 MacroGuardValidator::MacroDefined,其实逻辑非常简单: 对于 ArgsToEnclosed 数组中的每个标识符，我们都扫描宏体，以检查它的出现是否有括号作为它的前代和后代令牌。首先，让我们将其放入循环的框架，如下所示:

```
void MacroGuardValidator::MacroDefined(const Token
&MacroNameTok, const MacroDirective *MD) {
  const MacroInfo *MI = MD->getMacroInfo();
```

```
  // For each argument to be checked…
  for (const IdentifierInfo *ArgII : ArgsToEnclosed) {
    // Scanning the macro body
    for (auto TokIdx = 0U, TokSize = MI->getNumTokens();
    TokIdx < TokSize; ++TokIdx) {
      …
    }
  }
}
```

如果一个宏体令牌的 IdentifierInfo 参数与 ArgII 匹配，这意味着出现了一个宏参数，我们检查该令牌的前一个和下一个令牌，如下所示:

```
for (const IdentifierInfo *ArgII : ArgsToEnclosed) {
  for (auto TokIdx = 0U, TokSize = MI->getNumTokens();
  TokIdx < TokSize; ++TokIdx) {
    Token CurTok = *(MI->tokens_begin() + TokIdx);
    if (CurTok.getIdentifierInfo() == ArgII) {
      if (TokIdx > 0 && TokIdx < TokSize - 1) {
        auto PrevTok = *(MI->tokens_begin() + TokIdx - 1),
        NextTok = *(MI->tokens_begin() + TokIdx + 1);
        if (PrevTok.is(tok::l_paren) && NextTok.is
        (tok::r_paren))
        continue;
      }
      …
    }
  }
}
```

IdentifierInfo 实例的唯一性

回想一下，相同的标识符字符串总是由相同的 IdentifierInfo 对象表示。这就是这里可以简单地使用指针比较的原因。

MacroInfo::tokens_begin 函数返回一个迭代器，指向一个包含所有宏体令牌的数组开头。

最后，如果宏参数令牌没有括起来，则会打印一个警告消息，如下所示:

```
for (const IdentifierInfo *ArgII : ArgsToEnclosed) {
  for (auto TokIdx = 0U, TokSize = MI->getNumTokens();
  TokIdx < TokSize; ++TokIdx) {
    …
    if (CurTok.getIdentifierInfo() == ArgII) {
      if (TokIdx > 0 && TokIdx < TokSize - 1) {
        …
        if (PrevTok.is(tok::l_paren) && NextTok.is
        (tok::r_paren))
        continue;
      }
```

```
      SourceLocation TokLoc = CurTok.getLocation();
      errs() << "[WARNING] In " << TokLoc.printToString(SM)
      << ": ";
      errs() << "macro argument '" << ArgII->getName()
      << "' is not enclosed by parenthesis\n";
    }
  }
}
```

这部分就讲到这里了。读者们现在可以开发一个 PragmaHandler 插件，可以动态加载到 Clang 中来处理自定义的 #pragma 指令。还了解了如何实现 PPCallbacks，以便在预处理器事件发生时插入自定义逻辑。

6.5. 总结

本章从预处理器和词法分析器标记前端的开始。前者用其他文本内容替换预处理器指令，而后者将源代码划分成更有意义的标记 (或令牌)。本章中，我们学习了这两个组件如何相互协作，以提供一个令牌流的视图，以便在后面的阶段中使用。此外，还学习了各种重要的 API，比如：Preprocessor 类、Token 类，以及如何在 Clang 中表示宏——这些 API 可以用于本部分的开发，特别是用于创建支持自定义 #pragma 指令的处理程序插件，以及创建定制的预处理器回调，以便与预处理事件进行更深入的集成。

按照 Clang 编译阶段的顺序，下一章将展示如何使用**抽象语法树 (AST)**，以及如何开发一个 AST 插件，并将自定义逻辑插入其中。

6.6. 练习

这里有一些简单的问题和练习，读者们可以自己尝试一下:

1. 虽然大多数时候 Token 是从提供的源代码中获取的，但在某些情况下，Token 可能会在预处理器中动态生成。例如，__LINE__内置宏会扩展为当前行号，而__DATE__宏会扩展为当前日期。Clang 如何将生成的文本内容放入 SourceManager 的源代码缓冲区？Clang 如何分配 SourceLocation 给这些令牌？
2. 当我们讨论实现一个自定义 PragmaHandler 时，可以利用 Preprocessor::Lex 来获取 pragma 名称后面的 Token，直到我们遇到 eod 令牌类型。我们能继续在 eod 令牌之外进行词法分析吗？如果可以在 #pragma 指令后使用任意令牌，你会做哪些有趣的事情？
3. 在 macro 保护项目的开发自定义预处理器插件和回调部分，警告消息的格式为 [warning] In <source location>:…。显然，这不是我们通常从 clang 中看到的编译器警告。通常的警告会是这样，<source location>: warning:…，如下所示:

```
./simple_warn.c:2:7: warning: unused variable 'y'…
  int y = x + 1;
      ^


1 warning generated.
```

在某些终端中，警告字符串是彩色的。我们如何打印这样的警告消息？在 Clang 有这样的基础设施吗?

第 7 章 处理 AST

在前一章中，了解了 Clang 的预处理器如何处理 C 族语言中的预处理指令。我们还了解了如何编写不同类型的预处理器插件，比如：pragma 处理程序，以扩展 Clang 的功能。当涉及到实现特定领域的逻辑或甚至定制语言特性时，这些技能尤为有用。

本章中，将讨论解析后的原始源代码文件的**语义感知**表示，即**抽象语法树 (AST)**。AST 是一种包含丰富语义信息的格式，其中包括类型、表达式树和符号等。不仅可用作生成 LLVM IR 的蓝图，供以后的编译阶段使用，而且还是执行静态分析的推荐格式。除此之外，Clang 还提供了一个很好的框架，让开发者可以通过一个简单的插件接口在前端管道的中间拦截和操作 AST。

在本章中，我们将介绍如何在 Clang 中处理 AST，内存中 AST 表示的重要 API，以及如何编写 AST 插件来实现自定义逻辑。我们将讨论以下内容:

- Clang 中的 AST
- 编写 AST 插件

在本章结束时，将了解如何在 Clang 中使用 AST，以便在源代码级别对程序进行分析。此外，还将了解如何通过 AST 插件轻松地将自定义 AST 处理逻辑注入到 Clang 中。

7.1. 相关准备

本章需要构建 `clang` 可执行文件。如果没有，可以使用以下命令构建:

```
$ ninja clang
```

另外，可以使用下面的命令行标志来打印 AST 的文本表示:

```
$ clang -Xclang -ast-dump foo.c
```

例如，`foo.c` 包含以下内容:

```
1 int foo(int c) { return c + 1; }
```

使用`-Xclang -ast-dump` 命令行标志，我们可以为 `foo.c` 输出 AST:

```
TranslationUnitDecl 0x560f3929f5a8 <<invalid sloc>> <invalid sloc>
|···
`-FunctionDecl 0x560f392e1350 <foo.c:2:1, col:30> col:5 foo
'int (int)'
  |-ParmVarDecl 0x560f392e1280 <col:9, col:13> col:13 used c 'int'
```

```
`-CompoundStmt 0x560f392e14c8 <col:16, col:30>
  `-ReturnStmt 0x560f392e14b8 <col:17, col:28>
    `-BinaryOperator 0x560f392e1498 <col:24, col:28> 'int' '+'
      |-ImplicitCastExpr 0x560f392e1480 <col:24> 'int' <LValueToRValue>
      | `-DeclRefExpr 0x560f392e1440 <col:24> 'int' lvalue ParmVar
      0x560f392e1280 'c' 'int'
      `-IntegerLiteral 0x560f392e1460 <col:28> 'int' 1
```

这个标志对于找出用什么 C++ 类来表示代码非常有用。例如，正式的函数参数/参数由 `ParmVarDecl` 类表示，在前面的代码中突出显示了这个类。

本章的代码示例连接: https://github.com/PacktPublishing/LLVM-Techniques-Tips-and-Best-Practices-Clangand-Middle-End-Libraries/tree/main/Chapter07.

7.2. Clang 中的 AST

本节中，我们将了解 Clang 的 AST 内存表示和基本 API 的用法。本节的第一部分将提供 Clang AST 的层次结构的高级概述；第二部分将关注一个更具体的主题——关于 Clang AST 中的类型表示；最后一部分将展示 AST 匹配器的基本用法，这在编写 AST 插件时非常有用。

7.2.1 Clang 的 AST 内存表示

在 Clang 中，AST 的内存表示为一个层次结构，类似于 C 族语言程序的语法结构。从最顶层开始，这里有两个类值得一提:

- `TranslationUnitDecl`: 该类表示一个输入源文件，也称为翻译单元 (大多数情况下)。包含所有顶级声明——举几个例子，全局变量、类和函数——作为子声明，其中每个顶级声明都有自己的子树，子树递归地定义 AST 的其余部分。
- `ASTContext`: 顾名思义，这个类跟踪所有 AST 节点和来自输入源文件的其他元数据。如果有多个输入源文件，每个都有自己的 `TranslationUnitDecl`，但都共享相同的 `ASTContext`。

除了结构之外，AST 的主体 (AST 节点) 还可以进一步分为三个主要类别: **声明**、**语句**和**表达式**。这些类别中的节点分别由派生自 `Decl`、`Expr` 和 `Stmt` 类的子类表示。在后续的部分中，我们将介绍这些内存中的 AST 表示。

声明

语言结构，如变量声明 (全局和局部)、函数和结构/类声明，都是由 `Decl` 的子类表示。虽然我们不打算在这里详细介绍每一个子类，但下图展示了 C/C++ 中常见的声明结构和它们对应的 AST 类:

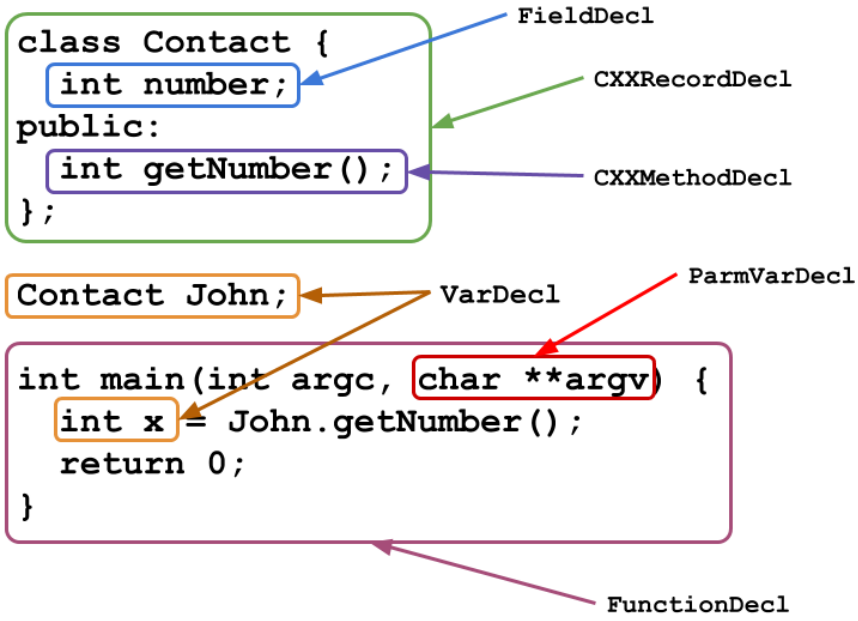

图 7.1 - C/C++ 中常见的声明及其 AST 类

在更具体的子类之间，比如 `FunctionDecl` 和 `Decl`，有几个重要的抽象类代表特定的语言概念:

- `NamedDecl`: 每个声明都有名称。
- `ValueDecl`: 声明的实例可以是一个值，从而声明可以与类型信息相关联。
- `DeclaratorDecl`: 对于每个声明 (基本上是以 `< 类型和限定符 > < 标识符名称 >` 形式的语句)，提供了除标识符之外的有关部件的额外信息。例如，提供对具有名称空间解析的内存对象的访问，该名称空间解析在声明器中充当了限定符。

要了解更多关于 AST 类的其他声明，可以在 LLVM 的官方 API 在线网站上浏览 `Decl` 的子类。

语句

程序中大多数表示动作概念的指令都可以视为语句，并由 `Stmt` 的子类表示，包括表达式 (我们很快就会讲到)。除了命令式语句 (如函数调用或返回站点) 之外，`Stmt` 还涵盖了结构概念，如 `for` 循环和 `if` 语句。下面这张图展示了 C/C++ 中 `Stmt`(表达式除外) 表示的通用语言结构，及其对应的 AST 类:

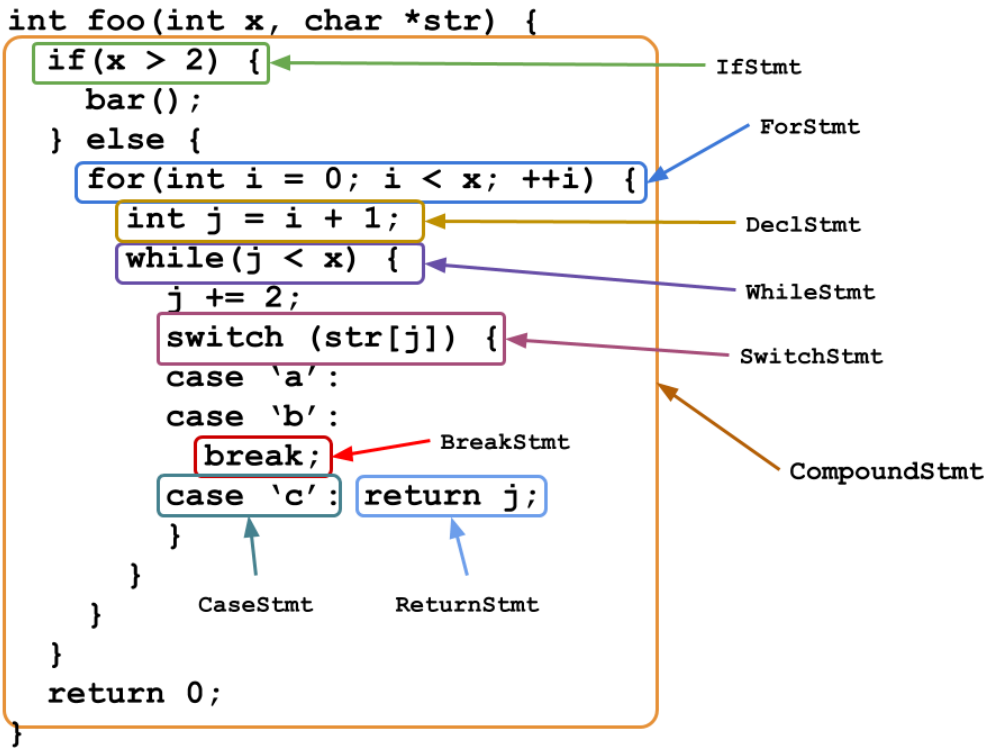

图 7.2 - C/C++ 及其 AST 类中的通用语句 (表达式除外)

关于上图，有两点值得注意:

- `CompoundStmt` 是一个包含多个语句的容器，不仅表示函数体，而且基本上表示由大括号 (`'{'`, `'}'`) 包围的任何代码块。因此，尽管由于缺少空间在前面的图中没有显示，但 `IfStmt`、`ForStmt`、`WhileStmt` 和 `SwitchStmt` 都有一个表示主体的 `CompoundStmt` 子节点。
- `CompoundStmt` 中的声明使用 `DeclStmt` 节点包装，其中真正的 `Decl` 实例是它的子节点。这是一个更简单的 AST 设计。

语句是 C/C++ 程序中最流行的指令之一。值得注意的是，许多语句都组织在一个层次结构中 (例如，`ForStmt` 及其循环体)，因此在找到所需的 `Stmt` 节点之前，可能需要执行额外的步骤来深入这个层次结构。

表达式

Clang AST 中的表达式是一种特殊的语句。与其他语句不同，表达式总是生成值。例如，期望一个简单的算术表达式 3 + 4 生成一个整数值。Clang AST 中的所有表达式都由 `Expr` 的子类表示。下面这张图展示了 C/C++ 中常用的 `Expr` 表示的语言结构及其对应的 AST 类:

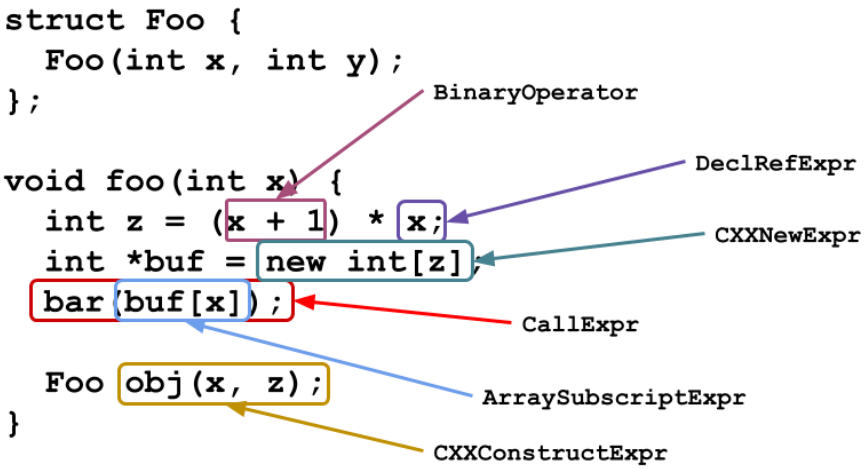

图 7.3 - C/C++ 中的常见表达式及其 AST 类

一个重要的 `Expr` 类是 `DeclRefExpr`,表示符号引用的概念。可以使用 `DeclRefExpr::getDecl()` 来检索引用符号的 `Decl` 对象。像这样方便的符号信息只有在生成 AST 之后才会出现，所以这就是为什么人们总是建议在 AST 上实现静态分析逻辑，而不是在更原始的形式上 (例如，在解析器中)。

另一个有趣的 `Expr` 类 (由于缺少空格，上图中没有突出显示) 是 `ParenExpr`，它表示包围表达式的圆括号。例如，在前面的图中，(x + 1) 是一个 `ParenExpr`，它带有一个 `BinaryOperator`，将 x + 1 表示为它的子元素。

7.2.2 Clang AST 中的类型

类型系统是现代编译器中最重要的组件之一，特别是对于 C/C++ 这样的静态类型语言。类型检查确保输入源代码是格式良好的 (某种程度上)，并在编译时捕获尽可能多的错误。虽然不需要在 Clang 中自己做类型检查，检查的操作由 Sema 子系统完成 (在第 5 章中介绍了这个子系统)。在处理 AST 时，可能需要利用这些信息。让我们了解如何在 Clang AST 中建模类型。

核心类型

Clang AST 类型系统的核心是 `clang::Type` 类。输入代码中的每种类型——包括 `int` 这样的基本类型和 struct/class 这样的用户定义类型——都由一个单例类型 (更具体地说，是 `Type` 的子类) 对象表示。

术语

本章的其余部分中，将调用输入源代码中的类型。

单例是一种设计模式，它强制一个资源或抽象概念只能由一个内存对象表示。我们的例子中，源代码类型就是资源，所以对于每一种类型，您只能找到一个 `Type` 对象。这种设计的最大优点是，有一种更简单的方法来比较两个 `Type` 对象。假设有两个 `Type` 指针，通过对它们做简单的指针比较 (非常快)，就可以知道它们是否表示相同的源代码类型。

单例设计的反例

如果 Clang AST 中的 `Type` 不是使用单例设计，为了比较两个 `Type` 指针是否代表相同的源代码类型，需要检查它们指向的对象的内容，不过这是无效的。

正如前面提到的，每个源代码类型实际上都由 `Type` 的子类表示。下面是一些常见的 `Type` 子类:

- `BuiltinType`: 基本类型，如 `int`、`char` 和 `float`。
- `PointerType`: 所有的指针类型，有一个名为 `PointerType::getPointee()` 的函数，用于检索它所指向的源代码类型。
- `ArrayType`: 所有的数组类型。这里，还有其他更特定的数组的子类，这些数组要么是定长的，要么是变长的。
- `RecordType`: 结构/类/联盟类型。它有一个名为 `RecordType::getDecl()` 的函数，用于检索底层的 `RecordDecl`。
- `FunctionType`: 用于表示函数的签名，用于表示函数的参数类型和返回类型 (以及其他属性，比如调用约定)。

现在让我们继续讨论限定类型。

限定类型

对于刚接触 Clang 代码库的人来说，最令人困惑的事情是，许多地方使用 `QualType` 类，而不是 `Type` 的子类来表示源代码类型。`QualType` 表示**限定类型**。它充当 `Type` 的包装器，以表示 `const <type>`、`volatile <type>` 和 `restrict <type>*`。

要从类型指针创建 `QualType`，可以使用以下代码:

```
// If `T` is representing 'int'···
QualType toConstVolatileTy(Type *T) {
  return QualType(T, Qualifier::Const | Qualifier::Volatile);
} // Then the returned QualType represents `volatile const int`
```

在本节中，我们了解了 Clang AST 中的类型系统。现在让我们继续了解 ASTMatcher——一种匹配模式的语法。

7.2.3 ASTMatcher

当处理程序的 AST 时——例如，检查是否有任何次优语法——搜索特定的 AST 节点模式通常是第一步，是人们最常做的事情。利用在前一节中学到的知识，我们知道这种模式匹配可以通过通过 AST 节点的内存类 API 迭代来完成。例如，给定一个 `FunctionDecl`(函数的 AST 类)，可以使用下面的代码来确定函数体中是否有一个 `while` 循环，以及该循环的退出条件是否总是一个布尔值，或者说是否为 `true`:

```
// `FD` has the type of `const FunctionDecl&`
const auto* Body = dyn_cast<CompoundStmt>(FD.getBody());
for(const auto* S : Body->body()) {
  if(const auto* L = dyn_cast<WhileStmt>(S)) {
```

```
    if(const auto* Cond = dyn_cast<CXXBoolLiteralExpr>
    (L->getCond()))
    if(Cond->getValue()) {
      // The exit condition is `true`!!
    }
  }
}
```

它创建了超过三层 (缩进) 的 `if` 来完成这样一个简单的检查。更不要说在实际情况中，我们需要在这些行之间插入更多的健全性检查! 虽然 Clang 的 AST 设计并不难理解，但我们需要更简洁的语法来完成模式匹配工作。幸运的是，Clang 已经提供了一个——**ASTMatcher**。

ASTMatcher 是一个实用工具，可以通过干净、简洁和高效的**领域特定语言 (DSL)** 编写 AST 模式匹配逻辑。使用 ASTMatcher，前面的代码中做相同的匹配只需要几行代码:

```
functionDecl(compountStmt(hasAnySubstatement(
  whileStmt(
    hasCondition(cxxBoolLiteral(equals(true)))))));
```

前面代码片段中的大多数指令都非常简单: 函数调用，如 `compoundStmt(…)` 和 `whileStmt(…)`，检查当前节点是否匹配特定的节点类型。这里，这些函数调用中的参数要么表示其子树上的模式匹配器，要么检查当前节点的其他属性。还有其他一些指令用于表示限定概念 (例如，对于这个循环体中的所有子语句，都存在一个返回值)，例如 `hasAnySubstatement(…)`，以及用于表示数据类型和常量值的指令，例如 `cxxBoolLiteral(equals(true))` 的组合。

简而言之，使用 ASTMatcher 可以使您的模式匹配逻辑更具表现力。本节中，我们展示了这个优雅的 DSL 的基本用法。

遍历 AST

在深入研究核心语法之前，我们先了解 ASTMatcher 是如何遍历 AST 的；以及在匹配过程完成后，是如何将结果返回给用户的。

`MatchFinder` 是模式匹配过程中常用的驱动程序，基本用法非常简单:

```
using namespace ast_matchers;
…
MatchFinder Finder;
// Add AST matching patterns to `MatchFinder`
Finder.addMatch(traverse(TK_AsIs, pattern1), Callback1);
Finder.addMatch(traverse(TK_AsIs, pattern2), Callback2);
…
// Match a given AST. `Tree` has the type of `ASTContext&`
// If there is a match in either of the above patterns,
// functions in Callback1 or Callback2 will be invoked
// accordingly
Finder.matchAST(Tree);

// …Or match a specific AST node. `FD` has the type of
// `FunctionDecl&`
Finder.match(FD, Tree);
```

pattern1 和 pattern2 是由 DSL 构造的模式对象，如前面所示。更有趣的是 traverse 函数和 TK_AsIs 参数。traverse 函数是模式匹配 DSL 的一部分，但它不是表示模式，而是描述遍历 AST 节点的操作。在此之上，TK_AsIs 参数可以表示遍历模式。

当我们在本章前面展示了以文本格式转储 AST 的命令行标志 (-Xclang -ast-dump) 时，您可能已经发现许多隐藏的 AST 节点会插入到树中，以帮助处理程序的语义，而不表示为程序员编写的真正代码。例如，ImplicitCastExpr 会插入到很多地方，以确保程序的类型正确性。在编写模式匹配逻辑时，处理这些节点可能是一种痛苦的体验。因此，遍历函数提供了另一种简化的遍历树的方法。假设我们有以下源码:

```
struct B {
  B(int);
};
B foo() { return 87; }
```

当传递 TK_AsIs 作为遍历的第一个参数时，其会先观察树，就像-ast-dump 那样:

```
FunctionDecl
`-CompoundStmt
  `-ReturnStmt
    `-ExprWithCleanups
      `-CXXConstructExpr
        `-MaterializeTemporaryExpr
          `-ImplicitCastExpr
            `-ImplicitCastExpr
              `-CXXConstructExpr
                `-IntegerLiteral 'int' 87
```

然而，通过将 TK_IgnoreUnlessSpelledInSource 作为第一个参数，观察到的树等于如下所示:

```
FunctionDecl
`-CompoundStmt
  `-ReturnStmt
    `-IntegerLiteral 'int' 87
```

顾名思义，TK_IgnoreUnlessSpelledInSource 只访问在源码中真正显示的节点。因为我们不再需要担心 AST 的本质细节，所以这极大地简化了编写匹配模式的过程。

另一方面，第一个代码中的 Callback1 和 Callback2 是 MatchFinder::MatchCallback 对象，它们描述了当有匹配时要执行的操作。下面是 MatchCallback 实现的框架:

```
struct MyMatchCallback : public MatchFinder::MatchCallback {
```

```
  void run(const MatchFinder::MatchResult &Result) override {
    // Reach here if there is a match on the corresponding
    // pattern
    // Handling "bound" result from `Result`, if there is any
  }
};
```

下一节中，我们将向您展示如何使用标记绑定模式的特定部分，并在 `MatchCallback` 中对其进行检索。

最后但同样重要的是，尽管在第一个代码片段中使用了 `MatchFinder::match` 和 `MatchFinder::matchAST` 来启动匹配过程，但还有其他方法也可以实现这一点。例如，可以使用 `MatchFinder::newASTConsumer` 来创建一个 `ASTConsumer` 实例，其将运行所描述的模式匹配活动。或者，可以使用 `ast_matchers::match(…)`(不是 `MatchFinder` 下的成员函数，而是一个独立函数) 在返回匹配的节点之前，在一次运行中对所提供的模式和 `ASTContext` 进行匹配。

ASTMatcher 的 DSL

ASTMatcher 提供了一个易于使用和简洁的 C++ DSL 来帮助匹配 AST。如前所示，所需模式的结构是通过嵌套函数调用表示的，其中每个函数表示要匹配的 AST 节点的类型。

使用此 DSL 来表示简单的模式就会非常简单。然而，当您尝试使用多个条件/谓词组合模式时，事情会变得有点复杂。虽然我们知道 `forStmt(…)` 指令可以匹配 for 循环 (例如，`for (I = 0; I < 10;++I){…}`)，但如何添加一个条件到它的初始化语句 `(I = 0)` 和退出条件 `(I < 10)`，或是循环体中呢？不仅官方 API 参考网站 (通常使用的 doxygen 网站) 在这方面缺乏明确的说明，大多数 DSL 函数在如何接受广泛的参数，在作为子模式方面也相当灵活。例如，在匹配 `for` 循环的问题之后，可以使用下面的代码只检查循环体:

```
forStmt(hasBody(…));
```

或者，可以检查它的循环体和退出条件，像这样:

```
forStmt(hasBody(…),
  hasCondition(…));
```

这个问题的一个泛化版本是，给定一个任意的 DSL 指令，我们如何知道哪些可用的指令可以与之结合？

为了回答这个问题，我们将利用一个专门为 ASTMatcher 创建的文档网站 LLVM: https://clang.llvm.org/docs/LibASTMatchersReference.html。这个网站由三大列表组成，表中显示了每种 DSL 指令的返回类型和参数类型:

图 7.4 - ASTMatcher DSL 参考的一部分

虽然这个表只是普通 API 引用的简化版本，但它已经展示了如何搜索候选指令。例如，现在了解了 `forStmt(…)` 可以接受 0 个或多个 `Matcher<forStmt>`，这样急救可以在这个表中搜索返回 `Matcher<forStmt>` 或 `Matcher<(forStmt 的父类)>` 的指令，例如：`Matcher<stmt>`。本例中，我们可以快速地找到 `hasCondition`、`hasBody`、`hasIncrement` 或 `hasLoopInit` 作为候选指令 (当然，也可以使用许多其他返回 `Matcher<stmt>` 的指令)。

当执行模式匹配时，不仅想知道一个模式是否匹配，还想获得匹配的 AST 节点。在 ASTMatcher 的上下文中，DSL 指令只检查 AST 节点的类型。如果想要检索 (部分) 匹配的具体 AST 节点，可以使用 `bind(…)`。下面是一个例子:

```
forStmt(
  hasCondition(
    expr().bind("exit_condition")));
```

这里，使用 `expr()` 作为通配符模式来匹配 `expr` 节点。这个指令还调用 `bind(…)` 来将匹配的 `Expr` AST 节点与 `exit_condition` 的名称相关联。

然后，在前面介绍的 `MatchCallback` 中，我们可以使用以下代码检索绑定节点:

```
…
void run(const MatchFinder::MatchResult &Result) override {
  cons auto& Nodes = Result.Nodes;
  const Expr* CondExpr = Nodes.getNodeAs<Expr>
  ("exit_condition");
  // Use `CondExpr`…
}
```

`getNodeAs<…>(…)` 函数试图获取给定名称下绑定的 AST 节点，并将其转换为模板参数建议的类型。

注意，这里允许以相同的名称绑定不同的 AST 节点，这种情况下，在 MatchCallback::run 中，只显示最后一个有界的节点。

合二为一

现在已经了解了模式匹配 DSL 语法和如何使用 ASTMatcher 遍历 AST，让我们把这两件事放在一起。

假设我们想知道一个简单的 `for` 循环 (循环索引从 0 开始，每次迭代递增 1，以一个整数为界) 在函数中的迭代次数——也称为 `trip count`:

1. 首先，必须编写以下代码来进行匹配和遍历:

```
auto PatExitCondition = binaryOperator(
                            hasOperatorName("<"),
                            hasRHS(integerLiteral()
                            .bind("trip_count")));
auto Pattern = functionDecl(
                   compountStmt(hasAnySubstatement(
               forStmt(hasCondition(PatExitCondition)))));

MatchFinder Finder;
auto* Callback = new MyMatchCallback();
Finder.addMatcher(traverse(TK_IgnoreUnlessSpelledInSource,
                            Pattern), Callback);
```

   前面的代码片段还展示了模块化 DSL 模式的情况。可以创建单独的模式，只要它们兼容，可以根据需要组合它们。

   最后，`MyMatchCallback::run` 是这样的:

```
void run(const MatchFinder::MatchResult &Result) override
{
  const auto& Nodes = Result.Nodes;
  const auto* TripCount =
  Nodes.getNodeAs<IntegerLiteral>("trip_count");
  if (TripCount)
  TripCount->dump(); // print to llvm::errs()
}
```

2. 在此之后，可以使用 `Finder` 在 AST 上匹配所需的模式 (通过调用 `MatchFinder::match` 或 `MatchFinder::matchAST`，或通过使用 `MatchFinder::newASTConsumer` 创建一个 `ASTConsumer`)。匹配的行程计数将打印到 `stderr`，例如：输入的源代码是 `for (int i = 0; i < 10; ++i) {…}`，输出结果将是 10。

本节中，我们了解了 Clang 如何构造 AST，如何在内存中表示 Clang AST，以及如何使用 ASTMatcher 来帮助开发人员进行 AST 模式匹配。有了这些知识，在下一节中，我们将展示如何创建一个 AST 插件，这是将自定义逻辑注入到 Clang 编译管道最简单的方法。

7.3. 编写 AST 插件

前一节中，我们了解了如何在 Clang 中表示 AST，以及它的内存类是什么样子的。我们还了解了一些有用的技巧，可以用来在 Clang AST 上执行模式匹配。这一节中，我们将了解如何编写插件，并将自定义的 AST 处理逻辑插入到 Clang 的编译管道中。

本节将分为三个部分:

- **项目概述**: 我们将在本节中创建的演示项目的目标和概述。

- **打印诊断消息**: 在我们深入开发插件的核心之前，我们将了解如何使用 Clang 的 `DiagnosticsEngine`，这是一个功能强大的子系统，可以打印出格式良好且有意义的诊断消息。这将使我们的演示项目更适用于实际场景。
- **创建 AST 插件**: 这一节将向你展示如何从头开始创建一个 AST 插件，填充所有实现细节，以及如何使用 Clang 运行它。

7.3.1 项目概述

本节中，我们将创建一个插件，每当输入代码中出现可转换为**三元操作符**的 `if-else` 语句时，该插件就会用警告消息提示用户。

快速复习——三元运算符

三元运算符 `x ? val_1 : val_2`，当 `x` 为真时，结果为 `val_1`，否则为 `val_2`。

例如，让我们看看下面的 C/C++ 代码:

```
int foo(int c) {
  if (c > 10) {
    return c + 100;
  } else {
    return 94;
  }
}

void bar(int x) {
  int a;
  if (x > 10) {
    a = 87;
  } else {
    a = x - 100;
  }
}
```

两个函数中的 `if-else` 语句都可以转换成三元操作符，如下所示:

```
int foo(int c) {
  return c > 10? c + 100 : 94;
}

void bar(int x) {
  int a;
  a = x > 10? 87 : x - 100;
}
```

在本项目中，我们只关注寻找两种潜在的三元操作符机会:

- `then` 块 (true 分支) 和 `else` 块 (false 分支) 都包含一个 `return` 语句。本例中，可以将它们的返回值和分支条件合并成一个三元操作符 (作为新的返回值)。

- then 块和 else 块都只包含一个赋值语句。这两个语句都使用一个 DeclRefExpr(即符号引用) 作为 LHS，并且两个 DeclRefExpr 对象都指向同一个 Decl(符号)。换句话说，讨论前面代码片段中的 bar 函数的情况。请注意，我们不讨论 LHS 更复杂的情况，例如：数组下标，a[i] 用作 LHS 的情况。

在识别这些模式之后，我们必须向用户提示警告信息，并提供额外的信息来帮助用户解决这个问题:

```
$ clang ···(flags to run the plugin) ./test.c
./test.c:2:3: warning: this if statement can be converted to ternary operator:
  if (c > 10) {
  ^
./test.c:3:12: note: with true expression being this:
    return c + 100;
           ^
./test.c:5:12: note: with false expression being this:
    return 94;
           ^
./test.c:11:3: warning: this if statement can be converted to ternary operator:
  if (x > 10) {
  ^
./test.c:12:9: note: with true expression being this:
    a = 87;
        ^
./test.c:14:9: note: with false expression being this:
    a = x - 100;
        ^
2 warnings generated.
```

每个警告消息 (说明哪个 if-else 语句可以转换为三元操作符) 后面都有两条注释，指出要为该操作符构造的可能表达式。

与手工编译器消息相比 (第 6 章)，这里我们使用 Clang 的诊断设施来打印带有更丰富信息的消息，比如：消息所引用的代码快照。接下来，我们将展示如何使用该诊断设施。

7.3.2 打印诊断消息

第 6 章中，我们提出是否可以开发自定义预处理器插件和回调的示例中改进警告消息格式的问题，以便它更接近 Clang 现有的消息格式。这个问题的解决方案之一是使用 Clang 的诊断框架。我们将在本节中继续来探讨这个问题。

Clang 的诊断框架由三个主要部分组成:

- 诊断 ID
- 诊断引擎
- 诊断用户 (客户端)

它们的关系如下图所示:

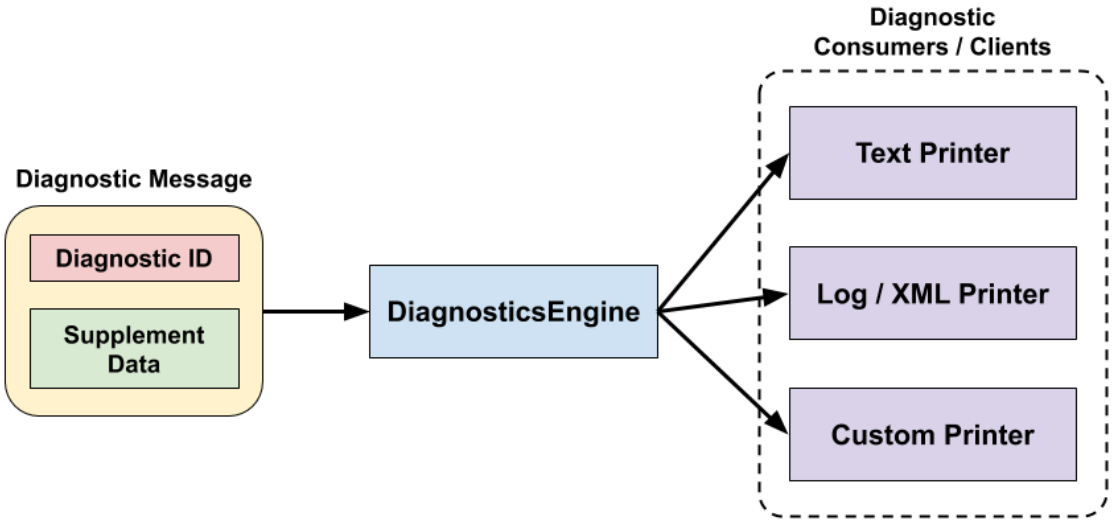

图 7.5 - Clang 诊断框架的高层组织结构

诊断信息

从上图的左边开始，诊断消息——例如，使用未声明的标识符“x”——与具有自己诊断 ID 的消息模板相关联，“使用未声明的标识符消息”的消息模板如下所示:

```
"use of undeclared identifier %0"
```

%0 是一个**占位符**，稍后将由相应数据填充。本例中，它是具体的标识符名称 (在前面的示例消息中是 x)。% 后面的数字也表明了它将使用的数据。我们稍后将会详细讨论这种格式。

模板通过 TableGen 语法注册到诊断引擎。例如，这里讨论的消息模板就放在 `clang/include/clang/Basic/DiagnosticSemaKinds.td` 中:

```
def err_undeclared_var_use : Error<"use of undeclared identifier %0">;
```

在前面的代码中，我们强调了两个部分。首先，这个消息模板的名称 `err_undeclare_var_use`，稍后将用作唯一的诊断 ID。第二，`Error` 是一个带有错误消息的 TableGen 类，或者更正式地说，带有的是诊断级别错误。

总之，诊断消息由唯一的诊断 ID(与消息模板及其诊断级别相关联) 和数据组成，这些数据用于插入模板的占位符 (如果有的话)。

诊断用户

诊断消息发送到诊断引擎 (由 DiagnosticsEngine 类表示) 之后，引擎将消息格式化为文本内容，并将它们发送给**诊断用户** (在代码库中也称为客户端)。

诊断用户 (DiagnosticConsumer 类的另一个实现) 对从 DiagnosticsEngine 发送的文本消息进行处理后，通过不同的介质导出它们。例如，默认的 TextDiagnosticPrinter 将消息打印到命令行界面; 另一方面，LogDiagnosticPrinter 将传入消息打印到日志文件之前，会用简单的 XML 标记来进行装饰。理论上，可以创建一个自定义的 DiagnosticConsumer 来发送诊断消息到远程主机!

报告诊断消息

现在你已经了解了 Clang 的诊断框架是如何工作的，让我们了解一下如何发送 (报告) 诊断消息到诊断引擎:

1. 首先，需要检索对 DiagnosticEngine 的引用。这个引擎位于 Clang 编译管道的核心，所以可以从各种主组件中获取，比如：ASTContext 和 SourceManager。示例如下:

```
// `Ctx` has the type of `ASTContext&`
DiagnosticsEngine& Diag = Ctx.getDiagnostics();
```

2. 接下来，需要使用 DiagnosticsEngine::Report 函数，将诊断 ID 作为其参数之一，例如：报告 err_undeclare _var_use(在前面介绍过)，就可以使用以下代码:

```
Diag.Report(diag::err_undeclared_var_use);
```

然而，err_undeclare_var_use 有一个占位符实参——即标识符名——通过 Report 函数和 << 操作符来提供:

```
Diag.Report(diag::err_undeclared_var_use) << ident_name_str;
```

3. err_undeclare_var_use 只有一个占位符%0，所以会获取 << 流中的第一个值。假设我们有一个诊断消息，err_invalid_placement，使用下面的模板:

```
"you cannot put %1 into %0"
```

4. 可以使用下面的代码进行报告:

```
Diag.Report(diag::err_invalid_placement)
            << "boiling oil" << "water";
```

5. 除了简单的占位符之外,另一个有用的特性是%select 指令,例如:有一个诊断消息 warn_exceed_limit，模板如下:

```
"you exceed the daily %select{wifi|cellular network}0 limit"
```

%select 指令由大括号组成，其中不同的消息选项由 | 分隔。在大括号之外，前面代码中的一个数字 (0) 表示使用哪些数据来填充大括号内的选项。下面是一个例子:

```
Diag.Report(diag::warn_exceed_limit) << 1;
```

上面的代码将输出 “**You exceed the daily cellular network limit**”。假设你使用 0 作为流操作符 («) 后面的参数:

```
Diag.Report(diag::warn_exceed_limit) << 0;
```

这将发出一个消息：超过了每天的 wifi 的流量限制 (**you exceed the daily wifi limit**)。

6. 假设使用另一个 `Report` 函数，它有一个 `SourceLocation` 参数:

```
// `SLoc` has the type of `SourceLocation`
Diag.Report(SLoc, diag::err_undeclared_var_use)
                  << ident_name_str;
```

输出消息将包含 `SLoc` 指向的部分源码:

```
test.cc:2:10: error: use of undeclared identifier 'x'
  return x + 1;
         ^
```

7. 最后,尽管大多数诊断消息是通过 Clang 源代码树中的 TableGen 代码注册到 `DiagnosticsEngine` 的，但这并不意味着开发者可以在修改 Clang 源代码树的情况下创建新的诊断消息。让我们来看下 `DiagnosticsEngine::getCustomDiagID(…)`，这个 API 可以从一个消息模板和开发人员提供的诊断级别创建新的诊断 ID:

```
auto MyDiagID = Diag.
getCustomDiagID(DiagnosticsEngine::Note,
"Today's weather is %0");
```

前面的代码创建的新诊断 ID 为 `MyDiagID`，它是在诊断级别上的消息模板，比如：**Today's weather is%0**。可以像使用其他 ID 一样使用这个诊断 ID:

```
Diag.Report(MyDiagID) << "cloudy";
```

在本节中，了解了如何利用 Clang 的诊断框架来打印消息，就像普通编译器消息一样。
接下来，我们将结合本章学到的所有技能来创建一个自定义的 AST 插件。

7.3.3 创建 AST 插件

本章的前几节中，我们探索了 Clang 的 AST，并了解了如何在内存 API 中使用。本节中，我们来了解如何编写一个插件，以一种简单的方式将自定义的 AST 处理逻辑插入到 Clang 的编译管道中。

在第 5 章，我们了解了使用 Clang (AST) 插件的优势: 即使使用一个预构建的 Clang 可执行文件，也可以进行开发使用，插件很容易编写，可以与现有的工具链和构建系统有很好的集成等。在第 6 章中，我们开发了一个用于预处理器中自定义编译处理的插件。本章中，我们还将编写一个插件，但这个插件为自定义 AST 处理而设计。这两个插件的代码框架也有很大的不同。

我们在项目概述一节中介绍了本节将使用的示例项目。如果输入代码中的某些 `if-else` 语句可以转换成三元运算符，这个插件会提示用户警告消息。此外，它还显示了关于构建三元操作符的候选表达式的提示。

下面是构建插件的详细步骤:

1. 与我们在第 6 章中看到的 pragma 插件类似，在 Clang 中创建一个插件基本上就像实现一个类。在开发 AST 插件时，将使用 `PluginASTAction` 类。

   `PluginASTAction` 是 `ASTFrontendAction` 的子类——专门用于处理 AST 的 `FrontendAction`(如果你不熟悉 `FrontendAction`,可以回顾一下第 5 章的内容)。因此,我们需要实现 `CreateASTConsumer` 成员函数:

```
struct TernaryConverterAction : public PluginASTAction {
  std::unique_ptr<ASTConsumer>
  CreateASTConsumer(CompilerInstance &CI,
                    StringRef InFile) override;
};
```

   我们将在稍后填充这个函数。

2. 除了 `CreateASTConsumer` 外,还可以重写另外两个成员函数来更改一些功能:`getActionType` 和 `ParseArgs`。前者告诉 Clang 如何执行这个插件，返回一个枚举值如下所示:

   a. `Cmdline`: 如果用户提供`-plugin <plugin name>`(前端的) 命令行标志，插件将在主操作之后执行。

   b. `ReplaceAction`: 这取代了 Clang 将要执行的原始动作，例如：Clang 将输入代码编译到一个目标文件 (`-c` 标志) 时，则执行插件的动作 (而不是在插件加载时执行)。

   c. `AddBefore/AfterMainAction`: 原来的 Clang 动作仍然会执行，并且插件动作会添加到它的前面或后面执行。

   这里，我们将使用 `Cmdline` 操作类型:

```
struct TernaryConverterAction : public PluginASTAction {
  ...
  ActionType getActionType() override { return Cmdline; }
};
```

   另一方面，`ParseArgs` 成员函数处理 (前端的) 特定于这个插件的命令行选项。换句话说，可以为插件创建自定义命令行标志。我们的例子中，将创建两个标志:`-no-detect-return` 和`-no-detect-assignment`。这允许我们决定是否希望检测返回语句或赋值语句的 (潜在) 三元转换:

```
struct TernaryConverterAction : public PluginASTAction {
  ...
  bool NoAssignment = false,
  NoReturn = false;
  bool ParseArgs(const CompilerInstance &CI,
  const std::vector<std::string> &Args) override {
    for (const auto &Arg : Args) {
      if (Arg == "-no-detect-assignment") NoAssignment =
      true;
      if (Arg == "-no-detect-return") NoReturn = true;
    }
      return true;
```

```
    }
};
```

如上面的代码所示，我们创建了两个布尔标记，NoReturn 和 NoAssignment，来存储命令行选项的值。ParseArgs 的返回值非常重要，ParseArgs 实际上是在返回插件是否应该继续执行，而不是返回是否解析了任何自定义标志。因此，在大多数情况下，总是返回 true。

3. 现在，我们将讨论 CreateASTConsumer 的内容。这个函数将返回一个 ASTConsumer 对象，它是自定义的逻辑主体。然而，我们不打算直接实现 ASTConsumer。相反，我们 ASTConsumer 对象是由 *ASTMatcher* 生成的 (已经在本章的前面内容中介绍过)。

   构建 MatchFinder 实例需要两个东西——ASTMatcher 中的主要模式匹配驱动 (用 ASTMatcher 自己的 DSL 编写的模式) 和一个 MatchCallback 实现。我们将模式和匹配器回调分为两类: 基于 return 语句检测可能的三元操作符的模式，以及基于赋值语句检测的模式。

   以下是 CreateASTConsumer 的框架:

```
using namespace ast_matchers;
struct TernaryConverterAction : public PluginASTAction {
  ...
private:
  std::unique_ptr<MatchFinder> ASTFinder;
  std::unique_ptr<MatchFinder::MatchCallback>
    ReturnMatchCB, AssignMatchCB;
};

std::unique_ptr<ASTConsumer>
TernaryConverterAction::CreateASTConsumer
(CompilerInstance &CI, StringRef InFile) {
  ASTFinder = std::make_unique<MatchFinder>();
  // Return matcher
  if (!NoReturn) {
    ReturnMatchCB = /*TODO: Build MatcherCallback
    instance*/
    ASTFinder->addMatcher(traverse
      (TK_IgnoreUnlessSpelledInSource,
        /*TODO: Patterns in DSL*/), ReturnMatchCB.get());
  }
  // Assignment matcher
  if (!NoAssignment) {
    AssignMatchCB = /*TODO: Build MatcherCallback
      instance*/
    ASTFinder->addMatcher(traverse
      (TK_IgnoreUnlessSpelledInSource,
        /*TODO: Patterns in DSL*/), AssignMatchCB.get());
  }
  return std::move(ASTFinder->newASTConsumer());
}
```

前面的代码创建了三个 unique_ptr 类型成员变量: 一个用于存储 MatchFinder，两个存储 MatchCallback 的用于基于返回和基于分配的模式。

为啥使用 unique_ptr?

使用 unique_ptr 来存储这三个对象——或者持久化存储这些对象——背后的原理是因为，要在 CreateASTConsumer (ASTFinder->newASTConsumer()) 末尾创建的 ASTConsumer 实例保留了对这三个对象的引用。因此，我们需要一种方法来保持它们在前端的生命周期内的存活性。

除此之外，通过使用 MatchFinder::addMatcher、traverse 函数和 MatchCallback 实例，在 MatchFinder 中注册了遍历模式。如果不熟悉这些 API，请会看关于 ASTMatcher 的部分章节。

现在，我们只需要组合匹配模式，并实现一些回调，以便在存在匹配时打印出警告消息——如前面代码中建议的 TODO 注释所示。

4. 先来处理模式。我们正在寻找的模式——基于返回和基于赋值的模式——在最外层的布局中有一个函数 (FunctionDecl 表示整个函数，CompoundStmt 表示函数体) 包围的 if-else 语句 (IfStmt)。两者内部，IfStmt 的真分支和假分支中，只能存在一条语句。这个结构可以这样描述:

```
FunctionDecl
  |_CompoundStmt
    |_(Other AST nodes we don't care)
    |_IfStmt
      |_(true branch: contain only one return/assign statement)
      |_(false branch: contain only one return/assign statement)
```

为了将这个概念转换成 ASTMatcher 的 DSL，下面是基于返回值和基于分配的模式之间共享的 DSL 代码:

```
functionDecl(
compoundStmt(hasAnySubstatement
    IfStmt(
      hasThen(/*TODO: Sub-pattern*/)
      hasElse(/*TODO: Sub-pattern*/)
    )
  )
);
```

要记住，当处理 CompoundStmt 时，应该使用限定指令，比如：让 hasAnySubstatement 来匹配它的主体语句。

我们将使用前面的 TODO 注释来定制这些基于返回值或基于分配的情况。来使用子模式变量来替换那些 TODO 注释，并把前面的代码放到另一个函数中:

```
StatementMatcher
buildIfStmtMatcher(StatementMatcher truePattern,
           StatementMatcher falsePattern) {
  return functionDecl(
    compoundStmt(hasAnySubstatement
```

```
        IfStmt(
      hasThen(truePattern)
      hasElse(falsePattern))));
}
```

5. 对于基于返回值的模式，相对于上一步中提到的两个 `if-else` 分支的子模式是稍微复杂一些。这里，我们也使用了一个单独的函数来创建这个模式:

```
StatementMatcher buildReturnMatcher() {
  return compoundStmt(statementCountIs(1),
            hasAnySubstatement(
            returnStmt(
            hasReturnValue(expr()))));
}
```

   如上面的代码所示，我们使用 `statementCountIs` 指令将代码块与只有一条语句匹配。此外，还通过 `hasReturnValue(…)` 指定了不为空的返回值。`hasReturnValue` 参数是必要的，因为后者至少接受一个参数，但由于我们不关心什么类型的节点，所以使用 `expr()` 作为某种通配符模式。

   对于基于赋值的模式，事情变得有点复杂: 我们不只是想在两个分支中匹配单个赋值语句 (由 `BinaryOperator` 类建模)——这些赋值的 LHS 需要是指向同一个 `Decl` 实例的 `DeclRefExpr` 表达式。不幸的是，我们不能使用 ASTMatch 的 DSL 来表达这些谓词。然而，可以将一些检查推到 `MatchCallback` 中，并且只使用 DSL 指令来检查我们想要的模式形状:

```
StatementMatcher buildAssignmentMatcher() {
  return compoundStmt(statementCountIs(1),
            hasAnySubstatement(
            binaryOperator(
            hasOperatorName("="),
            hasLHS(declRefExpr())
            )));
}
```

6. 现在我们已经完成了模式的框架,是时候来实现 `MatchCallback` 了。在 `MatchCallback::run` 中我们会做两件事。首先,对于基于分配的模式,检查那些匹配的分配候选 LHS 的 `DeclRefExpr` 是否指向相同的 `Decl`。其次，打印出帮助用户将 `if-else` 分支重写为三元运算符的消息。换句话说，我们需要一些匹配的 AST 节点的位置信息。

   让我们使用 *AST 节点绑定技术*来解决第一个任务。计划是绑定候选赋值的 LHS `DeclRefExpr` 节点，这样就可以从 `MatchCallback::run` 中对它们进行检索，并对它们的 `Decl` 节点执行进一步的检查。我们来对 `buildAssignmentMatch` 进行修改:

```
StatementMatcher buildAssignmentMatcher() {
  return compoundStmt(statementCountIs(1),
            hasAnySubstatement(
            binaryOperator(
            hasOperatorName("="),
            hasLHS(declRefExpr().
            bind("dest")))));
```

```
}
```

尽管前面的代码看起来很简单,但是在这个绑定方案中有一个问题: 在两个分支中,DeclRefExpr 的绑定使用了相同的名称，这意味着后面出现的 AST 节点将覆盖之前绑定的节点。因此，我们不会像之前计划的那样从两个分支获得 DeclRefExpr 节点。

因此，这里为 DeclRefExpr 使用一个不同的标签来匹配两个分支:dest.true 表示真分支，dest.false 表示假分支。让我们调整前面的代码来反映这个策略:

```
StatementMatcher buildAssignmentMatcher(StringRef Suffix)
{
  auto DestTag = ("dest" + Suffix).str();
  return compoundStmt(statementCountIs(1),
            hasAnySubstatement(
            binaryOperator(
            hasOperatorName("="),
            hasLHS(declRefExpr().
            bind(DestTag)))));
}
```

当调用 buildAssignmentMatcher 时,我们将为不同的分支传递不同的后缀——或者是.true 或者.false。

最后，必须在 MatchCallback::run 中检索绑定的节点。这里，我们为基于返回和基于分配的场景创建了不同的 MatchCallback 子类——分别是 MatchReturnCallback 和 MatchAssignmentCallback。下面是 MatchAssignmentCallback::run 中的部分代码:

```
void
MatchAssignmentCallback::run(const MatchResult &Result)
override {
  const auto& Nodes = Result.Nodes;
  // Check if destination of both assignments are the
  // same
  const auto *DestTrue =
      Nodes.getNodeAs<DeclRefExpr>("dest.true"),
      *DestFalse =
      Nodes.getNodeAs<DeclRefExpr>("dest.false");
  if (DestTrue->getDecl() == DestFalse->getDecl()) {
    // Can be converted into ternary operator!
  }
}
```

我们将在下一步解决第二个任务——打印有用的信息给用户。

7. 要打印有用的信息——包括代码的哪一部分可以转换成三元运算符，以及如何构建三元运算符——需要在获取其源位置信息之前，从匹配的模式中检索一些 AST 节点。为此，我们将使用一些节点绑定技巧。这一次，我们将修改所有的模式构建功能，也就是 buildIfStmtMatcher、buildReturnMatcher 和 builassignmentmatcher:

```
StatementMatcher
buildIfStmtMatcher(StatementMatcher truePattern,
           StatementMatcher falsePattern) {
```

```
  return functionDecl(
    compoundStmt(hasAnySubstatement
      IfStmt(
        hasThen(truePattern)
        hasElse(falsePattern)).bind("if_stmt")
    ));
}
```

这里，对匹配的 IfStmt 进行了绑定，因为我们想告诉用户，哪些地方可以转换为三元操作符:

```
StatementMatcher buildReturnMatcher(StringRef Suffix) {
  auto Tag = ("return" + Suffix).str();
  return compoundStmt(statementCountIs(1),
            hasAnySubstatement(
              returnStmt(hasReturnValue(
                expr().bind(Tag)
              ))));
}
StatementMatcher buildAssignmentMatcher(StringRef Suffix)
{
  auto DestTag = ("dest" + Suffix).str();
  auto ValTag = ("val" + Suffix).str();
  return compoundStmt(statementCountIs(1),
            hasAnySubstatement(
            binaryOperator(
            hasOperatorName("="),
            hasLHS(declRefExpr().
            bind(DestTag)),
            hasRHS(expr().bind(ValTag))
            )));
}
```

在这里使用的节点绑定技巧与前面的代码中使用的相同。在此之后，我们可以从 MatchCallback::run 中检索那些绑定的节点，并使用这些节点的 SourceLocation 信息进行打印。我们将在这里使用 Clang 的诊断框架来打印这些消息。由于预期的消息格式在 Clang 的代码库中并不存在，我们将通过 DiagnosticsEngine::getCustomDiagID(…) 创建自己的诊断 ID。下面是我们在 MatchAssignmentCallback::run 中做的事情 (因为 MatchReturnCallback 是类似的，所以这里只演示 MatchAssignmentCallback):

```
void
MatchAssignmentCallback::run(const MatchResult &Result)
override {
  …
  auto& Diag = Result.Context->getDiagnostics();
  auto DiagWarnMain = Diag.getCustomDiagID(
    DiagnosticsEngine::Warning,
    "this if statement can be converted to ternary
      operator:");
```

```
10
11  auto DiagNoteTrueExpr = Diag.getCustomDiagID(
12    DiagnosticsEngine::Note,
13    "with true expression being this:");
14
15  auto DiagNoteFalseExpr = Diag.getCustomDiagID(
16    DiagnosticsEngine::Note,
17    "with false expression being this:");
18  ...
19 }
```

结合绑定节点检索，下面是如何打印消息:

```
1  void
2  MatchAssignmentCallback::run(const MatchResult &Result)
3  override {
4    ...
5    if (DestTrue && DestFalse) {
6      if (DestTrue->getDecl() == DestFalse->getDecl()) {
7        // Can be converted to ternary!
8        const auto* If = Nodes.getNodeAs<IfStmt>
9        ("if_stmt");
10       Diag.Report(If->getBeginLoc(), DiagWarnMain);
11
12       const auto* TrueValExpr =
13                   Nodes.getNodeAs<Expr>("val.true");
14       const auto* FalseValExpr =
15                   Nodes.getNodeAs<Expr>("val.false");
16       Diag.Report(TrueValExpr->getBeginLoc(),
17                   DiagNoteTrueExpr);
18       Diag.Report(FalseValExpr->getBeginLoc(),
19                   DiagNoteFalseExpr);
20     }
21   }
22 }
```

8. 最后，返回 `CreateASTConsumer`。下面展示了，是所有内容是如何拼凑起来的:

```
1  std::unique_ptr<ASTConsumer>
2  TernaryConverterAction::CreateASTConsumer(
3  CompilerInstance &CI, StringRef InFile) {
4    ...
5    // Return matcher
6    if (!NoReturn) {
7    ReturnMatchCB = std::make_unique<MatchReturnCallback>();
8        ASTFinder->addMatcher(
9        traverse(TK_IgnoreUnlessSpelledInSource,
10                 buildIfStmtMatcher(
11                 buildReturnMatcher(".true"),
12                 buildReturnMatcher(".false"))),
13       ReturnMatchCB.get()
```

```
    );
  }

  // Assignment matcher
  if (!NoAssignment) {
  AssignMatchCB = std::make_
    unique<MatchAssignmentCallback>();
  ASTFinder->addMatcher(
      traverse(TK_IgnoreUnlessSpelledInSource,
               buildIfStmtMatcher(
               buildAssignmentMatcher(".true"),
               buildAssignmentMatcher(".false"))),
      AssignMatchCB.get()
    );
  }

  return std::move(ASTFinder->newASTConsumer());
}
```

这就是我们要做的所有事情!

9. 最后，运行插件的命令:

```
$ clang -fsyntax-only -fplugin=/path/to/TernaryConverter.
so -Xclang -plugin -Xclang ternary-converter \
  test.c
```

将得到一个类似于在**项目概述**中看到的输出。

要在这个项目中使用插件特定的标志，比如`-no-detect-return` 和`-no-detectassignment`，请在这里显式添加命令行选项:

```
$ clang -fsyntax-only -fplugin=/path/to/TernaryConverter.
so -Xclang -plugin -Xclang ternary-converter \
   -Xclang -plugin-arg-ternary-converter \
   -Xclang -no-detect-return \
    test.c
```

更具体地说，要以`-plugin-arg-<plugin name>` 格式添加第一个高亮显示的参数。

本节中，了解了如何编写 AST 插件，该插件在有 `if-else` 语句时向用户发送消息，表明该语句可以转换为三元操作符。可以利用本章所介绍的所有技术来做其他的事情，例如：Clang AST 的内存表示、ASTMatcher 和诊断框架。

7.4. 总结

当涉及到程序分析时，通常是推荐使用 AST，因为它具有丰富的语义信息和高级结构。本章中，我们了解了 Clang 中使用的强大的内存内 AST 表示，包括它的 C++ 类对象和 API。这可以让读者对正在分析的源代码有了一个清晰的了解。

此外，还通过 Clang 的 ASTMatcher 学习和实践了一种在 AST 上进行模式匹配的方法，AST 是程序分析的关键步骤。在从输入源代码中过滤出感兴趣的部分时，熟悉这种技术可以极大地提高效率。最后，我们了解了如何编写 AST 插件，它可以让你更容易地将自定义逻辑集成到默认的 Clang 编译管道中。

下一章中，我们将着眼于 Clang 中的驱动程序和工具链。我们将展示它们是如何工作的，以及如何进行定制。

第 8 章 使用编译器标志和工具链

前一章中，我们了解了如何处理 Clang 的 AST——一种最常见的程序分析格式。此外，我们还了解了如何开发 AST 插件，这是一种将自定义逻辑插入到 Clang 编译管道的简单方法。这些知识将增强，执行诸如源代码检测或发现潜在安全漏洞等任务的技能。

本章中，我们将从特定的子系统开始，并着眼于更大的场景——编译器**驱动**和**工具链**，根据用户的需求编排、配置和运行独立的 LLVM 和 Clang 组件。更具体地说，我们将关注如何添加新的编译器标志，以及如何创建自定义工具链。正如在第 5 章提到的，编译器驱动和工具链经常被低估，并长期被忽视。然而，如果没有这两个重要的工具，编译器将变得极其难以使用，例如：因为缺少标志转译，用户需要传递 10 个不同的编译器标志，仅仅是为了构建一个简单的 hello world 程序。因为没有驱动程序或工具链来调用汇编器和链接器，用户还需要运行至少三种不同类型的工具才能创建一个可执行程序来运行。本章中，将了解编译器驱动和工具链如何在 Clang 中工作，以及如何定制使用，如果想在新的操作系统或架构上支持 Clang，这章的知识将非常有用。

我们将讨论以下内容：

- Clang 中的驱动程序和工具链
- 添加自定义驱动标志
- 添加自定义工具链

8.1. 相关准备

本章中，我们仍然依赖于 `clang` 可执行文件，请先构建它:

```
$ ninja clang
```

因为使用了驱动，所以可以使用`-###` 命令行选项来打印从驱动程序中转译出来的前端标志：

```
$ clang++ -### -std=c++11 -Wall hello_world.cpp -o hello_world
"/path/to/clang" "-cc1" "-triple" "x86_64-apple-macosx11.0.0"
"-Wdeprecated-objc-isa-usage"
"-Werror=deprecated-objcisa-usage" "-Werror=implicit-function-declaration"
"-emitobj" "-mrelax-all" "-disable-free" "-disable-llvm-verifier"
… "-fno-strict-return" "-masm-verbose" "-munwind-tables"
"-target-sdk-version=11.0" … "-resource-dir" "/Library/
Developer/CommandLineTools/usr/lib/clang/12.0.0" "-isysroot"
"/Library/Developer/CommandLineTools/SDKs/MacOSX.sdk" "-I/
usr/local/include" "-stdlib=libc++" … "-Wall" "-Wno-reorderinit-list"
```

```
"-Wno-implicit-int-float-conversion" "-Wno-c99-
designator" ⋯ "-std=c++11" "-fdeprecated-macro"
 "-fdebugcompilation-dir"
"/Users/Rem" "-ferror-limit" "19"
"-fmessage-length" "87" "-stack-protector" "1" "-fstackcheck"
"-mdarwin-stkchk-strong-link" ⋯ "-fexceptions" ⋯
"-fdiagnostics-show-option" "-fcolor-diagnostics" "-o" "/path/
to/temp/hello_world-dEadBeEf.o" "-x" "c++" "hello_world.cpp"⋯
```

使用这个标志将不会运行其余的编译，而只是执行驱动程序和工具链。这个方法可以验证和调试特定标志，并检查它们是否正确地从驱动传播到前端。

最后，本章的最后一节，添加自定义工具链。我们将处理一个只能在 Linux 系统上运行的项目。另外，请提前安装 OpenSSL。在大多数 Linux 系统中，它通常是一个包。例如，在 Ubuntu 上，可以使用以下命令来安装:

```
$ sudo apt install openssl
```

我们只使用命令行工具，所以不需要安装任何通常用于开发的 OpenSSL 库。

本章的代码连接: https://github.com/PacktPublishing/LLVM-Techniques-Tips-and-Best-Practices-Clangand-Middle-End-Libraries/tree/main/Chapter08.

本章的第一节中，我们将简要介绍 Clang 的驱动和工具链基础结构。

8.2. Clang 中的驱动程序和工具链

在我们讨论 Clang 中的编译器驱动程序之前，有必要强调一下: 编译一段代码绝不是单一的任务 (也不是简单的任务)。学校里，老师们教导我们，编译器包括一个**词法分析器**，一个**解析器**，有时附带一个**优化器**，最后是一个**汇编代码打印器**。虽然在实际编译器中仍然可以看到这些阶段，但它们提供的只是文本汇编代码，而不是期望的可执行文件或库。此外，这个原生编译器的灵活性非常有限——不能移植到任何其他操作系统或平台上。

为了让这个玩具编译器更真实和可用，许多其他管道工工具需要放在一起，以及核心编译器: 一个将汇编代码转换为 (二进制格式) 对象文件的**汇编器**，一个将多个对象文件放入可执行文件或库的**链接器**，以及许多其他解析平台特定配置的例程，如：数据宽度、默认头文件路径或**应用二进制接口 (ABIs)**。只有在这些“水管工”的帮助下，我们才能通过输入几个单词来使用编译器:

```
$ clang hello_world.c -o hello_world
```

编译器驱动是组织这些“管道工”工作的软件。尽管在编译过程中有多个不同的任务要做，本

章中我们只关注两个最重要的任务——处理编译器标志和在不同平台上调用正确的工具——这就是工具链的设计目的。

下图显示了驱动、工具链和编译器其余部分之间的交互信息:

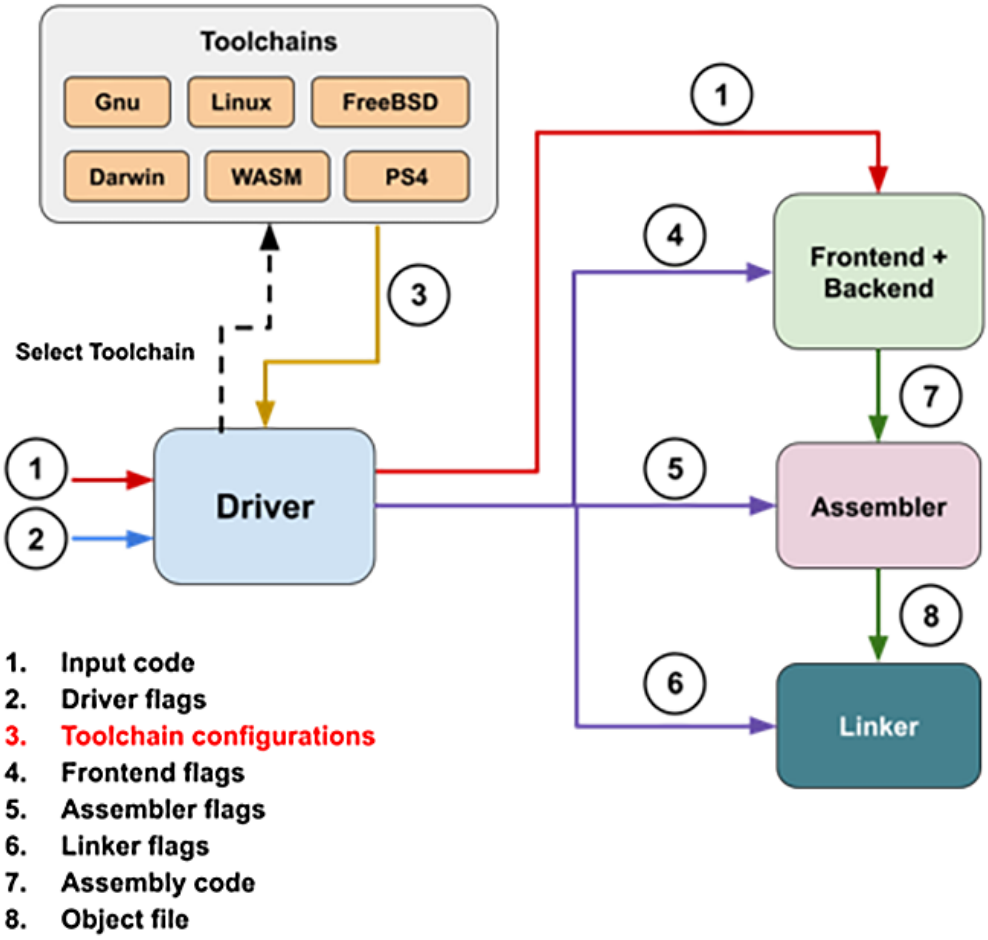

图 8.1 - Clang 的驱动程序、工具链和其他编译器的常规工作流程

如上图所示，Clang 的驱动充当了调度器，将标志和工作负载分配到每个编译阶段，即前端/后端、汇编器和链接器。为了更具体地了解每个阶段的标志是什么样子，回想一下在本章开始时介绍的`-###` 编译器选项。该选项打印的 (大量) 内容是前端的标志 (上图中为 4)，例如：在这些前端标志中，`-internal-system` 携带关于系统头文件路径的信息，包括 C/C++ 标准库头文件存储的路径。很明显，Clang 的前端需要知道标准库的头文件存储在哪里，但根据使用 `clang`(或 `gcc`) 的经验，使用者很少需要明确地告诉它们这些头文件存储在哪里——因为驱动来做这件事。同样的逻辑也适用于链接阶段。连接器通常需要不仅需要一个目标文件来正确地生成可执行文件或库，例如：需要知道 C/C++ 标准库的库文件位置 (比如在 Unix/Linux 系统上的 `*.a` 或 `*.so`)。在这种情况下，Clang 的驱动将通过链接器标志向链接器提供该信息。

提供给各个编译器阶段的标志和工作负载——简而言之，就是配置——从两个来源转换而来: 驱动标志 (上图中有 2 个) 和所选的工具链 (上图中有 3 个)。驱动标志是由用户通过命令行界面提供的——也就是编译器标志——例如：`-c`，`-Wall` 和`-std=c++11`。下一节中，我们将展示一些 Clang 如何将驱动标志转换为前端标志，甚至汇编器/链接器标志的示例。

另外，**工具链**是描述应该如何在特定平台上编译输入代码的实体。不同的硬件体系结构和操作系统 (OS)——简称平台——有自己的构建、加载和运行程序的方式。以 macOS X 和 Linux 为例，虽然它们都有一个类 unix 的环境，但当构建一个程序时，macOS X 的系统 (标准) 库总是在 Apple 的 XCode IDE 包中，而 Linux 通常将它们存储在普通的文件夹中，如`/usr/include` 和`/usr/lib`。此外，macOS X 使用一种称为 **Mach-O** 的可执行格式，与于 Linux 的 ELF 格式不同。这极大地影响了编译器 (Clang) 构建代码的方式。

Clang 要为不同的平台编译代码，使用工具链 (内部由 `ToolChain` C++ 类表示) 来封装特定于平台的信息和配置。编译的早期阶段，Clang 的驱动会根据当前运行的系统 (主机系统) 或者用户的偏好选择一个正确的工具链——可以使用`-target=` 驱动标志来要求 Clang 为不同于主机系统的特定平台构建一个程序，这是有效的**交叉编译**。然后，驱动将从选择的工具链中收集一些特定于平台的配置，然后将其与前面提到的驱动程序选项结合起来，并通过命令行标志将它们分派到各个编译器阶段。请注意，不同的平台通常使用不同的汇编器和连接器，例如：macOS X 目前只能使用 `ld64` 和 `lld` 连接器，而 Linux 可以使用 `ld`(BFD 连接器)、`ld.gold` 和 `lld` 作为连接器。因此，工具链还应该指定使用什么汇编器和链接器。本章的最后一节，添加一个自定义工具链，我们将通过一个示例项目来学习 Clang 的工具链是如何工作的。让我们开始旅程，如何在 Clang 的驱动标示是如何工作的。

8.3. 添加自定义驱动标志

前一节中，我们了解了 Clang 中驱动程序和工具链的角色。本节中，我们将了解 Clang 驱动如何通过向 Clang 添加一个自定义驱动标志来完成这种转换。同样，在单独展示详细步骤之前，我们将先对这个示例项目进行概述。

8.3.1 项目概述

本节中，我们将使用的示例项目将添加一个新的驱动标志，当用户给出该标志时，将隐式地在输入代码中包含一个头文件。

更具体地说，我们有一个头文件——`simple_log.h`——内容如下所示，定义了一些简单的 API 来打印日志消息:

```
#ifndef SIMPLE_LOG_H
#define SIMPLE_LOG_H
#include <iostream>
#include <string>
#ifdef SLG_ENABLE_DEBUG
inline void print_debug(const std::string &M) {
  std::cout << "[DEBUG] " << M << std::endl;
}
#endif

#ifdef SLG_ENABLE_ERROR
inline void print_error(const std::string &M) {
  std::cout << "[ERROR] " << M << std::endl;
}
```

```
#endif

#ifdef SLG_ENABLE_INFO
inline void print_info(const std::string &M) {
  std::cout << "[INFO] " << M << std::endl;
}
#endif

#endif
```

这里的目标是在我们的代码中使用这些 API，而不需要编写 `#include "simple_log.h"` 来导入头文件。只有当我们给 `clang` 一个自定义驱动标志`-fuse-simple-log` 时，这个特性才会启用，例如：下面的代码，`test.cc`:

```
int main() {
  print_info("Hello world!!");
  return 0;
}
```

尽管没有任何 `#include` 指令，仍然可以编译 (使用`-fussimple-log` 标志)，运行也没有任何问题:

```
$ clang++ -fuse-simple-log test.cc -o test
$ ./test
[INFO] Hello world!!
$
```

此外，可以使用`-fuse-<log level>-simple-log /-fno-use-<log level>-simple-log` 包含或排除特定日志级别的函数。例如，让我们前面的代码，在编译代码时添加`-fno-use-info-simple log`:

```
$ clang++ -fuse-simple-log -fno-use-info-simple-log test.cc -o
test
test.cc:2:3: error: use of undeclared identifier 'print_info'
  print_info("Hello World!!");
  ^
1 error generated
$
```

在 `simple_log.h` 中，每个日志打印函数的开关都由它周围的 `#ifdef` 语句控制，例如：只在定义了 `SLG_ENABLE_INFO` 时才会包含 `print_info`。之后，在翻译自定义驱动程序标志一节中，我们将展示如何通过驱动程序标志来切换这些宏定义。

最后，可以为 simple_log.h 文件指定自定义路径。默认情况下，我们的特性会在源代码的当前文件夹中包含 simple_log.h。可以通过提供-fsimple-log-path=< 文件路径 > 或-fuse-simple log=< 文件路径 > 来改变这一点，例如：想使用 simple_log.h 的另一个版本——advanced_log.h，存储在/home/user 目录下——提供具有相同接口但不同实现的函数。现在，就可以使用以下命令：

```
$ clang++ -fuse-simple-log=/home/user/advanced_log.h test.cc -o
test
[01/28/2021 20:51 PST][INFO] Hello World!!
$
```

下一节将向展示如何更改 Clang 驱动程序中的代码，以便实现这些特性。

8.3.2 声明自定义驱动标志

首先，完成声明定制驱动程序标志的步骤，例如：-fusesimple-log 和-fno-use-info-simple-log。然后，我们将把这些标志写入到实际的前端功能中。

Clang 使用 TableGen 语法来声明所有类型的编译器标志——包括驱动标志和前端标志。

TableGen

TableGen 是一种领域特定语言 (DSL)，用于声明结构和关系数据。要了解更多，请查看第 4 章。

所有这些标志声明都放在 clang/include/clang/Driver/Options.td 中，例如：以通用的-g 标志为例，会生成源级别的调试信息，声明如下：

```
def g_Flag : Flag<["-"], "g">, Group<g_Group>,
  HelpText<"Generate source-level debug information">;
```

TableGen 记录 g_Flag 是由几个 TableGen 类创建的：Flag、Group 和 HelpText。其中，我们最感兴趣的是 Flag，它的模板值 (["-"] 和"g") 描述了实际的命令行标志格式。请注意，声明布尔标志时——该标志的值由存在程度决定，没有随后的其他值——本例从 Flag 类继承。

如果想要声明一个标记，其值跟在等号 ("=") 后面，继承自 Joined 类，例如：-std=<C++ standard name> 的 TableGen 声明如下：

```
def std_EQ : Joined<["-", "--"], "std=">, Flags<[CC1Option]>,
…;
```

通常，这些类型的标记的记录名 (在本例中为 std_EQ) 有_EQ 作为后缀。

最后，Flags(复数) 类可以用来指定一些属性，例如：前面代码中的 CC1Options，就说明这个标志也可以是前端标志。

现在我们已经了解了驱动标志通常是如何声明的，是时候创建我们自己的驱动标志了：

1. 首先，处理-fuse-simple-log 标志。下面是声明：

```
def fuse_simple_log : Flag<["-"], "fuse-simple-log">,
                  Group<f_Group>, Flags<[NoXarchOption]>;
```

除了 Group 类和 NoXarchOption 之外，这段代码基本上与我们前面使用的示例没有区别。前者指定该标志所属的逻辑组，例如：f_Group 用于以-f 开头的标志。后者告诉我们这个标志只能在驱动中使，例如：不能将它传递给前端 (但是我们如何将标志直接传递给前端呢？我们将在本节的最后简短地回答这个问题)。

注意，在这里只声明-fuse-simple-log，而没有声明-fuse-simplelog=< 文件路径 >——这将在另一个标志中完成。

2. 接下来,我们将处理-fuse-<log level>-simple-log 和-fno-use-<log level>-simple-log。在 GCC 和 Clang 中，通常会使用-f<flag name>/-fno-<flag name> 来启用或禁用某个特性。因此，Clang 提供了一个方便的 TableGen 工具——BooleanFFlag——使创建成对的标记更容易。请参见下面代码中-fuse-error-simple-log/-fno-use-error-simple-log 的声明:

```
defm use_error_simple_log : BooleanFFlag<"use-errorsimple-log">,
   Group<f_Group>, Flags<[NoXarchOption]>;
```

BooleanFFlag 是一个多类 (所以确保使用的是 defm，而不是 def 来创建 TableGen 记录)。在底层中，同时为-f<flag name> 和-fno-<flag name> 创建 TableGen 记录。

现在我们已经了解了如何创建 use_error_simple_log，可以使用相同的技巧为其他日志级别创建 TableGen 记录:

```
defm use_debug_simple_log : BooleanFFlag<"use-debugsimple-log">,
   Group<f_Group>, Flags<[NoXarchOption]>;
defm use_info_simple_log : BooleanFFlag<"use-info-simplelog">,
   Group<f_Group>, Flags<[NoXarchOption]>;
```

3. 最后，我们声明-fuse-simple-log=< 文件路径 > 和-fsimple-log-path=< 文件路径 > 标志。前面的步骤中，只处理布尔标志，但这里，我们创建的标志的值跟随等号，所以我们使用了之前引入的 Joined 类:

```
def fsimple_log_path_EQ : Joined<["-"], "fsimple-logpath=">,
  Group<f_Group>, Flags<[NoXarchOption]>;
def fuse_simple_log_EQ : Joined<["-"], "fuse-simplelog=">,
  Group<f_Group>, Flags<[NoXarchOption]>;
```

同样，带值的标志通常会在其 TableGen 记录名称后缀中使用_EQ。

这就结束了声明自定义驱动程序标志的所有必要步骤。在 Clang 的构建过程中，这些 TableGen 指令将转换成 C++ 枚举和驱动程序使用的其他工具，例如：-fuse-simple-log=< 文件路径 > 将由一个枚举值表示，也就是 options::OPT_fuse_simple_log_EQ。下一节将展示如何从用户给出的所有命令行标志中查询这些标志，最重要的是，如何将自定义标志翻译成前端对应的标志。

8.3.3 翻译自定义驱动标志

编译器驱动程序在底层为用户做了很多事情，例如：根据编译目标找出正确的工具链，并翻译由用户指定的驱动标志，这就是我们接下来要做的事情。我们的例子中，当编译时给定了-fuse-simple-log，就要要包含头文件 simple_log.h，并根据-fuse-<log level>-simple-log/-fno-use-<log level>-simple-log 标志来定义 SLG_ENABLE_ERROR 这样的宏，来包含或排除某些日志打印函数。更具体地说，这些任务可以分为两个部分:

- 如果指定-fuse-simple-log，将其转换为前端标志:

```
-include "simple_log.h"
```

-include 前端标志，顾名思义，在编译源代码中隐式地包含指定的文件。
使用相同的逻辑，如果给出-fuse-simple-log=/other/file.h 或-fusessimple-log -fsimple-log-path=/other/file.h，将转换为以下内容:

```
-include "/other/file.h"
```

- 如果指定-fuse-<log level>-simple-log 或-fno-use-<log level>-simple-log，例如: -fuse-error-simple-log，将翻译成以下内容:

```
-D SLG_ENABLE_ERROR
```

-D 标志隐式地为编译源代码定义了一个宏变量。
但是，如果只指定了-fuse-simple-only，该标志将隐式地包含所有日志打印函数。换句话说，-fuse-simple-only 不仅会翻译成-include 标志，就像前面提到的，还会翻译成下面的标志:

```
-D SLG_ENABLE_ERROR -D SLG_ENABLE_DEBUG -D SLG_ENABLE_INFO
```

假设将-fuse-simple-log 和-fno-use-<log level>-simple-log 组合使用，例如:

```
-fuse-simple-log -fno-use-error-simple-log
```

将翻译成以下代码:

```
-include "simple_log.h" -D SLG_ENABLE_DEBUG -D SLG_ENABLE_INFO
```

最后，还允许以下组合:

```
-fuse-info-simple-log -fsimple-log-path="my_log.h"
```

也就是说，我们只启用了一个日志打印功能，而不使用`-fusessimple-log`，并使用一个定制的简单日志头文件。这些驱动标志将翻译成以下代码:

```
-include "my_log.h" -D SLG_ENABLE_INFO
```

前面提到的规则和标志的组合实际上可以以一种非常优雅的方式处理，尽管乍一看很复杂。我们将很快展示如何做到这一点。

既然我们已经了解了将要翻译的前端标志，那么现在是时候学习如何进行翻译了。

许多驱动标志转换发生的地方是在 `driver::tools::Clang` 类中。更具体地说，这发生在它的 `Clang::ConstructJob` 方法中，该方法位于 `clang/lib/Driver/ToolChains/Clang.cpp` 文件中。

关于 `driver::tools::Clang`

对于这个 C++ 类，最主要的问题可能是，它代表什么概念？为什么把它放在名为 *ToolChains* 的文件夹下？这是否意味着它也是一个工具链？虽然我们将在下一节中详细回答这些问题，但现在读者们可以把它看作为 Clang 前端的代表。这 (在某种程度上) 解释了为什么它负责将驱动标志翻译成前端标志。

下面是翻译定制驱动标志的步骤。下面的代码可以插入到 `Clang::ConstructJob` 方法的任何地方，在 `addDashXForInput` 函数调用前，进行包装翻译:

1. 首先，定义了一个助手类——`dSimpleLogOpts`——来存储我们的自定义标志信息:

```
struct SimpleLogOpts {
  // If a certain log level is enabled
  bool Error = false,
       Info = false,
       Debug = false;
  static inline SimpleLogOpts All() {
    return {true, true, true};
  }
  // If any of the log level is enabled
  inline operator bool() const {
    return Error || Info || Debug;
  }
};
```

```
// The object we are going to work on later
SimpleLogOpts SLG;
```

SimpleLogOpts 中的 bool 字段——Error，Info 和 Debug——表示自定义标志启用的日志级别。我们还定义了一个辅助函数 SimpleLogOpts::All() 来创建一个启用所有日志级别的 SimpleLogOpts，以及一个 bool 类型的转换操作符，这样就可以使用更清晰的语法 (如图所示) 来了解启用了哪个级别日志了:

```
if (SLG) {
  // At least one log level is enabled!
}
```

2. 我们先处理最简单的-fuse-simple-log 标志。在这一步中，只有在看到-fuse-simple-log 标志时，我们才会打开 SLG 中的所有的日志级别。
   在 Clang::ConstructJob 方法中，用户给出的驱动标志存储在 Args 变量中 (ConstructJob 的参数之一)，是 ArgList 类型。有很多方法可以查询 Args，但因为我们只关心-fuse-simple-log 的存在，所以使用 hasArg 是最合适的方式:

```
if (Args.hasArg(options::OPT_fuse_simple_log)) {
  SLG = SimpleLogOpts::All();
}
```

   我们在前面的代码中通过 TableGen 语法声明的每个标志都将在 options 名称空间下由唯一的枚举表示。此例中，枚举值为 OPT_fuse_simple_log。当声明该标志时，枚举值的名称通常以 OPT_开头，后面跟着 TableGen 记录名称 (也就是说，名称跟在 def 或 defm 后面)。如果给定的标志标识符出现在输入驱动标志中，ArgList::hasArg 函数将返回 true。
   除了-fuse-simple-log 外，当指定-fuse-simple-log=< 文件路径 > 时，尽管我们将只处理后面的文件路径，但还需要打开所有的日志级别。因此，我们将把前面的代码改为:

```
if (Args.hasArg(options::OPT_fuse_simple_log,
                options::OPT_fuse_simple_log_EQ)) {
  SLG = SimpleLogOpts::All();
}
```

   ArgList::hasArg 实际上可以接受多个标志标识符，如果其中任何一个出现在输入驱动标志中，则返回 true。同样，-fuse-simplelog=<···> 标志由 OPT_fuse_simple_log_EQ 表示，因为其 TableGen 记录名称是 fuse_simple_log_EQ。
3. 接下来，我们将处理-fuse-<log level>-simple-log/-fno-use-<log level>-simple-log。以错误级别为例 (其他级别的标志也以同样的方式使用，就不在这里演示)，我们使用了 ArgList::hasFlag 函数:

```
SLG.Error = Args.hasFlag(options::OPT_fuse_error_simple_
log, options::OPT_fno_use_error_simple_log, SLG.Error);
```

   如果第一个参数 (是 OPT_fuse_error_simple_log) 或第二个参数 (是 OPT_fno_use_error_simple_log) 所代表的标志分别出现在输入驱动标志中，hasFlag 函数将返回 true 或 false。如果这两个标志都不存在，hasFlag 将返回由第三个参数 (本例中为 SLG.Error)。
   使用这种机制，我们已经实现了一些 (复杂的) 规则和标志组合，在本节前面提到过:

a) `-fno-use-<log level>-simple-log` 标志可以在出现`-fuse-simple-log` 时禁用某些日志打印功能。`-fuse-simple-log` 包含了所有日志打印功能。

b) 即使没有`-fuse-simple-log`，我们仍然可以通过使用`-fuse-<log level>-simple-log` 标志来启用单个日志打印功能。

4. 目前，我们只是在使用 `SimpleLogOpts` 的数据结构。从下一步开始，我们将根据到目前为止构建的 `SimpleLogOpts` 实例生成前端标志。在这里生成的第一个前端标志是`-include < 文件路径 >`。首先，启用了日志级别才有意义。因此，我们将通过检查 `SLG` 将`-include` 的生成封装在 `if` 语句中:

```
if (SLG) {
  CmdArgs.push_back("-include");
  ...
}
```

`CmdArgs`(`Clang::ConstructJob` 中的一个局部变量——具有类向量类型) 是我们放置前端标志的地方。

注意，不能推送包含任何空格的前端标志。例如，不能这样做:

```
if (SLG) {
  CmdArgs.push_back("-include simple_log.h"); // Error
  ...
}
```

这是因为这个 vector(`CmdArgs`) 将视为 `argv`，这可以在 C/C++ 的主函数中看到，并且当这些参数实现时，单个参数中的任何空格都将创建失败。

相反，我们将把路径单独放到一个简单的日志头文件中，如下所示:

```
if (SLG) {
  CmdArgs.push_back("-include");
  if (Arg *A = Args.getLastArg(options::OPT_fuse_simple_
  log_EQ, options::OPT_fsimple_log_path_EQ))
    CmdArgs.push_back(A->getValue());
  else
    CmdArgs.push_back("simple_log.h");
  ...
}
```

`ArgList::getLastArg` 函数将检索该值 (如果多次出现同一个标志，则为最后一个值)，跟随给定的标志，如果这些标志都不存在，则返回 null。例如，在本例中标志是`-fuse-simple-log=`(第二个参数中的`-fsimple-log-path=` 就只是第一个参数的别名标志)。

5. 最后，我们将生成前端标志来控制应该启用哪些日志打印功能。同样，我们在这里只显示其中一个日志级别的代码，因为其他级别也用相同的方法实现:

```
if (SLG) {
  ...
  if (SLG.Error) {
    CmdArgs.push_back("-D");
    CmdArgs.push_back("SLG_ENABLE_ERROR");
```

```
  }
  ...
}
```

这些基本上是我们项目所需的所有修改。在继续之前，我们要做的最后一件事是验证我们的工作。回想一下**-###** 命令行标志，它用于打印传递给前端的所有标志。我们在这里可以使用它来查看我们的自定义驱动标志是否能正确翻译。

首先，试试下面的命令:

```
$ clang++ -### -fuse-simple-log -c test.cc
```

输出应该包含以下字符串:

```
"-include" "simple_log.h" "-D" "SLG_ENABLE_ERROR" "-D" "SLG_ENABLE_INFO"
"-D" "SLG_ENABLE_DEBUG"
```

再试试下面的命令:

```
$ clang++ -### -fuse-simple-log=my_log.h -fno-use-error-simplelog -c test.cc
```

输出应该包含以下字符串:

```
"-include" "my_log.h" "-D" "SLG_ENABLE_INFO" "-D" "SLG_ENABLE_DEBUG"
```

最后，在来试试这个命令：

```
$ clang++ -### -fuse-info-simple-log -fsimple-log-path=my_log.h
-c test.cc
```

输出应该包含以下字符串:

```
"-include" "my_log.h" "-D" "SLG_ENABLE_INFO"
```

本节的最后一小节中，我们将讨论一些将标志传递给前端的其他方法。

8.3.4 向前端传递标志

在前面的内容中，我们展示了驱动标志和前端标志之间的区别，它们之间的关系，以及 Clang 的驱动如何将前者转换为后者。此时，读者们可能想知道，我们是否可以跳过驱动，直接将标志传递到前端？哪些标志可以这样传递呢？

对第一个问题的回答是肯定的，实际上已经在前几章做过好几次了。回想一下，在第 7 章中，我们开发了一个插件——更确切地说，是一个 AST 插件。我们使用命令行参数在 Clang 中加载和运行了这个插件:

```
$ clang++ -fplugin=MyPlugin.so \
          -Xclang -plugin -Xclang ternary-converter \
          -fsyntax-only test.cc
```

我们需要在`-plugin` 和三元转换器参数之前加上`-Xclang` 标志。答案很简单: 这是因为`-plugin` (及其值三元转换器) 是一个仅面向前端的标志。

要将一个标志直接传递到前端，可以将`-Xclang` 放在它前面。但使用`-Xclang` 会有一个警告: 一个`-Xclang` 只会将一个后续的命令行参数 (不带任何空格的字符串) 传递给前端。换句话说，不能像这样重写之前的插件加载例子:

```
# Error: `ternary-converter` will not be recognized
$ clang++ -fplugin=MyPlugin.so \
          -Xclang -plugin ternary-converter \
          -fsyntax-only test.cc
```

这是因为`-Xclang` 只会将`-plugin` 转移到前端，而将三元转换器留在后面，在这种情况下，Clang 将不知道运行哪个插件。

另一种直接将标志传递给前端的方法是使用`-cc1`。当我们使用`-###` 来打印前几节中驱动翻译过的前端标志时，在这些前端标志中，跟随 `clang` 可执行文件路径的第一个始终是`-cc1`。这个标志可以收集所有命令行参数，并将它们发送到前端。尽管这看起来很方便——没有必要再用`-Xclang` 给每个要传递到前端的标志做前缀了——但要注意，不能在该标志列表中混合任何仅供驱动使用的标志，例如：本节前，当我们在 TableGen 语法中声明`-fuse-simple-log` 标志时，用 `NoXarchOption` 对该标志进行了注释，声明它只能由驱动使用。在这种情况下，`-cc1` 后面就不能再出现`-fuse-simple-log` 了。

这就引出了我们的最后一个问题: 哪些标志可以在驱动或前端程序使用，哪些标志二者都可以使用？答案可以通过刚刚提到的 `NoXarchOption` 得到。当在 TableGen 语法中为驱动或前端声明标志时，可以使用 `Flags<···>` TableGen 类，及其模板参数来强制一些约束。例如，使用以下指令，可以防止驱动使用`-foo` 标志:

```
def foo : Flag<["-"], "foo">, Flags<[NoDriverOption]>;
```

除了 `NoXarchOption` 和 `NoDriverOption` 外，这里还有一些可以在 `Flags<···>` 中使用的常见注释:

- `CoreOption`: 声明该标志可以由 `clang` 和 `clang-cl` 共享。`clong-cl` 是一个有趣的驱动，它与 MSVC (Microsoft Visual Studio 使用的编译器框架) 使用的命令行界面 (包括命令行参数) 兼容。
- `CC1Option`: 声明该标志可以传递给前端。但它并不是说这是一个只能在前端使用的标志。
- `Ignored`: 声明该标志将会让 Clang 的驱动忽略 (但继续编译过程)。GCC 有许多在 Clang 中不受支持的标志 (过时或不适用)。然而，Clang 实际上试图识别这些标志，但除了显示缺少实现的警告消息外，什么也不做。这背后的基本原因是，我们希望 Clang 可以替代 GCC，而不需要在许多项目中修改现有的构建脚本 (如果没有这个兼容层，Clang 会在看到未知标志时终止编译)。

本节中，我们了解了如何为 Clang 的驱动程序添加自定义标志，并实现将其转换为前端标志的逻辑。当想要以一种更直接和干净的方式切换自定义功能时，这种技能非常有用。

下一节中，我们将通过创建自定义工具链来了解工具链扮演的角色，以及其是如何在 Clang 中工作的。

8.4. 添加自定义工具链

上一节中，我们了解了如何在 Clang 中为驱动添加自定义标志，并了解了驱动如何将它们转换为前端接受的标志。本节中，我们将讨论工具链——驱动内部的一个重要模块，它能帮助驱动程序适应不同的平台。

记得在本章的第一部分，在图 8.1 中展示了驱动和工具链之间的关系: 驱动程序可以根据目标平台选择合适的工具链，然后利用其基本信息做以下事情:

1. LExecute 生成目标代码所需的正确的汇编器、链接器或任何工具。
2. 向编译器、汇编器或链接器传递特定于平台的标志。

这些信息对于构建源代码至关重要，因为每个平台可能都有自己独特的特征，比如：系统库路径和受支持的汇编/链接器变体。没有它们，甚至无法生成正确的可执行文件或库。

本节希望帮助大家了解如何在将来为定制平台创建 Clang 工具链。Clang 中的工具链框架足够强大，可以适应各种各样的用例，例如：可以创建一个类似于 Linux 上传统编译器的工具链——包括使用 GNU AS 进行组装和使用 GNU LD 进行链接——而不需要对默认库路径或编译器标志进行多次定制。另一方面，可以使用一个怪异的工具链，它甚至不使用 Clang 来编译源代码，而是使用一个专有的汇编器和带有不常见命令行标志的链接器。本节将尝试使用一个示例来演示最常见的用例，同时又能展示该框架的灵活性。

这一部分的组织如下: 将从我们将要从事的项目的概述开始。在此之后，我们将把项目工作负载分解为三个部分——添加定制编译器选项、设置定制汇编器和设置定制链接器——然后将它们放在一起，结束本节。

系统需求

作为另一个友好的提示，下面的项目只能在 Linux 系统上工作。请确保已安装 OpenSSL。

8.4.1 项目概述

我们将创建一个名为 **Zipline** 的工具链，它使用 Clang(前端和后端) 来进行普通的编译，但在汇编阶段使用 **Base64** 编码生成的汇编代码，并在链接阶段将这些 Base64 编码的文件打包成 **ZIP 文件** (或`.tarbell` 文件)。

Base64

Base64 是一种编码方案，通常用于将二进制文件转换为纯文本。它可以很容易地在不支持二进制格式的上下文中传输 (例如，HTTP 报头)。还可以将 Base64 应用于普通文本文件，就像我们的例子一样。

这个工具链在生产环境中基本上是无用的。它只是一个演示，模拟开发人员在为定制平台创建新工具链时可能遇到的常见情况。

这个工具链是通过自定义驱动标志`-ziplin`/`--zipline` 来启用。当提供该标志时，编译器会隐式地将 `my_include` 文件夹添加到你的主目录中，作为搜索头文件路径，例如：上一节中，添加自定义驱动标志，我们的自定义`-fuse-simple-log` 标志将隐式地在输入源代码中包含一个头文件 `simple_log.h`:

```
$ ls
main.cc simple_log.h
$ clang++ -fuse-simple-log -fsyntax-only main.cc
$ # OK
```

然而，如果 `simple_log.h` 不在当前目录中，就像前面的代码片段中那样，我们需要通过另一个标志指定它的路径:

```
$ ls .
# No simple_log.h in current folder
main.cc
$ clang++ -fuse-simple-log=/path/to/simple_log.h -fsyntax-only
main.cc
$ # OK
```

在 Zipline 的帮助下，可以把 `simple_log.h` 放在`/home/<user name>/my_include` 中，这样编译器就能找到它了:

```
$ ls .
# No simple_log.h in current folder
main.cc
$ ls ~/my_include
simple_log.h
$ clang++ -zipline -fuse-simple-log -fsyntax-only main.cc
$ # OK
```

Zipline 的第二个特性是，`clang` 可执行文件将源代码编译成由 Base64 在`-c` 标志下编码的汇编代码，该汇编代码将汇编文件 (来自编译器) 编译成一个目标文件。下面是命令示例:

```
$ clang -zipline -c test.c
$ file test.o
test.o: ASCII text # Not (binary) object file anymore
$ cat test.o
CS5OZXh0CgkuZmlsZQkidGVzdC5jYyIKCS
5nbG9ibAlfWjNmb29pCgkucDJhbGln

bgk0LCAweDkwCgkudHlwZQlfWjNmb29p
LEBmdW5jdGlvbgpfWjNmb29pOgoJLmNm

⋯ # Base64 encoded contents
$
```

前面的 `file` 命令显示生成的文件 `test.o`，从之前使用 `clang` 开始，就不再是二进制格式的对象文件。这个文件的内容就是编译器后端生成的汇编代码 (Base64 编码) 版本。

最后，Zipline 用一个定制阶段替换了原来的链接阶段，这个定制阶段将前面提到的 Base64 编码的程序集文件打包并压缩到一个`.zip` 文件中。下面是一个例子:

```
$ clang -zipline test.c -o test.zip
$ file test.zip
test.zip: Zip archive, at least v2.0 to extract
$
```

如果解压 `test.zip`，将发现这些解压的文件是 Case64 编码的程序集文件。

或者，我们可以使用 Linux 的 `tar` 和 `gzip` 工具将它们打包并压缩到 Zipline 中:

```
$ clang -zipline -fuse-ld=tar test.c -o test.tar.gz
$ file test.tar.gz
test.tar.gz: gzip compressed data, from Unix, original size…
$
```

通过使用现有的-fuse-ld=< 链接器名称 > 标志，我们可以在自定义链接阶段选择使用 zip、tar 或 gzip。

下一节中，我们将为这个工具链创建框架代码，并展示如何向头文件搜索路径添加额外的文件夹路径。

8.4.2 创建工具链并添加自定义的包含路径

本节中，我们将为 Zipline 工具链创建框架，并展示如何在 Zipline 的编译阶段添加一个额外的包含文件夹路径——更确切地说，一个额外的**系统包含路径**。具体步骤如下:

1. 在添加工具链实现之前，不要忘记使用自定义驱动标志-zipline/--zipline 来启用工具链。使用我们在前一节学到的技能，添加自定义驱动标志来完成。在 clang/include/clang/Driver/Options.td 里面，我们将增加以下行:

```
// zipline toolchain
def zipline : Flag<["-", "--"], "zipline">,
Flags<[NoXarchOption]>;
```

   同样，Flag 说明这是一个布尔标志，而 NoXarchOption 说明这个标志是驱动唯一的。我们将很快使用这个驱动标志。

2. Clang 中的工具链由 clang::driver::ToolChain 类表示。Clang 支持的每个工具链都是从它派生出来的，源文件在 clang/lib/Driver/ToolChains 文件夹下。我们将在那里创建两个新文件:Zipline.h 和 Zipline.cpp。

3. 对于 Zipline.h，先添加以下框架代码:

```
namespace clang {
namespace driver {
namespace toolchains {
struct LLVM_LIBRARY_VISIBILITY ZiplineToolChain
: public Generic_ELF {
  ZiplineToolChain(const Driver &D, const llvm::Triple
  &Triple, const llvm::opt::ArgList &Args)
  : Generic_ELF(D, Triple, Args) {}
  ~ZiplineToolChain() override {}
  // Disable the integrated assembler
  bool IsIntegratedAssemblerDefault() const override
  { return false; }
  bool useIntegratedAs() const override { return false; }
  void
  AddClangSystemIncludeArgs(const llvm::opt::ArgList
```

```
  &DriverArgs, llvm::opt::ArgStringList &CC1Args)
  const override;
protected:
  Tool *buildAssembler() const override;
  Tool *buildLinker() const override;
};
} // end namespace toolchains
} // end namespace driver
} // end namespace clang
```

我们在这里创建的 ZiplineToolChain 类，是从 Generic_ELF 派生而来，Generic_ELF 是 ToolChain 的子类，专门用于使用 ELF 作为其执行格式的系统——包括 Linux。除了父类之外，还有三个重要的方法，我们将在本节或后面的章节中实现:AddClangSystemIncludeArgs、buildAssembler 和 buildLinker。

4. buildAssembler 和 buildLinker 方法生成的 Tool 实例分别表示要在汇编和链接阶段运行的命令或程序 (我们将在下面几节中介绍它们)。现在,我们将实现 AddClangSystemIncludeArgs 方法。在 Zipline.cpp 中，我们将添加相应实现:

```
void ZiplineToolChain::AddClangSystemIncludeArgs(
                        const ArgList &DriverArgs,
                        ArgStringList &CC1Args) const {
  using namespace llvm;
  SmallString<16> CustomIncludePath;
  sys::fs::expand_tilde("~/my_include",
                         CustomIncludePath);
  addSystemInclude(DriverArgs,
                   CC1Args, CustomIncludePath.c_str());
}
```

我们在这里做的唯一一件事是使用 addSystemInclude 函数与位于主目录中的 my_include 文件夹的路径。由于每个用户的主目录不同，我们使用 sys::fs::expand_tilde 助手函数将 /my_include(其中 表示 Linux 和 Unix 系统中的主目录) 扩展到绝对路径中。另一方面，addSystemInclude 函数可以将“-internalisystem”“/path/to/my_include”标志添加到前端标志的列表中。-internal-issystem 标志用于指定系统头文件的文件夹，包括标准库头文件和平台特定的头文件。

5. 最后，当 Zipline 工具链看到新创建的-zipline/--zipline 驱动标志时，需要我们教会驱动使用 Zipline 工具链。我们需要修改 clang/lib/Driver/Driver.cpp 中的 Driver::getToolChain 方法。Driver::getToolChain 方法包含一个巨大的开关盒，用于根据目标操作系统和硬件架构选择不同的工具链 (具体的情况，请浏览处理 Linux 系统的代码)。我们将在这里添加一个额外的分支条件:

```
const ToolChain
&Driver::getToolChain(const ArgList &Args,
const llvm::Triple &Target) const {
  ...
  switch (Target.getOS()) {
    case llvm::Triple::Linux:
```

```
    ...
      else if (Args.hasArg(options::OPT_zipline))
        TC = std::make_unique<toolchains::ZiplineToolChain>
        (*this, Target, Args);
    ...
      break;
    case ...
    case ...
  }
}
```

额外的 `else-if` 语句基本上是说，如果目标操作系统是 Linux，那么当指定`-zipline`/`--zipline` 时，我们将使用 Zipline。

这样，您就添加了 Zipline 的框架，并成功地告诉驱动程序在给定自定义驱动程序标志时使用 Zipline。除此之外，还了解了如何向头文件搜索路径添加额外的系统库文件夹。

下一节中，我们将创建一个自定义汇编阶段，并将其连接到我们创建的工具链中。

8.4.3 创建自定义汇编阶段

正如在项目概述中提到的，我们不是在 Zipline 的汇编阶段将汇编代码转换为目标文件，而是调用一个程序来将 Clang 生成的汇编文件转换为 Base64 编码的对应版本。在深入其实现之前，先了解一下工具链中的每个阶段是如何表示的。

上一节中，我们了解到 Clang 中的工具链是由 `Toolchain` 类表示。每个工具链实例负责告知驱动在每个编译阶段运行什么工具——即编译、汇编和链接。这个信息封装在 `clang::driver::Tool` 类型中。上一节中 `buildAssembler` 和 `buildLinker`，它们返回工具类型的对象，分别描述要执行的动作和在汇编和链接阶段运行的工具。本节中，将展示如何实现 `buildAssembler` 返回的 `Tool` 对象。就让我们开始吧！

1. 回到 `Zipline.h`。我们在 `clang::driver::tools::zipline` 名称空间中添加了一个类 `Assembler`:

```
namespace clang {
namespace driver {
namespace tools {
namespace zipline {
  struct LLVM_LIBRARY_VISIBILITY Assembler : public Tool {
    Assembler(const ToolChain &TC)
      : Tool("zipeline::toBase64", "toBase64", TC) {}
    bool hasIntegratedCPP() const override { return false;
    }
  void ConstructJob(Compilation &C, const JobAction &JA,
  const InputInfo &Output,
  const InputInfoList &Inputs,
  const llvm::opt::ArgList &TCArgs,
  const char *LinkingOutput) const
  override;
};
} // end namespace zipline
```

```
} // end namespace tools

namespace toolchains {
struct LLVM_LIBRARY_VISIBILITY ZiplineToolChain ⋯ {
⋯
};
} // end namespace toolchains
} // end namespace driver
} // end namespace clang
```

这里需要注意，因为新创建的 `Assembler` 在 `clang::driver::tools::zipline` 名称空间中，而我们在上一节创建的 `ZiplineToolChain` 位于 `clang::driver::toochains` 中。

我们将把调用 Base64 编码工具的逻辑放在 `Assembler::ConstructJob` 方法中。

2. 我们将在 `Zipline.cpp` 中，实现 `Assembler::ConstructJob`:

```
void
tools::zipline::Assembler::ConstructJob(Compilation &C,
               const JobAction &JA,
               const InputInfo &Output,
               const InputInfoList &Inputs,
               const ArgList &Args,
               const char *LinkingOutput)
               const {
  ArgStringList CmdArgs;
  const InputInfo &II = Inputs[0];

  std::string Exec =
    Args.MakeArgString(getToolChain().
      GetProgramPath("openssl"));

  // opeenssl base64 arguments
  CmdArgs.push_back("base64");
  CmdArgs.push_back("-in");
  CmdArgs.push_back(II.getFilename());
  CmdArgs.push_back("-out");
  CmdArgs.push_back(Output.getFilename());

  C.addCommand(
    std::make_unique<Command>(
         JA, *this, ResponseFileSupport::None(),
           Args.MakeArgString(Exec), CmdArgs,
             Inputs, Output));
}
```

我们使用 OpenSSL 来做 Base64 编码，运行的命令如下:

```
$ openssl base64 -in <input file> -out <output file>
```

ConstructJob 方法的工作是构建程序调用来运行前面的命令，是由 C.addCommand(…) 函数调用 ConstructJob 实现的。传递给 addCommand 要调用的 Command 实例，表示在汇编阶段要运行的具体命令。它包含必要的信息，比如：可执行文件的路径 (Exec 变量)，及其参数 (CmdArgs 变量)。

对于 Exec 变量，工具链提供了一个工具，即 GetProgramPath 函数，用于解析可执行文件的绝对路径。

另一方面，我们为 openssl(CmdArgs 变量) 构建参数的方式，类似于在添加自定义驱动标志时所做的事情: 将驱动标志 (Args 参数) 和输入/输出文件信息 (Output 和 Inputs 参数) 转换为一组新的命令行参数，并存储在 CmdArgs 中。

3. 最后，通过实现 ZiplineToolChain::buildAssembler，将这个 Assembler 类与 Zipline ToolChain 连接起来:

```
Tool *ZiplineToolChain::buildAssembler() const {
  return new tools::zipline::Assembler(*this);
}
```

这些是我们需要完成的步骤，该工具实例中需要在 Zipline 工具链链接阶段运行的命令。

8.4.4 创建自定义链接阶段

现在我们已经完成了汇编阶段，是时候进入下一个阶段了——连接阶段。我们将使用与前一节相同的方法，创建一个自定义的工具类来表示链接器。

1. 在 Zipline.h 中，创建一个从 Tool 派生的 Linker 类:

```
namespace zipline {
struct LLVM_LIBRARY_VISIBILITY Assembler : public Tool {
  …
};

struct LLVM_LIBRARY_VISIBILITY Linker : public Tool {
  Linker(const ToolChain &TC)
    : Tool("zipeline::zipper", "zipper", TC) {}

  bool hasIntegratedCPP() const override { return false;
}

  bool isLinkJob() const override { return true; }

  void ConstructJob(Compilation &C, const JobAction &JA,
            const InputInfo &Output,
            const InputInfoList &Inputs,
            const llvm::opt::ArgList &TCArgs,
            const char *LinkingOutput) const
            override;
private:
  void buildZipArgs(const JobAction&, const InputInfo&,
            const InputInfoList&,
```

```
            const llvm::opt::ArgList&,
            llvm::opt::ArgStringList&) const;

  void buildTarArgs(const JobAction&,
            const InputInfo&,
            const InputInfoList&,
            const llvm::opt::ArgList&,
            llvm::opt::ArgStringList&) const;
};
} // end namespace zipline
```

在这个 Linker 类中，还需要实现 ConstructJob 方法，从而来告诉驱动在链接阶段要执行什么。与 Assembler 不同的是，由于需要同时支持 zip 和 tar + gzip 的打包/压缩方案，我们将添加两个额外的方法 buildZipArgs 和 buildTarArgs，来处理每个参数的构建。

2. 我们在 Zipline.cpp 中，将首先关注 Linker::ConstructJob 的实现:

```
void
tools::zipline::Linker::ConstructJob(Compilation &C,
const JobAction &JA,
const InputInfo &Output,
const InputInfoList &Inputs,
const ArgList &Args,
const char *LinkingOutput) const
{
  ArgStringList CmdArgs;
  std::string Compressor = "zip";
  if (Arg *A = Args.getLastArg(options::OPT_fuse_ld_EQ))
    Compressor = A->getValue();
  std::string Exec = Args.MakeArgString(
       getToolChain().GetProgramPath(Compressor.c_str()));

  if (Compressor == "zip")
      buildZipArgs(JA, Output, Inputs, Args, CmdArgs);
  if (Compressor == "tar" || Compressor == "gzip")
    buildTarArgs(JA, Output, Inputs, Args, CmdArgs);
  else
    llvm_unreachable("Unsupported compressor name");

  C.addCommand(
    std::make_unique<Command>(
      JA, *this, ResponseFileSupport::None(),
      Args.MakeArgString(Exec),
      CmdArgs, Inputs, Output));
}
```

在这个自定义链接阶段，我们希望使用 zip 或 tar 命令 (取决于用户指定的-fuse-ld 标志) 来打包自定义 Assembler 生成的 (Base64 编码的) 文件。

稍后将解释 zip 和 tar 的命令格式。前面的代码中,可以看到这里做的事情与 Assembler::ConstructJob 类似。Exec 变量携带 zip 或 tar 可执行文件的绝对路径。CmdArgs 变量，由 buildZipArgs

或 buildTarArgs 填充，稍后将对此进行解释，它保存了工具的命令行参数 (zip 或 tar)。与 Assembler::ConstructJob 的最大区别是，要执行的命令可以由用户提供的-fuse-ld 指定。因此，我们使用在添加自定义驱动标志部分中学到的技能，来读取驱动程序标志，并设置执行命令。

3. 如果用户决定将文件打包为 ZIP 文件 (这是默认方案，或者可以通过-fuse-ld=zip 显式指定)，将运行以下命令:

```
$ zip <output zip file> <input file 1> <input file 2>···
```

因此，我们将构建 Linker::buildZipArgs 方法，它为前面的命令构造一个参数列表，如下所示:

```
void
tools::zipline::Linker::buildZipArgs(const JobAction &JA,
                     const InputInfo &Output,
                     const InputInfoList &Inputs,
                     const ArgList &Args,
                     ArgStringList &CmdArgs) const {
  // output file
  CmdArgs.push_back(Output.getFilename());
  // input files
  AddLinkerInputs(getToolChain(), Inputs, Args, CmdArgs,
  JA);
}
```

buildZipArgs 的 CmdArgs 参数将是输出结果的地方。因为链接器可能一次接受多个输入，我们仍然使用相同的方法来获取输出文件名 (通过 Output.getfilename())。这里使用另一个辅助函数 AddLinkerInputs，将所有的输入文件名添加到 CmdArgs 中。

4. 如果用户决定使用 tar + gzip 的打包方案 (使用-fuseld=tar 或-fuse-ld=gzip)，我们将运行以下命令:

```
$ tar -czf <output tar.gz file> <input file 1> <input file 2>···
```

因此，我们将构建 Linker::buildTarArgs 方法，它为前面的命令构造一个参数列表，如下所示:

```
void
tools::zipline::Linker::buildTarArgs(const JobAction &JA,
                     const InputInfo &Output,
                     const InputInfoList &Inputs,
                     const ArgList &Args,
                     ArgStringList &CmdArgs)
                     const {
  // arguments and output file
```

```
  CmdArgs.push_back("-czf");
  CmdArgs.push_back(Output.getFilename());
  // input files
  AddLinkerInputs(getToolChain(), Inputs, Args, CmdArgs,
   JA);
}
```

与 buildZipArgs 一样,我们通过 Output. getFilename() 获取输出文件名,并使用 AddLinkerInput 将所有的输入文件名添加到 CmdArgs 中。

5. 最后，连接到 ZiplineToolChain:

```
Tool *ZiplineToolChain::buildLinker() const {
  return new tools::zipline::Linker(*this);
}
```

这就是我们为 Zipline 工具链，实现自定义链接阶段的所有步骤。

既然已经为 Zipline 工具链创建了必要的组件，那么当用户选择这个工具链时，就可以执行我们的定制特性了——对源文件进行编码并将它们打包成一个存档文件。下一节中，我们将了解如何验证这些功能。

8.4.5 验证自定义工具链

为了测试本章中实现的功能，我们可以运行项目概述中描述的示例命令，或者可以再次使用-### 驱动标志转储所有预期的编译器、汇编器和链接器的命令细节。

目前为止，我们已经知道-### 标志将显示所有被驱动程序翻译的前端标志。实际上，它还会显示计划运行的汇编器和链接器命令。我们使用以下命令:

```
$ clang -### -zipline -c test.c
```

因为-c 标志试图在 Clang 生成的汇编文件上运行汇编程序，所以在 Zipline 中的自定义汇编器 (即 Base64 编码器) 将触发。因此，将看到类似如下的输出:

```
$ clang -### -zipline -c test.c
"/path/to/clang" "-cc1" ···
"/usr/bin/openssl" "base64" "-in" "/tmp/test_ae4f5b.s" "-out"
"test.o"
$
```

以/path/to/clang -cc1 开头的行包含了在前面遇到的前端标志。下面一行是汇编器调用的命令，在我们的例子中是运行 openssl 来执行 Base64 编码。

奇怪的/tmp/test_ae4f5b.S 文件名是由驱动创建的临时文件，用于存放编译器生成的汇编代码。

使用相同的技巧，可以验证我们的定制链接器阶段，如下所示:

```
$ clang -### -zipline test.c -o test.zip
"/path/to/clang" "-cc1" ⋯
"/usr/bin/openssl" "base64" "-in" "/tmp/test_ae4f5b.s" "-out"
"/tmp/test_ae4f5b.o"
"/usr/bin/zip" "test.zip" "/tmp/test_ae4f5b.o"
$
```

由于`-o` 标志在前面的命令中使用，Clang 将由 `test.c` 构建一个完整的可执行文件，包括汇编器和链接器。因此，由于 `zip` 命令是从上一个汇编阶段获取的结果 (`/tmp/test_ae4f5b.o` 文件)。所以，这里可以添加加`-fuse-ld=tar` 标志，从而使用 `zip` 对 `tar` 命令进行替换。

本节中，展示了如何为 Clang 的驱动程序创建一个工具链。这是在定制或新平台上支持 Clang 的关键技能。还了解了 Clang 的工具链框架非常灵活，可以处理目标平台的各种任务。

8.5. 总结

本章中，我们首先介绍了 Clang 的驱动和工具链扮演的角色——提供平台特定信息的模块，如受支持的汇编器和链接器——用来辅助 Clang。然后，展示了定制驱动的一种常见的方法——添加一个新的驱动标志。之后，讨论了工具链，和如何创建一个自定义的工具链。当要在 Clang(甚至 LLVM) 中创建一个新特性，并且需要一个自定义的编译器标志来启用时，这些技能都非常有用。此外，开发自定义工具链的能力对于在新操作系统，甚至在新硬件架构对 Clang 的支持都是至关重要的。

这是这本书第二部分的最后一章。从下一章开始，我们将讨论 LLVM 的中端——平台无关的程序分析和优化框架。

8.6. 练习

1. 由于不同的平台倾向于支持不同的汇编器和连接器，所以通常会覆盖汇编和连接阶段。然而，是否可以覆盖编译阶段 (在 Clang 中)？如果有可能，应该怎么做呢？为什么希望这样做呢？
2. 当使用 `tools::zipline::Linker::ConstructJob` 时，如果用户通过`-fuse-ld` 标志提供了不支持的压缩器名称，只需使用 `llvm_unreachable` 来清理编译过程。我们是否可以用 Clang 的诊断框架 (第 7 章) 来替换，以打印出更合适的消息？
3. 就像使用`-Xclang` 将标志直接传递给前端一样，我们也可以使用驱动标志 (例如：`-Wa`(用于汇编器) 或`-Wl`(用于链接器)) 将特定于汇编器或链接器的标志，直接传递给汇编器或链接器。那我们如何在 Zipline 的自定义汇编器和链接器阶段使用这些标志？

第三部分：LLVM 的中端开发

与目标无关的转换和分析，也称为 **LLVM** 的“中端”，是整个框架的核心。尽管可以在网上找到讲解这一部分的资料，但仍然有许多隐藏的特性可以改善实际开发的体验，还会有许多陷阱会让开发陷入困境。在本节中，我们将为了解 **LLVM** 中端开发之旅做准备。

本节包括以下几章:

- 第 **9** 章，使用 **PassManager** 和 **AnalysisManager**
- 第 **10** 章，处理 **LLVM IR**
- 第 **11** 章，准备相关的工具
- 第 **12** 章，学习 **LLVM IR** 表达式

第 9 章 使用 PassManager 和 AnalysisManager

在本书的前一部分，前端开发中，我们首先介绍了 Clang 的内部原理，Clang 是 LLVM 为 C 族编程语言设计的官方前端。我们讨论了各种示例，包括技能和知识，它们可以处理与源代码紧密耦合的各种问题。

本书的这一部分，我们将使用 **LLVM IR**——用于编译器优化和代码生成 (目标无关) 的**中间表示 (IR)**。与 Clang 的**抽象语法树 (AST)** 相比，LLVM IR 通过封装相应的执行细节来实现更强大的程序分析和转换，从而提供了不同级别的抽象。除了设计 LLVM IR 之外，围绕这种 IR 格式还有一个成熟的生态系统，提供了无数的资源，比如库、工具和算法实现。我们将讨论有关 LLVM IR 的各种主题，包括最常见的 LLVM Pass 开发，使用和编写程序分析器，以及使用 LLVM IR API 的实践技巧。此外，我们还将回顾更高级的技能，如**程序引导优化 (PGO)** 和杀毒器的开发。

本章中，我们将讨论为新 **PassManager** 编写转换 **Pass** 和程序分析。LLVM Pass 是整个项目中最基本、最关键的概念之一。它允许开发人员将程序处理逻辑封装到一个模块化单元中，该单元可以根据情况由 **PassManager** 自由地与其他 Pass 组合。在 Pass 基础设施的设计方面，LLVM 实际上已经对 PassManager 和 AnalysisManager 进行了全面检查，以提高它们的运行时性能和优化质量。新 PassManager 使用了一个完全不同的接口来封装通行证。然而，新接口并不向后兼容旧接口，这就不能在新 PassManager 中运行旧 Pass，反之亦然。更糟糕的是，尽管现在 LLVM 和 Clang 默认都启用了这个新接口，但没有太多的在线学习资源讨论这个新接口。本章的内容将填补这一空白，并提供 LLVM 中这个关键系统的最新指南。

本章中，我们将讨论以下内容：

- 为新 PassManager 写一个 LLVM Pass
- 使用新 AnalysisManager
- 新 PassManager 中的设备

有了从本章学到的知识，读者们应该能够编写一个 LLVM Pass，使用新的 Pass 基础结构来转换，甚至优化代码。还可以利用 LLVM 程序分析框架提供的分析数据进一步提高 Pass 的质量。

9.1. 相关准备

本章中，我们将主要使用一个名为 `opt` 的命令行程序来测试我们的 Pass。可以使用这样的命令来构建它:

```
$ ninja opt
```

本章的代码连接：https://github.com/PacktPublishing/LLVM-Techniques-Tips-and-Best-Practices-Clangand-Middle-End-Libraries/tree/main/Chapter09.

9.2. 为新 PassManager 写一个 LLVM Pass

LLVM 中的 Pass 是对 LLVM IR 执行某些操作所必需的基本单元。这类似于工厂的单个生产步骤，需要加工的产品是 LLVM IR，工厂的工人就是 Pass。就像一个普通的工厂通常有多个制造步骤一样，LLVM 也包含多个按顺序执行的 Pass，称为 **Pass 流水**。图 9.1 展示了 Pass 管道的一个例子:

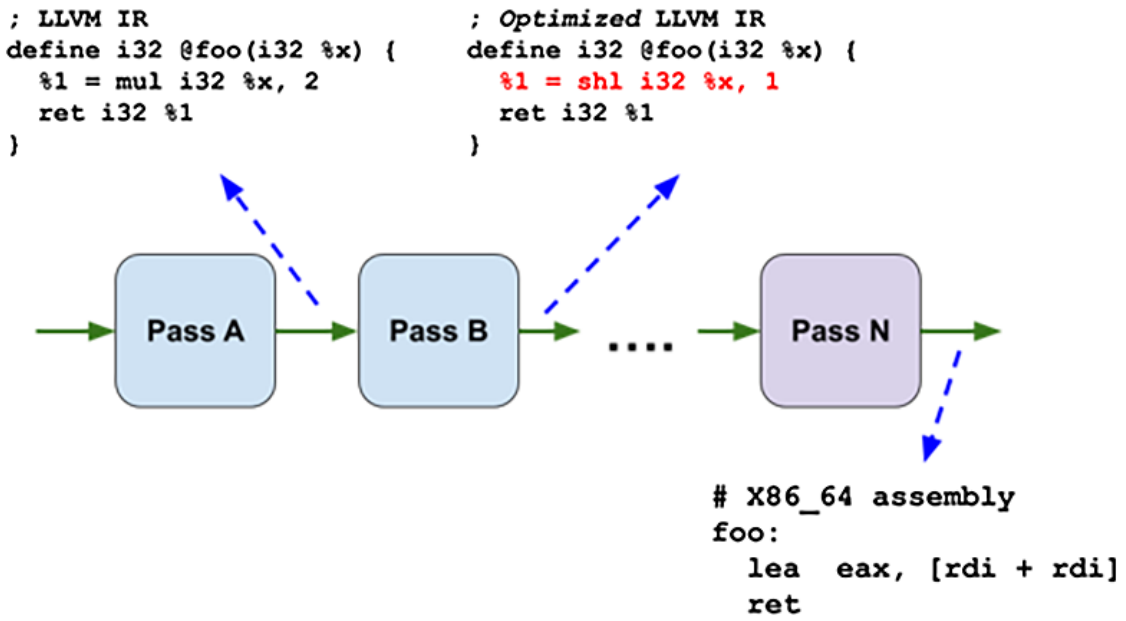

图 9.1 - LLVM Pass 流水的例子及其中间结果

上图中，多个通道沿一条直线排列。一个又一个 Pass 处理 `foo` 函数的 LLVM IR，例如：Pass B 对 `foo` 执行代码优化，用左移 (`shl`)1 替换算术乘法 (`mul`)2，在大多数硬件体系结构中，认为位移比乘法更快。此外，该图还说明了**代码生成**步骤建模为 Pass。LLVM 中的代码生成将 LLVM IR(与目标无关) 转换成特定硬件架构的汇编代码 (图 9.1 中的 **x86_64**)。每个详细的过程，比如：寄存器分配、指令选择或指令调度，都封装在一个 Pass 中，并按照一定的顺序执行。

代码生成 Pass

用于代码生成的 Pass 具有与普通 LLVM IR Pass 不同的 API。在代码生成阶段，LLVM IR 实际上转换成另一种 IR，称为 **Machine IR (MIR)**。本章中，我们只讨论 LLVM IR 和它的 Pass。

这个 Pass 流水在概念上是由 `PassManager` 的的基础设施管理。PassManager 拥有运行这些 Pass 的计划——例如：执行顺序。通常，交替使用术语 *Pass pipeline* 和 *PassManager*，因为它们有几乎相同的任务。在后面的小节中，我们将更详细地介绍流水本身，并讨论如何定制这些封装 Pass 的执行顺序。

现代编译器中的代码转换可能很复杂。因此，多个转换 Pass 可能需要相同的一组程序信息，这在 LLVM 中称为**分析**，以完成它们的工作。此外，为了实现效率最大话，LLVM 还缓存这些分析

数据，以便在可能的情况下重用这些数据。但是，由于转换 Pass 可能会更改 IR，一些缓存的分析数据 (以前收集的) 在运行该 Pass 之后就会过期。为了解决这些挑战，除了 PassManager 之外，LLVM 还创建了 **AnalysisManager** 来管理与程序分析相关的一切。

如本章介绍中提到的，LLVM 已经对其 Pass 和 PassManager(以及 AnalysisManager) 基础设施进行了一系列的检查。新的基础设施运行得更快，产生的结果质量更好。然而，新 Pass 在许多地方与旧的不同，我们将简要解释这些差异。除此之外，本章的其余部分，将只讨论新 Pass 的基础架构。

在本节中，我们将展示如何为新 PassManager 开发一个简单的 Pass。与往常一样，我们将从描述将要使用的示例项目开始。然后，展示创建 Pass 的步骤，该 Pass 可以使用 `opt` 从插件动态加载到 Pass 管道中 (前面提到过)。

9.2.1 项目概述

本节中，使用的示例项目称为 **StrictOpt**。它是一个 Pass 和 Pass 插件，为每个具有指针类型的函数形参添加 `noalias` 属性。实际上，它向 C 代码中的函数参数添加了一个 `restrict` 关键字。首先，让我们了解一下 `restrict` 关键字的作用。

C 和 C++ 中的 restrict 关键字

C99 中引入了 `restrict` 关键字。然而，它在 C++ 中没有对应的关键字。尽管如此，主流编译器如 Clang、GCC 和 MSVS 在 C++ 中都支持相同的功能。例如，在 Clang 和 GCC 中，可以在 C++ 代码中使用`__restrict__`或`__restrict`，效果与 C 中的 `restrict` 相同。

C 语言中，`restrict` 关键字可以与指针类型变量一起使用。在最常见的情况是，与指针类型的函数形参一起使用。示例如下:

```
int foo(int* restrict x, int* restrict y) {
  *x = *y + 1;
  return *y;
}
```

这个属性会告诉编译器，参数 `x` 永远不会指向与参数 `y` 相同的内存区域。换句话说，程序员可以使用这个关键字来说服编译器，他们永远不会像这样使用 `foo` 函数，如下所示:

```
...
// Programmers will NEVER write the following code
int main() {
  int V = 1;
  return foo(&V, &V);
}
```

这背后的基本原理是，如果编译器知道两个指针——本例中，两个指针参数——永远不会指向同一个内存区域，它就可以进行更积极的进行优化。更具体的理解是，如果比较 `foo` 函数的汇编代码与限制关键字和不限制关键字，后一个版本需要执行 5 条指令 (在 **x86_64** 上):

```
foo:
  mov eax, dword ptr [rsi]
```

```
  add eax, 1
  mov dword ptr [rdi], eax
  mov eax, dword ptr [rsi]
  ret
```

添加 restrict 关键字的版本只有四条指令:

```
foo:
  mov eax, dword ptr [rsi]
  lea ecx, [rax + 1]
  mov dword ptr [rdi], ecx
  ret
```

尽管这里的区别看起来很微妙，但在没有 restrict 的版本中，编译器需要插入额外的内存负载，以确保最后一个参数 *y(在原始 C 代码中) 总是读取最新的值。这些额外的成本可能会逐渐累积在更复杂的代码库中，并最终造成性能瓶颈。

现在，已经了解了 restrict 是如何工作的，以及对良好性能的重要性。在 LLVM IR 中，也有一个对应的指令来建模 restrict 关键字:noalias 属性。如果原始源代码中给出了诸如 restrict 之类的提示，则该属性则会附加到指针函数参数上。例如，foo 函数 (带有 restrict 关键字)，则翻译成如下 LLVM IR:

```
define i32 @foo(i32* noalias %0, i32* noalias %1) {
  %3 = load i32, i32* %1
  %4 = add i32 %3, 1
  store i32 %4, i32* %0
  ret i32 %3
}
```

此外，还可以在 C 代码中无 restrict 地生成 foo 函数的 LLVM IR 代码，如下所示:

```
define i32 @foo(i32* %0, i32* %1) {
  %3 = load i32, i32* %1
  %4 = add i32 %3, 1
  store i32 %4, i32* %0
  %5 = load i32, i32* %1
  ret i32 %5
}
```

这里，将有一个额外的内存负载 (如前面代码段突出显示的指令所示)，这与前面的程序集示例的情况类似。也就是说，LLVM 无法执行更积极的优化来移除内存负载，因为它不确定这些指针是否会重叠。

本节中，我们将编写一个 Pass 来为函数的每个指针参数添加 noalias 属性。Pass 将作为插件构建，当它加载到 opt 中时，用户可以使用--passes 参数显式触发 StrictOpt，如下所示:

```
$ opt --load-pass-plugin=StrictOpt.so \
      --passes="function(strict-opt)" \
      -S -o - test.ll
```

或者，如果优化级别大于或等于-O3，可以在其他优化之前运行 StrictOpt。示例如下:

```
$ opt -O3 --enable-new-pm \
    --load-pass-plugin=StrictOpt.so \
    -S -o - test.ll
```

我们将展示如何在这两种模式之间进行切换。

只是 Pass 的演示示例

注意，StrictOpt 仅仅是 Pass 的演示示例，并且在每个指针函数参数中添加 noalias 绝对不是在实际用例中应该做的事情。因为这样做可能会破坏程序的正确性。

下一节中，我们将展示如何创建此 Pass 的详细步骤。

9.2.2 编写 StrictOpt Pass

介绍如何将 StrictOpt 动态注册到 Pass 管道之前，下面将说明开发核心 Pass 逻辑的整个过程:

1. 我们只有两个源文件:StrictOpt.h 和 StrictOpt.cpp。前一个文件中，我们放置了 StrictOpt Pass 的架构:

```
#include "llvm/IR/PassManager.h"
struct StrictOpt : public PassInfoMixin<StrictOpt> {
  PreservedAnalyses run(Function &F,
    FunctionAnalysisManager &FAM);
};
```

在这里编写的 Pass 是一个函数，它运行在 FumctionIR 单元上。run 方法是 Pass 的主要入口，我们稍后将对其进行填充。它有两个参数: 我们将处理的 Function 类和可以分析数据的 FunctionAnalysisManager 类。返回一个 PreservedAnalyses 实例，会告诉 PassManager(和 AnalysisManager) 哪些分析数据对这个 Pass 是无效的。

如果之前有为旧 PassManager 编写 LLVM Pass 的经验，可能会发现旧新 Pass 之间的差异:

a) Pass 类不再从 FunctionPass、ModulePass 或 LoopPass 中派生。相反，在不同 IR 单元上运行的传递都是从 PassInfoMixin<YOUR_PASS> 派生。事实上，从 PassInfoMixin 派生甚至不再是功能 Pass 的必要条件——我们将把这个留给读者们作为练习。

b) 不是重写方法，比如：runOnFunction 或 runOnModule，而是定义一个普通的类成员方法 run(请注意，run 后面没有 override 关键字)，它操作相应的 IR 单元。

总的来说，新 Pass 的接口比旧版本更干净。这个区别还让新 PassManager 有更少的运行时开销。

2. 为了实现上一步的框架，我们将注意力转回 StrictOpt.cpp。这个文件中，我们首先创建以下方法定义:

```cpp
#include "StrictOpt.h"
using namespace llvm;
PreservedAnalyses StrictOpt::run(Function &F,
                                 FunctionAnalysisManager &FAM) {
  return PreservedAnalyses::all(); // Just a placeholder
}
```

返回的 `PreservedAnalyses::all()` 实例只是一个占位符，稍后将删除。

3. 现在，终于创建了将 `noalias` 属性添加到指针函数参数的代码。逻辑很简单: 对于 `Function` 类中的每个 `Argument` 实例，如果满足条件，则附加 `noalias` 属性:

```cpp
// Inside StrictOpt::run···
bool Modified = false;
for (auto &Arg : F.args()) {
  if (Arg.getType()->isPointerTy() &&
  !Arg.hasAttribute(Attribute::NoAlias)) {
    Arg.addAttr(Attribute::NoAlias);
    Modified |= true;
  }
}
```

`Function` 类的 `args()` 方法将返回一个表示所有形式参数的 `Argument` 实例。我们检查每个类型，以确保没有已添加 `noalias` 属性 (由 `attribute::noalias` 枚举表示) 的参数。如果一切正常，我们使用 `addAttr` 来添加 `noalias`。

这里的 `Modified` 标志记录了这个函数中是否有参数被修改，我们将很快使用这个标志。

4. 某些分析数据在转换 Pass 之后可能会过时，因为后者可能会改变 IR。因此，在编写 Pass 时，我们需要返回一个 `PreservedAnalysis` 实例，以显示哪些分析受到了影响，并且应该重新计算。虽然 LLVM 中有许多可用的分析，但不需要逐一列举。相反，有一些方便的实用函数来创建 `PreservedAnalyses` 的实例，表示所有的分析或不表示任何分析，这样我们只需要从中删去或添加 (不) 受影响的分析即可。以下是我们在 `StrictOpt` 中所做的工作:

```cpp
#include "llvm/Analysis/AliasAnalysis.h"
···
// Inside StrictOpt::run···
auto PA = PreservedAnalyses::all();
if (Modified)
  PA.abandon<AAManager>();
return PA;
```

这里，首先创建一个 `PreservedAnalyses` 实例 PA，表示所有的分析。然后，如果正在处理的 `Function` 类已经修改，则通过 `abandon` 方法丢弃 `AAManager` 的分析。`AAManager` 代表 LLVM 中的**别名分析**。

别名分析不深入细节了，就是查询两个指针是否指向相同的内存区域，或者它们指向的内存区域是否相互重叠。我们在这里讨论的 `noalias` 属性与这个分析有很强的关系，因为他们正在处理一个相同的问题。因此，如果生成新的 `noalias` 属性，那么所有缓存的别名分析数据都将过期。这就是我们要用 `abandon` 来使它无效原因。

注意，可以返回一个 `PreservedAnalyses::none()` 实例。可以告诉 AnalysisManager，如果不确定哪些分析受到了影响，则将每个分析标记为过时。当然，需要付出代价，因为 AnalysisManager 需要花费额外的时间，来重新进行计算和分析。

5. `StrictOpt` 的核心逻辑基本上已经介绍完了。现在，我们将展示如何将 Pass 动态注册到流水中。在 `StrictOpt.cpp` 中，我们创建了一个特殊的全局函数，叫做 `llvmGetPassPluginInfo`:

```
extern "C" ::llvm::PassPluginLibraryInfo LLVM_ATTRIBUTE_
WEAK
llvmGetPassPluginInfo() {
  return {
    LLVM_PLUGIN_API_VERSION, "StrictOpt", "v0.1",
    [](PassBuilder &PB) {…}
  };
}
```

这个函数返回一个 `PassPluginLibraryInfo` 实例，包含了各种各样的信息，比如：插件 API 版本 (`LLVM_PLUGIN_API_VERSION`) 和 Pass 名称 (`StrictOpt`)。它最重要的字段是一个 lambda 函数，只接受一个 `PassBuilder&` 参数。在这个特定的函数中，我们将把 `StrictOpt` 插入到 Pass 流水中的适当位置中。

`PassBuilder`，顾名思义，是 LLVM 用于构建 Pass 流水构建器。除了主要工作 (根据优化级别配置管道) 之外，还能帮助开发人员将 Pass 插入到管道中的一些地方。此外，为了增加灵活性，`PassBuilder` 允许通过使用 `opt` 上的`--passes` 参数来指定要运行的管道的文本描述，例如：依次执行 `InstCombine`、`PromoteMemToReg`、`SROA`(**SROA: Scalar Replacement of aggregate**) 命令:

```
$ opt --passes="instcombine,mem2reg,sroa" test.ll -S -o -
```

这一步中，我们要做的是确保在插件加载后，如果 `strict-opt` 出现在`--passes` 参数中，`opt` 将运行我们的 Pass，如下所示:

```
$ opt --passes="strict-opt" test.ll -S -o -
```

为此，我们使用 `PassBuilder` 中的 `registerPipelineParsingCallback` 方法:

```
…
[](PassBuilder &PB) {
  using PipelineElement = typename
  PassBuilder::PipelineElement;
  PB.registerPipelineParsingCallback(
  [](StringRef Name,
  FunctionPassManager &FPM, ArrayRef<PipelineElement>){
    if (Name == "strict-opt") {
      FPM.addPass(StrictOpt());
```

```
      return true;
    }
    return false;
  });
}
```

`registerPipelineParsingCallback` 方法接受另一个 lambda 回调作为参数。当 `PassBuilder` 在解析文本管道表示时遇到无法识别的 Pass 名称时，将进行回调。因此，在我们的实现中，当无法识别的 Pass 名称，即 `Name` 参数是 `strict-opt` 时，只需通过 `FunctionPassManager::addPass` 将 `StrictOptPass` 插入到流水中即可。

6. 或者，还希望在 Pass 流水的开始就触发我们的 `StrictOpt`，而不使用文本的流水描述。这意味着 Pass 将在其他 Pass 加载到 `opt` 之后马上运行，使用以下命令:

```
$ opt -O2 --enable-new-pm \
      --load-pass-plugin=StrictOpt.so test.ll -S -o -
```

(前面命令中的`--enable-new-pm` 标志强制选择使用新 PassManager，因为默认情况下它仍在使用旧 PassManager。我们以前没有使用过这个标志，因为`--passes` 默认启用新 PassManager。)

要做到这一点，我们不使用 `PassBuilder::registerPipelineParsingCallback` 来注册一个自定义 (管道) 解析器回调，而是使用 `registerPipelineStartEPCallback` 来处理。下面是前一步代码的另一个版本:

```
...
[](PassBuilder &PB) {
  using OptimizationLevel
  = typename PassBuilder::OptimizationLevel;
  PB.registerPipelineStartEPCallback(
  [](ModulePassManager &MPM, OptimizationLevel OL) {
    if (OL.getSpeedupLevel() >= 2) {
      MPM.addPass(
      createModuleToFunctionPassAdaptor(StrictOpt()));
    }
  });
}
```

- 在这里使用的 `registerPipelineStartEPCallback` 方法注册了一个回调，可以自定义 Pass 流水中的某些位置，称为**扩展点 (EPs)**。我们在这里定制的 EP 是计划中开始的环节之一。
- 与在 `registerPipelineParsingCallback` 中看到的 lambda 回调相比，`registerPipelineStartEPCallback` 的 lambda 回调只提供了 `ModulePassManager`，而不是 `FunctionPassManager`，来插入 `StrictOpt` Pass(这是一个 Pass 函数)。我们使用 `ModuleToFunctionPassAdapter` 来解决这个问题。

`ModuleToFunctionPassAdapter` 是一个模块 Pass，可以运行给定函数 Pass 模块的外围函数。它适合在只有 `ModulePassManager` 可用的上下文中运行一个 Pass 函数，比如在这个场景中：代码中显式的 `createModuleToFunctionPassAdaptor` 函数，用于从特定函数 Pass 创建新的 `ModuleToFunctionPassAdapter` 实例。

- 最后，在这个版本中，我们只在优化级别大于或等于`-O2` 时启用 `StrictOpt`。因此，我们利用传入 lambda 回调函数的 `OptimizationLevel` 参数，来决定是否要将 `StrictOpt` 插入流水中。

  通过这些 Pass 注册步骤，我们还了解了如何在不显式指定文本 Pass 流水的情况下触发我们的 `StrictOpt`。

总之，在本节中，我们了解了 LLVM Pass 和 Pass 流水的要点。通过 `StrictOpt` 项目，我们了解了如何为新 PassManager 开发一个 Pass(也封装为一个插件)，以及如何以两种不同的方式在 `opt` 中动态地在 Pass 流水中进行注册: 首先，通过文本描述显式触发 Pass；其次，在管道中的某个时间点 (EP) 运行它。我们还了解了如何根据 Pass 中所做的更改使分析失效。这些技能可以帮助您开发高质量和使用最新的 LLVM Pass，以一种可组合的方式 (以最大的灵活性) 处理 IR。下一节中，我们将深入研究 LLVM 的程序分析工具。这会提高了 LLVM 转换 Pass 的能力。

9.3. 使用新 AnalysisManager

现代编译器优化可能很复杂，通常需要来自目标程序的大量信息，以便做出正确的决策和最佳的转换，例如：在为新 PassManager 编写一个 LLVM Pass 时，LLVM 会使用 `noalias` 属性来计算内存别名信息，这可以用来删除冗余的内存负载。

其中一些信息——在 LLVM 中称为**分析**——评估代价相当昂贵。此外，单个分析也可能依赖于其他分析。因此，LLVM 创建 `AnalysisManager` 组件来处理所有与 LLVM 分析相关的任务。本节中，我们将展示如何在自己的 Pass 中使用 AnalysisManager，以便编写更强大和更复杂的程序转换或分析。我们还将使用一个示例项目 **HaltAnalyzer** 来演示我们的教程。下一节将提供 HaltAnalyzer 的概述，然后再进入详细的开发阶段。

9.3.1 项目概述

HaltAnalyzer 是在一个场景中设置的，目标程序使用一个特殊的函数 `my_halt`，当它调用时终止程序的执行。`my_halt` 函数类似于 `std::terminate` 函数，或者 `assert` 函数 (在完整性检查失败时)。

HaltAnalyzer 的工作是分析程序，以找到因 `my_halt` 函数而不可能运行的程序块。更具体地说，以如下代码为例:

```
int foo(int x, int y) {
  if (x < 43) {
    my_halt();
    if (y > 45)
    return x + 1;
    else {
      bar();
      return x;
```

```
    }
  } else {
    return y;
  }
}
```

因为 `my_halt` 在 `if (x < 43)` 语句的 true 分支的开头调用，所以该分支之后的代码永远不会执行 (也就是说，在到达这些行之前，`my_halt` 就停止了所有的程序执行)。

HaltAnalyzer 应该识别这些基本块，并将警告消息输出到 `stderr`。就像上一节的示例项目一样，HaltAnalyzer 也是一个封装在插件中的 Pass 函数。因此，如果使用前面的代码作为 HaltAnalyzer Pass 的输入，应该打印出如下信息:

```
$ opt --enable-new-pm --load-pass-plugin ./HaltAnalyzer.so \
      --disable-output ./test.ll
[WARNING] Unreachable BB: label %if.else
[WARNING] Unreachable BB: label %if.then2
$
```

`%if.elese` 和`%if.then2` 字符串只是 `if (y > 45)` 语句中基本块的名称 (不同的设备可能会看到不同的名称)。另一件值得注意的事情是`--disable-output` 命令行标志。默认情况下，`opt` 程序无论如何都会以二进制形式 (即 LLVM 位码) 输出 LLVM IR，除非用户通过`-o` 标志将输出重定向到其他地方。使用上述标志只是告诉 `opt` 不要这样做，因为我们对 LLVM IR 的最终内容不感兴趣 (不会去修改它)。

虽然 HaltAnalyzer 的算法看起来非常简单，但从头编写可能会很痛苦。这里，我们利用 LLVM 提供的分析: **支配树 (Dominator Tree, DT)**。**控制流图 (CFG)** 的概念已经在大多数入门级编译器课程中，所以我们不打算在这里深入解释它。简单地说，如果我们说一个基本块支配另一个块，每一个到达后者的执行流都保证先经过前者。DT 是 LLVM 中最重要、最常用的分析方法之一，大多数与控制流相关的转换离不开它。

把这个想法放到 HaltAnalyzer 中，我们只是简单地寻找所有的基本块，这些基本块包含对 `my_halt` 的函数调用 (我们从警告消息中排除了包含 `my_halt` 调用的基本块)。下一节中，我们将详细说明如何编写 HaltAnalyzer。

9.3.2 编写 HaltAnalyzer Pass

这个项目中，我们只创建一个源文件 `HaltAnalyzer.cpp`。大多数基础设施，包括 `CMakeListst.txt`，都可以从上一节的 `StrictOpt` 项目中拿过来直接用:

1. 在 `HaltAnalyzer.cpp` 中，我们首先创建以下 Pass 框架:

```
class HaltAnalyzer : public PassInfoMixin<HaltAnalyzer> {
  static constexpr const char* HaltFuncName = "my_halt";
  // All the call sites to "my_halt"
  SmallVector<Instruction*, 2> Calls;
```

```
  void findHaltCalls(Function &F);
public:
  PreservedAnalyses run(Function &F,
  FunctionAnalysisManager &FAM);
};
```

除了前一节中看到的 `run` 方法之外，我们还创建了一个额外的方法 `findHaltCalls`，它将收集当前函数中所有对 `my_halt` 的指令调用，并将它们存储在 `calls` vector 中。

2. 先实现 `findHaltCalls`:

```
void HaltAnalyzer::findHaltCalls(Function &F) {
  Calls.clear();
  for (auto &I : instructions(F)) {
    if (auto *CI = dyn_cast<CallInst>(&I)) {
      if (CI->getCalledFunction()->getName() ==
      HaltFuncName)
      Calls.push_back(&I);
    }
  }
}
```

这个方法使用 `llvm::instructions` 来遍历当前函数中的每个 `Instruction` 调用，并逐个检查。如果指令调用是 `CallInst`——代表典型的函数调用点——并且调用函数的名字是 `my_halt`，我们将把它放入到 `Calls` vector 中以供以后使用。

重建函数名

请注意，当一行 C++ 代码编译成 LLVM IR 或本机代码时，任何符号的名称——包括函数名——都将与原始源代码中的不同，例如：一个简单的函数，名称 `foo`，没有参数，可能在 LLVM IR 中称为`_Z3foov`。在 C++ 中，我们称这种转换为**名称重建**。不同的平台也会采用不同的名称转换方案，例如：Visual Studio 中，相同的函数名在 LLVM IR 中变成`?foo@@YAHH@Z`。

3. 现在，让我们回到 `HaltAnalyzer::run` 方法。要做两件事，将通过 `findHaltCalls` 收集 `my_halt` 的调用点 (这是我们刚刚写的)，然后检索 DT，分析数据:

```
#include "llvm/IR/Dominators.h"
...
PreservedAnalyses
HaltAnalyzer::run(Function &F, FunctionAnalysisManager
&FAM) {
  findHaltCalls(F);
  DominatorTree &DT = FAM.
  getResult<DominatorTreeAnalysis>(F);
  ...
}
```

前面代码中,向我们展示了如何利用 `FunctionAnalysisManager` 类型参数来检索特定 `Function` 类，再进行特定的分析数据 (在本例中为 `DominatorTree`)。

目前为止，我们 (在某种程度上) 交替使用了分析和分析数据这两个词，但在实际的 LLVM 实现中，它们是两个不同的实体。以我们这里用的 DT 为例:

a) **DominatorTreeAnalysis** 是一个 C++ 类，计算给定 `Function` 的关系。换句话说，它是执行分析的对象。

b) **DominatorTree** 是一个 C++ 类，表示 `DominatorTreeAnalysis` 生成的结果。这只是静态数据，AnalysisManager 会缓存它直到它失效为止。

此外,LLVM 要求每个分析通过 `Result` 成员类型来搞清其附属的结果类型。例如:`DominatorTreeAnalysis::`等于 `DominatorTree`。

为了使其更加正式，为了将分析类 `T` 的分析数据与 `Function` 变量 `F` 关联起来，可以使用如下的代码:

```
// `FAM` is a FunctionAnalysisManager
typename T::Result &Data = FAM.getResult<T>(F);
```

4. 在检索 `DominatorTree` 之后，是时候找到之前收集到的 `Instruction` 调用点，并控制的所有基本块了:

```
PreservedAnalyses
HaltAnalyzer::run(Function &F, FunctionAnalysisManager
&FAM) {
  ...
  SmallVector<BasicBlock*, 4> DomBBs;
  for (auto *I : Calls) {
    auto *BB = I->getParent();
    DomBBs.clear();
    DT.getDescendants(BB, DomBBs);
    for (auto *DomBB : DomBBs) {
      // excluding the block containing `my_halt` call site
      if (DomBB != BB) {
        DomBB->printAsOperand(
        errs() << "[WARNING] Unreachable BB: ");
        errs() << "\n";
      }
    }
  }
  return PreservedAnalyses::all();
}
```

通过使用 `DominatorTree::getDescendants` 方法，我们可以检索 `my_halt` 调用点所控制的所有基本块。注意，`getDescendants` 的结果也将包含放入的查询块 (本例中，是包含 `my_halt` 调用点的块)，因此需要我们在使用 `BasicBlock::printAsOperand` 方法打印基本块名称之前将其排除。

返回 `PreservedAnalyses::all()` 的末尾，它告诉 AnalysisManager 这个 Pass 不会使任何分析失效 (因为没有修改 IR)，我们将在这里包装 `HaltAnalyzer::run` 方法。

5. 最后，我们需要动态地将 HaltAnalyzer Pass 插入到 Pass 流水中。使用与上一节相同的方法，通过实现 `llvmGetPassPluginInfo` 函数，并使用 `PassBuilder` 将我们的 Pass 放在流水中

的某些 EP 上:

```
extern "C" ::llvm::PassPluginLibraryInfo LLVM_ATTRIBUTE_
WEAK
llvmGetPassPluginInfo() {
  return {
    LLVM_PLUGIN_API_VERSION, "HaltAnalyzer", "v0.1",
    [](PassBuilder &PB) {
      using OptimizationLevel
      = typename PassBuilder::OptimizationLevel;
      PB.registerOptimizerLastEPCallback(
      [](ModulePassManager &MPM, OptimizationLevel OL)
      {
        MPM.addPass(createModuleToFunctionPassAdaptor
        (HaltAnalyzer()));
      });
    }
  };
}
```

与前一节中的 `StrictOpt` 相比，我们使用 `registeroptimizerlastcallback` 在所有其他优化 Pass 之后插入 HaltAnalyzer。这背后的基本原理是，一些优化可能会修改基本块，所以过早地提示警告可能不是很有用。尽管如此，我们仍然利用 `ModuletoFunctionPassAdaptor` 来包装我们的 Pass，这是因为 `registeroptimizerlastcallback` 只提供了 `ModulePassManager`，让我们添加 Pass(其实就是一个 Pass 函数)。

这些就是实现 HaltAnalyzer 的所有步骤了。现在，我们已经了解了如何使用 LLVM 的程序分析基础结构，来获取 LLVM Pass 中关于目标程序的更多信息。在开发一个 Pass 时，这些技能可以让开发者对 IR 有更深入的理解。此外，这个设施允许重用来自 LLVM 的高质量、现成的程序分析算法，而不是自己重新创建轮子。要浏览 LLVM 提供的所有可用分析，在源代码树中的 `llvm/include/llvm/Analysis` 文件夹中的大多数头文件，都是可以使用的独立分析数据文件。

本章的最后一节中，我们将展示一些诊断技术，这些技术对调试 LLVM Pass 非常有用。

9.4. 新 PassManager 中的设施

LLVM 中的 PassManager 和 AnalysisManager 挺复杂，它们管理数百个 Pass 和分析之间的交互，当我们试图诊断由它们引起的问题时，可能会是一个严峻的挑战。此外，编译器工程师通常会修复编译器中崩溃或**编译错误**的问题。在这些场景中，提供了对 Pass 和 Pass 流水有用的设施可以极大地提高修复这些问题的效率。幸运的是，LLVM 已经想到了这点，并提供了许多这样的工具。

编译错误

编译错误通常指编译程序中的逻辑问题，这是由编译器引入，例如：过于激进的编译器优化会删除某些不应该删除的循环，从而导致编译后的软件故障，或者错误地重新排序内存屏障，从而使生成的代码中出现竞争条件。

在下面的每个部分中，我们将每次介绍一个工具。以下是他们的列表:

- 打印通道流水细节
- 打印每次 Pass 对 IR 的修改
- 将 Pass 流水一分为二

这些工具可以在 opt 的命令行界面中进行交互。事实上，你也可以创建自己的设施 (甚至不需要改变 LLVM 源树!)，我们会将把这个留给读者们作为练习。

9.4.1 打印通道流水细节

LLVM 中有许多不同的优化级别，即我们在使用 clang(或 opt) 时熟悉的-O1、-O2 或-Oz 标志。每个优化级别运行一组不同的 Pass，并将它们按照不同的顺序排列。在某些情况下，这可能会在性能或正确性方面极大地影响生成的代码。有时候，为了清楚地理解我们将要处理的问题，了解这些配置很重要。

要打印出所有的 Pass,以及它们当前在 opt 中运行的顺序,我们可以使用--debug-pass-manager 标志，例如：给定下面的 C 代码 test.c，我们将看到以下内容:

```c
int bar(int x) {
  int y = x;
  return y * 4;
}
int foo(int z) {
  return z + z * 2;
}
```

首先使用以下命令生成 IR:

```
$ clang -O0 -Xclang -disable-O0-optnone -emit-llvm -S test.c
```

-disable-O0-optnone 标志

默认情况下，clang 将为-O0 优化级别下的每个函数附加一个特殊属性 optnone。此属性将阻止附加函数的任何进一步优化。这里，-disableo-optnone (前端) 标志会阻止 clang 附加这个属性。

然后，使用以下命令打印出所有在-O2 优化级别下运行的 Pass:

```
$ opt -O2 --disable-output --debug-pass-manager test.ll
Starting llvm::Module pass manager run.
...
Running pass: Annotation2MetadataPass on ./test.ll
Running pass: ForceFunctionAttrsPass on ./test.ll
...
```

```
Starting llvm::Function pass manager run.
Running pass: SimplifyCFGPass on bar
Running pass: SROA on bar
Running analysis: DominatorTreeAnalysis on bar
Running pass: EarlyCSEPass on bar
...
Finished llvm::Function pass manager run.
...
Starting llvm::Function pass manager run.
Running pass: SimplifyCFGPass on foo
...
Finished llvm::Function pass manager run.
Invalidating analysis: VerifierAnalysis on ./test.ll
... $
```

前面的命令行输出告诉我们,opt 首先运行一组模块级优化,这些 Pass 的顺序 (例如,Annotation2MetadataPass 和 ForceFunctionAttrsPass) 也进行列出。之后，在对 foo 函数运行这些优化之前，对 bar 函数 (例如，SROA) 执行一系列函数级优化。此外，还显示了流水中使用的分析 (例如，DominatorTreeAnalysis)，以及一条提示信息，说明已失效 (由某个 Pass) 组件。

总而言之，--debug-pass-manager 是一个有用的工具，可以查看 Pass 以及在某个优化级别上，Pass 流水运行的顺序。了解这些信息可以了解 Pass 和分析如何与输入 IR 进行交互。

9.4.2 打印每次 Pass 对 IR 的修改

要了解特定转换 Pass 对目标程序的影响，最直接的方法是比较该 Pass 处理 IR 之前和之后的 IR。更具体地说，我们对特定转换 Pass 所做的更改感兴趣，例如：LLVM 错误地删除了一个它不应该进行的循环，我们想知道 Pass 会做什么，以及什么时候在 Pass 流水中进行了删除操作。

通过在 opt 中使用--print-changed 标志 (以及稍后介绍的其他一些标志)，我们可以在每个 Pass 之后打印出 IR，如果它被那个 Pass 修改过。使用上一段中的 test.c(及其 IR 文件 test.ll) 示例代码，我们可以使用以下命令打印每个 Pass 所做的更改 (如果有更改的话):

```
$ opt -O2 --disable-output --print-changed ./test.ll
*** IR Dump At Start: ***
...
define dso_local i32 @bar(i32 %x) #0 {
  entry:
  %x.addr = alloca i32, align 4
```

```
  %y = alloca i32, align 4
  ...
  %1 = load i32, i32* %y, align 4
  %mul = mul nsw i32 %1, 4
  ret i32 %mul
}
...
*** IR Dump After VerifierPass (module) omitted because no
change ***
...
...
*** IR Dump After SROA *** (function: bar)
; Function Attrs: noinline nounwind uwtable
define dso_local i32 @bar(i32 %x) #0 {
  entry:
  %mul = mul nsw i32 %x, 4
  ret i32 %mul
}
...
$
```

这里，我们只显示了少量的输出。在代码中，我们可以看到该工具将首先打印出原始 IR(IR Dump At Start)，然后在每个 Pass 处理后显示 IR，例如：前面的代码显示，在 SROA Pass 之后，bar 函数变短了。如果 Pass 没有修改 IR，将忽略 IR 转储，以减少噪声的数量。

有时，我们只对发生在特定函数集上的变化感兴趣，例如：本例中的 foo 函数。我们可以添加--filter-print-funcs=< 函数名 > 标志 (而不是打印整个模块的更改日志)，从而仅打印函数子集的 IR 更改，例如：只打印 foo 函数的 IR 更改，可以使用以下命令:

```
$ opt -O2 --disable-output \
        --print-changed --filter-print-funcs=foo ./test.ll
```

就像--filter-print-funcs,有时我们只希望看到由特定的一组传递所做的更改,比如:SROA 和 InstCombine Pass。这种情况下，可以添加--filter-passes=<Pass 名称 >，例如：只查看与 SROA 和 InstCombine 相关的内容，我们可以使用以下命令:

```
$ opt -O2 --disable-output \
        --print-changed \
        --filter-passes=SROA,InstCombinePass ./test.ll
```

现在，已经了解了如何打印流水中所有 Pass 之间的 IR 差异，使用额外的过滤器可以进一步关注特定的功能或 Pass。换句话说，这个工具可以轻松地观察 Pass 流水中 IR 的变化，并快速地发现可能感兴趣的任何痕迹。下一节中，我们将了解如何通过将 Pass 流水一分为二来调试代码优化中出现的问题。

9.4.3 将 Pass 流水一分为二

上一节中，介绍了`--print-changed` 标志，它在 Pass 流水中打印出 IR 更改日志。我们还提到，对于感兴趣的 IR 改变这个标志是有用的，例如：导致错误编译错误的无效代码转换。或者，也可以将 Pass 流水一分为二，以实现类似的目的。更具体地说，可以使用 `opt` 中的`--opt-bisect-limit=<n>` 标志，通过禁用除前 `N` 个以外的所有 Pass，来分割 Pass 管道。下面的命令展示了一个这样的例子:

```
$ opt -O2 --opt-bisect-limit=5 -S -o - test.ll
BISECT: running pass (1) Annotation2MetadataPass on module (./test.ll)
BISECT: running pass (2) ForceFunctionAttrsPass on module (./test.ll)
BISECT: running pass (3) InferFunctionAttrsPass on module (./test.ll)
BISECT: running pass (4) SimplifyCFGPass on function (bar)
BISECT: running pass (5) SROA on function (bar)
BISECT: NOT running pass (6) EarlyCSEPass on function (bar)
BISECT: NOT running pass (7) LowerExpectIntrinsicPass on function (bar)
BISECT: NOT running pass (8) SimplifyCFGPass on function (foo)
BISECT: NOT running pass (9) SROA on function (foo)
BISECT: NOT running pass (10) EarlyCSEPass on function (foo)
...
define dso_local i32 @bar(i32 %x) #0 {
  entry:
  %mul = mul nsw i32 %x, 4
  ret i32 %mul
}
define dso_local i32 @foo(i32 %y) #0 {
  entry:
  %y.addr = alloca i32, align 4
  store i32 %y, i32* %y.addr, align 4
```

```
  %0 = load i32, i32* %y.addr, align 4
  %1 = load i32, i32* %y.addr, align 4
  %mul = mul nsw i32 %1, 2
  %add = add nsw i32 %0, %mul
  ret i32 %add
} $
```

(请注意，这与前面几节中的例子不同，上面的命令同时打印了`--opt-bisect-limit` 和最后的 IR。)

由于我们实现了`--opt-bisect-limit=5` 标志，Pass 流水只运行前 5 个 Pass。正如从诊断消息中看到的，SROA 应用于 `bar`，而不是 `foo` 函数，这使得 `foo` 的最终 IR 不是最优的。

通过改变`--opt-bisect-limit` 后面的数字，我们可以调整切割点，直到出现某些代码更改或触发某个 bug(例如，崩溃)。这是非常有用的 (前期) 过滤步骤，可以将原始问题缩小到更小的范围。此外，由于它使用数值作为参数，这个特性非常适合于自动化环境，例如：自动崩溃报告工具或性能回归跟踪工具。

本节中，介绍了几种用于调试和诊断 Pass 流水的工具。当涉及到修复问题 (如编译器崩溃、(目标程序上的) 性能倒退和编译错误) 时，这些工具可以极大地提高工作效率。

9.5. 总结

本章中，我们了解了如何为新 PassManager 编写 LLVM Pass，以及如何通过 AnalysisManager，在 Pass 中使用程序分析数据。我们还了解了如何在使用 Pass 流水时，利用各种设施来改善开发的经验。有了从本章中获得的技能，现在可以编写一个 Pass 来处理 LLVM IR，甚至可以用来转换甚至优化程序。

这些主题是在开始任何 IR 级别转换或分析任务之前，需要学习的一些最基本和关键性的技能。如果一直在使用旧 PassManager，这些技能也可以将代码迁移到新 PassManager 系统 (现在默认启用了新 PassManager 系统)。

下一章中，将展示使用 LLVM IR API 时应该知道的各种技巧和实践经验。

9.6. 问题

1. 在 `StrictOpt` 的例子中，为新 PassManager 部分写一个 LLVM Pass 时，如何在不派生 PassInfoMixin 类的情况下写一个 Pass？
2. 如何为新 PassManager 开发一个自定义工具？另外，如何在不修改 LLVM 源树的情况下做到这一点？(提示: 考虑一下本章中的 Pass 插件。)

第 10 章 处理 LLVM IR

前一章中，我们了解了 LLVM 中的 PassManager 和 AnalysisManager。了解了 LLVM Pass 的开发教程，以及如何通过 AnalysisManager 检索程序分析数据。相应的知识和技能有助于为开发人员创建，用于代码转换和程序分析的可组合构建块。

本章中，将重点讨论 LLVM IR 的处理方法。LLVM IR 是一种用于程序分析和编译器转换的 (目标无关的) 中间表示。可以将 LLVM IR 看作需要优化和编译的代码的另一种形式。然而，与熟悉的 C/C++ 代码不同，LLVM IR 以一种不同的方式描述了这个程序——稍后将更具体的说明。在 LLVM IR 上执行编译过程后，LLVM 所做的大部分工作是使输入程序运行更快或体积更小。第 9 章，使用 PassManager 和 AnalysisManager 时，我们描述了不同的 Pass 是如何以流水方式组织的——这是 LLVM 转换输入代码的高级结构。本章中，我们将展示如何以一种有效的方式修改 LLVM IR 的信息

虽然查看 LLVM IR 最直接和可视化的方式是通过文本表示，但 LLVM 提供了包含一组强大的现代 C++ API 的库来与 IR 进行交互。这些 API 可以检查 LLVM IR 在内存中的表示，并帮助我们操作，这将有效地改变正在编译的目标程序。这些 LLVM IR 库可以嵌入到各种各样的应用程序中，允许开发人员转换和分析目标源代码。

适用于不同编程语言的 LLVM API

LLVM 官方只支持两种语言的 API: C 和 C++。C++ 是功能最完整、更新最及时的，但也有最不稳定的接口——它可能在没有向后兼容的情况下，会随时进行更改。另一方面，C API 有稳定的接口，但代价是滞后于新特性的更新，甚至使某些特性缺失。OCaml、Go 和 Python 的 API 绑定在源代码树中，作为社区驱动的项目。

我们将尝试用普遍适用的模块来进行引导，这些模块由常见的主题和任务驱动，并由许多现实的示例支持。以下是我们将在本章讨论的内容:

- 学习 LLVM IR 基础知识
- 使用值和指令
- 使用循环

我们将首先介绍 LLVM IR。然后，了解 LLVM 中两个最基本的元素 IR 值和指令后，将了解一下 LLVM 对于循环的处理——一个更高级的主题，这对于性能敏感的应用至关重要。

10.1. 相关准备

本章中需要的工具是 `opt` 命令行程序和 `clang`。请使用以下命令构建它们:

```
$ ninja opt clang
```

本章中的大部分代码都可以在 LLVM Pass 和 Pass 插件中实现 (如第 9 章)。另外，请安装 Graphviz 工具。可以参考以下页面获取系统的安装指南:https://graphviz.org/download。例如，

在 Ubuntu 上，可以通过以下命令安装该包:

```
$ sudo apt install graphviz
```

我们将使用 Graphviz 提供的命令行工具——dot——来可视化函数的控制流。

如果没有另外指定的话，本章中提到的代码示例可以在 LLVM Pass 中实现。

10.2. 学习 LLVM IR 基础知识

LLVM IR 是要优化和编译程序的另一种形式。然而，它的结构不同于 C/C++ 等普通编程语言。LLVM IR 以分层的方式组织。这个层次结构中的层次——从顶部开始计算——是模块、函数、基本块和指令。下面的图表显示了它们的结构:

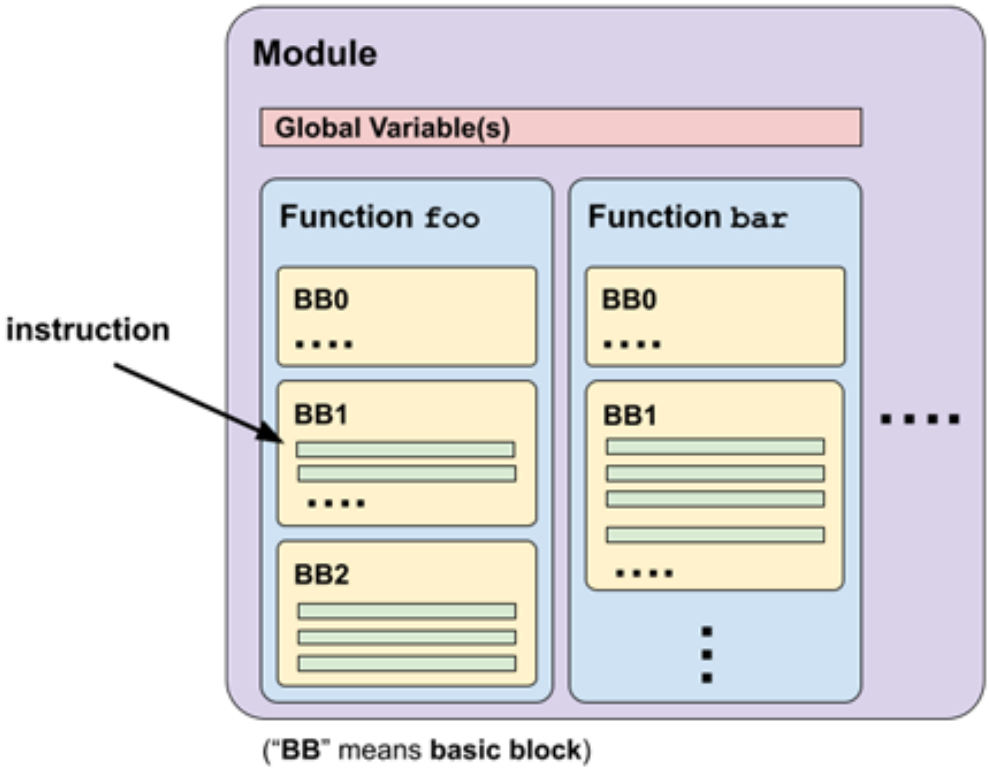

图 10.1 - LLVM IR 的层级结构

一个**模块**代表一个翻译单元——通常是一个源文件。每个模块可以包含多个**函数** (或全局变量)。每个都包含一个**基本块**列表，每个基本块包含一个**指令**列表。

简要介绍——基本块

基本块表示一个只有一个入口和一个出口的指令列表。换句话说，如果执行一个基本块，控制流需要保证遍历块中的每一条指令。

了解 LLVM IR 的结构后，让我们看一个 LLVM IR 的例子。假设我们有下面的 C 代码 `foo.c`:

```c
int foo(int a, int b) {
  return a > 0? a - b : a + b;
}
```

我们可以使用下面的 `clang` 命令来生成文本形式的 LLVM IR:

```
$ clang -emit-llvm -S foo.c
```

结果将放到 `foo.ll` 文件中。下图显示了其部分内容，并对相应的 IR 单元进行了注释:

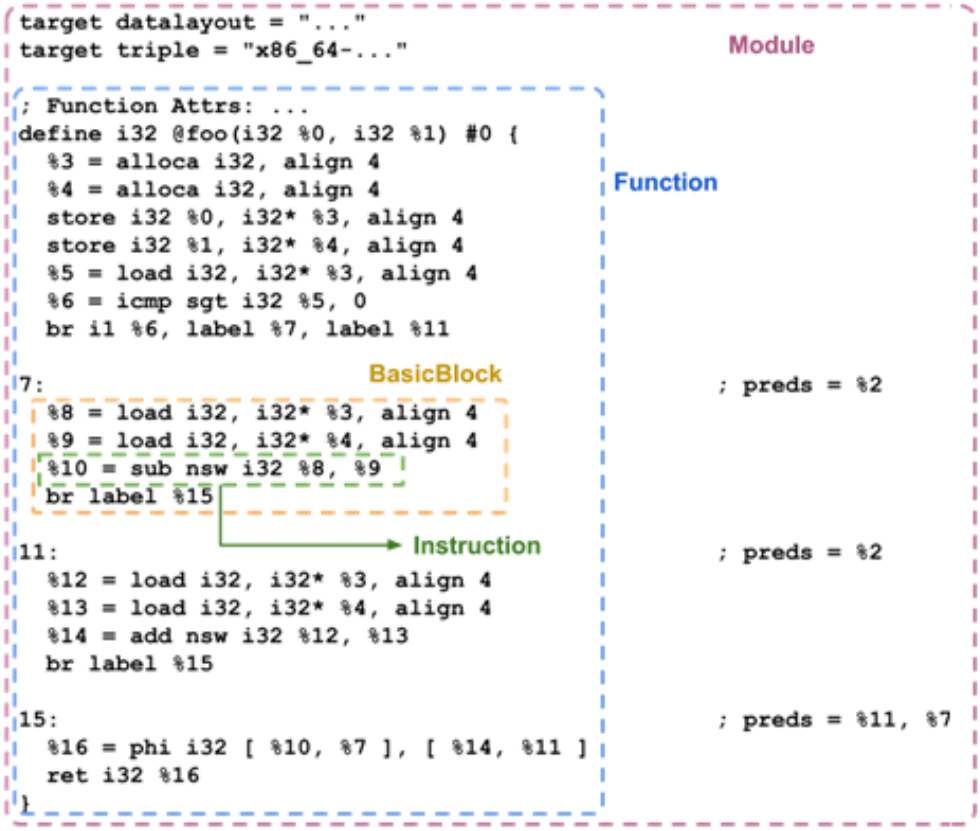

图 10.2 - foo.ll 中的部分内容，使用相应的 IR 单元进行标注

在文本形式中，指令通常以以下格式呈现:

```
<result> = <operator / op-code> <type>, [operand1, operand2, ⋯]
```

例如，假设我们有以下指令:

```
%12 = load i32, i32* %3
```

在这里，`%12` 是结果值，`load` 是操作码，`i32` 是该指令的数据类型，`%3` 是唯一的操作数。

除了文本表示之外，LLVM IR 中的几乎每个组件都有一个名称相同的 C++ 类对应项，例如：函数和基本块分别由 `Function` 和 `BasicBlock` C++ 类简单地表示。

不同种类的指令由类表示，这些类都由 `Instruction` 类派生，例如：`BinaryOperator` 类表示二进制操作指令，而 `Returnst` 类表示返回语句。稍后，我们将更详细地介绍 `Instruction` 及其子类。

图 10.1 所示的层次结构是 LLVM IR 的具体结构，这就是它们在内存中的存储方式。在此基础上，LLVM 还提供了其他逻辑结构来查看不同 IR 单元的关系。它们通常对具体的结构进行评估，并作为辅助数据结构存储或作为分析结果处理。以下是 LLVM 中最重要的一些内容:

- **控制流图 (CFG)**: 这是一种组织成基本块的图结构，以显示控制流关系。图中的顶点代表基本块，而边代表单个控制流传输路径。
- **循环**: 这表示我们熟悉的循环，它由多个基本块组成，至少有一条后边——一条返回父顶点或祖先顶点的控制流边。我们将在本章的最后一节中更详细地进行讨论。
- **调用图**: 与 CFG 类似，调用图也显示了控制流传输，但是顶点变成了单独的函数，而边变成了函数调用关系。

下一节中，我们将了解如何在具体和逻辑结构中迭代不同的 IR 单元。

10.2.1 迭代不同的 IR 单元

迭代 IR 单元——基本块或指令——对于 LLVM IR 开是必不可少。这通常是我们在许多转换或分析算法中必须完成的第一步——扫描整个代码，并找到一个感兴趣的区域，以便应用某些测量工具。本节中，将了解如何迭代不同 IR 单元。我们将讨论以下内容:

- 迭代指令
- 迭代基本块
- 迭代调用图
- 了解 GraphTraits

让我们从讨论如何迭代指令开始。

10.2.2 迭代指令

指令是 LLVM IR 中最基本的元素之一，通常表示程序中的单个操作，例：如算术操作或函数调用。遍历单个基本块或函数中的所有指令是大多数程序分析和编译器优化的基础。

要遍历一个基本块中的所有指令，只需要在块上使用一个简单的 for-each 循环:

```
1 // `BB` has the type of `BasicBlock&`
2 for (Instruction &I : BB) {
3   // Work on `I`
4 }
```

我们可以用两种方法遍历函数中的所有指令。首先，在访问指令之前，可以遍历函数中的所有基本块。下面是一个例子:

```
// `F` has the type of `Function&`
for (BasicBlock &BB : F) {
  for (Instruction &I : BB) {
    // Work on `I`
  }
}
```

其次，可以利用一个名为 inst_iterator 的程序。下面是一个例子:

```
#include "llvm/IR/InstIterator.h"
...
// `F` has the type of `Function&`
for (Instruction &I : instructions(F)) {
  // Work on `I`
}
```

使用前面的代码，可以检索这个函数中的所有指令。

观察指令

在很多情况下，我们希望对一个基本块或函数中的不同类型的指令应用有不同的处理方法。例如，我们有以下代码:

```
for (Instruction &I : instructions(F)) {
  switch (I.getOpcode()) {
    case Instruction::BinaryOperator:
    // this instruction is a binary operator like `add` or `sub`
      break;
    case Instruction::Return:
    // this is a return instruction
      break;
    ...
  }
}
```

不同种类的指令由来自 Instruction 的 (不同) 类进行建模。因此，一个 Instruction 实例可以表示它们中的任何一个。前面代码片段中显示的 getOpcode 方法可以为您提供唯一的令牌——即给定代码中的 Instruction::BinaryOperator 和 Instruction::Return——展示有关底层类的信息。然而，如果想处理派生类 (在本例中为 Returnst) 实例，而不是“原始”Instruction，则需要进行一些类型转换。

LLVM 提供了一种更好的方法来实现这种访问模式——Instvisitor。InstVisitor 是一个类，每个成员方法都是特定指令类型的回调函数。可以在继承了 InstVisitor 类后定义自己的回调函数。例如，下面的代码:

```
#include "llvm/IR/InstVisitor.h"
class MyInstVisitor : public InstVisitor<MyInstVisitor> {
  void visitBinaryOperator(BinaryOperator &BOp) {
```

```
    // Work on binary operator instruction
    ...
  }
  void visitReturnInst(ReturnInst &RI) {
    // Work on return instruction
    ...
  }
};
```

这里显示的每个 `visitXXX` 方法都是特定指令类型的回调函数。请注意，我们没有覆盖这些方法 (没有覆盖关键字附加到方法)。而且，`InstVisitor` 不是为所有指令类型定义回调函数，而是允许定义那些我们感兴趣的指令类型。

当定义 `MyInstVisitor` 后，就可以简单地创建它的实例，并调用 `visit` 方法来启动访问流程。以下面的代码为例:

```
// `F` has the type of `Function&`
MyInstVisitor Visitor;
Visitor.visit(F);
```

还有 `Instruction`、`BasicBlock` 和 `Module` 的 `visit` 方法。

基本块和指令的排序

我们在本节中介绍的所有技能都假设，基本块的排序或访问指令 (这不是主要关注点)。然而，重要的是要知道 `Function` 不会以特定的线性顺序存储，或迭代其闭环的 `BasicBlock` 实例。我们将展示如何以各种有意义的顺序，是如何遍历所有基本块的。

在此基础上，了解了从基本块或函数迭代指令的几种方法。现在，让我们了解一下如何在函数中迭代基本块。

10.2.3 迭代基本块

前一节中，了解了如何使用简单的 for 循环迭代函数的基本块。然而，开发人员只能以这种方式接收任意顺序的基本块——这种顺序既没有给定执行顺序，也没有给定块之间的控制流信息。本节中，我们将展示如何以更有意义的方式迭代基本块。

基本块是表达函数控制流的重要元素，可以用有向图表示——即 CFG。为了让你对典型的 CFG 有一个具体的了解，我们利用 `opt` 工具中的特性。假设你有一个 LLVM IR 文件 `foo.ll`，可以使用以下命令以 Graphviz 格式打印出每个函数的 CFG:

```
$ opt -dot-cfg -disable-output foo.ll
```

这个命令将为 `foo.ll` 中的每个函数生成一个`.dot` 文件。

.dot 文件可能会被隐藏

CFG 的.dot 文件中每个函数的文件名通常以点字符 (.) 开头。在 Linux/Unix 系统上，这有将会让文件隐藏起来，不能使用普通的 `ls` 看到。因此，需要使用 `ls -a` 才能来显示这些文件。

每个 texttt.dot 文件包含该函数的 CFG 的 Graphviz 表示。Graphviz 是表示图形的通用文本格式，人们通常在研究 texttt.dot 文件之前将其转换为其他 (图像) 格式，例如：使用下面的命令，可以将 texttt.dot 文件转换为 PNG 图像文件，以直观地显示图形:

```
$ dot -Tpng foo.cfg.dot > foo.cfg.png
```

下图有两个示例:

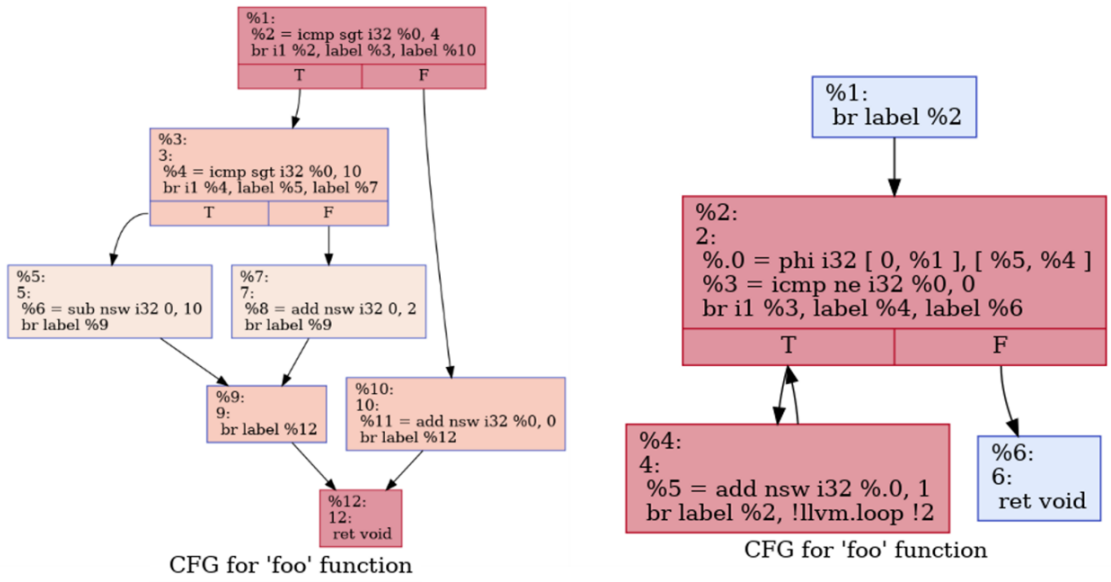

图 10.3 -左: 包含分支的函数的 CFG; 右:CFG 用于包含循环的函数

上图的左边显示了包含几个分支的函数的 CFG，右边显示了一个包含单循环的函数的 CFG。

现在，我们知道基本块组织成一个有向图——也就是 CFG。可以迭代这个 CFG，使它沿着边和节点吗？LLVM 通过提供四种不同方式迭代图的实用工具来回答这个问题: 拓扑顺序、深度优先 (本质上是做 DFS)、广度优先 (本质上是做 BFS) 和强连接组件 (SCCs)。我们将在下面的小节中学习如何使用这些工具。

让我们从拓扑顺序开始。

拓扑顺序

拓扑顺序是一种简单的线性排序，保证图中的每个节点只有在访问了它的所有父 (前任) 节点之后才会访问到。LLVM 提供了 `po_iterator` 和其他一些函数来在 CFG 上实现反向的拓扑顺序

(反向的拓扑顺序更容易实现)。下面的代码是一个使用 po_iterator 的例子:

```
#include "llvm/ADT/PostOrderIterator.h"
#include "llvm/IR/CFG.h"
// `F` has the type of `Function*`
for (BasicBlock *BB : post_order(F)) {
  BB->printAsOperand(errs());
  errs() << "\n";
}
```

post_order 函数只是一个辅助函数,用于创建 po_iterator 迭代的范围。请注意,llvm/IR/CFG.h 头文件是必要的，可以让 po_iterator 在 Function 和 BasicBlock 上工作。

如果将上述代码应用于包含分支的函数，将得到以下命令行输出:

```
  label %12
  label %9
  label %5
  label %7
  label %3
  label %10
  label %1
```

或者，可以使用几乎相同的语法从一个特定的基本块遍历:

```
// `F` has the type of `Function*`
BasicBlock &EntryBB = F->getEntryBlock();
for (BasicBlock *BB : post_order(&EntryBB)) {
  BB->printAsOperand(errs());
  errs() << "\n";
}
```

将获得与前面代码段相同的结果，因为它是从入口块开始运行的。不过，开发者可以自由地从任意块开始遍历。

深度优先和宽度优先遍历

DFS 和 **BFS** 是访问拓扑结构 (如图或树) 两个最著名和最具标志性的算法。对于树或图中的每个节点，DFS 总是在访问共享相同父节点 (即兄弟节点) 的其他节点之前，尝试访问其子节点。另一方面，BFS 将在移动到其子节点之前遍历所有兄弟节点。

LLVM 提供 df_iterator 和 bf_iterator(以及其他一些实用函数) 来分别实现深度优先和宽度优先的排序。由于用法几乎相同，我们只在这里演示 df_iterator:

```
#include "llvm/ADT/DepthFirstIterator.h"
#include "llvm/IR/CFG.h"
// `F` has the type of `Function*`
for (BasicBlock *BB : depth_first(F)) {
```

```
  BB->printAsOperand(errs());
  errs() << "\n";
}
```

po_iterator 和 post_order 类似，depth_first 只是一个函数，用于创建 df_iterator 的迭代范围。要使用 bf_iterator，只需将 depth_first 替换为 breadth_first。如果把上面的代码应用到上图中包含的分支上，会给获得如下的命令行输出:

```
label %1
label %3
label %5
label %9
label %12
label %7
label %10
```

当使用 bf_iterator/breadth_first 时，将获得以下命令行输出:

```
label %1
label %3
label %10
label %5
label %7
label %12
label %9
```

df_iterator 和 bf_iterator 也可以和 BasicBlock 一起使用，方法和前面的 po_iterator 一样。

SSC 遍历

SCC 表示一个子图，其中每个外围节点都可以从其他节点到达。在 CFG 的上下文中，用循环遍历 CFG 是很有用的。

我们前面介绍的基本块遍历方法，是分析函数中控制流的工具。对于一个无循环的函数，这些方法提供了一个线性视图，反映了封闭的基本块的执行顺序。但是，对于包含循环的函数，这些 (线性) 遍历方法不能显示由循环创建的“循环执行流”。

重复控制流

循环并不是在函数中创建循环控制流的唯一编程构造。其他一些指令——C/C++ 中的 goto 语法——也将引入循环控制流。然而，这些极端情况会使分析控制流变得更加复杂 (这也是不应该在代码中使用 goto 的原因之一)，所以当我们谈论循环控制流时，只指循环。

使用 LLVM 中的 scc_iterator，我们可以遍历 CFG 中强连接的基本块。有了这些信息，可以快速地发现循环的控制流，这对于一些分析和程序转换任务至关重要，例如：需要知道后面的边和循环的基本块，以便沿着控制流的边准确地传播分支概率数据。

下面是一个使用 scc_iterator 的例子:

```
#include "llvm/ADT/SCCIterator.h"
#include "llvm/IR/CFG.h"
// `F` has the type of `Function*`
for (auto SCCI = scc_begin(&F); !SCCI.isAtEnd(); ++SCCI) {
  const std::vector<BasicBlock*> &SCC = *SCCI;
  for (auto *BB : SCC) {
    BB->printAsOperand(errs());
    errs() << "\n";
  }
  errs() << "====\n";
}
```

与前面的遍历方法不同,scc_iterator 没有提供方便的范围式迭代。相反,需要使用 scc_begin 创建 scc_iterator 实例，并手动递增。更重要的是，应该使用 isAtEnd 方法来检查退出条件，而不是像我们通常在 C++ STL 容器中做的那样与“end”迭代器进行比较。BasicBlock 的 vector 对象可以从单个 scc_iterator 对象中进行解引用。这些 BasicBlock 实例是 SCC 中的基本块。这些实例之间的顺序与拓扑顺序反向后的顺序大致相同——也就是我们前面看到的后序。

如果在上图中包含一个循环的函数上运行上述代码，会获得如下的命令行输出:

```
label %6
====
label %4
label %2
====
label %1
====
```

这表明基本块%4 和%2 在同一个 SCC 中。

在此基础上，了解了如何以不同的方式迭代函数中的基本块。

下一节中，我们将了解如何通过调用图来迭代模块中的函数.

10.2.4 迭代调用图

调用图是表示模块中函数调用关系的直接图，在**过程间**代码转换和分析中扮演着重要的角色，即跨多个函数分析或优化代码。**函数内联**的优化就是一个例子。

在深入研究调用图中迭代节点的细节之前，让我们先看看如何构建调用图。LLVM 使用 `CallGraph` 类来表示单个模块的调用图。下面的示例代码使用了一个 Pass 模块来构建 `CallGraph`:

```
#include "llvm/Analysis/CallGraph.h"
struct SimpleIPO : public PassInfoMixin<SimpleIPO> {
  PreservedAnalyses run(Module &M, ModuleAnalysisManager &MAM)
  {
    CallGraph CG(M);
    for (auto &Node : CG) {
      // Print function name
      if (Node.first)
      errs() << Node.first->getName() << "\n";
    }
    return PreservedAnalysis::all();
  }
};
```

这个代码在遍历所有函数，并打印它们的名称之前构建了一个 `CallGraph` 实例。

就像 `Module` 和 `Function` 一样，`CallGraph` 只提供最基本的方法来枚举所有组件。那么，我们如何以不同的方式遍历 `CallGraph`——通过使用 SCC——正如前一节中看到的那样？这个问题的答案非常简单: 以完全相同的方式——使用相同的 API 集和相同的用法。

这“幕后黑手”是一个叫做 GraphTraits 的东西。

10.2.5 了解 GraphTraits

`GraphTraits` 是一个类，旨在为 LLVM-CFG 中的各种不同的图和调用图提供抽象接口，允许其他 LLVM 组件 (如我们在前一节中看到的分析、转换或迭代器实用程序) 独立于底层图构建它们的工作。与要求 LLVM 中的每个图都继承 `GraphTraits`，并实现所需的函数不同，`GraphTraits` 采用了一种完全不同的方法，即使用**模板特化**。

假设你写了一个简单的 C++ 类，它有一个接受任意类型的模板参数，如下所示:

```
template <typename T>
struct Distance {
  static T compute(T &PointA, T &PointB) {
    return PointA - PointB;
  }
};
```

这个 C++ 类将在调用 `distance::compute` 方法时，会计算两点之间的距离。这些点的类型由 `T` 模板参数参数化。

如果 `T` 是数字类型，比如: `int` 或 `float`，那么一切都没问题。但是，如果 `T` 是类的一个结构体，那么默认的 `compute` 方法实现将无法编译:

```
Distance<int>::compute(94, 87); // Success
```

```
…
struct SimplePoint {
  float X, Y;
};
SimplePoint A, B;
Distance<SimplePoint>::compute(A, B); // Compilation Error
```

为了解决这个问题，可以为 SimplePoint 实现一个减法运算符，或者可以使用模板特化，如下所示:

```
// After the original declaration of struct Distance…
template<>
struct Distance<SimplePoint> {
  SimplePoint compute(SimplePoint &A, SimplePoint &B) {
    return std::sqrt(std::pow(A.X - B.X, 2),…);
  }
};
…
SimplePoint A, B;
Distance<SimplePoint>::compute(A, B); // Success
```

前面的代码中,Distance<SimplePoint> 描述了当 T 等于 SimplePoint 时距离 Distance<T> 的样子。可以将原始的 Distance<T> 看作某种接口，而 Distance<SimplePoint>> 则是其**实现**之一。但是请注意，Distance<SimplePoint>> 中的计算方法不是 Distance<T> 中的原始计算的覆写方法。这不同于普通的类继承 (和虚方法)。

LLVM 中的 GraphTraits 是一个模板类，为各种图形算法提供了接口，比如：df_iterator 和 scc_iterator。LLVM 中的每个图都将通过模板特化实现这个接口，例如：下面的 GraphTraits 特化，用来建模一个函数的 **CFG**:

```
template<>
struct GraphTraits<Function*> {…}
```

GraphTraits<Function*> 的主体中，有几个 (静态) 方法和 typedef 语句实现了所需的接口，例如：nodes_iterator 是用于遍历 CFG 中的所有顶点的类型，而 nodes_begin 提供了这个 CFG 的入口/起始节点:

```
template<>
struct GraphTraits<Function*> {
  typedef pointer_iterator<Function::iterator> nodes_iterator;
  static node_iterator nodes_begin(Function *F) {
    return nodes_iterator(F->begin());
  }
  …
};
```

本例中，nodes_iterator 就是 Function::iterator。node_begin 只返回函数中的第一个基本块 (通过迭代器)。如果我们了解 CallGraph 的 GraphTraits，它有着对 nodes_iterator 和 nodes_begin 完全不同的实现:

```
template<>
struct GraphTraits<CallGraph*> {
  typedef mapped_iterator<CallGraph::iterator,
  decltype(&CGGetValuePtr)> nodes_iterator;
  static node_iterator nodes_begin(CallGraph *CG) {
    return nodes_iterator(CG->begin(), &CGGetValuePtr);
  }
};
```

当开发人员实现一种新的图形算法时，他们可以使用 GraphTraits 作为接口来访问任意图形的关键属性 (而不是在 LLVM 中为每种图形硬编码算法)。

例如，想创建一个新的图算法 find_tail，它找到图中没有子节点的第一个节点。下面是 find_tail 的框架:

```
template<class GraphTy,
typename GT = GraphTraits<GraphTy>>
auto find_tail(GraphTy G) {
  for(auto NI = GT::nodes_begin(G); NI != GT::nodes_end(G);
  ++NI) {
    // A node in this graph
    auto Node = *NI;
    // Child iterator for this particular node
    auto ChildIt = GT::child_begin(Node);
    auto ChildItEnd = GT::child_end(Node);
    if (ChildIt == ChildItEnd)
      // No child nodes
      return Node;
  }
  ...
}
```

在这个模板和 GraphTraits 的帮助下，我们可以在 Function、CallGraph 或 LLVM 中的任何图形上重用这个函数，例如:

```
// `F` has the type of `Function*`
BasicBlock *TailBB = find_tail(F);
// `CG` has the type of `CallGraph*`
CallGraphNode *TailCGN = find_tail(CG);
```

简而言之，GraphTraits 使用模板特化技术将算法——df_iterator 和 scc_iterator——在 LLVM 中推广到任意图形算法。这是为可重用组件定义接口的一种干净而有效的方式。

本节中，我们蓼莪及了 LLVM IR 的层次结构，以及如何迭代不同的 IR 单元——无论是具体的还是逻辑的单元，比如：CFG。我们还了解习了 GraphTraits 在封装不同的图 (CFG 和调用图) 方面的重要作用，并为 LLVM 中的各种算法提供了一个通用接口，从而使这些算法更加简洁和可重用。

下一节中，我们将了解值是如何在 LLVM 中表示的，不同的 LLVM IR 组件是如何相互关联的。此外，我们还将了解如何在 LLVM 中正确而有效地操作和更新值。

10.3. 值和指令

在 LLVM 中，值是有独特的构造——不仅表示存储在变量中的值，还模拟了从常量、全局变量、单个指令，甚至基本块等广泛的概念。换句话说，它是 LLVM IR 的基础。

值的概念对于指令特别重要，因为它直接与 IR 中的值交互。因此，在本节将对它们进行相同的讨论。我们将了解值在 LLVM IR 中是如何工作的，以及值是如何与指令相关联的。在此之上，我们将了解如何创建和插入新的指令，以及如何更新它们。

为了学习如何在 LLVM IR 中使用值，我们必须理解这个系统背后的重要理论，它规定了 LLVM 指令的行为和格式——单一静态赋值 (SSA) 形式。

10.3.1 SSA

SSA 是一种构造和设计 IR 的方法，可以使程序分析和编译器转换更容易执行。在 SSA 中，一个变量 (在 IR 中) 只会赋值一次。这意味着不能像这样操作一个变量:

```
// the following code is NOT in SSA form
x = 94;
x = 87; // `x` is assigned the second time, not SSA!
```

尽管一个变量只能被赋值一次，但它可以在任意指令中多次使用:

```
x = 94;
y = x + 4; // first time `x` is used
z = x + 2; // second time `x` is used
```

读者们可能想知道普通的 C/C++ 代码 (显然不是 SSA 形式) 如何转换成 SSA 形式的 IR，比如：LLVM。虽然有很多不同的算法和研究论文可以回答这个问题，但我们不打算在这里讨论，大多数简单的 C/C++ 代码都可以通过一些简单的技术进行转换，比如：重命名。例如，假设我们有以下 (非 SSA)C 代码:

```
x = 94;
x = x * y; // `x` is assigned more than once, not SSA!
x = x + 5;
```

这里，我们可以在第一个赋值语句中重命名 `x`，在第二个和第三个赋值语句的左边分别使用 `x0` 和 `x` 这样的名称，例如：`x1` 和 `x2`:

```
x0 = 94;
x1 = x0 * y;
x2 = x1 + 5;
```

通过这些简单的技巧，我们可以获得具有相同行为的原始代码的 SSA。

为了对 SSA 有一个更全面的理解，必须改变我们对程序中指令的思考方式。在**命令式编程语言** (如 C/C++) 中，经常将每个语句 (指令) 视为一个**动作**，例如：在下面的图中，在左边，第一行表示“将 94 赋给变量 x”的操作，第二行表示“在将结果存储到变量 x 之前，使用 x 和 y 做一些乘法”:

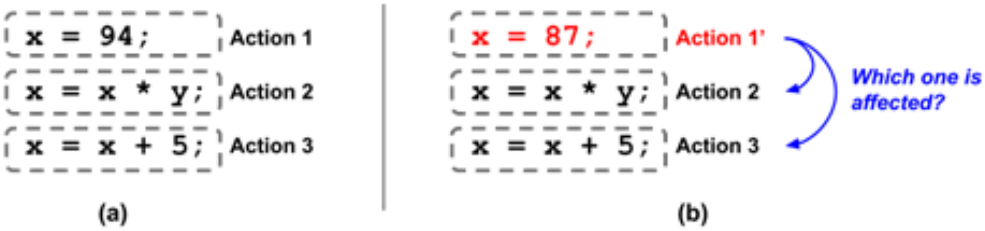

图 10.4 - 将指令视为“动作”

这些解释听起来很直观。然而，当我们对这些指令进行一些转换时 (当然，这是编译器中常见的事情)，事情就变得棘手了。在图中，在右边，当第一条指令变成 `x = 87` 时，我们不知道这个修改是否会**影响**其他指令。如果会影响，我们还不确定哪一个指令会受到影响。这个信息不仅告诉我们是否有其他潜在的优化机会，而且它也是编译器转换正确性的关键因素——毕竟，没有人希望在启用优化时编译器会破坏原始的代码。更糟糕的是，假设我们要检查**动作 3** 右边的 `x` 变量。我们感兴趣的是修改这个 `x` 的最后几条指令。在这里，我们只能列出所有左边有 `x` 的指令 (也就是说，使 `x` 作为目标)，这非常的低效。

我们可以把注意力集中在指令所产生的数据上，从而清楚地了解每条指令的来源——也就是结果值可以到达的区域。此外，我们可以很容易地找到任意变量/值的原点。下图说明了这样做的优势:

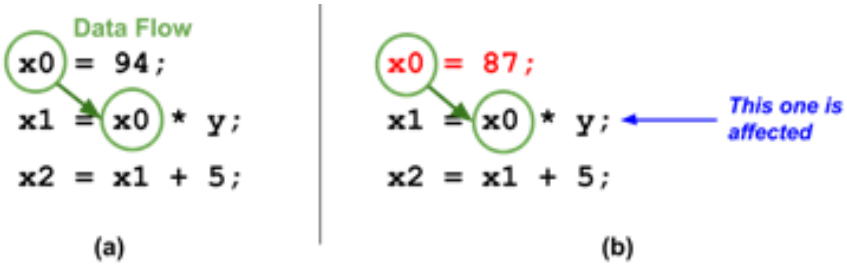

图 10.5 - SSA 显示指令之间的数据流

换句话说，SSA 显示程序中的**数据流**，以便编译器更容易地跟踪、分析和修改指令。

LLVM 中的指令以 SSA 形式组织。这意味着我们更感兴趣的是由指令生成的值或数据流，而不是将结果存储在哪个变量中。因为 LLVM IR 中的每条指令只能产生一个结果值，`Instruction` 对象——`Instruction` 是在 LLVM IR 中表示一条指令的 C++ 类——也可以表示它的**结果值**。更具体地说，LLVM IR 中值的概念是由一个 C++ 类 `Value` 表示的。`Instruction` 是它的子类。这意味着给定一个 `Instruction` 对象，当然，我们可以将它强制转换为一个 `Value` 对象。这个特定的 `Value` 对象实际上是该 `Instruction` 的结果:

```
// let's say `I` represents an instruction `x = a + b`
Instruction *I = ⋯;
Value *V = I; // `V` effectively represents the value `x`
```

这是使用 LLVM IR 最重要的事情之一，特别是在使用它 API 时。

虽然 `Instruction` 对象表示它自己的结果值，但也有作为指令输入的操作数。你猜怎么着？我们也使用 `Value` 对象作为操作数。例如，有以下代码:

```
Instruction *BinI = BinaryOperator::Create(Instruction::Add,…);
Instruction *RetI = ReturnInst::Create(…, BinI, …);
```

前面的代码片段基本上创建了一个算术加法指令 (由 BinaryOperator 表示)，它的结果值将是另一个返回指令的操作数。生成的 IR 等价于下面的 C/C++ 代码:

```
x = a + b;
return x;
```

除了 Instruction 之外，Constant(C++ 中用于不同类型常量的类)、GlobalVariable(C++ 中用于全局变量的类) 和 BasicBlock 都是 Value 的子类。这意味着它们也是以 SSA 的形式组织的，并且可以将它们用作指令的操作数。

现在，了解了 SSA 是什么，并了解了它对 LLVM IR 设计的影响。下一节中，我们将讨论如何修改和更新 LLVM IR 中的值。

10.3.2 值

SSA 使我们关注指令之间的数据流。由于我们对值如何从一条指令传递到另一条指令有了清晰的认识，因此很容易在指令中替换某些值的使用。但“值”的概念在 LLVM 中是如何体现的呢？下面的图表显示了两个回答这个问题的重要 C++ 类——User 和 Use:

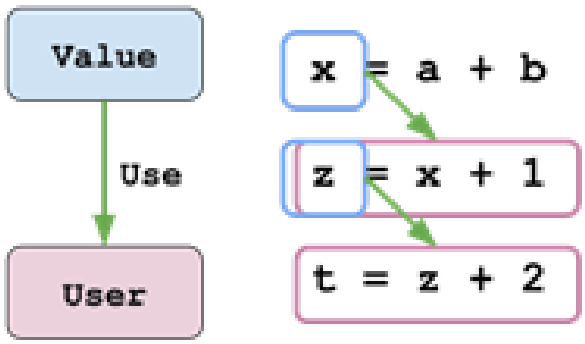

图 10.6 - Value、User 和 Use 之间的关系

User 代表了一个 IR 实例 (例如，一条指令) 的概念，它使用了一个特定的 Value。此外，LLVM 还使用了另一个类 Use 来创建 Value 和 User 之间的边界。Instruction 是 Value 的子类——Value 表示由该 Instruction 生成的结果。事实上，Instruction 也是从 User 派生出来的，因为所有的 Instruction 都至少有一个操作数。

多个 Use 实例可能指向一个 User，这意味着 User 使用了多个 Value 实例。可以使用 User 提供的 value_op_iterator 来检索每个 Value 实例，例如:

```
// `Usr` has the type of `User*`
for (Value *V : Usr->operand_values()) {
  // Working with `V`
}
```

同样，operand_values 只是一个实用函数，用于生成 value_op_iterator 的范围。

下面是我们为什么要遍历 Value 的所有 User 实例的一个例子: 假设我们正在分析一个程序，其中一个 Function 实例将返回敏感信息——比如，一个 get_password 函数。我们的目标是确保每当在函数中调用 get_password 时，返回值 (敏感信息) 不会通过另一个函数调用而遭到泄露。例如，我们想要检测以下的模式，并发出警报:

```
void vulnerable() {
  v = get_password();
  ...
  bar(v); // WARNING: sensitive information leak to `bar`!
}
```

实现此分析的最常用的原生方法是，检查敏感 Value 的所有 User 实例。下面是示例代码:

```
User *find_leakage(CallInst *GetPWDCall) {
  for (auto *Usr : GetPWDCall->users()) {
    if (isa<CallInst>(Usr)) {
      return Usr;
    }
  }
...
}
```

find_leaks 函数接受一个 CallInst 参数——表示调用 get_password 函数——并返回任何使用该 get_password 调用返回的 Value 实例的 User 实例。

一个 Value 实例可以被多个不同的 User 实例使用。类似地，我们可以使用下面的片段来遍历它们:

```
// `V` has the type of `Value*`
for (User *Usr : V->users()) {
  // Working with `Usr`
}
```

至此，我们了解了如何检查 Value 的 User 实例，或者当 User 使用的 Value 实例。此外，当我们开发编译器转换时，将 User 使用的 Value 实例更改为另一个实例是很常见的操作。LLVM 提供了一些方便的工具来完成这项工作。

首先，Value::replaceAllUsesWith 方法可以，正如它的名字所示的那样，告诉它的所有 User 实例使用另一个 Value 来代替原始的。下图说明了它的效果:

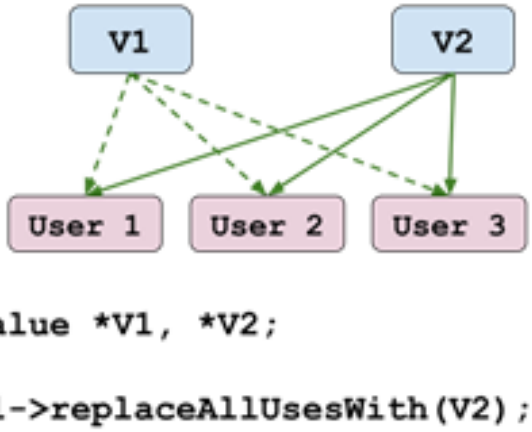

图 10.7 - Value::replaceAllUsesWith 的效果

当用另一个 Instruction 替换相应 Instruction 时，这个方法非常有用。使用前面的图表来解释这一点，V1 是原来的 Instruction，V2 是新 Instruction。

另一个实现类似功能的函数是 User::replaceUsesOfWith(From,To)。此方法有效地扫描 User 中的所有操作数，并将特定 Value(From 参数) 的使用替换为另一个 Value(To 参数)。

在本节中学习的技能是在 LLVM 中开发程序转换的一些基本工具。下一节中，我们将讨论如何创建和修改指令。

10.3.3 指令

之前，我们了解了 Value 的基础知识——包括与指令的关系——以及在 SSA 框架下更新 Value 实例的方法。本节中，我们将了解一些更基本的知识和技能，可以更好地理解指令，并以正确和有效的方式修改指令实例，这是开发一个成功编译器优化的关键。

下面是我们将在本节讨论的内容:

- 不同指令类型之间的强制转换
- 插入新指令
- 更换指令
- 批量处理指令

让我们从不同的指令类型开始。

不同指令类型之间的强制转换

前一节中，我们了解了一个名为 InstVisitor 的程序，InstVisitor 类可以确定指令实例的底层类。它还省去了在不同指令类型之间进行强制转换的工作。然而，我们不能总是依赖 InstVisitor 来处理每一个涉及到指令，及其派生类之间类型转换的任务。更简单地说，我们想要一个父类和子类之间类型转换的简单解决方案。

C++ 已经通过 dynamic_cast 指令提供了这种机制，对吗？下面是一个 dynamic_cast 的例子:

```
1 class Parent {…};
2 class Child1 : public Parent {…};
```

```
class Child2 : public Parent {···};

void foo() {
  Parent *P = new Child1();
  Child1 *C = dynamic_cast<Child1*>(P); // OK
  Child2 *O = dynamic_cast<Child2*>(P); // Error: bails out at
  // runtime
}
```

在代码中使用的 foo 函数中，可以在它的第二行中看到，可以将 P 转换为一个 Child1 实例，因为这是它的底层类型。另一方面，我们不能将 P 转换成 Child2——如果这样做，程序会在运行时崩溃。

实际上，dynamic_cast 具有我们正在寻找的确切功能——更正式地说，是**运行时类型信息 (RTTI)** 特性——但它在运行时性能方面也有很高的开销。糟糕的是，C++ 中 RTTI 的默认实现相当复杂，使得生成的程序难以优化。因此，LLVM 默认关闭 RTTI 功能。由于这个原因，LLVM 提出了自己的运行时类型转换系统，这个系统更加简单和高效。本节中，我们将讨论如何使用它。

LLVM 的类型转换框架为动态类型转换提供了三个功能:

- isa<T>(val)
- cast<T>(val)
- dyn_cast<T>(val)

第一个函数 isa<T>——发音为"is-a"——检查 val 指针类型是否可以转换为 T 类型的指针。下面是一个例子:

```
// `I` has the type of `Instruction*`
if (isa<BinaryOperator>(I)) {
  // `I` can be casted to `BinaryOperator*`
}
```

注意，与 dynamic_cast 不同，在这种情况下，不需要将 BinaryOperator* 作为模板参数——只需要一个没有指针限定符的类型。

cast<T> 函数执行从 (指针类型)val 到 T 类型指针的实类型转换。下面是一个例子:

```
// `I` has the type of `Instruction*`
if (isa<BinaryOperator>(I)) {
  BinaryOperator *BinOp = cast<BinaryOperator>(I);
}
```

同样，不需要将 BinaryOperator* 作为模板参数。注意，如果在调用 cast<T> 前没有使用 isa<T> 执行类型检查，程序将在运行时崩溃。

最后一个函数 dyn_cast<T>，是 isa<T> 和 cast<T> 的组合。如果适用，可以执行类型转换。否则，返回 null。下面是一个例子:

```
// `I` has the type of `Instruction*`
if (BinaryOperator *BinOp = dyn_cast<BinaryOperator>(I)) {
  // Work with `BinOp`
}
```

这里，我们可以看到一些简洁的语法，将变量声明 (BinOp) 与 if 语句结合在一起。

注意，这些 API 都不能以 null 作为参数。相反，dyn_cast_or_null<T> 则没有这个限制，是一个接受 null 作为输入的 dyn_cast<T> 的 API。

现在，了解了如何检查任意指令实例，并将其转换为其基础指令类型。从下一节开始，我们将创建和修改一些指令。

插入新指令

在之前的理解 SSA 部分的一个代码示例中，我们看到了这样一段代码:

```
Instruction *BinI = BinaryOperator::Create(…);
Instruction *RetI = ReturnInst::Create(…, BinI, …);
```

正如方法名 Create 所示,我们可以推断这两行创建了一个 BinaryOperator 和一个 returnst 指令。

LLVM 中的大多数指令类都提供了工厂方法——例如这里的 Create——来构建一个新的实例。LLVM 鼓励使用这些工厂方法，而不是通过 new 关键字或 malloc 函数手动分配指令对象。将指令插入到 BasicBlock 中时，LLVM 将为你管理指令对象的内存。有几种方法可以将一条新指令插入到 BasicBlock 中:

- 某些指令类中的工厂方法提供了一个选项，可以在指令创建后立即插入指令，例如：Binary Operator 中的 Create 方法变体允许将其插入创建之后的另一个指令之前。下面是一个例子:

  ```
  Instruction *BeforeI = …;
  auto *BinOp = BinaryOperator::Create(Opcode, LHS, RHS,
    "new_bin_op", BeforeI);
  ```

  在这种情况下，BinOp 表示的指令将放置在 BeforeI 表示的指令之前。然而，这个方法不能在不同的指令类之间移植。并不是每个指令类都有提供此特性的工厂方法，即使它们提供了这些方法，API 也可能不相同。
- 我们可以使用 Instruction 类提供的 insertBefore/insertAfter 方法来插入新指令。由于所有的指令类都是 Instruction 的子类，我们可以使用 insertBefore 或 insertAfter 将新创建的指令实例插入到另一个指令之前或之后。
- 我们也可以使用 IRBuilder 类，他是一个强大的工具，可以自动化一些指令创建和插入步骤，实现了一个构建器设计模式。当开发人员调用它的一个创建方法时，可以逐个地插入新的指令:

  ```
  // `BB` has the type of `BasicBlock*`
  IRBuilder<> Builder(BB /*the insertion point*/);
  // insert a new addition instruction at the end of `BB`
  auto *AddI = Builder.CreateAdd(LHS, RHS);
  // Create a new `ReturnInst`, which returns the result
  // of `AddI`, and insert after `AddI`
  Builer.CreateRet(AddI);
  ```

首先，创建一个 IRBuilder 实例时，需要指定一个插入点作为构造函数参数。这个插入点参数可以是 BasicBlock，这意味着我们想在 BasicBlock 的末尾插入一条新指令，也可以是一个 Instruction 实例，这意味着新的指令将插入到那个特定的 Instruction 之前。
如果可能，当需要按顺序创建和插入新的指令时，推荐使用 IRBuilder。

至此，我们已经了解了如何创建和插入新的指令。现在，让我们看看如何用其他指令替换现有的指令。

替换指令

有时，我们需要替换现有的指令，例如：当一个乘法操作数是 2 的幂整数常数时，一个简单的优化器可以用左移指令替换算术乘法指令。本例中，我们可以简单地通过改变原始 Instruction 中的操作符 (操作码) 和操作数来实现。然而，这并不是推荐的方式。

要替换 LLVM 中的指令，需要创建一个新的 Instruction(作为替换指令)，并将所有 SSA 定义和用法从原来的 Instruction 变更到替换的指令。让我们用 2 次幂乘法作为例子:

1. 我们要实现的函数叫做 replacePow2Mul，它的参数是要处理的乘法指令 (假设已经确保乘法具有一个常数，2 的幂次整数操作数)。首先，我们将检索常量整数 (由 ConstantInt 类表示) 的操作数，并将其转换为其以 2 为底的对数值 (通过 getLog2 函数，具体实现留给读者们做为练习):

```
void replacePow2Mul(BinaryOperator &Mul) {
  // Find the operand that is a power-of-2 integer
  // constant
  int ConstIdx = isa<ConstantInt>(Mul.getOperand(0))? 0
    : 1;
  ConstantInt *ShiftAmount = getLog2(Mul.
    getOperand(ConstIdx));
}
```

2. 接下来，我们将创建一个新的左移指令——由 ShlOperator 类表示:

```
void replacePow2Mul(BinaryOperator &Mul) {
  ...
  // Get the other operand from the original instruction
  auto *Base = Mul.getOperand(ConstIdx? 0 : 1);
  // Create an instruction representing left-shifting
  IRBuilder<> Builder(&Mul);
  auto *Shl = Builder.CreateShl(Base, ShiftAmount);
}
```

3. 最后，在删除 Mul 指令之前，我们需要告诉原始 Mul 的所有用户使用新创建的 Shl:

```
void replacePow2Mul(BinaryOperator &Mul) {
  ...
  // Using `replaceAllUsesWith` to update users of `Mul`
  Mul.replaceAllUsesWith(Shl);
  Mul.eraseFromParent(); // remove the original
  // instruction
}
```

现在，所有 Mul 的原始用户都改用了 Shl。这样，我们就可以安全地将 Mul 从程序中移除了。

这样，我们了解了如何正确地替换现有的指令。在最后的小节中，我们将讨论在 BasicBlock 或函数中处理多个指令的技巧。

批量处理指令

我们已经了解了如何插入、删除和替换一个 Instruction。然而，在实际中，我们通常在一系列 Instruction 实例 (例如，在 BasicBlock 中) 上执行这样的操作。让我们试着把我们学过的东西放到一个 for 循环中它会遍历 BasicBlock 中的所有指令，例如:

```
// `BB` has the type of `BasicBlock&`
for (Instruction &I : BB) {
  if (auto *BinOp = dyn_cast<BinaryOperator>(&I)) {
    if (isMulWithPowerOf2(BinOp))
    replacePow2Mul(BinOp);
  }
}
```

前面的代码使用了我们在前一节中的 replacePow2Mul 函数，如果乘法满足某些条件，则用左移指令替换 BasicBlock 中的乘法。(isMulWithPowerOf2 函数进行了检查。同样，这个函数的具体实现，就留给读者们作为练习。)

这段代码看起来非常简单，但不幸的是，它会在运行转换时崩溃。在运行了的 replacePow2Mul 之后，用于枚举 BasicBlock 中的 Instruction 实例的**迭代器**就过时了。Instruction 迭代器无法更新这个 BasicBlock 中已经应用到 Instruction 实例的更改。换句话说，在迭代 Instruction 实例时更改它们非常困难。

解决这个问题的最简单的方法是延迟更改:

```
// `BB` has the type of `BasicBlock&`
std::vector<BinaryOperator*> Worklist;
// Only perform the feasibility check
for (auto &I : BB) {
  if (auto *BinOp = dyn_cast<BinaryOperator>(&I)) {
    if (isMulWithPowerOf2(BinOp)) Worklist.push_back(BinOp);
  }
}
// Replace the target instructions at once
for (auto *BinOp : Worklist) {
  replacePow2Mul(BinOp);
}
```

前面的代码将前面的代码示例分为两部分 (作为两个独立的 for 循环)。第一个 for 循环仍然在遍历 BasicBlock 中的所有指令实例。但这一次，它只执行检查 (也就是说，调用 isMulWithPowerOf2)，而不会在通过检查后立即替换指令实例。相反，这个 for 循环将候选指令推送到数组存储中——一个工作列表。在完成第一个 for 循环之后，第二个 for 循环会检查工作列表，并通过对每个工作列表项调用 replacePow2Mul 来执行替换。由于第二个 for 循环中的替换不会使任何迭代器失效，所以最终可以在不发生任何崩溃的情况下转换代码。

当然，还有其他方法可以避免前面提到的迭代器问题，但它们大多比较复杂，可读性也较差。使用工作列表是批量修改指令最安全、最具表现力的方式。

`Value` 是 LLVM 中的一级构造，它描述了不同实体 (如指令) 之间的数据流。本节中，我们介绍了如何在 LLVM IR 中表示值，以及 SSA 模型，该模型使分析和转换变得更容易。我们还学习了如何以有效的方式更新值，以及一些操作指令的有用技能。这将帮助我们为使用 LLVM 构建更复杂和高级编译器优化打下基础。

下一节中，我们将研究一个稍微复杂一点的 IR 单元——循环。我们将学习如何在 LLVM IR 中表示循环，以及如何使用。

10.4. 循环

到目前为止，我们已经学习了几个 IR 单元，如模块、功能、基本模块和指令。我们还学习了一些逻辑单元，如 CFG 和调用图。本节中，我们将看看一个更符合逻辑的 IR 单元: 循环。

循环是程序员经常使用的结构，更不用说几乎每一种编程语言也包含这个概念。循环重复执行一定数量的指令多次，这为程序员节省了大量自己重复代码的工作。但是，如果循环包含任何效率低下的代码——耗时的内存负载总是传递相同的值——性能下降也会随着迭代的次数而放大。

因此，编译器的工作就是从循环中消除尽可能多的缺陷。除了从循环中移除次优代码，因为循环在运行时性能的关键路径上，人们一直试图通过特殊的基于硬件的加速来进一步优化它们，例如：用向量指令替换循环，可以在几个周期内处理多个标量值。简而言之，循环优化是生成更快、更高效程序的关键。这在高性能和科学计算社区中尤为重要。

本节中，我们将学习如何使用 LLVM 处理循环。我们将尝试通过两个部分来解决这个问题:

- LLVM 中的循环表示
- LLVM 中的循环设施

在 LLVM 中，循环比其他 (逻辑)IR 单元稍微复杂一些。因此，我们将首先了解 LLVM 中循环的高级概念及其术语。然后，第二部分中，我们将接触 LLVM 中用于处理循环的设施和工具。

让我们从第一部分开始。

10.4.1 LLVM 中的循环表示

在 LLVM 中，循环由 `Loop` 类表示。这个类可以捕获任何控制流结构，该结构具有来自前块中的封闭基本块的后边 [译者：这里的描述有点绕，可以看一下图 10.8。]。在深入其细节之前，让我们先学习如何检索 `Loop` 实例。

在 LLVM IR 中，循环是一个逻辑 IR 单元。也就是说，是从物理 IR 单位推导 (或计算) 出来的。这样，我们需要从 `AnalysisManager` 中检索计算出的循环实例。下面是如何在 `Pass` 函数中检索 `Loop` 的例子:

```
#include "llvm/Analysis/LoopInfo.h"
…
PreservedAnalyses run(Function &F, FunctionAnalysisManager
&FAM) {
  LoopInfo &LI = FAM.getResult<LoopAnalysis>(F);
  // `LI` contains ALL `Loop` instances in `F`
```

```
  for (Loop *LP : LI) {
    // Working with one of the loops, `LP`
  }
...
}
```

`LoopAnalysis` 是一个 LLVM 分析类,为我们提供了一个 `LoopInfo` 实例,包含了一个 `Function` 中的所有 `Loop` 实例。我们可以遍历一个 `LoopInfo` 实例，以获得一个单独的 `Loop` 实例。

现在，让我们了解一下 `Loop` 实例。

循环术语

一个循环实例包含一个特定循环的多个 `BasicBlock` 实例。LLVM 为其中一些块以及它们之间的 (控制流) 边分配了一个特殊的含义/名称。下图显示了这个术语:

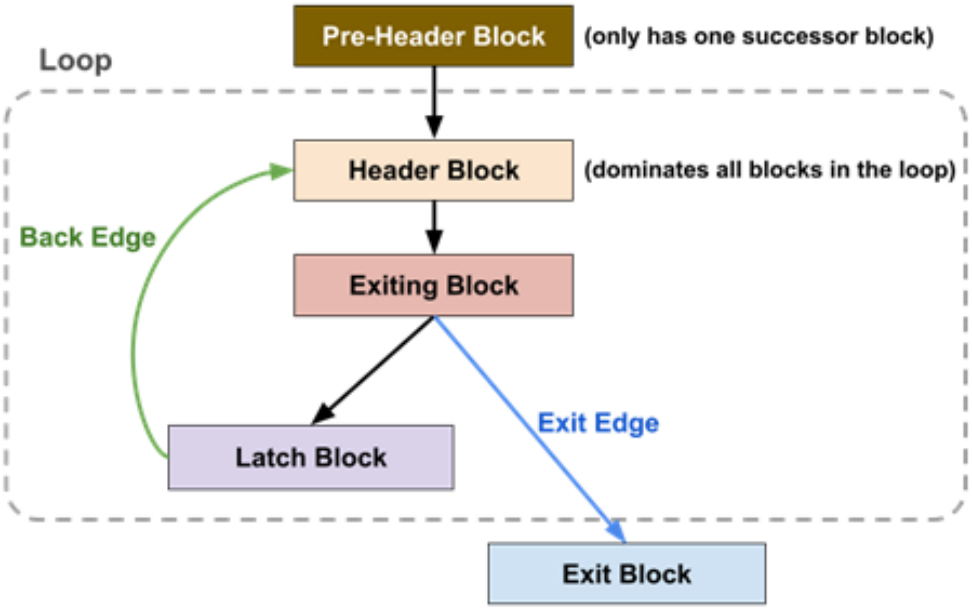

图 10.8 - 循环中的结构和术语

这里，每个矩形都是一个 `BasicBlock` 实例。但是，只有在虚线区域内的块才包含在 `Loop` 实例中。上图还显示了两个重要的控制流边。让我们详细解释这些术语:

- **Header Block**: 这个块标记了一个循环的入口。正式地说，它支配着循环中的所有块，也是 Back Edge 的目的地。
- **Pre-Header Block**: 虽然不是循环的一部分，但表示的是头块作为唯一的后续块。换句话说，它是 Header Block 的唯一前身。

  Pre-header 块的存在使得编写一些循环转换变得更容易，例如：当我们想要将一条指令提升到循环的外部，以便它在进入循环之前只执行一次时，Pre-header 块可是放置这个指令的好地方。如果没有 Pre-header 块，则需要为 Pre-header 块的每一个前身复制这条指令。
- **Back Edge**: 这是从循环中的一个块到报头块的控制流边。一个循环可能包含几个 Back Edge。

- **Latch Block**: 位于后边的源块。
- **Exiting Block 和 Exit Block**: 这两个名称有点令人困惑:Exiting Block 是具有控制流边 (Exit Edge) 的块，从而跳出循环。出口边的另一端，不属于循环的一部分，是 Exit Block。一个循环可以包含多个退出块 (和出口块)。

这些是循环实例中的块的重要术语。除了控制流结构之外，编译器工程师还对可能存在于循环中的一个特殊值感兴趣: **引导变量**。例如，在下面的代码片段中，变量 `i` 是引导变量:

```
for (int i = 0; i < 87; ++i){…}
```

循环可能不包含引导变量——C/C++ 中的许多 while 循环都没有引导变量。此外，要找出一个引导变量，以及它的边界 (开始、结束和停止值) 并不总是很容易。下一节中，我们将展示一些工具来帮助我们完成这项任务。但在那之前，我们将讨论一个有趣的话题，关于循环的标准形式。

规范循环

前一节中，我们了解了 LLVM 中循环的几个术语，包括 Pre-header 前块。Pre-header 块的存在使开发循环转换变得更容易，因为它创建了更简单的循环结构。在这个讨论之后，还有其他一些属性可以让我们更容易地编写循环转换。如果循环实例有这些漂亮的属性，我们通常称它为**规范循环**。LLVM 中的优化流水在将一个循环发送到任何循环转换之前，会尝试将其“更改”成这种规范形式。

目前，LLVM 有两种 `Loop` 形式: **简化**和**旋转**。简化后具有以下属性:

- 一个 Pre-header 块。
- 单一的 Back Edge(因此只有单个 Latch Block)。
- 退出块的前身来自循环。换句话说，Header 区块支配着所有的出口块。

要获得一个简化的循环，可以在原始循环上运行 `LoopSimplfyPass`。此外，可以使用 `Loop::isLoopSimplifyForm` 方法来检查是否存在 `Loop`。

拥有单一 Back Edge 的好处包括，可以更容易地分析递归数据流——引导变量。对于最后一个属性，如果每个出口块都由循环控制，那么可以更容易地在不受其他控制流路径干扰的情况下将指令“嵌入”到循环指令中。

让我们看看旋转后的形式。最初，在 LLVM 的循环优化管道中，旋转形式并不是正式的规范形式。但是随着越来越多的循环传递依赖于它，其就成为“事实上的”规范形式。下面的图表显示了旋转循环的样子:

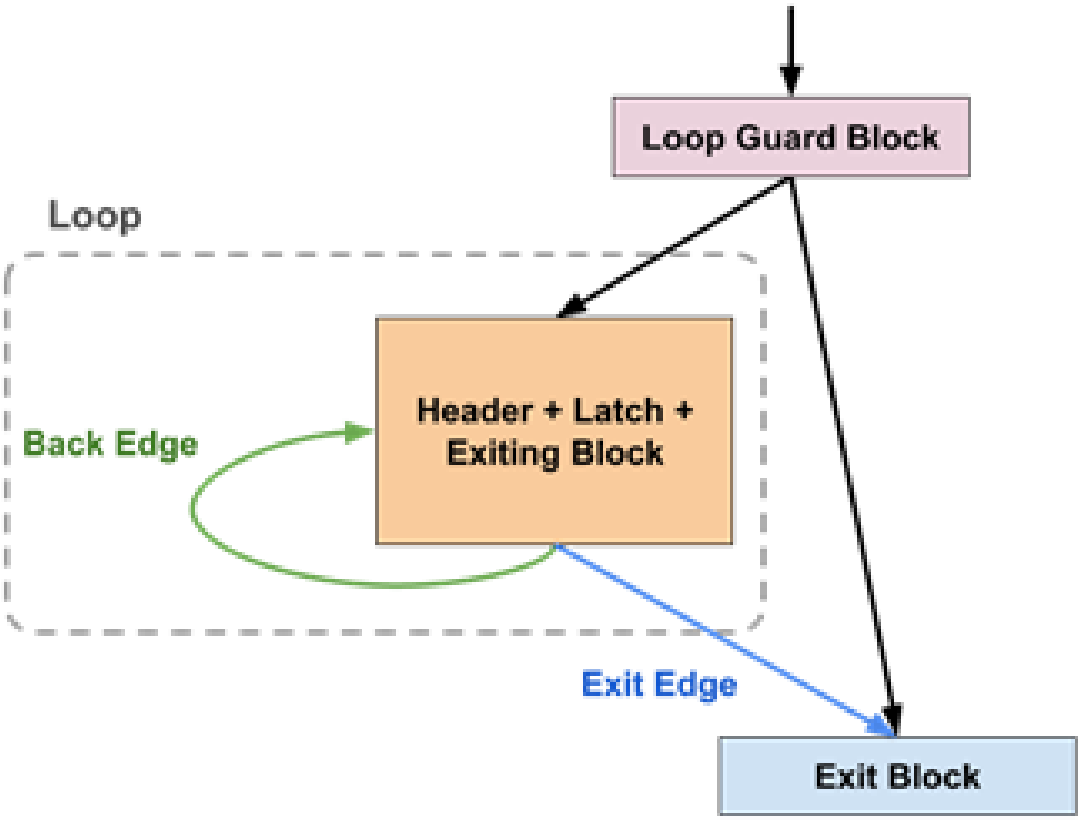

图 10.9 -旋转环路的结构和术语

要获得一个旋转的循环，可以在原始循环上运行 `LoopRotationPass`。要检查一个循环是否旋转，可以使用 `loop::isRotatedForm` 方法。

这种旋转后的形式基本上是，将任意循环转换为带有一些额外检查的 `do…while(…)` 循环 (在 C/C++ 中)。假设我们有以下 `for` 循环:

```
// `N` is not a constant
for (int i = 0; i < N; ++i){…}
```

循环旋转有效地将其转换为以下代码:

```
if (i < N) {
  do {
    …
    ++i;
  } while(i < N);
}
```

前面代码中显示的边界检查用于确保 `i` 变量在一开始就超出边界时，则不执行循环。我们也称其为检查**循环保护**，如上图所示。

除了循环保护，我们还发现旋转循环有一个合并的头、锁存器和退出块。这样做的基本原理是为了确保这个块中的每个指令都有相同的执行计数。这对于编译器优化 (如循环向量化) 是一个有用的属性。

在此基础上，我们了解了 LLVM 中的各种循环术语和规范循环的定义。在下一节中，我们将学习一些 API，可以帮助我们以一种有效的方式检查这些属性和处理循环。

10.4.2 LLVM 中的循环设施

在 LLVM 中了解循环表示的章节节中，我们学习了 LLVM IR 中循环的高层结构和重要属性。本节中，我们将看到有哪些 API 可用来检查这些属性，并进一步转换循环。让我们从循环传递开始讨论——应用于 Loop 实例的 LLVM Pass。

在第 9 章我们了解到有不同种类的 LLVM Pass 在不同的 IR 单元上工作——函数和模块的 Pass。这两种 Pass 的运行方法有一个类似的函数签名——LLVM Pass 的主要入口点——如下所示:

```
PreservedAnalyses run(<IR unit class> &Unit,
            <IR unit>AnalysisManager &AM);
```

所有 run 方法都有两个参数—对 IR 单元实例和 AnalysisManager 实例的引用。

相反，循环 Pass 有一个稍微复杂一点的 run 方法签名，如下所示:

```
PreservedAnalyses run(Loop &LP, LoopAnalysisManager &LAM,
            LoopStandardAnalysisResults &LAR,
            LPMUpdater &U);
```

run 方法有四个参数，但前两个我们已经知道了。以下是对另外两个的描述:

- 第三个参数，LoopStandardAnalysisResults 提供了一些分析数据实例，例如：AAResults(别名分析数据)、DominatorTree 和 LoopInfo。这些分析可以在许多循环优化中使用。然而，其中的大多数是由 FunctionAnalysisManager 或 ModuleAnalysisManager 管理。这意味着，开发人员需要实现更复杂的方法——使用 OuterAnalysisManagerProxy 类——来检索它们。LoopStandardAnalysisResults 实例可以帮助我们提前检索这些分析数据。
- 最后一个参数用于通知 PassManager 新添加的循环，以便可以在以后处理这些新循环之前，将它们放入队列中。它还可以告诉 PassManager 将当前循环再次放入队列中。

当我们编写一个 Pass 时，我们想要使用 AnalysisManager 提供的分析数据——在本例中，是 LoopAnalysisManager 实例。LoopAnalysisManager 与前一章中了解的 AnalysisManager(例如：FunctionAnalysisManager) 的其他版本有类似的用法。唯一的区别是，我们需要为 getResult 方法提供一个参数。下面是例子:

```
PreservedAnalyses run(Loop &LP, LoopAnalysisManager &LAM,
            LoopStandardAnalysisResults &LAR,
            LPMUpdater &U) {
  ...
  LoopNest &LN = LAM.getResult<LoopNestAnalysis>(LP, LAR);
  ...
}
```

LoopNest 是由 LoopNestAnalysis 生成的分析数据。(我们将在处理嵌套循环一节中讨论这两种情况。)

如前面的代码所示,LoopAnalysisManager::getResult 接受另一个 LoopStandarAnalysisResults 类型参数 (除了 Loop 实例之外)。

除了不同的签名和略微不同的 LoopAnalysisManager 用法外，开发人员可以用与其他类型的 Pass 相同的方式构建他们的循环 Pass。现在我们已经了解了循环 Pass 和 AnalysisManager 所提供的基础，现在是时候了解一些特殊的循环了。我们首先要介绍的是嵌套循环。

处理嵌套循环

到目前为止，一直在讨论只有一个层的循环。然而，嵌套循环——其中包含其他循环的循环——在实际场景中也很常见，例如：大多数矩阵乘法实现至少需要两层循环。

嵌套循环通常被描述为树——称为循环树。在循环树中，每个节点代表一个循环。如果一个节点有一个父节点，这意味着对应的循环被包含在父节点所创建的循环中。下面的图表展示了一个例子:

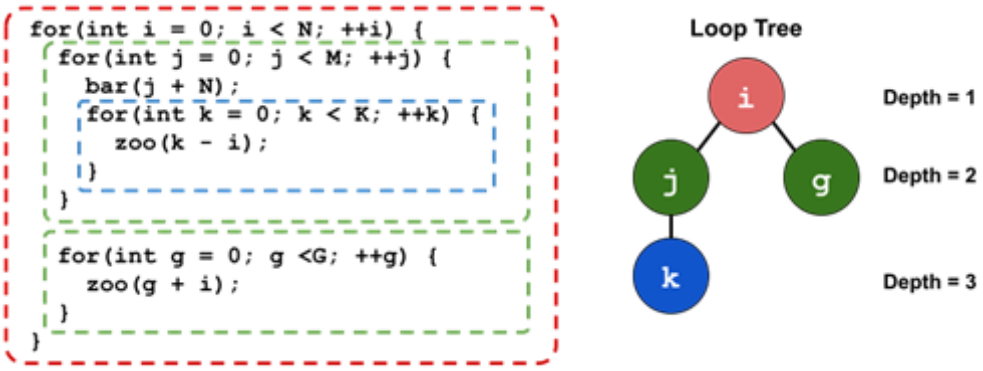

图 10.10 - 一棵循环树

上图中，循环 `j` 和 `g` 包围在循环 `i` 中，所以它们都是循环树中循环 `i` 的子节点。类似地，循环 `k`——最内部的循环——是树中循环 `j` 的子节点。

循环树的根也表示函数中的顶级循环。在前面，我们学习了如何通过遍历 `LoopInfo` 对象来检索函数中的所有 `Loop` 实例——以这种方式检索的每个 `Loop` 实例都是顶级循环。对于给定的 `Loop` 实例，我们可以以类似的方式在下一层检索它的子循环。下面是一个例子:

```
// `LP` has the type of `Loop&`
for (Loop *SubLP : LP) {
  // `SubLP` is one of the sub-loops at the next layer
}
```

注意，前面的代码只遍历了下一层的子循环，而不是所有的后代循环。要遍历树中的所有后代循环，有两个选项:

- 使用 `Loop::getLoopsInPreorder()` 方法，可以以预先排序的方式遍历一个 `Loop` 实例的所有后代循环。
- 迭代不同 IR 单元一节中，我们了解了什么是 `GraphTraits`，以及 LLVM 如何使用它来遍历图。LLVM 也为循环树提供了一个默认的 `GraphTraits` 实现。因此，可以使用 LLVM 中现有的图迭代器遍历循环树，例如：后排序和深度优先等。下面的代码试图以深度优先的方式遍历一个以 `RootL` 为根的循环树:

```
#include "llvm/Analysis/LoopInfo.h"
#include "llvm/ADT/DepthFirstIterator.h"
...
```

```
// `RootL` has the type of `Loop*`
for (Loop *L : depth_first(RootL)) {
  // Work with `L`
}
```

在 GraphTraits 的帮助下，可以更灵活地遍历一个循环树。

除了在循环树中处理单个循环之外,LLVM 还提供了一个包装器类来表示整个结构——LoopNest。

LoopNest 是由 LoopNestAnalysis 生成的分析数据。它将所有子循环封装在一个给定的 Loop 实例中，并为常用功能提供了几个“快捷”API。以下是一些重要的接口:

- getOutermostLoop()/getInnermostLoop(): 这些 API 可以哟哪里检索最外层/最内层的 Loop 实例。因为许多循环优化只适用于内部或最外层的循环，所以非常好用。
- areAllLoopsSimplifyForm()/areAllLoopsRotatedForm(): 这些接口会表明，是否所有的循环都是某种规范形式。
- getPerfectLoops(…): 可以使用它来获得当前循环层次结构中的所有完美循环。所谓完美循环，我们指的是嵌套在一起的循环，它们之间没有“间隙”。下面是一个完美循环和非完美循环的例子:

```
// Perfect loops
for(int i=…) {
  for(int j=…){…}
}
// Non-perfect loops
for(int x=…) {
  foo();
  for(int y=…){…}
}
```

非完美循环的例子中，foo 调用点是上循环和下循环之间的间隙。

许多循环优化中，完美循环是更好的选择，例如：展开完美嵌套的循环更容易——理想情况下，只需要复制最内层的循环体。

至此，我们已经了解了如何使用嵌套循环。下一节中，我们；了解学习循环优化的另一个重要主题: 引导变量。

检索引导变量的范围

引导变量是在每次循环迭代中按特定模式前进的变量，是许多循环优化的关键。为了向量化一个循环，我们需要知道在循环中数组如何使用归纳变量——我们想要放入一个向量的数据。引导变量还可以帮助我们解决一个循环的行程计数 (迭代的总次数)。在深入了解细节之前，下图展示了一些与引导变量相关的术语，以及它们在循环中的位置:

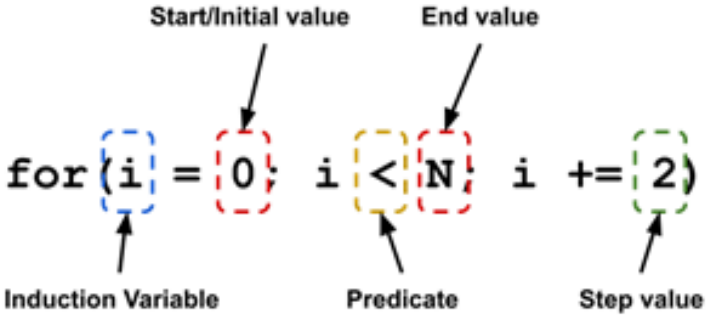

图 10.11 - 引导变量的术语

现在，我们来了解一些 API，它们可以帮助我们检索上图中所示的组件。

首先，来讨论一下引导变量。`Loop` 类已经提供了两个方便的方法来检索引导变量:`getCanonicalInductionVariable` 和 `getInductionVariable`。这两个方法都返回一个 `PHINode` 实例作为引导变量 (如果有的话)。第一种方法只能在引导变量从 0 开始并且每次迭代只增加 1 的情况下使用。另一方面，第二种方法可以处理更复杂的情况，但需要一个 `ScalarEvolution` 实例作为参数。

`ScalarEvolution` 是 LLVM 中一个有趣且强大的框架。简单地说，它试图跟踪值如何变化——通过算术运算——在程序路径上。将此放到循环优化的上下文中，可用于捕获循环中的递归值改变行为，这与引导变量有很强的关系。

要了解更多关于循环中引导变量的行为，可以通过 `loop::getInductionDescriptor` 检索 `InductionDescriptor` 实例。`InductionDescriptor` 实例提供了一些信息，比如：初始值、步长值，以及在每次迭代时更新引导变量的指令。`Loop` 类还提供了另一个类似的数据结构，用于实现引导变量的边界:`Loop::LoopBounds` 类。`LoopBounds` 不仅提供了引导变量的初始值和步长值，还提供了预期的结束值，以及用于检查退出条件的谓词。可以通过 `Loop::getBounds` 方法获取 `LoopBounds` 实例。

循环对程序的运行时性能至关重要。本节中，我们了解了如何在 LLVM IR 中表示循环，以及如何使用它们。我们还研究了它们的高级概念和各种用于检索所需循环属性的 API。有了这些知识，我们就离创建更有效、更积极的循环优化和从目标应用程序获得更高的性能更近了一步。

10.5. 总结

本节中，我们了解了 LLVM IR——目标无关的中间表示，它位于整个 LLVM 框架的核心。我们介绍了 LLVM IR 的高级结构，并给出了如何在其层次结构中遍历不同单元的实用指南。我们还关注于指令、值和 SSA，这些对于高效地使用 LLVM IR 至关重要。我们还介绍了一些关于同一主题的实用技能、技巧和例子。最后，我们学习了如何在 LLVM IR 中处理循环——这是优化对性能敏感的应用程序的一项重要技术。有了这些知识，就可以在 LLVM IR 上执行更广泛的程序分析和代码优化任务。

下一章中，我们将了解另一组 LLVM 工具 API，它们可以在使用 LLVM 进行开发、诊断和调试时，从而可以提高我们的工作效率。

第 11 章 配套工具

前一章中，我们学习了**低层虚拟机 (LLVM) 的中间表示 (IR)**(LLVM 中与目标无关的中间表示) 的基础知识，以及如何使用 C++ **应用程序编程接口 (API)** 检查和操作它，这些是在 LLVM 中进行程序分析和转换的核心技术。除了这些功能集，LLVM 还提供了许多配套工具，以提高编译器开发人员使用 LLVM IR 时的生产效率。我们将在本章中讨论这些工具。

编译器是一种复杂的软件。它不仅需要处理数千种不同的情况——包括不同类型的输入程序和各种各样的目标体系结构——而且编译器的正确性也是一个重要的话题: 编译后的代码需要具有与原始代码相同的行为。LLVM 虽然是一个大型编译器框架 (可能是最大的编译器框架之一)，但也不能例外。

为了解决这些复杂性问题，LLVM 提供了一系列小工具来改善开发体验。本章中，我们将展示如何使用这些工具。本文介绍的工具，可以帮助诊断正在开发的 LLVM 代码中出现的问题。这包括更高效的调试、错误处理和分析能力，例如：其中一个工具可以收集关键组件的统计数字——比如：特定 Pass 处理的基本块的数量——并自动生成报告。另一个例子是 LLVM 自己的错误处理框架，可以尽可能多地防止未处理的错误 (一种常见的编程错误)。

以下是我们将在本章中介绍的内容:

- 打印诊断信息
- 收集统计信息
- 添加时间测量
- LLVM 中的错误处理工具
- 了解 `Expected` 和 `ErrorOr` 类

在这些实用程序的帮助下，我们将有更好的时间调试和诊断 LLVM 代码，从而使开发者能够专注于想要用 LLVM 实现的核心逻辑。

11.1. 相关准备

本节中，我们还将使用 LLVM Pass 作为平台来展示不同的 API 用法。因此，请确保已经构建了命令行工具 `opt`，如下所示:

```
$ ninja opt
```

请注意，本章中的一些内容仅适用于调试构建版本的 LLVM。请查看第 1 章，以了解如何在调试模式下构建 LLVM。

如果你不确定如何创建一个新 LLVM Pass，也可以回顾一下第 9 章。

本章的示例代码的连接:https://github.com/PacktPublishing/LLVM-Techniques-Tips-and-Best-Practices-Clang-and-Middle-End-Libraries/tree/main/Chapter11

11.2. 打印诊断消息

在软件开发中，有许多方法可以诊断错误——例如，使用调试器、在程序中插入杀毒器 (例如，捕获无效的内存访问)，或者简单地使用最简单，但最有效的方法之一: 添加打印语句。虽然最后一个选项听起来不是很聪明，但在其他选项不能充分发挥其潜力的许多情况下 (例如，调试信息质量较差的发布模式二进制文件或多线程程序)，其实非常有用。

LLVM 提供了一个小工具，不仅可以打印调试消息，还过滤要显示的消息。假设我们有一个 LLVM Pass——`SimpleMulOpt`——用左移操作替换 2 的幂次乘法 (这是我们在前一章处理 LLVM IR 的最后一节中所完成的)。下面是它的运行方法的部分实现:

```
PreservedAnalyses
SimpleMulOpt::run(Function &F, FunctionAnalysisManager &FAM) {
  for (auto &I : instructions(F)) {
    if (auto *BinOp = dyn_cast<BinaryOperator>(&I) &&
    BinOp->getOpcode() == Instruction::Mul) {
      auto *LHS = BinOp->getOperand(0),
      *RHS = BinOp->getOperand(1);
      // `BinOp` is a multiplication, `LHS` and `RHS` are its
      // operands, now trying to optimize this instruction…
      …
    }
  }
  …
}
```

前面的代码遍历给定函数中的所有指令，然后查找表示算术乘法的指令。如果有，Pass 将与 LHS 和 RHS 操作数一起工作 (它们出现在代码的其余部分中——这里没有显示)。

假设我们想在开发过程中打印出操作数变量。大多数原生的方法是使用我们的“老朋友”`errs()`，它可以将任意消息发送到 `stderr`。如下所示:

```
// (extracted from the previous snippet)
…
auto *LHS = BinOp->getOperand(0),
     *RHS = BinOp->getOperand(1);
errs() << "Found a multiplication with operands ";
LHS->printAsOperand(errs());
errs() << " and ";
RHS->printAsOperand(errs());
…
```

代码中使用的 `printAsOperand` 将 `Value` 的文本表示输出到给定流中 (在本例中为 `errs()`)。

一切看起来都很正常，除了这些消息无论如何都会被打印出来。不过是在生产环境中，这不是我们想要的。要么需要在交付产品之前删除这些代码，要么在这些代码周围添加一些宏保护 (例如，`#ifndef NDEBUG`)，要么可以使用 LLVM 提供的调试工具。下面是一个例子:

```
#include "llvm/Support/Debug.h"
#define DEBUG_TYPE "simple-mul-opt"
…
auto *LHS = BinOp->getOperand(0),
```

```
    *RHS = BinOp->getOperand(1);
LLVM_DEBUG(dbgs() << "Found a multiplication with operands ");
LLVM_DEBUG(LHS->printAsOperand(dbgs()));
LLVM_DEBUG(dbgs() << " and ");
LLVM_DEBUG(RHS->printAsOperand(dbgs()));
…
```

上面的代码基本上做了以下三件事:

- 用 `dbgs()` 替换 `errors()`。这两个流基本上做的是相同的事情，但后者将向输出消息添加一个漂亮的格式 (调试日志输出)。
- 使用 `LLVM_DEBUG(…)` 宏函数包装所有与调试打印相关的行。使用这个宏可以确保只在开发模式下编译封闭行。它还对调试消息类别进行编码，稍后我们将介绍这个类别。
- 使用 `LLVM_DEBUG(…)` 宏函数之前，请确保您将 `DEBUG_TYPE` 定义为所需的调试类别字符串 (本例中为 `simple-mult-opt`)。

除了上述代码修改之外，我们还需要使用一个额外的命令行标志`-debug`，可以选择打印那些调试消息。下面是一个例子:

```
$ opt -O3 -debug -load-pass-plugin=… …
```

但是，会发现输出非常嘈杂。有大量来自其他 LLVM PAss 的调试消息。在本例中，我们只对来自我们 Pass 的消息感兴趣。

要过滤掉不相关的消息，可以使用`-debug-only` 命令行标志。下面是一个例子:

```
$ opt -O3 -debug-only=simple-mul-opt -load-pass-plugin=… …
```

`-debug-only` 后面的值是我们在前面的代码片段中定义的 `DEBUG_TYPE` 值。换句话说，我们可以使用由每个 Pass 定义的 `DEBUG_TYPE` 来过滤所需的调试消息。我们还可以选择要打印的多个调试类别，使用以下命令:

```
$ opt -O3 -debug-only=sroa,simple-mul-opt -load-pass-plugin=… …
```

这个命令不仅打印来自 `SimpleMulOpt` Pass 的调试消息，还打印来自 `SROA` Pass 的调试消息——包含在 `O3` 优化管道中的 LLVM Pass。

除了为 LLVM Pass 定义一个单独的调试类别 (`DEBUG_TYPE`) 之外，实际上可以自由地在 Pass 中使用任意多的类别，例如：当希望对 Pass 的不同部分使用单独的调试类别时，这是很有用的。我们可以在 `SimpleMulOpt` Pass 中为每个操作数使用单独的类别。可以这样做:

```
…
#define DEBUG_TYPE "simple-mul-opt"
```

```
auto *LHS = BinOp->getOperand(0),
     *RHS = BinOp->getOperand(1);
LLVM_DEBUG(dbgs() << "Found a multiplication instruction");
DEBUG_WITH_TYPE("simple-mul-opt-lhs",
                LHS->printAsOperand(dbgs() << "LHS operand: "));
DEBUG_WITH_TYPE("simple-mul-opt-rhs",
                RHS->printAsOperand(dbgs() << "RHS operand: "));
...
```

DEBUG_WITH_TYPE 是 LLVM_DEBUG 的特殊版本。在第二个参数处执行代码，第一个参数作为调试类别，它可以不同于当前定义的 DEBUG_TYPE 值。前面的代码中，除了使用 simplemul-opt 类别打印 Found a multiplication instruction 外，我们还使用 simple-multi-opt-lhs 打印与**左侧 (LHS)** 操作数相关的消息，并使用 simple-mul-opt-rhs 打印其他操作数的消息。有了这个特性，就可以更细粒度地通过 opt 命令选择调试消息类别了。

现在，已经了解了如何使用 LLVM 提供的实用程序仅在开发环境中打印调试消息，以及如何在需要时对它们进行过滤。下一节中，我们将了解如何在运行 LLVM Pass 时收集关键统计信息。

11.3. 收集统计信息

如前一节所述，编译器是一种复杂的软件。收集统计数字 (例如，特定优化处理的基本块的数量) 是快速了解编译器运行时行为的最简单和最有效的方法。

在 LLVM 中收集统计信息有几种方法。本节中，我们将学习三种最常见和有用的方法，这些方法在这里概述:

- 使用 Statistic 类
- 使用优化注释
- 添加耗时测量

第一个选项是通过简单的计数器，收集统计信息的通用工具。第二个选项是专门设计用来分析编译器优化的。最后一项，用于在编译器中收集计时信息。

从第一个开始。

11.3.1 使用 Statistic 类

本节中，我们将通过对上一节中的 LLVM Pass——SimpleMulOpt——进行修改来演示新特性。首先，我们不仅想从乘法指令中打印出操作数 Value，而且还想计算 Pass 已经处理了多少乘法指令。首先，尝试使用我们刚刚了解的 LLVM_DEBUG 基础架构来实现这个特性，如下所示:

```
#define DEBUG_TYPE "simple-mul-opt"
PreservedAnalyses
SimpleMulOpt::run(Function &F, FunctionAnalysisManager &FAM) {
  unsigned NumMul = 0;
  for (auto &I : instructions(F)) {
    if (auto *BinOp = dyn_cast<BinaryOperator>(&I) &&
    BinOp->getOpcode() == Instruction::Mul) {
      ++NumMul;
```

```
      ...
    }
  }
  LLVM_DEBUG(dbgs() << "Number of multiplication: " << NumMul);
  ...
}
```

这种方法看起来非常简单。但它有一个缺点——我们感兴趣的统计数字与其他调试消息混杂在一起。我们需要采取额外的操作来解析或过滤我们想要的值，尽管可能会有读者说这些问题可以通过为每个计数器变量使用单独的 DEBUG_TYPE 标签来解决，但当计数器变量的数量增加时，可能会发现自己创建了大量的冗余代码。

一个很好的解决方案是使用 LLVM 提供的 `Statistic` 类 (和相关工具)。下面是使用这个解决方案重写的版本:

```
#include "llvm/ADT/Statistic.h"
#define DEBUG_TYPE "simple-mul-opt"
STATISTIC(NumMul, "Number of multiplications processed");
PreservedAnalyses
SimpleMulOpt::run(Function &F, FunctionAnalysisManager &FAM) {
  for (auto &I : instructions(F)) {
    if (auto *BinOp = dyn_cast<BinaryOperator>(&I) &&
    BinOp->getOpcode() == Instruction::Mul) {
      ++NumMul;
      ...
    }
  }
  ...
}
```

前面的代码展示了 `Statistic` 的用法，调用 `STATISTIC` 宏函数来创建一个 `Statistic` 类型变量 (带有文本描述)，并像使用普通整数计数器变量一样使用即可。

这个解决方案只需要修改原始代码中的几行，而且它收集所有的计数器值，并在优化结束时将它们打印出来，例如：在 `opt` 中使用`-stats` 标志运行 `SimpleMulOpt` Pass，会得到以下输出:

```
$ opt -stats -load-pass-plugin=... ...
===-------------------------------===
      ... Statistics Collected ...
===-------------------------------===
87 simple-mul-opt - Number of multiplications processed
$
```

87 是 `SimpleMulOpt` 中处理的乘法指令数。当然，可以随意添加任意数量的统计计数器，以便收集不同的统计信息。如果在管道中运行多个 Pass，那么所有的统计数字都将显示在同一个表中，例如：如果在 `SimpleMulOpt` 中添加另一个统计计数器，以从乘法指令中收集多个常数操作数的非

幂数，并运行 Pass SROA，可以得到类似于下面所示的输出:

```
$ opt -stats -load-pass-plugin=… --passes="sroa,simple-multopt" …
===-------------------------------===
      … Statistics Collected …
===-------------------------------===
94 simple-mul-opt - Number of multiplications processed
87 simple-mul-opt - Number of none-power-of-two constant
operands
100 sroa - Number of alloca partition uses rewritten
34 sroa - Number of instructions deleted
…
$
```

前面代码中的第二列是原始 Pass 的名称，它由 `DEBUG_TYPE` 指定，在任何调用 `STATISTIC` 之前定义。

或者，可以通过在 `opt` 中添加`-stats-json` 标志来输出 JSON 格式的结果:

```
$ opt -stats -stats-json -load-pass-plugin=… …
{
  "simple-mul-opt.NumMul": 87
}
$
```

在 JSON 格式中，统计项的字段名不是用文本描述打印统计值，而是使用这种格式:"< 传递名称 >.< 统计变量名 >"(这里的 Pass 名称也是 `DEBUG_TYPE` 的值)。此外，可以使用`-infooutput-file=<` 文件名 `>` 命令行选项将统计结果 (默认或 JSON 格式) 输出到文件中:

```
$ opt -stats -stats-json -info-output-file=my_stats.json …
$ cat my_stats.json
{
  "simple-mul-opt.NumMul": 87
}
$
```

现在，我们已经了解了如何使用 `statistic` 类收集简单的统计值。下一节中，我们将了解一种编译器优化所特有的统计数据收集方法。

11.3.2 使用优化注释

编译器优化通常包括两个阶段: 从输入代码中搜索所需的模式,然后修改代码。以我们的 `SimpleMulOpt` Pass 为例: 第一阶段是寻找具有 2 次幂常量操作数的乘法指令 (`BinaryOperator`,带有 `Instruction::Mul` 的**操作码 (opcode)**)。在第二阶段，通过 `IRBuilder::CreateShl(…)` 创建左移指令，并用这些替换所有旧的乘法指令。

有时，由于输入代码不可行，优化算法在第一阶段简单地“退出”，例如：在 `SimpleMulOpt` 中，我们正在寻找一条乘法指令，但是如果传入的指令不是 `BinaryOperator`，Pass 将不会继续到第二阶段 (并继续到下一个指令)。有时，我们想知道这种情况背后的原因，这可以帮助我们改进优化算法或诊断错误/次优化的编译器优化。LLVM 提供了一个很好的工具，称为**优化注释**，用于收集和报告在优化过程中发生的这种情况 (或任何类型的信息)。

例如，我们有以下输入代码:

```
int foo(int *a, int N) {
  int x = a[5];
  for (int i = 0; i < N; i += 3) {
    a[i] += 2;
    x = a[5];
  }
  return x;
}
```

理论上，我们可以使用**循环不变代码方式 (LICM)** 将这段代码优化为一个等价的代码库，例如:

```
int foo(int *a, int N) {
  for (int i = 0; i < N; i += 3) {
    a[i] += 2;
  }
  return a[5];
}
```

我们可以这样做，因为第 5 个数组元素 `a[5]` 在循环中永远不会改变它的值。然而，如果我们在原始代码上运行 LLVM 的 LICM Pass，它将无法执行预期的优化。

要诊断这个问题,我们可以调用带有附加选项的 opt 命令:`--pass- comments-output=<filename>`。文件名将是一个 `YAML Ain't Markup Language (YAML)` 文件，其中优化注释打印出 LICM 未能优化的可能原因:

```
$ opt -licm input.ll  -pass-remarks-output=licm_remarks.yaml …
$ cat licm_remarks.yaml
…
--- !Missed
```

```
Pass:           licm
Name:           LoadWithLoopInvariantAddressInvalidated
Function:       foo
Args:
  - String:     failed to move load with loop-invariant
address because the loop may invalidate its value
...
$
```

上面的 `cat` 命令显示了 `licm_comments.yaml` 中的一个优化注释条目。这个条目告诉我们，在处理 `foo` 函数时，LICM Pass 中错过的优化，还告诉了我们原因:LICM 不确定特定的内存地址是否会在循环内修改 (无效)。尽管此消息没有提供细粒度的详细信息，但我们仍然可以推断出与 LICM 有关的有问题的内存地址可能是 `a[5]`。LICM 不确定 `a[i] += 2` 语句是否修改了 `a[5]` 的内容。

有了这些知识，编译器开发人员就可以动手改进 LICM 了——例如，教 LICM 识别步长大于 1 的归纳变量 (即循环中的 i 变量)(在本例中是 3，因为 `i += 3`)。

要生成优化注释 (如前面输出中所示)，编译器开发人员需要将特定的实用程序 API 集成到优化 Pass 中。为了向展示如何在自己的 Pass 中做到这一点，我们将重用 `SimpleMulOpt` Pass 作为示例。下面是在 `SimpleMulOpt` 中执行第一阶段的部分代码——搜索两个常量操作数的幂次乘法:

```
...
for (auto &I : instructions(F)) {
  if (auto *BinOp = dyn_cast<BinaryOperator>(&I))
  if (BinOp->getOpcode() == Instruction::Mul) {
    auto *LHS = BinOp->getOperand(0),
         *RHS = BinOp->getOperand(1);
    // Has no constant operand
    if (!isa<Constant>(RHS)) continue;
    const APInt &Const = cast<ConstantInt>(RHS)->getValue();
    // Constant operand is not power of two
    if (!Const.isPowerOf2()) continue;
    ...
  }
}
```

前面的代码在确保操作数也是 2 的幂次之前检查操作数是否为常量。如果其中任何一个检查失败，算法将退出，继续执行函数中的下一条指令。

我们故意在这段代码中插入了一个小缺陷，使其功能不那么强大，我们将向您展示如何使用优化注释来找到该问题。以下是实现这一目标的步骤:

1. 首先,需要一个 `OptimizationRemarkEmitter` 实例,可以发送注释消息。这可以从它的父分析器 `OptimizationRemarkEmitterAnalysis` 获得。下面是如何将它包含在 `SimpleMulOpt::run` 方法的起始段:

```
#include "llvm/Analysis/OptimizationRemarkEmitter.h"
PreservedAnalyses
SimpleMulOpt::run(Function &F, FunctionAnalysisManager
&FAM) {
  OptimizationRemarkEmitter &ORE
    = FAM.getResult<OptimizationRemarkEmitterAnalysis>(F);
  ...
}
```

2. 然后，如果乘法指令缺少一个常量操作数，我们将使用这个 `OptimizationRemarkEmitter` 实例来发出一个优化注释，如下所示:

```
#include "llvm/IR/DiagnosticInfo.h"
...
if (auto *BinOp = dyn_cast<BinaryOperator>(&I))
if (BinOp->getOpcode() == Instruction::Mul) {
  auto *LHS = BinOp->getOperand(0),
  *RHS = BinOp->getOperand(1);
  // Has no constant operand
  if (!isa<ConstantInt>(RHS)) {
    std::string InstStr;
    raw_string_ostream SS(InstStr);
    I.print(SS);
    ORE.emit([&]() {
      return OptimizationRemarkMissed(DEBUG_TYPE,
        "NoConstOperand", &F)
        << "Instruction" <<
        << ore::NV("Inst", SS.str())
        << " does not have any constant operand";
      });
      continue;
    }
  }
...
```

这里有几点需要注意，如下:

- 方法 `OptimizationRemarkEmitter::emit` 接受 lambda 函数作为参数。如果优化注释特性使能,将使用这个 lambda 函数发出一个优化注释对象 (例如,`-pass-remarks-output` 命令行选项)。
- `OptimizationRemarkMissed` 类 (注意，它不是在 `OptimizationRemarkEmitter.h` 中声明的，而是在 `DiagnosticInfo.h` 头文件中声明的) 表示错过的**优化机会**的注释。这种情况下，错过的机会是指令 `I` 没有任何常数操作数。`OptimizationRemarkMissed` 的构造函数有三个参数:Pass 的名称、错过的优化机会名称和包含的 IR 单元 (在本例中，我们使用包含的 `Function`)。除了构造一个 `optimizationcommentmissed` 对象外，我们还通过尾部的流操作符 (`<<`) 连接几个对象。这些对象最终将放在 YAML 文件中每个优化注释条目的 `Args` 部分中。

除了使用 OptimizationRemarkMissed 通知错过了的优化机会外，还可以使用 DiagnosticInfoOptimizationBasee 的其他类来表示不同种类的信息，例如：使用 OptimizationRemark 来查找哪一个优化应用成功，并使用 OptimizationRemarkAnalysis 来保存分析数据/实际运行日志。

- 在由流操作符连接的对象中，ore::NV(…) 似乎是一种特殊情况。在优化注释 YAML 文件中，Args 部分下的每一行都是一个键-值对 (例如，String: failed to move load with…. 中，String 是键)。ore::NV 对象允许自定义键值对。本例中，使用 Inst 作为键，而 SS.str() 作为值。这个特性为解析优化注释 YAML 文件提供了更大的灵活性——例如，想编写一个小工具来可视化优化注释，自定义 Args 键可以 (在解析阶段) 更轻松地将关键数据与其他字符串区分开来。

3. 既然已经插入了发出优化注释的代码，现在就可以对其进行测试了。这一次，我们将使用以下 IR 函数作为输入代码:

```
define i32 @bar(i32 %0) {
  %2 = mul nsw i32 %0, 3
  %3 = mul nsw i32 8, %3
  ret %3
}
```

可以重新构建 SimpleMulOpt Pass，并使用如下命令运行:

```
$ opt -load-pass-plugin=… -passes="simple-mul-opt" \
-pass-remarks-output=remark.yaml -disable-output
input.ll
$ cat remark.yaml
--- !Missed
Pass:      simple-mul-opt
Name:      NoConstOperand
Function:  bar
Args:
  - String: 'Instruction'
  - Inst:   ' %3 = mul nsw i32 8, %3'
  - String: ' does not contain any constant
operand'
...
$
```

从这个优化注释条目中，可以发现 SimpleMulOpt 失败了，因为它无法在其中一条 (乘法) 指令中找到一个常量操作数。Args 部分详细说明了这样做的原因。

有了这些信息，我们意识到 SimpleMulOpt 无法优化其第一个操作数 (LHS 操作数) 是 2 的幂次常数的乘法，尽管这是一个适当的优化机会。因此，我们现在可以修复 SimpleMulOpt

的实现来检查操作数是否为常量，如下所示:

```
…
if (BinOp->getOpcode() == Instruction::Mul) {
  auto *LHS = BinOp->getOperand(0),
  *RHS = BinOp->getOperand(1);
  // Has no constant operand
  if (!isa<ConstantInt>(RHS) && !isa<ConstantInt>(LHS)) {
    ORE.emit([&]() {
      return …
    });
    continue;
  }
  …
} …
```

现在，我们已经了解了如何在 LLVM Pass 中发出优化注释，以及如何使用生成的报告来发现潜在的优化机会。

到目前为止，我们只研究了生成的优化备注 YAML 文件。虽然它提供了有价值的诊断信息，但如果能够获得更细粒度和更直观的位置信息，从而准确地知道这些评论发生在哪里，那就更好了。幸运的是，Clang 和 LLVM 提供了一种实现这一目标的方法。

在 Clang 的帮助下，我们可以实际生成附带源文件位置 (即原始源文件中的行号和列号) 的优化备注。此外，LLVM 还为您提供了一个小工具，可以将优化注释与其相应的源代码位置关联起来，并在网页上可视化结果:

1. 让我们重用下面的代码作为输入:

```
int foo(int *a, int N) {
  for (int i = 0; i < N; i += 3) {
    a[i] += 2;
  }
  return a[5];
}
```

首先，我们使用 `clang` 命令生成优化注释:

```
$ clang -O3 -foptimization-record-file=licm.remark.yaml \
        -S opt_remark_licm.c
```

虽然使用了不同的名称，但`-foptimization-record-file` 是命令行选项，用于指定文件名，在文件中记录优化注释。

2. 生成 `licm.remark.yaml` 之后，使用一个名为 `opt-viewer.py` 的工具对注释进行可视化。默认情况下，`opt-viewer.py` 脚本不在 `<install path>/bin`(例如`/usr/bin`) 中，而是安装在 `<install path>/share/opt-viewer` (例如`/usr/share/opt-viewer`) 中。我们将使用以下命令行选项来调用这个脚本:

```
$ opt-viewer.py --source-dir=$PWD \
--target-dir=licm_remark licm.remark.yaml
```

(请注意，`opt-viewer.py` 依赖于几个 Python 包，如：pyyaml 和 pyplets。请在使用 `opt-viewer.py` 之前安装它们。)

3. 在 `licm_remark` 文件夹中会生成一个 HTML 文件 `index.html`。打开网页之文件前，请将原始的源代码 `opt_remark_licm.c` 也复制到那个文件夹中。之后，将能够看到这样一个网页:

Source Location	Hotness	Function	Pass
./opt_remark_licm.c:0:0		foo	asm-printer
./opt_remark_licm.c:0:0		foo	gvn
./opt_remark_licm.c:1:0		foo	asm-printer
./opt_remark_licm.c:1:0		foo	prologepilog
./opt_remark_licm.c:3:3		foo	asm-printer
./opt_remark_licm.c:3:3		foo	asm-printer
./opt_remark_licm.c:3:3		foo	asm-printer
./opt_remark_licm.c:3:3		foo	asm-printer
./opt_remark_licm.c:3:3		foo	loop-unroll
./opt_remark_licm.c:3:3		foo	loop-vectorize
./opt_remark_licm.c:3:3		foo	loop-vectorize
./opt_remark_licm.c:3:21		foo	asm-printer
./opt_remark_licm.c:4:5		foo	licm
./opt_remark_licm.c:4:5		foo	licm
./opt_remark_licm.c:4:10		foo	asm-printer
./opt_remark_licm.c:4:10		foo	asm-printer

图 11.1 - Web 页面的优化注释与源文件的结合

我们对其中的两列特别感兴趣:**Source Location** 和 **Pass**。后一列显示了 Pass 的名称和优化注释的类型——以红色、绿色和白色呈现的 `Missed`、`Passed` 或 `Analyzed`，分别附加在 **Source Location** 列的给定行上。

如果点击源位置列中的链接，这将导航到这样一个页面，看起来像这样:

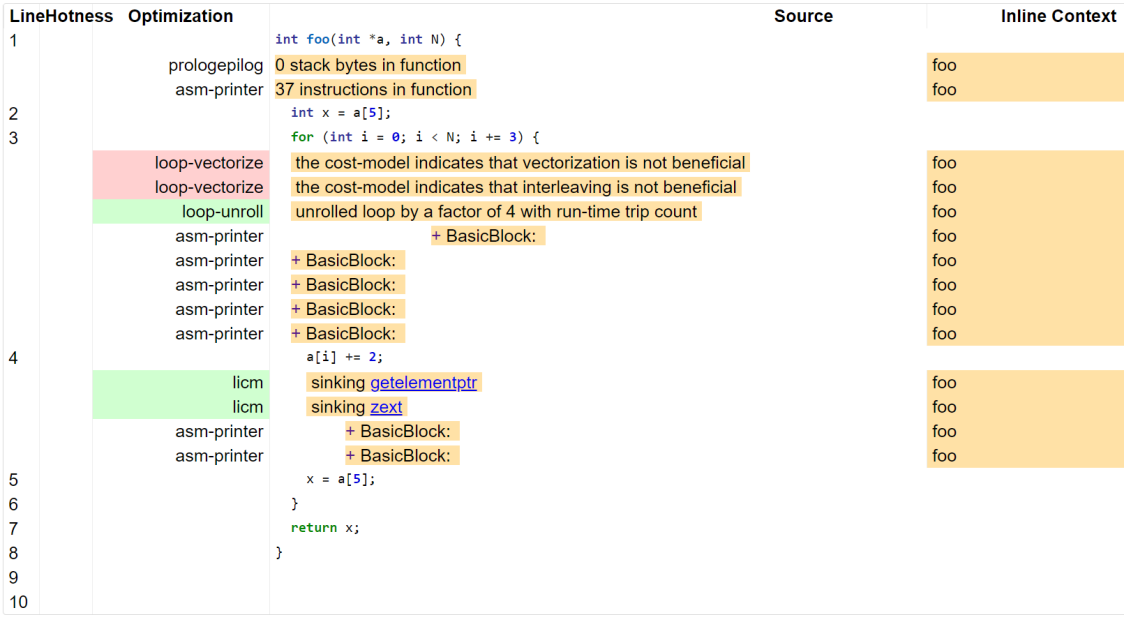

图 11.2 - 优化注释的详细信息

这个页面提供了优化注释细节的一个很好的视图，并与原始源代码行交叉显示。例如，在第 3 行，`loop-vectorize` Pass 说它不能向量化这个循环，因为它的模型不认为这样做是有益的。

现在，我们已经了解了如何使用优化注释来深入了解优化 Pass，这在调试丢失的优化机会或修复编译错误时特别有用。

下一节中，我们将了解一些有用的技能来分析 LLVM 的执行时间。

11.4. 添加耗时测量

LLVM 是一个庞大的软件，由数百个组件紧密协作，其不断增加的运行时间正慢慢成为一个问题。这影响了许多对编译时间敏感的用例，例如：即时 (JIT) 编译器。为了系统地诊断这个问题，LLVM 提供了一些有用的工具来分析执行时间。

运行时长分析一直是软件开发中的一个重要话题。通过从单个软件组件中收集运行时间，我们可以更容易地发现性能瓶颈。本节中，我们将了解 LLVM 提供的两个工具：`Timer` 类和 `TimeTraceScope` 类。我们先从 `Timer` 类开始。

11.4.1 Timer 类

`Timer` 类，顾名思义，可以度量代码区域的执行时间。下面是一个例子：

```
#include "llvm/Support/Timer.h"
…
Timer T("MyTimer", "A simple timer");
T.startTimer();
// Do some time-consuming works…
T.stopTimer();
```

前面的代码中，`Timer` 实例 `T` 测量在该区域中花费的时间，该区域使用 `startTimer` 和 `stopTimer` 方法进行耗时测量。

现在我们已经收集了耗时数据，让我们试着将其打印出来。下面是一个例子:

```
Timer T(…);
…
TimeRecord TR = T.getTotalTime();
TR.print(TR, errs());
```

在前面的代码中,TimeRecord 实例封装了由 Timer 类收集的数据。然后,使用 TimeRecord::print 将其打印到流中——在本例中是 errors() 流。此外，指定了另一个 TimeRecord 实例——通过 print 的第一个参数——作为总耗时。来看看这段代码的输出:

```
===-------------------------------------------------------===
               Miscellaneous Ungrouped Timers
===-------------------------------------------------------===
---User Time--- --User+System-- ---Wall Time--- ---Name ---
 0.0002 (100.0%) 0.0002 (100.0%) 0.0002 (100.0%) A simple timer
 0.0002 (100.0%) 0.0002 (100.0%) 0.0002 (100.0%) Total
 0.0002 (100.0%) 0.0002 (100.0%) 0.0002 (100.0%)
```

前面的输出中，第一行显示了从前面的 Timer 实例收集的 TimeRecord 实例，而第二行显示了总时间——TimeRecord::print 的第一个参数。

现在,我们知道如何打印由单个 Timer 实例收集的计时数据,但是如果有多个计时器呢?LLVM 为 Timer 类提供了另一个支持工具:TimerGroup 类。下面是使用 TimerGroup 类的例子:

```
TimerGroup TG("MyTimerGroup", "My collection of timers");

Timer T("MyTimer", "A simple timer", TG);
T.startTimer();
// Do some time-consuming works…
T.stopTimer();

Timer T2("MyTimer2", "Yet another simple timer", TG);
T2.startTimer();
// Do some time-consuming works…
T2.stopTimer();

TG.print(errs());
```

在前面的代码中，我们声明了一个 TimerGroup 实例 TG，并将它用作创建 Timer 实例的第三个构造函数参数。最后，使用 TimerGroup::print 进行打印。下面是这段代码的输出:

```
===-------------------------------------------------------===
                  My collection of timers
===-------------------------------------------------------===
Total Execution Time: 0.0004 seconds (0.0004 wall clock)
  ---User Time--- --User+System-- ---Wall Time--- ---Name ---
  0.0002 ( 62.8%) 0.0002 ( 62.8%) 0.0002 ( 62.8%) A simple timer
  0.0001 ( 37.2%) 0.0001 ( 37.2%) 0.0001 ( 37.2%) Yet another simple timer
  0.0004 (100.0%) 0.0004 (100.0%) 0.0004 (100.0%) Total
```

输出中的每一行 (最后一行除外) 都是该组中 `Timer` 的 `TimeRecord` 实例。

目前为止，我们一直使用 `Timer::startTimer` 和 `Timer::stopTimer` 来切换计时器。为了在不手动调用这两个方法的情况下，更容易地测量代码块内的时间间隔 (即用大括号 `{}` 括起来的区域)，LLVM 提供了另一个工具，可以在进入代码块时自动启动计时器，并在退出时关闭计时器。让我们看看如何使用 `TimeRegion` 类:

```
TimerGroup TG("MyTimerGroup", "My collection of timers");
{
  Timer T("MyTimer", "A simple timer", TG);
  TimeRegion TR(T);
  // Do some time-consuming works···
} {
  Timer T("MyTimer2", "Yet another simple timer", TG);
  TimeRegion TR(T);
  // Do some time-consuming works···
}

TG.print(errs());
```

这里，我们没有调用 `startTimer/stopTimer`，而是将要测量的代码放入单独的代码块中，并使用 `TimeRegion` 变量来自动切换计时器，这段代码将输出与前一个示例相同的内容。`TimeRegion` 的帮助下，可以有一个更简洁的语法，并避免忘记关闭计时器导致的错误。

现在，我们已经了解了如何使用 `Timer`，及其支持的实用程序来测量某个代码区域的执行时间。下一节中，我们将了解一种更高级的时间测量方式 (可以捕获程序的层次结构)。

11.4.2 收集时间轨迹

前一节中，我们了解了如何使用 `Timer` 来收集一段代码区域的执行时间。虽然这提供了编译器运行时性能的描述，但有时我们需要一个更结构化的时间配置文件，以便完全理解系统问题。

`TimeTraceScope` 是 LLVM 提供的一个类，用于执行全局范围的时间分析。它的用法非常简单: 类似于在前一节中看到的 `TimeRegion`，`TimeTraceScope` 实例在进入和退出代码块时自动打开和关闭时间分析器。下面是一个例子:

```
TimeTraceScope OuterTimeScope("TheOuterScope");
for (int i = 0; i < 50; ++i) {
```

```
  {
    TimeTraceScope InnerTimeScope("TheInnerScope");
    foo();
  }
  bar();
}
```

前面的代码片段中,我们创建了两个 `TimeTraceScope` 实例:`OuterTimeScope` 和 `InnerTimeScope`。它们分别尝试分析整个区域的执行时间和 `foo` 函数耗费的时间。

通常，我们会使用 `Timer`，而不是 `TimeTraceScope`，虽然它只能提供从每个计时器收集到的累计持续时间。然而，在这种情况下，我们更感兴趣的是代码的不同部分如何在时间轴上分配自己的时间，例如：`foo` 函数是否每次循环迭代都耗费相同的时间？如果不是，那么哪个迭代会比其他迭代耗费更多的时间？

要查看结果，需要在运行 Pass 时向 `opt` 命令添加额外的命令行选项 (假设在 Pass 中使用 `TimeTraceScope`)。下面是一个例子:

```
$ opt -passes="…" -time-trace -time-trace-file=my_trace.json …
```

`-time-trace` 标志要求 `opt` 将 `TimeTraceScope` 收集的所有轨迹信息,导出到`-time-trace-file` 选项指定的文件中。

运行此命令后，将得到一个新文件 `my_trace.json`。这个文件的内容不是人能看的，但想看也有办法。可以使用 **Chrome** 网络浏览器将其可视化。以下是实现这一目标的步骤:

1. 打开 Chrome 浏览器,在 **URL(Uniform Resource Locator)** 栏中输入 `Chrome://tracing`。会看到这样一个界面:

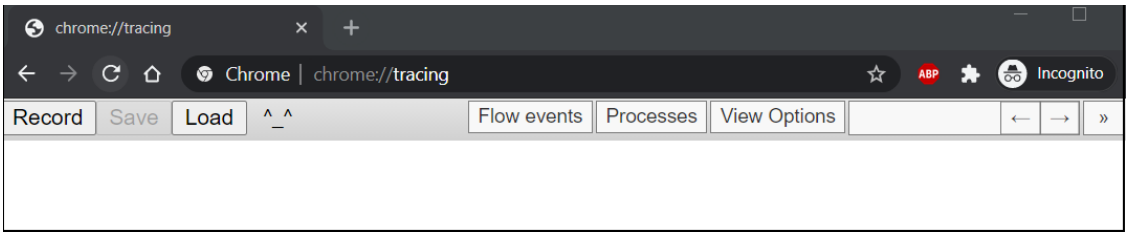

图 11.3 - Chrome 中的跟踪可视化工具

2. 点击左上角的 **Load** 按钮，选择 `my_trace.json` 文件。会看到这样一个页面:

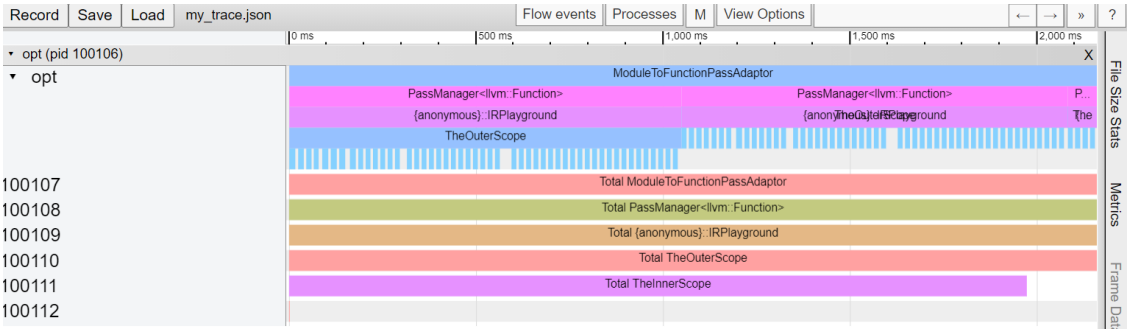

图 11.4 - 打开 `my_trace.json` 后的视图

每个色块代表由 `TimeTraceScope` 实例收集的耗时情况。

3. 来仔细看看: 请按数字键 3 切换到缩放模式。之后，应该能够通过点击和拖动鼠标向上或向下，进行放大或缩小的操作。同时，可以使用方向键向左或向右滚动时间轴。这是我们放大后时间线的一部分:

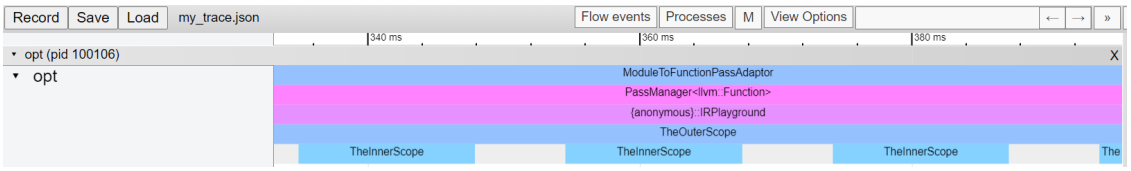

图 11.5 - 时间线的一部分

如图 11.5 所示，有几个层叠加在一起。这个布局反映了不同的 `TimeTraceScope` 实例是如何在 `opt`(和我们的 Pass) 中执行的，例如：我们的 `TimeTraceScope` 实例名为 `TheOuterScope`，它堆叠在多个 `TheInnerScope` 块之上。每个 `TheInnerScope` 块代表了，的 `foo` 函数在每次循环迭代中耗费的时间。

4. 我们可以通过点击一个块来进一步检查它的属性，例如：如果点击其中一个 `TheInnerScope` 块，它的计时属性将显示在屏幕的下半部分:

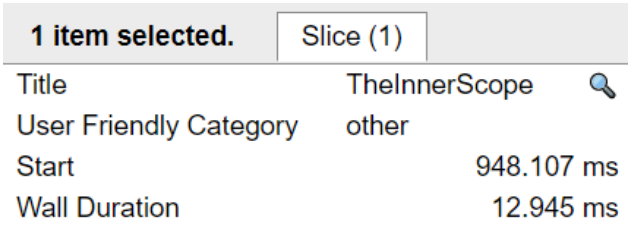

图 11.6 - 耗时块的详细信息

这将为我们提供耗时和开始时间等信息。

通过这种可视化，可以将计时信息与编译器的结构结合起来，从而帮助我们更快地发现性能瓶颈。

除了 `opt` 之外，`clang` 还可以生成相同的跟踪 JSON 文件 (请添加`-ftime-trace`)。下面是一个例子:

```
$ clang -O3 -ftime-trace -c foo.c
```

这将生成与输入文件同名的 JSON 跟踪文件 (本例中是 `foo.json`)，可以用我们刚学过的方法把它可视化。

本节中，我们了解了一些从 LLVM 收集统计信息的有用技能。`Statistic` 类可以用作整数计数器，来记录在优化过程中发生的事件的数量。另一方面，优化注释可以让我们了解优化 Pass 中的一些决策过程，使编译器开发人员更容易判断错过的优化机会。使用 `Timer` 和 `TimeTraceScope`，开发人员可以以一种更易于管理的方式监控 LLVM 的执行时间，并有信心地优化编译速度。这些技术可以提高 LLVM 开发人员在创造新发明，或解决具有挑战性的问题时的效率。

本章的下一节中，我们将蓼莪及如何使用 LLVM 提供的工具，以一种高效的方式编写错误处理的代码。

11.5. LLVM 中的错误处理工具

错误处理一直是软件开发中广泛讨论的话题。可以像返回错误代码一样简单——例如：在许多 Linux API 中 (例如，`open` 函数)——或者使用高级机制，例如：抛出异常，这已经被许多现代编程语言 (如 Java 和 C++) 广泛采用。

虽然 C++ 内置了对异常处理的支持，但 LLVM 在其代码库中根本没有使用。这一决定背后的基本原因是，尽管方便表达语法，但 C++ 中的异常处理在性能方面需要付出了很高的代价。简单地说，异常处理使原始代码更加复杂，并妨碍编译器对其进行优化。此外，在运行时，程序通常需要花费更多的时间从异常中恢复。因此，LLVM 在其代码库中默认禁用异常处理，转而使用其他错误处理方法——例如，在返回值中携带一个错误或使用本节中的工具。

本节的前半部分，我们将讨论 `Error` 类，它代表错误。这与传统的错误表示不同——例如，当使用整数作为错误代码时，不能忽略生成的 `Error` 实例而不处理。我们稍后将对此进行解释。

除了 `Error` 类，在 LLVM 的代码库中，许多错误处理代码都有一个共同的模式:API 可以返回结果或错误，但不能同时返回结果或错误。例如，当我们调用读取文件的 API 时，期望得到该文件的内容 (结果) 或出错时的错误 (例如，没有这样的文件)。本节的第二部分中，我们将了解两个实现此模式的工具类。

首先介绍 `Error` 类。

11.5.1 Error 类

`Error` 类所表示的概念非常简单: 一个带有附加描述的错误，比如错误消息或错误代码，通过一个值 (作为函数的参数) 传递或从函数返回。开发人员也可以创建自己的 `Error` 实例，例如：如果想创建一个 `FileNotFoundError` 实例来告诉用户某个文件不存在，代码如下：

```
#include "llvm/Support/Error.h"
#include <system_error>
// In the header file…
struct FileNotFoundError : public ErrorInfo<FileNoteFoundError>
{
```

```
  StringRef FileName;
  explicit FileNotFoundError(StringRef Name) : FileName(Name)
  {}
  static char ID;
  std::error_code convertToErrorCode() const override {
    return std::errc::no_such_file_or_directory;
  }
  void log(raw_ostream &OS) const override {
    OS << FileName << ": No such file";
  }
};
// In the CPP file…
char FileNotFoundError::ID = 0;
```

实现自定义 Error 实例有几个要求:

- 从 ErrorInfo<T> 类派生，其中 T 是自定义类型。
- 声明唯一的 ID 变量。本例中，我们使用一个静态类成员变量。
- 实现 convertToErrorCode,这个方法为 Error 实例指定 std::error_code 实例。std::error_code 是 C++ 标准库中使用的错误类型 (从 C++11 开始)。请参考 C++ 参考文档获取 (预定义的)std::error_code 的实例。
- 实现 log 的方式打印出错误消息。

要创建 Error 实例，可以使用 make_error 函数:

```
Error NoSuchFileErr = make_error<FileNotFoundError>("foo.txt");
```

make_error 函数接受一个 Error 类 (在本例中是 FileNotFoundError 类) 作为模板参数和函数参数 (在本例中是 foo.txt)。这些将传递给其构造函数。

如果尝试运行上面的代码 (在调试构建中)，而不对 NoSuchFileErr 变量做任何操作，程序将会崩溃，并显示如下错误消息:

```
Program aborted due to an unhandled Error:
foo.txt: No such file
```

结果是，每个 Error 实例都需要在其生命周期结束之前 (即在调用其析构函数方法时) 进行检查和处理。

先解释一下检查 Error 实例意味着什么。除了表示真实的错误外，Error 类还可以表示成功状态，即没有错误。为了更具体地了解这一点，许多 LLVM API 都有以下错误处理的结构:

```
Error readFile(StringRef FileName) {
  if (openFile(FileName)) {
    // Success
    // Read the file content…
    return ErrorSuccess();
  } else
  return make_error<FileNotFoundError>(FileName);
```

```
}
```

注意，百分之百确定它处于成功状态，也要需要检查 Error 实例，否则程序仍然会中止运行。

前面的代码段提供了一个很好过渡到处理 Error 实例的例程。如果一个 Error 实例代表了一个真实的错误，就需要使用一个特殊的 API 来处理它:handleErrors。下面是如何使用它:

```
Error E = readFile(…);
if (E) {
  // TODO: Handle the error
} else {
    // Success!
}
```

handleErrors 函数获得 Error 实例的所有权 (通过 std::move(E))，并使用提供的 lambda 函数来处理错误。我们注意到 handleErrors 会返回一个 Error 实例，来表示未处理的错误。这是什么意思呢？

前面的 readFile 函数示例中，返回的 Error 实例，既可以表示 Success 状态，也可以表示 FileNotFoundError 状态。我们可以稍微修改这个函数，当打开的文件为空时，返回一个 FileEmpty Error 实例:

```
Error E = readFile(…);
if (E) {
  Error UnhandledErr = handleErrors(
  std::move(E),
  [&](const FileNotFoundError &NotFound) {
    NotFound.log(errs() << "Error occurred: ");
    errs() << "\n";
  });
  …
}
```

现在，readFile 返回的 Error 实例可以是一个 Success 状态，或一个 FileNotFoundError 实例，再或一个 FileEmptyError 实例。然而，我们之前写的 handleErrors 代码，只能处理 FileNotFoundError 的情况.

因此，我们需要使用以下代码来处理 FileEmptyError 的情况:

```
Error readFile(StringRef FileName) {
  if (openFile(FileName)) {
    // Success
    …
    if (Buffer.empty())
      return make_error<FileEmptyError>();
    else
      return ErrorSuccess();
  } else
    return make_error<FileNotFoundError>(FileName);
}
```

注意，在使用 handleErrors 时，需要获取 Error 实例的所有权。

或者，可以通过为每种错误类型使用多个 lambda 函数参数将两个 handleErrors 函数调用合并为一个:

```
Error E = readFile(…);
if (E) {
  Error UnhandledErr = handleErrors(
  std::move(E),
  [&](const FileNotFoundError &NotFound) {…},
  [&](const FileEmptyError &IsEmpty) {…});
  …
}
```

换句话说，handleErrors 函数的作用类似于 Error 实例的 switch 语句，运行起来像下面的伪代码一样:

```
Error E = readFile(…);
if (E) {
  switch (E) {
    case FileNotFoundError: …
    case FileEmptyError: …
    default:
      // generate the UnhandledError
  }
}
```

现在，由于 handleErrors 总是返回一个表示未处理错误的 Error，就不能忽略返回的实例，否则程序将中止运行，这时我们应该如何结束这个“错误处理链”？有两种方法，让我们来了解一下:

- 如果 100% 确信已经处理了所有可能的错误类型——这意味着未处理的 Error 变量处于 Success 状态——可以使用 cantFail 函数进行断言:

  ```
  if (E) {
    Error UnhandledErr = handleErrors(
      std::move(E),
      [&](const FileNotFoundError &NotFound) {…},
      [&](const FileEmptyError &IsEmpty) {…});
    cantFail(UnhandledErr);
  }
  ```

  如果 UnhandledErr 仍然包含一个错误，cantFail 函数将中止程序执行，并打印一条错误消息。

- 一个更优雅的解决方案是使用 handleAllErrors 函数:

  ```
  if (E) {
    handleAllErrors(
    std::move(E),
    [&](const FileNotFoundError &NotFound) {…},
    [&](const FileEmptyError &IsEmpty) {…});
    …
  }
  ```

这个函数不会返回任何东西，假设给定的 lambda 函数足以处理所有可能的错误类型。当然，如果缺少，handleAllErrors 仍然会中止程序。

现在，我们已经了解了如何使用 Error 类，以及如何正确处理错误。虽然 Error 的设计乍一看似乎有些烦 (也就是说，我们需要处理所有可能的错误类型，否则执行将中途中止)，但这些限制可以减少程序员犯的错误数量，并创建更健壮的程序。

接下来，我们将介绍另外两个工具类，它们可以进一步提升 LLVM 中的错误处理表达式。

11.6. 了解 Expected 和 ErrorOr 类

在 LLVM 的代码库中，经常可以看到这样的编码模式: 如果出现问题，API 希望返回结果或错误。LLVM 试图通过创建在单个对象中多重处理结果和错误的工具类 (它们是 Expected 和 ErrorOr 类)，使这种模式更易于使用。

11.6.1 Expected 类

Expect 类自带一个 Success 结果，或一个错误——例如，LLVM 中的 JSON 库使用它来表示解析传入字符串的结果:

```
#include "llvm/Support/JSON.h"
using namespace llvm;
…
// `InputStr` has the type of `StringRef`
Expected<json::Value> JsonOrErr = json::parse(InputStr);
if (JsonOrErr) {
  // Success!
  json::Value &Json = *JsonOrErr;
  …
} else {
  // Something goes wrong…
  Error Err = JsonOrErr.takeError();
  // Start to handle `Err`…
}
```

前面的 JsonOrErr 类的类型是 Expected<json::Value>。这意味着这个 Expected 变量要么带了一个 json::Value 类型的 Success 结果，要么带了一个错误，这个错误由 Error 类表示。

与 Error 类相同，每个 Expected 实例都需要检查。如果它表示错误，这个 Error 实例也需要处理。为了检查期望值实例的状态，我们还可以将其转换为布尔类型。然而，与 Error 不同的是，如果一个 Expected 实例包含一个 Success 结果，在转换为布尔值后将为真。

如果 Expected 实例表示 Success 结果，则可以使用 * 操作符、-> 操作符或 get 方法获取结果。要不，可以在处理 Error 实例之前使用 takeError 方法来检索错误 (使用在前一节中学习到的技能)。

或者，如果确定预期实例处于 Error 状态，可以通过使用 errorIsA 方法来确定底层的错误类型，而无需检索底层的 Error 实例，例如：下面的代码检查错误是否是一个 FileNotFoundError 实例 (我们在前一节中创建的):

```
if (JsonOrErr) {
  // Success!
  ...
} else {
  // Something goes wrong...
  if (JsonOrErr.errorIsA<FileNotFoundError>()) {
    ...
  }
}
```

这些是使用个 Expected 变量的技巧。要创建一个 Expected 实例，最常见的方法是使用 Expected 的隐式类型转换:

```
Expected<std::string> readFile(StringRef FileName) {
  if (openFile(FileName)) {
    std::string Content;
    // Reading the file...
    return Content;
  } else
    return make_error<FileNotFoundError>(FileName);
}
```

上面的代码表明，在出现错误的情况下，我们可以简单地返回一个 Error 实例，该实例将隐式地转换为表示该错误的 Expect 实例。类似地，如果一切都很顺利，那么 Success 结果 (本例中是 std::string 类型变量 Content) 也将隐式地转换为具有 Success 状态的 Expect 实例。

现在，我们已经了解了如何使用 Expected 类了。本节的最后一部分将展示如何使用它的兄弟类:ErrorOr。

11.6.2 ErrorOr 类

ErrorOr 类使用的方式与 Expected 类几乎相同——要么是一个 Success 结果，要么是一个错误。与 Expected 类不同，ErrorOr 使用 std::error_code 来表示错误。下面是使用 MemoryBuffer API 读取 foo.txt，并将其内容存储到 MemoryBuffer 对象的例子:

```
#include "llvm/Support/MemoryBuffer.h"
...
ErrorOr<std::unique_ptr<MemoryBuffer>> ErrOrBuffer
= MemoryBuffer::getFile("foo.txt");
if (ErrOrBuffer) {
  // Success!
  std::unique_ptr<MemoryBuffer> &MB = *ErrOrBuffer;
} else {
  // Something goes wrong...
  std::error_code EC = ErrOrBuffer.getError();
  ...
}
```

上面的代码显示了类似的结构，Expect 的示例代码中:std::unique_ptr<memorybuffer> 实例是这里的成功结果类型。我们也可以在检查 ErrOrBuffer 的状态后，使用 * 操作符对其进行检索。

唯一的区别是，如果 ErrOrBuffer 处于 Error 状态，则错误由 std::error_code 实例表示，而不是 Error。开发人员没有必要去处理 std::error_code 实例——换句话说，他们可以忽略这个错误，这可能会增加其他开发人员在代码中犯错的机会。尽管如此，使用 ErrorOr 类可以让开发者与 C++ 标准库 API 有更好的互动，因为它们中很多都使用 std::error_code 来表示错误。关于如何使用 std::error_code 的详细信息，可以参考 C++ 的文档。

最后，为了创建 ErrorOr 实例，我们使用了与 Expect 类利用隐式转换相同的技巧:

```
#include <system_error>
ErrorOr<std::string> readFile(StringRef FileName) {
  if (openFile(FileName)) {
    std::string Content;
    // Reading the file···
    return Content;
  } else
    return std::errc::no_such_file_or_directory;
}
```

std::errc::no_such_file_or_directory 对象是 system_error 头文件中预定义的 std::error_code 对象。

本节中，我们了解了如何使用 LLVM 提供的错误处理工具，这些工具是对未处理的错误施加严格规则的 Error 类，以及 Expected 和 ErrorOr 类，它们为我们提供了一种简便的方法，将程序结果和错误状态以多路复用的方式放在了单个对象中。在使用 LLVM 进行开发时，这些工具可以帮助我们编写富有表现力，且健壮的错误处理代码。

11.7. 总结

本章中，我们了解了许多有用的实用工具，它们可以在使用 LLVM 进行开发时提高我们的工作效率。其中一些 (如优化注释或计时器) 对诊断 LLVM 引起的问题很有用，而另一些 (如 Error 类) 可以帮助构建更健壮的代码，这些代码可以很好地适应我们自定义编译器的复杂性。

本书的最后一章中，我们将了解配置文件引导的优化 (PGO) 和杀毒器的开发，这是读者们不能错过的高级主题哦！

第 12 章 LLVM IR 表达式

前一章中，我们了解了如何在使用 LLVM 进行开发时利用各种工具来提高工作效率，这些技能可以让我们在诊断出 LLVM 的问题时，处理起来更加顺畅。其中一些工具可以减少编译器工程师犯错的机会。本章中，我们将了解在 LLVM IR 中工具是如何工作的。

我们在这里所指的**工具**是一种技术，将一些检测插入到我们正在编译的代码中，以收集运行时信息，例如：可以收集关于某个函数调用多少次的信息——只有在目标程序执行之后才可用。这种技术的优点是提供了关于目标程序行为的准确信息，这些信息可以以几种不同的方式使用，例如：可以使用收集到的值，再次编译和优化相同的代码——但是这一次，由于有准确的数据，可以执行以前无法执行的优化。这种技术也称为**数据导向优化 (Profile-Guided Optimization，PGO)**。在另一个例子中，将使用插入的检测来捕获运行时不希望发生的事件——缓冲区溢出、条件竞争和双重释放内存等等。用于此目的的检测器，也称为**杀毒器**。

要在 LLVM 中实现相应的工具，不仅需要 LLVM Pass 的帮助，还需要 LLVM-**Clang**、**LLVM IR Transformation** 和 **Compiler-RT** 中的多个子项目之间的协同。在本章中，我们将介绍 Compiler-RT，更重要的是，了解如何将这些子系统组合在一起，从而达到测试的目的。

下面是我们将要讨论的内容:

- 开发杀毒器
- 使用 PGO

本章的第一部分，将看到如何在 Clang 和 LLVM 中实现杀毒器，然后自己创建一个简单的杀毒器。本章的后半部分将展示如何在 LLVM 中使用 PGO 框架，以及如何扩展它。

12.1. 相关准备

本章中，将使用多个子项目。其中之一——Compiler-RT——需要通过修改 CMake 配置来包含在你的构建中。请打开 `CMakeCache.txt` 文件，并将 `compiler-rt` 字符串添加到 `LLVM_ENABLE_PROJECTS` 变量中。下面是一个例子:

```
//Semicolon-separated list of projects to build…
LLVM_ENABLE_PROJECTS:STRING="clang;compiler-rt"
```

编辑该文件后，用任何构建目标重新启动构建，CMake 会重新配置。

当一切就绪，就可以构建本章所需的组件了。下面是一个命令示例:

```
$ ninja clang compiler-rt opt llvm-profdata
```

这将构建我们都熟悉的 `clang` 工具和一系列 Compiler-RT 库。

可以在 GitHub 库中找到本章的示例代码: https://github.com/PacktPublishing/LLVM-Techniques-Tips-and-Best-Practices-Clang-and-Middle-End-Libraries/tree/main/Chapter12.

12.2. 开发杀毒器

杀毒器是一种检查编译器插入的代码 (检测) 的某些运行时技术。人们通常使用杀毒器来确保程序的正确性或执行安全策略。为了了解消毒剂是如何工作的，让我们以 Clang 中最流行的杀毒器为例——**地址杀毒器 (address sanitizer)**。

12.2.1 使用地址杀毒器的例子

假设我们有一些简单的 C 代码，例如:

```
int main(int argc, char **argv) {
  int buffer[3];
  for (int i = 1; i < argc; ++i)
    buffer[i-1] = atoi(argv[i]);

  for (int i = 1; i < argc; ++i)
    printf("%d ", buffer[i-1]);
  printf("\n");
  return 0;
}
```

前面的代码将命令行参数转换为整数，并将它们存储在大小为 3 的缓冲区中。然后，把它们打印出来。

应该能够很容易地发现一个突出的问题: 当 `argc` 的值大于 3(缓冲区的大小) 时，它的值可以任意大。这里，会将值存储在一个无效的内存位置。然而，编译这段代码时，编译器什么都不会报告。下面是一个例子:

```
$ clang -Wall buffer_overflow.c -o buffer_overflow
$ # No error or warning
```

命令中，即使我们通过`-Wall` 标志启用了所有的编译器警告，`clang` 也不会抱怨存在的错误。

如果尝试执行 `buffer_overflow` 时，当我们向它传递三个以上的命令行参数后，程序将会崩溃:

```
$ ./buffer_overflow 1 2 3
1 2 3
$ ./buffer_overflow 1 2 3 4
Segmentation fault (core dumped)
$
```

更糟糕的是，崩溃 `buffer_overflow` 的命令行参数的数量实际上因机器而异，那么调试就会更加困难。总之，我们在这里遇到的问题是:`buffer_overflow` 只会在某些输入上出错，而编译器

未能捕捉到这个问题。

现在，让我们试着用一个地址杀毒器来捕捉这个 bug。下面的命令要求 `clang` 用地址杀毒器编译相同的代码:

```
$ clang -fsanitize=address buffer_overflow.c -o san_buffer_overflow
```

让我们再次执行这个程序。输出如下:

```
$ ./san_buffer_overflow 1 2 3
1 2 3
$ ./san_buffer_overflow 1 2 3 4
=================================================================
===
==137791==ERROR: AddressSanitizer: stack-buffer-overflow on
address 0x7ffea06bccac at pc 0x0000004f96df bp 0x7ffea06bcc70…
WRITE of size 4 at 0x7ffea06bccac thread T0
…
  This frame has 1 object(s):
    [32, 44) 'buffer' <== Memory access at offset 44 overflows this variable
…
==137791==ABORTING
$
```

除了崩溃，地址杀毒器还向我们提供了关于运行时提出的问题的许多细节: 杀毒器告诉我们，检测到堆栈上的缓冲区溢出，这可能是缓冲区变量。

这些信息非常有用。假设正在处理一个复杂得多的软件项目时，当一个奇怪的内存错误发生时，而不是仅仅崩溃或改变程序的逻辑，地址杀毒器可以指出有问题的区域，且具有很高的准确性。

为了更深入地了解它的机制，下面的图表说明了地址杀毒器如何检测缓冲区溢出:

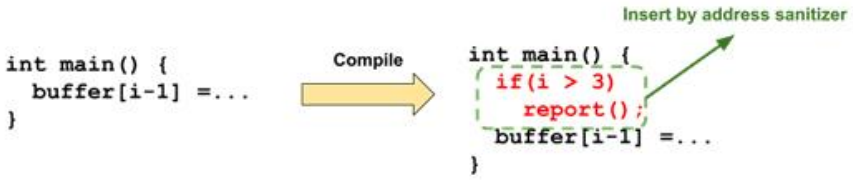

图 12.1 -地址杀毒器插入的检测代码

这里，可以看到地址杀毒器正在有效地将边界检查，插入到用于访问缓冲区的数组索引中。通过这个检查 (将在运行时执行)，目标程序可以在违反内存访问之前带着错误详细信息退出。更一般

地说，在编译期间，杀毒器将插入一些检测代码 (到目标程序中)，这些代码最终将在运行时执行，以检查或保护某些属性。

使用地址杀毒器检测溢出

上面的图表显示了地址杀毒器工作原理的简化版本。实际上，地址杀毒器会利用多种策略来监视程序中的内存访问，例如：地址杀毒器可以使用一个特殊的内存分配器来分配内存，并在无效的内存区域放置陷阱。

虽然地址杀毒器专门用于捕获非法内存访问，但 **ThreadSanitizer** 可以用于捕获数据竞争条件，也就是多个线程对同一块数据的无效访问。Clang 中的杀毒器的其他例子是 **LeakSanitizer**，用于检测敏感数据，如：密码泄露。以及 **MemorySanitizer**，用于检测对未初始化内存的读取。

当然，使用杀毒器也有一些缺点。最突出的问题是性能影响: 以线程杀毒器 (Clang 内置的) 为例，用一个线程杀毒器编译的程序比原始版本慢 5 15 倍。而且，由于杀毒器会向程序中插入额外的代码，这可能会阻碍一些优化机会，甚至会影响原始程序的逻辑! 换句话说，这是在目标程序的健壮性和性能之间的权衡。

至此，我们了解了杀毒器的概念。现在，可以尝试自己创建一个杀毒器，以理解 Clang 和 LLVM 如何实现杀毒器器。下一节包含的代码比前几章的任何示例都要多，更不用说 LLVM 中不同子项目之间的变更了。为了关注最重要的知识点，我们不会深入一些支持代码的细节——例如，对 CMake 构建脚本所做的更改。相反，我们将提供一个简短的介绍，并指出在本书的 GitHub 库中，你可以在哪里找到它。

让我们从概述将要创建的项目开始。

12.2.2 创建循环计数器杀毒器

为了 (稍微) 简化我们的任务，将要创建的杀毒器 (循环计数器杀毒器，简称 **LPCSan**) 看起来就像杀毒器，只是它不检查任何重要的程序属性。相反，我们希望使用它来打印一个循环的真实的、具体的计数 (迭代次数)，这只能在运行时可用。

例如，我们有以下输入代码:

```
void foo(int S, int E, int ST, int *a) {
  for (int i = S; i < E; i += ST) {
    a[i] = a[i + 1];
  }
}
int main(int argc, char **argv) {
  int start = atoi(argv[1]),
      end = atoi(argv[2]),
      step = atoi(argv[3]);
  int a[100];
  foo(start, end, step, a);
  return 0;
}
```

我们可以使用 LPCSan 编译:

```
$ clang -O1 -fsanitize=loop-counter test_lpcsan.c -o test_lpcsan
```

注意，编译时优化值大于-O0 是必要的，稍后会解释原因。

当执行 test_lpcsan(带有一些命令行参数) 时，我们可以在 foo 函数中打印出循环计数:

```
$ ./test_lpcsan 0 100 1
==143813==INFO: Found a loop with trip count 100
$ ./test_lpcsan 0 50 2
==143814==INFO: Found a loop with trip count 25
$
```

以上代码的消息，是由我们的杀毒程序代码打印的。

现在，让我们深入了解创建 LPCSan 的步骤。我们将把这个过程分为三个部分:

- 开发 IR 转换
- 添加 Compiler-RT 组件
- 将 LPCSan 加入 Clang

我们将从这个杀毒器的 IR 转换部分开始。

开发 IR 转换

前面，我们了解了地址杀毒器——或者只是一般的杀毒器——通常将代码插入到目标程序中，以检查特定的运行时属性或收集数据。在第 9 章和第 10 章中，我们了解了如何修改/转换 LLVM IR，包括在其中插入新代码，所以这对于制作我们的 LPCSan，是一个很好的入手点。

本节中，我们将开发一个名为 LoopCounterSanitizer 的 LLVM Pass，它将插入特殊的函数调用来收集模块中每个循环的准确行程计数。具体步骤如下:

1. 首先，让我们创建两个文件:llvm/lib/Transforms/Instrumentation 文件夹下的 LoopCounterSanitizer.cpp 和 llvm/include/llvm/Transforms/Instrumentation 文件夹下的相应头文件。头文件中，我们将放置这个 Pass 的声明:

```
struct LoopCounterSanitizer
: public PassInfoMixin<LoopCounterSanitizer> {
  PreservedAnalyses run(Loop&, LoopAnalysisManager&,
              LoopStandardAnalysisResults&,
              LPMUpdater&);
private:
  // Sanitizer functions
  FunctionCallee LPCSetStartFn, LPCAtEndFn;
  void initializeSanitizerFuncs(Loop&);
};
```

上面的代码展示了我们在第 10 章中看到的典型循环传递结构。值得注意的变化是 LPCSetStartFn 和 LPCAtEndFn 内存变量——将存储收集循环次数的 Function 实例 (FunctionCallee 是一个包装函数，只提供额外的函数签名信息)。

2. 最后，在 LoopCounterSanitizer.cpp 中，我们实现了 Pass 的框架代码:

```
PreservedAnalyses
LoopCounterSanitizer::run(Loop &LP, LoopAnalysisManager
&LAM, LoopStandardAnalysisResults &LSR, LPMUpdater &U) {
  initializeSanitizerFuncs(LP);
  return PreservedAnalyses::all();
}
```

以上代码中的 initializeSanitizerFuncs 方法将赋予 LPCSetStartFn 和 LPCAtEndFn。在我们深入 initializeSanitizerFuncs 的细节之前，让了解一下 LPCSetStartFn 和 LPCAtEndFn。

3. 为了计算出准确的循环计数，将使用存储在 LPCSetStartFn 中的 Function 实例来收集循环的初始引导变量值。另一方面，存储在 LPCAtEndFn 中的 Function 实例，将用于收集最终的归纳变量值和循环的步长值。为了给有一个具体的概念，了解这两个函数实例是如何一起工作的，我们假设有以下伪代码作为输入程序:

```
void foo(int S, int E, int ST) {
  for (int i = S; i < E; i += ST) {
    ...
  }
}
```

前面的代码中，S、E 和 ST 变量分别表示循环的初始值、最终值和步长值。LoopCounterSanitizer Pass 的目标是插入 LPCSetStartFn 和 LPCAtEndFn，方法如下:

```
void foo(int S, int E, int ST) {
  for (int i = S; i < E; i += ST) {
    lpc_set_start(S);
    ...
    lpc_at_end(E, ST);
  }
}
```

以上代码中的 lpc_set_start 和 lpc_at_end 分别是存储在 LPCSetStartFn 和 lpccatendfn 的函数实例中。下面是这两个函数可能的 (伪) 实现:

```
static int CurrentStartVal = 0;
void lpc_set_start(int start) {
  CurrentStartVal = start;
}
void lpc_at_end(int end, int step) {
  int trip_count = (end - CurrentStartVal) / step;
  printf("Found a loop with trip count %d\n",
  trip_count);
}
```

现在我们已经了解了 LPCSetStartFn 和 LPCAtEndFn 的角色，接下来看看 initializeSanitizerFuncs 是如何初始化它们的。

4. 下面是 initializeSanitizerFuncs 内部代码:

```
void LoopCounterSanitizer::initializeSanitizerFuncs(Loop
&LP) {
  Module &M = *LP.getHeader()->getModule();
  auto &Ctx = M.getContext();
  Type *VoidTy = Type::getVoidTy(Ctx),
       *ArgTy = Type::getInt32Ty(Ctx);
  LPCSetStartFn
    = M.getOrInsertFunction("__lpcsan_set_loop_start",
                            VoidTy, ArgTy);
  LPCAtEndFn = M.getOrInsertFunction("__lpcsan_at_loop_
    end", VoidTy, ArgTy, ArgTy);
}
```

前面的代码基本上是从模块中获取两个函数，__lpcsan_set_loop_ start 和__lpcsan_at_loop_end，并分别将它们的函数实例存储在 LPCSetStartFn 和 LPCAtEndFn 中。

Module::getOrInsertFunction 要么从模块中获取给定函数名的 Function 实例，要么创建一个不存在的 Function 实例。如果是一个新创建的实例，它有一个空的函数体；换句话说，这就是一个函数声明。

同样值得注意的是，Module::getOrInsertFunction 的第二个参数是 Function 查询的返回类型。其余的 (getOrInsertFunction 参数) 表示该 Function 的参数类型。

设置了 LPCSetStartFn 和 LPCAtEndFn 后，我们来了解一下如何将它们插入到 IR 中的正确位置。

5. 在第 10 章中，我们了解了几个用于处理 Loop 的工具类。其中之一就是 LoopBounds，可以给出一个循环的边界。可以通过包含归纳变量的开始、结束和步长值来实现这一点，这正是我们要寻找的信息。下面是检索 LoopBounds 实例的代码:

```
PreservedAnalyses
LoopCounterSanitizer::run(Loop &LP, LoopAnalysisManager
&LAM, LoopStandardAnalysisResults &LSR, LPMUpdater &U) {
  initializeSanitizerFuncs(LP);
  ScalarEvolution &SE = LSR.SE;

  using LoopBounds = typename Loop::LoopBounds;
  auto MaybeLB = LP.getBounds(SE);
  if (!MaybeLB) {
    errs() << "WARNING: Failed to get loop bounds\n";
    return PreservedAnalyses::all();
  }
  LoopBounds &LB = *MaybeLB;
  …
  Value *StartVal = &LB.getInitialIVValue(),
      *EndVal = &LB.getFinalIVValue(),
      *StepVal = LB.getStepValue();
}
```

上面代码中的 Loop::getBounds 返回一个 Optional<LoopBounds> 实例。Optional<T> 类是一个有用的容器，要么存储 T 类型的实例，要么为空。可以把它看作是空指针的替换: 通常，用 T* 来表示计算结果，空指针时则为空值。但是，如果开发者忘记先检查空指针，就会有对空指针进行解引用的风险。Optional<T> 类就没有这个问题。

使用 LoopBounds 实例，我们可以检索归纳变量的范围，并将其存储在 StartVal、EndVal 和 StepVal 变量中。

6. StartVal 是由__lpcsan_set_loop_ start 收集的 Value 实例，而__lpcsan_at_loop_end 将在运行时收集 EndVal 和 StepVal。现在，问题是，我们应该在哪里插入函数__lpcsan_set_loop_start 和__lpcsan_at_loop_end 来正确地收集这些值呢？

   根据经验，需要将这些函数调用插入到这些值的定义之后。虽然我们可以找到定义这些值的确切位置，但我们可以在一些固定的位置 (目标值总是可用的位置) 插入工具函数来尝试简化这个问题。

   对于__lpcsan_set_loop_start，我们将它插入到**循环头**块的末尾，因为初始的感应变量值永远不会在这个块之后定义:

```
// Inside LoopCounterSanitizer::run …
…
BasicBlock *Header = LP.getHeader();
Instruction *LastInst = Header->getTerminator();
IRBuilder<> Builder(LastInst);
Type *ArgTy = LPCSetStartFn.getFunctionType()-
>getParamType(0);
if (StartVal->getType() != ArgTy) {
  // cast to argument type first
  StartVal = Builder.CreateIntCast(StartVal, ArgTy,
  true);
}
Builder.CreateCall(LPCSetStartFn, {StartVal});
…
```

   上面的代码中，我们使用 getTerminator 从头代码块中获取最后一个指令。然后，使用 IRBuilder<>——作为最后一条指令作为插入点——来插入新的 Instruction 实例。

   在将 StartVal 作为参数传递给新的__lpcsan_set_loop_ start 函数之前，我们需要将它的 IR 类型 (由 Type 类表示) 转换为一个兼容的类型。IRBuilder::CreateInstCast 是一个方便的工具，根据给定的 Value 和 Type 实例，自动生成一条扩展整数位宽的指令，或者一条截断位宽的指令。

   最后，可以通过 IRBuilder::CreateCall 创建一个对__lpcsan_set_loop_start 的函数调用，使用 StartVal 作为函数参数。

7. 对于__lpcsan_at_loop_end，使用相同的技巧来收集 EndVal 和 StepVal 的运行时值:

```
BasicBlock *ExitBlock = LP.getExitBlock();
Instruction *FirstInst = ExitBlock->getFirstNonPHI();
IRBuilder<> Builder(FirstInst);
FunctionType *LPCAtEndTy = LPCAtEndFn.getFunctionType();
```

```
Type *EndArgTy = LPCAtEndTy->getParamType(0),
     *StepArgTy = LPCAtEndTy->getParamType(1);

if (EndVal->getType() != EndArgTy)
  EndVal = Builder.CreateIntCast(EndVal, EndArgTy, true);
if (StepVal->getType() != StepArgTy)
  StepVal = Builder.CreateIntCast(StepVal, StepArgTy,
    true);

Builder.CreateCall(LPCAtEndFn, {EndVal, StepVal});
```

与前面的步骤不同，我们将函数调用插入到退出块的开始处的`__lpcsan_at_loop_end`。这是因为可以期望在离开循环之前定义引导变量的结束值和步长值。

这些是 `LoopCounterSanitizer` Pass 的所有实现细节。

8. 结束本节之前，我们还需要编辑几个文件以确保一切正常。请查看本章示例代码文件夹中的 `Changes-LLVM.diff` 文件。以下是在其他支持文件中所做的修改:

   i. `llvm/lib/Transforms/Instrumentation/CMakeLists.txt` 中的更改: 将新 Pass 源文件添加到构建中。

   ii. `llvm/lib/Passes/PassRegistry.def` 中的变化: 将的 Pass 添加到可用的 Pass 列表中，这样我们就可以使用 `opt` 来测试它。

至此，我们完成了对 LLVM 部分的所有必要修改。

在继续下一节之前，让我们测试一下新创建的 `LoopCounterSanitizer` Pass。我们将使用本节前面看到的相同的 C 代码，下面是包含我们想要检测的循环的函数:

```
void foo(int S, int E, int ST, int *a) {
  for (int i = S; i < E; i += ST) {
    a[i] = a[i + 1];
  }
}
```

请注意，虽然我们没有明确检查 Pass 中的循环形式，但在通道中使用的一些 API 实际上需要旋转循环，所以请生成具有 `O1` 优化级别的 LLVM IR 代码，以确保循环旋转的 Pass 生效:

下面是 `foo` 函数的 (简化)LLVM IR:

```
define void @foo(i32 %S, i32 %E, i32 %ST, i32* %a) {
  %cmp9 = icmp slt i32 %S, %E
  br i1 %cmp9, label %for.body.preheader, label %for.cond.
  cleanup
for.body.preheader:
  %0 = sext i32 %S to i64
  %1 = sext i32 %ST to i64
  %2 = sext i32 %E to i64
  br label %for.body
  ...
for.body:
  %indvars.iv = phi i64 [ %0, %for.body.preheader ], [
  %indvars.iv.next, %for.body ]
```

```
  ...
  %indvars.iv.next = add i64 %indvars.iv, %1
  %cmp = icmp slt i64 %indvars.iv.next, %2
  br i1 %cmp, label %for.body, label %for.cond.cleanup
}
```

标签是该循环的头文件和循环体块。由于这个循环已旋转，`for.body` 块是这个循环的头、锁存器和退出块。

现在，让使用下面的命令来转换这个 IR:

```
$ opt -S  -passes="loop(lpcsan)" input.ll -o -
```

在`-passes` 命令行选项中,要求 `opt` 运行我们的 `LoopCounterSanitizer` Pass(名称为 `lpcsan`，在 `PassRegistry.def` 文件中注册)。封闭的 `loop(…)` 字符串只是告诉 `opt`，`lpcsan` 是一个循环 Pass(实际上可以省略这个修饰，因为 `opt` 在大多数情况下可以找到正确的 Pass)。

以下是简化后的结果:

```
declare void @__lpcsan_set_loop_start(i32)
declare void @__lpcsan_at_loop_end(i32, i32)

define void @foo(i32 %S, i32 %E, i32* %a) {
  %cmp8 = icmp slt i32 %S, %E
  br i1 %cmp8, label %for.body.preheader, label %for.cond.
cleanup

for.body.preheader:
  %0 = sext i32 %S to i64
  %wide.trip.count = sext i32 %E to i64
  br label %for.body

for.cond.cleanup.loopexit:
  %1 = trunc i64 %wide.trip.count to i32
  call void @__lpcsan_at_loop_end(i32 %1, i32 1)
  br label %for.cond.cleanup

for.body:
  ...
  %3 = trunc i64 %0 to i32
  call void @__lpcsan_set_loop_start(i32 %3)
  br i1 %exitcond.not, label %for.cond.cleanup.loopexit, label
    %for.body
}
```

`__lpcsan_set_loop_start` 和`__lpcsan_at_loop_end` 已经分别正确地插入了头区块和退出区块。他们也在收集与循环行程计数相关的期望值。

现在，最大的问题是:`__lpcsan_set_loop_start` 和`__lpcsan_at_loop_end` 的函数体在哪里？

两者在前面的 IR 代码中只有声明。

下一节中，我们将使用 Compiler-RT 来回答这个问题。

添加 Compiler-RT 组件

Compiler-RT 表示**编译器运行时**。运行时的用法在这里有点模棱两可，因为在普通的编译管道中有太多的东西可以称为运行时。但 Compiler-RT 确实包含了用于完全不同任务的库，这些库的共同之处在于，它们为目标程序提供了补充代码，以实现在其他情况下不存在的增强特性或功能。要记住，Compiler-RT 库不是用来构建编译器或相关工具的——它们应该与我们正在编译的程序相链接。

Compiler-RT 中最常用的特性之一是**内置函数**。正如你可能听说过的，现在越来越多的计算机体系结构支持向量运算。也就是说，在硬件的支持下，可以同时处理多个数据元素。下面是一些用 C 编写的使用向量操作的示例代码:

```
typedef int v4si __attribute__((__vector_size__(16)));
v4si v1 = (v4si){1, 2, 3, 4};
v4si v2 = (v4si){5, 6, 7, 8};
v4si v3 = v1 + v2; // = {6, 8, 10, 12}
```

上面的代码使用了一种非标准化的 (目前，只能在 Clang 和 GCC 中使用这种语法)C/C++ 向量扩展来声明两个向量 `v1` 和 `v2`，然后将它们加和到生成的第三个向量中。

在 x86-64 平台上，这段代码将编译为使用向量指令集的代码，比如：SSE 或 AVX。在 ARM 平台上，生成的二进制文件可能使用 NEON 向量指令集。但是如果你的目标平台没有向量指令集怎么办？最明显的解决方案是用可用的指令“合成”这些不受支持的操作，例如：在这种情况下，我们应该编写一个 `for` 循环来替换向量求和。更具体地说，每当编译时看到一个向量求和，我们就用一个函数来替换它，这个函数包含了 `for` 循环的合成实现。函数体可以放在任何地方，只要它最终与程序链接即可。下图说明了这一过程:

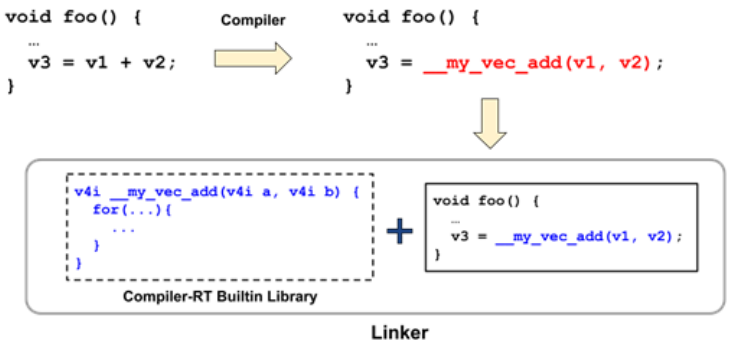

图 12.2 - 内置 Compiler-RT 的工作流程

这里显示的工作流类，似于我们在 LPCSan 中的需求: 前一节中，我们开发了一个 LLVM Pass，该 Pass 插入了额外的函数调用来收集循环行程计数，但是我们仍然需要实现这些收集器函数。如果使用上图所示的工作流，我们可以得出如下图所示的设计:

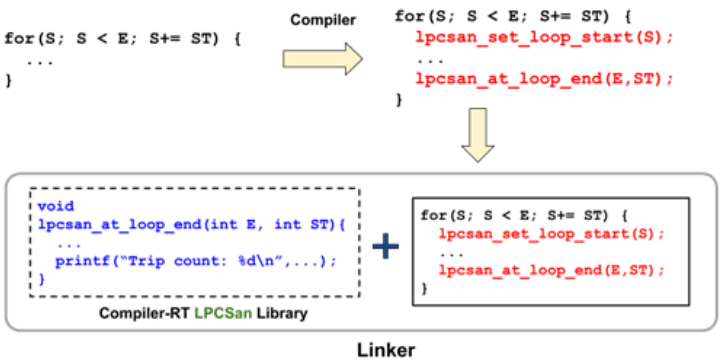

图 12.3 - Compiler-RT LPCSan 组件的工作流程

上图显示了`__lpcsan_set_loop_start`和`__lpcsan_at_loop_end`的函数体放在一个 Compiler-RT 库中，这个库最终将与最终的二进制文件相链接。在这两个函数中，我们使用输入参数计算迭代计数，并输出结果。本节的其余部分中，我们将展示如何为 LPCSan 创建这样的 Compiler-RT 库。让我们开始:

- 首先，将文件夹切换到`llvm-project/compiler-rt`，即 Compiler-RT 的根目录。这个子项目中，我们创建一个名为`lib/lpcsan`的新文件夹，然后再将一个新的`lpcsan.cpp`文件放入其中。在这个文件中，创建检测函数的框架:

```
#include "sanitizer_common/sanitizer_common.h"
#include "sanitizer_common/sanitizer_internal_defs.h"
using namespace __sanitizer;

extern "C" SANITIZER_INTERFACE_ATTRIBUTE
void __lpcsan_set_loop_start(s32 start){
  // TODO
}
extern "C" SANITIZER_INTERFACE_ATTRIBUTE
void __lpcsan_at_loop_end(s32 end, s32 step){
  // TODO
}
```

这里有两件事值得注意: 首先，使用 Compiler-RT 提供的原始数据类型，例如：上面的代码中，我们使用`s32`—在`__sanitizer`命名空间下可用—来表示有符号的 32 位整数，而不是普通的整数。这背后的基本原因是，我们可能需要为不同的硬件架构或平台构建 Compiler-RT 库，其中一些`int`的宽度可能不是 32 位。

其次，虽然我们使用 C++ 来实现检测函数，但需要将它们公开为 C 函数，因为 C 函数具有更稳定的**应用程序二进制接口 (ABI)**。因此，请确保在要导出的函数中添加`extern "C"`。宏`SANITIZER_INTERFACE_ATTRIBUTE`也确保函数将正确地暴露在库接口上，所以也请添加这个宏。

- 接下来，向这两个函数添加必要的代码:

```
static s32 CurLoopStart = 0;
```

```
extern "C" SANITIZER_INTERFACE_ATTRIBUTE
void __lpcsan_set_loop_start(s32 start){
  CurLoopStart = start;
}
extern "C" SANITIZER_INTERFACE_ATTRIBUTE
void __lpcsan_at_loop_end(s32 end, s32 step){
  s32 trip_count = (end - CurLoopStart) / step;
  s32 abs_trip_count
    = trip_count >= 0? trip_count : -trip_count;
  Report("INFO: Found a loop with "
         "trip count %d\n", abs_trip_count);
}
```

我们在这里使用的实现非常简单:CurLoopStart 是一个全局变量，存储当前循环的初始索引变量值，并且可以用__lpcsan_set_loop_start 进行更新。

当一个循环完成时，将调用__lpcsan_at_loop_end。当发生这种情况时，在打印结果之前，我们使用存储在 CurLoopStart 中的值，以及 end 和 step 参数来计算当前循环的准确迭代计数。

- 既然我们已经实现了核心逻辑，现在就可以构建这个库了。在 lib/lpcsan 文件夹中，创建 CMakeLists.txt 文件，并插入以下代码:

```
...
set(LPCSAN_RTL_SOURCES
  lpcsan.cpp)
add_compiler_rt_component(lpcsan)

foreach(arch ${LPCSAN_SUPPORTED_ARCH})
  set(LPCSAN_CFLAGS ${LPCSAN_COMMON_CFLAGS})
  add_compiler_rt_runtime(clang_rt.lpcsan
    STATIC
    ARCHS ${arch}
    SOURCES ${LPCSAN_RTL_SOURCES}
        $<TARGET_OBJECTS:RTSanitizerCommon.${arch}>
        $<TARGET_OBJECTS:RTSanitizerCommonLibc.${arch}>
    ADDITIONAL_HEADERS ${LPCSAN_RTL_HEADERS}
    CFLAGS ${LPCSAN_CFLAGS}
    PARENT_TARGET lpcsan)
  ...
endforeach()
```

上面的代码中，只显示了 CMakeLists.txt 中最重要的部分。以下是一些重点:

i. Compiler-RT 有自己的 CMake 宏/函数集。在这里,我们使用它们中的两个,add_compiler_rt_compo
和 add_compiler_rt_runtime，分别为整个 LPCSan 创建一个伪构建目标和库构建目标。

ii. 与传统的构建目标不同，如果杀毒器想在 Compiler-RT 中使用支持/实用程序库 (例如，前面代码中的 RTSanitizerCommon)，通常链接它们的对象文件，而不是它们的库文件。更具体地说，我们可以使用 $<TARGET_OBJECTS:…> 指令导入支持/实用程序组件作为

输入源。

iii. 杀毒库可以支持多个体系结构和平台。在 Compiler-RT 中，我们列举了所有受支持的体系结构，并为每个体系结构创建了一个“杀毒库”。

同样，上面的代码片段只是构建脚本的一小部分。请参考我们的示例代码文件夹中完整的 CMakeLists.txt 文件。

- 为了成功构建 LPCSan，我们仍然需要在 Compiler-RT 中做一些更改。相同文件夹中的 `Base-CompilerRT.diff` 补丁提供了构建杀毒器所需的所有更改。将其应用于 Compiler-RT 的源代码树。以下是这个补丁的摘要:

  i. 改变 `compiler-rt/cmake/config-ix.cmake` 基本上指定了 LPCSan 所支持的体系结构和操作系统。我们在前面的代码片段中看到的 CMake 变量 `LPCSAN_SUPPORTED_ARCH` 就来自这里。

  ii. 整个 `compiler-rt/test/lpcsan` 文件夹实际上是一个占位符。与 LLVM 不同，出于某种原因，Compiler-RT 中的每个杀毒器都需要测试。因此，我们在这里放置一个空的测试文件夹，以通过这个强加的要求。

这些是为我们的 LPCSan 生成 Compiler-RT 组件的所有步骤。

要构建我们的 LPCSan 库，使用以下命令:

```
$ ninja lpcsan
```

不幸的是，在我们修改了 Clang 中的编译流水之前，我们无法测试 LPCSan 库。本节的最后一部分中，我们将了解如何实现这个任务。

将 LPCSan 加入 Clang

前一节中，了解了 Compiler-RT 库如何为目标程序提供补充功能，或使用特殊工具 (例如刚创建的杀毒器) 提供帮助。这一节中，我们将把所有的东西放在一起，这样我们就可以使用 LPCSan 了，只需将`-fsanitize=loop-counter` 标志传递给 `clang` 即可。

回想一下在图 12.3 中，Compiler-RT 库需要与我们正在编译的程序相链接。另外，为了将检测代码插入到目标程序中，我们必须运行 `LoopCounterSanitizer` 循环。本节中，我们将修改 Clang 中的编译流水，以便它在特定时间运行我们的 LLVM Pass，并为我们的 Compiler-RT 库设置正确的配置。更具体地说，下图显示了每个组件运行 LPCSan 需要完成的任务:

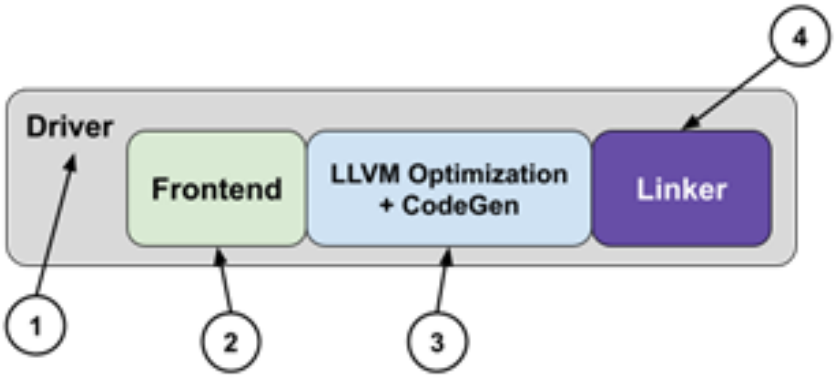

图 12.4 - 流水中每个组件的任务

以下是上图中每个数字的描述:

1. 驱动需要识别`-fsanitize=loop-counter` 标志。
2. 当前端要从**抽象语法树 (AST)** 生成 LLVM IR 时，需要正确配置 LLVM Pass 流水，以便它包括 `LoopCounterSanitizer` Pass。
3. LLVM Pass 流水需要运行我们的 `LoopCounterSanitizer`(如果前一个任务正确完成，不需要担心这个任务)。
4. 链接器需要将 Compiler-RT 库链接到目标程序。

尽管这个工作流看起来有点复杂，但不要被这些工作所吓倒——只要提供足够的信息，Clang 实际上可以完成大部分任务。在本节的其余部分中，我们将展示如何实现前面图中所示的任务，以将我们的 LPCSan 完全集成到 Clang 编译流水中 (以下教程在 `llvm-project/Clang` 文件夹中工作)。让我们开始吧！

1. 首先，必须修改 `include/clang/Basic/Sanitizers.def` 来添加杀毒器:

```
...
// Shadow Call Stack
SANITIZER("shadow-call-stack", ShadowCallStack)

// Loop Counter Sanitizer
SANITIZER("loop-counter", LoopCounter)
...
```

这将有效地为 `SanitizerKind` 类添加一个新的枚举值 `LoopCounter`。
结果是，驱动将解析`-fsanitize` 命令行选项，并根据我们在 `sanitizer.def` 中提供的信息，自动将循环计数器转换为 `SanitizerKind::LoopCounter`。

2. 接下来，让我们处理驱动部分。打开 `include/clang/Driver/SanitizerArgs.h`，并向 `SanitizerArgs` 类添加一个新的工具方法 `needsLpcsanRt`。代码如下:

```
bool needsLsanRt() const {...}
bool needsLpcsanRt() const {
  return Sanitizers.has(SanitizerKind::LoopCounter);
}
```

我们在这里创建的工具方法可以在驱动中的其他地方使用，以检查我们的杀毒器是否需要 Compiler-RT 组件。

3. 现在，让我们切换到 `lib/Driver/ToolChains/CommonArgs.cpp` 文件。这里，我们向 `collectSanitizerRuntimes` 函数添加了几行代码:

```
...
if (SanArgs.needsLsanRt() && SanArgs.linkRuntimes())
  StaticRuntimes.push_back("lsan");
if (SanArgs.needsLpcsanRt() && SanArgs.linkRuntimes())
  StaticRuntimes.push_back("lpcsan");
...
```

上面的代码有效地使链接器将正确的 Compiler-RT 库链接到目标二进制文件。

4. 我们将对驱动做的最后一个更改是在 `lib/Driver /ToolChains/Linux.cpp` 中。这里，我们在 `Linux::getSupportedSanitizers` 方法中添加以下代码:

```
SanitizerMask Res = ToolChain::getSupportedSanitizers();
...
Res |= SanitizerKind::LoopCounter;
...
```

前面的代码本质上是告诉驱动，在当前的工具链 (Linux 的工具链) 中支持 LPCSan。注意，为了简化示例，我们只支持 Linux 中的 LPCSan。如果想在其他平台和体系结构中支持这个自定义杀毒器，请修改其他工具链实现。如果需要，请参阅第 8 章了解更多细节。

5. 最后，我们将把 `LoopCounterSanitizer` Pass 插入到 LLVM Pass 流水中。打开 `lib/CodeGen/BackendUtil.cpp`，并添加以下代码到 `addSanitizers` 函数中:

```
...
// `PB` has the type of `PassBuilder`
PB.registerOptimizerLastEPCallback(
[&](ModulePassManager &MPM,
PassBuilder::OptimizationLevel Level) {
  ...
  if (LangOpts.Sanitize.has(SanitizerKind::LoopCounter))
  {
    auto FA
    =
    createFunctionToLoopPassAdaptor(LoopCounterSanitizer());
    MPM.addPass(
    createModuleToFunctionPassAdaptor(std::move(FA)));
  }
});
...
```

该文件的外围文件夹 `CodeGen` 是 Clang 和 LLVM 库的集合点。因此，在这里能看到几个 LLVM API。这个 `CodeGen` 组件主要有两个任务:

a. 将 Clang AST 转换为等效的 LLVM IR 模块

b. 构建一个 LLVM Pass 流水来优化 IR 和生成的机器码

上面的代码试图定制第二个任务——即定制 LLVM Pass 流水。我们在这里修改的特定功能 `addSanitizers`，其负责将杀毒器 Pass 到 Pass 流水中。为了更好地理解这段代码，让我们关注它两个组件:

i. `PassBuilder`: 这个类为每个优化级别提供了预定义的 Pass 流水配置——即我们熟悉的 O0 O3 符号 (以及用于体积优化的 Os 和 Oz)。除了这些预定义的布局之外，开发人员还可以利用**扩展点 (EP)** 自由地定制流水。

EP 是 (预定义的)Pass 流水中的一个特定位置，可以在这里插入新的 Pass。目前，`PassBuilder` 支持几种 EP，例如：Pass 的开始、Pass 的结束或向量化过程的结束等。在前面的代码中可以找到使用 EP 的例子，其中我们使用了 `PassBuilder::registeroptimizerlastcallback` 方法和一个 lambda 函数来定制位于 Pass 管道末端的 EP。lambda 函数有两个参数:`Mo dulePassManager`(表示 Pass 流水) 和当前优化级别。开发者可以使用 `ModulePassManager::addPass` 来插入任意的 LLVM Pass 到这个 EP 中。

ii. `ModulePassManager`: 这个类表示一个 Pass 流水——或者更具体地说，是 `Module` 的流水。当然，对于不同的 IR 单元，还有其他的 `PassManager` 类，比如：`Function` 的 `FunctionPassManager`。

前面的代码中，只要 `sanitierkind::LoopCounter` 是由用户指定的杀毒器，就尝试使用 `ModulePassManager` 实例来插入我们的 `LoopCounterSanitizer` Pass。因为 `LoopCounterSanitizer` 是一个循环 Pass，而不是一个模块 Pass，所以我们需要在传递和 PassManager 之间添加一些适配器。我们在这里使用的 `createFunctionToLoopPassAdaptor` 和 `createModuleToFunctionPassAdaptor` 函数创建了一个特殊的实例，该实例可以对不同的 IR 单元的 PassManager 进行适配。

这是 Clang 编译流水中支持 LPCSan 的所有程序逻辑。

6. 最后，我们必须对构建系统做一些小修改。打开 `runtime/CMakeLists.txt` 文件，修改以下 CMake 变量:

```
...
set(COMPILER_RT_RUNTIMES fuzzer asan builtins ··· lpcsan)
foreach(runtime ${COMPILER_RT_RUNTIMES})
...
```

我们对 `COMPILER_RT_RUNTIMES` 所做的更改，可以将我们的 LPCSan 的 Compiler-RT 库导入到构建中。

这些都是在 Clang 中支持 LPCSan 所必需的步骤。现在，我们终于可以使用 LPCSan 了:

```
$ clang -O1 -fsanitize=loop-counter input.c -o input
```

本节中，我们了解了如何创建杀毒器。杀毒器是一种有用的工具，可以在不修改原始程序代码的情况下捕获运行时行为。创建杀毒器的能力增加了编译器开发人员，为自己的用例定制定制诊断工具的灵活性。开发杀毒器需要全面了解 Clang、LLVM 和 Compiler-RT: 创建一个新的 LLVM

Pass，创建一个新的 Compiler-RT 组件，并在 Clang 中定制编译流水。读者们可以使用本节的内容，来强化您在本书前几章中所学到的知识。

在本章的最后一节中，我们将再看一种工具:PGO。

12.3. PGO

前一节中，我们了解了杀毒器如何使用仅在运行时可用的数据，帮助开发人员以更高的精度执行健全性检查。我们还了解了如何制作自定义杀毒器。在本节中，我们将继续讨论利用运行时数据的思想，将这些信息用于另一个方面——编译器优化。

PGO 是一种技术，它使用在运行时收集的统计信息，来支持更积极的编译器优化 (相关文件已经收集了足够的运行时数据)。为了了解这样的数据是如何增强优化的，假设我们有以下 C 代码:

```
void foo(int N) {
  if (N > 100)
    bar();
  else
    zoo();
}
```

这段代码中，我们有三个函数:foo、bar 和 zoo。第一个函数有条件地调用后两个函数。

当我们尝试优化这段代码时，优化器通常会尝试将函数内联扩展到调用函数中。这种情况下，bar 或 zoo 可能会内联到 foo 中。但是，如果 bar 或 zoo 有一个较大的函数体，内联两者可能会使最终二进制文件的体积膨胀。理想情况下，如果只能内联执行最频繁的那一个，那就太好了。遗憾的是，从统计的角度来看，我们不知道哪个函数的执行频率最高，因为 foo 函数根据一个 (非常量) 变量有条件地使用它们中的一个。

使用 PGO，我们可以在运行时收集 bar 和 zoo 的执行频率，并使用这些数据再次编译 (和优化) 相同的代码。下图展示了这一理念的高层概述:

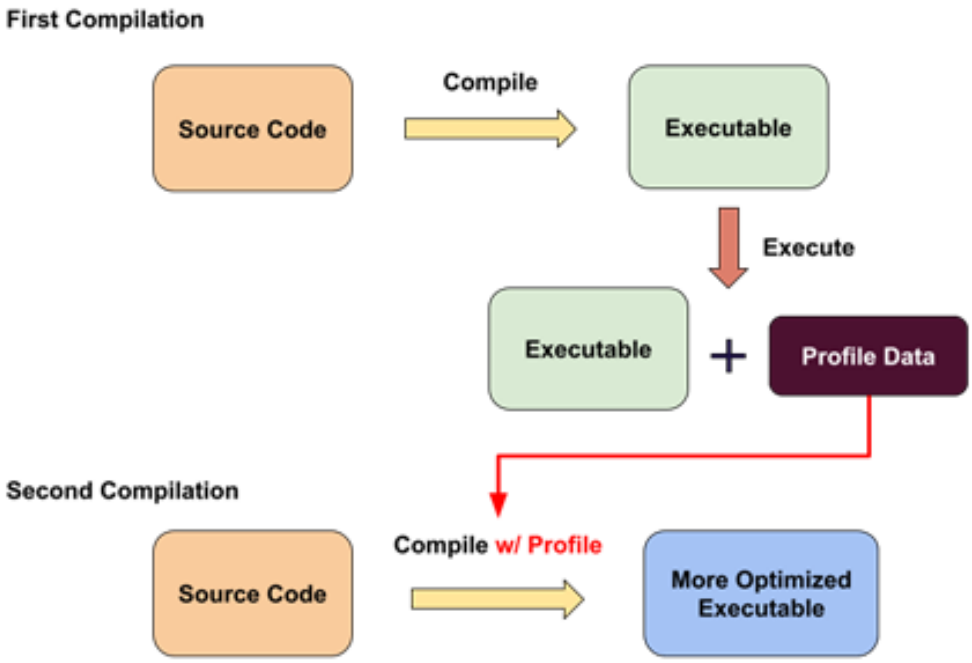

图 12.5 - PGO 工作流

这里，第一个编译阶段通常编译和优化代码。在执行编译后的程序 (任意次数) 之后，能够收集到足够的运行数据文件。在第二个编译阶段，我们不仅像以前那样优化了代码，而且还将收集到的数据集成到优化中，以使它们更积极地发挥作用。

PGO 主要有两种方法来收集运行时分析数据: 插入**检测**代码或利用**采样**数据。

12.3.1 基于检测的 PGO

基于检测的 PGO 在第一个编译阶段将检测代码插入到目标程序中。这段代码测量我们感兴趣的程序构造的执行频率——例如，基本块和函数——并将结果写入文件。这与杀毒器的工作原理类似。

基于检测的 PGO 通常生成精度更高的数据。这是因为编译器可以以一种为其他优化提供最大好处的方式插入检测代码。然而，就像杀毒器一样，基于检测的 PGO 改变了目标程序的执行流，不过这增加了性能回归的风险 (对于从第一个编译阶段生成的二进制文件)。

12.3.2 基于采样的 PGO

基于采样的 PGO 使用外部工具收集分析数据。开发人员使用 `perf` 或 `valgrind` 等分析器来诊断性能问题。这些工具通常利用高级的系统特性，甚至硬件特性来收集程序的运行时行为，例如: `perf` 可以让您了解分支预测和缓存缺失。

因为我们正在利用来自其他工具的数据，所以不需要修改原始代码来收集概要文件。因此，基于采样的 PGO 通常具有极低的运行时开销 (通常小于 1%)。此外，我们不需要为了分析目的而重新编译代码。然而，以这种方式生成的分析数据通常不那么精确。第二个编译阶段中，将分析数据映射回原始代码也更加困难。

本节的其余部分，我们将重点关注基于检测的 PGO。我们将学习如何利用它与 LLVM IR。然而，LLVM 中的这两种 PGO 策略共享许多公共的工具，因此代码是可移植的。下面是我们将要讨

论的内容:

- 使用分析数据
- 了解访问分析数据的 API

第一个主题将向我们展示如何使用 Clang 创建和使用基于检测的 PGO 数据文件，以及一些可以帮助我们检查和修改概要数据文件的工具。第二个主题将提供更多关于如何使用 LLVM API 访问分析数据的详细信息。如果想创建自己的 PGO Pass，这是很有用的。

使用分析数据

本节中，我们将了解如何使用生成、检查甚至修改基于工具的分析数据。让我们从下面的例子开始:

```
__attribute__((noinline))
void foo(int x) {
  if (get_random() > 5)
  printf("Hello %d\n", x * 3);
}
int main(int argc, char **argv) {
  for (int i = 0; i < argc + 10; ++i) {
    foo(i);
  }
  return 0;
}
```

上面的代码中，`get_random` 是一个函数，生成一个从 1 到 10 的均匀分布的随机数。换句话说，`foo` 函数中突 if 语句应该有 50% 的几率被执行。除了 `foo` 函数之外，`main` 中的 `for` 循环的迭代计数取决于命令行参数的数量。

现在，让我们尝试使用基于检测的 PGO 构建这段代码。以下是步骤:

1. 第一件事是为 PGO 分析生成一个可执行文件。命令如下:

   ```
   $ clang -O1 -fprofile-generate=pgo_prof.dir pgo.cpp -o pgo
   ```

   `-fprofile-generate` 选项支持基于检测的 PGO，我们在此标志之后添加的路径是存储分析数据的目录。

2. 接下来，必须使用三个命令行参数运行 `pgo`:

```
$ ./pgo `seq 1 3`
Hello 0
Hello 6
...
Hello 36
Hello 39
$
```

每个人可能会得到一个完全不同的输出，因为只有 50% 的机会打印字符串。

在这之后，pgo_prof.dir 文件夹应该包含 default_<hash>_<n>.profraw 文件，如下所示:

```
$ ls pgo_prof.dir
default_10799426541722168222_0.profraw
```

文件名中的哈希值是基于代码计算的哈希值。

3. 不能直接使用 *.profraw 文件用于第二个编译阶段。相反，必须使用 llvm-profdata 工具将其转换为另一种二进制形式。命令如下:

```
$ llvm-profdata merge pgo_prof.dir/ -o pgo_prof.profdata
```

llvm-profdata 是一个强大的工具，用于检查、转换和合并分析数据文件，我们将在后面更详细地讨论它。上面的命令中，我们正在合并和转换 pgo_prof.dir 下的所有数据文件，到单个 *.profdata 文件中。

4. 最后，我们可以使用合并的文件进行第二阶段的编译。命令如下:

```
$ clang -O1 -fprofile-use=pgo_prof.profdata pgo.cpp \
       -emit-llvm -S -o pgo.after.ll
```

这里，-fprofile-use 选项告诉 clang 使用存储在 pgo_prof.profdata 中的分析数据来优化代码。在完成这个优化之后，我们再来看看 LLVM IR 代码。

打开 pgo.after.ll，并找到到 foo 函数。下面是 foo 的简化版本:

```
define void @foo(i32 %x) !prof !71 {
entry:
  %call = call i32 @get_random()
  %cmp = icmp sgt i32 %call, 5
  br i1 %cmp, label %if.then, label %if.end, !prof !72

```

```
if.then:
  %mul = mul nsw i32 %x, 3
  ...
}
```

在之前的 LLVM IR 代码中，有两个地方与原来的 IR 不同，紧跟在函数头文件和分支指令之后的!prof 标记，它对应于我们前面看到的 if(get_random() > 5)。

在 LLVM IR 中，我们可以将**元数据**附加到不同的 IR 单元以提供补充信息。元数据将以一个以感叹号 ('!') 开头的标签出现在文本 LLVM IR 中。前面代码中的!prof、!71 和!72 是表示我们收集的分析数据的元数据标记。更具体地说，如果我们有与 IR 单元相关联的分析数据，总是以!prof 开头，然后是另一个元数据标记包含所需值。这些元数据值放在 IR 文件的最底部。如果在那里查找，会看到!71 和!72 的内容。代码如下:

```
!71 = !{!"function_entry_count", i64 110}
!72 = !{!"branch_weights", i32 57, i32 54}
```

这两个元数据是包含两个和三个元素的元组。!71，正如它的第一个元素所表示的，表示 foo 函数被调用的次数 (在本例中，调用了 110 次)。

!72 表示 if(get_random() > 5) 语句中每个分支执行的次数。本例中，真分支走了 57 次，假分支走了 54 次。之所以会得到这样的数字，是因为我们使用均匀分布生成随机数 (也就是说，每个分支有 50% 的概率)。

本节的第二部分中，我们将了解如何访问这些值，以便开发更积极的编译器优化。但在这样做之前，让我们更深入地了解我们刚刚收集的分析数据文件。

我们刚才使用的 llvm-profdata 工具，不仅可以帮助我们转换分析数据的格式，而且还可以为我们提供其内容的快速预览。下面的命令可以打印出 pgo_prof.profdata 的内容，包括从每个函数中收集的分析值:

```
$ llvm-profdata show --all-functions --counts pgo_prof.profdata
...
  foo:
    Hash: 0x0ae15a44542b0f02
    Counters: 2
    Block counts: [54, 57]
```

```
  main:
    Hash: 0x0209aa3e1d398548
    Counters: 2
    Block counts: [110, 1]
…
Instrumentation level: IR entry_first = 0
Functions shown: 9
Total functions: 9
Maximum function count: …
Maximum internal block count: …
```

这里，我们可以看到每个函数的分析数据条目。每个条目都有一个数字列表，表示所有封闭的基本块的执行频率。

或者，可以先将相同的分析数据文件转换为文本文件，再来检查它。命令如下:

```
$ llvm-profdata merge  --text pgo_prof.profdata -o pgo_prof.
proftext
$ cat pgo_prof.proftext
# IR level Instrumentation Flag
:ir
…
foo
# Func Hash:
784007059655560962
# Num Counters:
2
# Counter Values:
54
57
…
```

*.proftext 文件是一种可读的文本格式，所有的分析数据都简单地放在相应行中。

这种文本表示实际上可以转换回 *.profdata 格式使用类似的命令:

```
$ llvm-profdata merge  --binary pgo_prof.proftext -o pgo_prof.profdata
```

因此，*.proftext 对于手动编辑分析数据比较友好。

在深入研究 PGO 的 API 之前，我们还需要了解一个概念: 检测级别。

检测级别

到目前为止，我们已经了解到基于检测的 PGO 可以插入检测代码来收集运行时的分析数据，插入检测代码的位置，及其粒度也很重要。这个属性在基于工具的 PGO 中称为**检测级别**。LLVM 目前支持三种不同的检测级别。以下是对它们的描述:

- **IR**: 检测代码是基于 LLVM IR 插入的，例如：收集已获取分支数量的代码直接插入到分支指令之前，`-fprofile-generate` 命令行选项将用这个检测级别生成分析数据。假设我们有以下 C 代码:

```c
void foo(int x) {
  if (x > 10)
    puts("hello");
  else
    puts("world");
}
```

对应的 IR——不启用基于检测的 PGO——如下所示:

```
define void @foo(i32 %0) {
  ...
  %4 = icmp sgt i32 %3, 10
  br i1 %4, label %5, label %7

5:
  %6 = call i32 @puts(…"hello"…)
  br label %9
7:
  %8 = call i32 @puts(…"world"…)
  br label %9

9:
  ret void
}
```

这里有一个基本块的分支：%5 或%7。现在，让我们使用下面的命令来生成基于检测的 PGO 的 IR:

```
$ clang -fprofile-generate -emit-llvm -S input.c
```

这与我们在使用分析数据部分中，用于第一个 PGO 编译阶段的命令相同，只不过我们生成的是 LLVM IR，而不是可执行文件。得到的 IR 如下所示:

```
define void @foo(i32 %0) {
  ...
  %4 = icmp sgt i32 %3, 10
  br i1 %4, label %5, label %9
```

```
5:
  %6 = load i64, i64* ⋯ @__profc_foo.0, align 8
  %7 = add i64 %6, 1
  store i64 %7, i64* ⋯ @__profc_foo.0, align 8
  %8 = call i32 @puts(⋯"hello"⋯)
  br label %13
9:
  %10 = load i64, i64* ⋯ @__profc_foo.1, align 8
  %11 = add i64 %10, 1
  store i64 %11, i64* ⋯ @__profc_foo.1, align 8
  %12 = call i32 @puts(⋯"world"⋯)
  br label %13

13:
  ret void
}
```

上面的 IR 中，两个分支中的基本块都有新的代码。更具体地说，两者都增加全局变量的值——`@__profc_foo.0` 或 `@__profc_foo.1`。这两个变量中的值最终将导出为分支的分析数据，从而表示每个分支的使用次数。

这个检测级别提供了相当好的精度，但会受到编译器更改的影响。更具体地说，如果 Clang 改变了它生成 LLVM IR 的方式，那么插入检测代码的位置也会不同。这意味着对于相同的输入代码，使用旧版本的 LLVM 生成的分析数据可能与使用新版本的 LLVM 生成的分析数据不兼容。

- **AST**: 检测代码是基于 AST 插入的，例如：收集已获取分支数量的代码可以作为 `IfStmt`(一个 AST 节点) 中的一个新的 textttStmt AST 节点插入。使用此方法，检测代码几乎不受编译器更改的影响，我们可以在不同编译器版本之间拥有更稳定的分析数据格式。这个检测级的缺点是它的精度低于 IR 检测级。

  第一次编译使用 `clang` 时，只需使用`-fprofile-instrgenerate` 命令行选项代替`-fprofile-generate`，就可以采用这种检测级别。但是，在第二次编译时不需要更改命令。

- **上下文敏感**: 使用分析数据部分中，我们了解到可以收集关于分支执行次数的信息。然而，在 IR 检测级别上，几乎不可能知道哪个是使用最多的分支。换句话说，传统的 IR 检测级别失去了调用的上下文。上下文敏感的检测级别，试图通过在函数内联后，再次收集概要文件来解决这个问题，从而获取更高精度的数据。

  然而，在 Clang 中使用这个特性有点麻烦——我们需要编译相同的代码三次，而不是两次。以下是使用上下文敏感的 PGO 步骤:

```
$ clang -fprofile-generate=first_prof foo.c -o foo_exe
$ ./foo_exe ⋯
$ llvm-profdata merge first_prof -o first_prof.profdata
```

首先，使用我们在“使用分析数据”一节中学到的内容，生成一些普通的 (IR 检测级) 分析数

据:

```
$ clang -fprofile-use=first_prof.profdata \
       -fcs-profile-generate=second_prof foo.c -o foo_exe2
$ ./foo_exe2
$ llvm-profdata merge first_prof.profdata second_prof \
         -o combined_prof.profdata
```

然后，使用两个 PGO 命令行选项运行 `clang`，`-fprofile-use` 和`-fc-profile-generate`，分别带有上一步的配置文件路径和预期的输出路径。当使用 `llvm-profdata` 来做后处理时，我们合并了所有的数据文件:

```
$ clang -fprofile-use=combined_prof.profdata \
       foo.c -o optimized_foo
```

最后，将组合的分析文件输入到 Clang 中，这样 Clang 就可以使用上下文敏感的分析数据来更准确地描述程序的运行时行为。

注意，不同的检测级别只会影响数据的准确性，它们不会影响我们如何检索这些数据，这将在下一节中进行讨论。

本节的最后一部分，我们将了解如何通过 LLVM 提供的 API 访问 LLVM Pass 中的分析数据。

了解访问分析数据的 API

前一节中，我们学习了如何使用 Clang 运行基于检测的 PGO，并使用 `llvm-profdata` 查看分析数据文件。本节中，我们将了解如何在 LLVM Pass 中访问数据，以助于开发自己的 PGO。

深入开发细节之前，让我们了解一下如何将这些分析数据文件使用到 `opt` 中，因为使用它更容易测试单个 LLVM Pass。下面是一个示例命令:

```
$ opt -pgo-test-profile-file=pgo_prof.profdata \
    --passes="pgo-instr-use,my-pass…" pgo.ll …
```

上面的命令中有两个关键:

- 使用`-pgo-test-profile-file` 指定要写入的分析数据文件。
- ”pgo-in-use” 字符串表示 `PGOInstrumentaitonUse` Pass，它读取 (基于检测的) 数据文件，并在 LLVM IR 上注释数据。但是，即使在预定义的优化级别 (即 O0 O3、Os 和 Oz) 中，也不会默认运行。如果 Pass 不在 pass 流水的前部，我们就无法访问任何分析数据。因此，我们需要显式地将其添加到优化流水中。前面的示例命令演示了如何在管道中运行自定义

LLVM Pass(my-pass) 之前运行。如果想在任何预定义的优化流水 (例如 O1) 之前运行，必须指定--passes="pgo-inst -use,default<O1>" 命令行选项。

现在，在将分析数据读入 opt 之后会发生什么？由第二个编译阶段 (pgo.after.ll) 生成的 LLVM IR 文件，会为我们提供了这个问题的答案。

在 pgo.after.ll 中，我们看到一些分支使用元数据装饰，这些元数据获取它们的次数。类似的元数据出现在函数中，表示这些函数调用的总次数。

更一般地说，LLVM 通过元数据直接将数据 (从文件中读取) 与相关的 IR 结构结合在一起。这种策略的最大优点是，不需要在整个优化过程中携带原始分析数据——IR 本身包含了这个分析信息。

现在的问题是，我们如何访问附加到 IR 的元数据？LLVM 的元数据可以附加到多种 IR 单元上。让我们先来看看最常见的一个: 访问附加到指令的元数据。下面的代码向我们展示了如何读取分析元数据——!prof !71，它附加到一个分支指令上:

```
// `BB` has the type of `BasicBlock&`
Instruction *BranchInst = BB.getTerminator();
MDNode *BrWeightMD = BranchInst->getMetadata(LLVMContext::MD_
prof);
```

上面的代码段中,我们使用 BasicBlock::getTerminator 来获取基本块中的最后一条指令,这在大多数情况下是一条分支指令。然后,我们尝试使用 MD_prof 元数据检索分析元数据。BrWeightMD 是我们正在寻找的结果。

BrWeightMD 的类型 MDNode 表示单个元数据节点。不同的 MDNode 实例可以组合在一起。一个 MDNode 实例可以使用其他 MDNode 实例作为它的操作数——类似于在第 10 章中的实例。复合 MDNode 可以表达更复杂的概念。

在本例中，BrWeightMD 中的每个操作数表示获取每个分支的次数。下面是访问代码:

```
if (BrWeightMD->getNumOperands() > 2) {
  // Taken counts for true branch
  MDNode *TrueBranchMD = BrWeightMD->getOperand(1);
  // Taken counts for false branch
  MDNode *FalseBranchMD = BrWeightMD->getOperand(2);
}
```

计数也表示为 MDNode。

两个分支的操作数索引

注意，两个分支的数据都从索引 1 开始的操作数上，而不是索引 0。

如果希望将这些分支 MDNode 实例转换为常量，可以利用 mdconst 名称空间提供的一个小工具。下面是一个例子:

```
if (BrWeightMD->getNumOperands() > 2) {
  // Taken counts for true branch
  MDNode *TrueBranchMD = BrWeightMD->getOperand(1);
  ConstantInt *NumTrueBrTaken
    = mdconst::dyn_extract<ConstantInt>(TrueBranchMD);
```

```
  ...
}
```

前面的代码解包装了一个 MDNode 实例并提取了底层的 ConstantInt 实例。

对于 Function，可以用更简单的方法得到调用的次数:

```
// `F` has the type of `Function&`
Function::ProfileCount EntryCount = F.getEntryCount();
uint64_t EntryCountVal = EntryCount.getCount();
```

Function 使用一种略有不同的方式来表示其调用的频率。但是检索数值分析值仍然非常容易，如上的代码所示。

值得注意的是，虽然这里只关注基于检测的 PGO，但对于基于采样的 PGO, LLVM 也使用相同的编程接口来公开其数据。换句话说，即使正在使用从使用不同的 opt 命令的采样工具收集的分析数据，分析数据也将标注在 IR 单元上，仍然可以使用上述方法访问。实际上，在本节的其余部分中，我们将介绍的工具和 API，大多与分析数据源无关。

目前为止，我们一直在处理从分析数据中检索到的实际值。然而，在开发编译器优化或程序分析算法方面，这些低层值不能帮助我们走得太远——通常，我们更感兴趣的是高级概念，如“最频繁执行的函数”或“最少执行的分支”。为了满足这些需求，LLVM 在分析数据的基础上构建了几个分析，以提供这种高级的结构化信息。

下一节中，我们将介绍其中的一些分析，以及在 LLVM Pass 中的用法。

12.3.3 使用性能数据进行分析

在本节中，我们将学习三个分析类，它们可以帮助我们推断基本块和函数在运行时的执行频率。具体如下:

- BranchProbabilityInfo
- BlockFrequencyInfo
- ProfileSummaryInfo

这个列表是按照它们在 IR 中的分析范围排序的——从本地到全局。让我们从前两个开始。

使用 BranchProbabilityInfo 和 BlockFrequencyInfo

在前一节中，我们了解了如何访问附加到每个分支指令的分析元数据——**分支权重**。LLVM 中的分析框架通过 BranchProbabilityInfo 类提供了一种更简单的方法来访问这些值。下面是一些示例代码，展示了如何在 (函数)Pass 中使用它:

```
#include "llvm/Analysis/BranchProbabilityInfo.h"
PreservedAnalyses run(Function &F, FunctionAnalysisManager
&FAM) {
  BranchProbabilityInfo &BPI
    = FAM.getResult<BranchProbabilityAnalysis>(F);
  BasicBlock *Entry = F.getEntryBlock();
  BranchProbability BP = BPI.getEdgeProbability(Entry, 0);
  ...
}
```

前面的代码检索了 `BranchProbabilityInfo` 的实例，是 `BranchProbabilityAnalysis` 的结果，并试图从入口块获取到其第一个后续块的权重。

返回值是一个 `BranchProbability` 实例，它以百分比的形式给出了分支的概率。可以使用 `BranchProbability::getNumerator` 检索值（“分母”默认为 100）。`BranchProbability` 类还提供了一些方便的实用方法，用于在两个分支概率之间执行算术，或按特定因子缩放概率。尽管使用 `BranchProbabilityInfo` 可以很容易地知道哪个分支更有可能走到，但是没有额外的数据，我们不能在整个函数中告诉执行分支的概率，例如：假设我们有以下 CFG:

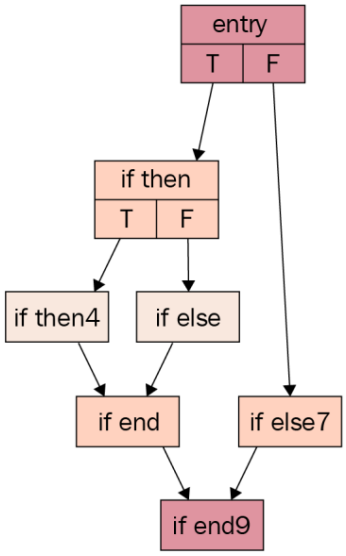

图 12.6 - CFG 嵌套分支

对于上图，我们有以下基本块的分析计数器值:

- **`if.then4`**: 2
- **`if.else`**: 10
- **`if.else7`**: 20

如果我们只看分支权重元数据块 `if.then4` 和 `if.else`——即 `if.then` 的真和假分支。然后，我们可能会产生一种错觉，`if.else` 块有 83% 的几率。但事实是，只有 31% 的概率，因为控制流进入在进入 `if.then` 之前，`if.else7` 的概率更高。在这种情况下，我们可以通过计算来得出正确的答案，但当 CFG 变得更大更复杂时，我们就很难自己做判断了。

`BlockFrequencyInfo` 类提供了解决这个问题的捷径。它可以告诉我们每个基本块在其封闭函数的上下文中执行的频率。下面是它在 Pass 中的用法:

```
#include "llvm/Analysis/BlockFrequencyInfo.h"
PreservedAnalyses run(Function &F, FunctionAnalysisManager
&FAM) {
  BlockFrequencyInfo &BFI
    = FAM.getResult<BlockFrequencyAnalysis>(F);
  for (BasicBlock *BB : F) {
    BlockFrequency BF = BFI.getBlockFreq(BB);
```

```
  }
  ...
}
```

前面的代码检索了 BlockFrequencyInfo 实例，是 BlockFrequencyAnalysis 的结果，并尝试计算函数中每个基本块的块频率。

类似于 BranchProbability 类，BlockFrequency 也提供了方法来计算其他 BlockFrequency 实例。但与 BranchProbability 不同，从 BlockFrequency 检索到的数值不是以百分比表示的。BlockFrequency::getFrequency 返回一个整数，是相对于当前函数的入口块的频率。要获得基于百分比的频率，可以使用下面的代码:

```
// `BB` has the type of `BasicBlock*`
// `Entry` has the type of `BasicBlock*` and represents entry
// block
BlockFrequency BBFreq = BFI.getBlockFreq(BB),
               EntryFreq = BFI.getBlockFreq(Entry);
auto FreqInPercent
  = (BBFreq.getFrequency() / EntryFreq.getFrequency()) * 100;
```

FreqInPercent 表示 BB 的阻塞频率，以百分比表示。

BlockFrequencyInfo 在函数的上下文中计算特定基本块的频率——但是整个模块呢？如果在这种情况下引入调用图，是否可以将我们刚刚学过的块频率与调用函数的频率相乘，以得到全局范围内的执行频率？幸运的是，LLVM 已经准备了一个有用的类来解决这个问题——ProfileSummaryInfo。

使用 ProfileSummaryInfo

ProfileSummaryInfo 类提供了一个模块中所有分析数据的全局视图。下面是一个在模块中检索实例的例子:

```
#include "llvm/Analysis/ProfileSummaryInfo.h"
PreservedAnalyses run(Module &M, ModuleAnalysisManager &MAM) {
  ProfileSummaryInfo &PSI = MAM.
  getResult<ProfileSummaryAnalysis>(M);
  ...
}
```

ProfileSummaryInfo 提供了各种各样的功能。让我们来看看它最有意思的三个方法:

- isFunctionEntryCold/Hot(Function*): 这两种方法将 Function 的输入计数 (有效地反映了函数被调用的次数) 与同一模块中其他函数的输入计数进行比较，并告诉我们查询函数在这个指标中的排名是高还是低。
- isHot/ColdBlock(BasicBlock*, BlockFrequencyInfo&): 这两个方法的工作方式与前面的项目一相似，但比较的是 BasicBlock 与模块中所有其他块的执行频率.
- isFunctionCold/HotInCallGraph(Function*, BlockFrequencyInfo&): 这两个方法结合了前面两个要点的方法，可以根据函数的入口计数，或其封闭基本块的执行频率来判断函数是“热”的，还是“冷”的。当一个函数有一个很低的输入计数 (也就是说，它不经常

被调用)，但是包含一个具有极高的基本块执行频率的循环时，这是很有用的。在本例中，`isFunctionHotInCallGraph` 方法可以给我们一个更准确的评估。

这些 API 还有变体，可以自定义“热”与“冷”的分界点。有关更多信息，请参阅 API 文档。

很长一段时间，编译器只能用静态视图分析和优化源代码。对于程序内部的动态因素——例如，有计数的分支——编译器只能做出一个近似值。PGO 打开了格局，为编译器提供额外的信息，以查看目标程序的运行时行为，以便做出更明确和更积极的决策。本节中，我们了解了如何使用 LLVM 收集和使用运行时概要信息 (PGO 的关键)。我们了解了如何在 LLVM 中使用相关的工具来收集和生成这样的概要数据。我们还了解了我们用来访问数据的编程接口——以及在此基础上构建的一些高级分析——以帮助我们在 LLVM Pass 中进行开发。有了这些功能，LLVM 开发人员就可以插入这些运行时信息，进一步提高现有优化过程的质量和精度。

12.4. 总结

本章中，我们通过处理静态源代码和捕获程序的运行时行为来扩充编译器的工作空间。本章的第一部分中，我们了解了如何使用 LLVM 提供的工具来创建一个杀毒器——一种将检测代码插入到目标程序中以检查特定运行时属性的技术。通过使用杀毒器，软件工程师可以轻松和高精度地提高他们的开发质量。本章的第二部分，我们将这些运行时数据的使用扩展到编译器优化的领域——PGO 是一种使用动态信息 (如基本块或函数的执行频率) 来做出更积极的优化代码的决策的技术。最后，我们了解了如何使用 LLVM Pass 访问这些数据，这使我们能够使用 PGO 强化现有的优化策略。

恭喜你，读完了最后一章! 非常感谢您阅读这本书。在计算机科学中，编译器开发从来都不是一门简单的学科——如果不是晦涩难懂的话。在过去的十年里，LLVM 提供了健壮而灵活的模块，从根本上改变了人们对编译器的看法，从而大大降低了这一课题的难度。编译器不再只是像 `gcc` 或 `clang` 那样的单个可执行文件——它是一组构建块的集合，为开发人员提供了无数种方法创建工具，来处理编程语言领域的难题。

然而，选择如此之多，当我还是 LLVM 新手的时候，我经常会迷失和困惑。这个项目中的每个 API 都有文档，但我不知道如何将它们组合在一起。我希望有一本书能指出 LLVM 中每个重要组件的大致方向，告诉我它是什么，以及我如何利用它。这就是我在 LLVM 职业生涯开始时希望得到的那本书——你刚刚读完的那本书。我希望你读完这本书后不要停止你的 LLVM 之旅，了进一步提高你的技能，并加强你从这本书中学到的东西，我建议你查看官方文档页面 (https://llvm.org/docs)，以获得补充这本书的内容。更重要的是，我鼓励你通过邮件列表 (https://lists.llvm.org/cgi-bin/mailman/listinfo/llvmdev) 或话语论坛 (https://llvm.discourse.group/) 参与 LLVM 社区，特别是第一个论坛——尽管邮件列表可能听起来很老派，那里有很多有才华的人愿意回答你的问题，并提供有用的学习资源。最后，每年的 LLVM 开发会议 (https://llvm.org/devmtg/)，在美国和欧洲 (有着非常最好的活动)，在这里你可以学习新的 LLVM 技能，并与真正维护 LLVM 的人面对面聊天。

我希望这本书在掌握 LLVM 的道路上对你有所启发，并帮助你找到制作编译器的乐趣。

练习答案

本部分包含了所有章节问题的答案。

第 6 章

1. 大多数情况下，令牌是从提供的源代码中获取的，但在某些情况下，令牌可以在 Preprocessor 中动态生成，例如：__LINE__内置宏扩展为当前行号，而__DATE__宏扩展为当前日历日期。Clang 如何将生成的文本内容放入 SourceManager 的源代码缓冲区？Clang 如何分配 SourceLocation 给这些令牌？

   - 开发人员可以利用 clang::ScratchBuffer 类来插入动态 Token 实例。

2. 当我们讨论实现一个自定义的 PragmaHandler 时，我们使用 Preprocessor::Lex 来获取 pragma 名称后面的令牌，直到我们遇到 eod 令牌类型。我们能继续在 eod 标记之外进行词法分析吗？如果可以在 #pragma 指令后使用任意标记，你会做什么有趣的事情？

   - 是的，我们可以继续在 eod 令牌之外进行词法分析，它只使用 #pragma 行后面的内容。通过这种方式，你可以创建一个自定义的 #pragma，它允许你编写任意内容 (在它下面) ——例如，编写 Clang 不支持的编程语言。下面是一个例子:

     ```
     #pragma javascript function
     const my_func = (arg) => {
       console.log(`Hello ${arg}`);
     };
     ```

     前面的代码片段展示了如何创建一个自定义的 #pragma，允许您在它下面定义一个 JavaScript 函数。

3. 在宏保护项目的开发自定义预处理器插件和回调部分，警告消息的格式为 [WARNING] In <source location>: ⋯。显然，这不是我们从 Clang 中看到的典型的编译器警告，它看起来像 <source location="">: warning:⋯:

   ```
   ./simple_warn.c:2:7: warning: unused variable 'y'⋯
   int y = x + 1;
       ^
   1 warning generated.
   ```

   在受支持的终端中，警告字符串甚至是彩色的。我们如何打印这样的警告消息呢？在 Clang 有这样的基础设施吗？

   - 开发人员可以使用 Clang 中的诊断框架来打印这样的消息。在第 7 章“处理 AST”的“打印诊断消息”一节中，其中我们将向您展示这个框架的用法。

第 8 章

1. 重写汇编和链接阶段是很常见的，因为不同的平台倾向于支持不同的汇编器和链接器。但是是否可以覆盖编译阶段 (即 Clang)？如果有可能，我们该怎么做？人们这样做的原因可能是什么？

   - 可以覆盖 `ToolChain::SelectTool` 方法，并根据参数提供一个替代的 `Tool` 实例 (表示编译阶段)。下面是一个例子:

```
Tool*
MyToolChain::SelectTool(const JobAction &JA) const
override {
  if (JA.getKind() == Action::CompileJobClass &&
      getTriple().getArch() == CUSTOM_HARDWARE)
    return new MyCompiler(…);
  …
  // Run the default `SelectTool` otherwise
  return ToolChain::SelectTool(JA);
}
```

     在上的代码片段中，我们提供了自己的编译器实例——`MyCompiler`——它是一个从 `Tool` 派生出来的类。如果我们要为特定的硬件架构编译代码的话，可以这样做。
     当目标平台 (例如，前面代码中的 `CUSTOM_HARDWARE`) 或输入文件 Clang 不支持时，提供一个替代编译器实例是有用的，但你仍然想为所有的构建工作使用相同的 Clang 命令行接口，例如：假设正在尝试将相同的项目交叉编译到多个不同的体系结构，但是 Clang 还不支持其中的一些。因此，当 Clang 命令行工具要求为那些体系结构构建项目时，您可以创建一个自定义的 Clang 工具链，并将编译工作外包给其他 (重定向到外部) 编译器 (例如，`gcc`)。

2. 当我们使用 `tools::zipline::Linker::ConstructJob` 时，如果用户通过`-fuse-ld` 标志提供了不支持的压缩器名称，我们只需使用 `llvm_unreachable` 来清理编译过程。我们是否可以用在第 7 章“处理 AST”中的**诊断**框架来替换它，以打印出更好的消息？

   - `Driver` 类提供了访问诊断框架的快捷方式。在 `Tool` 的派生类中，你可以使用 `get ToolChain().getdriver()` 来获取一个 `Driver` 实例，然后使用 `Driver::Diag` 方法打印出诊断消息。

3. 就像可以使用`-Xclang` 将标志直接传递给前端一样，我们也可以使用驱动标志 (例如`-Wa`(用于汇编器) 和`-Wl`(用于链接器)) 将特定于汇编器或链接器的标志直接传递给汇编器或链接器。我们如何在 Zipline 的自定义汇编器和链接器阶段使用这些标志？

   - 在 `ConstructJob` 方法中，可以读取 `options::OPT_Wa_COMMA` 和 `options::OPT _Wl_COMMA` 的值来分别检索特定于汇编器和链接器的命令行标志。下面是一个例子:

```
void
MyAssembler::ConstructJob(Compilation &C,
              const JobAction &JA,
              const InputInfo &Output,
```

```
                const InputInfoList &Inputs,
                const ArgList &Args,
                const char *LinkingOutput)
                const {
  if (Arg *A = Args.getLastArg(options::OPT_Wl_COMMA)) {
    // `A` contains linker-specific flags
    ...
  }
  ...
}
```

第 9 章

1. 在为新 PassManager 部分编写 LLVM Pass 的 `StrictOpt` 示例中，我们如何在不派生 `PassInfoMixin` 类的情况下编写 Pass？

   - `PassInfoMixin` 类只为您定义了一个实用函数 `name`，该函数返回此 Pass 的名称。因此，可以很容易地自己创建一个。下面是一个例子:

   ```
   struct MyPass {
       static StringRef name() { return "MyPass"; }
     PreservedAnalyses run(Function&, FunctionAnalysisManager&);
   };
   ```

2. 我们如何为新 PassManager 开发自定义工具？在不修改 LLVM 源树的情况下，我们如何做到这一点？(提示: 使用我们在本章中学习的 Pass 插件。)

   - Pass 检测是在 LLVM 传递之前和/或之后运行的一段代码。这篇博客文章演示了一个通过 Pass 插件开发自定义 Pass 工具的例子:https://medium.com/@mshockwave/writing-passinstrument-for-llvm-newpm-f17c57d3369f。