
Getting Started with LLVM Core Libraries

潘立丰

2023 年 10 月 06 日

1	第 1 章编译和安装 LLVM	3
1.1	理解 LLVM 的版本	4
1.2	获取预编译安装包	4
1.2.1	获取官方预编译二进制文件	4
1.2.2	使用包管理器	5
1.3	从源代码编译	6
1.3.1	系统要求	7
1.3.2	获取源代码	7
1.3.3	编译和安装 LLVM	8
1.3.4	Windows 和 Microsoft Visual Studio	13
1.3.5	Mac OS X 和 Xcode	15
1.4	总结	20
2	第 2 章外部项目	21
2.1	介绍 Clang extras	22
2.1.1	编译和安装 Clang 附加 (extra) 工具	22
2.1.2	理解 Compiler-RT	23
2.1.3	试验 Compiler-RT	24
2.2	使用 DragonEgg 插件	25
2.2.1	编译 DragonEgg	25
2.2.2	理解 DragonEgg 和 LLVM 工具的编译流水线	26
2.2.3	理解 LLVM 测试套件	27
2.2.4	使用 LLDB	28
2.2.5	介绍 libc++ 标准库	30
2.3	总结	32
3	第 3 章工具和设计	33

3.1	LLVM 的基本设计原则和它的历史	33
3.2	理解如今的 LLVM	35
3.3	跟编译器驱动器交互	36
3.4	使用独立工具	37
3.5	探究 LLVM 内部设计	40
3.5.1	了解 LLVM 基础库	40
3.5.2	演示插件式 Pass 接口	44
3.6	编写第一个 LLVM 项目	45
3.6.1	编写 Makefile	45
3.6.2	编写代码	47
3.7	浏览 LLVM 源代码——一般建议	48
3.7.1	理解代码如文档	48
3.7.2	向社区求助	49
3.7.3	处理更新——SVN 日志用作文档	49
3.7.4	结束语	51
3.8	总结	51
4	第 4 章前端	53
4.1	介绍 Clang	53
4.1.1	前端的活动	54
4.1.2	程序库	55
4.1.3	理解 Clang 诊断	57
4.2	通过 Clang 学习前端的步骤	61
4.2.1	词法分析	61
4.2.2	语法分析	67
4.2.3	语义分析	73
4.2.4	生成 LLVM IR 代码	75
4.3	组合在一起	76
4.4	总结	79
5	第 5 章 LLVM 中间表示	81
5.1	概述	81
5.1.1	理解 LLVM IR 的目标依赖	82
5.2	练习基础工具转换 IR 格式	83
5.3	介绍 LLVM IR 语言的语法	84
5.3.1	介绍 LLVM IR 内存中的模型	87
5.4	编写一个定制的 LLVM IR 生成器	88
5.4.1	编译并运行 IR 生成器	91
5.4.2	学习如何用 C++ 后端生成任意 IR 的构造代码	92
5.5	IR 层次的优化	92
5.5.1	编译时和链接时优化	92
5.5.2	发现哪些 Pass 有用	94

5.5.3	理解 Pass 依赖关系	95
5.5.4	理解 Pass API	96
5.5.5	写一个定制的 Pass	97
5.6	总结	101
6	第 6 章后端	103
6.1	概述	104
6.1.1	使用后端工具	105
6.2	学习后端代码结构	105
6.3	了解后端程序库	106
6.4	学习 LLVM 后端如何利用 TableGen	107
6.4.1	语言	108
6.4.2	了解代码生成器.td 文件	109
6.5	理解指令选择过程	113
6.5.1	SelectionDAG 类	113
6.5.2	低层化	115
6.5.3	DAG 结合与合法化	116
6.5.4	DAG 到 DAG 的指令选择	117
6.5.5	可视化指令选择过程	120
6.5.6	快速指令选择	121
6.6	调度	121
6.6.1	指令延迟表	121
6.6.2	风险检测	122
6.6.3	调度单元	122
6.7	机器指令	123
6.8	寄存器分配	123
6.8.1	寄存器合并器	124
6.8.2	虚拟寄存器重写	127
6.8.3	目标钩子	128
6.9	序曲和尾声	129
6.9.1	帧索引	129
6.10	理解机器代码框架	129
6.10.1	MC 指令	130
6.10.2	代码输出	130
6.11	编写你自己的机器 Pass	132
6.12	总结	134
7	第 7 章 Just-in-Time 编译器	137
7.1	了解 LLVM JIT 引擎基础	138
7.1.1	介绍执行引擎	138
7.1.2	内存管理	139
7.2	介绍 llvm::JIT 基础结构	140

7.2.1	数据块写到内存	140
7.2.2	使用 <code>JITMemoryManager</code>	140
7.2.3	目标代码输出	141
7.2.4	目标信息	142
7.2.5	学习如何使用 <code>JIT</code> 类	143
7.3	介绍 <code>llvm::MCJIT</code> 框架	147
7.3.1	<code>MCJIT</code> 引擎	147
7.3.2	理解 <code>MCJIT</code> 如何编译模块	148
7.3.3	使用 <code>MCJIT</code> 引擎	151
7.4	使用 <code>LLVM</code> <code>JIT</code> 编译工具	153
7.4.1	使用 <code>lli</code> 工具	153
7.4.2	使用 <code>llvm-rtldyld</code> 工具	154
7.5	其它的资源	155
7.6	总结	156
8	第 8 章交叉平台编译	157
8.1	比较 <code>GCC</code> 和 <code>LLVM</code>	158
8.2	理解目标三元组	159
8.3	准备工具链	160
8.3.1	标准 <code>C</code> 和 <code>C++</code> 库	161
8.3.2	运行时库	161
8.3.3	汇编器和链接器	161
8.3.4	<code>Clang</code> 前端	162
8.4	<code>Clang</code> 命令行参数交叉编译	162
8.4.1	驱动器的目标选项	163
8.4.2	依赖	163
8.4.3	交叉编译	164
8.4.4	修改系统根目录	165
8.5	生成一个 <code>Clang</code> 交叉编译器	166
8.5.1	配置选项	166
8.5.2	编译和安装你的基于 <code>Clang</code> 的交叉编译器	167
8.5.3	别的编译方法	168
8.6	测试	169
8.6.1	开发板	169
8.6.2	仿真器	169
8.7	额外的资源	170
8.8	总结	170
9	第 9 章 <code>Clang</code> 静态分析器	171
9.1	理解静态分析器的角色	172
9.1.1	对比经典的警告和 <code>Clang</code> 静态分析器	172
9.1.2	符号化执行引擎的力量	175

9.2	测试静态分析器	177
9.2.1	使用驱动器和编译器	177
9.2.2	了解可用的检查器	177
9.2.3	在 Xcode IDE 中使用静态分析器	179
9.2.4	在 HTML 中生成图形化报告	180
9.2.5	处理大型项目	181
9.3	用你自己的检查器扩展静态分析器	185
9.3.1	熟悉项目的架构	185
9.3.2	编写你自己的检查器	186
9.4	更多资源	195
9.5	总结	195
10	第 10 章 Clang 工具和 LibTooling	197
10.1	生成编译命令 database	197
10.2	clang-tidy 工具	198
10.2.1	利用 clang-tidy 检查你的代码	199
10.3	重构工具	200
10.3.1	Clang Modernizer	200
10.3.2	Clang Apply Replacements	201
10.3.3	ClangFormat	202
10.3.4	Modularize	205
10.3.5	PPTrace	210
10.3.6	Clang Query	212
10.3.7	Clang Check	213
10.3.8	去除 c_str() 调用	213
10.4	编写你自己的工具	214
10.4.1	问题定义-编写一个 C++ 代码重构工具	214
10.4.2	配置你的源代码的位置	214
10.4.3	剖析工具样板代码	215
10.4.4	使用 AST 匹配器	218
10.4.5	编写回调函数	222
10.4.6	测试你的新重构工具	223
10.5	更多资源	224
10.6	总结	224
11	Indices and tables	225

Bruno Cardoso Lopes, Rafael Auler 著

潘立丰译

第 1 章编译和安装 LLVM

LLVM 基础设施适用于若干 **Unix** 系统（**GNU/Linux**，**FreeBSD**，**Mac OS**）和 **Windows** 系统。在本章中，我们一步一步地说明如何让 LLVM 在这些系统上工作。某些系统可获得 LLVM 和 Clang 的预编译安装包，但是也可以从源代码编译得到它们。

LLVM 的新手使用者必须明白，基本的 LLVM 编译器包括 LLVM 和 Clang 的库和工具。因此，本章的目的在于说明如何编译（build）并安装（install）它们。本书自始至终都将聚焦 LLVM 版本 3.4。然而，LLVM 是一个年轻的项目，处于活跃的开发期，因此它在不断地改变；知道这一点是重要的。

备注：写作本书的时候，LLVM 3.5 还没有发布。尽管这本书讲解 LLVM 3.4，但是我们计划在 2014 年 9 月的第三个星期发布一个附录，将本书中的例子更新到 LLVM 3.5，让你能够以 LLVM 最新的版本练习本书的内容。这个附录将可以从 https://www.packtpub.com/sites/default/files/downloads/6924OS_Appendix.pdf 得到。

本章将讨论以下内容：

- 理解 LLVM 的版本
- 利用预编译二进制文件安装 LLVM
- 利用包管理器安装 LLVM
- 在 Linux 上从源代码编译 LLVM
- 在 Windows 上利用 Visual Studio 从源代码编译 LLVM
- 在 Mac OS X 上利用 Xcode 从源代码编译 LLVM

1.1 理解 LLVM 的版本

由于众多程序员的贡献，LLVM 项目的更新节奏很快。截止版本 3.4，它的 SVN（subversion，版本控制系统）仓库记录了超过 200,000 次提交，而它的第一次发布发生在十多年以前。仅在 2013 年，项目有接近 30,000 次新的提交。因此，新的特性不断地被提出，其它特性很快变得过时。与任何大型项目一样，开发者需要遵守严格的规范，当项目状态良好并通过多种测试时，会发布稳定的程序，让使用者既能够使用经过仔细测试的版本，又能够体验最新的特性。

从历史上看，LLVM 项目总是每年发布两个稳定的版本。版本发布时增加小版本数字，每次增加 1。例如，从版本 3.3 到版本 3.4 的更新是一个小版本更新。一旦小版本数字达到了 9，下一个版本就会增加大版本数字，也是增加 1，比如 LLVM 2.9 后面是 LLVM 3.0。相比前一个版本，大版本数字更新不必是一个大的修改，但是它们大致代表编译器五年的开发演进，如果和最近的大版本更新相比的话。

对于依赖 LLVM 的项目，常见的做法是使用主干版本，也就是在 SVN 仓库可以得到的最新的版本，其代价是，这个版本可能是不稳定的。最近，从版本 3.4 开始，LLVM 社区开始着手推出点发布，引入一个新的修订版数字。这项工作的第一个产品是 LLVM 3.4.1。点发布的目标是，从主干向最新的标签版本向后移植代码修正，而不引入新的特性，这样保持完全的兼容。每次点发布应该间隔三个月。由于这个新系统还处于婴儿期，在本章中我们将主要介绍 LLVM 3.4 的安装。LLVM 3.4 的预编译包的数量是很大的，但是你应该能够编译 LLVM 3.4.1，或者任意其它版本，只要遵循我们的指令，就不会有问题。

1.2 获取预编译安装包

为了让你能够轻松地在你的系统上安装软件，LLVM 贡献者准备了预编译的安装包，就是为特定平台编译的二进制文件，而不需要你自己编译它们。在某种境况下，编译任何软件都可能不是一目了然的；它可能需要花费时间，一般是不必要的，除非你在使用一个特别的平台，或者你在活跃地从事项目开发。因此，如果你想快捷地了解 LLVM，就去看一看可获得的预编译安装包。然而，在本书中，我们将鼓励你直接探索 LLVM 源代码树。你应该准备好能够自己从源代码树编译 LLVM。

获取 LLVM 预编译安装包有两种通常的方法：你可以从官方网站获取二进制文件，也可以从第三方获取 GNU/Linux 发布的安装包和 Windows 的安装文件。

1.2.1 获取官方预编译二进制文件

对于版本 3.4，可以从官方 LLVM 网站下载以下预编译包：

Architecture	Version
x86_64	Ubuntu (12.04, 13.10), Fedora 19, Fedora 20, FreeBSD 9.2, Mac OS X 10.9, Windows, and openSUSE 13.1
i386	openSUSE 13.1, FreeBSD 9.2, Fedora 19, Fedora 20, and openSUSE 13.1
ARMv7/ARMv7a	Linux-generic

要想查看针对不同版本的所有选项, 请访问 <http://www.llvm.org/releases/download.html>, 查看 **Pre-built Binaries** 小节有关你想下载的版本。例如, 为了在 Ubuntu 13.10 上下载 LLVM 并作系统范围的安装, 我们从站点得到文件的 URL, 并使用下面的命令:

```
$ sudo mkdir -p /usr/local; cd /usr/local
$ sudo wget http://llvm.org/releases/3.4/clang+llvm-3.4-x86_64-linux-gnuubuntu-13.10.
→tar.xz
$ sudo tar xvf clang+llvm-3.4-x86_64-linux-gnu-ubuntu-13.10.tar.xz
$ sudo mv clang+llvm-3.4-x86_64-linux-gnu-ubuntu-13.10 llvm-3.4
$ export PATH="$PATH:/usr/local/llvm-3.4/bin"
```

现在已经可以使用 LLVM 和 Clang 了。记住你需要永久地更新系统的 PATH 环境变量, 因为我们在上面最后一行所作的更新只对当前 shell 会话有效。你可以执行一个 Clang 的简单命令来测试它, 这个命令会打印出你所安装的 Clang 的版本:

```
$ clang -v
```

如果你在运行 Clang 的时候遇到问题, 试着从安装 Clang 的文件夹直接运行二进制文件, 以确定你的问题不是错误配置 PATH 环境变量的问题。如果它还是不能工作, 你可能下载了一个不兼容系统的预编译二进制文件。记住, 当被编译的时候, 二进制文件链接特定版本的动态库。在运行应用程序的时候发生链接的错误, 是一个清楚的征兆说明你在使用一个不兼容你的系统的二进制文件。

备注: 在 Linux 上, 举例来说, 报告链接错误的时候, 它会打印出二进制文件的名字, 无法加载的动态库的名字, 以及错误消息。注意在屏幕上打印出来的动态库的名字。这是一个清楚的信号, 说明系统的动态链接器和加载器无法加载这个库, 因为这个程序不是为兼容系统准备的。

要在其它系统上安装预编译的包, 可以遵循相同的步骤, 除了 Windows。Windows 的预编译包是一个易用的安装器, 它将 LLVM 树结构解开到你的 Program Files 文件夹的一个子文件夹。这个安装器还有一个选项以自动地更新你的 PATH 环境变量, 让你能够在任意的命令提示窗口使用 Clang 可执行文件。

1.2.2 使用包管理器

包管理器应用程序可用于多种系统, 也是一种获取和安装 LLVM/Clang 二进制文件的容易的方法。对于大多数用户, 通常这是推荐的安装 LLVM 和 Clang 的方法, 因为它自动处理依赖关系, 确保你的系统兼容所安装的二进制文件。

例如, 在 Ubuntu (10.04 以上), 你应该用下面的命令:

```
$ sudo apt-get install llvm clang
```

在 Fedora 18 上, 所用的命令行是类似的, 但是包管理器是不同的:

```
$ sudo yum install llvm clang
```

保持快照包的更新

包也可以从每晚的源代码快照编译出来，它包含 LLVM subversion 仓库上最新的提交。LLVM 开发者和使用者希望测试新近的版本，LLVM 第三方使用者想让他们本地的项目和主线的开发保持同步，快照对他们来说都是有用的。

Linux

Debian 和 Ubuntu Linux (i386 和 amd64) 仓库可用于下载从 LLVM subversion 仓库编译得到的快照。你可以在 <http://llvm.org/apt> 查看详情。

例如，要想在 Ubuntu 13.10 上安装 LLVM 和 Clang 的按天发布的版本，可以用下面的命令序列：

```
$ sudo echo "deb http://llvm.org/apt/raring/ llvm-toolchain-raring main" >> /etc/apt/
→sources.list
$ wget -O - http://llvm.org/apt/llvm-snapshot.gpg.key | sudo apt-key add -
$ sudo apt-get update
$ sudo apt-get install clang-3.5 llvm-3.5
```

Windows

Windows 特定的 LLVM/Clang 快照安装器可以从 <http://llvm.org/builds> 的 Windows snapshot builds 小节下载得到。最终的 LLVM/Clang 工具默认被安装在 C:\Program Files\LLVM\bin (这个位置可能会变，取决于发布)。注意，有一个单独的模仿 Visual C++ cl.exe 的 Clang 驱动器，名为 clang-cl.exe。如果你打算使用经典的 GCC 兼容的驱动器，就用 clang.exe。

备注：注意，快照发布是不稳定，可能是高度实验性的。

1.3 从源代码编译

没有预编译二进制文件时，LLVM 和 Clang 可以从头编译，首先获取源代码。从源代码编译项目是理解 LLVM 详细结构的好方法。此外，你将能够微调配置参数以获得一个定制的编译器。

1.3.1 系统要求

最新的 LLVM 支持的平台的列表可以在 <http://llvm.org/docs/GettingStarted.html#hardware> 找到。另外，<http://llvm.org/docs/GettingStarted.html#software> 描述了详细的且最新的编译 LLVM 所需的软件集合。在 Ubuntu 系统上，举例来说，软件依赖关系可以用下面的命令解决：

```
$ sudo apt-get install build-essential zlib1g-dev python
```

如果你在用一个旧版的 Linux 发布，其软件包已过时，就花点功夫更新一下系统。LLVM 源代码对编译它们的 C++ 编译器要求很严，使用旧版的 C++ 编译器很可能会导致编译失败。

1.3.2 获取源代码

LLVM 源代码的发布遵循一个 BSD 风格的许可证，可以从官方网站或者 SVN 仓库下载。要下载 3.4 版本的源代码，要么去网站，<http://llvm.org/releases/download.html#3.4>，要么依照以下方法直接下载并为编译准备好源代码。注意你总是会需要 Clang 和 LLVM，但是 clang-tools-extra 是可选的。然而，如果你打算练习第 10 章（Clang 工具和 LibTooling）的教程，你会用到它的。参考下一章了解如何编译其它项目。用下面的命令以下载和安装 LLVM、Clang、和附加工具：

```
$ wget http://llvm.org/releases/3.4/llvm-3.4.src.tar.gz
$ wget http://llvm.org/releases/3.4/clang-3.4.src.tar.gz
$ wget http://llvm.org/releases/3.4/clang-tools-extra-3.4.src.tar.gz
$ tar xzf llvm-3.4.src.tar.gz; tar xzf clang-3.4.src.tar.gz
$ tar xzf clang-tools-extra-3.4.src.tar.gz
$ mv llvm-3.4 llvm
$ mv clang-3.4 llvm/tools/clang
$ mv clang-tools-extra-3.4 llvm/tools/clang/tools/extra
```

在 Windows 下载的源代码可以用 gnuzip、WinZip、或者其它可用的解压缩工具解包。

SVN

要想直接从 SVN 仓库获取源代码，首先请确认你的系统上安装了 subversion 软件包。下一步是决定你是想要仓库中的最新的版本，还是一个稳定的版本。如果想要最新的版本（主干），你可以用下面的命令序列，假设你的当前文件夹就是你想存放源代码的地方：

```
$ svn co http://llvm.org/svn/llvm-project/llvm/trunk llvm
$ cd llvm/tools
$ svn co http://llvm.org/svn/llvm-project/cfe/trunk clang
$ cd ../projects
$ svn co http://llvm.org/svn/llvm-project/compiler-rt/trunk compiler-rt
$ cd ../tools/clang/tools
$ svn co http://llvm.org/svn/llvm-project/clang-tools-extra/trunk extra
```

如果你想获取一个稳定的版本（例如，版本 3.4），用 `tags/RELEASE_34/final` 代替所用命令中的 `trunk`。你可能还关心浏览 LLVM SVN 仓库的简易方法，以查看提交历史、日志、和源代码树结构。对此，你可以访问 <http://llvm.org/viewvc>。

Git

你也可以从和 SVN 保持同步的 Git 镜像仓库获取源代码：

```
$ git clone http://llvm.org/git/llvm.git
$ cd llvm/tools
$ git clone http://llvm.org/git/clang.git
$ cd ../projects
$ git clone http://llvm.org/git/compiler-rt.git
$ cd ../tools/clang/tools
$ git clone http://llvm.org/git/clang-tools-extra.git
```

1.3.3 编译和安装 LLVM

这里会解释编译和安装 LLVM 的多种方法。

利用自动工具生成的配置脚本

编译 LLVM 的标准方法，是通过配置脚本生成平台特定的 `Makefile`，这些脚本是由 GNU 的自动工具创建的。此编译系统是相当流行的，你大概知道它。它支持若干不同的配置选项。

备注：只有当你想要修改 LLVM 编译系统的时候，你才需要在你的机器上安装 GNU 自动工具，在这种情况下，你将生成新的配置脚本。通常，这是不必要的。

花一点时间，用下面的命令看一下可能的选项：

```
$ cd llvm
$ ./configure --help
```

其中一些选项需要简单的解释：

- `--enable-optimized`：这个选项让我们能够编译得到不支持调试且开启优化的 LLVM/Clang。默认，这个选项是关闭的。调试支持，还有关闭优化，是被推荐的，如果你用 LLVM 库作开发，但是对于实际应用，就应该舍弃之，因为关闭优化会导致 LLVM 极大地降速。
- `--enable-assertions`：这个选项开启代码中的断言。在开发 LLVM 核心库的时候，这个选项非常有用。默认，它是开启的。

- `--enable-shared`：这个选项让我们能够将 LLVM/Clang 库编译为共享库，并用之链接 LLVM 工具。如果你计划在 LLVM 编译系统之外开发一个工具，并且希望动态地链接 LLVM 库，就应该开启它。默认，这个选项是关闭的。
- `--enable-jit`：这个选项为所有支持即时编译（Just-In-Time Compilation）的目标开启即时编译。默认，它是开启的。
- `--prefix`：这是安装目录的路径，最终的 LLVM/Clang 工具和库会被安装到这里；例如，`--prefix=/usr/local/llvm`，这样二进制文件会安装到 `/usr/local/llvm/bin`，库文件会安装到 `/usr/local/llvm/lib`。
- `--enable-targets`：这个选项让我们能够选择一组目标，编译器必须能够为这些目标生成代码。值得提及的是，LLVM 能够作交叉编译，也就是说，编译得到的程序将在其它平台上运行，例如 ARM、MIPS 等。这个选项指定代码生成库需要包含哪些后端。默认，所有的目标都会被编译，但是通过仅仅指定你所关心的目标，你可以节省编译时间。

备注：这个选项不足以生成独立的交叉编译器。参考第 8 章（交叉平台编译）了解生成交叉编译器的必要的步骤。

以期望的参数运行配置命令之后，你需要用经典的 `make` 和 `make install` 二重奏完成编译。我们接下来会给你一个例子。

以 Unix 编译和配置

在这个例子中，我们会用一组命令序列编译得到一个不优化（支持调试）的 LLVM/Clang，这些命令适用于任意基于 Unix 的系统或者 Cygwin。我们会编译它，将它安装在我们的 `home` 目录，而不是如前面的例子所示的那样，将它安装在 `/usr/local/llvm`，以说明如何在无根权限的情况下安装 LLVM。这是作为一个开发者所习以为常的。这样，你还可以安装并维护多个版本。如果你想要，你可以修改安装文件夹为 `/usr/local/llvm`，作系统范围的安装。只不过记得在创建安装目录并运行 `make install` 的时候，使用 `sudo` 命令。所用的命令序列如下：

```
$ mkdir where-you-want-to-install
$ mkdir where-you-want-to-build
$ cd where-you-want-to-build
```

在这一节，我们会创建一个单独的目录来存放目标文件，即编译中间副产品。不要在存放源代码的相同的文件夹中编译。使用下面的命令，其中的参数在前面小节解释过了：

```
$ /PATH_TO_SOURCE/configure --disable-optimized --prefix=../where-youwant-to-install
$ make && make install
```

可选地，你可以用 `make -jN` 命令来启动最多 `N` 个编译器实例并行地工作，以加速编译过程。例如，你可以试验 `make -j4`（或者更大一点的数字），如果你的处理器是四核的。

编译并安装所有组件需要一点时间。注意，编译脚本也会处理你所下载的存放在 LLVM 源代码树中的其它仓

库目录。不需要单独地配置 Clang 或 Clang 附加工具。为了检查编译是否成功，使用 shell 命令 `echo $?` 总是可行的。shell 变量 `$?` 返回你在 shell 会话中运行的最后一个进程的退出码，而 `echo` 将它打印在屏幕上。因此，重要的是在你的 `make` 命令之后立即运行这个命令。如果编译成功了，`make` 命令总是返回 0，如其它成功完成执行的程序一样：

```
$ echo $?  
0
```

配置你的 shell 的 `PATH` 环境变量，使得能够轻松地访问刚刚安装的二进制文件，然后通过查询 Clang 版本来完成第一次测试：

```
$ export PATH="$PATH:where-you-want-to-install/bin"  
$ clang -v  
clang version 3.4
```

利用 CMake 和 Ninja

LLVM 给出了另一种交叉平台编译系统，代替传统的配置脚本，它是基于 CMake 的。CMake 可以为你的平台生成专用的 Makefile，其生成方法和配置脚本一样，但是 CMake 更加灵活，还可以为其它系统生成编译文件，例如 Ninja，Xcode，和 Visual Studio。

另一方面，Ninja 是一种小巧且快速的编译系统，代替 GNU Make 和它关联的 Makefile。如果你对 Ninja 背后的动机和故事感到好奇，就去访问 <http://aosabook.org/en/posa/ninja.html>。CMake 可以被配置为生成 Ninja 编译文件，而不是 Makefile，让你可以选择使用 CMake 和 GNU Make，或者 CMake 和 Ninja。

然而，利用后者，可以让你享受非常快的来回的时光，当你修改 LLVM 源代码并重编译它的时候。这种场景会尤其有用，如果你想要在 LLVM 源代码树内部开发一个工具或者插件，并且依靠 LLVM 编译系统来编译你的项目。

确定你已经安装 CMake 和 Ninja。例如，在 Ubuntu 系统上，运行下面的命令：

```
$ sudo apt-get install cmake ninja-build
```

LLVM 和 CMake 还提供了若干编译定制选项。完整的选项列表可以从 <http://llvm.org/docs/CMake.html> 得到。下面给出了一个选项列表，它和我们之前介绍的基于自动工具的编译系统的选项集相对应。这些选项的默认值和相应的配置脚本选项的默认值一样：

- `CMAKE_BUILD_TYPE`：这是一个字符串值，指定编译类型是 `Release` 还是 `Debug`。`Release` 编译等价于配置脚本中的 `--enable-optimized` 选项，而 `Debug` 编译等价于 `--disable-optimized` 选项。
- `CMAKE_ENABLE_ASSERTIONS`：这是一个布尔值，对应 `--enable-assertions` 配置选项。
- `BUILD_SHARED_LIBS`：这是一个布尔值，对应 `--enable-shared` 配置选项，指定这些库是共享的还是静态的。Windows 平台不支持共享库。
- `CMAKE_INSTALL_PREFIX`：这是一个字符串值，对应 `--prefix` 配置脚本，指定安装路径。

- `LLVM_TARGETS_TO_BUILD`：这是一个以分号分隔的要编译的目标的列表，大致对应 `-enable-targets` 配置选项中以逗号分隔的目标的列表。

要想设置这些成对的参数-数值中的任意一个，就将 `-DPARAMETER=value` 参数传送给 `cmake` 命令。

在 Unix 上利用 CMake 和 Ninja 编译

我们将重新产生之前为配置脚本给出的相同的例子，但是这次，我们将用 CMake 和 Ninja 编译它：

首先，创建一个文件夹以存放编译和安装文件：

```
$ mkdir where-you-want-to-build
$ mkdir where-you-want-to-install
$ cd where-you-want-to-build
```

记住，你需要用一个和存放 LLVM 源代码的文件夹不同的文件夹。接下来，是时候以你选择的选项集合启动 CMake 了：

```
$ cmake /PATHTOSOURCE -G Ninja -DCMAKE_BUILD_TYPE="Debug" -DCMAKE_INSTALL_PREFIX="../  
↪where-you-want-to-install"
```

你应该用你的 LLVM 源代码文件夹的绝对位置代替 `/PATHTOSOURCE`。你可以省去参数 `-G Ninja`，如果你想使用传统的 GNU Makefile。现在，根据你的选择，执行 `ninja` 或者 `make`，以完成编译。对于 `ninja` 来说，用下面的命令：

```
$ ninja && ninja install
```

对于 `make` 来说，使用下面的命令：

```
$ make && make install
```

如之前我们在上一个例子中所做的那样，我们可以输入一个简单的命令来检查编译成功与否。记住，在最后的编译命令之后立即使用它，中间不能运行其它命令，因为它返回的是当前 `shell` 会话中你运行的最后的程序的退出码：

```
$ echo $?  
0
```

如果前面的命令返回 0，就说明编译成功了。最后，配置你的 `PATH` 环境变量，使用你的新的编译器：

```
$ export PATH=$PATH:where-you-want-to-install/bin
$ clang -v
```

解决编译错误

如果编译命令返回一个非零值，就意味着发生了错误。在这种情况下，Make 或者 Ninja 会打印这个错误让你查看它。务必集中分析出现的第一个错误。在一个 LLVM 的稳定发布版本中，编译错误典型地发生在你的系统未达到所需的软件版本的标准的时候。最常见的问题源于使用了一个过时的编译器。例如，使用 GNU g++ 版本 4.4.3 编译 LLVM 3.4 会导致下面的编译错误，在成功地编译了过半的 LLVM 源代码之后：

```
[1385/2218] Building CXX object projects/compiler-rt/lib/interception/
CMakeFiles/RTInterception.i386.dir/interception_type_test.cc.o
FAILED: /usr/bin/c++ (...)_test.cc.o -c /local/llvm-3.3/llvm/projects/
compiler-rt/lib/interception/interception_type_test.cc
test.cc:28: error: reference to 'OFF64_T' is ambiguous
interception.h:31: error: candidates are: typedef __sanitizer::OFF64_T
OFF64_T
sanitizer_internal_defs.h:80: error: typedef __
sanitizer::u64 __sanitizer::OFF64_T
```

为了解决这个错误，你要改动 LLVM 源代码以规避这个问题（如果你上网搜索或者亲自去查看源代码，你会找到解决它的方法），但是你不希望修正你想要编译的每一个 LLVM 版本。更新你的编译器简单多了，肯定也是最适当的解决方案。

一般来说，当你在一个稳定版本中遇到编译错误时，就专心地去寻找你的系统和推荐的设置之间的差异。记住，稳定的版本已经在若干平台上测试过了。另一方面，如果你尝试着编译一个不稳定的 SVN 发布版本，那么一个近期的提交破坏了在你的系统上的编译是可能的，而回退到一个可用的 SVN 发布版本也是容易的。

利用其它的 Unix 方法

一些 Unix 系统提供了包管理器，它们自动从源代码编译并安装应用程序。它们提供了对等的源代码编译功能，此功能预先在你的系统上测试过，也会尝试解决包依赖问题。现在我们将在编译并安装 LLVM 和 Clang 的上下文中评估这样的平台：

- 对于使用 MacPorts 的 Mac OS X，我们可以使用下面的命令：

```
$ port install llvm-3.4 clang-3.4
```

- 对于使用 Homebrew 的 Mac OS X，我们可以使用下面的命令：

```
$ brew install llvm --with-clang
```

- 对于使用 ports 的 FreeBSD 9.1，我们可以使用下面的命令（注意，从 FreeBSD 10 开始，Clang 是默认的编译器，因此它已经安装好了）：

```
$ cd /usr/ports/devel/llvm34
$ make install
```

(续下页)

(接上页)

```
$ cd /usr/ports/lang/clang34
$ make install
```

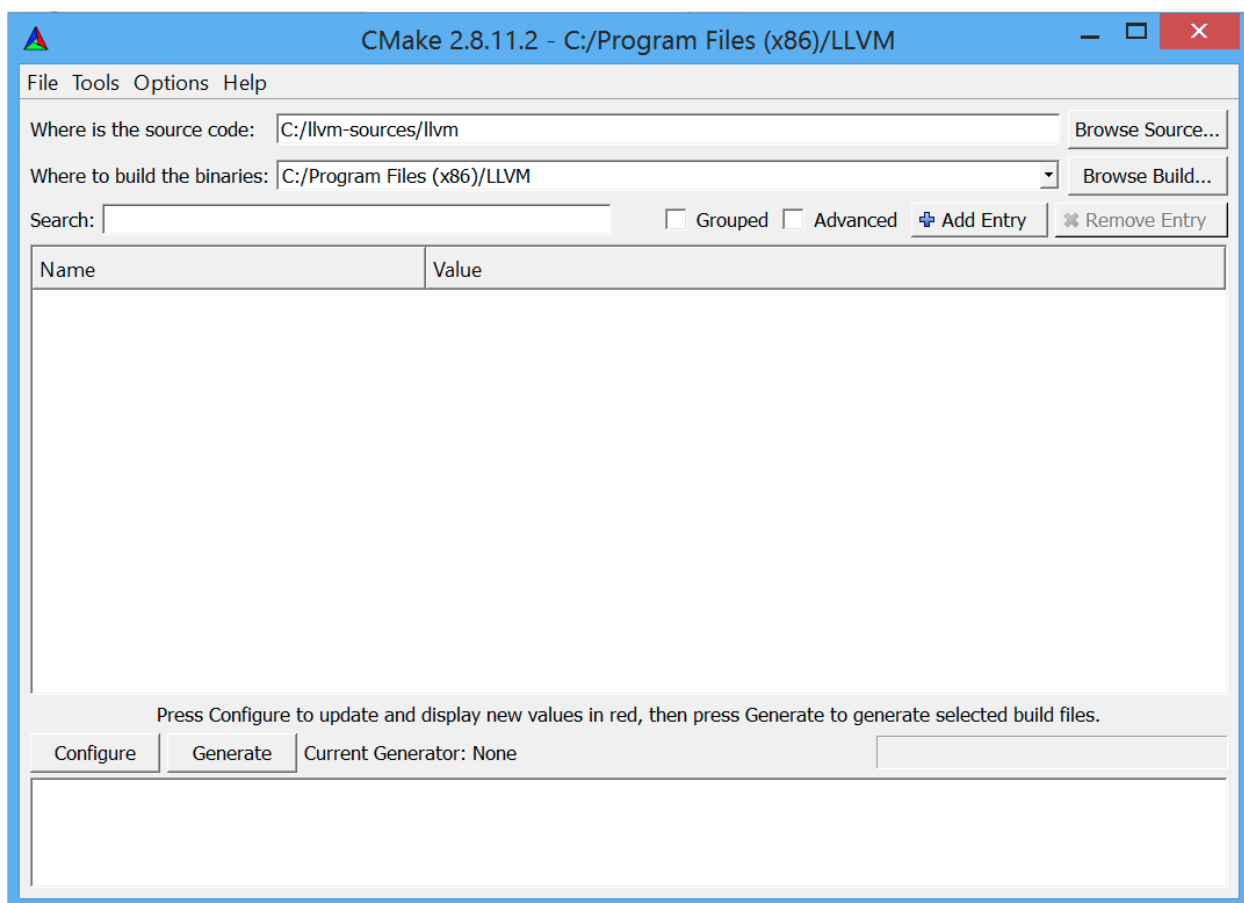
- 对于 Gentoo Linux，我们可以使用下面的命令：

```
$ emerge sys-devel/llvm-3.4 sys-devel/clang-3.4
```

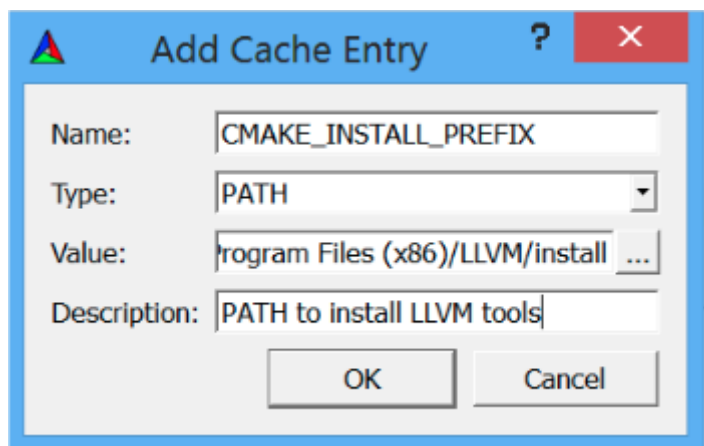
1.3.4 Windows 和 Microsoft Visual Studio

为了在 Microsoft Windows 上编译 LLVM 和 Clang，我们要使用 Microsoft Visual Studio 2012 和 Windows 8。执行下面的步骤：

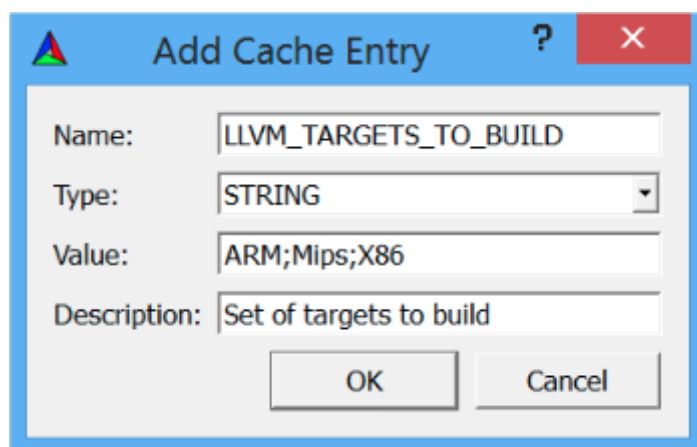
1. 获取一份 Microsoft Visual Studio 2012 的副本。
2. 下载并安装官方的 CMake 工具二进制发布版本，下载地址 <http://www.cmake.org>。在安装的时候，务必勾选 Add CMake to the system PATH for all users 选项。
3. CMake 会生成 Visual Studio 所需的项目文件以配置并编译 LLVM。首先，运行 cmake-gui 图形工具。然后，点击 Browse Source ... 按钮，选择 LLVM 源代码目录。接着，点击 Browse Build 按钮，选择一个存放 CMake 生成文件的目录，将来 Visual Studio 会使用它，如下面的截屏所示：



4. 点击 Add Entry 并定义 CMAKE_INSTALL_PREFIX 以指定 LLVM 工具的安装路径，如下面的截屏所示：



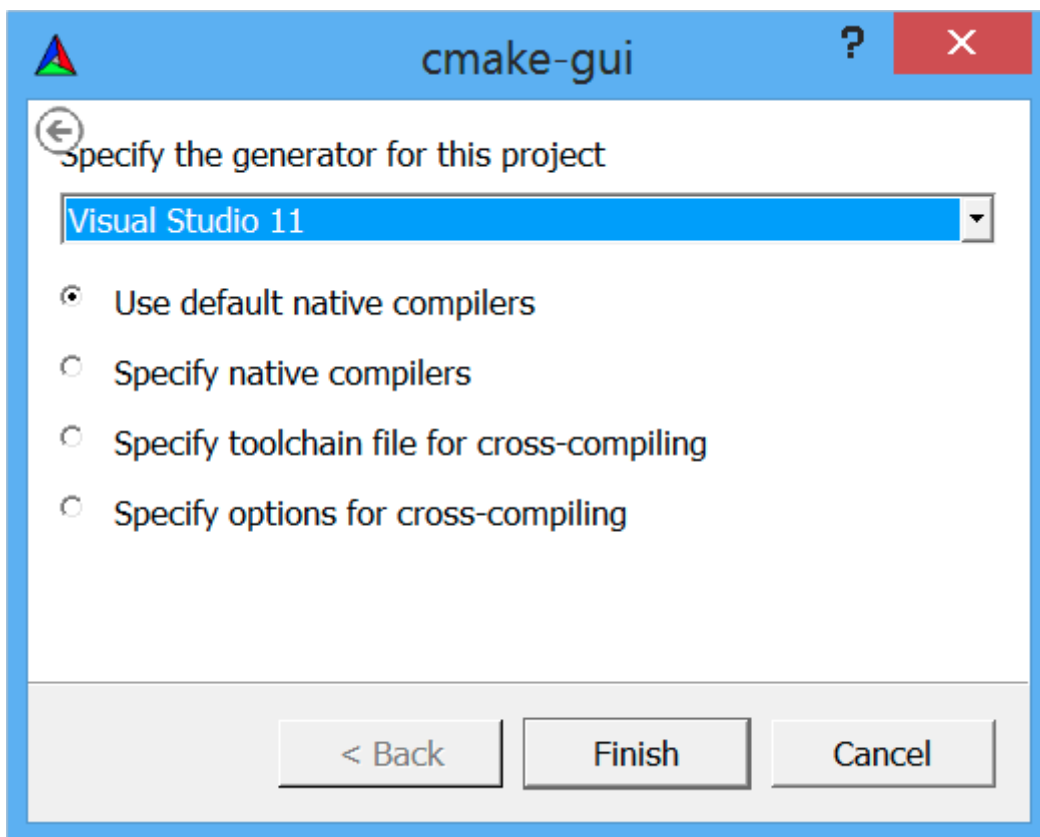
5. 此外，可以通过 LLVM_TARGETS_TO_BUILD 指定支持的目标集合，如下面的截屏所示。可选地，你可以添加任意其它的项来定义我们之前讨论过的 CMake 参数。



6. 点击 Configure 按钮。弹出的窗口询问使用什么项目生成器和编译器；选择 Visual Studio 2012（选项 Visual Studio 11）和 Use default native compilers。点击 Finish，如下面的截屏所示：

备注： 对于 Visual Studio 2013，使用 Visual Studio 12 生成器。生成器的名字使用 Visual Studio 版本，而不是它的商业名称。

7. 在配置完成之后，点击 Generate 按钮。这样，Visual Studio solution 文件，LLVM.sln，会被写到指定的 build 目录中。进入这个目标，双击这个文件；它会在 Visual Studio 中打开 LLVM solution。
8. 要想自动地编译并安装 LLVM/Clang，在左边的树视图中，展开 CMakePredefinedTargets，右击 INSTALL，选择 Build 选项。预定义的 INSTALL 目标会指示系统编译并安装所有 LLVM/Clang 的工具和库，如下面的截屏所示：
9. 要想有选择地编译并安装指定的工具和库，就在左侧的树视图窗口中选择相应的项，右击它并选择 Build 选项。



10. 将 LLVM 二进制安装目录添加到系统的 PATH 环境变量。

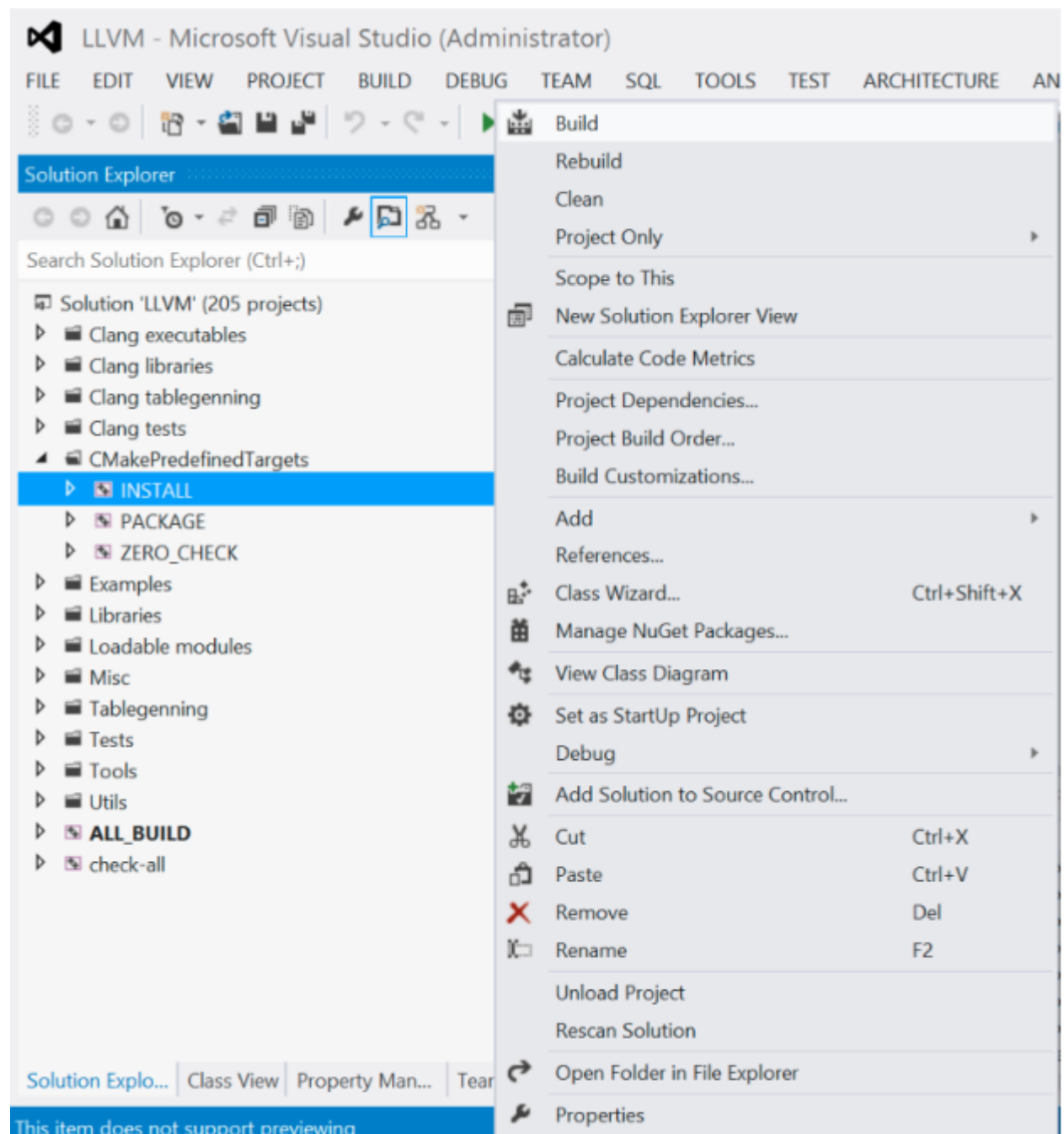
在我们的例子中，安装目录是 C:\Program Files (X86)\LLVM\install\bin。若要不更新 PATH 环境变量就测试安装成功与否，就在命令提示窗口运行下面的命令：

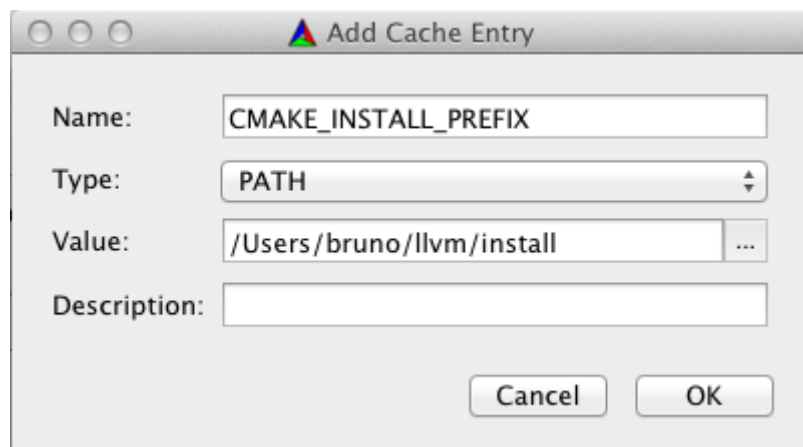
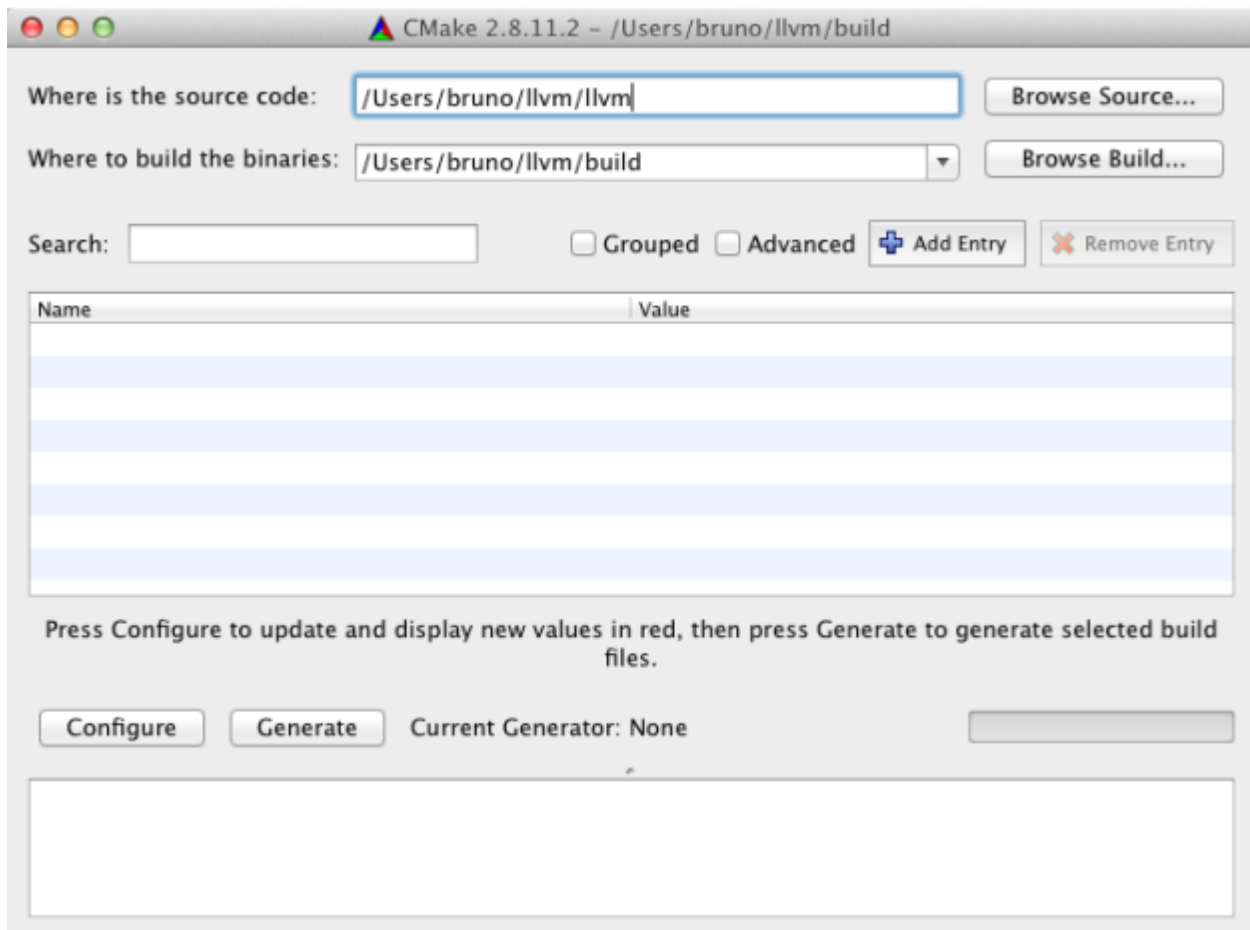
```
C:>"C:\Program Files (x86)\LLVM\install\bin\clang.exe" -v
clang version 3.4...
```

1.3.5 Mac OS X 和 Xcode

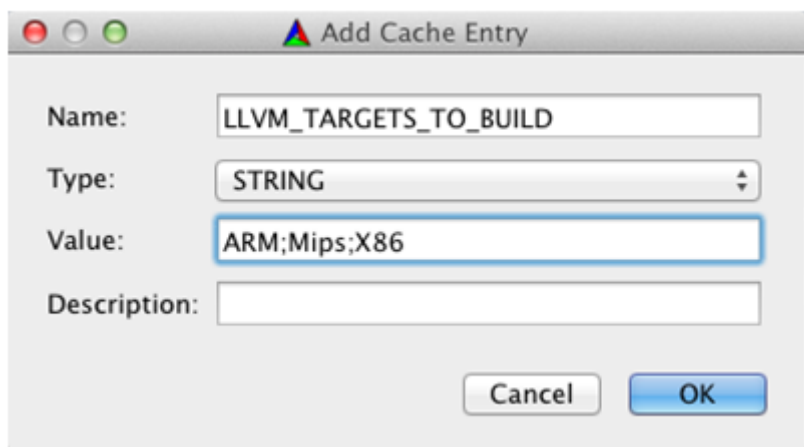
尽管利用前面描述的常规 Unix 指令可以为 Mac OS X 编译 LLVM，但是也可以利用 Xcode：

1. 获取 Xcode 的副本。
2. 下载并安装官方 CMake 工具的二进制发布，下载地址 <http://www.cmake.org>。不要忘记勾选 Add CMake to the system PATH for all users 选项。
3. CMake 能够生成 Xcode 的项目文件。首先，运行 cmake-gui 图形工具。然后，如前面的截屏所示，点击 Browse Source 按钮并选择 LLVM 源代码目录。接着，点击 Browse Build 按钮并选择存放 CMake 生成文件的目录，Xcode 会使用这些文件。
4. 点击 Add Entry，定义 CMAKE_INSTALL_PREFIX 以指定 LLVM 工具的安装路径。

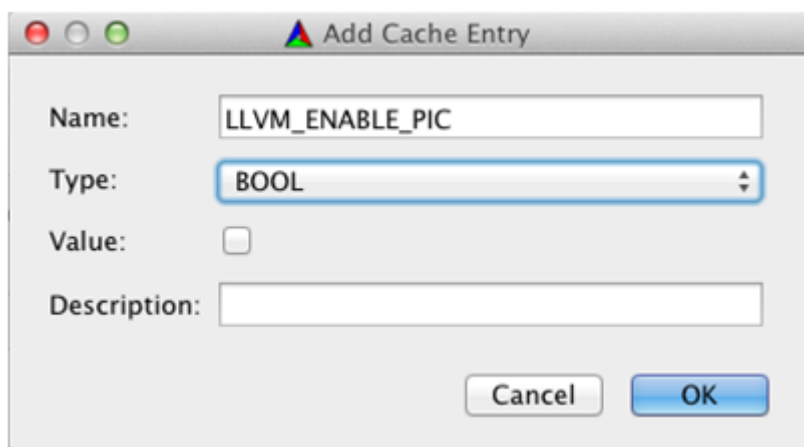




5. 还有，支持的目标集合可以通过 `LLVM_TARGETS_TO_BUILD` 定义。可选地，你可以添加任意其它的定义 CMake 参数的项，我们之前讨论过这些参数。

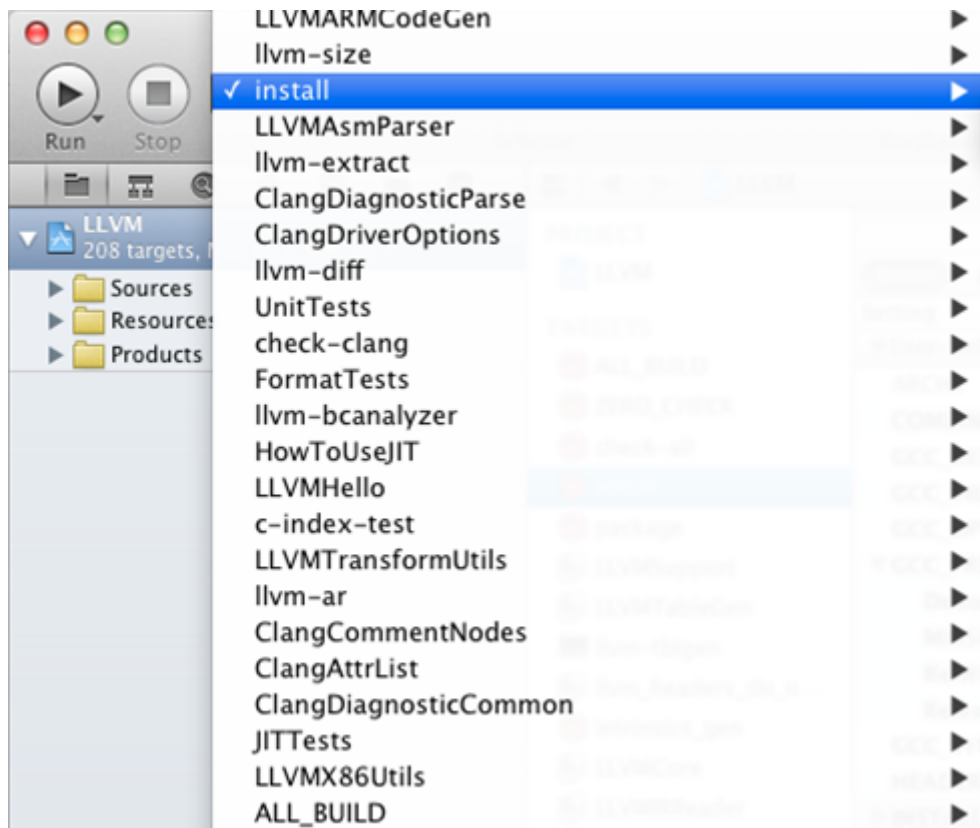
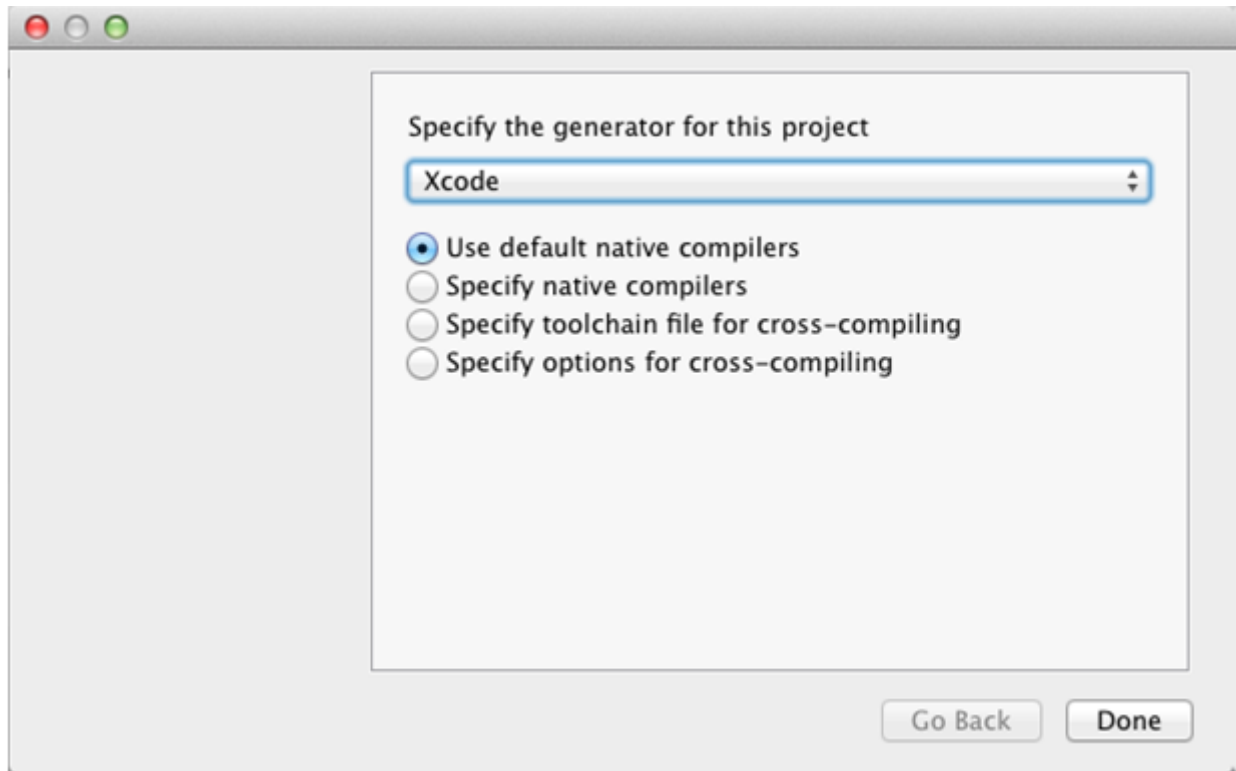


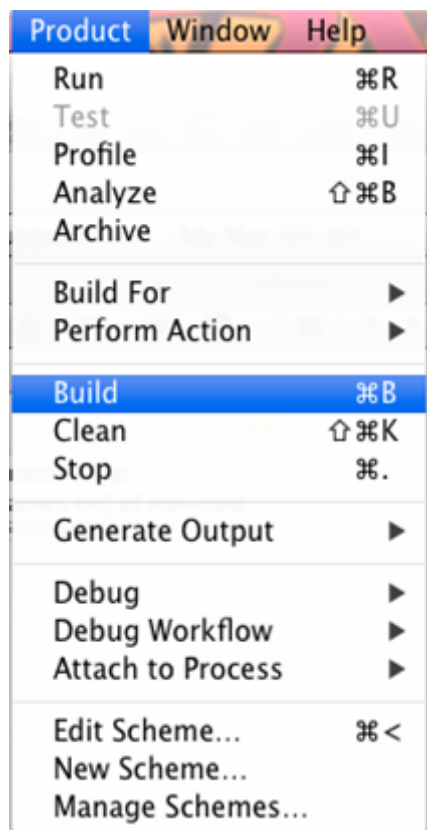
6. Xcode 不支持生成 LLVM 位置无关代码（PIC，Position Independent Code）库。点击 Add Entry 并添加 `LLVM_ENABLE_PIC` 变量，它是 `BOOL` 类型，空着勾选框，如下面的截屏所示：



7. 点击 Configure 按钮。弹出的对话框询问这个项目的生成器和要用的编译器。选择 Xcode 和 Use default native compilers。点击 Finish 按钮结束配置过程，如下面的截屏所示：
8. 完成配置之后，点击 Generate 按钮。随即，`LLVM.xcodeproj` 文件会被写到 `build` 目录中，这个目录是之前设置的。进入这个目录并双击这个文件，这样就会在 Xcode 中打开 LLVM 项目。
9. 为了编译并安装 LLVM/Clang，选择 `install` 方案。
10. 接下来，点击 Product 菜单，然后选择 Build 选项，如下面的截屏所示：
11. 将 LLVM 二进制文件的安装目录添加到系统的 `PATH` 环境变量中。

在我们的例子中，安装二进制的文件夹是 `/Users/Bruno/llvm/install/bin`。为了测试安装是否成功，从安装目录运行 `clang` 工具，如下所示：





```
$ /Users/Bruno/llvm/install/bin/clang -v  
clang version 3.4...
```

1.4 总结

本章详细地说明了如何安装 LLVM 和 Clang，演示了如何使用官方预编译安装包的现成二进制文件，第三方的包管理器，以及每天的代码快照。此外，我们详细介绍了如何在不同的操作系统环境下从源代码编译 LLVM 项目，利用标准的 Unix 工具和 IDE。

在下一章，我们将介绍如何安装其它的基于 LLVM 的项目，你可能会用到它们。典型地，这些外部项目实现了一些工具，它们不属于主 LLVM SVN 仓库，并且是单独发布的。

第 2 章外部项目

核心 LLVM 和 Clang 代码库之外的项目是外部项目，需要单独下载。在本章中，我们会介绍多种官方 LLVM 的外部项目，解释如何编译安装它们。只关注核心 LLVM 工具的读者可以跳过本章，想阅读时再回来。

在本章中，我们将介绍以下项目，包括如何安装它们：

- Clang extra tools
- Compiler-RT
- DragonEgg
- LLVM test suite
- LLDB
- Libc++

除此之外，还有两个本书未包括的官方 LLVM 项目：Polly（多面优化器），lld（LLVM 链接器）。当前 lld 正在开发之中。

预编译的安装包不包含本章将讨论的任何外部项目，除了 Compiler-RT。因此，不同于前一章，我们将只介绍如何下载并编译它们。

不要期望这些项目的成熟度能够比肩核心 LLVM/Clang 项目。其中一些还处于试验期或者婴儿期。

2.1 介绍 Clang extras

LLVM 最显著的设计决策是分离前端和后端，分别为单独的 LLVM 核心库和 Clang。LLVM 起步于一套以 LLVM 中间表示 (IR) 为基础的工具，依赖于一个定制的 GCC，用 GCC 将高级语言翻译为它的特殊 IR，存储为 bitcode (位码) 文件。Bitcode 是效仿 Java bytecode 文件而新造的一个术语。Clang 作为由 LLVM 团队特别设计的第一前端，当它如核心 LLVM 一般具备高水准的品质、清晰的文档、库组织方式时，它成为 LLVM 项目的一个重要里程碑。Clang 不仅将 C 和 C++ 转换为 LLVM IR，而且能够监督整个编译过程，作为一个灵活的编译器驱动器，努力与 GCC 兼容共处。

自此以后，我们将 Clang 看作一个前端编译器，而不是一个编译器驱动器，它负责将 C 和 C++ 程序翻译为 LLVM IR。利用 Clang 可以写出强大的工具，让 C++ 程序员能够自由地探索 C++ 热点技术，例如 C++ 代码重构工具和源代码分析工具。这是 Clang 程序库激动人心的一面。Clang 自带的一些工具或许能让你见识其程序库的用途：

- Clang Check：它能够作语法检查，实施快速修正以解决常见的问题，还能够输出任意程序的内部 Clang 抽象语法树 (AST, Abstract Syntax Tree)
- Clang Format：它是一个工具，也是一个库，LibFormat，它既能整理代码缩进，也能格式化任意的 C++ 代码，使之符合 LLVM 编码标准、Google 的风格规范、Chromium 的风格规范、Mozilla 的风格规范、和 WebKit 的风格规范

代码仓库 clang-tools-extra 收集了更多建立在 Clang 之上的应用程序。它们能够读入大型 C 或 C++ 代码库并执行各种代码重构或分析。下面列举其中一些工具，但不仅限于此：

- Clang Modernizer：这是一个代码重构工具，它扫描 C++ 代码并将旧风格的结构转换为符合更现代的风格，这些新风格是由新的标准提议的，例如 C++-11 标准
- Clang Tidy：这是一个剥绒机工具，它检查常见的编程错误，这些错误违背了 LLVM 或者 Google 的编码标准。
- Modularize：它帮你找出适合组成一个模块 (module) 的 C++ 头文件，模块是 C++ 标准委员会目前正在讨论的一个新概念 (请参考第 10 章，Clang 工具和 LibTooling，以了解更多信息)
- PPTTrace：这是一个跟踪 Clang C++ 预处理器的活动的简单工具

至于如何运用这些工具，以及如何编译我们自己的工具，第 10 章 (Clang 工具和 LibTooling) 将详细地介绍它们。

2.1.1 编译和安装 Clang 附加 (extra) 工具

你可以获取这个项目的官方源代码，例如 3.4 版本：<http://releases.llvm.org/3.4/clang-tools-extra-3.4.src.tar.gz>。另外，你也可以查看所有可获取的版本：<http://releases.llvm.org/>。凭借 LLVM 编译系统，将这套工具和核心 LLVM、Clang 源代码一起编译，编译轻而易举。这要求按如下方式将其源代码目录放在 Clang 源代码树中：

```
$ wget http://releases.llvm.org/3.4/clang-tools-extra-3.4.src.tar.gz
$ tar xzf clang-tools-extra-3.4.src.tar.gz
$ mv clang-tools-extra-3.4 llvm/tools/clang/tools/extra
```

你也可以直接从 LLVM 官方 subversion 仓库获取源代码：

```
$ cd llvm/tools/clang/tools
$ svn co http://llvm.org/svn/llvm-project/clang-tools-extra/trunk extra
```

如前一章说过的那样，如果你想获取稳定的 3.4 版源代码，就用 `tags/RELEASE_34/final` 替换 `trunk`。另外，如果你喜欢使用 GIT 版本控制软件，你可以用下面的命令下载：

```
$ cd llvm/tools/clang/tools
$ git clone http://llvm.org/git/clang-tools-extra.git extra
```

把源代码放在 Clang 树中之后，你必须用 Cmake 或者自动工具生成的 `configure` 脚本去编译它，参考第 1 章（编译和安装 LLVM）。按如下方式运行 `clang-modernize` 工具，可测试是否安装成功：

```
$ clang-modernize --version
clang-modernizer version 3.4
```

2.1.2 理解 Compiler-RT

Compiler-RT (runtime) 项目为硬件不支持的低级功能提供目标特定的支持。举例来说，32 位目标通常缺少 64 位除法指令。Compiler-RT 提供一个目标特定的优化的函数，它用 32 位指令实现 64 位除法，从而解决这个问题。它是 libgcc 的 LLVM 等价物，提供相同的功能。而且，它运行时支持地址和内存清洁 (sanitizer) 工具。你可以下载 Compiler-RT 3.4 版源代码：<http://releases.llvm.org/3.4/compiler-rt-3.4.src.tar.gz>，或者查看更多的版本：<http://releases.llvm.org/>。

Compiler-RT 是基于 LLVM 的编译器工具链中的重要组件，我们在前一章已经介绍了如何安装它。如果你还不清楚，记住把它的源代码放在 LLVM 源代码树中的 `projects` 文件夹中，如以下命令所示：

```
$ wget http://releases.llvm.org/3.4/compiler-rt-3.4.src.tar.gz
$ tar xzf compiler-rt-3.4.src.tar.gz
$ mv compiler-rt-3.4 llvm/projects/compiler-rt
```

或者，你可以从它的 SVN 仓库下载：

```
$ cd llvm/projects
$ svn checkout http://llvm.org/svn/llvm-project/compiler-rt/trunk compiler-rt
```

也可以从一个 GIT 镜像下载：

```
$ cd llvm/projects
$ git clone http://llvm.org/git/compiler-rt.git
```

备注：此外，Compiler-RT 还能在 GNU/Linux、Darwin、FreeBSD 和 NetBSD 系统上工作。已支持的架构包括：i386, x86_64, PowerPC, SPARC64, ARM。

2.1.3 试验 Compiler-RT

为了看到 Compiler-RT 程序库运作的典型情境，你可以做一个简单的实验。写一个执行 64 位除法的 C 程序：

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

int main() {
    uint64_t a = 0ULL, b = 0ULL;
    scanf("%lld %lld", &a, &b);
    printf("64-bit division is %lld\n", a / b);
    return EXIT_SUCCESS;
}
```

备注：下载示例代码

你可以从 <http://www.packtpub.com> 用你的帐号下载示例代码文件，为所有你已经购买的 Packt 书籍。如果你在别处购买了这本书，你就访问 <http://www.packtpub.com/support> 并注册，这样代码文件就可以通过邮件直接发送给你。

如果你有 64 位 x86 系统，用你的 LLVM 编译器执行如下两个命令：

```
$ clang -S -m32 test.c -o test-32bit.S
$ clang -S test.c -o test-64bit.S
```

参数-S 指示编译器生成 x86 汇编语言，-m32 指示编译器生成 32 位 x86 程序。这里，为这个程序生成汇编语言文件 test-32bit.S。当你打开这个文件时，你会看到，在程序执行除法的地方有一个古怪的调用（call）：

```
calll    __udivdi3
```

这个函数由 Compiler-RT 定义，展示了怎么使用该程序库。然而，假如省去-m32 参数，使用 64 位 x86 编译器，如第 2 个编译命令，生成 test-64bit.S 汇编语言文件，你将看不到上述调用，因为这个程序不需要 Compiler-RT 协助，它简单地用一条指令执行除法：

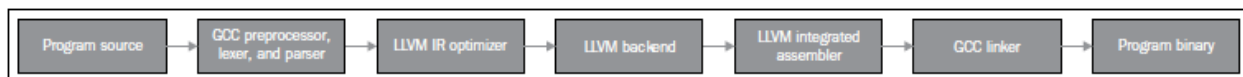

```
divq    -24(%rbp)
```

2.2 使用 DragonEgg 插件

之前解释过，起初 LLVM 是一个依赖于 GCC 的项目，当时它还没有自己的 C/C++ 前端编译器。在那种情况下使用 LLVM，你需要下载一份经过修改的 GCC 源代码，称为 `llvm-gcc`，并且整体编译它。因为要编译完整的 GCC 程序包，所以编译过程十分漫长，并且有点复杂，靠自己重新编译它需要用到完整的 GCC 相关门类的知识。DragonEgg 项目的出现为此提供了一个简明的方案，它利用 GCC 插件系统将 LLVM 逻辑分离到它自己的相对较小的代码树中。以这种方式，使用者不再需要重新编译整个 GCC 程序包，仅仅编译一个插件，然后将它加载到 GCC 中。DragonEgg 也是 LLVM 项目伞中唯一一个以 GPL 许可的项目。

Clang 出现之后，DragonEgg 依然存留直到今天，因为 Clang 只处理 C 和 C++ 语言，而 GCC 能够解析多种多样的语言。通过 DragonEgg 插件，你能够将 GCC 用作 LLVM 编译器的前端，能够编译大多数 GCC 支持的语言：Ada、C、C++ 和 FORTRAN，部分支持 Go、Java、Obj-C 和 Obj-C++。

这个插件将 GCC 的中后端替换成 LLVM 的中后端，自动执行所有的编译步骤，如你对第一流编译器驱动器期望的那样。下图描绘了这种新的编译流水线情景：



根据你的愿望，你可以用参数 `-fplugin-arg-dragonegg-emit-ir -S` 让编译流水线在生成 LLVM IR 阶段停止，而后用 LLVM 工具分析和调查前端的结果，或者自己用 LLVM 工具完成编译。待会我们将举一个例子。

作为一个 LLVM 附属项目，DragonEgg 不放在和 LLVM 主项目相同的地方。写作本文之时，最新的 DragonEgg 稳定版是 3.3，依赖 LLVM 3.3 工具集。因此，对于生成的 LLVM `bitcode`，即存储在磁盘上的 LLVM IR 程序文件，你只能使用 3.3 版 LLVM 工具分析它，优化它，或者继续编译它。DragonEgg 的官方网站是 <http://dragonegg.llvm.org>

2.2.1 编译 DragonEgg

为了编译和安装 DragonEgg，首先从这个链接获取源代码：<http://releases.llvm.org/3.3/dragonegg-3.3.src.tar.gz>。对于 Ubuntu 系统，用以下命令：

```
$ wget http://releases.llvm.org/3.3/dragonegg-3.3.src.tar.gz
$ tar xzvf dragonegg-3.3.src.tar.gz
$ cd dragonegg-3.3.src
```

如果你想探索 SVN 上最新的但不稳定的源代码，用以下命令：

```
$ svn checkout http://llvm.org/svn/llvm-project/dragonegg/trunk dragonegg
```

对于 GIT 镜像，用以下命令：

```
$ git clone http://llvm.org/git/dragonegg.git
```

为了编译和安装，你需要提供 LLVM 安装路径。LLVM 的版本必须和正在安装的 DragonEgg 的版本匹配。假设安装路径前缀是 `/usr/local/llvm`（同第 1 章，编译和安装 LLVM），假设 GCC 4.6 已安装并且你的 shell 变量 `PATH` 已包含它的路径，那么你可以使用以下命令：

```
$ GCC=gcc-4.6 LLVM_CONFIG=/usr/local/llvm/bin/llvm-config make
$ cp -a dragonegg.so /usr/local/llvm/lib
```

注意，这个项目缺失自动工具或者 CMake 项目文件。你应该直接用 `make` 命令编译它。如果你的 `gcc` 命令已经指定你想要的正确的版本，当你运行 `make` 时可以省略前缀 `GCC=gcc-4.6`。编译得到的共享库 `dragonegg.so` 正是 DragonEgg 插件，你可以用下面的 GCC 命令引用它。考虑你正在编译经典的“Hello, World!” C 代码。

```
$ gcc-4.6 -fplugin=/usr/local/llvm/lib/dragonegg.so hello.c -o hello
```

备注：虽然 DragonEgg 理论上支持 GCC 4.5 和更高的版本，但是我们强烈推荐 GCC 4.6。对于其它的 GCC 版本，DragonEgg 未作广泛的测试和维护。

2.2.2 理解 DragonEgg 和 LLVM 工具的编译流水线

如果你想看前端的运行，请用参数 `-S -fplugin-arg-dragonegg-emit-ir`，如此将输出一个人类可读的 LLVM IR 代码文件。

```
$ gcc-4.6 -fplugin=/usr/local/llvm/lib/dragonegg.so -S -fplugin-arg-dragonegg-emit-ir \
→hello.c -o hello.ll
$ cat hello.ll
```

编译器一旦将程序翻译为 IR 就停止编译，并且将这种驻留内存的表示序列化到磁盘，这种能力是 LLVM 的一种特别的特性。多数其它的编译器做不到。理解 LLVM IR 如何表示你的程序之后，你可以用若干 LLVM 工具手动地继续编译过程。下面的命令调用一个特殊的汇编器，将文本形式的 LLVM IR 转化为位码形式，还是存储在磁盘上：

```
$ llvm-as hello.ll -o hello.bc
$ file hello.bc
hello.bc: LLVM bitcode
```

如果你想，你可以用一个特殊的 IR 反汇编器（`llvm-dis`）将它转回人类可读形式。下面的工具将执行目标无关的优化，显示成功的代码转换的统计信息：

```
$ opt -stats hello.bc -o hello.bc
```

参数 `-stats` 是可选的。然后，你可以用 LLVM 后端工具将它翻译为目标机器的汇编语言：

```
$ llc -stats hello.bc -o hello.S
```

同样，参数-stats 是可选的。这是一个汇编文件，你可以用 GNU binutils 汇编器或者 LLVM 汇编器处理它。下面的命令用 LLVM 汇编器：

```
$ llvm-mc -filetype=obj hello.S -o hello.o
```

LLVM 默认使用系统链接器，因为 LLVM 链接器项目，lld，当前正在开发，还没有集成到核心 LLVM 项目中。因此，如果你没有 lld，你可以用常规的编译器驱动器完成编译，它将调用系统链接器：

```
$ gcc hello.o -o hello
```

记住，出于性能的原因，真正的 LLVM 编译器驱动器不会将任何阶段的程序表示序列化到磁盘，除了目标文件，因为它还没有集成的链接器。它利用驻留内存的表示，协调 LLVM 若干组件进行编译。

2.2.3 理解 LLVM 测试套件

LLVM 测试套件包括一套官方的程序和 benchmark，用于测试 LLVM 编译器。对于 LLVM 开发者来说，测试套件是非常有用的。开发者通过编译和运行这些程序验证优化算法和对编译器的改进。如果你正在使用一个不稳定的 LLVM 版本，或者你修改了 LLVM 的源代码，怀疑某些情况不能正常工作，这时自己运行测试套件是非常有用的。然而记住，LLVM 主目录包含简单回归测试和单元测试，你可以容易地用命令 `make check-all` 运行它们。测试套件和经典的回归测试和单元测试不同，因为它包含整个 benchmark。

你必须将 LLVM 测试套件放在 LLVM 源代码树中，让 LLVM 编译系统能够识别它。你可以从这个链接获取版本 3.4 的源代码：<http://releases.llvm.org/3.4/test-suite-3.4.src.tar.gz>。

用下面的命令下载源代码：

```
$ wget http://releases.llvm.org/3.4/test-suite-3.4.src.tar.gz
$ tar xzf test-suite-3.4.src.tar.gz
$ mv test-suite-3.4 llvm/projects/test-suite
```

如果你喜欢通过 SVN 下载最新的可能不稳定的版本，用以下命令：

```
$ cd llvm/projects
$ svn checkout http://llvm.org/svn/llvm-project/test-suite/trunk test-suite
```

如果你喜欢通过 GIT 下载，用以下命令：

```
$ cd llvm/projects
$ git clone http://llvm.org/git/llvm-project/test-suite.git
```

为了使用这个测试套件，你需要重新生成 LLVM 编译文件。这种情况有点特殊，不能使用 CMake。你必须让经典的 `configure` 脚本在测试套件目录中工作。请仿照第 1 章（编译和安装 LLVM）中描述的配置步骤。

这个测试套件有一套 Makefile，测试和检查 benchmark。你也可以提供定制的 Makefile，评估定制的程序。将定制 Makefile 放在测试套件源代码目录中，命名模板：llvm/projects/test-suite/TEST.<custom>.Makefile，其中标签 <custom> 必须替换为你选择的名字。例子：llvm/projects/test-suite/TEST.example.Makefile。

备注： 你需要重新生成 LLVM 编译文件，以使定制或修改的 Makefile 生效。

配置过程中，将会在 LLVM 目标文件目录中创建一个目录，测试套件的程序和 benchmark 将在其中运行。若要运行和测试 example Makefile，则进入第 1 章（编译和安装 LLVM）中提到的目标文件目录，执行下面的命令：

```
$ cd your-llvm-build-folder/projects/test-suite
$ make TEST=" example" report
```

2.2.4 使用 LLDB

LLDB (Low Level Debugger) 项目以 LLVM 基础设施构造一个调试器。它作为 Mac OS X 系统的 Xcode 5 调试器，正在活跃地开发和发布。由于 2011 年开发之初就被置于 Xcode 范围之外，LLDB 一直未发布一个稳定的版本，直到写作本文之时。你可以从这个链接获取 LLDB 源代码：<http://releases.llvm.org/3.4/lldb-3.4.src.tar.gz>。如同其它依赖 LLVM 的项目，将它集成到 LLVM 编译系统中，就可以轻松地编译它。这就是说，将源代码放在 LLVM tools 文件夹，如下所示：

```
$ wget http://releases.llvm.org/3.4/lldb-3.4.src.tar.gz
$ tar xvf lldb-3.4.src.tar.gz
$ mv lldb-3.4 llvm/tools/lldb
```

或者你可从 SVN 仓库获取最新版本：

```
$ cd llvm/tools
$ svn checkout http://llvm.org/svn/llvm-project/lldb/trunk lldb
```

或者如你所愿从 GIT 镜像获取：

```
$ cd llvm/tools
$ git clone http://llvm.org/git/llvm-project/lldb.git
```

备注： 在 GNU/Linux 系统上，LLDB 还在试验之中。

编译 LLDB 之前，必须先解决软件依赖：Swig，libedit（仅针对 Linux），和 Python。举例来说，在 Ubuntu 系统上，你可以用以下命令解决这些依赖：

```
$ sudo apt-get install swig libedit-dev python
```

记住，像本章介绍的其它项目一样，你需要重新生成 LLVM 编译文件，以使 LLDB 能够编译。请仿照第 1 章（编译和安装 LLVM）中描述的从源代码编译 LLVM 的步骤。

当你新安装 lldb 之后，为了简单测试，以参数 -v 运行它，打印它的版本：

```
$ lldb -v
lldb version 3.4 ( revision )
```

LLDB 调试练习

为了见识怎么使用 LLDB，我们将发起一个调试会话以分析 Clang 程序。Clang 程序包含很多 C++ 符号 (symbol)，我们可以探查它们。如果你用默认选项编译 LLVM/Clang 项目，得到的 Clang 程序就包含调试符号。所谓默认选项，就是当你运行配置脚本生成 LLVM Makefile 时省略 -enable-optimized 参数，或者当你运行 CMake 时设置 -DCMAKE_BUILD_TYPE=" Debug"，这是默认的编译类型。

如果你熟悉 GDB，你可能对一个表感兴趣，它将常用的 GDB 命令映射到相应的 LLDB 命令，见 <http://lldb.llvm.org/lldb-gdb.html>。

像 GDB 那样，我们以待调试可执行程序的路径为命令行参数启动 LLDB：

```
$ lldb where-your-llvm-is-installed/bin/clang
Current executable set to 'where-your-llvm-is-installed/bin/clang' (x86_64).
(lldb) break main
Breakpoint 1: where = clang`main + 48 at driver.cpp:293, address = 0x00000001000109e0
```

我们的命令行参数是 Clang 程序的路径，这样开始调试它。我们以参数 -v 运行程序，这应该打印 Clang 的版本：

```
(lldb) run -v
```

LLDB 停在断点之后，我们可以用 next 命令随意地单步通过每一行 C++ 代码。如同 GDB，LLDB 接受任意命令缩写，例如 n 代表 next，只要没有歧义：

```
(lldb) n
```

为了查看 LLDB 如何打印 C++ 对象，单步通过直到到达声明 argv 或 ArgAllocator 对象后的代码行，并打印它：

```
(lldb) n
(lldb) p ArgAllocator
(lldb::SpecificBumpPtrAllocator<char>) $0 = {
  Allocator = {
    SlabSize = 4096
```

(续下页)

(接上页)

```

    SizeThreshld = 4096
    DefaultSlabAllocator = (Allocator = llvm::MallocAllocator @ 0x00007f85f1497f68)
    Allocator = 0x0000007ffffbfff200
    CurSlab = 0x0000000000000000
    CurPtr = 0x0000000000000000
    End = 0x0000000000000000
    BytesAllocated = 0
}
}

```

当你玩够了之后，用 `q` 命令退出调试器：

```

(lldb) q
Quitting LLDB will kill one or more processes. Do you really want to proceed: [Y/n] y

```

2.2.5 介绍 libc++ 标准库

libc++ 库是一个为 LLVM 项目集而重写的 C++ 标准库，支持最新的 C++ 标准，包括 C++11 和 C++1y，以 MIT 许可证和 UIUC 许可证双授权方式发布。libc++ 库是 Compiler-RT 的一个重要伙伴，作为运行时库的一部分，和 libclc（OpenCL 运行时库）一起如若需要，Clang++ 用它们生成最终的可执行程序。它不同于 Compiler-RT，因为生成 libc++ 不是关键性的。Clang 不受限于 libc++，没有它时，可以让你的程序链接 GNU libstdc++。如果两个库你都有，你可以用 `-stdlib` 开关选择 Clang++ 用哪个库。libc++ 库支持 x86 和 x86_64 处理器，它是为 Mac OS X 和 GNU/Linux 系统设计的 GNU libstdc++ 的替代品。

GNU/Linux 上的 libc++ 还在开发中，不像 Mac OS X 上的 libc++ 那样稳定。

根据 libc++ 开发者，继续使用 GNU libstdc++ 的一个主要障碍是，它需要重写大部分代码以支持较新的 C++ 标准，libstdc++ 主分支的开发切换到一个 GPLv3 许可证，这是支持 LLVM 项目的一些公司所不能使用的。注意，LLVM 项目通常应用于商业产品，以一种和 GPL 哲学不相容的方式。面对这些难题，LLVM 社区决定开发一个新的 C++ 标准库，主要为 Mac OS X，也支持 Linux。

在苹果电脑上获取 libc++ 最容易的方法是安装 Xcode 4.2 或更新版本。

如果你打算为 GNU/Linux 机器自己编译这个库，记住 C++ 标准库包括它本身和一个低层库，后者实现异常处理和运行时类型信息（RTTI）的功能。这种分离的设计使得 C++ 标准库更易于移植到其它系统。这也给了你不同的选项，当你编译自己的 C++ 标准库时。你可以选择 libc++ 链接 libsupc++（GNU 实现的底层库），或者 libc++abi（LLVM 团队实现的底层库）。然而，目前 libc++abi 仅支持 Mac OS X 系统。

想要在 GNU/Linux 上用 libsupc++ 编译 libc++，首先下载如下源代码：

```

$ wget http://releases.llvm.org/3.4/libcxx-3.4.src.tar.gz
$ tar xvf libcxx-3.4.src.tar.gz
$ mv libcxx-3.4 libcxx

```

直到本文写作之时，还是不能依靠 LLVM 编译系统来编译这个库，如我们编译其它项目那样。因此注意，这次我们不将 libc++ 源代码放在 LLVM 源代码树中。

作为选择，可以从 SVN 代码仓库获取最新的试验版本：

```
$ svn co http://llvm.org/svn/llvm-project/libcxx/trunk libcxx
```

也可以使用 GIT 镜像：

```
$ git clone http://llvm.org/git/llvm-project/libcxx.git
```

一旦你用上了一个 LLVM 编译器，你需要生成 libc++ 编译文件，它们具体地调用这个新的 LLVM 编译器。在我们的例子中，假设我们的 PATH 中已存在一个可用的 LLVM 3.4 编译器。

为了使用 libsupc++，首先需要找出它的头文件安装在系统的何处。在 GNU/Linux 上，它是常规 GCC 编译器的一部分，因此可以用下面的命令寻找它们：

```
$ echo | g++ -Wp,-v -x c++ - -fsyntax-only
#include "... search starts here:
#include <...> search starts here:
/usr/include/c++/4.7.0
/usr/include/c++/4.7.0/x86_64-pc-linux-gnu
(Subsequent entries omitted)
```

通常地，前两条路径指明 libsupc++ 头文件在何处。为了验证，查看一个 libsupc++ 头文件是否存在，例如 bits/exception_ptr.h：

```
$ find /usr/include/c++/4.7.0 | grep bits/exception_ptr.h
```

然后，生成 libc++ 编译文件，用 LLVM 编译器编译它。这需要改写 shell 中 CC 和 CXX 环境变量，它们分别定义系统 C 和 C++ 编译器，改写为你想要集成 libc++ 的 LLVM 编译器。如果采用 CMake 方法用 libsupc++ 编译 libc++，需要定义 CMake 参数 LIBCXX_CXX_ABI，它指定使用哪个低层库，还有 LIBCXX_LIBSUPCXX_INCLUDE_PATHS，它指定之前找到的 libsupc++ 头文件路径列表，路径之间用分号分隔。示例如下：

```
$ mkdir where-you-want-to-build
$ cd where-you-want-to-build
$ CC=clang CXX=clang++ cmake -DLIBCXX_CXX_ABI=libstdc++ -DLIBCXX_LIBSUPCXX_INCLUDE_
→PATHS="/usr/include/c++/4.7.0;/usr/include/c++/4.7.0/x86_64-pc-linux-gnu" -DCMAKE_
→INSTALL_PREFIX="/usr" ../libcxx
```

这里，需确保../libcxx 是从当前目录到 libc++ 源代码文件夹的正确路径。运行 make 命令以编译项目。为安装命令使用 sudo，因为我们将安装这个库到/usr，让 clang++ 以后能找到它。

```
$ make && sudo make install
```


你可以对新的库和最新的 C++ 标准做个试验，当你用 `clang++` 编译一个 C++ 项目时，输入参数 `-stdlib=libc++`。

为了检验新的库在起作用，用下面的命令编译一个简单的 C++ 应用：

```
$ clang++ -stdlib=libc++ hello.cpp -o hello
```

用 `readelf` 命令分析这个 `hello` 可执行文件，确认它的确链接了新的 `libc++` 库。这个简单的实验是可行的：

```
$ readelf d hello
Dynamic section at offset 0x2f00 contains 25 entries:
  Tag                Type              Name/Value
0x00000001 (NEEDED)   Shared library: [libc++.so.1]
```

上面的代码省略了后续条目。我们清楚地看到，第 1 个 ELF 动态 section 条目明确地要求加载 `libc++.so.1` 共享库（正是我们刚刚编译的），证实了我们的 C++ 程序现在在用新的 LLVM C++ 标准库。你可以从官方项目站点获得更多信息：<http://libcxx.llvm.org>。

2.3 总结

LLVM 由若干项目组成，对主编译器驱动器来说，其中一些不是必需的，但它们是有用的工具和程序库。在本章中，我们展示了如何编译和安装这些部件。后续章节将深入探索其中的一些工具。建议读者到时再回到本章阅读编译和安装说明。

在下一章，我们将介绍 LLVM 核心库和工具的设计。

LLVM 项目由若干程序库和工具组成，它们一起构成一套宏大的编译器基础设施。细致的架构设计是连结这些组件的关键。然而，LLVM 强调这样的哲学，即一切都是程序库，这样，只有相对少量的代码不是立即可重用的，也不是特定工具的一部分。大量工具让使用者能够通过命令行以多种方式练习程序库。在本章中，我将介绍以下内容：

- LLVM 核心库设计概述
- 编译器驱动器如何工作
- 编译器驱动器之外：认识 LLVM 中间工具
- 如何写出第一个 LLVM 工具
- 浏览 LLVM 源代码的一般建议

3.1 LLVM 的基本设计原则和它的历史

LLVM 以其易于学习闻名于世，它的工具是高度组织的，让好奇的使用者能够观察到编译的很多步骤。这种设计决策可追溯到十多年前它的第 1 版，当时项目深度专注于后端算法，还依靠 GCC 将高级语言例如 C 翻译为 LLVM 中间表示（IR，Intermediate Representation）。今天，LLVM 设计的核心正是它的 IR。它使用静态单赋值形式（SSA），具有两个重要特征：

- 代码组织为三地址指令序列
- 寄存器数量无限制

然而，这并不意味着 LLVM 使用单一的程序表示方式。从整个编译过程来看，其它的中间数据结构也用于表达程序逻辑，在重要的阶段实现程序转换。从技术来说，它们也是程序的中间表示形式。举例来说，LLVM 在编译的不同阶段还采用如下附加的数据结构：

- 在 C 或 C++ 程序翻译为 LLVM IR 时，Clang 用抽象语法树（AST）结构（TranslationUnitDecl class）表示驻留内存的程序。
- 在 LLVM IR 翻译为一种机器特定的汇编语言时，LLVM 首先将程序变换为有向无环图（DAG）形式，让指令选择（SelectionDAG class）变得容易，然后将它变换回三地址指令表示，让指令调度（MachineFunction class）顺利进行。
- 为了实现汇编器和链接器，LLVM 用第 4 种中间数据结构表示目标文件上下文中的程序。

LLVM 中除了其它的程序表示形式，LLVM IR 是最重要的一种。它不仅是一种驻留内存的表示，而且可以存储在磁盘上。这是它的特质。LLVM IR 采用一种特定的编码方式存在于外部世界，这是项目初期的又一个重要决策，体现了当时人们研究程序终生优化的学术兴趣。

在这种哲学观念下，编译器超越编译时优化，去探索安装时、运行时、空闲时（当程序不在运行）的优化机会。以这种方式，优化贯穿程序的全部生命周期，由此解释了这个概念的名字。举例来说，当程序不在运行并且计算机空闲时，操作系统可以发起一个编译器守护进程，让它分析运行时收集的用户数据，为特定的使用情境重新优化程序。

注意，作为实现程序终生优化的关键，LLVM IR 能够存储到磁盘，这提供了另一种编码整个程序的方法。当整个程序以一种编译器 IR 形式存储时，一系列新的非常有效的过程间（inter-procedural）优化是可能的，它们跨越单个翻译单元或者 C 文件的边界。如此，我们也能实现强大的链接时优化。

另一方面，在程序终生优化变得可行之前，程序需要以 LLVM IR 形式发布，这并没有实现。这大概暗示着，LLVM 将成为一个平台或者虚拟机，和 Java 竞争，这也是一个巨大的挑战。例如，LLVM IR 不是目标无关的，像 Java 那样。LLVM 没有投资于强大的反馈导向的后安装时（post-installation time）优化。对于有兴趣了解这些技术挑战的细节的读者，我们建议阅读有益的 LLVMdev 邮件讨论会话：<http://lists.cs.uiuc.edu/pipermail/llvmdev/2011-October/043719.html>。

随着项目逐渐成熟，为了链接时优化的需要，编译器 IR 存储到磁盘的设计决策一直保持着，而略少关注程序终生优化的初始想法。终于，LLVM 不再掩饰无意让其核心库成为一个平台，正式宣布放弃缩略词含义——低级虚拟机，而由于历史原因保留名字 LLVM。

然而，存储到磁盘的表示形式自身应用前景广阔，包括链接时优化，若干团队正在努力将它应用到真实世界。例如，FreeBSD 社区企图将程序的 LLVM 表示嵌入到程序可执行文件中，以实现安装时或者离线微架构优化。在这种情境下，尽管程序被编译为一般的 x86 可执行文件，举例来说，对于 Intel Haswell x86 处理器，LLVM 编译器可以重新编译可执行文件中的 LLVM 表示，让它使用 Haswell 支持的新的专用指令。尽管这是一个目前正在评估的新想法，但是它展示出存储到磁盘的表示形式允许激进的方法。这种期望只是针对微架构优化的，因为在 LLVM 上实现如同 Java 的完全平台无关是不切实际的。目前仅有外部项目还在探索这个课题（见 PNaCl，Chromium's Portable Native Client）。

LLVM IR 作为一种编译器 IR，它的两个基本原则指导着核心库的开发：

- SSA 表示和无限寄存器让优化能够快速执行。
- 整个程序的 IR 存储到磁盘让链接时优化易于实现。

3.2 理解如今的 LLVM

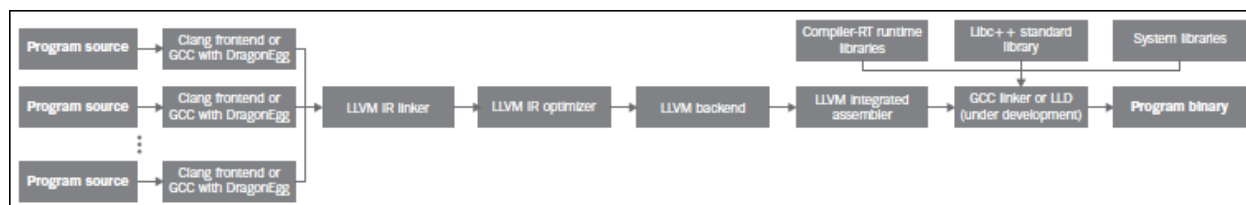
今天，LLVM 项目发展为一个巨大的编译器相关的工具集合。实际上，名字 LLVM 可能表示如下含义：

- **LLVM 项目/基础设施**：这是指若干项目的集合，一起构成一个完整的编译器：前端，后端，优化器，汇编器，链接器，libc++，compiler-rt，JIT 引擎。举例来说，“LLVM 是由若干项目组成的”，此句中的“LLVM”是这个含义。
- **基于 LLVM 的编译器**：这是指部分或者完全地由 LLVM 基础设施构建的编译器。举例来说，一个编译器可能用 LLVM 作前端和后端，但是用 GCC 和 GNU 系统库执行最终链接。举例来说，“我用 LLVM 为 MIPS 平台编译 C 程序”，此句中的“LLVM”是这个含义。
- **LLVM 程序库**：这是指 LLVM 基础设施的可重用代码部分。举例来说，“我的项目用 LLVM 的 Just-in-Time 编译框架生成代码”，此句中的“LLVM”是这个含义。
- **LLVM 核心**：中间表示优化和后端算法构成 LLVM 核心，这是项目最初的起点。“LLVM 和 Clang 是两个不同的项目”，此句中的“LLVM”是这个含义。
- **LLVM IR**：这是 LLVM 编译器的中间表示。“我开发了一个前端，将我自己的语言翻译为 LLVM”，此句中的“LLVM”是这个含义。

为了理解 LLVM 项目，你需要明白基础设施的最重要的部分：

- **前端**：这是一个编译阶段，它将计算机编程语言（例如 C，C++，Objective-C）翻译为 LLVM 编译器 IR。它包含词法分析器，语法解析器，语义分析器，和 LLVM IR 代码生成器。Clang 项目实现了全部前端相关的步骤，同时提供了插件接口和一个单独的静态分析器工具，用于深度分析。想了解详情，你可翻阅第 4 章（前端）、第 9 章（Clang 静态分析器）、第 10 章（Clang 工具和 LibTooling）。
- **IR**：LLVM IR 既有人类可读形式，又有二进制编码形式。很多工具和程序库提供了用于构建、汇编、反汇编 IR 的接口。LLVM 优化器也操作 IR，大部分优化在此发生。我们在第 5 章（LLVM 中间表示）详细解释 IR。
- **后端**：这是负责代码生成的编译阶段。它将 LLVM IR 变换为机器特定的汇编代码或目标代码二进制。寄存器分配、循环转换、窥孔优化、机器特定的优化/转换等属于后端。我们在第 6 章（后端）展开深入分析。

下图阐明了各个组件，展示了整个基础设施在一种具体应用配置下的概观。注意，我们可以组织安排各个组件，以一种不同的方式运用它们，例如，不使用 LLVM IR 链接器，假如我们不想探索链接时优化的话。



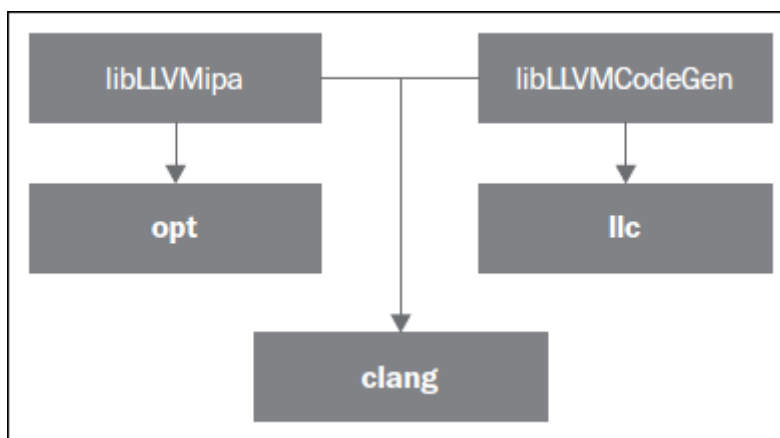
这些编译器组件之间以下面两种方式交互：

- **在内存中**：这种方式通过一个总管工具（例如 Clang）发生，每个 LLVM 组件都是它的程序库，根据分配在内存中的数据结构，将一个阶段的输出作为输入传送给下一个阶段。

- 通过文件：这种方式通过小型的独立工具发生，用户运行一个独立工具，它将一个特定组件的结果写到磁盘上的一个文件，接着用户运行下一个独立工具，它以这个文件为输入。

因此，高层工具（例如 Clang）通过链接实现小工具的功能的程序库，可以吸收多个小工具的功能。这是成立的，因为 LLVM 的设计强调最大化地重用代码，代码成为程序库。而且，独立工具实体化少量的程序库是有益的，因为这使得用户能够通过命令行直接跟一个具体的 LLVM 组件交互。

例如，考虑下面的示意图。方框中工具的名字是粗体字体，用以实现工具功能的程序库的名字是常规字体，它们的方框是分开的。在这个例子中，LLVM 后端工具，llc，用 libLLVMCodeGen 程序库实现它的部分功能，而运行纯粹 LLVM IR 层次优化器的 opt 命令，用另一个称为 libLLVMMipa 的程序库实现目标无关的过程间优化。我们还看到 clang，这个大型的工具同时用了这两个程序库，超越了 llc 和 opt，为用户提供更简单的接口。因此，任何由这样的高层工具完成的任务，都可分解成一连串低层工具，达到相同的结果。实际上，Clang 能够开展全部编译过程，而不仅仅是 opt 和 llc 的工作。这解释了为什么在静态编译下，Clang 可执行文件经常是最大的，因为它链接并运行整个 LLVM 生态系统。



3.3 跟编译器驱动器交互

编译器驱动器类似汉堡店里的店员，他跟你交互，确认你的订单，让后厨制作汉堡，然后将汉堡交付给你，或许附带可口可乐、番茄酱小袋等，如此完成你的订单。驱动器负责集成所有必需的程序库和工具，为用户提供友好的体验，让用户不必调用单体的编译器工具，例如前端、后端、汇编器、链接器等。每当你输入你的程序源代码，编译器驱动器就生成可执行文件。对于 LLVM 和 Clang 来说，编译器驱动器就是 clang 工具。

考虑简单的 C 程序，hello.c:

```
#include <stdio.h>

int main() {
    printf("Hello, World!\n");
    return 0;
}
```

为了生成这个简单程序的可执行文件，执行下面的命令：

```
$ clang hello.c -o hello
```

备注：参考第 1 章（编译和安装 LLVM）中的说明，获得一份现成的 LLVM。

对于熟悉 GCC 的人，注意前面的命令和 GCC 命令非常相似。事实上，Clang 编译器驱动器被设计成跟 GCC 的参数和命令结构相兼容，使得在很多项目中能够用 LLVM 代替 GCC。对于 Windows，Clang 有一个称为 clang-cl.exe 的版本，它模仿 Visual Studio C++ 编译器命令行接口。Clang 编译器驱动器隐式地调用所有其它的工具，从前端到链接器。

为了看清楚驱动器编译你的程序时所调用的所有后续工具，使用 `-###` 命令行参数：

```
$ clang -### hello.c -o hello
clang version 3.4 (tags/RELEASE_34/final)
Target: x86_64-apple-darwin11.4.2
Thread model: posix
"/bin/clang" -cc1 -triple x86_64-apple-macosx10.7.0 ... -main-file-name hello.c ... /
examples/hello/hello.o -x c hello.c
"/opt/local/bin/ld" ... -o hello /examples/hello/hello.o ...
```

Clang 驱动器调用的第一个工具是 clang 本身，给以参数 `-cc1`，关闭编译器驱动器模式而开启编译器模式。它还用了许多其它参数以调整 C/C++ 选项。由于 LLVM 组件都是程序库，clang-cc1 链接了 IR 生成器、目标机器代码生成器、汇编器等程序库。因此，解析源代码之后，clang-cc1 自己能够调用其它的程序库，监督在内存中进行的编译流水线，直到生成目标文件。然后，Clang 驱动器（不同于编译器 clang-cc1）调用链接器，一个外部工具，以生成可执行文件，如前面的打印输出所示。它用系统链接器完成编译，因为 LLVM 链接器，lld，还在开发之中。

注意，编译发生在内存中比发生在磁盘上快得多，这使得中间编译文件失去魅力。这解释了为什么 Clang（LLVM 前端和接收输入的第一个工具）有责任将剩余的编译工作放在内存中进行，而不是将中间结果写到文件，再由后续工具读取它。

3.4 使用独立工具

我们可以利用 LLVM 独立工具来练习之前描述的编译流程，连接不同工具的输出。尽管这种做法让编译过程变慢，由于存储中间文件到磁盘，但是这种练习让人观察编译流水线，充满乐趣，富于教诲。这也让你能够微调中间工具的输入参数。列举部分工具如下：

- `opt`：这个工具致力于在 IR 层次优化程序。输入必须是一个 LLVM bitcode（编码的 LLVM IR）文件，生成的输出文件也是这种类型。
- `llc`：这个工具通过一个具体的后端将 LLVM bitcode 变换为目标机器汇编语言文件或目标文件。你可以通过参数选择优化级别，开启调试选项，开或者关目标特定的优化。

- `llvm-mc`: 这个工具能够为多种目标格式（例如 ELF、MachO、PE）汇编指令和生成目标文件。它也能够反汇编同样的目标，打印输出等价的汇编信息和内部 LLVM 机器指令数据结构。
- `lli`: 这个工具为 LLVM IR 实现了解释器和 JIT 编译器。
- `llvm-link`: 这个工具连接聚合若干 LLVM bitcode，产生单个 LLVM bitcode，容纳所有输入。
- `llvm-as`: 这个工具将人类可读的 LLVM IR 文件，称为 LLVM 汇编，转换为 LLVM bitcode。
- `llvm-dis`: 这个工具解码 LLVM bitcode，生成 LLVM 汇编。

让我们考虑一个简单的 C 程序，由多个函数组成，它们来自多个源文件。第一个源文件是 `main.c`，代码如下：

```
#include <stdio.h>

int sum(int x, int y);

int main() {
    int r = sum(3, 4);
    printf("r = %d\n", r);
    return 0;
}
```

第二个源文件是 `sum.c`，代码如下：

```
int sum(int x, int y) {
    return x + y;
}
```

我们可以用下面的命令编译这个 C 程序：

```
$ clang main.c sum.c -o sum
```

然而，利用独立工具可以实现相同的结果。首先，用不同的参数调用 `clang`，让它为 C 源文件生成 LLVM bitcode，然后就此停止，而不是继续整个编译：

```
$ clang -emit-llvm -c main.c -o main.bc
$ clang -emit-llvm -c sum.c -o sum.bc
```

参数 `-emit-llvm` 让 `clang` 生成 LLVM bitcode 或 LLVM 汇编文件，依据输入的参数是 `-c` 还是 `-S`。在这个的例子中，参数 `-emit-llvm` 和 `-c` 让 `clang` 生成 LLVM bitcode 格式的目标文件。用组合参数 `-fllvm -c` 得到相同的结果。如果想要生成 LLVM 汇编，它是人类可读的，用下面这对命令代替：

```
$ clang -emit-llvm -S -c main.c -o main.ll
$ clang -emit-llvm -S -c sum.c -o sum.ll
```


备注：注意，不用参数`-emit-llvm`或`-flto`时，参数`-c`用目标机器语言生成目标文件，而参数`-S`生成目标汇编语言文件。这种行为是和 GCC 兼容的。

这里`.bc`和`.ll`分别是 LLVM bitcode 和汇编文件的文件扩展名。为了继续编译，我们采用下面两种方式：

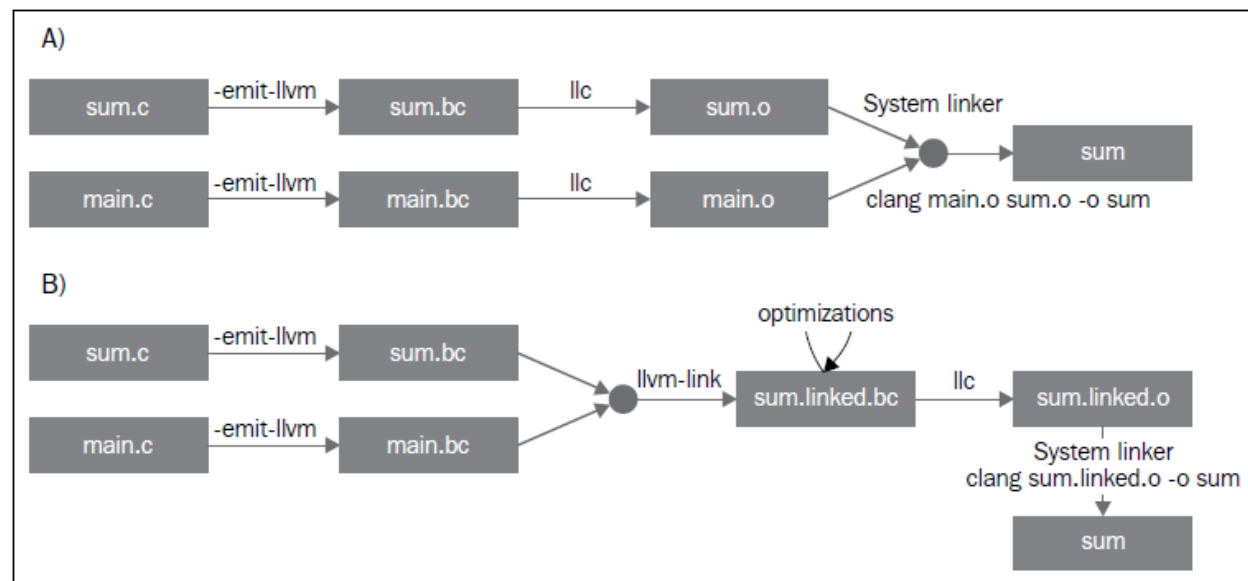
- 为每个 LLVM bitcode 生成目标特定的目标文件，用系统链接器链接它们以生成程序可执行文件（下图中的 A 部分）：

```
$ llc -filetype=obj main.bc -o main.o
$ llc -filetype=obj sum.bc -o sum.o
$ clang main.o sum.o -o sum
```

- 首先，链接这两个 LLVM bitcode 为一个最终的 LLVM bitcode。然后，为这个最终的 bitcode 生成目标特定的目标文件，通过调用系统链接器生成程序可执行文件（下图中的 B 部分）：

```
$ llvm-link main.bc sum.bc -o sum.linked.bc
$ llc -filetype=obj sum.linked.bc -o sum.linked.o
$ clang sum.linked.o -o sum
```

参数`-filetype=obj`指定输出目标文件，而不是输出目标汇编文件。我们利用 Clang 驱动器，`clang`，来调用链接器。然而这个系统链接器可以被直接调用，假如你知道你的系统链接器链接系统库所需要的所有参数。



调用后端（`llc`）之前链接 IR 文件，使得 `opt` 工具提供的链接时优化（例子见第 5 章，LLVM 中间表示）能够进一步优化最终产生的 IR。作为替代，`llc` 工具能够生成汇编输出，可以利用 `llvm-mc` 进一步汇编它。我们在第 6 章（后端）说明这种接口的详细内容。

3.5 探究 LLVM 内部设计

为了解耦编译器使之成为若干工具，LLVM 设计典型地强调，组件之间的交互发生在高层次抽象。将不同的组件隔离为单独的程序库。用 C++ 写成，使用面向对象范式，提供插件式的 Pass 接口，让它易于在整个编译流水线中集成转换和优化。

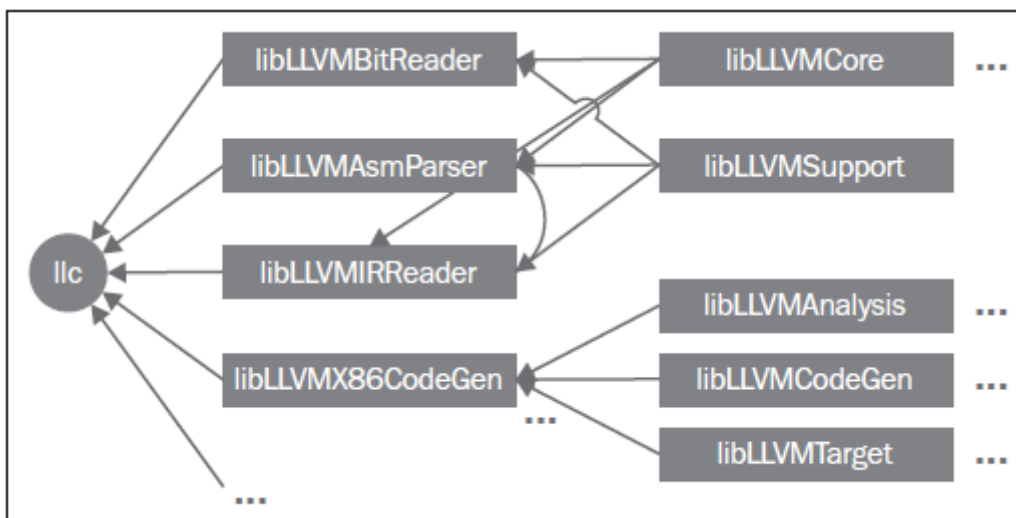
3.5.1 了解 LLVM 基础库

LLVM 和 Clang 逻辑被精心地组织成如下的程序库：

- **libLLVMCore**：包含所有 LLVM IR 相关的逻辑：IR 构造（数据布局、指令、基本块、函数）和 IR 验证。它也提供 Pass 管理器。
- **libLLVMAnalysis**：包含若干 IR 分析 Pass，例如别名分析、依赖分析、常量合并、循环信息、内存依赖分析、指令简化等。
- **libLLVMCodeGen**：实现了目标无关的代码生成和机器层次（低层版本 LLVM IR）的分析和转换。
- **libLLVMTarget**：通过通用目标抽象，提供对目标机器信息的访问。**libLLVMCodeGen** 实现了通用后端算法，目标特定的逻辑留给了即将介绍的下一个库，而高层抽象为两者提供了交流的通道。
- **libLLVMX86CodeGen**：包含 x86 目标特定的代码生成信息、转换和分析 Pass，它们组成了 x86 后端。注意，每个机器目标都有自己不同的库，例如 **LLVMARMCodeGen** 和 **LLVMMipsCodeGen**，分别实现了 ARM 和 MIPS 后端。
- **libLLVMSupport**：包含一组常用的实用工具。错误处理、整数和浮点数处理、命令行解析、调试、文件支持、字符串操作，这些是这个库实现的算法的例子，LLVM 的各个组件到处都在使用它们。
- **libclang**：它实现一套 C 接口（对比 C++，LLVM 代码的默认实现语言）以暴露 Clang 的前端功能——诊断报告、AST 遍历、代码补全、光标和源代码间映射。C 接口相当简单，使得采用其它语言（例如 Python）的项目能够容易地使用 Clang 的功能，尽管用 C 设计接口是为了更稳定，并让外部项目能够利用它。它仅覆盖内部 LLVM 组件所用的 C++ 接口的子集。
- **libclangDriver**：它包含一套 C++ 类，编译器驱动器利用它们理解类 GCC 的命令行参数，以准备编译任务，为外部工具组织适当的参数，以完成编译的不同步骤。根据目标平台，它可以采用不同的编译策略。
- **libclangAnalysis**：这是一系列 Clang 提供的前端层次分析。它的特性诸如构造 CFG 和调用图、可达代码、安全格式化字符串，等等。

作为一个例子说明如何用这些库构建 LLVM 工具，下图展示了 **llc** 工具对 **libLLVMCodeGen**、**libLLVMTarget** 和其它库的依赖，以及这些库对其它库的依赖。注意，上述列表还不完全。

此处是 LLVM 程序库概览，省略的库留到后续章节介绍。对于版本 3.0，LLVM 团队写了一份好的文档，说明所有 LLVM 库之间的依赖关系。尽管文档已经过时，它仍然是一份针对程序库组织的有趣的概述。文档页面：<http://releases.llvm.org/3.0/docs/UsingLibraries.html>



介绍 LLVM 中的 C++ 实践

LLVM 的程序库和工具用 C++ 写成，以利用面向对象的编程范式，加强各个部分之间的互操作性。另外，为了尽可能地消除低效率代码，强制实行良好的 C++ 编程实践。

多态实践

通过在基类中实现通用代码生成算法，继承和多态方法抽象了不同后端共同的任务。以这种设计，每个具体的后端可以专注于实现它的特性，通过编写少量必需的函数以覆盖超类的通用操作。libLLVMCodeGen 包含共同的算法，libLLVMTarget 包含抽象个体机器的接口。下面的代码片段（源自 llvm/lib/Target/Mips/MipsTargetMachine.h）演示了 MIPS 目标机器的描述类是如何声明为 LLVMTargetMachine 类的子类的，阐明了这种概念。此代码是 LLVMMipsCodeGen 程序库的一部分：

```
class MipsTargetMachine : public LLVMTargetMachine {
    MipsSubtarget Subtarget;
    const DataLayout DL;
    ...
}
```

为了进一步阐明这种设计理念，我们展示另一个后端的例子，在这个例子中，目标无关的寄存器分配器（它是所有后端共同的）需要知道那些寄存器是保留而不能用以分配的。此信息依赖于具体的目标，不能在通用的超类中确定。这通过调用函数 MachineRegisterInfo::getReservedRegs() 确定，每个目标必须重写这个通用方法。下面的代码片段（源自 llvm/lib/Target/Sparc/SparcRegisterInfo.cpp）演示了 SPARC 目标如何重写这个方法：

```
BitVector SparcRegisterInfo::getReservedRegs(...) const {
    BitVector Reserved(getNumRegs());
    Reserved.set(SP::G1);
    Reserved.set(SP::G2);
    ...
}
```

此代码中，通过给出一个位向量，SPARC 后端个别地选定了哪些寄存器不能用以通用寄存器分配。

介绍 LLVM 中的 C++ 模板

LLVM 常常使用 C++ 模板，尽管特别谨慎以控制编译时间，这是滥用模板的 C++ 项目的典型特征。在任何可能的时候应用模板特化，实施快速的循环利用的通用任务。作为一个 LLVM 代码中的模板例子，这里介绍一个函数，它检查作为参数传入的一个整数是否能用给定长度的位表示，这个长度是模板参数（源自 `llvm/include/llvm/Support/MathExtras.h`）：

```
template<unsigned N>
inline bool isInt(int64_t x) {
    return N >= 64 || (-(INT64_C(1)<<(N-1)) <= x && x < (INT64_C(1)<<(N-1)));
}
```

此代码中，注意模板代码如何处理任意的位长度值 N。先作一次比较，当位长度大于 64 时返回真。如果不是，计算两个表达式，它们分别是这个位长度的上界和下界，检查 x 是否在它们之间。将此代码和下面的模板特化比较，这是为常用的 8 位长度生成更快的代码的：

```
template<>
inline bool isInt<8>(int64_t x) {
    return static_cast<int8_t>(x) == x;
}
```

此代码将比较次数从三次降到两次，因而是一种正当的特化。

在 LLVM 中强制最好的 C++ 实践

编程时常常无意地产生 bug，区别在于如何控制你的 bug。LLVM 的哲学建议你在任何可能的时候使用 `libLLVMSupport` 实现的断言机制。注意，调试编译器是特别困难的，因为编译的产物是一个不同的程序。因此，如果能够更早地发现古怪的行为，在输出变得复杂之前，复杂到不容易确定它是否正确，你将节省很多时间。例如，我们来看一个 ARM 后端 Pass 的代码，它修改常量 pools 的布局，重新分配它们，遍布若干较小的 pools “岛屿”，遍布一个函数。ARM 程序常常采用这种策略加载大型常量，针对受限的 PC 相对寻址机制，因为单个大型的 pool 被置于一个距离指令太远的位置以致指令无法访问它。代码位于 `llvm/lib/Target/ARM/ARMConstantIslandPass.cpp`，下面是它的摘录：

```
const DataLayout &TD = *MF->getTarget().getDataLayout();
for (unsigned i = 0, e = CPs.size(); i != e; ++i) {
    unsigned Size = TD.getTypeAllocSize(CPs[i].getType());
    assert(Size >= 4 && "Too small constant pool entry");
    unsigned Align = CPs[i].getAlignment();
    assert(isPowerOf2_32(Align) && "Invalid alignment");
    // Verify that all constant pool entries are a multiple of their alignment.
```

(续下页)

(接上页)

```
// If not, we would have to pad them out so that instructions stay aligned.
assert((Size % Align) == 0 && "CP Entry not multiple of 4 bytes!");
```

此代码遍历一个代表 ARM 常量 pools 的数据结构，程序员期望这个对象的每个字段遵从特定的约束。注意程序员如何控制数据语义，通过使用 `assert` 调用。如果发生了不同于编写代码时的期望的事情，程序会立即停止执行，并打印失败的断言信息。程序员使用布尔表达式 `&&` “错误原因”的句式，这不会干扰 `assert` 布尔表达式的估值，反而会给出关于断言失败的简短文字解释，当这个表达式由于失败事件被打印时。使用 `assert` 会影响性能，当 LLVM 项目以 `release` 编译时，这种影响被完全消除，因为断言被关闭了。

在 LLVM 中你将频繁看到的另一种常用实践是运用智能指针。一旦变量生命期终结，智能指针自动释放内存，举例来说，LLVM 代码用它们存放目标信息和模块 (module)。过去，LLVM 提供了一种特殊的智能指针类，称为 `OwningPtr`，在 `llvm/include/llvm/ADT/OwningPtr.h` 中定义。自 LLVM3.5 起，这个类被弃用，代之以 C++11 标准提出的 `std::unique_ptr()`。

如果你有兴趣了解 LLVM 项目采用的 C++ 最好实践的完整列表，查阅 <http://llvm.org/docs/CodingStandards.html>。每个 C++ 程序员值得一读。

在 LLVM 中让字符串引用轻量化

LLVM 项目有大量的数据结构程序库，支持常见的算法，而字符串在 LLVM 程序库中占据特殊的位置。C++ 中的字符串是人们激烈讨论的对象：什么时候我们该用简单的 `char*`，相对 C++ 标准库的 `string` 类？为了在 LLVM 上下文中讨论这个问题，考虑 LLVM 程序库到处密集地使用字符串引用，以引用 LLVM 模块的名字、函数的名字、值的名字，等等。有些时候，字符串 LLVM 句柄包含 `null` 字符，致使以 `const char*` 指针传递常量字符串引用的方法是行不通的，因为 `null` 字符终结 C 类型的字符串。另一方面，使用 `const std::string&` 会频繁地引起额外的堆内存分配，因为 `string` 类需要拥有字符缓冲区。在下面的例子中我们会看到以上问题：

```
bool hasComma(const std::string &a) {
    // code
}

void myfunc() {
    char buffer[40];
    // code to create our string in our own buffer
    hasComma(buffer); // C++ compiler is forced to create a new string object,
    ↪ duplicating the buffer
    hasComma("hello, world!"); // Likewise
}
```

注意我们每次试图在自己的缓冲区创建一个字符串时，我们将付出额外的堆内存分配以复制字符串到 `string` 对象的内部缓冲区，它必须拥有自己的缓冲区。第一种情况，字符串分配在栈上；第二种情况，字符串是一个全局常量。对于这些情况，C++ 缺少一个简单的类，当我们仅仅需要引用一个字符串时，它能够避免不必要的分配。尽管我们严格地对待 `string` 对象，消除不必要的堆内存分配，引用一个字符串对象意味着两次间接访问。因为 `string` 类已经用一个内部指针存放它的数据，当我们访问实际的数据时，传递一个 `string` 对象的指针带来了双引用的额外开销。

我们可以使用一个 LLVM 类以更加高效地处理字符串引用：StringRef。这是一个轻量的类，可以如同 `const char*` 那样以值传递，但是它记录字符串的长度，允许 `null` 字符。然而，相比 `string` 对象，它不拥有缓冲区，因而从不分配堆内存空间，仅仅引用存在于外部的一个字符串。其它 C++ 项目也在探索这种思想：例如 Chromium，用 `StringPiece` 类实现了相同的想法。

LLVM 还设计了另一个字符串操作类。LLVM 提供 `Twine` 类，用以通过若干串联构建新的字符串。它延迟实际的串联，通过仅记录若干字符串的引用，它们将组成最终的产物。这是在前 C++11 时期设计的，当时字符串串联的代价高。

如果你有兴趣找出 LLVM 提供给它的编程者的其它通用类，你的书签值得收藏一份非常重要的文档，就是 LLVM 编程者手册，手册讨论所有 LLVM 通用数据结构，它们对一般编程都是有益的。此手册位于 <http://llvm.org/docs/ProgrammersManual.html>。

3.5.2 演示插件式 Pass 接口

Pass 是一种转换分析或优化。通过 LLVM API 你可轻松注册任意 Pass，在程序编译生命期的不同阶段，这是 LLVM 设计值得赞许的地方。Pass 管理器用以注册 Pass、调度 Pass、声明 Pass 之间的依赖关系。因此，不同的编译阶段到处都可获得 `PassManager` 类的实例。

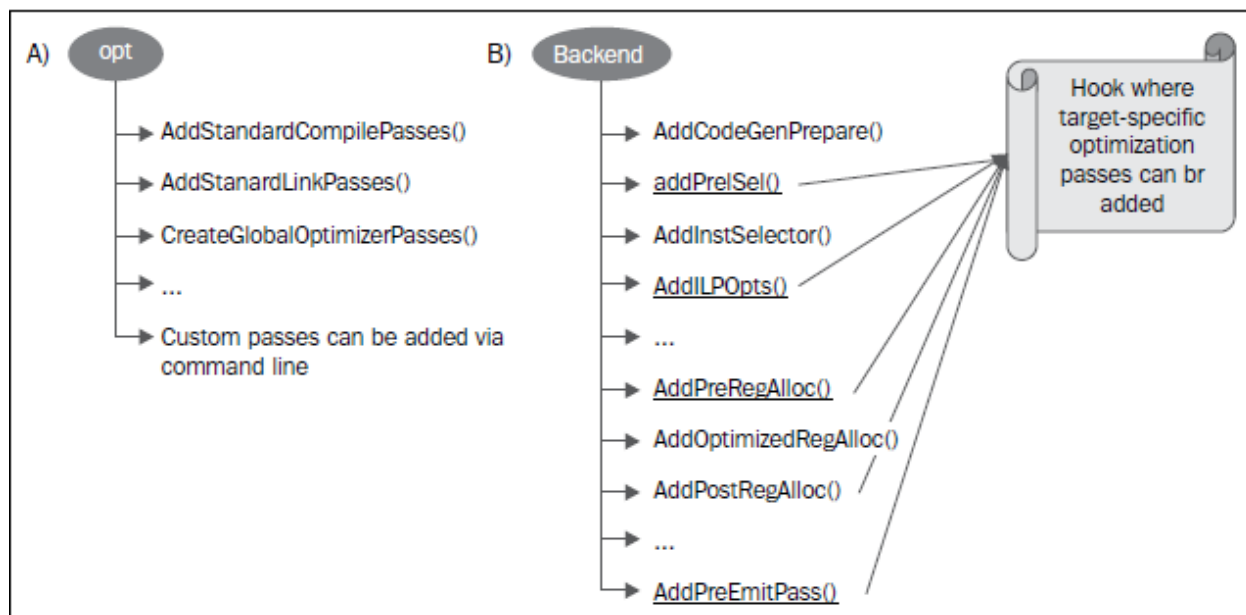
举例来说，目标可自由地在代码生成的若干位置应用定制的优化，例如寄存器分配之前之后，或者汇编输出之前。为了阐明这个问题，我们展示一个 X86 目标的例子，它有条件地在汇编输出之前注册一对定制的 Pass（源自 `lib/Target/X86/X86TargetMachine.cpp`）：

```
bool X86PassConfig::addPreEmitPass() {
    ...
    if (getOptLevel() != CodeGenOpt::None && getX86Subtarget().hasSSE2()) {
        addPass(createExecutionDependencyFixPass(&X86::VR128RegClass));
        ...
    }

    if (getOptLevel() != CodeGenOpt::None &&
        getX86Subtarget().padShortFunctions()) {
        addPass(createX86PadShortFunctions());
        ...
    }
    ...
}
```

注意后端如何根据具体的目标信息决定是否添加某个 Pass。在添加第一个 Pass 之前，X86 目标检查是否支持 SSE2 多媒体扩展。对于第二个 Pass，检查是否特别地要求填充。

下图中的 A 部分举例说明了在 `opt` 工具中优化 Pass 是如何被插入的；B 部分说明了代码生成阶段的几个目标挂钩函数，用于插入定制的目标优化。注意插入点分布于不同的代码生成阶段。当你第一次编写 Pass 需要决定在何处运行时，这张图是特别有用的。第 5 章（LLVM 中间表示）将详细描述 `PassManager` 接口。



3.6 编写第一个 LLVM 项目

在本节中，我们将演示如何用 LLVM 程序库编写你的第一个项目。在前面的章节中，我们介绍了如何用 LLVM 工具为程序产生相应的中间语言文件，即 `bitcode` 文件。现在我们将创建一个程序，它读入 `bitcode` 文件，打印程序定义的函数的名字，函数的基本块数量，展示使用 LLVM 程序库是多么容易。

3.6.1 编写 Makefile

链接 LLVM 程序库要编写很长的命令行，不借助编译系统，这是不现实的。在下面的代码中，我们展示一个 Makefile 用以完成上述工作，它基于 DragonEgg 所用的 Makefile，我们一边呈现一边解释各个部分。如果你复制粘贴此代码，会丢失 `tab` 字符；记住 Makefile 依靠 `tab` 字符指定定义规则的命令。因此，你应该手动地输入它们：

```
LLVM_CONFIG?=llvm-config

ifndef VERBOSE
QUIET:=@
endif

SRC_DIR?=$(PWD)
LD_FLAGS+=$(shell $(LLVM_CONFIG) --ldflags)
COMMON_FLAGS=-Wall -Wextra
CXX_FLAGS+=$(COMMON_FLAGS) $(shell $(LLVM_CONFIG) --cxxflags)
CPP_FLAGS+=$(shell $(LLVM_CONFIG) --cppflags) -I$(SRC_DIR)
```

这第一部分定义了若干 Makefile 变量，用作编译选项。第一个变量决定 `llvm-config` 程序的位置。对于这

种情况，它需要在你的执行路径中。llvm-config 工具是一个 LLVM 程序，打印各种各样有用的信息，用以构建需要链接 LLVM 程序库的外部项目。

举例来说，在定义用于 C++ 编译器的一系列选项时，注意我们请求 Make 运行 `llvm-config -cxxflags shell` 命令，让它打印用于编译 LLVM 项目的一系列选项。以这种方式，我们让项目源代码的编译和 LLVM 源代码兼容。最后的变量定义的一系列选项将传送给编译器预处理器。

```
HELLO=helloworld
HELLO_OBJECTS=hello.o
default: $(HELLO)

%.o : $(SRC_DIR)/%.cpp
    @echo Compiling $*.cpp
    $(QUIET)$(CXX) -c $(CPPFLAGS) $(CXXFLAGS) $<

$(HELLO) : $(HELLO_OBJECTS)
    @echo Linking $@
    $(QUIET)$(CXX) -o $@ $(CXXFLAGS) $(LDFLAGS) $^ `$(LLVM_CONFIG) --libs bitreader_
    ↪core support`
```

这第二部分定义了 Makefile 规则。第一条规则总是默认规则，我们将它关联为编译我们的 `hello-world` 可执行文件。第二条规则是通用规则，将所有 C++ 文件编译为目标文件。我们传给它预处理选项和 C++ 编译器选项。我们用 `$(QUIET)` 变量关闭全部窗口命令行输出，但是如果你想要冗长的编译日志，可以在运行 GNU Make 时定义 `VERBOSE`。

最后的规则链接所有的目标文件（此处只有一个），链接 LLVM 程序库，生成我们的项目可执行文件。这部分有链接器完成，但是有些 C++ 选项可能会起作用。因此，将 C++ 和链接器选项都传递给命令行。这由‘命令’句式完成，它指示 shell 用‘命令’的输出替代此部分。在我们的例子中，命令是 `llvm-config -libs bitreader core support`。-libs 选项要求 `llvm-config` 为我们提供链接器选项清单，用以链接所要求的 LLVM 程序库。这里，我们要求链接 `libLLVMBitReader`、`libLLVMCore`、`libLLVMSupport`。

`llvm-config` 返回的选项清单是一系列 -l 链接器参数，如 `-lLLVMCore -lLLVMSupport`。然而注意，传给链接器的参数顺序是有关系的，要求将依赖于其它库的库排在前面。例如，由于 `libLLVMCore` 使用 `libLLVMSupport` 提供的通用功能，正确的顺序是 `-lLLVMCore -lLLVMSupport`。

顺序之所以有关系，是因为程序库是目标文件的集合，当链接一个项目和一个库时，链接器只会选择能够解决已知未定义符号的那些目标文件。因此，如果在处理命令行参数中的最后一个库时，发现它引用前面已经处理过的库中的符号，大多数链接器（包括 GNU ld）都不会回过头去添加可能遗漏的目标文件，导致编译失败。

如果你不想担此责任，强制链接器迭代地访问每个库，直到包含所有需要的目标文件，你必须在程序库清单的开头和结尾使用选项 `-start-group` 和 `-end-group`，但是这会减慢链接速度。为了摆脱这种烦恼，即构建整个依赖关系图以确定链接器参数的顺序，你可以简单地调用 `llvm-config -libs`，让它为你代劳，如前面所做的那样。

Makefile 的最后一个部分定义清理规则，以删除所有编译器生成的文件，重新从头开始编译。清理规则是这样写的：


```
clean:
    $(QUIET) rm -f $(HELLO) $(HELLO_OBJECTS)
```

3.6.2 编写代码

我们将完整地给出一个 Pass 的代码。代码相对较短，因为它建立在 LLVM Pass 基础设施之上，后者为我们做了大部分工作。

```
#include "llvm/Bitcode/ReaderWriter.h"
#include "llvm/IR/Function.h"
#include "llvm/IR/Module.h"
#include "llvm/Support/CommandLine.h"
#include "llvm/Support/MemoryBuffer.h"
#include "llvm/Support/raw_os_ostream.h"
#include "llvm/Support/system_error.h"
#include <iostream>

using namespace llvm;

static cl::opt<std::string> FileName(cl::Positional, cl::desc("Bitcode file"),
    ↪cl::Required);

int main(int argc, char** argv) {
    cl::ParseCommandLineOptions(argc, argv, "LLVM hello world\n");
    LLVMContext context;
    std::string error;
    OwningPtr<MemoryBuffer> mb;
    MemoryBuffer::getFile(FileName, mb);
    Module *m = ParseBitcodeFile(mb.get(), context, &error);
    if (m==0) {
        std::cerr << "Error reading bitcode: " << error << std::endl;
        return -1;
    }
    raw_os_ostream O(std::cout);
    for (Module::const_iterator i = m->getFunctionList().begin(),
        e = m->getFunctionList().end(); i != e; ++i) {
        if (!i->isDeclaration()) {
            O << i->getName() << " has " << i->size() << " basic block(s).\n";
        }
    }
    return 0;
}
```

我们的程序利用 LLVM cl 名字空间的工具（cl 代表 command line）来实现命令行接口。调用函数 ParseC-

ommandLineOptions, 声明一个全局变量, 它的类型是 `cl::opt<std::string>`, 以此说明我们的程序接收单个参数, 它的类型是 `string`, 存放 `bitcode` 文件名。

然后, 实例化一个 `LLVMContext` 对象, 以存放一次 LLVM 编译的从属数据, 使得 LLVM 线程安全。`MemoryBuffer` 类为内存块定义了只读的接口。`parseBitcodeFile` 函数利用它读取输入文件的内容, 解析文件中的 LLVM IR。在执行错误检查确信一切完好之后, 遍历文件中模块的所有函数。LLVM 模块的概念类似于一个翻译单元, 包含 `bitcode` 文件所编码的一切内容, 作为 LLVM 层级的顶端实体, 下属若干函数, 然后基本块, 最后指令。如果函数只是一个声明, 我们忽略它, 因为我们想检验函数的定义。当我们找到函数定义时, 我们打印它们的名字, 以及所含基本块的数量。

程序编译之后, 以参数 `-help` 运行它, 看一看已经为你的程序准备好的 LLVM 命令行功能。然后, 找一个你想变换为 LLVM IR 的 C 或 C++ 文件, 变换, 再用你的程序分析。

```
$ clang -c -emit-llvm mysource.c -o mysource.bc
$ helloworld mysource.bc
```

如果你想进一步探索函数中可提取的内容, 参考 LLVM 关于 `llvm::Function` 类的 doxygen 文档: http://llvm.org/docs/doxygen/html/classllvm_1_1Function.html。作为练习, 尝试扩展这个例子, 打印每个函数的参数清单。

3.7 浏览 LLVM 源代码——一般建议

在继续学习更多的 LLVM 代码之前, 有必要理解若干观点, 主要对于开源软件世界的新手编程者。如果你在一个公司内部的闭源项目中工作, 你可能从同事编程者处得到很多帮助, 他们加入项目的时间比你早, 对很多设计决定有深入的理解, 这些起初对你来说是晦涩的。如果你遇到问题, 组件的作者可能愿意口头给你解释。口头解释的效能很高, 因为他一边解释, 一边甚至能够读懂你的面部表情, 看出你不理解的地方, 为你调整话语, 给出一种定制的解释。

然而, 远程工作的时候, 如大多数社区项目那样, 不能会面, 口头交流很少。因此, 开源社区愈加鼓励人们写作有力的文档。另一方面, 文档可能并不是大多数人通常的期望, 如一份英语写作的文档, 清楚地陈述了所有设计决定。文档很大程度上是代码本身, 在这种意义上, 为了让他人没有英语文档也能够理解代码的意图, 编程者有责任写出清晰的代码。

3.7.1 理解代码如文档

尽管 LLVM 的最重要部分都有英语文档, 在本书中我们到处引用它们, 我们最终的目标是让你准备好直接阅读代码, 因为这是深入探索 LLVM 基础设施的前提。我们向你介绍基础的概念, 这是理解 LLVM 如何工作所必需的, 有了它们, 你会发现理解 LLVM 代码所带来的乐趣, 无需阅读英语文档, 或者能够阅读很多没有任何英语文档的 LLVM 的部分。尽管这充满挑战, 当你开始实践, 你将在内心感官上深化对项目的理解, 更有信心去修改它。不知不觉地, 你将成为一个掌握 LLVM 本质高级知识的编程者, 帮助邮件列表中的其他人。

3.7.2 向社区求助

邮件列表提醒着你不是孤单的。Clang 前端的邮件列表是 cfe-dev，LLVM 核心库的邮件列表是 llvmdev。在下面的地址花点时间订阅它们：

- Clang 前端开发者列表 (<http://lists.llvm.org/mailman/listinfo/cfe-dev>)
- LLVM 核心开发者列表 (<http://lists.llvm.org/mailman/listinfo/llvm-dev>)

为项目工作的人很多，有的正在尝试着实现你也感兴趣的東西。因此，很有可能，你想问的问题，其他人已经处理过了。

在寻求帮助之前，最好演练一下你的思路，尝试独自修改代码。看看依靠自己你能走多远，尽你可能进化你的知识。当你遇到迷惑你的问题时，向邮件列表写个邮件，写清楚在寻求帮助之前你已经调查过的事情。依照这些指南，你极有可能得到针对你的问题的最好回答。

3.7.3 处理更新—SVN 日志用作文档

LLVM 项目时常地在变化，你更新 LLVM 版本之后，可能你的软件不能工作了，它与 LLVM 程序库交互。这是十分常见的处境。在你重新阅读代码以了解它如何变化之前，查看代码修改日志是有益的。

为了在实践中了解这是怎么工作的，我们来练习将 Clang 前端从 3.4 更新到 3.5。假设你为静态分析器写了一段代码，实例化一个 *BugType* 对象：

```
BugType *bugType = new BugType("This is a bug name", "This is a bug category name");
```

这个对象用于让你自己的检查器（细节见第 9 章，Clang 静态分析器）报告某种 bug。现在，让我们将整个 LLVM 和 Clang 代码树更新到版本 3.5，然后编译以上代码。我们得到以下输出：

```
error: no matching constructor for initialization of 'clang::ento::BugType' BugType_
↳ *bugType = new BugType("This is a bug name",
    ^
~~~~~
```

出现这个错误，是因为 *BugType* 构造器从一个版本变到了另一个。如果你弄不清楚怎么调整你的代码，你需要阅读修改日志，这是重要的文档，它陈述某段时间内的代码改动。幸运的是，对于采用代码修改系统的开源项目，通过查询代码修改服务器，我们可以轻松地得到作用一个具体文件的提交的消息。对于 LLVM，你甚至可以用浏览器通过 ViewVC 查询：<http://llvm.org/viewvc>。

在此例中，我们想看一看定义这个构造器方法的头文件发生什么变化。查看 LLVM 源代码树，找到文件 `include/clang/StaticAnalyzer/Core/BugReporter/BugType.h`。

备注：如果你在用文本模式的编辑器，一定用工具导航 LLVM 源代码。比如，花一点时间看看如何在你的编辑器中使用 CTAGS。你将轻易地在 LLVM 源代码树中找到定义你所感兴趣的类的每个文件。如果你固执地不想用 CTAGS 或其它工具（例如 Visual Studio 的 IntelliSense 或 Xcode），帮助你导航大型 C/C++ 项目，你总

是可以运用一个命令，例如 `grep -re "keyword" *`，在项目的根目录中执行它，以列出所有包含此关键词的文件。通过使用智能关键词，你可以轻易地找到定义文件。

为了查看作用这个具体的头文件的提交消息，我们访问 <http://llvm.org/viewvc/llvm-project/cfe/trunk/include/clang/StaticAnalyzer/Core/BugReporter/BugType.h?view=log>，它在浏览器中打印出日志。现在，我们看到一次具体的修订，发生在写作此书三个月之前，当时 LLVM 更新到 v3.5:

```
Revision 201186 - (view) (download) (annotate) - [select for diffs]
Modified Tue Feb 11 15:49:21 2014 CST (3 months, 1 week ago) by alexfh
File length: 2618 byte(s)
Diff to previous 198686

Expose the name of the checker producing each diagnostic message.

Summary:
In clang-tidy we'd like to know the name of the checker producing each
diagnostic message. PathDiagnostic has BugType and Category fields, which are
both arbitrary human-readable strings, but we need to know the exact name of the
checker in the form that can be used in the CheckersControlList option to
enable/disable the specific checker.

This patch adds the CheckName field to the CheckerBase class, and sets it in
the CheckerManager::registerChecker() method, which gets them from the
CheckerRegistry.

Checkers that implement multiple checks have to store the names of each check
in the respective registerXXXChecker method.

Reviewers: jordan_rose, krememek
Reviewed By: jordan_rose
CC: cfe-commits
Differential Revision: http://llvm-reviews.chandlerc.com/D2557
```

这个提交消息是非常透彻的，解释了修改 `BugType` 构造器的全部原因：之前用两个字符串实例化这个对象，这不足以知道哪个检查器发现了一个具体的 bug。因此，现在你必须通过传递你的检查器对象的一个实例来实例化对象，它将被存储在 `BugType` 对象中，使得易于发现哪个检查器产生了每个 bug。

现在，我们修改代码，以遵照下面更新后的接口。我们假设这段代码作为 `Checker` 类的一个成员函数的部分运行，如通常实现静态分析器检查器那样。因此，`this` 关键词应该返回一个 `Checker` 对象：

```
BugType *bugType = new BugType(this, "This is a bug name", "This is a bug category_
↪name");
```

3.7.4 结束语

当你听说 LLVM 项目文档详实的时候，不要期望找到这样的英文网页，它精确地描述了每个位和代码片段。这意味着，当你阅读代码、接口、注释、提交消息的时候，你将能够不断地深入理解 LLVM 项目，让自己知悉最新的变化。不要忘记练习修改源代码，发现它是怎么运作的，这意味着你需要你的 CTAGS 作好探索的准备。

3.8 总结

在本章中，我们从历史的视角向你介绍了 LLVM 项目采用的设计决策，概述了若干最重要的组成部分。我们还演示了如何以两种不同的方式使用 LLVM 组件。第一，通过使用编译器驱动器，它作为一种高层工具以单个命令为你执行整个编译过程。第二，通过使用单独的 LLVM 独立工具。除了存储中间结果到磁盘，这降低了编译速度，这些工具通过命令行让我们能够与 LLVM 程序库的特定部分交互，更精细地控制编译过程。这是学习 LLVM 如何工作的卓越的方法。我们还介绍了一些 LLVM 用到的 C++ 编码风格，解释了你该如何面对 LLVM 代码文档，以及如何向社区求助。

在下一章，我们将详细介绍 Clang 前端的实现和它的程序库。

编译器前端将源代码变换为编译器的中间表示，它处于代码生成之前，后者是针对具体目标的。因为编程语言有独特的语法和语义的域，所以通常来说，前端只处理一种语言或者一组类似的语言。比如 Clang，处理 C、C++、objective-C 源代码。在本章中，我们将介绍以下内容：

- 程序如何链接 Clang 程序库，如何使用 libclang
- Clang 诊断和 Clang 前端阶段
- 词法、语法、语义分析和 libclang 的例子
- 如何利用 C++ Clang 库编写一个简单的编译器驱动器

4.1 介绍 Clang

Clang 项目被认为是 C、C++、Objective-C 官方的 LLVM 前端。Clang 的官方网站是 <http://clang.llvm.org>，我们在第 1 章（编译和安装 LLVM），介绍了 Clang 的配置、编译和安装。

名字 LLVM 令人困惑，由于它含义多样。类似地，Clang 可能指代三种不同的实体：

1. 前端（由 Clang 程序库实现）。
2. 编译器驱动器（由 clang 命令和 Clang 驱动器程序库实现）。
3. 实际的编译器（由 clang -cc1 命令实现）。clang -cc1 中的编译器不单用 Clang 程序库实现，还大量地用到了 LLVM 程序库以实现编译器的中端和后端，还有集成的汇编器。

在本章中，我们重点讨论 Clang 程序库和 LLVM C 家族前端。

为了理解驱动器和编译器如何工作，我们从分析 clang 编译器驱动器的命令行开始。

```
$ clang hello.c -o hello
```

在解析命令行参数之后，Clang 驱动器通过衍生自身的另一个实例，以-cc1 选项调用内部的编译器。在编译器驱动器中使用-Xclang <option>，你可以向这个工具输入具体的参数，这不受模仿 GCC 命令行接口的约束，不像驱动器。例如，clang -cc1 工具有一个特殊的选项，可打印 Clang 抽象语法树（AST）。你可以用下面的命令结构激活这个功能：

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

你也可以直接调用 clang -cc1 而不是驱动器：

```
$ clang -cc1 -ast-dump hello.c
```

然而，记住编译器驱动器的任务之一是为调用编译器准备所有必需的参数。用选项-### 运行驱动器，可看到它用什么参数去调用 clang -cc1 编译器。例如，如果你手动地调用 clang -cc1，你也需要自己指定所有系统头文件的位置，通过-I 选项。

4.1.1 前端的活动

clang -cc1 工具的一个重要方面（困惑的源头）是，它不仅实现了编译器前端，而且利用 LLVM 程序库，构建了编译所必需的所有其它 LLVM 组件，直到 LLVM 能够处理为止。因此，它几乎实现了完整的编译器。典型地，对于 X86 目标，clang -cc1 在生成目标文件之后就中止了，因为 LLVM 链接器还在实验中，还没有集成进来。在这个时候，它将控制权让渡回驱动器，后者将调用外部工具以链接项目。选项-### 会显示 Clang 驱动器所调用的程序清单，示例如下：

```
$ clang hello.c -###
clang version 3.4
(tags/RELEASE_34/final 211335)
Target: i386-pc-linux-gnu
Thread model: posix
"clang" "-cc1" (...parameters) "hello.c" "-o" "/tmp/hello-ddafc1.o"
"/usr/bin/ld" (...parameters) "/tmp/hello-ddafc1.o" "-o" "hello"
```

我们省略了驱动器所用的完整参数清单。第一行显示 clang -cc1 从 C 源文件开始编译，直到生成目标文件。然后，第二行显示 Clang 仍然依赖系统链接器来完成编译。

从内部来说，每次 clang -cc1 调用都是由一种主要前端活动来控制的。完整的活动集合的定义在源文件 include/clang/Frontend/FrontendOptions.h 中。下表列出了一些例子，描述了 clang -cc1 可能执行的不同的任务：

活动	描述
ASTView	解析抽象语法树并用 Graphviz 显示
EmitBC	输出 LLVM bitcode .bc 文件
EmitObj	输出目标特定的.o 文件
FixIt	解析并应用所有 fixit 到源代码
PluginAction	运行一个插件活动
RunAnalysis	运行一次或多次源代码分析

选项-cc1 触发 cc1_main 函数的执行（详情参见源代码文件 tools/driver/cc1_main.cpp）。例如，当通过 clang hello.c -o hello 间接地调用-cc1 时，这个函数初始化目标特定的信息，建立诊断基础设施，执行 EmitObj 活动。这个活动由 CodeGenAction 实现，FrontendAction 的一个子类。此代码会实例化所有 Clang 和 LLVM 组件，指挥它们生成目标文件。

不同前端活动的共存，让 Clang 能够为了编译以外的目的运行编译管线，例如静态分析。还有，你可以通过-target 命令行参数为 clang 指定目标，根据这个目标，它将加载不同的 ToolChain 对象。这会改变哪些任务应该由-cc1 执行，通过执行不同的前端活动，也会改变哪些任务应该由外部工具执行，使用哪些外部工具。例如，某个目标可能使用 GNU 汇编器和 GNU 链接器以完成编译，而另一个可能使用 LLVM 集成的汇编器和 GNU 链接器。如果你不清楚 Clang 为你的目标在使用了哪些外部工具，你总是可以借助-### 选项打印驱动器命令。我们在第 8 章（交叉平台编译），对不同的目标作了更多的讨论。

4.1.2 程序库

自此以后，我们会将 Clang 视作一套程序库——它们实现了一个编译器前端，而不是驱动器和编译器应用程序。在这个意义上，Clang 是模块化设计的，由若干程序库组成。libclang (http://clang.llvm.org/doxygen/group__CINDEX.html) 是为外部 Clang 用户设计的最重要的接口，它通过 C API 提供了大量的前端功能。它包含若干 Clang 程序库，也可以单独使用它们，跟你的项目链接在一起。下面为本章列出最相关的程序库：

- libclangLex：这个库用于预处理和词法分析，处理宏、标记、pragma 构造
- libclangAST：这个库提供编译、操作、遍历抽象语法树的功能
- libclangParse：这个库用于解析程序逻辑，利用词法阶段的结果
- libclangSema：这个库用于语义分析，为 AST 验证提供动作
- libclangCodeGen：这个库利用目标特定信息进行 LLVM IR 代码生成
- libclangAnalysis：这个库包含静态分析的资源
- libclangRewrite：这个库支持代码重写，为编译代码重构工具提供基础设施（详情参见第 10 章，Clang 工具和 LibTooling）
- libclangBasic：这个库提供一系列实用工具——内存分配抽象、源代码位置、诊断，等。

使用 libclang

贯穿本章，我们将解释 Clang 前端的各个部分，通过实例介绍 libclang 的 C 接口。尽管它不是直接访问 Clang 内部类的 C++ API，使用 clang 的很大的优势就是它的稳定；由于依赖 libclang 的客户程序很多，Clang 团队设计它时考虑到了跟之前版本向后兼容。然而无论何时，你都应该可以随意地使用规则的 C++ LLVM 接口，如同在第 3 章（工具和设计）的实例中，你利用规则的 C++ LLVM 接口读取 bitcode 函数的名字。

在你的 LLVM 安装文件夹的 include 子文件夹中，查看子文件夹 clang-c，这是存放 libclang C 头文件的地方。为了运行本章中的例子，需要包含 Index.h 头文件，这是 Clang C 接口的主入口点。起初，开发者创建这个接口以帮助集成开发环境，例如 Xcode，导航 C 源代码文件，生成快速代码纠正、代码补全、索引，这是主头文件的名字 Index.h 的由来。我们将阐明如何通过 C++ 接口使用 Clang，但是要等到本章末尾。

不同于第 3 章（工具和设计）中的例子，那时我们用 llvm-config 生成待链接的 LLVM 程序库，对于 Clang 程序库我们没有这样的工具。为了链接 libclang，你可以将第 3 章（工具和设计）中的 Makefile 改为下面的代码。和前一章一样，记得手动插入 tab 字符，使得 Makefile 能够正常工作。因为这是一个为所有示例准备的通用 Makefile，注意我们用了 llvm-config -libs 选项，没有任何参数，这将返回完整的 LLVM 程序库清单。

```
LLVM_CONFIG?=llvm-config

ifndef VERBOSE
QUIET:=@
endif

SRC_DIR?=$(PWD)
LD_FLAGS+=$(shell $(LLVM_CONFIG) --ldflags)
COMMON_FLAGS=-Wall -Wextra
CXX_FLAGS+=$(COMMON_FLAGS) $(shell $(LLVM_CONFIG) --cxxflags) -fno-rtti
CPP_FLAGS+=$(shell $(LLVM_CONFIG) --cppflags) -I$(SRC_DIR)
CLANGLIBS=\
    -Wl,--start-group\
    -lclang\
    -lclangFrontend\
    -lclangDriver\
    -lclangSerialization\
    -lclangParse\
    -lclangSema\
    -lclangAnalysis\
    -lclangEdit\
    -lclangAST\
    -lclangLex\
    -lclangBasic\
    -Wl,--end-group
LLVMLIBS=$(shell $(LLVM_CONFIG) --libs)
SYSTEMLIBS=$(shell $(LLVM_CONFIG) --system-libs)
```

(续下页)

(接上页)

```
PROJECT=myproject
PROJECT_OBJECTS=project.o

default: $(PROJECT)

%.o : $(SRC_DIR)/%.cpp
    @echo Compiling $*.cpp
    $(QUIET)$(CXX) -c $(CPPFLAGS) $(CXXFLAGS) $<

$(PROJECT) : $(PROJECT_OBJECTS)
    @echo Linking $@
    $(QUIET)$(CXX) -o $@ $(CXXFLAGS) $(LDFLAGS) $^ $(CLANGLIBS) $(LLVMLIBS)
    ↪ $(SYSTEMLIBS)

clean::
    $(QUIET)rm -f $(PROJECT) $(PROJECT_OBJECTS)
```

如果你在使用动态程序库，而 LLVM 安装在一个非标准的位置，记住配置 PATH 环境变量是不够的，你的动态链接器和加载器需要知道 LLVM 共享库的位置。否则，当你运行项目程序时，它将找不到要求的共享库，如果链接了任意一个。按照以下方式配置程序库路径：

```
$ export LD_LIBRARY_PATH=$(LD_LIBRARY_PATH):/your/llvm/installation/lib
```

以你的 LLVM 安装位置的完整路径替代 /your/llvm/installation，参考第 1 章（编译和安装 LLVM）。

4.1.3 理解 Clang 诊断

诊断信息是一个编译器和用户交互的必不可少的部分。它们是编译器给用户的信息，指示错误、警告或者建议。Clang 以良好的编译诊断信息为特色，打印优美，C++ 错误消息可读性高。内部地，Clang 根据类别划分诊断信息：每个不同的前端阶段都有一个独特的类别和它自己的诊断集合。例如，在文件 `include/clang/Basic/DiagnosticParseKinds.td` 中定义了诊断信息。Clang 还根据所报告问题的严重程度分类诊断信息：NOTE, WARNING, EXTENSION, EXTWARN, ERROR。它将这些严重程度映射为 `Diagnostic::Level` 枚举。

你可以引入新的诊断机制，通过在文件 `include/clang/Basic/Diagnostic*Kinds.td` 中增加新的 TableGen 定义，编写能够检测期望条件的代码，输出相应的诊断信息。在 LLVM 源代码中所有的 .td 文件都是用 TableGen 语言编写的。

TableGen 是一个 LLVM 工具，LLVM 编译系统用它为编译器的若干部分生成 C++ 代码，以机械化的方式合成这些代码。这种想法开始于 LLVM 后端，它可以基于目标机器的描述生成大量代码，如今整个 LLVM 项目到处都在运用这种方法。源于设计，TableGen 以一种简明的方式表达信息：通过记录。例如，`DiagnosticParseKinds.td` 包含如下表达诊断信息的记录定义：

```
def err_invalid_sign_spec : Error<"'%0' cannot be signed or unsigned">;
def err_invalid_short_spec: Error<"'short %0' is invalid">;
```

在此例中，`def` 是 TableGen 关键字，定义一个新的记录。这些记录必须包含哪些字段，完全取决于将使用哪个 TableGen 后端，所生成文件的每个类型都有一个具体的后端。TableGen 总是输出 `.inc` 文件，被另一个 LLVM 源文件包含。此处，TableGen 需要生成 `DiagnosticsParseKinds.inc`，它定义宏以解释了每种诊断方法。

`err_invalid_sign_spec` 和 `err_invalid_short_spec` 是记录标识，而 `Error` 是 TableGen 的类。注意，这种语义跟 C++ 有点不同，不完全对应 C++ 实体。每个 TableGen 类，不同于 C++，是一个记录模板，定义了信息的字段，可以被其它字段继承。然而，如同 C++，TableGen 支持类的层级。

这种像模板一样的语法用于为定义指定参数，基于 `Error` 类，它接收单个字符串作为参数。所有从这个类派生的定义都是 `ERROR` 类型的诊断，类的参数编码了具体的消息，例如 “‘short %0’ is invalid”。TableGen 的语法是相当简单的，与此同时，由于在 TableGen 条目中编码的信息量很大，读者容易感到困惑。遇到疑问时请参考 <http://llvm.org/docs/TableGen/LangRef.html>。

阅读诊断

下面我们给出一个例子，用 `libclang C` 接口读取并输出所有诊断信息，这些信息是 Clang 在读一个给定的源文件时产生的。

```
extern "C" {
#include "clang-c/Index.h"
}
#include "llvm/Support/CommandLine.h"
#include <iostream>

using namespace llvm;

static cl::opt<std::string>
FileName(cl::Positional, cl::desc("Input file"),
         cl::Required);

int main(int argc, char** argv)
{
    cl::ParseCommandLineOptions(argc, argv, "Diagnostics Example\n");
    CXIndex index = clang_createIndex(0, 0);
    const char *args[] = {
        "-I/usr/include",
        "-I."
    };
    CXTranslationUnit translationUnit =
        clang_parseTranslationUnit(index, FileName.c_str(),
                                   args, 2, NULL, 0, CXTranslationUnit_None);
```

(续下页)

(接上页)

```

unsigned diagnosticCount = clang_getNumDiagnostics(translationUnit);
for (unsigned i = 0; i < diagnosticCount; ++i) {
    CXDiagnostic diagnostic = clang_getDiagnostic(translationUnit, i);
    CXString category = clang_getDiagnosticCategoryText(diagnostic);
    CXString message = clang_getDiagnosticSpelling(diagnostic);
    int severity = clang_getDiagnosticSeverity(diagnostic);
    CXSourceLocation loc = clang_getDiagnosticLocation(diagnostic);
    CXString fName;
    unsigned line = 0, col = 0;
    clang_getPresumedLocation(loc, &fName, &line, &col);
    std::cout << "Severity: " << severity << " File: "
               << clang_getCString(fName) << " Line: "
               << line << " Col: " << col << " Category: \"\"
               << clang_getCString(category) << "\" Message: \"
               << clang_getCString(message) << std::endl;

    clang_disposeString(fName);
    clang_disposeString(message);
    clang_disposeString(category);
    clang_disposeDiagnostic(diagnostic);
}
clang_disposeTranslationUnit(translationUnit);
clang_disposeIndex(index);
return 0;
}

```

在此 C++ 源文件中，包含 libclang C 头文件之前，用了 extern “C” 环境，让 C++ 编译器把这个头文件当作 C 代码编译。

我们再次使用了前一章用过的 cl 名字空间，以解析我们的程序的命令行参数。然后我们使用了 libclang 接口的若干函数 (http://clang.llvm.org/doxygen/group__CINDEX.html)。首先，通过调用 clang_createIndex() 函数创建一个索引，libclang 所用的顶层上下文结构。它接收两个整数编码的布尔值为参数：第一个为真表示我们想排除来自预编译 (PCH) 头文件的声明；第二个为真表示我们想显示诊断信息。我们把两个都设为假 (零)，因为我们想自己显示诊断信息。

接着，让 Clang 解析一个翻译单元，通过函数 clang_parseTranslationUnit() (http://clang.llvm.org/doxygen/group__CINDEX__TRANSLATION__UNIT.html)。它接收一个作为参数的待解析的源文件的名称，从全局变量 FileName 中获取它。这个变量对应一个字符串参数，我们用它启动我们的工具。还需要通过一组 (两个) 参数指定 include 文件的位置—请自由地调整这些参数以适应你的系统。

备注：实现我们自己的 Clang 工具的困难之处，在于缺少驱动器的参数猜测能力，它提供充足的参数以在你的系统上处理源文件。举例来说，如果你在开发 Clang 插件，你不会有这样的忧虑。为了解决这个问题，你可以使用编译器命令数据库，它给出恰当的参数集，用以处理你想要分析的每个输入源文件，见第 10 章 (Clang 工具和 LibTooling) 中的讨论。这种情况，我们可以用 CMake 生成数据库。不过，在我们的例子中，我们自

已提供这些参数。

信息经过解析并存储在 `CXTranslationUnit` C 数据结构之后，我们实现了一个循环，遍历 Clang 产生的所有诊断，并把它们输出到屏幕。为此，首先利用 `clang_getNumDiagnostics()` 获取解析这个文件时产生的诊断数量，并决定循环的界限（参见 http://clang.llvm.org/doxygen/group__CINDEX__DIAG.html）。然后，对于每次循环遍历，利用 `clang_getDiagnostic()` 获取当前的诊断，利用 `clang_getDiagnosticCategoryText()` 获取描述这个诊断类型的字符串，利用 `clang_getDiagnosticSpelling()` 获取显示给用户的消息，利用 `clang_getDiagnosticLocation()` 获取它所发生的准确代码位置。我们还利用 `clang_getDiagnosticSeverity()` 获取代表此诊断的严重程度的枚举数字（NOTE，WARNING，EXTENSION，EXTWARN，或 ERROR），但是为简单起见，我们将它变换为无符号数，并当作数字打印它。

因为这种 C 接口缺少 C++ `string` 类，当处理字符串时，这些函数经常返回一个特殊的 `CXString` 对象，这需要你调用 `clang_getCString()` 得到内部的 `char` 指针以打印它，之后调用 `clang_disposeString()` 以删除它。

记住，你的输入源文件可能包含了其它文件，这要求诊断引擎还要记录文件名，除了行号和列号。文件、行号、列号三元属性组让你能够定位所引用的代码的位置。一个特殊的对象，`CXSourceLocation`，代表这个三元组。为了将它翻译为文件名、行号、列号，必须调用 `clang_getPresumedLocation()` 函数，输入作为引用的 `CXString` 和 `int` 参数，它们会被相应地填写。

完成之后，我们通过函数 `clang_disposeDiagnostic()`、`clang_disposeTranslationUnit()`、`clang_disposeIndex()` 删除各个对象。

让我们用如下的 `hello.c` 文件测试一下：

```
int main() {  
    printf("hello, world!\n")  
}
```

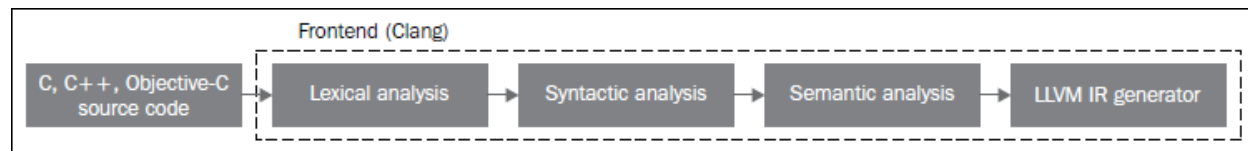
这个 C 源文件有两个错误：缺少包含正确的头文件，漏写一个分号。编译我们的项目，然后运行它，看看 Clang 将给出怎样的诊断：

```
$ make  
$ ./myproject hello.c  
Severity: 2 File: hello.c Line: 2 Col: 9 Category: "Semantic Issue" Message:↵  
↵implicitly declaring library function 'printf' with type 'int (const char *, ...)'  
Severity: 3 File: hello.c Line: 2 Col: 24 Category: "Parse Issue" Message: expected ';'↵  
↵' after expression
```

我们看到，这两个诊断由前端的不同阶段产生，语义和（语法）解析阶段。我们将在下一节探索它们。

4.2 通过 Clang 学习前端的步骤

为了将源代码程序转换为 LLVM IR `bitcode`，源代码必须经历几个中间步骤。下图阐明了这些中间步骤，它们是这一节的主题。



4.2.1 词法分析

前端的第一个步骤处理源代码的文本输入，将语言结构分解为一组单词和标记，去除注释、空白、制表符等。每个单词或者标记必须属于语言子集，语言的保留字被变换为编译器内部表示。文件 `include/clang/Basic/TokenKinds.def` 定义了保留字。例如，在下面的 `TokenKinds.def` 摘要中，两个已知的 C/C++ 标记，保留字 `while` 和符号 `<`，它们的定义被高亮了。

```

TOK(identifier)           // abcde123
// C++11 String Literals.
TOK(utf32_string_literal) // U"foo"
...
PUNCTUATOR(r_paren,      ") ")
PUNCTUATOR(l_brace,      "{ ")
PUNCTUATOR(r_brace,      "} ")
PUNCTUATOR(starequal,    "*=")
PUNCTUATOR(plus,         "+ ")
PUNCTUATOR(plusplus,     "++ ")
PUNCTUATOR(arrow,        "-> ")
PUNCTUATOR(minusminus,   "-- ")
PUNCTUATOR(less,         "< ")
...
KEYWORD(float,            , KEYALL)
KEYWORD(goto,              , KEYALL)
KEYWORD(inline,            , KEYC99 | KEYCXX | KEYGNU)
KEYWORD(int,               , KEYALL)
KEYWORD(return,            , KEYALL)
KEYWORD(short,             , KEYALL)
KEYWORD(while,             , KEYALL)
  
```

这个文件中的定义被纳入 `tok` 名字空间。这样，每当编译器需要在词法处理之后检查保留字是否出现，可以通过这个名字空间访问它们。例如，可以通过枚举元素 `tok::l_brace`、`tok::less`、`tok::kw_goto`、`tok::kw_while` 访问 `{`、`<`、`goto`、`while` 结构。

考虑下面的 `min.c` 的 C 代码：

```
int min(int a, int b) {  
    if (a < b)  
        return a;  
    return b;  
}
```

每个标记都包含一个 `SourceLocation` 类的实例，用以记录它在程序源代码中的位置。记住，你曾经使用了它的 C 对应物 `CXSourceLocation`，但是两者引用相同的数据。通过下面的 `clang -cc1` 命令行，我们依靠词法分析输出标记和它们的 `SourceLocation` 结果：

```
$ clang -cc1 -dump-tokens min.c
```

例如，高亮的 `if` 语句的输出是：

```
if 'if' [StartOfLine] [LeadingSpace] Loc=<min.c:2:3>  
l_paren '(' [LeadingSpace] Loc=<min.c:2:6>  
identifier 'a' Loc=<min.c:2:7>  
less '<' [LeadingSpace] Loc=<min.c:2:9>  
identifier 'b' [LeadingSpace] Loc=<min.c:2:11>  
r_paren ')' Loc=<min.c:2:12>  
return 'return' [StartOfLine] [LeadingSpace] Loc=<min.c:3:5>  
identifier 'a' [LeadingSpace] Loc=<min.c:3:12>  
semi ';' Loc=<min.c:3:13>
```

注意每个语言结构都以它的类型为前缀：) 是 `r_paren`，< 是 `less`，未匹配保留字的字符串是 `identifier`，等。

练习词法错误

考虑源代码 `lex.c`：

```
int a = 08000;
```

此代码中的错误源于八进制常数的错误拼写：一个八进制常数不能含有大于 7 的数字。这触发一个词法错误，如下所示：

```
$ clang -c lex.c  
lex.c:1:10: error: invalid digit '8' in octal constant  
int a = 08000;  
        ^  
1 error generated.
```

下面，我们以这个例子运行诊断小节中制作的程序：

```
$ ./myproject lex.c
Severity: 3 File: lex.c Line: 1 Col: 10 Category: "Lexical or Preprocessor Issue"
↳Message: invalid digit '8' in octal constant
```

我们看到，我们的项目程序识别出它有词法问题，正如我们的预期。

利用词法器编写 libclang 代码

这里演示一个运用 libclang 的例子，它利用 LLVM 词法器标记化（tokenize）一个源文件的前 60 个字符流：

```
extern "C" {
#include "clang-c/Index.h"
}
#include "llvm/Support/CommandLine.h"
#include <iostream>

using namespace llvm;

static cl::opt<std::string>
FileName(cl::Positional, cl::desc("Input file"),
        cl::Required);

int main(int argc, char** argv)
{
    cl::ParseCommandLineOptions(argc, argv, "My tokenizer\n");
    CXIndex index = clang_createIndex(0, 0);
    const char *args[] = {
        "-I/usr/include",
        "-I."
    };
    CXTranslationUnit translationUnit =
        clang_parseTranslationUnit(index, FileName.c_str(),
                                   args, 2, NULL, 0, CXTranslationUnit_None);
    CXFile file = clang_getFile(translationUnit, FileName.c_str());
    CXSourceLocation loc_start =
        clang_getLocationForOffset(translationUnit, file, 0);
    CXSourceLocation loc_end =
        clang_getLocationForOffset(translationUnit, file, 60);
    CXSourceRange range = clang_getRange(loc_start, loc_end);
    unsigned numTokens = 0;
    CXToken *tokens = NULL;
    clang_tokenize(translationUnit, range, &tokens, &numTokens);
    for (unsigned i = 0; i < numTokens; ++i) {
```

(续下页)

```

enum CXTokenKind kind = clang_getTokenKind(tokens[i]);
CXString name = clang_getTokenSpelling(translationUnit, tokens[i]);
switch (kind) {
case CXToken_Punctuation:
    std::cout << "PUNCTUATION(" << clang_getCString(name) << ") ";
    break;
case CXToken_Keyword:
    std::cout << "KEYWORD(" << clang_getCString(name) << ") ";
    break;
case CXToken_Identifier:
    std::cout << "IDENTIFIER(" << clang_getCString(name) << ") ";
    break;
case CXToken_Literal:
    std::cout << "COMMENT(" << clang_getCString(name) << ") ";
    break;
default:
    std::cout << "UNKNOWN(" << clang_getCString(name) << ") ";
    break;
}
clang_disposeString(name);
}
std::cout << std::endl;
clang_disposeTokens(translationUnit, tokens, numTokens);
clang_disposeTranslationUnit(translationUnit);
return 0;
}

```

为了简单起见，开头用了相同的样板代码初始化命令行参数，调用前面例子见过的 `clang_createIndex()/clang_parseTranslationUnit()`。变化出现在后面。相对查询诊断，我们为 `clang_tokenize()` 准备参数，它运行 Clang 词法器，为我们返回标记流。为此，我们必须建立一个 `CXSourceRange` 对象，指定我们想运行词法器的源代码范围（起点和终点）。这个对象由两个 `CXSourceLocation` 对象组成，一个指定起点，另一个指定终点。它们由 `clang_getLocationForOffset()` 得到，这个函数返回一个 `CXSourceLocation`，指定 `CXFile` 中的一个偏移，而 `CXFile` 由 `clang_getFile()` 获得。

为了用两个 `CXSourceLocation` 建立 `CXSourceRange`，我们调用 `clang_getRange()` 函数。有了它，我们准备好了调用 `clang_tokenize()` 函数，以引用方式输入两个重要的参数：`CXToken` 指针，将存储标记流；`unsigned` 类型指针，将返回流的标记的数目。根据这个数目，我们建立一个循环遍历所有的标记。

对于每个标记，利用 `clang_getTokenKind()` 得到它的类型，还利用 `clang_getTokenSpelling()` 得到相应的代码。然后用一个 `switch` 结构，根据标记的类型打印不同的文本，还有对应这个标记的代码。在下面的示例中，我们会看到结果。

以下的代码输入这个项目程序：


```
#include <stdio.h>
int main() {
    printf("hello, world!");
}
```

运行我们的标记化程序后，得到下面的输出：

```
PUNCTUATION(#) IDENTIFIER(include) PUNCTUATION(<) IDENTIFIER(stdio) PUNCTUATION(.)
↪ IDENTIFIER(h) PUNCTUATION(>) KEYWORD(int) IDENTIFIER(main) PUNCTUATION(( )
↪ PUNCTUATION()) PUNCTUATION({) IDENTIFIER(printf) PUNCTUATION(( ) COMMENT("hello,
↪ world!") PUNCTUATION()) PUNCTUATION(;) PUNCTUATION({})
```

预处理

C/C++ 预处理器在语义分析之前运行，负责展开宏，包含文件，根据各种以 # 开头的预处理器指示（preprocessor directive）略去部分代码。预处理器和词法器紧密关联，两者持续地相互交互。由于预处理器在前端的早期工作，在语义分析器尝试从代码中提取任何表意之前，你可以利用宏做奇怪的事情，例如利用宏展开改变函数的声明。注意这让我们能够激进地改变语言的语法。如果你喜欢，你甚至可以这样编码：

这是 Adrian Cable 的代码，22 届 International Obfuscated C Code Contest (IOCCC) 的一位获奖者，这个竞赛允许我们依照 Create Commons Attribute-Share Alike 3.0 许可证重现参赛者的源代码，供我们消遣。这是一个 8086 模拟器。如果你想学习如何格式化此代码，阅读第 10 章（Clang 工具和 LibTooling）中的 ClangFormat 小节。为了展开宏，你也可以用 -E 选项运行编译器驱动器，这将只运行预处理器，然后中断编译，不作进一步分析。

预处理器允许我们将源代码转换为难以理解的文本片段，这警示我们该适度地使用宏。这是良好的建议，题外话。词法器预处理标记流，处理预处理指示，例如宏和 pragma。预处理器用一个符号表存放定义的宏，当一个宏实例出现时，用存储在符号表中的标记替代当前的宏实例。

如果你安装了扩展工具（第 2 章，外部项目），你可以在命令行运行 pp-trace。这个工具揭露预处理器的活动。

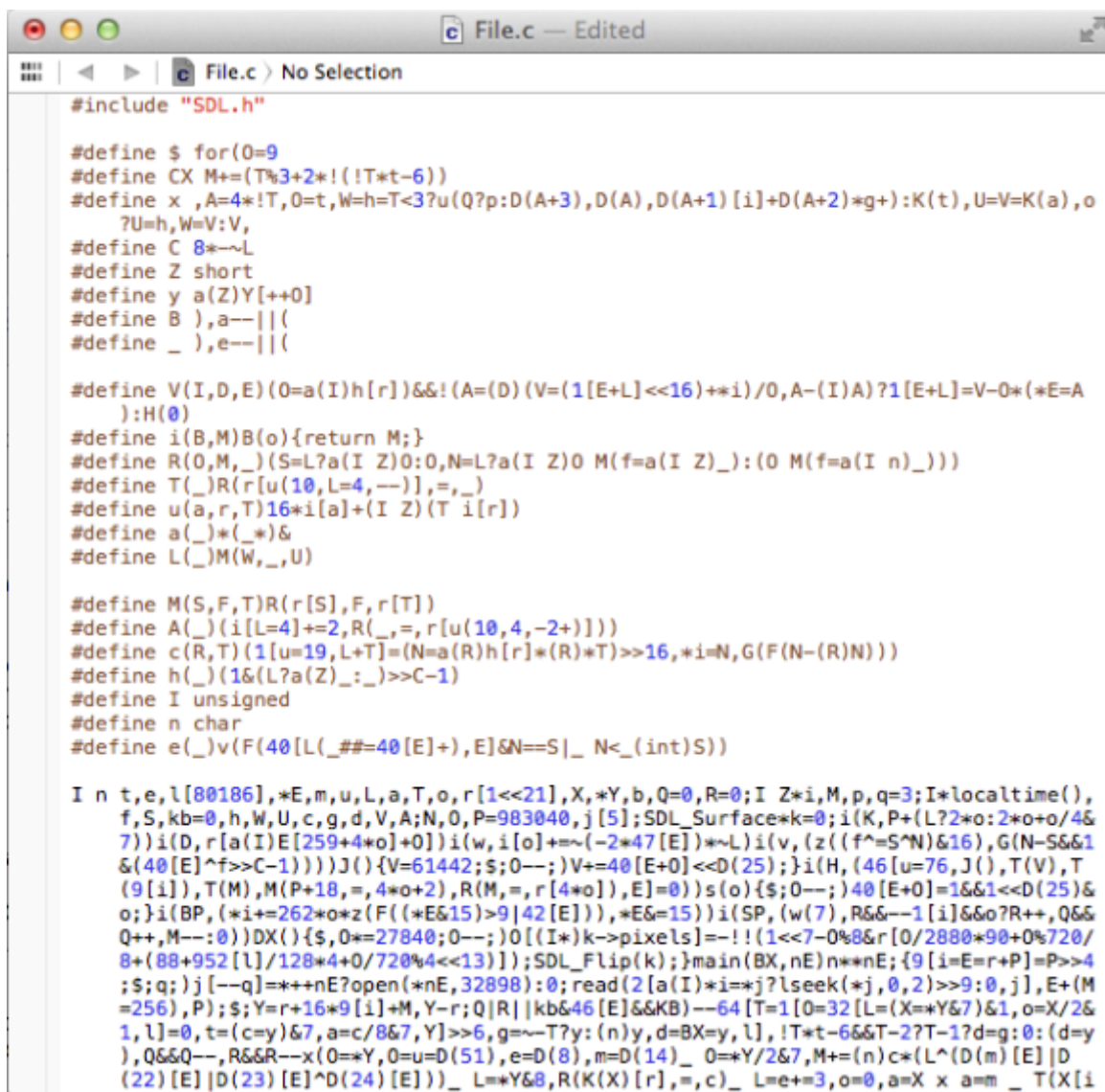
考虑下面的例子 pp.c：

```
#define EXIT_SUCCESS 0
int main() {
    return EXIT_SUCCESS;
}
```

如果我们用 -E 选项运行编译器驱动器，我们会看到下面的输出：

```
$ clang -E pp.c -o pp2.c && cat pp2.c
...
int main() {
```

(续下页)



```

#include "SDL.h"

#define $ for(0=9
#define CX M+=(T%3+2*!(T*t-6))
#define x ,A=4*!T,0=t,W=h=T<3?u(Q?p:D(A+3),D(A),D(A+1)[i]+D(A+2)*g+):K(t),U=V=K(a),o
?U=h,W=V:V,
#define C 8*~L
#define Z short
#define y a(Z)Y[++0]
#define B ),a--||(
#define _ ),e--||(

#define V(I,D,E)(0=a(I)h[r])&&!(A=(D)(V=(1[E+L]<<16)+i)/0,A-(I)A)?1[E+L]=V-0*(~E=A
):H(0)
#define i(B,M)B(o){return M;}
#define R(0,M,_)(S=L?a(I Z)0:0,N=L?a(I Z)0 M(f=a(I Z)_):(0 M(f=a(I n)_)))
#define T(_)R(r[u(10,L=4,--)],_,_)
#define u(a,r,T)16*i[a]+(I Z)(T i[r])
#define a(_)*(*)&
#define L(_)(M(W,_,U)

#define M(S,F,T)R(r[S],F,r[T])
#define A(_)(i[L=4]+2,R(_),r[u(10,4,-2+)]))
#define c(R,T)(1[u=19,L+T]=(N=a(R)h[r]*(R)*T)>>16,*i=N,G(F(N-(R)N)))
#define h(_)(1&(L?a(Z)_:_)>>C-1)
#define I unsigned
#define n char
#define e(_)(v(F(40[L(_##=40[E+)],E)&N==S|_ N<_(int)S))

I n t,e,l[80186],*E,m,u,L,a,T,o,r[1<<21],X,*Y,b,Q=0,R=0;I Z*i,M,p,q=3;I*localtime(),
f,S,kb=0,h,W,U,c,g,d,V,A;N,O,P=983040,j[5];SDL_Surface*k=0;i(K,P+(L?2*o:2*o+o/4&
7))i(D,r[a(I)E[259+4*o]+0])i(w,i[o]+~(-2*47[E])*~L)i(v,(z((f^=S^N)&16),G(N-S&&1
&(40[E]^f>>C-1)))J(J){V=61442;s;0--;)V+=40[E+0]<<D(25);}i(H,(46[u=76,J(),T(V),T
(9[i]),T(M),M(P+18,=,4*o+2),R(M,=, r[4*o]),E=0))s(o){s;0--;)40[E+0]=1&&1<<D(25)&
o;}i(BP,(*i+=262*o*z(F((~E&15)>9|42[E])),*E&=15))i(SP,(w(7),R&&--1[i]&o?R++,Q&&
Q++,M--:0))DX(){s,0*=27840;0--;)O[(I*)k->pixels]=~!(1<<7-0%8&r[0/2880*90+0%720/
8+(88+952[l]/128*4+0/720%4<<13]));SDL_Flip(k);}main(BX,nE)n**nE;{9[i=E=r+P]=P>>4
;s;q;}j[--q]=**+nE?open(*nE,32898):0;read(2[a(I)*i=j]?lseek(*j,0,2)>>9:0,j),E+(M
=256),P);s;Y=r+16*9[i]+M,Y-r;q|R|kb&46[E]&&KB)--64[T=1[0=32[L=(X=*Y&7)&1,o=X/2&
1,l]=0,t=(c=y)&7,a=c/8&7,Y]>>6,g=~T?y:(n)y,d=BX=y,l,!T*t-6&&T-2?T-1?d=g:0:(d=y
),Q&&Q--,R&&R--x(0=*Y,0=u=D(51),e=D(8),m=D(14)_ 0=*Y/2&7,M+=(n)c*(L^(D(m)[E]|D
(22)[E]|D(23)[E]^D(24)[E]))_ L=*Y&8,R(K(X)[r],=,c)_ L=e+=3,o=0,a=X x a=m _ T(X[i

```

(接上页)

```
return 0;
}
```

如果我们运行 `pp-trace` 工具，我们会看到下面的输出：

```
$ pp-trace pp.c
...
- Callback: MacroDefined
  MacroNameTok: EXIT_SUCCESS
  MacroDirective: MD_Define
- Callback: MacroExpands
  MacroNameTok: EXIT_SUCCESS
  MacroDirective: MD_Define
  Range: ["/examples/pp.c:3:10", "/examples/pp.c:3:10"]
  Args: (null)
- Callback: EndOfFile
```

我们省略了在开始预处理实际的文件之前 `pp-trace` 输出的很长的内建宏的列表。实际上，这个列表是非常有用的，如果你想知道你的编译器驱动器在编译源代码时默认定义了什么宏。`pp-trace` 的实现方法，是重写预处理器回调函数，这意味着，你可以在你的工具中实现一个功能函数，每当预处理器要采取动作的时候就执行这个函数。此例中，有两次动作：读取 `EXIT_SUCCESS` 宏定义，之后在第 3 行展开它。`pp-trace` 工具还打印你的工具接收的参数，如果你实现了 `MacroDefined` 回调函数。这个工具相当小，如果你想实现预处理器回调函数，阅读它的源代码是一个好的开始。

4.2.2 语法分析

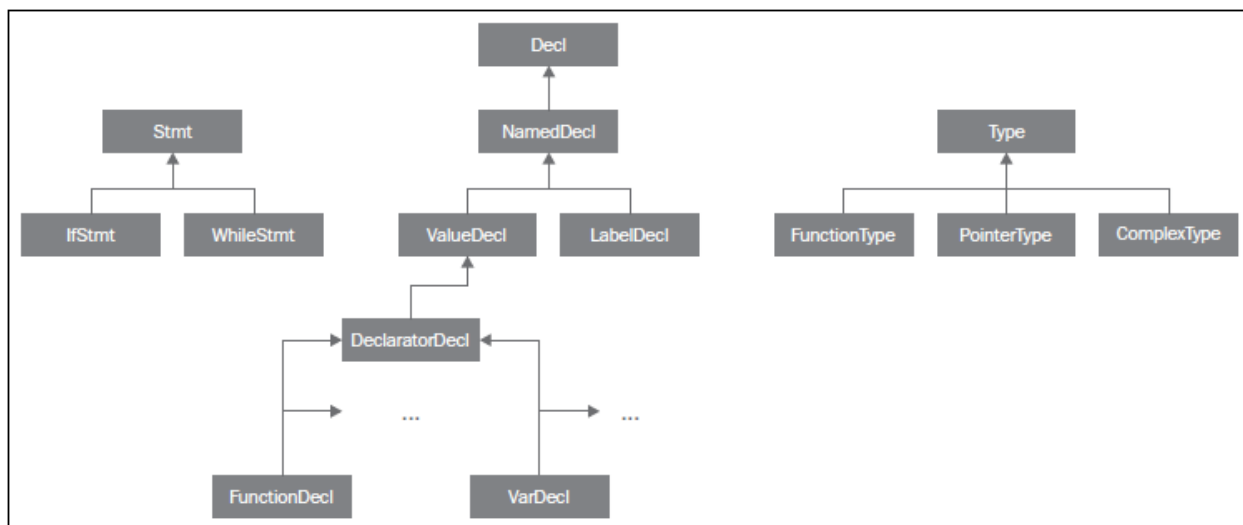
在词法分析标记化源代码之后，语法分析发生了，它分组标记以形成表达式、语句、函数体等。它检查一组标记是否有意义，考虑它们的物理布局，但是还未分析代码的意思，就像英语中的语法分析，不关心你说了什么，只考虑句子是否正确。这种分析也称为解析，它接收标记流作为输入，输出语法树（AST）。

理解 Clang AST 节点

一个 AST 节点表示声明、语句、类型。因此，有三个表示 AST 的核心类：`Decl`、`Stmt`、`Type`。在 Clang 中，每个 C 或 C++ 语言结构都表示为一个 C++ 类，它们必须继承上述核心类之一。下图说明了其部分分类级。例如，`IfStmt` 类（表示一个完整的 if 语句体）直接继承 `Stmt` 类。另一方面，`FunctionDecl` 和 `VarDecl`——用以存放函数和变量的声明或定义——继承多个类，并且只是间接继承 `Decl`。

为了查看完整的示意图，请浏览每个类的 `doxygen` 页面。例如，对于 `Stmt`，访问 http://clang.llvm.org/doxygen/classclang_1_1Stmt.html；点击子类，发现它们的直接派生类。

顶层的 AST 节点是 `TranslationUnitDecl`。它是所有其它 AST 节点的根，代表整个翻译单元。以 `min.c` 源代码为例子，记住我们可以用 `-ast-dump` 开关输出它的 AST：



```

$ clang -fsyntax-only -Xclang -ast-dump min.c
TranslationUnitDecl ...
|-TypedefDecl ... __int128_t '__int128'
|-TypedefDecl ... __uint128_t 'unsigned __int128'
|-TypedefDecl ... __builtin_va_list '__va_list_tag [1]'
  \-FunctionDecl ... <min.c:1:1,
  ↳ line:5:1> min 'int (int, int)'
    |-ParmVarDecl ... <line:1:7, col:11> a 'int'
    |-ParmVarDecl ... <col:14, col:18> b 'int'
    \-CompoundStmt ... <col:21, line:5:1>
  ...

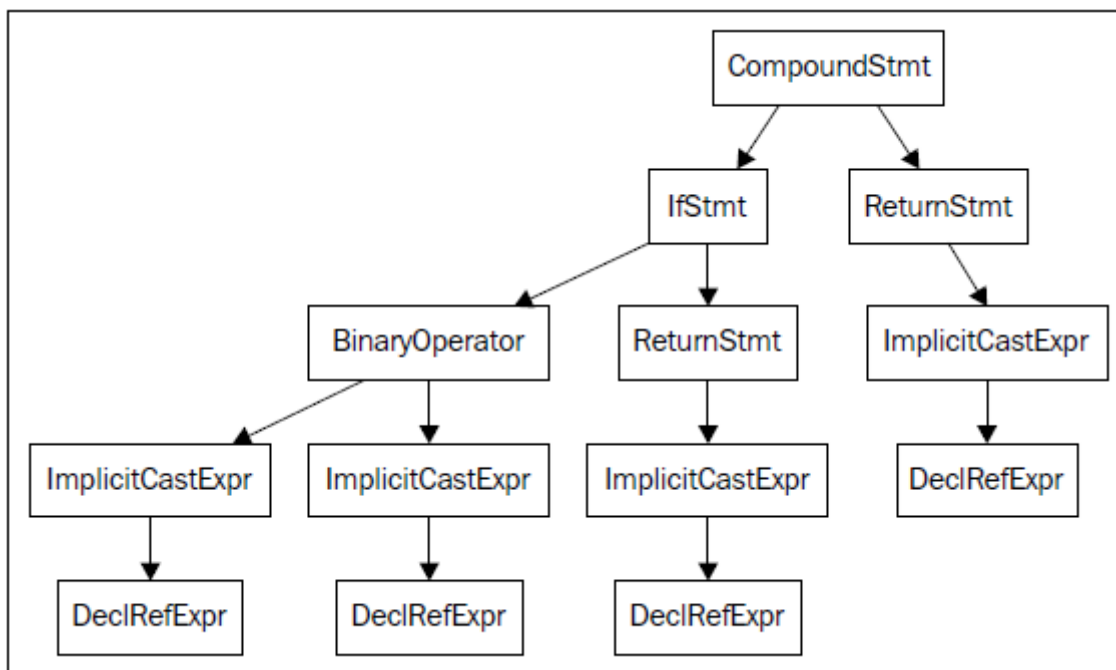
```

注意出现了顶层翻译单元的声明，`TranslationUnitDecl`，和 `min` 函数的声明，`FunctionDecl`。`CompoundStmt` 声明包含了其它的语句和表达式。下图是 AST 的图形视图，可用下面的命令得到：

```
$ clang -fsyntax-only -Xclang -ast-view min.c
```

AST 节点 `CompoundStmt` 包含 `if` 和 `return` 语句，`IfStmt` 和 `ReturnStmt`。每次对 `a` 和 `b` 的使用都生成一个到 `int` 类型的 `ImplicitCastExpr`，如 C 标准的要求。

`ASTContext` 类包含翻译单元的完整 AST。利用 `ASTContext::getTranslationUnitDecl()` 接口，从顶层 `TranslationUnitDecl` 实例开始，我们可以访问任何一个 AST 节点。



通过调试器理解解析器动作

解析器接收并处理在词法阶段生成的标记序列，每当发现一组要求的标记在一起的时候，生成一个 AST 节点。例如，每当发现一个标记 tok::kw_if，就调用 ParseIfStatement 函数，处理 if 语句体中的所有标记，为它们生成所有必需的孩子 AST 节点，以及一个 IfStmt 根节点。请看下面的代码，它来自文件 lib/Parse/ParseStmt.cpp (212 行)：

```

...
case tok::kw_if:                // C99 6.8.4.1: if-statement
    return ParseIfStatement(TrailingElseLoc);
case tok::kw_switch:            // C99 6.8.4.2: switch-statement
    return ParseSwitchStatement(TrailingElseLoc);
...

```

通过在调试器中输出调用堆栈，我们可以更好地理解 Clang 编译 min.c 时怎样调用 ParseIfStatement 函数：

```

$ gdb clang
$ b ParseStmt.cpp:213
$ r -cc1 -fsyntax-only min.c
...
213     return ParseIfStatement(TrailingElseLoc);
(gdb) backtrace
#0 clang::Parser::ParseStatementOrDeclarationAfterAttributes
#1 clang::Parser::ParseStatementOrDeclaration
#2 clang::Parser::ParseCompoundStatementBody

```

(续下页)

(接上页)

```
#3 clang::Parser::ParseFunctionStatementBody
#4 clang::Parser::ParseFunctionDefinition
#5 clang::Parser::ParseDeclGroup
#6 clang::Parser::ParseDeclOrFunctionDefInternal
#7 clang::Parser::ParseDeclarationOrFunctionDefinition
#8 clang::Parser::ParseExternalDeclaration
#9 clang::Parser::ParseTopLevelDecl
#10 clang::ParseAST
#11 clang::ASTFrontendAction::ExecuteAction
#12 clang::FrontendAction::Execute
#13 clang::CompilerInstance::ExecuteAction
#14 clang::ExecuteCompilerInvocation
#15 ccl_main
#16 main
```

ParseAST() 函数解析一个翻译单元，先利用 Parser::ParseTopLevelDecl() 读取顶层声明。然后，它处理所有后续 AST 节点，接收关联的标记，把每个新的 AST 节点附着到它的 AST 父节点。只有当解析器接收了所有标记，才会返回到 ParseAST()。接着，解析器的用户可以从顶层 TranslationUnitDecl 访问各个 AST 节点。

练习解析错误

考虑下面 parse.c 中的 for 语句：

```
void func() {
    int n;
    for (n = 0 n < 10; n++);
}
```

此代码中的错误是 n = 0 之后漏掉一个分号。下面是 Clang 编译它时输出的诊断信息：

```
$ clang -c parse.c
parse.c:3:14: error: expected ';' in 'for' statement specifier
    for (n = 0 n < 10; n++);
           ^
1 error generated.
```

下面运行我们的诊断程序：

```
$ ./myproject parse.c
Severity: 3 File: parse.c Line: 3 Col: 14 Category: "Parse Issue" Message: expected ';'
↪ ' in 'for' statement specifier
```

这个例子中的所有标记都是正确的，因此词法器成功地结束了，没有产生诊断信息。然而，在构建 AST 时，将若干标记组合在一起，看看它们是否有意义，解析器注意到 for 结构漏掉一个分号。在这种情况下，我

们的诊断将它归类为解析问题 (Parse Issue)。

写代码遍历 Clang AST

libclang 接口让你能够通过一个节点指针对象遍历 Clang AST，它指向当前 AST 的一个节点。你可以利用 clang_getTranslationUnitCursor() 函数得到顶层节点指针。在下面的例子中，我将编写一个工具，它输出一个 C 或 C++ 源文件中包含的所有 C 函数或 C++ 方法：

```
extern "C" {
#include "clang-c/Index.h"
}
#include "llvm/Support/CommandLine.h"
#include <iostream>

using namespace llvm;

static cl::opt<std::string>
FileName(cl::Positional, cl::desc("Input file"),
        cl::Required);

enum CXChildVisitResult visitNode (CXCursor cursor, CXCursor parent,
                                   CXClientData client_data) {
    if (clang_getCursorKind(cursor) == CXCursor_CXXMethod ||
        clang_getCursorKind(cursor) == CXCursor_FunctionDecl) {
        CXString name = clang_getCursorSpelling(cursor);
        CXSourceLocation loc = clang_getCursorLocation (cursor);
        CXString fName;
        unsigned line = 0, col = 0;
        clang_getPresumedLocation(loc, &fName, &line, &col);
        std::cout << clang_getCString(fName) << ":"
                  << line << ":" << col << " declares "
                  << clang_getCString(name) << std::endl;
        clang_disposeString(fName);
        clang_disposeString(name);
        return CXChildVisit_Continue;
    }
    return CXChildVisit_Recurse;
}

int main(int argc, char** argv)
{
    cl::ParseCommandLineOptions(argc, argv, "AST Traversal Example\n");
    CXIndex index = clang_createIndex(0, 0);
    const char *args[] = {
```

(续下页)


```

    "-I/usr/include",
    "-I."
};
CXTranslationUnit translationUnit =
    clang_parseTranslationUnit(index, FileName.c_str(),
                              args, 2, NULL, 0, CXTranslationUnit_None);
CXCursor cur = clang_getTranslationUnitCursor(translationUnit);
clang_visitChildren(cur, visitNode, NULL);
clang_disposeTranslationUnit(translationUnit);
clang_disposeIndex(index);
return 0;
}

```

此例中最重要的函数是 `clang_visitChildren()`，它的输入参数是节点指针，它递归地访问节点指针的所有子节点，每次访问调用一个回调函数。代码开始处，我们定义这个回调函数，命名为 `visitNode()`。这个函数必须返回枚举 `CXChildVisitResult` 的一个成员值，它仅有三种可能：

- 返回 `CXChildVisit_Recurse`，当我们期望 `clang_visitChildren()` 继续遍历 AST，访问当前节点的子节点；
- 返回 `CXChildVisit_Continue`，当我们期望它继续访问，但是略过当前节点的子节点；
- 返回 `CXChildVisit_Break`，当我们已经满足，期望 `clang_visitChildren()` 不再访问更多的节点。

我们的回调函数接收三个参数：代表我们当前正在访问的 AST 节点的 `cursor`；代表这个节点的父节点的另一个 `cursor`；以及一个 `CXClientData` 对象，它是 `void` 指针的 `typedef`。这个指针让你能够在不同回调函数调用之间传递任意的数据结构，其中包含你想维护的状态。假如你想建立一种分析，它是有用的。

备注：虽然可以用此代码结构建立分析，但是，如果你感到你的分析相当复杂，需要一种像控制流图（CFG）这样的结构，就不要用 `cursor` 或 `libclang`——将你的分析实现为一个 Clang 插件更合适，它直接调用 Clang C++ API 用 AST 创建 CFG（参见 <http://clang.llvm.org/docs/ClangPlugins.html> 和 `CFG::buildCFG` 方法）。通常来说，直接根据 AST 建立分析比利用 CFG 建立分析来得更加困难。你还应该看一看第 9 章（Clang 静态分析器），它解释如何建立强大的 Clang 静态分析。

在前面的例子中，我们忽略了 `client_data` 和 `parent` 参数。我们简单地利用 `clang_getCursorKind()` 函数检测当前 `cursor` 是否指向一个 C 函数声明（`CXCursor_FunctionDecl`）或者 C++ 方法（`CXCursor_CXXMethod`）。当我们确信正在访问正确的 `cursor` 时，我们会利用两个函数从 `cursor` 提取信息：用 `clang_getCursorSpelling()` 得到这个 AST 节点对应的代码，用 `clang_getCursorLocation()` 得到和它关联的 `CXSourceLocation` 对象。接着，打印这些信息——所用的方式和之前诊断项目所用的相似，并返回 `CXChildVisit_Continue` 以结束函数。我们之所以用这个选项，是因为我们确信不存在嵌套的函数声明，继续遍历访问这个 `cursor` 的子节点是没有意义的。

如果 `cursor` 不是我们所期望的，我们就简单地继续 AST 递归遍历，通过返回 `CXChildVisit_Recurse`。

实现了回调函数 `visitNode` 后，剩余的代码是相当简单的。我们用最初的样板代码解析命令行参数和输入文件。接着，我们以顶层 `cursor` 和我们的回调函数调用 `visitChildren()`。最后一个参数是用户数据，我们不用

它，设为 NULL。

我们对下面的输入文件运行这个程序：

```
#include <stdio.h>
int main() {
    printf("hello, world!");
}
```

输出如下：

```
$ ./myproject hello.c
hello.c:2:5 declares main
```

这个项目还打印了大量的信息，指出 `stdio.h` 头文件中声明函数的每一行，但是为简单起见我们在此处省略了它们。

以预编译头文件序列化 AST

我们可以序列化 Clang AST，将它保存到一个 PCH 扩展文件中。这个特性避免了每次一个项目的源文件包含相同的头文件时重复处理它们，加快了编译速度。当选择使用 PCH 文件时，所有头文件都被预编译成单个 PCH 文件，在编译一个翻译单元时，编译器快捷地从预编译的头文件得到信息。

举例来说，想要为 C 生成 PCH 文件，你应该使用跟 GCC 一样的语法，即用 `-x c-header` 选项开启预编译头文件生成，如下所示：

```
$ clang -x c-header myheader.h -o myheader.h.pch
```

想要使用你的 PCH 文件，你应该应用 `-include` 选项，如下：

```
$ clang -include myheader.h myproject.c -o myproject
```

4.2.3 语义分析

语义分析借助一个符号表检验代码没有违背语言类型系统。这个表存储标识符（符号）和它们各自的类型之间的映射，以及其它内容。类型检查的一种直觉的方法是，在解析之后，遍历 AST 的同时从符号表收集关于类型的信息。

与众不同的是，Clang 并不在解析之后遍历 AST。相反，它在 AST 节点生成过程中即时检查类型。让我们回过头去再看一看解析 `min.c` 的例子。此例中，`ParseIfStatement` 函数调用语义动作 `ActOnIfStmt`，为 if 语句作语义检查，输出相应的诊断。在 `lib/Parse/ParseStmt.cpp` 中，第 1082 行，我们观察到控制转移，以进行语义分析。

```
...
    return Actions.ActOnIfStmt(IfLoc, FullCondExp, ...);
...
```

为了协助语义分析，DeclContext 基类为每个作用域包含各个 Decl 节点的引用，从第一个到最后一个。这让语义分析变得轻松，因为语义分析引擎可以通过查看从 DeclContext 派生的 AST 节点找到符号的声明，以查找名字引用的符号并检查符号类型，检查符号是否实际存在。这样的 AST 节点的例子有 TranslationUnitDecl、FunctionDecl、LabelDecl。

以 min.c 为例，你可以用 Clang 输出声明上下文，如下所示：

```
$ clang -fsyntax-only -Xclang -print-decl-contexts min.c
[translation unit] 0x7faf320288f0
    <typedef> __int128_t
    <typedef> __uint128_t
    <typedef> __builtin_va_list
    [function] f(a, b)
        <parameter> a
        <parameter> b
```

注意，结果中只有 TranslationUnitDecl 和 FunctionDecl 之内的声明，因为只有它们是派生于 DeclContext 的节点。

练习语义错误

下面的 sema.c 文件包含两个用到标识符 a 的定义：

```
int a[4];
int a[5];
```

这里的错误源于两个不同的变量用了相同的名字，它们的类型不同。这个错误必须在语义分析时被发现，相应地 Clang 报告了这个问题：

```
$ clang -c sema.c
sema.c:3:5: error: redefinition of 'a' with a different type
int a[5];
    ^
sema.c:2:5: note: previous definition is here
int a[4];
    ^
1 error generated.
```

如果运行我们的诊断程序，会得到以下输出：

```
$ ./myproject sema.c
Severity: 3 File: sema.c Line: 2 Col:5 Category: "Semantic Issue" Message:↵
↵redefinition of 'a' with a different type: 'int [5]' vs 'int [4]'
```

4.2.4 生成 LLVM IR 代码

经过解析和语义分析的联合处理之后，ParseAST 函数调用 HandleTranslationUnit 方法以触发有意接收最终的 AST 的客户。如果编译器驱动器运用 CodeGenAction 前端动作，这个用户就是 BackendConsumer，它将遍历 AST，生成 LLVM IR，实现完全相同的语法树所表示的程序行为。到 LLVM IR 翻译工作，从顶层声明 TranslationUnitDecl 开始。

我们继续考察例子 min.c，if 语句通过函数 EmitIfStmt 被变换为 LLVM IR，在文件 lib/CodeGen/CGStmt.cpp 中，第 130 行。利用调试器的函数堆栈，我们可以看到从 ParseAST 函数到 EmitIfStmt 调用路径：

```
$ gdb clang
(gdb) b CGStmt.cpp:130
(gdb) r -cc1 -emit-obj min.c
...
130 case Stmt::IfStmtClass: EmitIfStmt (cast<IfStmt>(*S)); break;
(gdb) backtrace
#0 clang::CodeGen::CodeGenFunction::EmitStmt
#1 clang::CodeGen::CodeGenFunction::EmitCompoundStmtWithoutScope
#2 clang::CodeGen::CodeGenFunction::EmitFunctionBody
#3 clang::CodeGen::CodeGenFunction::GenerateCode
#4 clang::CodeGen::CodeGenModule::EmitGlobalFunctionDefinition
#5 clang::CodeGen::CodeGenModule::EmitGlobalDefinition
#6 clang::CodeGen::CodeGenModule::EmitGlobal
#7 clang::CodeGen::CodeGenModule::EmitTopLevelDecl
#8 (anonymous namespace)::CodeGeneratorImpl::HandleTopLevelDecl
#9 clang::BackendConsumer::HandleTopLevelDecl
#10 clang::ParseAST
```

当代码被翻译为 LLVM IR 时，我们的前端之旅结束了。如果我们继续常规的流水线，接下来，LLVM IR 程序库会优化 LLVM IR 代码，后端会生成目标代码。如果你想为自己的语言实现一个前端，Kaleidoscope 前端教程值得一读，它是极好的教程，见 <http://llvm.org/docs/tutorial>。在下一节中，我们将呈现如何编写一个简化的 Clang 驱动器，它将利用以上旅程中讨论了的一样的前端阶段。

4.3 组合在一起

在下面的例子中，我们将借机介绍 Clang C++ 接口，而不再依赖 libclang C 接口。我们将创建一个程序，它将利用内部的 Clang C++ 类，对输入文件运用词法器、解析器、语义分析；这样，我们将有机会做一个简单 FrontendAction 对象的工作。你可以继续使用我们在本章开始时给出的 Makefile。然而，你可能希望关闭 -Wall -Wextra 编译器选项，因为它将产生大量关于 Clang 头文件未使用参数的警告。

下面是这个例子的源代码：

```
#include "llvm/ADT/IntrusiveRefCntPtr.h"
#include "llvm/Support/CommandLine.h"
#include "llvm/Support/Host.h"
#include "clang/AST/ASTContext.h"
#include "clang/AST/ASTConsumer.h"
#include "clang/Basic/Diagnostic.h"
#include "clang/Basic/DiagnosticOptions.h"
#include "clang/Basic/FileManager.h"
#include "clang/Basic/SourceManager.h"
#include "clang/Basic/LangOptions.h"
#include "clang/Basic/TargetInfo.h"
#include "clang/Basic/TargetOptions.h"
#include "clang/Frontend/ASTConsumers.h"
#include "clang/Frontend/CompilerInstance.h"
#include "clang/Frontend/TextDiagnosticPrinter.h"
#include "clang/Lex/Preprocessor.h"
#include "clang/Parse/Parser.h"
#include "clang/Parse/ParseAST.h"
#include <iostream>

using namespace llvm;
using namespace clang;

static cl::opt<std::string>
FileName(cl::Positional, cl::desc("Input file"), cl::Required);

int main(int argc, char **argv) {
    cl::ParseCommandLineOptions(argc, argv, "My simple front end\n");
    CompilerInstance CI;
    DiagnosticOptions diagnosticOptions;
    CI.createDiagnostics();

    IntrusiveRefCntPtr<TargetOptions> PTO(new TargetOptions());
    PTO->Triple = sys::getDefaultTargetTriple();
    TargetInfo *PTI = TargetInfo::CreateTargetInfo(CI.getDiagnostics(),
```

(续下页)

(接上页)

```

PTO);

CI.setTarget (PTI);

CI.createFileManager();
CI.createSourceManager(CI.getFileManager());
CI.createPreprocessor(TU_Complete);
CI.getPreprocessorOpts().UsePredefines = false;
ASTConsumer *astConsumer = CreateASTPrinter(NULL, "");
CI.setASTConsumer(astConsumer);

CI.createASTContext();
CI.createSema(TU_Complete, NULL);
const FileEntry *pFile = CI.getFileManager().getFile(FileName);
if (!pFile) {
    std::cerr << "File not found: " << FileName << std::endl;
    return 1;
}
CI.getSourceManager().createMainFileID(pFile);
CI.getDiagnosticClient().BeginSourceFile(CI.getLangOpts(), 0);
ParseAST(CI.getSema());
// Print AST statistics
CI.getASTContext().PrintStats();
CI.getASTContext().Idents.PrintStats();

return 0;
}

```

以上代码对输入的源文件运行词法器、解析器、语义分析，输入文件可从命令行指定。它打印解析的源代码和 AST 统计，然后结束运行。此代码执行了以下步骤：

1. `CompilerInstance` 类管理整个编译过程基础设施（参见 http://clang.llvm.org/doxygen/classclang_1_1CompilerInstance.html）。第一步实例化这个类，存为 `CI`。
2. 通常，`clang -cc1` 会实例化一个具体的 `FrontendAction`，它执行这里介绍的所有步骤。因为我们想向你暴露这些步骤，所有不使用 `FrontendAction`；作为替代，我们将自己配置 `CompilerInstance`。我们利用一个 `CompilerInstance` 方法创建诊断引擎，从系统获得一个目标三元组，作为当前的目标。
3. 现在我们实例化三个新的资源：一个文件管理器，一个源代码管理器，一个预处理器。第一个是读源文件所必需的，第二个负责管理 `SourceLocation` 实例，为词法器和解析器所用。
4. 我们创建一个 `ASTConsumer` 引用，传给 `CI`。这让前端客户能够以其自己的方式接收最终的 AST（在解析和语义分析之后）。例如，如果我们想让驱动器生成 LLVM IR 代码，就需要提供一个具体的代码生成 `ASTConsumer` 实例（称为 `BackendConsumer`），这正好是 `CodeGenAction` 设置它的 `CompilerInstance` 的 `ASTConsumer` 的方式。在此例中，我们包含了头文件 `ASTConsumers.h`，它为我们提供了各式各样的用作实验 consumer（接收者），而我们用了一个仅仅打印 AST 到控制台的 consumer。我们借助 `CreateAST-`

Printer() 调用创建了它。如果你感兴趣，就花点时间实现你自己的 ASTConsumer 子类，执行任何你感兴趣的前端分析（从阅读 lib/Frontend/ASTConsumers.cpp 开始，它有一些实现的例子）。

5. 我们创建一个新的 ASTContext 和 Sema，分别为解析器和语义分析器所用，把它们推送给 CI 对象。我们还初始化了诊断 consumer（这里，我们的标准 consumer 也仅仅打印诊断到屏幕）。
6. 我们调用 ParseAST 以执行词法和语法分析，它们会接着借助 HandleTranslationUnit 函数调用，去调用我们的 ASTConsumer。Clang 也会打印诊断并中断流水线，如果在任何前端阶段发现一个严重的错误。
7. 打印 AST 统计信息到标准输出。

让我们用下面的文件测试这个简单的前端工具：

```
int main() {  
    char *msg = "Hello, world!\n";  
    write(1, msg, 14);  
    return 0;  
}
```

产生的输出如下：

```
$ ./myproject test.c  
int main() {  
    char *msg = "Hello, world!\n";  
    write(1, msg, 14);  
    return 0;  
}  
*** AST Context Stats:  
39 types total.  
31 Builtin types  
3 Complex types  
3 Pointer types  
1 ConstantArray types  
1 FunctionNoProto types  
Total bytes = 544  
0/0 implicit default constructors created  
0/0 implicit copy constructors created  
0/0 implicit copy assignment operators created  
0/0 implicit destructors created  
  
Number of memory regions: 1  
Bytes used: 1594  
Bytes allocated: 4096  
Bytes wastes: 2502 (includes alignment, etc)
```

4.4 总结

在本章中，我们介绍了 Clang 前端。我们解释了 Clang 前端程序库、编译器驱动器、和 clang -cc1 工具中的实际编译器之间的区别。我们还讨论了诊断，演示了一个 libclang 的小程序，输出诊断信息。接着，我们开启前端之旅，学习了所有的步骤：词法器、解析器、语义分析和代码生成，展示了 Clang 如何实现这些步骤。最后，我们以一个例子结束了本章，这个例子演示了怎么编写一个简单的编译器驱动器，激活所有的前端阶段。如果你有意阅读关于 AST 更多的资料，这里有一份好的社区文档：<http://clang.llvm.org/docs/IntroductionToTheClangAST.html>。如果你有意阅读关于 Clang 设计的更多的资料，在你阅读实际的源代码之前，你应该阅读 <http://clang.llvm.org/docs/InternalsManual.html>。

在下一章，我们将进入编译流水线的下一步：LLVM 中间表示。

第 5 章 LLVM 中间表示

LLVM 中间表示 (IR) 是连接前端和后端的中枢, 让 LLVM 能够解析多种源语言, 为多种目标生成代码。前端产生 IR, 而后端接收 IR。IR 也是大部分 LLVM 目标无关的优化发生的地方。在本章中, 我们将介绍以下内容:

- LLVM IR 的特性
- LLVM IR 语言的语法
- 怎样写一个生成 LLVM IR 的工具
- LLVM IR Pass 的结构
- 怎样写你自己的 IR Pass

5.1 概述

对于编译器 IR 的选择是非常重要的决定。它决定了优化器能够得到多少信息用以优化代码使之运行得更快。一方面, 非常高层的 IR 让优化器能够轻松地提炼出原始源代码的意图。另一方面, 低层的 IR 让编译器能够更容易地生成为特定硬件优化的代码。对目标机器知道得越多, 发掘机器特性的机会就越多。此外, 低层的工作必须小心对待。当编译器将程序翻译为一种更接近机器指令的表示时, 映射程序片段到原始源代码会变得愈发困难。更进一步, 如果编译器设计夸张地使用一种这样的表示, 它非常接近地表示了一种具体的目标机器, 那么为其它具有不同结构的机器生成代码会变得很棘手。

这种设计权衡导致了编译器之间不同的选择。例如, 有的编译器不支持多种目标的代码生成, 而是专注一种机器架构。这让他们能够使用专门的 IR, 贯穿整个流水线, 针对单一的架构, 让编译器生成高效代码。Intel C++ 编译器 (icc) 就是这种例子。然而, 编写编译器为单一架构生成代码, 这是一种昂贵的方案, 如果

你打算支持多种目标。在这种情况下，为每种架构写一个不同的编译器是不现实的，最好设计出一个编译器，它能够为多种目标机器生成代码——这是如 GCC 和 LLVM 这样的编译器的使命。

对于可变目标的编译器（retargetable compiler），它的项目显著地面临着更多的挑战，需要协调多个目标的代码生成。最小化构建一个多目标编译器的关键，在于使用一种通用的 IR，它让不同的后端以相同的方式领会源代码程序，并将它翻译为相异的机器指令集。使用通用的 IR，可以在多种后端之间共用一系列目标无关的优化算法，但是这要求设计者提升通用 IR 的层级（level），让它不过度表示某一个机器。因为编译器在较高的层级工作无法运用目标特定的技巧，一个优秀的可变目标编译器也采用其它 IR 在不同的更低的层级执行优化。

LLVM 项目开始于一种比 Java 字节码更低层级的 IR，因此，初始的首字母缩略词是 Low Level Virtual Machine。它的想法是发掘低层优化的机会，采用链接时优化。将 IR 作为字节码写到磁盘，这让链接时优化成为可能。字节码让用户能够在同一个文件中混合多个模块，然后运用过程间优化。这样，优化在多个编译单元发生，就像它们在同一模块一样。

在第 3 章（工具和设计）中，我们解释了如今 LLVM 既不是 Java 的竞争者，也不是一种虚拟机，它使用其它的中间表示以生成高效代码。例如，除了作为通用 IR 的 LLVM IR——这是执行目标无关优化的地方，当程序被表示为 MachineFunction 和 MachineInstr 类之后，每个后端可能执行目标相关的优化。这些类利用目标机器指令表示程序。

另一方面，Function 和 Instruction 类显然是最重要的类，因为它们表示了通用 IR，为多种目标所共享。这种中间表示主要是目标无关的（但不完全），是官方的 LLVM 中间表示。LLVM 也用其它层级表示程序，从技术上说，这让它们也成了 IR，但是为了避免混淆，我们不把它们称作 LLVM IR；不管怎样，Instruction 类以及其它构成了官方的通用中间表示，我们为之保留 LLVM IR 这个名字。LLVM 文档也采用了这个术语。

起初 LLVM 是一系列工具，它们围绕 LLVM IR 运转，在这个层级运行的优化器数量众多，表现成熟，这是 LLVM IR 的功劳。这种 IR 有三种等价形式：

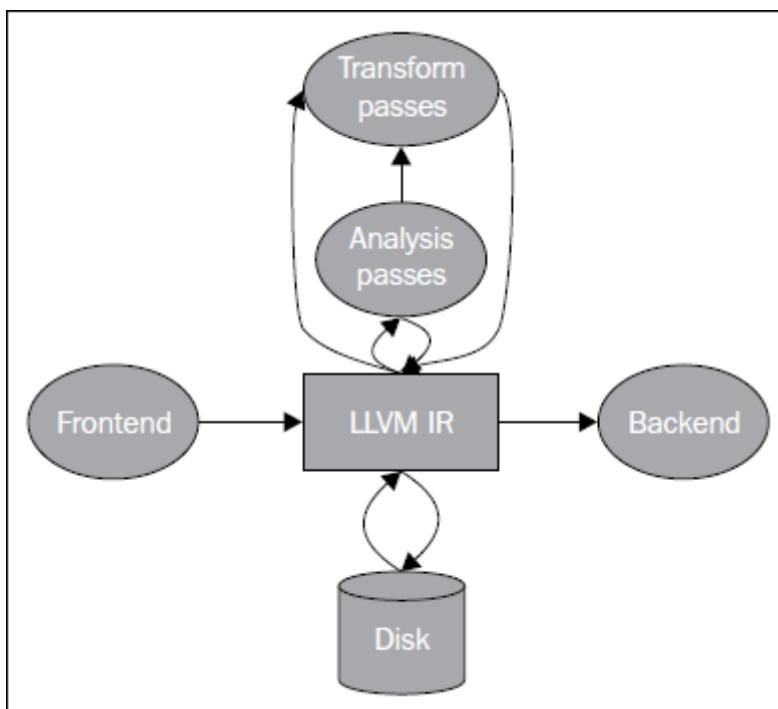
- 驻留内存的表示（指令类等）
- 磁盘上的以空间高效方式编码的位表示（bitcode 文件）
- 磁盘上的人类可读文本表示（LLVM 汇编文件）

LLVM 提供了各种工具和程序库，让我们能够操作处理任何形式的 IR。因此，这些工具可以从内存到磁盘转换 IR，或者反过来，也可以执行优化算法，如下图阐明的那样：

5.1.1 理解 LLVM IR 的目标依赖

LLVM IR 被设计为尽可能地与目标无关，但是它仍然表现出某些目标特定的属性。多数人批评 C/C++ 语言内在的目标依赖的本性。为了理解这个观点，考虑当你在 Linux 系统上使用标准 C 头文件时，例如，你的程序隐式地导入一些头文件，从 Linux 头文件目录 bits。这个目录包含目标相关的头文件，其中的一些宏定义约束某些实体使用一个特别的类型，它符合此内核机器的 syscalls 的期望。随后，举例来说，当前端解析你的代码的时候，也需要为 int 使用不同的长度，取决于打算在什么目标机器上运行此代码。

因此，程序库头文件和 C 类型已然都是目标相关的，这使得生成目标无关的 IR 充满挑战，这种 IR 可以随后被翻译到不同的目标。如果你只考虑目标相关的 C 标准库头文件，解析一个给定的编译单元得到的 AST



已然是目标相关的，甚至在翻译为 LLVM IR 之前。而且，前端生成的 IR 代码用了类型长度、调用惯例、特殊库调用，这些都匹配每个目标的 ABI 所定义的内容。还有，LLVM IR 是相当灵活多面的，能够以一种抽象的方法处理各种独特的目标。

5.2 练习基础工具转换 IR 格式

我们提到 LLVM IR 可以在磁盘上存储为两种格式：bitcode 和汇编文本。下面我们将学习如何使用它们。考虑下面的 sum.c 源代码：

```
int sum(int a, int b) {  
    return a+b;  
}
```

为了让 Clang 生成 bitcode，可以用下面的命令：

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

为了生成汇编表示，可以用下面的命令：

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

还可以汇编 LLVM IR 汇编文本，生成 bitcode：

```
$ llvm-as sum.ll -o sum.bc
```

为了将 bitcode 变换为 IR 汇编，这是反向的，可以使用反汇编器：

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

llvm-extract 工具能提取 IR 函数、全局变量，还能从 IR 模块中删除全局变量。例如，用下面的命令从 sum.bc 中提取函数 sum：

```
$ llvm-extract -func=sum sum.bc -o sum-fn.bc
```

在这个特别的例子中，从 sum.bc 到 sum-fn.bc 没有任何变化，因为 sum 已然是这个模块中唯一的函数。

5.3 介绍 LLVM IR 语言的语法

观察如下 LLVM IR 汇编文件 sum.ll：

```
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-
↪f32:32:32-f64:64:64-v64:64:64-v128:128:128-a0:0:64-s0:64:64-f80:128:128-n8:16:32:64-
↪S128"
target triple = "x86_64-apple-macosx10.7.0"

define i32 @sum(i32 %a, i32 %b) #0 {
entry:
    %a.addr = alloca i32, align 4
    %b.addr = alloca i32, align 4
    store i32 %a, i32* %a.addr, align 4
    store i32 %b, i32* %b.addr, align 4
    %0 = load i32, i32* %a.addr, align 4
    %1 = load i32, i32* %b.addr, align 4
    %add = add nsw i32 %0, %1
    ret i32 %add
}

attributes #0 = { nounwind ssp uwtable ... }
```

整个 LLVM 文件的内容，无论汇编或者 bitcode，定义了一个所谓的 LLVM 模块 (module)。模块是 LLVM IR 的顶层数据结构。每个模块包含一系列函数，每个函数包含一系列基本块，每个基本块包含一系列指令。模块还包含一些外围实体以支持其模型，例如全局变量、目标数据布局、外部函数原型，还有数据结构声明。

LLVM 局部值是汇编语言中的寄存器的模拟，有一个以 % 符号开头的任意的名字。如此，%add = add nsw i32 %0, %1 表示相加局部值 %0 和 %1，结果存放到新的局部值 %add。你可自由地赋予这些值任意的名字，但是如果你缺乏创造力，你可以只是用数字。在这个短小的例子中，我们已然看到 LLVM 如何表达它的基本性质：

- 它采用静态单赋值 (SSA) 形式。注意没有一个值是被重复赋值的；每个值只有单一赋值定义了它。每次使用一个值，可以立刻向后追溯到给出其定义的唯一指令。这可以极大地简化优化，因为 SSA 形式建立了平凡的 use-def 链，也就是一个值到达使用之处的定义的列表。如果 LLVM 不采用 SSA 形式，我们将需要单独运行一次数据流分析，以计算 use-def 链，对于经典的优化，这是必不可少的，例如常量传播和公共子表达式消除。
- 它以三地址指令组织代码。数据处理指令有两个源操作数，有一个独特的目标操作数以存放结果。
- 它有无限数量的寄存器。注意 LLVM 局部值可以命名为任意以 % 符号开头的名字，包括从 0 开始的数字，例如 %0, %1, 等等，不限制不同的值的最大数量。

字段 target datalayout 包含 target triple 的字节顺序和类型长度信息，它由 target host 描述。有些优化必须知道目标的数据布局，才能正确地转换代码。我们来观察 layout 是如何声明的：

```
target datalayout = "e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-
↪f32:32:32-f64:64:64-v64:64:64-v128:128:128-a0:0:64-s0:64:64-f80:128:128-n8:16:32:64-
↪S128"
target triple = "x86_64-apple-macosx10.7.0"
```

从上面的字符串，我们可以得知如下事实：

- 目标是一个运行 macOSX 10.7.0 的 x86_64 处理器。它是小端字节顺序，这由 layout 中的第一个字母（小写的 e）表示。大端字节顺序用大写的 E 表示。
- 类型的信息以 type:<size>:<abi>:<preferred> 的格式提供。在上面的例子中，p:64:64:64 表示一个长度为 64 位的指针，ABI 和首选对齐方式都以 64 位边界对齐。ABI 对齐设置一个类型最小所需的对齐，而首选对齐设置一个可能更大的值，如果这是可获利的。32 位整数类型 i32:32:32，长度是 32 位，32 位 ABI 和首选对齐，等等。

函数声明深度仿效 C 的语法：

```
define i32 @sum(i32 %a, i32 %b) #0 {
```

这个函数返回一个 i32 类型的值，有两个 i32 参数，%a 和 %b。局部标识符总是使用前缀 %，而全局标识符使用 @。LLVM 支持广泛的类型，但是下面是其最重要的类型：

- 任意长度的整数，表示形式：iN；通常的例子是 i32, i64, 和 i128。
- 浮点类型，例如 32 位单精度浮点和 64 位双精度浮点。
- 向量类型，表示格式：<#elements> x <elementtype>。包含四个 i32 元素的向量写为 <4 x i32>。

函数声明中的标签 #0 映射到一组函数属性，这也非常类似于 C/C++ 的函数和方法所用的属性。在文件的末尾定义了一组属性：

```
attributes #0 = { nounwind ssp uwtable "disable-tail-calls"="false" "less-precise-
↪fpmad"="false" "no-frame-pointer-elim"="true" "no-frame-pointer-elim-non-leaf" "no-
↪infs-fp-math"="false" "no-nans-fp-math"="false" "stack-protector-buffer-size"="8"
```

(续下页)

(接上页)

```
→ "target-cpu"="core2" "target-features"="+cx16,+fxsr,+mmx,+sse,+sse2,+sse3,+ssse3"
→ "unsafe-fp-math"="false" "use-soft-float"="false" }
```

举例来说，`nounwind` 标注一个函数或者方法不抛出异常，`ssp` 告诉代码生成器使用 `stack smash protector`，尽力提供代码安全，防御攻击。

函数体被显式地划分成基本块（BB: basic block），标签（label）用于开始一个新的基本块。一个标签关联一个基本块，如同一个值的定义关联一条指令。如果一个标签声明遗漏了，LLVM 汇编器会自动生成一个，运用它自己的命名方案。基本块是指令的序列，它的第一条指令是其单一入口点，它的最后一条指令是其单一出口点。这样，当代码跳跃到对应一个基本块的标签时，我们知道它将执行这个基本块中的所有指令，直到最后一条指令——这条指令将改变控制流，跳跃到其它的基本块。基本块和它们关联的标签，需要遵从下面的条件：

- 每个 BB 需要以一个终结者指令结束，它跳跃到其它 BB 或者从函数返回
- 第一个 BB，称为入口 BB，它在一个 LLVM 函数中是特殊的，不能作为任何跳转指令的目标

我们的 LLVM 文件，`sum.ll`，只有一个 BB，因为它没有跳跃、循环或者调用。函数的开头以 `entry` 标签标记，它以返回指令 `ret` 结束：

```
entry:
  %a.addr = alloca i32, align 4
  %b.addr = alloca i32, align 4
  store i32 %a, i32* %a.addr, align 4
  store i32 %b, i32* %b.addr, align 4
  %0 = load i32, i32* %a.addr, align 4
  %1 = load i32, i32* %b.addr, align 4
  %add = add nsw i32 %0, %1
  ret i32 %add
```

指令 `alloca` 在当前函数的栈帧上预留空间。空间的大小取决于元素类型的长度，而且遵从指定的对齐方式。第一条指令，`%a.addr = alloca i32, align 4`，分配了一个 4 字节的栈元素，它遵从 4 字节对齐。指向栈元素的指针存储在局部标识符 `%a.addr` 中。指令 `alloca` 通常用以表示局部（自动）变量。

利用 `store` 指令，参数 `%a` 和 `%b` 被存储到栈位置 `%a.addr` 和 `%b.addr`。这些值通过 `load` 指令被加载回来，从相同的内存位置，它们在加法指令 `%add = add nsw i32 %0, %1` 中被使用。最后，加法的结果 `%add` 由函数返回。`nsw` 标记指定这个加法操作是“no signed wrap”的，表示该操作是已知不会溢出的，允许作某些优化。如果你对 `nsw` 标记背后的历史感兴趣，这份 LLVMdev 帖子是值得阅读的：<http://lists.cs.uiuc.edu/pipermail/llvmdev/2011-November/045730.html>，作者 Dan Gohman。

实际上，这里的 `load` 和 `store` 指令是多余的，函数参数可以直接为加法指令所用。Clang 默认使用 `-O0`（无优化），不会消除无用的 `load` 和 `store`。如果改为用 `-O1` 编译，输出的代码简单得多，如下所示：

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

(续下页)

(接上页)

```
entry:
  %add = add nsw i32 %b, %a
  ret i32 %add
}
...
```

编写短小的例子测试目标后端，或者以此学习基础的 LLVM 概念，这时直接使用 LLVM 汇编是非常便利的。然而，对于前端编写者，我们推荐利用程序库接口构建 LLVM IR，这是下一节的主题。你可以在此处查看完整的 LLVM IR 汇编语法文档：<http://llvm.org/docs/LangRef.html>。

5.3.1 介绍 LLVM IR 内存中的模型

驻留内存的表示严密地建模了我们刚刚介绍的 LLVM 语言语法。表述 IR 的 C++ 类的头文件位于 `include/llvm/IR`。下面列举了其中最重要的类：

- **Module** 类聚合了整个翻译单元用到的所有数据，它是 LLVM 术语中的“module”的同义词。它声明了 `Module::iterator` typedef，作为遍历这个模块中的函数的简便方法。你可以用 `begin()` 和 `end()` 方法获取这些迭代器。在此处查看它的全部接口：http://llvm.org/docs/doxygen/html/classllvm_1_1Module.html。
- **Function** 类包含有关函数定义和声明的所有对象。对于声明来说（用 `isDeclaration()` 检查它是否为声明），它仅包含函数原型。无论定义或者声明，它都包含函数参数的列表，可通过 `getArgumentList()` 方法或者 `arg_begin()` 和 `arg_end()` 这对方法访问它。你可以通过 `Function::arg_iterator` typedef 遍历它们。如果 **Function** 对象代表函数定义，你可以通过这样的语句遍历它的内容：`for (Function::iterator i = function.begin(), e = function.end(); i != e; ++i)`，你将遍历它的基本块。可在此处查看它的全部接口：http://llvm.org/docs/doxygen/html/classllvm_1_1Function.html。
- **BasicBlock** 类封装了 LLVM 指令序列，可通过 `begin()/end()` 访问它们。你可以利用 `getTerminator()` 方法直接访问它的最后一条指令，你还可以用一些辅助函数遍历 CFG，例如通过 `getSinglePredecessor()` 访问前驱基本块，当一个基本块有单一前驱时。然而，如果它有多个前驱基本块，就需要自己遍历前驱列表，这也不难，你只要逐个遍历基本块，查看它们的终结指令的目标基本块。可在此处查看它的全部接口：http://llvm.org/docs/doxygen/html/classllvm_1_1BasicBlock.html。
- **Instruction** 类表示 LLVM IR 的运算原子，一个单一的指令。利用一些方法可获得高层级的断言，例如 `isAssociative()`, `isCommutative()`, `isIdempotent()`, 和 `isTerminator()`，但是它的精确的功能可通过 `getOpcode()` 获知，它返回 `llvm::Instruction` 枚举的一个成员，代表了 LLVM IR opcode。可通过 `op_begin()` 和 `op_end()` 这对方法访问它的操作数，它从 **User** 超类继承得到，我们很快将介绍这个超类。可在此处查看它的全部接口：http://llvm.org/docs/doxygen/html/classllvm_1_1Instruction.html。

我们还没介绍 LLVM 最强大的部分（依托 SSA 形式）：**Value** 和 **User** 接口；它们让你能够轻松操作 use-def 和 def-use 链。在 LLVM 驻留内存的 IR 中，一个继承自 **Value** 的类意味着，它定义了一个结果，可被其它 IR 使用。而继承自 **User** 的子类意味着，这个实体使用了一个或者多个 **Value** 接口。**Function** 和 **Instruction** 同时是 **Value** 和 **User** 的子类，而 **BasicBlock** 只是 **Value** 的子类。为了理解以上内容，让我们深入地分析这两个类：

- **Value** 类定义了 `use_begin()` 和 `use_end()` 方法，让你能够遍历各个 **User**，为访问它的 def-use 链提供了轻

松的方法。对于每个 Value 类，你可以通过 getName() 方法访问它的名字。这个模型决定了任何 LLVM 值都有一个和它关联的不同的标识。例如，%add1 可以标识一个加法指令的结果，BB1 可以标识一个基本块，myfunc 可以标识一个函数。Value 还有一个强大的方法，称为 replaceAllUsesWith(Value *)，它遍历这个值的所有使用者，用某个其它的值替代它。这是一个好的例子，演示如何替换指令和编写快速的优化。可在此处查看它的全部接口：http://llvm.org/docs/doxygen/html/classllvm_1_1Value.html。

- User 类定义了 op_begin() 和 op_end() 方法，让你能够快速访问所有它用到的 Value 接口。注意这代表了 use-def 链。你也可以利用一个辅助函数，称为 replaceUsesOfWith(Value *From, Value *To)，替换所有它用到的值。可在此处查看它的全部接口：http://llvm.org/docs/doxygen/html/classllvm_1_1User.html。

5.4 编写一个定制的 LLVM IR 生成器

利用 LLVM IR 生成器 API，程序化地为 sum.ll 构建 IR (sum.ll 是以 -O0 优化级别创建的，即没有优化)，这是可能的。在这个小节，我们将一步一步地介绍如何实现它。首先，看一看我们需要的头文件：

- #include <llvm/ADT/SmallVector.h>：这是为了引入 SmallVector<> 模板，这个数据结构帮助我们构建高效的向量，当元素数量不大的时候。查看 <http://llvm.org/docs/ProgrammersManual.html> 关于 LLVM 数据结构的介绍。
- #include <llvm/Analysis/Verifier.h>：验证 Pass 是一个重要的分析，检查你的 LLVM 模块是否恰当地被构建，遵从 IR 规则。
- #include <llvm/IR/BasicBlock.h>：这个头文件声明 BasicBlock 类，这是我们已经介绍过的重要的 IR 实体。
- #include <llvm/IR/CallingConv.h>：这个头文件定义函数调用用到的一套 ABI 规则，例如在何处存储函数参数。
- #include <llvm/IR/Function.h>：这个头文件声明 Function 类，一种 IR 实体。
- #include <llvm/IR/Instructions.h>：这个头文件声明 Instruction 类的所有子类，一种基本的 IR 数据结构。
- #include <llvm/IR/LLVMContext.h>：这个头文件存储 LLVM 程序库的全局域数据，每个线程使用不同的 context，让多线程实现正确工作。
- #include <llvm/IR/Module.h>：这个头文件声明 Module 类，IR 层级结构的顶层实体。
- #include <llvm/Bitcode/ReaderWriter.h>：这个头文件为我们提供了读写 LLVM bitcode 文件的代码。
- #include <llvm/Support/ToolOutputFile.h>：这个头文件声明了一个辅助类，用以写输出文件。

在这个例子中，我们还从 llvm 名字空间导入符号：

```
using namespace llvm;
```

现在，是时候以分步的方式编写代码了：

1. 我们要写的第一份代码是定义一个新的辅助函数，称为 makeLLVMModule，它返回一个指针指向我们的模块实例，即包含所有其它 IR 对象的顶层 IR 实体：


```
Module *makeLLVMModule() {
    Module *mod = new Module("sum.ll", getGlobalContext());
    mod->setDataLayout("e-p:64:64:64-i1:8:8-i8:8:8-i16:16:16-i32:32:32-i64:64:64-
    ↪f32:32:32-f64:64:64-v64:64:64-v128:128:128-a0:0:64-s0:64:64-f80:128:128-n8:16:32:64-
    ↪S128");
    mod->setTargetTriple("x86_64-apple-macosx10.7.0");
}
```

如果我们在模块中指定三元组 (triple) 和数据布局 (data layout) 对象, 就开启了依赖这些信息的优化, 但是需要匹配 LLVM 后端用到的数据布局和三元组字符串。然而, 你可以不指定它们, 如果你不关心依赖布局的优化, 打算在后端中显式地指定使用什么目标。为了创建一个模块, 我们从 `getGlobalContext()` 得到当前的 LLVM 上下文 (context), 定义模块的名字。我们选择使用被用作模型的文件的名字, `sum.ll`, 但是你可以选择任意其它的模块名字。上下文是 `LLVMContext` 类的一个实例, 为了保证线程安全, 必须按照顺序访问它, 因为多线程的 IR 生成必须给予每个线程一个上下文。`setDataLayout()` 和 `setTargetTriple()` 函数让我们能够设置字符串, 这些字符串定义了我们的模块的数据布局 and 三元组。

2. 为了声明我们的 `sum` 函数, 首先定义函数的签名

```
SmallVector<Type*, 2> FuncTyArgs;
FuncTyArgs.push_back(IntegerType::get(mod->getContext(), 32));
FuncTyArgs.push_back(IntegerType::get(mod->getContext(), 32));
FunctionType *FuncTy = FunctionType::get(/*Result=*/IntegerType::get
                                         (mod->getContext(), 32),
                                         /*Params=*/FuncTyArgs,
                                         /*isVarArg=*/false);
```

我们的 `FunctionType` 对象指定了一个函数, 它返回 32-bit 整数类型, 没有变量参数, 有两个 32-bit 整数参数。

3. 我们利用 `Function::Create()` 静态方法创建了一个函数——输入前面定义的函数类型 `FuncTy`, 还有链接类型和模块实例。`GlobalValue::ExternalLinkage` 枚举成员表明这个函数可以被其它模块 (翻译单元) 引用。

```
Function *funcSum =
    Function::Create(FuncTy, GlobalValue::ExternalLinkage, "sum", mod);
funcSum->setCallingConv(CallingConv::C);
```

4. 接着, 我们需要存储参数的值指针, 为了能够在后面使用它们。为此, 我们用到了函数参数的迭代器。`int32_a` 和 `int32_b` 分别指向函数的第一个和第二个参数。我们还设置了参数的名字, 这是可选的, 因为 LLVM 可以提供临时名字:

```
Function::arg_iterator args = funcSum->arg_begin();
Value *int32_a = args++;
int32_a->setName("a");
Value *int32_b = args++;
int32_b->setName("b");
```

5. 作为函数体的开始, 我们用标签 (或值名字) `entry` 创建了第一个基本块, 将其存储为 `labelEntry` 指针。

我们需要输入这个基本块的所属函数的引用：

```
BasicBlock *labelEntry =  
    BasicBlock::Create(mod->getContext(), "entry", funcSum, 0);
```

6. 现在基本块 `entry` 已准备好填充指令了。我们为基本块添加两个 `alloca` 指令，建立 4 字节对齐的 32-bit 栈元素。调用指令的构建的方法时，需要给出指令所属基本块的引用。默认地，新的指令被插入到基本块的末尾，如下：

```
AllocaInst *ptrA =  
    new AllocaInst(IntegerType::get(mod->getContext(), 32), "a.addr",  
                    labelEntry);  
ptrA->setAlignment(4);  
AllocaInst *ptrB =  
    new AllocaInst(IntegerType::get(mod->getContext(), 32), "b.addr",  
                    labelEntry);  
ptrB->setAlignment(4);
```

备注： 可选地，你可以使用被称作 `IRBuilder<>` 的辅助模板类建造 IR 指令（见 http://llvm.org/docs/doxygen/html/classllvm_1_1IRBuilder.html）。然而，为了能够向你呈现原始的接口，我们选择不使用它。如果你想使用它，只需要包含头文件 `llvm/IR/IRBuilder.h`，以 LLVM Context 对象实例化这个类，调用 `SetInsertPoint()` 方法指定你想插入新指令的位置。然后，即可调用任意的指令创建方法，例如 `CreateAlloca()`。

7. 利用 `alloca` 指令返回的指针 `ptrA` 和 `ptrB`，我们将函数参数 `int32_a` 和 `int32_b` 存储到堆栈位置。在此例中，尽管 `store` 指令在随后的代码中被 `st0` 和 `st1` 引用，但是这些指针不会被用到，因为 `store` 指令不产生结果。`StoreInst` 的第三个参数指定 `store` 是否易变（`volatile`），此处为 `false`：

```
StoreInst *st0 = new StoreInst(int32_a, ptrA, false, labelEntry);  
st0->setAlignment(4);  
StoreInst *st1 = new StoreInst(int32_b, ptrB, false, labelEntry);  
st1->setAlignment(4);
```

8. 我们还创建了非易变的 `load` 指令，从堆栈位置 `ld0` 和 `ld1` 加载值。然后，这些值被用作 `add` 指令的参数，加法运算的结果——`addRes`，被作为函数 `sum` 的返回值。接着，`makeLLVMModule` 函数返回 LLVM IR 模块，它包含我们刚刚创建的函数 `sum`：

```
LoadInst *ld0 = new LoadInst(ptrA, "", false, labelEntry);  
ld0->setAlignment(4);  
LoadInst *ld1 = new LoadInst(ptrB, "", false, labelEntry);  
ld1->setAlignment(4);  
  
BinaryOperator *addRes = BinaryOperator::Create(Instruction::Add, ld0, ld1,  
                                                "add", labelEntry);
```

(续下页)

(接上页)

```
ReturnInst::Create(mod->getContext(), addRes, labelEntry);

return mod;
```

备注：每个指令的创建函数都有大量变种。查阅头文件 `include/llvm/IR` 或者 `doxygen` 文档，了解所有可能的选项。

9. IR 生成程序作为一个单独的工具，它需要一个 `main()` 函数。在此 `main()` 函数中，我们调用 `makeLLVMModule()` 创建一个模块，调用 `verifyModule()` 验证 IR 的构建。枚举成员 `PrintMessageAction` 指示输出错误消息到 `stderr`，当验证失败的时候。最后，利用函数 `WriteBitcodeToFile`，模块 `bitcode` 被写到磁盘，如下面的代码所示：

```
int main() {
    Module *Mod = makeLLVMModule();
    verifyModule(*Mod, PrintMessageAction);
    std::string ErrorInfo;
    OwningPtr<tool_output_file> Out(new tool_output_file("./sum.bc", ErrorInfo,
↪sys::fs::F_None));
    if (!ErrorInfo.empty()) {
        errs() << ErrorInfo << '\n';
        return -1;
    }
    WriteBitcodeToFile(Mod, Out->os());
    Out->keep(); // Declare success
    return 0;
}
```

5.4.1 编译并运行 IR 生成器

为了编译这个工具，你可以使用第 3 章（工具和设计）中的同样的 `Makefile`。`Makefile` 的最关键的部分是 `llvm-config -libs` 调用，它定义你的项目将链接哪些 LLVM 程序库。在此项目中，将使用 `bitwriter` 部件，而不是第 3 章（工具和设计）所用的 `bitreader` 部件。因此，修改 `llvm-config` 调用为 `llvm-config -libs bitwriter core support`。用下面的命令编译、运行和检查生成的 IR：

```
$ make && ./sum && llvm-dis < sum.bc
...
define i32 @sum(i32 %a, i32 %b) {
entry:
    %a.addr = alloca i32, align 4
    %b.addr = alloca i32, align 4
```

(续下页)

(接上页)

```
store i32 %a, i32* %a.addr, align 4
store i32 %b, i32* %b.addr, align 4
%0 = load i32, i32* %a.addr, align 4
%1 = load i32, i32* %b.addr, align 4
%add = add nsw i32 %0, %1
ret i32 %add
}
```

5.4.2 学习如何用 C++ 后端生成任意 IR 的构造代码

llc 工具——第 6 章（后端）有详细的说明——有一个有趣的功能，就是辅助开发者构建 IR。这个 llc 工具能够为一个给定的 LLVM IR 文件（bitcode 或汇编）生成 C++ 源代码，此源代码可以构建生成相同的 IR 文件。这让构建 IR 的 API 易于使用，因为我们能够借助其它已知的 IR 文件学习如何构建甚至最易懂的 IR 表达式。LLVM 通过 C++ 后端实现这个功能，给 llc 工具输入参数 `-march=cpp` 即可使用这个功能：

```
$ llc -march=cpp sum.bc -o sum.cpp
```

打开 `sum.cpp` 文件，注意到生成的 C++ 代码跟我们前面小节所写的很相似。

备注：如果你配置 LLVM 编译时选择所有目标，那么 C++ 后端是默认包含的。然而，如果你在配置时指定目标，就需要包含 C++ 后端。使用后端名字 `cpp` 以包含 C++ 后端，例如，`-enable-targets=x86,arm,mips,cpp`。

5.5 IR 层次的优化

一旦翻译为 LLVM IR，一个程序将经受各种各样的目标无关的代码优化。举例来说，优化可一次作用于一个函数，或者一次作用于一个模块。当优化是过程间优化时，使用后者。为了强化过程间优化的作用，使用者可以利用 `llvm-link` 将几个 LLVM 模块链接在一起成为单个模块。这让优化能够在更大的作用域运行；有时这称为链接时优化，因为它们是编译器中唯一可能超越翻译单元的优化。一个 LLVM 使用者可以访问所有这些优化，可以利用 `opt` 工具个别地调用它们。

5.5.1 编译时和链接时优化

`opt` 工具使用一套优化选项，和 Clang 编译器驱动器的一样：`-O0`，`-O1`，`-O2`，`-O3`，`-Os` 和 `Oz`。Clang 还支持 `-O4`，但是 `opt` 不支持。选项 `-O4` 是 `-O3` 和链接时优化（`-flto`）的同义词，但是如我们讨论的，在 LLVM 中开启链接时优化依赖于你如何组织输入文件。每个选项激活不同的优化流水线，它包含一套以特定顺序运行的优化。从 Clang 手册页面，我们看到下面的说明：

`-Ox` 选项：指定优化级别。`-O0` 表示“不作优化”：这个级别编译最快，生成的代码调试信息最丰富。`-O2` 是一个适度的优化级别，开启了大部分优化。`-Os` 和 `-O2` 相似，它额外开启减小代码长度的优化。`-Oz` 和 `-Os` 相

似，(也和-O2 相似)，但是它进一步减小代码长度。-O3 和-O2 相似，除了它开启更多的优化，这些优化执行更长的时间，或者可能产生更长的代码（以试图让程序运行得更快）。在所支持的平台上，-O4 开启链接时优化；目标文件以 *LLVM bitcode* 文件格式存储，整个程序的优化在链接时进行。-O1 是介于-O0 和-O2 之间的优化级别。

为了利用任意的这些预定义的优化序列，你可以运行 `opt` 工具，它操作 `bitcode` 文件。例如，下面的命令优化 `sum.bc` `bitcode`：

```
$ opt -O3 sum.bc -o sum-O3.bc
```

你还可以利用选项激活标准的编译时优化：

```
$ opt -std-compile-opts sum.bc -o sum-stdc.bc
```

或者，你使用一套标准的链接时优化：

```
$ llvm-link file1.bc file2.bc file3.bc -o=all.bc
$ opt -std-link-opts all.bc -o all-stdl.bc
```

通过 `opt` 应用个别的 `Pass` 也是可能的。一个非常重要的 LLVM `Pass` 是 `mem2reg`，它将 `alloca` 提升为 LLVM 局部值，可能会将它们变换为 SSA 形式，如果它们变换为局部值之后接受多个赋值。这种情况下，变换将引入 `phi` 函数（参考 http://llvm.org/doxygen/classllvm_1_1PHINode.html）——你若自己生成 LLVM IR，这是棘手的，但是对于 SSA 形式是必要的。因此，程序员更喜欢编写依赖 `alloca`、`load` 和 `store` 的次优代码，留待 `mem2reg` `Pass` 生成 SSA 版本，它包含生命期较长的局部值。这个 `Pass` 负责优化前面小节的例子 `sum.c`。举例来说，为了运行 `mem2reg`，然后计数模块中的每个指令，以这样的顺序，我们可以执行下面的命令（`Pass` 参数的顺序是要紧的）：

```
$ opt sum.bc -mem2reg -instcount -o sum-tmp.bc -stats
Statistics collected ...
1 instcount - Number of Add insts
1 instcount - Number of Ret insts
1 instcount - Number of basic blocks
2 instcount - Number of instructions (of all types)
1 instcount - Number of non-external functions
2 mem2reg - Number of alloca's promoted
2 mem2reg - Number of alloca's promoted with a single store
```

我们利用选项 `-stats` 强制让 LLVM 打印每个 `Pass` 的统计信息。否则，指令计数 `Pass` 将无声地结束，不报告指令的数目。

利用选项 `-time-passes`，我们还可以看到每个优化在总的执行时间中占用了多少执行时间。

```
$ opt sum.bc -time-passes -domtree -instcount -o sum-tmp.bc
```

这里列出了 LLVM 的分析、转换、辅助 `Pass` 的完整清单：<http://llvm.org/docs/Passes.html>。

备注：Pass 顺序问题指出，对代码应用优化的顺序极大地影响它的性能收益，让不同的程序得到最佳优化的顺序是不同的。选项-Ox 采用了预定义的优化序列，你应该明白它对于你的程序来说可能不是最佳的。如果你想做一个实验以揭示优化之间复杂的交互，就试着对你的代码运行 `opt -O3` 两次，看看它的性能和运行 `opt -O3` 一次有何不同（不一定更好）。

5.5.2 发现哪些 Pass 有用

优化通常由分析 Pass 和转换 Pass 组成。前者发掘性质和优化机会，生成必需的数据结构，后续为后者所用。两者都实现为 LLVM Pass，可能有依赖链。

在例子 `sum.ll` 中，我们看到在优化级别-O0 之下，用到了若干 `alloca`、`load` 和 `store` 指令。然而，当应用-O1 时，所有这些冗余的指令消失了，因为-O1 包含 `mem2reg` Pass。然而，如果你不知道 `mem2reg` 是重要的，你如何发现哪些 Pass 对你的程序有用呢？为了解这个问题，我们把未优化版本称为 `sum-O0.ll`，把优化后版本称为 `sum-O1.ll`。运用-O1 就可以得到后者：

```
$ opt -O1 sum-O0.ll -S -o sum-O1.ll
```

然而，如果你想得到更精细的信息，关于哪些转换实际上影响着结果，你可以向 `clang` 前端输入 `-print-stats` 选项（或者向 `opt` 输入 `-stats`）：

```
$ clang -Xclang -print-stats -emit-llvm -O1 sum.c -c -o sum-O1.bc
=====
-----
... Statistics Collected ...
=====
-----
1 cgsccl-pasmmgr - Maximum CGSCCPassMgr iterations on one SCC
1 functionattrs - Number of functions marked readnone
2 mem2reg - Number of alloca's promoted with a single store
1 reassociate - Number of insts reassociated
1 sroa - Maximum number of partitions per alloca
2 sroa - Maximum number of uses of a partition
4 sroa - Number of alloca partition uses rewritten
2 sroa - Number of alloca partitions formed
2 sroa - Number of allocas analyzed for replacement
2 sroa - Number of allocas promoted to SSA values
4 sroa - Number of instructions deleted
```

以上输出表明，`mem2reg` 和 `sroa` (scalar replacement of aggregates) 都去除了冗余的 `alloca`。为了查看一个 Pass 如何运作，试着只运行 `sroa`：

```
$ opt -sum-O0.ll -stats -sroa -o sum-O1.ll
=====
-----
... Statistics Collected ...
=====
-----
1 cgsccl-pasmgr - Maximum CGSCCPassMgr iterations on one SCC
1 functionattrs - Number of functions marked readnone
2 mem2reg - Number of alloca's promoted with a single store
1 reassociate - Number of insts reassociated
1 sroa - Maximum number of partitions per alloca
2 sroa - Maximum number of uses of a partition
4 sroa - Number of alloca partition uses rewritten
2 sroa - Number of alloca partitions formed
2 sroa - Number of allocas analyzed for replacement
2 sroa - Number of allocas promoted to SSA values
4 sroa - Number of instructions deleted
```

注意，sroa 也调用 mem2reg，即使没有在命令行显式地指定。如果只开启 mem2reg，你将看到相同的改进：

```
$ opt -sum-O0.ll -stats -mem2reg -o sum-O1.ll
=====
-----
... Statistics Collected ...
=====
-----
2 mem2reg - Number of alloca's promoted
2 mem2reg - Number of alloca's promoted with a single store
```

5.5.3 理解 Pass 依赖关系

在转换 Pass 和分析 Pass 之间，有两种主要的依赖类型：

- 显式依赖：转换 Pass 需要一种分析，则 Pass 管理器自动地安排它所依赖的分析 Pass 在它之前运行。如果你运行单个 Pass，它依赖其它 Pass，则 Pass 管理器会无声地安排必需的 Pass 在它之前运行。Loop Info 和 Dominator Tree 就是这种分析的例子，它们为其它 Pass 提供信息。支配者树（dominator tree）是重要的数据结构，它让 SSA 构建算法能够决定在何处放置 phi 函数。这样，举例来说，mem2reg 在其实现中请求支配者树，通过建立这两个 Pass 之间的依赖关系：

```
DominatorTree &DT = getAnalysis<DominatorTree>(Func);
```

- 隐式依赖：有些转换或者分析 Pass 要求 IR 代码运用特定的成语。以这种方式，它可以轻易地识别模式，即使 IR 有许多表达相同计算的其它方式。举例来说，如果一个 Pass 专门地被设计成刚好在另一个转换

Pass 之后运行，这种隐式依赖就可能出现。因此，这个 Pass 可能特殊地处理符合特定成语句式的代码（来自前一个 Pass）。这种情况，因为这种微妙的依赖是对于一个转换 Pass，而不是分析 Pass，所以你需要手动地以正确的顺序把这个 Pass 加到 Pass 队列中，通过命令行工具（clang 或者 opt）或者 Pass 管理器。如果进来的 IR 不使用这个 Pass 所期望的成语，这个 Pass 就无声地跳过其转换，因为它无法匹配代码。一个给定的优化级别所包含的 Pass 集合是自包含的，不会出现依赖问题。

利用 opt 工具你可以获取相关的信息，关于 Pass 管理器如何安排 Pass，会使用哪些依赖 Pass。例如，当你只请求运行 mem2reg Pass 时，想知道所用到的完整的 Pass 清单，你可以输入下面的命令：

```
$ opt sum-00.ll -debug-pass=Structure -mem2reg -S -o sum-01.ll
Pass Arguments: -targetlibinfo -tti -assumption-cache-tracker -domtree -mem2reg -
↳verify -print-module
Target Library Information
Target Transform Information
Assumption Cache Tracker
ModulePass Manager
  FunctionPass Manager
    Dominator Tree Construction
    Promote Memory to Register
    Module Verifier
Print module to stderr
```

在 Pass 参数列表中，我们看到 Pass 管理器极大地扩展了 Pass 的数量，使得 mem2reg Pass 正确运行。例如，domtree Pass 是 mem2reg 所要求的，因此 Pass 管理器自动包含了它。接着，它详细输出了用于运行每个 Pass 的结构；直接出现在 ModulePass Manager 之后的层次状的 Pass 是基于每个模块运行的，而在 FunctionPass 下面的层次状的 Pass 是基于每个函数运行的。我们还可以看到 Pass 执行的顺序，Promote Memory to Register Pass 在它的依赖者 Dominator Tree Construction Pass 之后运行。

5.5.4 理解 Pass API

Pass 类是实现优化的主要资源。然而，我们从不直接使用它，而是通过清楚的子类使用它。当实现一个 Pass 时，你应该选择适合你的 Pass 的最佳粒度，适合此粒度的最佳子类，例如基于函数、模块、循环、强联通区域，等等。常见的这些子类如下：

- **ModulePass**：这是最通用的 Pass；它一次分析整个模块，函数的次序不确定。它不限定使用者的行为，允许删除函数和其它修改。为了使用它，你需要写一个类继承 ModulePass，并重载 runOnModule() 方法。
- **FunctionPass**：这个子类允许一次处理一个函数，处理函数的次序不确定。这是应用最多的 Pass 类型。它禁止修改外部函数、删除函数、删除全局变量。为了使用它，需要写一个它的子类，重载 runOnFunction() 方法。
- **BasicBlockPass**：这个类的粒度是基本块。FunctionPass 类禁止的修改在这里也是禁止的。它还禁止修改或者删除外部基本块。使用者需要写一个类继承 BasicBlockPass，并重载它的 runOnBasicBlock() 方法。

被重载的入口函数 runOnModule()、runOnFunction()、runOnBasicBlock() 返回布尔值 false，如果被分析的单元（模块、函数和基本块）保持不变，否则返回布尔值 true。参考关于 Pass 子类的完整文档：<http://llvm.org>。

org/docs/WritingAnLLVMPass.html。

5.5.5 写一个定制的 Pass

假设我们想要计数一个程序中每个函数的参数的数目，输出函数的名字。让我们写一个 Pass 实现它。首先，我们需要选择正确的 Pass 的子类。FunctionPass 看起来是适合的，因为我们对函数次序没有要求，不需要删除任何东西。

我们把 Pass 命名为 FnArgCnt，放在 LLVM 源代码树中：

```
$ cd <llvm_source_tree>
$ mkdir lib/Transforms/FnArgCnt
$ cd lib/Transforms/FnArgCnt
```

文件 FnArgCnt.cpp，位于 lib/Transforms/FnArgCnt，需要实现这个 Pass，内容如下：

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

using namespace llvm;

namespace {
  class FnArgCnt : public FunctionPass {
  public:
    static char ID;
    FnArgCnt() : FunctionPass(ID) {}

    virtual bool runOnFunction(Function &F) {
      errs() << "FnArgCnt --- ";
      errs() << F.getName() << ": ";
      errs() << F.getArgumentList().size() << '\n';
      return false;
    }
  };
}

char FnArgCnt::ID = 0;
static RegisterPass<FnArgCnt> X("fnargcnt", "Function Argument Count Pass", false,
↪false);
```

首先，包含必需的头文件，从 llvm 名字空间采集符号：

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

(续下页)

(接上页)

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

using namespace llvm;
```

接着，我们声明 `FnArgCnt`——我们的函数 Pass 子类——并在 `runOnFunction()` 方法中实现主要的 Pass 功能。在每个函数的上下文中，我们打印函数名字和它接收的参数的数目。这个函数返回 `false`，因为没有修改被分析的函数。我们的子类的代码如下：

```
namespace {
  class FnArgCnt : public FunctionPass {
  public:
    static char ID;
    FnArgCnt() : FunctionPass(ID) {}

    virtual bool runOnFunction(Function &F) {
      errs() << "FnArgCnt --- ";
      errs() << F.getName() << ": ";
      errs() << F.getArgumentList().size() << '\n';
      return false;
    }
  };
}
```

ID 由 LLVM 内部决定，用以识别一个 Pass，它可以声明为任意值：

```
char FnArgCnt::ID = 0;
```

最后，我们处理 Pass 注册机制，它用当前 Pass 管理器在 Pass 加载时间注册它：

```
char FnArgCnt::ID = 0;
static RegisterPass<FnArgCnt> X("fnargcnt", "Function Argument Count Pass", false,
    false);
```

第 1 个参数，`fnargcnt`，是 Pass 的名字，`opt` 工具用它识别这个 Pass，而第 2 个参数是它的扩充名字。第 3 个参数指示这个 Pass 是否修改当前 CFG，最后的参数指示它是不是一个分析 Pass。

在 LLVM 编译系统中编译和运行你的新 Pass

为了编译和安装这个 Pass，我们需要一个 Makefile，放在源代码的目录中。和前面的项目不同，我们不再是编译一个独立工具，这个 Makefile 将被集成到 LLVM 编译系统。因为它依赖 LLVM 主 Makefile，主 Makefile 实现了大量规则，所以它的内容比独立工具的 Makefile 简单得多。参考下面的代码：

```
# Makefile for FnArgCnt pass

# Path to top level of LLVM hierarchy
LEVEL = ../../..

# Name of the library to build
LIBRARYNAME = LLVMFnArgCnt

# Make the shared library become a loadable module so the tools can
# dlopen/dlsym on the resulting library.
LOADABLE_MODULE = 1

# Include the makefile implementation stuff
include $(LEVEL)/Makefile.common
```

Makefile 中的注释是自我解释的，这里利用公用的 LLVM Makefile 创建了一个共享库。利用此基础设施，我们的 Pass 和其它的标准 Pass 被安装在一起，可以直接被 opt 加载，但是这需要你重新编译安装 LLVM。

我们还想让我们的 Pass 在目标目录中编译，这需要在 Transforms 目录的 Makefile 中包含我们的 Pass。因此，在 lib/Transforms/Makefile 中，需要修改 PARALLEL_DIRS 变量，让它包含 FnArgCnt Pass：

```
PARALLEL_DIRS = Utils Instrumentation Scalar InstCombine IPO Vectorize Hello ObjCARC
↪FnArgCnt
```

根据第 1 章（编译和安装 LLVM）的说明，需要重新配置 LLVM 项目：

```
$ cd path-to-build-dir
$ /PATH_TO_SOURCE/configure --prefix=/your/installation/folder
```

现在，离开目标目录，转到新 Pass 的目录，运行 make：

```
$ cd lib/Transforms/FnArgCnt
$ make
```

一个共享库将会出现在编译树下的 Debug+Asserts/lib 目录中。Debug+Asserts 应该被替换为你的配置模式，例如 Release，如果你配置了 release build。下面，调用 opt 运行这个定制的 Pass（在 Mac OS X 中）：

```
$ opt -load <path_to_build_dir>/Debug+Asserts/lib/LLVMFnArgCnt.dylib -fnargcnt < sum.
↪bc >/dev/null
FnArgCnt --- sum: 2
```

在 Linux 中需要使用恰当的共享库扩展名 (.so)。和期望的一样, sum.bc 模块只有一个函数, 它有两个整数参数, 如前面的输出显示的那样。

你还可以选择重新编译整个 LLVM 系统并重新安装。编译系统会安装一个新的 opt 程序, 不需要输入 -load 命令行参数, 它就能识别你的 Pass。

用你自己的 Makefile 编译和安装你的新 Pass

依赖于 LLVM 编译系统可能是件麻烦事, 例如需要重新配置整个项目, 或者重新编译新的代码和所有 LLVM 工具。然而, 我们可以创建一个独立的 Makefile, 它在 LLVM 源代码树之外编译我们的 Pass, 和之前我们编译项目一样。不依赖于 LLVM 源代码树是令人舒适的, 有时这是值得付出额外的努力建立你自己的 Makefile 的。

我们的独立 Makefile 将以第 3 章 (工具和设计) 中的 Makefile 为基础。这里的挑战是, 我们不再是编译一个工具, 而是一个共享库, 即编译我们的 Pass 的代码得到一个共享库, 它可以被 opt 工具随时地加载。

首先, 我们为项目创建一个单独的文件夹, 它不在 LLVM 源代码树中。我们把 Pass 的实现代码 FnArgCnt.cpp 文件放在里面。第二, 我们创建如下的 Makefile:

```
LLVM_CONFIG?=llvm-config

ifndef VERBOSE
QUIET:=@
endif

SRC_DIR?=$(PWD)
LD_FLAGS+=$(shell $(LLVM_CONFIG) --ldflags)
COMMON_FLAGS=-Wall -Wextra
CXX_FLAGS+=$(COMMON_FLAGS) $(shell $(LLVM_CONFIG) --cxxflags) -fno-rtti
CPP_FLAGS+=$(shell $(LLVM_CONFIG) --cppflags) -I$(SRC_DIR)

ifeq ($(shell uname), Darwin)
LOADABLE_MODULE_OPTIONS=-bundle -undefined dynamic_lookup
else
LOADABLE_MODULE_OPTIONS=-shared -Wl,-O1
endif

FNARGPASS=fnarg.so
FNARGPASS_OBJECTS=FnArgCnt.o

default: $(FNARGPASS)

%.o : $(SRC_DIR)/%.cpp
    @echo Compiling $*.cpp
    $(QUIET) $(CXX) -c $(CPP_FLAGS) $(CXX_FLAGS) $<
```

(续下页)

(接上页)

```
$(FNARGPASS) : $(FNARGPASS_OBJECTS)
    @echo Linking $@
    $(QUIET)$ (CXX) -o $@ $(LOADABLE_MODULE_OPTIONS) $(CXXFLAGS) $(LDFLAGS) $^

clean::
    $(QUIET)rm -rf $(FNARGPASS_OBJECTS) $(FNARGPASS)
```

对比第 3 章（工具和设计）中的 Makefile，这个 Makefile 的新颖之处（代码中高亮的部分），在于条件化定义 `LOADABLE_MODULE_OPTIONS` 变量，链接我们的共享库的命令行会用到它。它定义了平台相关的一套编译器选项，指导生成一个共享库而不是可执行文件。例如，对于 Linux，它使用 `-shared` 选项以创建共享库，以及 `-Wl -O1` 选项，`-O1` 选项传递给 GNU ld。这个选项要求 GNU 链接器执行符号表优化，减少程序库加载时间。如果你不使用 GNU 链接器，可以忽略这个选项。

我们还从链接器命令中去除了 `llvm-config -libs` 这个 shell 命令。这个命令用于给出我们的项目要链接的程序库。因为我们知道 `opt` 可执行文件已经含有我们用到的所有符号，所以我们简单地不包含任何冗余的程序库，以加快链接速度。

用下面的命令编译你的项目：

```
$ make
```

你的 Pass 被编译成了 `fnarg.so`，用下面的命令运行它：

```
$ opt -load=fnarg.so -fnargcnt < sum.c > /dev/null
FnArgCnt --- sum: 2
```

5.6 总结

LLVM IR 是前端（frontend）和后端（backend）的桥梁。这是目标无关优化发生的地方。在本章中，我们介绍了操纵 LLVM IR 的工具，研究了汇编语法，以及如何编写一个定制的 IR 代码生成器。此外，我们展示了 Pass 接口如何工作，如何应用优化，然后通过例子介绍如何编写我们自己的 IR 转换或者分析 Pass。

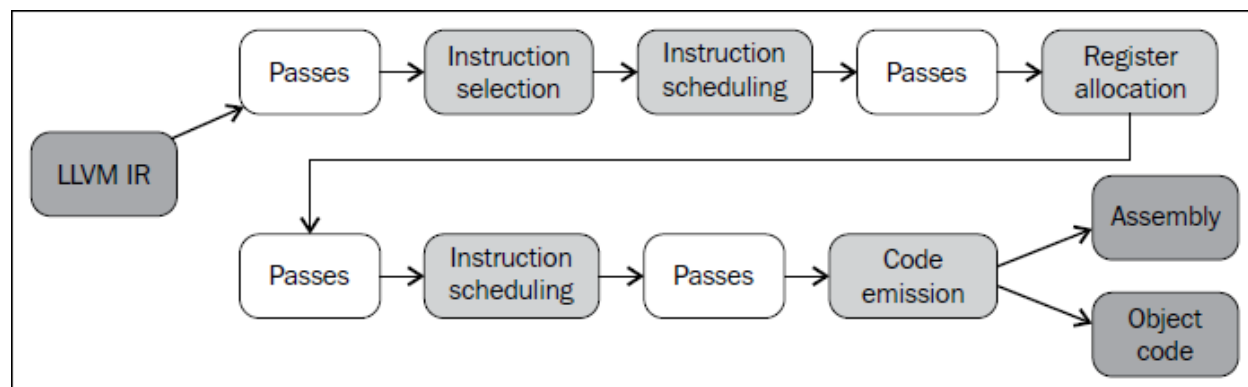
在下一章中，我们将讨论 LLVM 后端如何工作，如何建立自己的后端将 LLVM IR 代码翻译为一个定制的架构的指令。

后端 (backend) 由一套分析和转换 Pass 组成，它们的任务是代码生成，即将 LLVM 中间表示 (IR) 变换为目标代码 (或者汇编)。LLVM 支持广泛的目标：ARM，AArch64，Hexagon，MSP430，MIPS，Nvidia PTX，PowerPC，R600，SPARC，SystemZ，X86，和 XCore。所有这些后端共享一套共用的接口，它是目标无关代码生成器的一部分，以通用 API 的方法抽象化后端任务。每个目标必须特殊化代码生成通用类，以实现目标特定的行为。在本章中，我们将介绍 LLVM 后端的多种一般性质，这对于感兴趣的读者来说，无论他想编写一个新的后端，维护一个已有的后端，或者编写一个后端 Pass，都是很有用的。我们将介绍以下内容：

- LLVM 后端的组织结构概述
- 如何解释各种描述后端的 TableGen 文件
- 什么是 LLVM 指令选择以及如何运作
- 指令调度和寄存器分配的任务是什么
- 指令输出如何工作
- 编写你自己的后端 Pass

6.1 概述

将 LLVM IR 转换为目标汇编代码需要经历若干步骤。IR 被变换为后端友好的指令、函数、全局变量的表示。这种表示随着程序经历各种后端阶段而变化，越来越接近实际的目标指令。下图给出了必需的步骤的概观，从 LLVM IR 到目标代码或者汇编，而如白色框所指示的，可以执行非必需的优化 Pass 以进一步改进翻译的质量。



这个翻译流水线由后端的多个阶段组成，它们表示为浅灰色的中间框。它们在内部也称为 *super pass*，因为它们由若干小的 *Pass* 实现。它们和白色框的区别在于，前者这些 *Pass* 对后端的成功很关键，而后者对于提高所生成的代码的效率更重要。下面我们简略描述上图所说明的代码生成的各个阶段：

指令选择（*Instruction Selection*）过程将内存中的 IR 表示变换为目标特定的 *SelectionDAG* 节点。起初，这个过程将三地址结构的 LLVM IR 变换为 DAG（*Directed Acyclic Graph*）形式，这是有向无环图。每个 DAG 能够表示单一基本块的计算，这意味着每个基本块关联不同的 DAG。典型地节点表示指令，而边编码了它们之间的数据流依赖，但不限于此。转换为 DAG 是重要的，这让 LLVM 代码生成程序库能够运用基于树的模式匹配指令选择算法，它经过一些调整，也能工作在 DAG 上（而不仅仅是树）。到这个过程结束时，DAG 已将它所有的 LLVM IR 节点变换为目标机器节点，这些节点表示机器指令而不是 LLVM 指令。

指令选择之后，对于使用哪些目标指令执行每个基本块的计算，我们已经有了清楚的概念。这编码在 *SelectionDAG* 类中。然而，我们需要返回三地址表示形式，以决定基本块内部的指令顺序，因为 DAG 并不暗示互不依赖的指令之间的顺序。第 1 次指令调度（*Instruction Scheduling*），也称为前寄存器分配（RA）调度，对指令排序，同时尝试发现尽可能多的指令层次的并行。然后这些指令被变换为 *MachineInstr* 三地址表示。

回想一下，LLVM IR 的寄存器集是无限的。这个性质一直保持着，直到寄存器分配（*Register Allocation*），它将无限的虚拟寄存器的引用转换为有限的目标特定的寄存器集，寄存器不够时挤出（*spill*）到内存。

第 2 次指令调度，也称为后寄存器分配（RA）调度，在此时发生。因为此时在这个点可获得真实的寄存器信息，某些类型寄存器存在额外的风险和延迟，它们可被用以改进指令顺序。

代码输出（*Code Emission*）阶段将指令从 *MachineInstr* 表示变换为 *MCIInst* 实例。这种新的表示更适合汇编器和链接器，它有两种选择：输出汇编代码或者输出二进制块（*blob*）到一种特定的目标代码格式。

如此，整个后端流水线用到了四种不同层次的指令表示：内存中的 LLVM IR，*SelectionDAG* 节点，*MachineInstr*，和 *MCIInst*。

6.1.1 使用后端工具

llc 是后端的主要工具。如果我们带着前面一章的 sum.bc bitcode 继续旅程，我们可以用下面的命令生成它的汇编代码：

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

或者生成目标代码，可以用下面的命令：

```
$ llc sum.bc -filetype=obj -o sum.o
```

使用以上命令时，llc 会尝试选择一个后端匹配 sum.bc bitcode 中指定的目标三元组。使用 -march 选项可覆盖它而选择特定的后端。例如，用下面的命令生成 MIPS 目标代码：

```
$ llc -march=mips -filetype=obj sum.bc -o sum.o
```

如果你运行命令 llc -version，llc 会显示所支持的 -march 选项的完整列表。注意，这个列表和 LLVM 配置（详情见第 1 章，编译和安装 LLVM）中用到的 -enable-targets 选项兼容。

然而要注意的是，我们刚才强制让 llc 使用一个不同的后端为 bitcode 生成代码，这个 bitcode 起初是为 x86 编译的。在第 5 章（LLVM 中间表示）中，我们解释了 IR 具有目标相关的一面，尽管它是为所有后端设计的共同语言。因为 C/C++ 语言具有目标相关的属性，所以这种相关性会体现在 LLVM IR 中。

因此，当 bitcode 的目标三元组和运行 llc -march 的目标不匹配时，必须谨慎。这种情况可能会导致 ABI 不匹配，坏的程序行为，有时还会导致代码生成器失败。然而在大多数情况下，代码生成器不会失败，它生成的代码含有微妙的 bug，这是更糟糕的。

为了理解 IR 的目标依赖性在实践中怎样表现，让我们看一个例子。考虑你的程序分配了 char 指针的一个 vector，用以存储不同的字符串，你用通用的 C 语句 malloc(sizeof(char*)*n) 来为字符串 vector 分配内存。如果你在前端时指定了目标，比如 32 位 MIPS 架构，它生成的代码会让 malloc 分配 n x 4 字节的内存，因为在 32 位 MIPS 上每个指针是 4 字节。然而，如果你用 llc 编译这个 bitcode 而强制指定 x86_64 架构，它将生成坏的程序。在运行时，会发生潜在的分段错误（segmentation fault），因为 x86_64 架构的每个指针是 8 字节，这使得 malloc 分配的内存不足够。在 x86_64 上正确的 malloc 调用将分配 n x 8 字节。

6.2 学习后端代码结构

后端的实现分散在 LLVM 源代码树的不同目录中。代码生成背后的主要程序库位于 lib 目录和它的子文件夹 CodeGen、MC、TableGen、和 Target 中：

- CodeGen 目录包含的文件和头文件实现了所有通用的代码生成算法：指令选择，指令调度，寄存器分配，和所有它们需要的分析。
- MC 目录实现了低层次功能，包括汇编器（汇编解析器）、松弛算法（反汇编器）、和特定的目标文件格式如 ELF、COFF、Macho 等等。

- TableGen 目录包含 TableGen 工具的完整实现，它可以根据 .td 文件中的高层次的目标描述生成 C++ 代码。
- 每个目标的实现在 Target 的子文件夹中，如 Target/Mips，包括若干 .cpp、.h、和 .td 文件。为不同目标实现类似功能的文件倾向于共用类似的名字。

如果你编写一个新的后端，你的代码将仅仅出现在 Target 文件夹中的一个子文件夹。作为一个例子，我们用 Sparc 来阐明 Target/Sparc 子文件夹中的组织：

Filenames	Description
SparcInstrInfo.td SparcInstrFormats.td	Instruction and format definitions
SparcRegisterInfo.td	Registers and register classes definitions
SparcISelDAGToDAG.cpp	Instruction selection
SparcISelLowering.cpp	SelectionDAG node lowering
SparcTargetMachine.cpp	Information about target-specific, properties such as the data layout and the ABI
Sparc.td	Definition of machine features, CPU variations, and extension features
SparcAsmPrinter.cpp	Assembly code emission
SparcCallingConv.td	ABI-defined calling conventions

通常后端都遵从这样的代码组织结构，因此开发者很容易地将一个后端的具体问题映射到另一个后端中。例如，你正在编写 Sparc 后端的寄存器信息文件 SparcRegisterInfo.td，并且想知道 x86 后端是如何实现它的，你只要查看 Target/X86 文件夹中的 X86RegisterInfo.td 文件。

6.3 了解后端程序库

llc 的非共享代码是相当小的（见 tools/llc/llc.cpp），其大部分功能被实现为可重用的库，如同其它 LLVM 工具。对于 llc 的情况，它的功能由代码生成的库提供。这组程序库可分成目标相关的部分和目标无关的部分。代码生成的目标相关的库和目标无关的库在不同的文件中，这让你能够链接所期望的有限的目标后端。例如，在配置 LLVM 的时候设置 `-enable-targets=x86, arm`，这样 llc 就只会链接 x86 和 ARM 的后端程序库。

回想所有的 LLVM 程序库都以 libLLVM 为前缀。为清楚起见，我们在此省略这个前缀。下面列出了目标无关的代码生成器程序库：

- AsmParser.a：这个库包含解析汇编文本的代码，实现了一个汇编器
- AsmPrinter.a：这个库包含打印汇编语言的代码，实现了一个生成汇编文件的后端
- CodeGen.a：这个库包含代码生成算法
- MC.a：这个库包含 MCInst 类及其相关的类，用于以 LLVM 允许的最低层级表示程序
- MCDisassembler.a：这个库实现了一个反汇编器，它读取目标代码文件，将字节解码为 MCInst 对象
- MCJIT.a：这个库实现了 just-in-time（即时）代码生成器

- **MCParser.a**: 这个库包含导出 **MCAsmParser** 类的接口，用于实现解析汇编文本的组件，执行汇编器的部分工作
- **SelectionDAG.a**: 这个库包含 **SelectionDAG** 及其相关的类
- **Target.a**: 这个库包含的接口能够让目标无关的算法请求目标相关的功能，尽管此功能实质上是由其它库（目标相关部分）实现的

另一方面，下面是目标特定的程序库：

- **<Target>AsmParser.a**: 这个库包含 **AsmParser** 库的目标特定的部分，负责为目标机器实现汇编器
- **<Target>AsmPrinter.a**: 这个库包含打印目标指令的功能，让后端能够生成汇编语言文件
- **<Target>CodeGen.a**: 这个库包含后端目标相关功能的主体，包括具体的寄存器处理规则、指令选择、和调度
- **<Target>Desc.a**: 这个库包含关于低层级 MC 设施的目标机器信息，负责注册目标特定的 MC 对象，例如 **MCCodeEmitter**
- **<Target>Disassembler.a**: 这个库补足了 **MCDisassembler** 库的目标相关的功能，以建造一个能够读取字节并将它们解码成 **MCInst** 目标指令的系统
- **<Target>Info.a**: 这个库负责在 LLVM 代码生成器系统中注册目标，提供了让目标无关的代码生成器程序库能够访问目标特定功能的门面类。

在这些库的名字中，**<Target>** 必须被替换为目标名字，例如，**X86AsmParser.a** 是 X86 后端的解析程序库的名字。完整的 LLVM 安装将包含这些库，在 **<LLVM_INSTALL_PATH>/lib** 目录中。

6.4 学习 LLVM 后端如何利用 TableGen

LLVM 使用记录导向语言 **TableGen** 来描述若干编译器阶段用到的信息。例如，在第 4 章（前端）中，我们简单讨论了如何用 **TableGen** 文件（以 **.td** 为扩展名）描述前端的不同诊断信息。最初，LLVM 团队开发 **TableGen** 是为了帮助程序员编写 LLVM 后端的。尽管代码生成器程序库的设计强调清楚地分离不同的目标特性，例如，用不同的 **class** 表示寄存器信息和指令，但是最终后端程序员写出的代码不得不在若干不同的文件中表示相同的某种机器特征。这种方法的问题在于，不仅付出额外的努力编写后端代码，而且在代码中引入了信息冗余，必须手工同步。

例如，你想修改后端如何处理一个寄存器，将需要修改代码中几处不同的部分：在寄存器分配器中说明支持哪些寄存器类型；在汇编打印器中体现如何打印这个寄存器；在汇编解析器中体现它在汇编语言代码中如何解析；以及在反汇编器中，它需要知道寄存器的编码方式。这样，维护一个后端的代码变得很复杂。

为了减轻这种复杂性，人们创造了 **TableGen**，它对描述文件来说是一种声明式编程语言，这些文件成为关于目标的中央信息库。其想法是这样的：在一个单一位置声明机器的某种特性，例如在 **<Target>InstrInfo.td** 中描述机器指令，然后 **TableGen** 后端用这个信息库去达成一个具体的目的，例如生成模式匹配指令选择算法，这个算法你自己编写的话是很冗长乏味的。

如今，**TableGen** 被用于描述所有种类的目标特定的信息，如指令格式、指令、寄存器、模式匹配 DAG、指令选择匹配顺序、调用惯例、和目标 CPU 属性（支持的指令集架构（ISA）特征和处理器族）。

备注：在计算机架构研究领域，完全自动地为处理器生成后端、模拟器、和硬件综合描述文件已经成为一个长期追求的目标，并且仍然是一个开放的问题。典型的方法是用一个类似 TableGen 的声明描述语言表示所有的机器信息，然后利用工具派生出所需要的各种软件（和硬件），并评估、测试处理器架构。如同期望的那样，这是非常困难的，和手工编写的工具相比，自动生成的工具的质量不尽如人意。LLVM TableGen 的方法是辅助程序员完成较小的代码编写任务，仍然给予程序员完整的控制权，让他们用 C++ 代码来实现任意定制的逻辑。

6.4.1 语言

TableGen 语言由定义和类（class）组成，它们用于建立记录。定义 def 用于根据 class 和 multiclass 关键字实例化记录。这些记录由 TableGen 后端进一步处理，为以下部件生成域特定的信息：代码生成器、Clang 诊断、Clang 驱动器选项、和静态分析器检查器。因此，记录所表示的实际意思由后端给出，而记录仅仅存放信息。

让我们示范一个简单的例子来阐述 TableGen 如何工作。假设你想为一个假设的架构定义 ADD 和 SUB 指令，而 ADD 有以下两种形式：所有操作数都是寄存器，操作数一个是寄存器一个是立即数。

SUB 指令只有第 1 种形式。看下面 insns.td 文件的示例代码：

```
class Insn<bits <4> MajOpc, bit MinOpc> {
  bits<32> insnEncoding;
  let insnEncoding{15-12} = MajOpc;
  let insnEncoding{11} = MinOpc;
}
multiclass RegAndImmInsn<bits <4> opcode> {
  def rr : Insn<opcode, 0>;
  def ri : Insn<opcode, 1>;
}
def SUB : Insn<0x00, 0>;
defm ADD : RegAndImmInsn<0x01>;
```

Insn class 表示一个常规指令，RegAndImmInsn multiclass 表示上面所提到的形式的指令。def SUB 定义了 SUB 记录，而 defm ADD 定义了两个记录：ADDrr 和 ADDri。利用 llvm-tblgen 工具，你可以处理一个 .td 文件并检查结果记录：

```
$ llvm-tblgen -print-records insns.td
----- Classes -----
class Insn<bits<4> Insn:MajOpc = { ?, ?, ?, ? }, bit Insn:MinOpc = ?> {
  bits<5> insnEncoding = { Insn:MinOpc, Insn:MajOpc{0},
    Insn:MajOpc{1}, Insn:MajOpc{2}, Insn:MajOpc{3} };
  string NAME = ?;
}
----- Defs -----
```

(续下页)

(接上页)

```
def ADDri { // Insn ri
  bits<5> insnEncoding = { 1, 1, 0, 0, 0 };
  string NAME = "ADD";
}
def ADDrr { // Insn rr
  bits<5> insnEncoding = { 0, 1, 0, 0, 0 };
  string NAME = "ADD";
}
def SUB { // Insn
  bits<5> insnEncoding = { 0, 0, 0, 0, 0 };
  string NAME = ?;
}
```

通过 `llvm-tblgen` 工具还可使用 TableGen 后端；输入 `llvm-tblgen -help`，会列出所有后端选项。注意此例子没有用 LLVM 特定的域，它不能用于一个后端。关于 TableGen 语言的更多信息，请参考网页 <http://llvm.org/docs/TableGenFundamentals.html>。

6.4.2 了解代码生成器.td 文件

如前所述，代码生成器广泛地使用 TableGen 记录来表达目标特定的信息。在这个子小节，我们来浏览一下致力于代码生成的 TableGen 文件。

目标属性

<Target>.td 文件（例如，X86.td）定义了所支持的 ISA 特性和处理器家族。例如，X86.td 定义了 AVX2 扩展：

```
def FeatureAVX2 : SubtargetFeature<" avx2" , "X86SSELevel" , "AVX2" ,
                                   "Enable AVX2 instructions" ,
                                   [FeatureAVX]>;
```

关键字 `def` 通过 `class` 类型 `SubtargetFeature` 定义了记录 `FeatureAVX2`。最后一个参数是已经包含在定义中的其它特性的一个列表。因此，一个具有 AVX2 的处理器包含所有 AVX 指令。

此外，我们还可以定义一个处理器类型，包含它所能提供的 ISA 扩展和特性：

```
def : ProcessorModel<" corei7-avx" , SandyBridgeModel,
                    [FeatureAVX, FeatureCMPXCHG16B, ...,
                     FeaturePCLMUL]>;
```

<Target>.td 文件还包含了所有其它的.td 文件，是描述目标特定域信息的主文件。`llvm-tblgen` 工具必须总是从它那获得一个目标任意的 TableGen 记录。例如，为了输出 x86 所有可能的记录，执行下面的命令：

```
$ cd <llvm_source>/lib/Target/X86
$ llvm-tblgen -print-records X86.td -I ../../../../include
```

X86.td 文件含有 TableGen 用以生成 X86GenSubtargetInfo.inc 文件的部分信息，但是所用的信息不限于此，一般来说，没有从一个.td 文件到一个.inc 文件的直接映射。为了理解这个表述，考虑 <Target>.td 文件是一个重要的顶层文件，它利用 TableGen 的 include 指令包含了所有其它的.td 文件。因此，当生成 C++ 代码时，TableGen 总是解析所有的后端.td 文件，这意味着你可以自由地将记录放到任意你觉得最合适的地方。即使 X86.td 包含了所有其它的后端.td 文件，除了 include 指令，这个文件的内容也是和 Subtarget x86 子类的定义保持一致的。

如果你查看实现 x86Subtarget 类的 X86Subtarget.cpp 文件，你会发现一个 C++ 预处理器指令，” #include “X86GenSubtargetInfo.inc”，这揭示了我们如何在常规的 code base 中嵌入 TableGen 生成的 C++ 代码。这个特别的 include 文件包含了处理器特性常量，处理器特性向量——它关联了特性和它的字符串描述，以及其它相关的资源。

寄存器

寄存器和寄存器类在 <Target>RegisterInfo.td 文件中定义。寄存器类用于在之后的指令定义中连结指令操作数和特定的寄存器集合。例如，X86RegisterInfo.td 用下面的语句定义了 16 位的寄存器：

```
let SubRegIndices = [sub_8bit, sub_8bit_hi], ... in {
def AX : X86Reg<"ax", 0, [AL,AH]>;
def DX : X86Reg<"dx", 2, [DL,DH]>;
def CX : X86Reg<"cx", 1, [CL,CH]>;
def BX : X86Reg<"bx", 3, [BL,BH]>;
...
}
```

此处 let 构建指令用于定义一个额外的字段 SubRegIndices，在以 { 开始并以 } 结束的区域中的所有记录都会存放这个字段。16 位的寄存器的定义是从 X86Reg 类派生而来的，它为每个寄存器保存它的名字、数目、和一个 8 位的子寄存器的列表。16 位寄存器的寄存器类的定义被重新产生，如下所示：

```
def GR16 : RegisterClass<"X86", [i16], 16,
    (add AX, CX, DX, ..., BX, BP, SP,
     R8W, R9W, ..., R15W, R12W, R13W)>;
```

GR16 寄存器类包含所有的 16 位寄存器和它们各自的寄存器分配首选的顺序。在 TableGen 处理之后，每个寄存器类的名字会得到后缀 RegClass，例如，GR16 变成了 GR16RegClass。TableGen 会生成寄存器和寄存器类的定义，收集它们的相关信息的方法，汇编器的二进制编码，和它们的 DWARF（Linux 调试记录格式）信息。你可以用 llvm-tblgen 查看 TableGen 生成的代码：

```
$ cd <llvm_source>/lib/Target/X86
$ llvm-tblgen -gen-register-info X86.td -I ../../../../include
```

可选地，你可以查看 LLVM 编译过程中生成的 C++ 文件 <LLVM_BUILD_DIR>/lib/Target/X86/X86GenRegisterInfo.inc。

这个文件被 `X86RegisterInfo.cpp` 包含，辅助它定义 `X86RegisterInfo` 类。其中包含了处理器寄存器的枚举，当你在调试你的后端并且不知道数字 16 表示什么寄存器（这是你的调试器所能给你的最好的猜测）的时候，这个文件是一份有用的参考指引。

指令

指令格式和指令分别在 `<Target>InstrFormats.td` 和 `<Target>InstInfo.td` 文件中被定义。指令格式包含所必需的指令编码字段，用于写二进制形式的指令，而指令记录表示指令，一条记录表示一条指令。你可以创建中间指令类，就是用于派生指令记录的 `TableGen` 类，以析出公共特性因子，例如相似的数据处理指令的共同编码方式。然而，每个指令或者格式必须是 `Instruction TableGen` 类的直接或间接的子类，`Instruction` 类在 `include/llvm/Target/Target.td` 中被定义。它的字段显示了 `TableGen` 后端期望在指令记录中找到什么内容：

```
class Instruction {
    dag OutOperandList;
    dag InOperandList;
    string AsmString = "";
    list<dag> Pattern;
    list<Register> Uses = [];
    list<Register> Defs = [];
    list<Predicate> Predicates = [];
    bit isReturn = 0;
    bit isBranch = 0;
    ...
}
```

`dag` 是一种特别的 `TableGen` 类型，用于存放 `SelectionDAG` 节点。这些节点表示指令选择过程的操作符、寄存器、或常量。代码中出现的字段扮演如下角色：

- `OutOperandList` 字段存储结果节点，让后端能够找到代表指令输出的 `DAG` 节点。例如，在 MIPS `ADD` 指令中，这个字段被定义为 `(outs GPR32Opnd:$rd)`。在此例中：
- `outs` 是一个特别的 `DAG` 节点，用于指示它的孩子是输出操作数
- `GPR32Opnd` 是一个 MIPS 专有的 `DAG` 节点，用于指示 MIPS 32 位的通用寄存器的一个实例
- `$rd` 是一个任意的寄存器名字，用于识别节点
- `InOperandList` 字段存放输入节点，例如，在 MIPS `ADD` 指令中，它是 `(ins GRP32Opnd:$rs, GRP32Opnd:$rt)`。
- `AsmString` 字段表示指令的汇编字符串，例如，在 MIPS `ADD` 指令中，它是 `"add $rd, $rs, $rt"`。
- `Pattern` 是 `dag` 对象的列表，它将在指令选择时被用于执行模式匹配。如果一个模式被匹配了，指令选择过程会用这条指令替换匹配的节点。例如，在 MIPS `ADD` 指令的 `[(set GPR32Opnd:$rd, (add GPR32Opnd:$rs, GPR32Opnd:$rt))]` 模式中，`[and]` 表示只有一个 `dag` 元素的列表的内容，它是在类似 LISP 表示法的小括号之间被定义的。
- `Uses` 和 `Defs` 记录在指令执行期间被隐式使用和定义的寄存器的列表。例如，RISC 处理器的 `return` 指令隐式地返回地址寄存器，而 `call` 指令隐式地定义返回地址寄存器。

- `Predicates` 字段存储在指令选择尝试匹配指令之前要检查的先决条件的列表。如果检查失败了，就没有匹配。例如，一个 `predicate` 可能说明这个指令只对特定的子目标有效。如果你所运行代码生成器的目标三元组选择了另一个子目标，这个 `predicate` 会评估为假，这个指令永远不会匹配。
- 此外，其它的字段包括 `isReturn` 和 `isBranch`，它们向代码生成器说明关于指令行为的信息。例如，如果 `isBranch = 1`，代码生成器就知道这个指令是分支指令，必须处在一个基本块的末尾。

在下面的代码块中，我们看到在 `SparcInstrInfo.td` 中的 `XNORrr` 指令的定义。它用到了 `F3_1` 格式（在 `SparcInstrFormats.td` 中被定义），它包括了来自 `SPARC V8` 架构手册的 `F3` 格式的一部分：

```
def XNORrr : F3_1<2, 0b000111,
  (outs IntRegs:$dst), (ins IntRegs:$b, IntRegs:$c),
  "xnor $b, $c, $dst",
  [(set i32:$dst, (not (xor i32:$b, i32:$c)))]>;
```

这个 `XNORrr` 指令有两个 `IntRegs`（一个表示 `SPARC 32` 位整数寄存器类的目标特定的 `DAG` 节点）源操作数和一个 `IntRegs` 结果，也就是 `OutOperandList = (outs IntRegs:$dst)` 和 `InOperandList = (ins IntRegs:$b, IntRegs:$c)`。

`AsmString` 汇编通过 `$` 记号引用指定的操作数：“`xnor $b, $c, $dst`”。`Pattern` 列表元素包含那个 `SelectionDAG` 节点，它应该被匹配到这个指令。举例来说，每当 `xor` 的结果由一个 `not` 反转位，并且 `xor` 的两个操作数都是寄存器的时候，`XNORrr` 指令被匹配。

为了查看 `XNORrr` 指令记录的字段，你可以执行下面的命令序列：

```
$ cd <llvm_sources>/lib/Target/Sparc
$ llvm-tblgen -print-records Sparc.td -I ../../../../include | grep XNORrr -A 10
```

多个 `TableGen` 后端利用指令记录的信息以履行它们的职能，从相同的指令记录生成不同的 `.inc` 文件。这跟 `TableGen` 的目标是一致的，即创建一个中央仓库，利用它给后端的各个部分生成代码。下面的每个文件是由不同的 `TableGen` 后端生成的：

- `<Target>GenDAGISel.inc`：这个文件利用指令记录中 `patterns` 字段的信息以输出选择 `SelectionDAG` 数据结构的指令的代码。`<Target>ISelDAGtoDAG.cpp` 文件包含这个文件。
- `<Target>GenInstrInfo.inc`：这个文件包含列出目标的所有指令的枚举，除了其它的描述指令的表。`<Target>InstrInfo.cpp`、`<Target>InstrInfo.h`、`<Target>MCTargetDesc.cpp`、和 `<Target>MCTargetDesc.h` 包含这个文件。然而，每个文件会在包含 `TableGen` 生成的文件之前定义一组特定的宏，改变了这个文件如何在每个上下文中被解析和使用。
- `<Target>GenAsmWriter.inc`：这个文件包含映射字符串的代码，该字符串被用来打印每个指令的汇编。`<Target>AsmPrinter.cpp` 文件包含这个文件。
- `<Target>GenCodeEmitter.inc`：这个文件包含这样的函数，它们为每条指令映射并输出二进制代码，从而生成机器代码以填写目标文件。`<Target>CodeEmitter.cpp` 包含这个文件。
- `<Target>GenDisassemblerTables.inc`：这个文件实现这样的表和算法，它们能够解码一个字节序列并识别出它表示的目标指令。它用于实现反汇编工具，`<Target>Disassembler.cpp` 文件包含它。

- `<Target>GenAsmMatcher.inc`: 这个文件实现目标指令的汇编器的解析器。`<Target>AsmParser.cpp` 文件包含它两次，每次都有一组不同的预处理宏，从而改变如何解析这个文件。

6.5 理解指令选择过程

指令选择是将 LLVM IR 转换为代表目标指令的 SelectionDAG 节点 (SDNode) 的过程。第一步是根据 LLVM IR 指令建立 DAG，创建 SelectionDAG 对象，其节点保存 IR 操作。接着，这些节点经过低层化、DAG 结合、和合法化等过程，使它更容易匹配目标指令。然后，指令选择用节点模式匹配方法执行 DAG 到 DAG 的变换，将 SelectionDAG 节点转换为代表目标指令的节点。

备注： 指令选择是其中一个最耗时的后端 Pass。一项编译 SPEC CPU2006 基准测试的函数的研究揭示，在 LLVM 3.0 中，以 -O2 运行 llc 工具，平均来说，指令选择 Pass 几乎花去一半的时间。如果你有兴趣了解所有 -O2 级别的目标无关和目标有关的 Pass 的平均占用时间，你可以查看 LLVM JIT 编译成本分析的技术报告的附件：<http://www.ic.unicamp.br/~reltech/2013/13-13.pdf>。

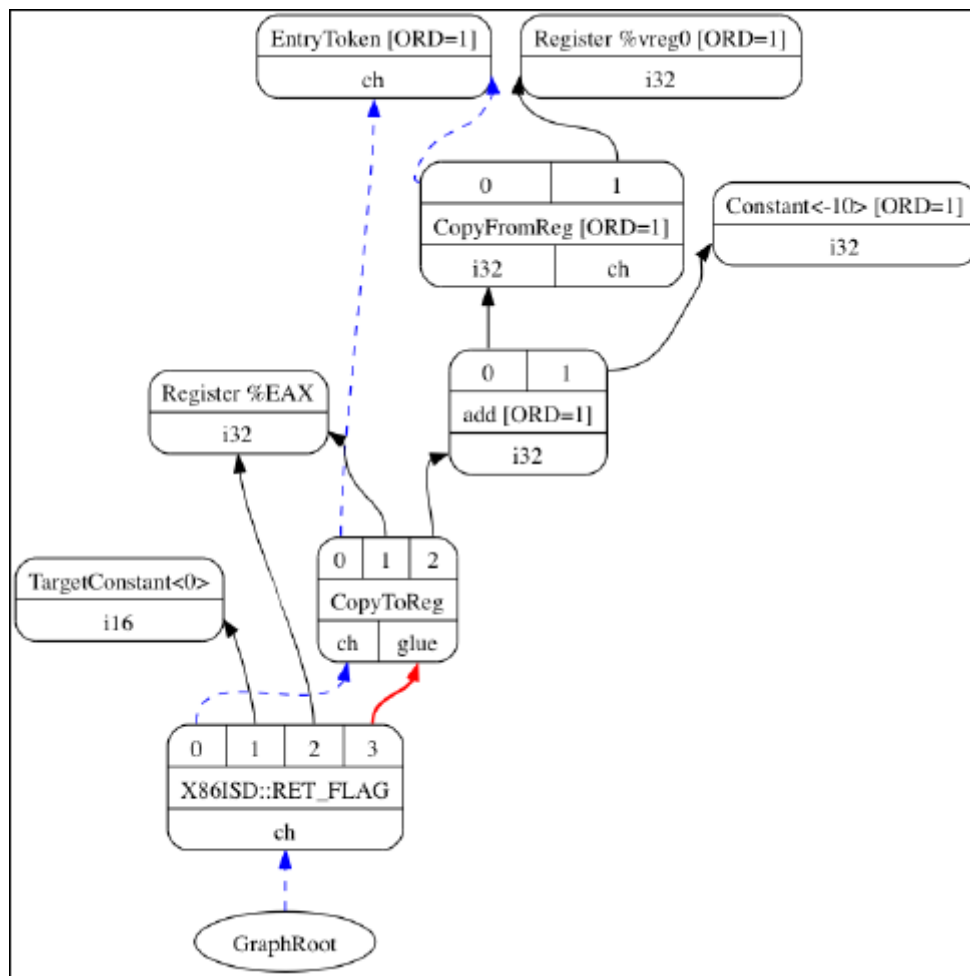
6.5.1 SelectionDAG 类

SelectionDAG 类 (class) 用一个 DAG 表示每个基本块的计算，每个 SDNode 对应一个指令或者操作数。下图由 LLVM 生成，展示了 `sum.bc` 的 DAG，它只有一个函数和一个基本块：

DAG 的连线 (edge) 通过 use-def 关系强制它的操作之间的顺序。如果节点 B (例如，`add`) 有一条出去的连线到节点 A (例如，`Constant<-10>`)，这意味着节点 A 定义了一个值 (32 位整数 -10)，而节点 B 使用它 (作为加法的一个操作数)。因此，A 操作必须在 B 之前执行。黑色箭头表示常规连线，指示数据流依赖，正如例子 `add`。虚线蓝色箭头表示非数据流链，用以强制两条指令的顺序，否则它们就是不相关的，例如，`load` 和 `store` 指令必须固定它们原始的程序顺序，如果它们访问相同的内存位置。在前面的图中，我们知道 `CopyToReg` 操作必须在 `X86ISD::RET_FLAG` 之前发生，由于虚线蓝色箭头。红色连线保证它相邻的节点必须粘合在一起，这意味着它们必须紧挨着执行，它们之间不可有其它指令。例如，我们指定相同的节点 `CopyToReg` 和 `X86ISD::RET_FLAG` 必须安排为紧挨着，由于红色的连线。

每个节点可以提供一个不同的值类型，依赖于它和它的使用者的关系。一个值不必是具体的，也可能是一个抽象的标记 (token)。它可能有任意如下类型：

- 节点所提供的值可以是一个具体的值类型，表示整数、浮点数、向量、或指针。数据处理节点根据它的操作数计算一个新的值，其结果是这种类别的一个例子。类型可以是 `i32`、`i64`、`f32`、`v2f32` (有两个 `f32` 元素的向量)、和 `iPTR` 等。在 LLVM 示意图中，当另一个节点使用这个值的时候，生产者-消费者关系是由一条常规的黑色连线描绘的。
- `Other` 类型是一个抽象的标记，用于表示链值 (示意图中的 `ch`)。在 LLVM 示意图中，当另一个节点使用一个 `Other` 类型的值的时候，连接两者的连线被打印为蓝色的虚线。
- `Glue` 类型表示粘合。在 LLVM 示意图中，当另一个节点使用一个 `Glue` 类型的值的时候，连接两者的连线被画成红色。



SelectionDAG 对象有一个特别的标记基本块入口的 `EntryToken`，它提供一个 `Other` 类型的值，让成链的节点能够以它为起点。SelectionDAG 对象还会引用图的根节点，这个根节点正好是最后一条指令的后续节点，它们的关系也被编码为 `Other` 类型的值的一个链。

在这个阶段，目标无关和目标特定的节点可以同时存在，这是执行预备步骤的结果，例如低层化和合法化，这些预备步骤负责为指令选择准备 DAG。然而，等到指令选择结束的时候，所有被目标指令匹配的节点都会是目标特定的。在前面的示意图中，有如下目标无关的节点：`CopyToReg`，`CopyFromReg`，`Register (%vreg0)`，`add`，和 `Constant`。此外，有如下已经被预处理并且是目标特定的节点（尽管它们在指令选择之后仍然可以改变）：`TargetConstant`，`Register (%EAX)`，和 `X86ISD::REG_Flag`。

在示意图所示的例子中，我们可能观察到下面的语义：

- **Register**：这个节点可能引用虚拟或者（目标特定的）物理的寄存器。
- **CopyFromReg**：这个节点复制一个在当前基本块作用域之外定义的寄存器——在我们的例子中，它复制了一个函数的参数。
- **CopyToReg**：这个节点将一个值复制到一个特定的寄存器，可是不提供任何具体的值让其它节点使用它。然而，这个节点产生一个链的值（`Other` 类型），和不生成具体的值的其它节点构成链。举例来说，为了使用被写到 `EAX` 的值，`X86ISD::RET_FLAG` 节点使用由 `Register (%EAX)` 提供的 `i32` 结果，并且还接收由 `CopyToReg` 产生的链，这样确保 `%EAX` 是通过 `CopyToReg` 更新了的，因为这个链会强制 `CopyToReg` 被安排在 `X86ISD::RET_FLAG` 之前。

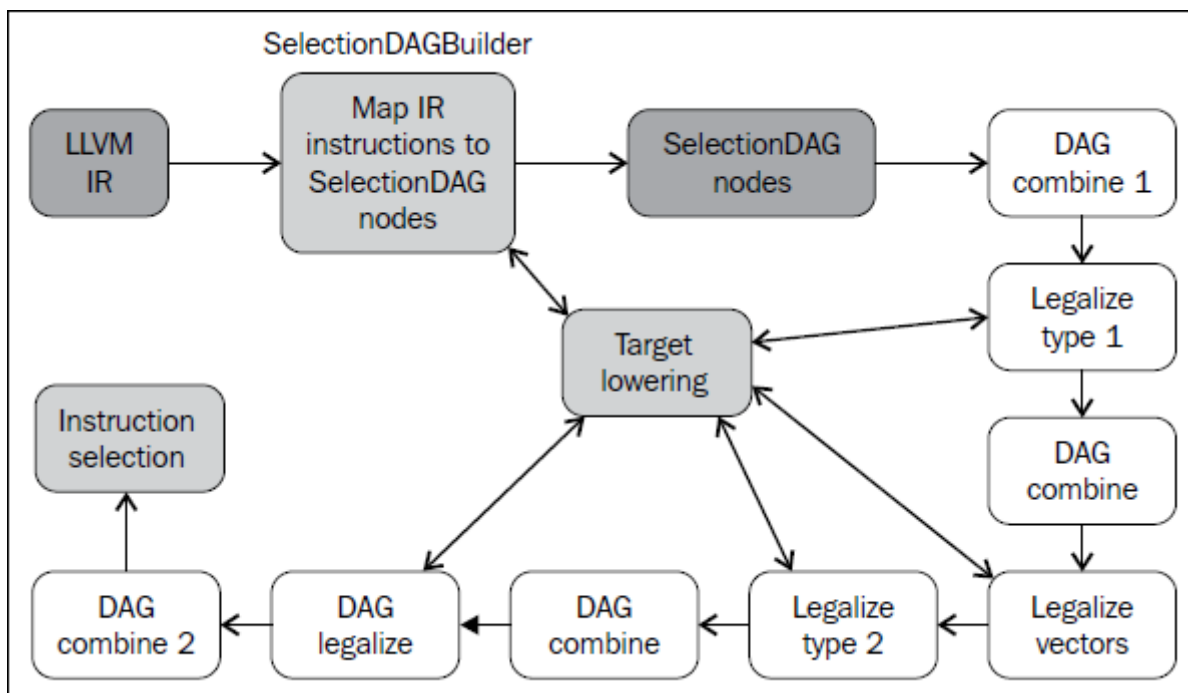
想要深入了解 SelectionDAG 类的细节，请参考 `llvm/include/llvm/CodeGen/SelectionDAG.h` 头文件。对于节点的结果类型，你应该参考 `llvm/include/llvm/CodeGen/ValueTypes.h` 头文件。头文件 `llvm/include/llvm/CodeGen/ISDOpcodes.h` 定义了目标无关的节点，而头文件 `lib/Target/<Target>/<Target>ISelLowering.h` 定义了目标特定的节点。

6.5.2 低层化

在前面的子小节中，我们展示的图中目标特定的和目标无关的节点是并存的。你可能会问自己，一些目标特定的节点怎么已经在 SelectionDAG 中了，如果这是指令选择的输入？为了理解这个问题，我们首先在下图中给出所有先于指令选择的步骤的全局图，在左上角从 LLVM IR 步骤开始：

首先，一个 `SelectionDAGBuilder` 实例（详情见 `SelectionDAGISel.cpp`）访问每个函数，为每个基本块创建一个 SelectionDAG 对象。在此过程期间，一些特殊的 IR 指令例如 `call` 和 `ret` 已经要求目标特定的语句——例如，如何传递调用参数和如何从一个函数返回——被转换为 SelectionDAG 节点。为了解决这个问题，`TargetLowering` class 中的算法第一次被使用。这个 class 是每个目标都必须实现的抽象接口，但是还有大量共用的功能被所有后端所使用。

为了实现这个抽象接口，每个目标声明一个 `TargetLowering` 的子类，命名为 `<Target>TargetLowering`。每个目标还重载方法，它们实现一个具体的目标无关的高层次的节点应该如何被低层化到一个层次，它接近这个机器的节点。如期望那样，仅有小部分节点必须以这种方式低层化，而大部分其它节点在指令选择时被匹配和替换。例如，在 `sum.bc` 的 SelectionDAG 中，用 `X86TargetLowering::LowerReturn()` 方法（参见 `lib/Target/X86/X86ISelLowering.cpp`）低层化 `ret` IR 指令。同时，生成了 `X86ISD::RET_FLAG` 节点，它将函数结果复制到 `EAX`——一种处理函数返回的目标特定的方式。



6.5.3 DAG 结合与合法化

从 SelectionDAGBuilder 输出的 SelectionDAG 并不能直接作指令选择，必须经历附加的转换——如前面图中所显示的。先于指令选择执行的 Pass 序列如下：

- DAG 结合 Pass 优化欠优化的 SelectionDAG 结构，通过匹配一系列节点并用简化的结构替换它们，当可获利时。例如，子图 (add (Register X), (constant 0)) 可以合并为 (Register X)。类似地，目标特定的结合方法可以识别节点模式，并决定结合合并它们是否将提高此目标的指令选择的质量。你可以在 lib/CodeGen/SelectionDAG/DAGCombiner.cpp 文件中找到 LLVM 通用的 DAG 结合的实现，在 lib/Target/<Target_Name>/<Target>ISelLowering.cpp 文件中找到目标特定的结合的实现。方法 setTargetDAGCombine() 标记目标想要结合的节点。举例来说，MIPS 后端尝试结合加法——见 lib/Target/Mips/MipsISelLowering.cpp 中的 setTargetDAGCombine(ISD::ADD) 和 performADDCombine()。

备注：DAG 结合在每次合法化之后运行，以最小化任何 SelectionDAG 冗余。而且，DAG 结合知道在 Pass 链的何处运行，（例如在类型合法化或者向量合法化之后），能够运用这些信息以变得更精确。

- 类型合法化 Pass 确保指令选择只需要处理合法的类型。合法的类型是指目标天然地支持的类型。例如，在只支持 i32 类型的目标上，i64 操作数的加法是非法的。在这种情况下，类型合法化动作整数展开把 i64 操作数破分为两个 i32 操作数，同时生成合适的节点以操作它们。目标定义了每种类型所关联的寄存器，显式地声明了支持的类型。这样，非法的类型必须被删除并相应地处理：标量类型可以被提升，展开，或者软件化，而向量类型可以被分解，标量化，或者放宽——见 llvm/include/llvm/Target/TargetLowering.h 对每种情况的解释。此外，目标还可以设置定制的方法来合法化类型。类型合法化运行两次，在第一次 DAG 结合之后和在向量合法化之后。

- 有的时候，后端直接支持向量类型，这意味着有一个这样的寄存器类，但是没有处理给定向量类型的具体的操作。例如，x86 的 SSE2 支持 v4i32 向量类型。然而，并没有 x86 指令支持 v4i32 类型的 ISD::OR 操作，而只有 v2i64 的。因此，向量合法化会处理这种情况，提升或者扩展操作，为指令使用合法的类型。目标还可以通过定制的方式处理合法化。对于前面提到的 ISD::OR，操作会被提升而使用 v2i64 类型。看一看下面的 lib/Target/X86/X86ISelLowering.cpp 的代码片段：

```
setOperationAction(ISD::OR, v4i32, Promote);
AddPromotedToType (ISD::OR, v4i32, MVT::v2i64);
```

备注：对于某些类型，扩展会消除向量而使用标量。这可能引入目标不支持的标量类型。然而，后续的类型合法化会清理这种情况。

- DAG 合法化扮演向量合法化一样的角色，但是它处理任意剩余的具有不支持的类型（标量或向量）的操作。它支持相同的动作：提升、扩展、和定制节点的处理。举例来说，x86 不支持以下三种情形：i8 类型的有符号整数到浮点数的转化操作（ISD::SINT_TO_FP），请求合法化提升它；i32 操作数的有符号除法（ISD::SDIV），发起一个扩展请求，产生一个库调用以处理这个除法；f32 操作数的浮点数绝对值，利用定制的句柄生成具有相同效果的等价的代码。x86 以如下方式发起这些动作（参见 lib/Target/X86/X86ISelLowering.cpp）：

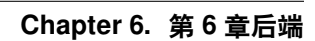
```
setOperationAction(ISD::SINT_TO_FP, MVT::i8, Promote);
setOperationAction(ISD::SDIV, MVT::i32, Expand);
setOperationAction(ISD::FABS, MVT::f32, Custom);
```

6.5.4 DAG 到 DAG 的指令选择

DAG 到 DAG 的指令选择的目的，是利用模式匹配将目标无关的节点转换为目标特定的节点。指令选择的算法是局部的，每次作用 SelectionDAG（基本块）的实例。

作为例子，后面给出了指令选择之后我们最终的 SelectionDAG 结构。CopyToReg、CopyFromReg、和 Register 节点保持不变，直到寄存器分配。实际上，指令选择过程甚至可能增加节点。指令选择之后，ISD::ADD 节点被转换为 X86 指令 ADD32ri8，X86ISD::RET_FLAG 变为 RET。

备注：注意，三种指令表示类型可能在同一个 DAG 中并存：通用的 LLVM ISD 节点比如 ISD::ADD，目标特定的 <Target>ISD 节点比如 X86ISD::REG_FLAG，目标物理指令比如 X86::ADD32ri8。



模式匹配

每个目标都有 `SelectionDAGISel` 子类，命名为 `<Target_Name>DAGToDAGISel`。它通过实现子类的 `Select` 方法来处理指令选择。例如 SPARC 中的 `SparcDAGToDAGISel::Select()`（参见 `lib/Target/Sparc/SparcISelDAGToDAG.cpp` 文件）。这个方法接收将要被匹配的 `SDNode` 参数，返回一个代表物理指令的 `SDNode` 值；否则发生一个错误。

`Select()` 方法允许用两种方式来匹配物理指令。最直接的方式是调用产生自 TableGen 模式的匹配代码，如下面列表中的步骤一。然而，模式可能表达不够清楚，使得有些指令的奇怪行为不能被处理。这种情况下，必须在这个方法中实现定制的 C++ 匹配逻辑，如下面列表中的步骤二。下面详细介绍这两种方式：

1. `Select()` 方法调用 `SelectCode()`。TableGen 为每个目标生成 `SelectCode()` 方法，在此代码中，TableGen 还生成 `MatcherTable`，它将 ISD 和 `<Target>ISD` 映射为物理指令节点。这个匹配器表是从 `.td` 文件（通常为 `<Target>InstrInfo.td`）中的指令定义生成的。`SelectCode()` 方法以调用 `SelectCodeCommon()` 结束，这是一个目标无关的方法，它根据目标的匹配器表匹配节点。TableGen 有一个专门的指令选择后端，用以生成这些方法和表：

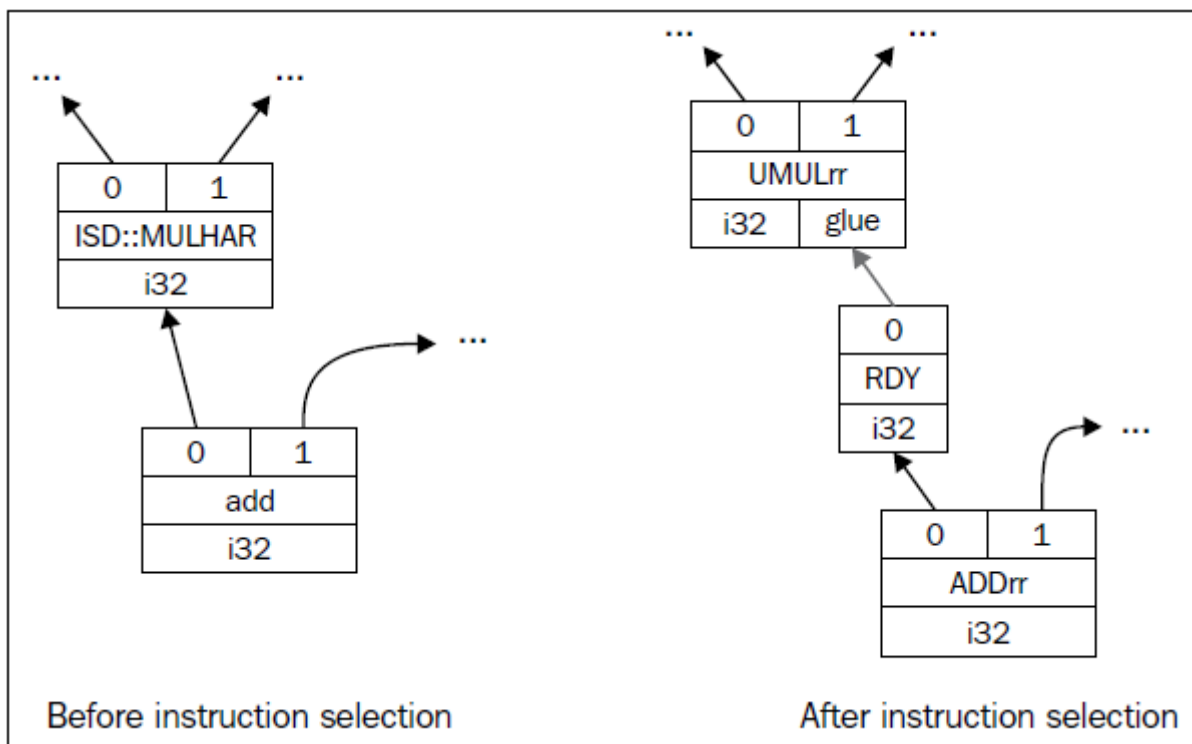
```
$ cd <llvm_source>/lib/Target/Sparc
$ llvm-tblgen -gen-dag-isel Sparc.td -I ../../../../include
```

为每个目标的输出在 `<build_dir>/lib/Target/<Target>/<Target>GenDAGISel.inc` C++ 文件中；例如，在 SPARC 中，可在 `<build_dir>/lib/Target/Sparc/SparcGenDAGISel.inc` 文件中获得这些方法和表。

2. `Select()` 方法中在 `SelectCode` 调用前提供定制的匹配代码。例如，i32 节点 `ISD::MULHU` 执行两个 i32 的乘，产生一个 i64 结果，并返回高 i32 部分。在 32 位 SPARC 上，乘法指令 `SP::UMULrr` 在特殊寄存器 Y 中返回高位部分，它需要由 `SP::RDY` 指令读取它。TableGen 无法表达这个逻辑，但是我们可以用下面的代码解决这个问题：

```
case ISD::MULHU: {
    SDValue MulLHS = N->getOperand(0);
    SDValue MulRHS = N->getOperand(1);
    SDNode *Mul = CurDAG->getMachineNode(SP::UMULrr, dl, MVT::i32, MVT::Glue, MulLHS,
    ↪MulRHS);
    return CurDAG->SelectNodeTo(N, SP::RDY, MVT::i32, SDValue(Mul, 1));
}
```

这里，`N` 是待匹配的 `SDNode` 参数，在此上下文中，`N` 等于 `ISD::MULHU`。因为在这个 `case` 语句之前已经作了细致的检查，这里生成 SPARC 特定的 `opcode` 以替换 `ISD::MULHU`。为此，我们通过调用 `CurDAG->getMachineNode()` 以 `SP::UMULrr` 创建一个物理指令节点。接着，通过 `CurDAG->SelectNodeTo()`，我们创建一个 `SP::RDY` 指令节点，并将指向 `ISD::MULHU` 的结果的所有 `use`（引用）改变为指向 `SP::RDY` 的结果。下图显示了这个例子指令选择前后的 `SelectionDAG` 结构。前面的 C++ 代码片段是 `lib/Target/Sparc/SparcISelDAGToDAG.cpp` 中的代码的简化版本。



6.5.5 可视化指令选择过程

若干 llc 的选项可以在不同的指令选择过程可视化 SelectionDAG。如果你使用了这些选项中的任意一个，llc 将生成一个 .dot 图，类似于本章早前展示的那样，但是你需要用 dot 程序来显示它，或者用 dotty 编辑它。你可以在 www.graphviz.org 的 Graphviz 包中找到它们。下图按照执行的顺序列出了每个选项：

llc 选项	过程
-view-dag-combine1-dags	DAG 结合-1 之前
-view-legalize-types-dags	类型合法化之前
-view-dag-combine-lt-dags	类型合法化-2 之后 DAG 结合之前
-view-legalize-dags	合法化之前
-view-dag-combine2-dags	DAG 结合-2 之前
-view-isel-dags	指令选择之前
-view-sched-dags	指令选择之后指令调度之前

6.5.6 快速指令选择

LLVM 还支持可选的指令选择实现，称为快速指令选择（FastISel class，位于 `<llvm_source>/lib/CodeGen/SelectionDAG/FastISel.cpp` 文件）。快速指令选择的目标是快速生成代码，以损失代码质量为代价，它适合-O0 优化级别的编译哲学。通过省略复杂的合并和低级化逻辑，编译得到提速。TableGen 描述也被用于简单的操作，但是更复杂的指令匹配需要目标特定的代码来处理。

备注：-O0 管线编译还用到了快速但非优化的寄存器分配器和调度器，以代码质量换取编译速度。我们将在下一个子小节阐述它们。

6.6 调度

指令选择之后，SelectionDAG 结构的节点表示了物理指令——处理器直接支持它们。下一个阶段是前寄存器分配调度器，工作在 SelectionDAG 节点（SDNode）之上。有几个不同的调度器可供选择，它们都是 ScheduleDAGSDNodes 的子类（见文件 `<llvm_source>/lib/CodeGen/SelectionDAG/ScheduleDAGSDNodes.cpp`）。在 llc 工具中可以通过 `-pre-RA-sched=<scheduler>` 选项选择调度器类型。可能的 `<scheduler>` 值如下：

- `list-ilp`, `list-hybrid`, `source`, 和 `list-burr`: 这些选项指定表调度算法，它由 ScheduleDAGRRList class 实现（见文件 `<llvm_source>/lib/CodeGen/SelectionDAG/ScheduleDAGRRList.cpp`）。
- `fast`: ScheduleDAGFast class（`<llvm_source>/lib/CodeGen/SelectionDAG/ScheduleDAGFast.cpp`）实现了一个非优化但快速的调度器。
- `view-td`: 一个 VLIW 特定的调度器，由 ScheduleDAGVLIW class 实现（见文件 `<llvm_source>/lib/CodeGen/SelectionDAG/ScheduleDAGVLIW.cpp`）。

`default` 选项为目标选择一个预定义的最佳的调度器，而 `linearize` 选项不作调度。可获得的调度器可能使用指令行程表和 risk识别器的信息，以更好地调度指令。

备注：在代码生成器中有三个不同的调度器：两个在寄存器分配之前，一个在寄存器分配之后。第一个工作在 SelectionDAG 节点之上，而其它两个工作在机器指令之上，本章将进一步解释它们。

6.6.1 指令延迟表

有些目标提供了指令行程表，表示指令延迟和硬件管线信息。调度器在作调度决策时利用这些属性以最大化吞吐量，避免性能处罚。这些信息由每个目标目录中的 TableGen 文件，通常命名为 `<Target>Schedule.td`（例如 `X86Schedule.td`）。

LLVM 提供了 ProcessorItineraries TableGen class，在 `<llvm_source>/include/llvm/Target/TargetItinerary.td`，如下：

```
class ProcessorItineraries<list<FuncUnit> fu, list<Bypass> bp,
list<InstrItinData> iid> {
    ...
}
```

目标可能为一个芯片或者处理器家族定义处理器行程表。要描述它们，目标必须提供函数单元（FuncUnit）列表、管线支路（Bypass）、和指令行程数据（InstrItinData）。例如，ARM Cortex A8 指令的行程表在 `<llvm_source>/lib/Target/ARM/ARMScheduleA8.td`，如下

```
def CortexA8Itineraries : ProcessorItineraries<
  [A8_Pipe0, A8_Pipe1, A8_LSPipe, A8_NPipe, A8_NLSPipe],
  [], [
    ...
    InstrItinData<IIC_iALUi , [InstrStage<1, [A8_Pipe0, A8_Pipe1]>], [2, 2]>,
    ...
  ]>;
```

这里，我们没有看到支路（bypass）。我们看到了这个处理器的函数单元列表（A8_Pipe0, A8_Pipe1 等），以及来自类型 IIC_iALUi 的指令行程数据。这种类型是形如 `reg = reg + immediate` 的二元运算指令的 class，例如 ADDri 和 SUBri 指令。这些指令的执行时间是一个机器时钟周期，以完成 A8_Pipe0 和 A8_Pipe1 函数单元，如 InstrStage<1, [A8_Pipe0, A8_Pipe1] 定义的那样。

后面，列表 [2, 2] 表示指令发射之后读取或者定义每个操作数所用的时钟周期。此处，目标寄存器（index 0）和源寄存器（index 1）都在 2 个时钟周期之后可用。

6.6.2 风险检测

风险识别器利用处理器指令行程表的信息计算风险。ScheduleHazardRecognizer class 为风险识别器的实现提供了接口，ScoreboardHazardRecognizer subclass 实现了记分牌风险识别器（见文件 `<llvm_source>/lib/CodeGen/ScoreboardHazardRecognizer.cpp`），它是 LLVM 的默认识别器。

目标提供自己的识别器是允许的。这是必需的，因为 TableGen 可能无法表达具体的约束，这时必须提供定制的实现。例如，ARM 和 PowerPC 都提供了 ScoreboardHazardRecognizer subclass。

6.6.3 调度单元

调度器在寄存器分配之前和之后运行。然而，只有前者可使用 SDNode 指令表示，而后者使用 MachineInstr class。为了兼顾 SDNode 和 MachineInstr，SUnit class（见文件 `<llvm_source>/include/llvm/CodeGen/ScheduleDAG.h`）抽象了背后的指令表示，作为指令调度期间的单元。llc 工具可以用选项 `-view-sunit-dags` 输出调度单元。

6.7 机器指令

寄存器分配器工作在一种由 `MachineInstr` class (简称 `MI`) 给出的指令表示之上，它的定义在 `<llvm_source>/include/llvm/CodeGen/MachineInstr.h`。在指令调度之后，`InstrEmitter` Pass 会被运行，它将 `SDNode` 格式转换为 `MachineInstr` 格式。如名字的含义，这种表示比 `IR` 指令更接近实际的目标指令。与 `SDNode` 格式及其 DAG 形式不同，`MI` 格式是程序的三地址表示，即指令的序列而不是 DAG，这让编译器能够高效地表达一个具体的调度决定，也就是决定每个指令的顺序。每个 `MI` 有一个操作码 (opcode) 数字和几个操作数，操作码只对一个具体的后端有意义。

利用 `llc` 选项 `-print-machineinstrs`，可以输出所有注册的 Pass 之后的机器指令，或者利用选项 `-print-machineinstrs=<pass-name>` 输出一个特定的 Pass 之后的机器指令。我们从 LLVM 源代码中查找这些 Pass 的名字。为此，进入 LLVM 源代码文件夹，运行 `grep` 查找 Pass 注册它们的名字时常用到的宏：

```
$ grep -r INITIALIZE_PASS_BEGIN * CodeGen/
PHIElimination.cpp:INITIALIZE_PASS_BEGIN(PHIElimination, "phi-node-elimination"
(...)
```

例如，看下面 `sum.bc` 的每个 Pass 之后的 SPARC 机器指令：

```
$ llc -march=sparc -print-machineinstrs sum.bc
Function Live Ins: %I0 in %vreg0, %I1 in %vreg1
BB#0: derived from LLVM BB %entry Live Ins: %I0 %I1
%vreg1<def> = COPY %I1; IntRegs: %vreg1
%vreg0<def> = COPY %I0; IntRegs: %vreg0
%vreg2<def> = ADDrr %vreg1, %vreg0; IntRegs: %vreg2, %vreg1, %vreg0
%I0<def> = COPY %vreg2; IntRegs: %vreg2
RETL 8, %I0<imp-use>
```

`MI` 包含关于指令的重要元信息：它存储被使用和被定义的寄存器，区别寄存器和内存操作数（以及其它类型），存储指令类型（分支、返回、调用、结束，等等），预测运算是否可交换，等等。保存这些信息甚至在像 `MI` 这样的低层次是重要的，因为在 `InstrEmitter` 之后代码输出之前运行的 Pass 要根据这些字段执行它们的分析。

6.8 寄存器分配

寄存器分配的基本任务是将无限数量的虚拟寄存器转换为有限的物理寄存器。由于目标的物理寄存器数量有限，有些虚拟寄存器被安排到内存位置，即 `spill slot`。然而，有些 `MI` 代码可能已经用到了物理寄存器，甚至在寄存器分配之前。当机器指令需要将结果写到特定的寄存器，或者出于 `ABI` 的需求，这种情况就会发生。对此，寄存器分配器承认先前的分配行为，在此基础上将其余的物理寄存器分配给剩余的虚拟寄存器。

LLVM 寄存器分配器的另一个重要任务是解构 `IR` 的 `SSA` 形式。直到此时，机器指令可能还包含 `phi` 指令，它们从原始的 LLVM `IR` 复制而来，为了支持 `SSA` 形式它们是必需的。如此，你可以方便地在 `SSA` 之上实现机

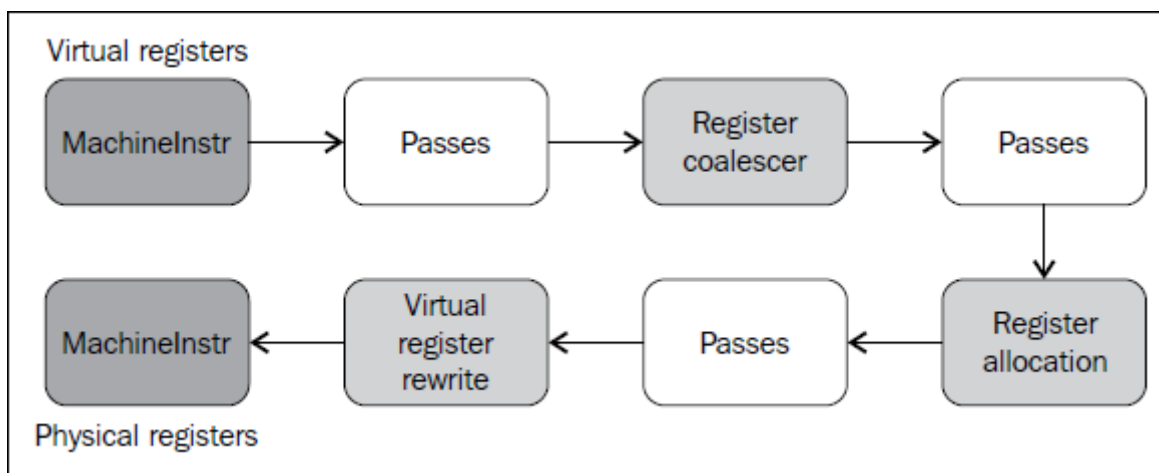
器特定的优化。然而，传统的将 phi 指令转换为常规指令的方法，是用复制指令替换它们。这样，SSA 解构不能晚于寄存器分配，这个阶段将会分配寄存器并且消除冗余的复制操作。

LLVM 有四种寄存器分配方法，这可以在 llc 中选择，通过 `-regalloc=<regalloc_name>` 选项。可选的 `<regalloc_name>` 有：pbqp, greedy, basic, 和 fast。

pbqp: 这种方法将寄存器分配映射为分区布尔二次规划 (PBQP: Partitioned Boolean Quadratic Programming) 问题。一个 PBQP 解决方法用于将这个结果映射回寄存器。greedy: 这种方法给出一种高效的全局（函数范围）寄存器分配实现，支持活跃区域分割以最小化挤出 (spill)。这里给出了关于这个算法的生动的解释：<http://blog.llvm.org/2011/09/greedy-register-allocation-in-llvm-30.html>。basic: 这种方法是一种很简单的分配器，并提供扩展接口。因此，它为开发新的寄存器分配器提供基础，被用作寄存器分配效率的基线。在前面的关于 greedy 算法的 blog 链接中，也有关于这个算法的内容。fast: 这种分配器是局部的（作用于各个基本块），它尽量地将值保持在寄存器中并重用它们。

default 分配器被映射为这四种方法的其中之一，根据当前的优化级别（-O 选项）作出选择。

虽然寄存器分配器在一个单一的 Pass 中实现，不管选择何种算法，但是它仍然依赖其它的分析，这构成了分配器框架。分配器框架用到一些 Pass，这里我们介绍寄存器合并器和寄存器重写，解释它们的概念。下图阐明了这些 Pass 如何相互交互。



6.8.1 寄存器合并器

寄存器合并器 (coalescer) 通过结合值区间 (interval) 去除冗余的复制指令 (COPY)。RegisterCoalescer class 实现了这种合并 (见 lib/CodeGen/RegisterCoalescer.cpp)，它是一个机器函数 Pass。机器函数 Pass 类似于 IR Pass，它运行在每个函数之上，只是处理的不是 IR 指令，而是 MachineInstr 指令。在合并期间，方法 joinAllIntervals() 复制指令的列表。方法 joinCopy() 从机器复制指令创建 CoalescerPair 实例，并且在可能的时候合并掉复制指令。

值区间 (interval) 表示程序中的一对点，开始和结束，它从一个值被产生时开始，直到这个值最终被使用，也就是说，被消灭 (killed)，期间它被保存在临时位置上。让我们看看合并器运行在我们的 sum.bc bitcode 例子上时会发生什么。

我们利用 llc 中的 regalloc 调试选项来查看合并器的调试输出：

```

$ llc -march=sparc -debug-only=regalloc sum.bc 2>&1 | head -n30
Computing live-in reg-units in ABI blocks.
0B BB#0 IO#0 I1#0
***** INTERVALS *****
IO [0B,32r:0) [112r,128r:1) 0@0B-phi 1@112r
I1 [0B,16r:0) 0@0B-phi
%vreg0 [32r,48r:0) 0@32r
%vreg1 [16r,96r:0) 0@16r
%vreg2 [80r,96r:0) 0@80r
%vreg3 [96r,112r:0) 0@96r
RegMasks:
***** MACHINEINSTRS *****
# Machine code for function sum: Post SSA
Frame Objects:
fi#0: size=4, align=4, at location[SP]
fi#1: size=4, align=4, at location[SP]
Function Live Ins: $IO in $vreg0, $I1 in %vreg1

0B BB#0: derived from LLVM BB %entry
Live Ins: %IO %I1
16B %vreg1<def> = COPY %I1<kill>; IntRegs:%vreg1
32B %vreg0<def> = COPY %IO<kill>; IntRegs:%vreg0
48B STri <fi#0>, 0, %vreg0<kill>; mem:ST4[%a.addr]
IntRegs:%vreg0
64B STri <fi#1>, 0, %vreg1; mem:ST4[%b.addr] IntRegs:$vreg1
80B %vreg2<def> = LDri <fi#0>, 0; mem:LD4[%a.addr]
IntRegs:%vreg2
96B %vreg3<def> = ADDrr %vreg2<kill>, %vreg1<kill>;
IntRegs:%vreg3,%vreg2,%vreg1
112B %IO<def> = COPY %vreg3<kill>; IntRegs:%vreg3
128B RETL 8, %IO<imp-use,kill>

# End machine code for function sum.

```

备注： 你可以用 `-debug-only` 选项对一个特定的 LLVM pass 或者组件开启内部调试消息。为了找出调试的组件，可在 LLVM 源代码文件夹中运行 `grep -r "DEBUG_TYPE" *`。DEBUG_TYPE 定义标记选项，它激活当前文件的调试消息，例如在寄存器分配的实现文件中有 `#define DEBUG_TYPE "regalloc"`。

注意，我们用 `2>&1` 重定向了打印调试信息的标准错误输出到标准输出。然后，管道标准输出（包含调试信息）到 `head -n30`，只打印前面的 30 行。以这种方式，我们控制了显示在终端上的信息量，因为调试信息可能相当繁琐。

首先让我们来看 `** MACHINEINSTRS **` 输出。这打印了作为寄存器合并器输入的所有机器指令——如果你

用 `-print-machine-insts=phi-node-elimination` 选项输出（运行于合并器之前的）phi 节点消除 pass 之后的机器指令，将得到相同的内容。然而，合并器调试器的输出，用索引信息给每条机器指令作提示：0B, 16B, 32B 等。我们需要它们以正确地解释值区间（interval）。

这些索引也被称为 slot indexes，给每个活跃区域（live range）赋予一个不同的数字。字母 B 对应基本块（block），被用于活跃区域进入或者离开一个基本块的边界。在此例中，我们的指令打印为索引跟着 B，因为这是默认单元（slot）。在值区间中，有一个不同的单元，字母 r，它表示寄存器，用于指示普通寄存器的使用或者定义。

通过阅读机器指令序列，我们已经知道了寄存器分配器超级 Pass（若干小 Pass 的组合）的重要内容：`%vreg0`, `%vreg1`, `%vreg2`, `%vreg3` 都是虚拟寄存器，需要为它们分配物理寄存器。因此，最多要使用 4 个物理寄存器，除了 `%I0` 和 `%I1` 之外，它们已经在使用了。其原因是为了遵守 ABI 调用惯例，它要求函数参数存于这些寄存器中。由于活跃变量分析 Pass 在寄存器合并之前运行，代码也标注了活跃变量信息，展示了每个寄存器在何处被定义和杀死，这让我们能够看清楚哪些寄存器相互冲突，即哪些寄存器同时活跃，需要保持在不同的物理寄存器中。

另一方面，合并器不依赖寄存器分配器的结果，它只是寻找寄存器复制。对于寄存器到寄存器的复制，合并器会尝试结合源寄存器和目标寄存器的值区间，让它们保持在相同的物理寄存器中，消除复制指令，就像索引 16 和 32 的复制。

紧跟着 ***** INTERVALS ***** 的消息，来自寄存器合并所依赖的另一个分析：活跃值区间分析（不同于活跃变量分析），它由 `lib/CodeGen/LiveIntervalAnalysis.cpp` 实现。合并器需要知道每个虚拟寄存器所活跃的值区间，这样才能发现哪些值区间可以合并。例如，我们可以从输出中看到，虚拟寄存器 `%vreg0` 的值区间被确定为 `[32r:48r:0)`。

这意味着这个半开放的值区间 `%vreg0` 在 32 处被定义，在 48 处被杀死。48r 后面的数字 0 是一个代码，它显示这个值区间在何处被第一次定义，这个意思恰好在值区间后面被打印出来：`o:32r`。这样，定义 o 出现在索引 32，这是我们已经知道的。然而，这可以让我们有效地追踪原始定义，监控值区间是否分裂。最后，`RegMasks` 显示了调用现场，它清理了很多寄存器，是冲突的一个大源头。因为这个函数没有任何调用，所以没有 `RegMasks` 位置。

通过观察值区间，我们有喜人的发现：`%I0` 寄存器的值区间是 `[0B, 32r:0)`，`%vreg0` 寄存器的值区间是 `[32r, 48r:0)`，在 32 处，有一条复制指令，它复制 `%I0` 到 `%vreg0`。这就是合并发生的前提：结合值区间 `[0B, 32r:0)` 和 `[32r, 48r:0)`，赋给 `%I0` 和 `%vreg0` 相同的寄存器。

下面，让我们打印其余的调试输出，看看发生了什么：

```
$ llc -match=sparc -debug-only=regalloc sum.bc
...
entry:
16B %vreg1<def> = COPY %I1;
IntRegs: %vreg1
    Considering merging %vreg1 with %I1
    Can only merge into reserved registers.
32B %vreg0<def> = COPY %I0;
IntRegs: %vreg0
    Considering merging %vreg0 with %I0
```

(续下页)

(接上页)

```

    Can only merge into reserved registers.
64B %I0<def> = COPY %vreg2;
IntRegs:%vreg2
    Considering merging %vreg2 with %I0
    Can only merge into reserved registers.
...

```

我们看到，合并器考虑结合%vreg0 和%I0，如我们希望的那样。然而，当寄存器是物理寄存器时，例如%I0，它实行了特殊的规则。物理寄存器必须保留以连接它的值区间。这意味着，不能将物理寄存器分配给其它的活跃区域，而%I0 的情况并非如此。因此，合并器放弃了这个机会，它担心过早地把%I0 分配给这个区间到最后可能无法获益，留由寄存器分配器作这个决定。

因此，程序 sum.bc 没有合并的机会。虽然它试图结合虚拟寄存器和函数参数寄存器，但是失败了，因为在此阶段它只能将虚拟寄存器和保留的——非常规可分配的——物理寄存器相结合。

6.8.2 虚拟寄存器重写

寄存器分配 Pass 为每个虚拟寄存器选择物理寄存器。随后，VirtRegMap 保存了寄存器分配的结果，它将虚拟寄存器映射到物理寄存器。接着，虚拟寄存器重写 Pass——由 VirtRegRewriter class 表示，它的实现在文件 <llvm_source>/lib/CodeGen/VirtRegMap.cpp 中——利用 VirtRegMap 将虚拟寄存器替换为物理寄存器。根据情况会生成 spill 代码。而且，剩下的恒等复制 reg = COPY reg 会被删除。例如，让我们利用 -debug-only=regalloc 选项分析分配器和重写器如何处理 sum.bc。首先，greedy 分配器输出如下文本：

```

...
assigning %vreg1 to %I1: I1
...
assigning %vreg0 to %I0: I0
...
assigning %vreg2 to %I0: I0

```

虚拟寄存器 1, 0, 2 分别被分配以物理寄存器%I1, %I0, %I0。VirtRegMap 输出中给出了相同的内容，如下：

```

[%vreg0 -> %I0] IntRegs
[%vreg1 -> %I1] IntRegs
[%vreg2 -> %I0] IntRegs

```

然后，重写器将所有虚拟寄存器替换为物理寄存器，并删除恒等的复制：

```

> %I1<def> = COPY %I1
Deleting identity copy.
> %I0<def> = COPY %I0
Deleting identity copy.
...

```


我们看到，尽管合并器无法去除这些复制，但是寄存器分配器能够为两个活跃区域赋以相同的寄存器，并删除复制操作，如我们希望的那样。最终，作为结果的 `sum` 函数的机器指令极大地简化了：

```
0B BB#0: derived from LLVM BB
%entry
Live Ins: %I0 %I1
48B %I0<def> = ADDrr %I1<kill>, %I0<kill>
80B RETL 8, %I0<imp-use>
```

注意，复制指令被删除了，没有剩下虚拟寄存器。

备注：只有当 LLVM 以 `debug` 模式编译（通过在配置时刻设置 `-disable-optimized`）后，才能使用 `llc` 程序的选项 `-debug` 或者 `-debug-only=<name>`。你可以在第 1 章（编译和安装 LLVM）的 **Building and installing LLVM** 小节找到更多相关内容。

在任何编译器中，寄存器分配和指令调度都是天生的敌人。寄存器分配的任务是尽可能让活跃区域短一点，减少冲突图的边的数目，而减少所需寄存器的数目，以避免挤出（`spill`）。因而，寄存器分配器喜欢以串行的模式排列指令（让指令紧跟在其所依赖指令的后面），因为用这种方法代码所用的寄存器相对较少。指令调度的任务是相反的：为了提升指令级别的并行，需要尽可能地让很多无关而并行的运算保持活跃，要用很多寄存器保存中间值，增加活跃区域之间冲突的数量。设计一个有效的算法来协同地处理指令调度和寄存器分配，是一个开放的研究课题。

6.8.3 目标钩子

在合并的时候，虚拟寄存器来自相容的寄存器类别，需要被顺利地合并。代码生成器从目标特定的描述获得这类信息，而描述由抽象方法给出。分配器可以从 `TargetRegisterInfo` 的子类（例如 `X86GenRegisterInfo`）获得所有关于一个寄存器的信息。这些信息包括，是否为保留的，父寄存器类别，是物理的还是虚拟的寄存器。

`<Target>InstrInfo` 类是另一个提供寄存器分配器所需要的目标特定的信息的数据结构。这里讨论一些例子：

- `<Target>InstrInfo` 的 `isLoadFromStackSlot()` 和 `isStoreToStackSlot()` 方法，用于在挤出代码生成期间发现机器指令访问栈单元的内存。
- 此外，它用 `storeRegToStackSlot()` 和 `loadRegFromStackSlot()` 方法生成访问栈单元的目标特定的内存访问指令。
- `COPY` 指令可能在寄存器重写之后保留下来，因为它们没有被合并掉，而且不是同一的复制。在这种情况下，`copyPhysReg()` 方法用于生成目标特定的寄存器复制，在需要时甚至在不同寄存器类别之间。`SparcInstrInfo::copyPhysReg()` 的例子是这样的：

```
if (SP::IntRegsRegClass.contains(DestReg, SrcReg))
    BuildMI(MBB, I, DL, get(SP::ORrr), DestReg).addReg(SP::G0)
        .addReg(SrcReg, getKillRegState(KillSrc));
...
```


`BuildMI()` 方法在代码生成器中到处可见，它用于生成机器指令。在这个例子中，`SP::ORrr` 指令用于复制一个 CPU 寄存器到另一个 CPU 寄存器。

6.9 序曲和尾声

完整的函数都需要序曲（prologue）和尾声（epilogue）。前者在函数的开始处设置堆栈帧和被调用者保存的寄存器，而后者在函数返回前清理堆栈帧。在例子 `sum.bc` 中，当为 SPARC 编译时，插入序曲和尾声之后，机器指令看起来是这样的：

```
%06<def> = SAVEri %06, -96
%I0<def> = ADDrr %I1<kill>, %I0<kill>
%G0<def> = RESTORErr %G0, %G0
RETL 8, %I0<imp-use>
```

此例中，`SAVEri` 指令是序曲，`RESTORErr` 是尾声，执行堆栈帧相关的设置和清理。序曲和尾声的生成是目标特定的，由方法 `<Target>FrameLowering::emitPrologue()` 和 `<Target>FrameLowering::emitEpilogue()` 定义（参见文件 `<llvm_source>/lib/Target/<Target>/<Target>FrameLowering.cpp`）。

6.9.1 帧索引

LLVM 在代码生成期间用到一个虚拟堆栈帧，利用帧索引引用堆栈元素。序曲的插入会分配堆栈帧，给出充足的目标特定的信息，让代码生成器得以将虚拟帧索引替换为实际的（目标特定）堆栈引用。

`<Target>RegisterInfo` 类的 `eliminateFrameIndex()` 方法实现了所述替换，就是检查所有包含堆栈引用（通常为 `load` 和 `store`）的机器指令，将每个帧索引转换为实际的堆栈偏移。当需要额外的堆栈偏移算术运算时，也会生成额外的指令。参见文件 `<llvm_source>/lib/Target/<Target>/<Target>RegisterInfo.cpp` 作为例子。

6.10 理解机器代码框架

机器代码（简称 MC）类包含整个低层操作函数和指令的框架。对比其它的后端组件，这是一个新设计的框架，助于创建基于 LLVM 的汇编器和反汇编器。之前，LLVM 缺少一个集成的汇编器，编译过程只能进行到汇编语言生成这一步，它创建一个汇编文本文件，要依靠外部的工具继续剩余的编译工作（汇编器和链接器）。

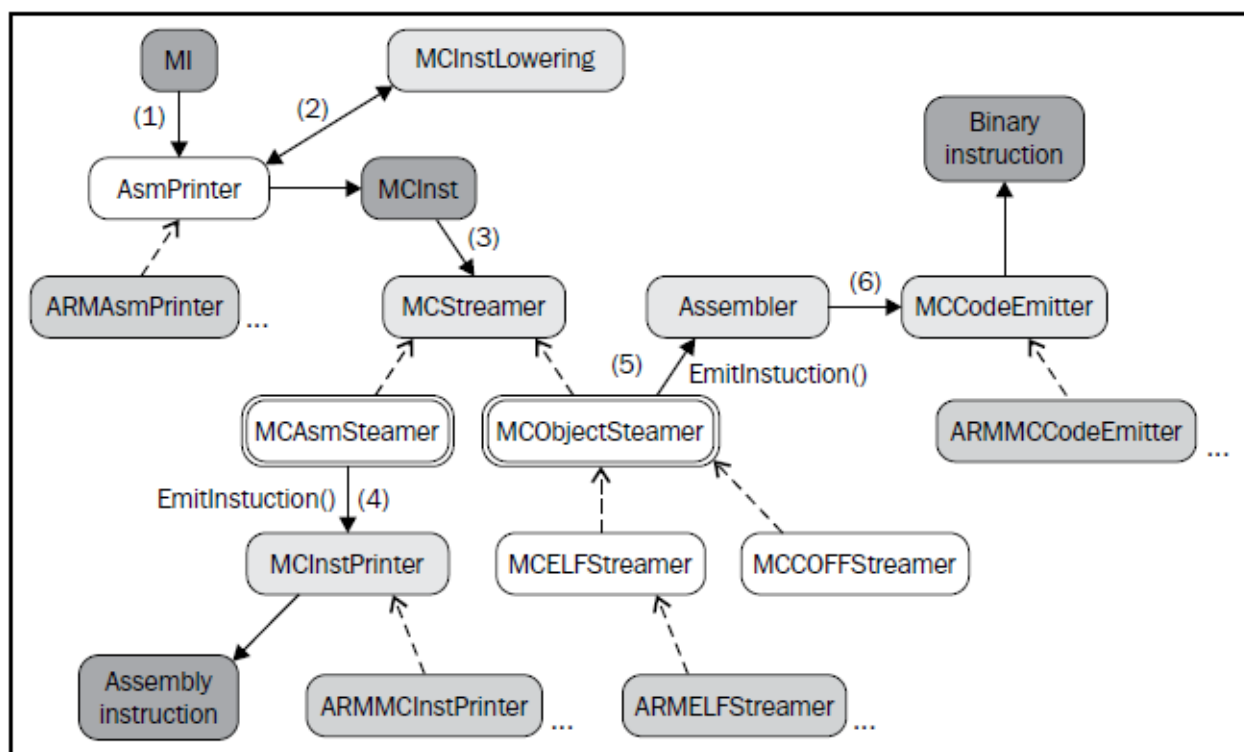
6.10.1 MC 指令

在 MC 框架中，机器代码指令 (MCInst) 替代了机器指令 (MachineInstr)。在文件 `<llvm_source>/include/llvm/MC/MCInst.h` 中定义的 MCInst 类，定义了对指令的轻量表示。对比 MI (机器指令)，MCInst 记录较少的程序信息。例如，MCInst 实例不仅可以由后端创建，而且可以由反汇编器只根据二进制代码创建，注意反汇编器是一个缺少指令上下文信息的环境。事实上，它融入了汇编器的理念，也就是说，其目的不是应用丰富的优化，而是组织指令生成目标文件。

每个操作数可以是一个寄存器，立即数 (整数或浮点数)，表达式 (表示为 MCEExpr)，或者另一个 MCInstr 实例。表达式用于表示标记 (label) 运算和重定位。MI 指令在代码生成阶段的早期被转换为 MCInst 实例，这是下个小节的主题。

6.10.2 代码输出

代码输出阶段处于所有后寄存器分配 Pass 之后。尽管名字似乎让人难于理解，代码生成从汇编打印 (AsmPrinter) 开始。下面的示意图给出了从 MI 指令到 MCInst 接着到汇编或者二进制指令的步骤：



让我们逐一介绍上图所示的步骤：

1. AsmPrinter 是一个机器函数 Pass，它首先生成函数头，然后遍历所有基本块，每次发送一个 MI 指令到方法 `EmitInstruction()`，以作进一步处理。每个目标会提供一个 AsmPrinter 子类，它重载这个方法。
2. `<Target>AsmPrinter::EmitInstruction()` 方法接收 MI 指令作为输入，凭借 MCInstLowering 接口将它转变为 MCInst 实例——每个目标会提供这个接口的子类，自定义生成这些 MCInst 实例的程序。

3. 此刻,可以接着生成汇编或者二进制指令。`MCStreamer` 类处理 `MCInst` 指令流,通过两个子类, `MCAsmStreamer` 和 `MCOBJECTStreamer`, 将指令输出为所选的格式。前者将 `MCInst` 转换为汇编语言,而后者将它转换为二进制指令。
4. 如果生成汇编指令,就会调用 `MCAsmStreamer::EmitInstruction()`, 由一个目标特定的 `MCInstPrinter` 子类打印汇编指令到文件。
5. 如果生成二进制指令, `MCOBJECTStreamer::EmitInstruction()` 的一个目标 (target) 专用的、目标代码 (object) 特定的版本就会调用 LLVM 目标代码编译器。
6. 编译器会利用一个专用的 `MCCodeEmitter::EncodeInstruction()` 方法, 蜕变 `MCInst` 实例, 编码和输出二进制指令数据块到文件, 以一种目标特定的方式。

此外, 你可以用 `llc` 工具输出 `MCInst` 片段。例如, 要将 `MCInst` 编码为汇编注释, 可以用下面的命令:

```
$ llc sum.bc -march=x86-64 -show-mc-inst -o -
...
pushq %rbp          ## <MCInst #2114 PUSH64r
                    ## <MCOperand Reg: 107>>
```

...

然而, 如果你想要将每条指令的二进制编码显示为汇编注释, 就用下面的命令:

```
$ llc sum.bc -march=x86-64 -show-mc-encoding -o -
...
push %rbp           ## encoding: [0x55]
...
```

`llvm-mc` 工具还让你能够测试和使用 MC 框架。例如, 为了查明一条特定指令的汇编编码, 使用选项 `-show-encoding`。下面是 x86 指令的一个例子:

```
$ echo "movq 48879(,%riz), %rax" | llvm-mc -triple=x86_64 --show-encoding
##encoding:
[0x48, 0x8b, 0x04, 0x25, 0xef, 0xbe, 0x00, 0x00]
```

这个工具还提供了反汇编的功能, 如下:

```
$ echo "0x8d 0x4c 0x24 0x04" | llvm-mc --disassemble -triple=x86_64
leal 4(%rsp), %ecx
```

另外, 选项 `-show-inst` 为经过汇编或反汇编的指令显示 `MCInst` 指令:

```
$ echo "0x8d 0x4c 0x24 0x04" | llvm-mc --disassemble --show-inst -triple=x86_64
leal 4(%rsp), %ecx    # <MCInst #1105 LEA64_32r
                     # <MCOperand Reg:46>
                     # <MCOperand Reg:115>
```

(续下页)

(接上页)

```
# <MCOperand Imm:1>
# <MCOperand Reg:0>
# <MCOperand Imm:4>
# <MCOperand Reg:0>>
```

MC 框架让 LLVM 能够为经典的目标文件阅读器提供可选择的工具。例如，目前默认编译 LLVM 会安装 `llvm-objdump` 和 `llvm-readobj` 工具。两者都用到了 MC 反汇编库，实现了跟 GNU Binutils 软件包中的等价物 (`objdump` 和 `readelf`) 相类似的功能。

6.11 编写你自己的机器 Pass

在这个章节，我们将示范如何编写一个定制的机器 Pass，它正好在代码生成之前，统计每个函数有多少机器指令。不同于 IR Pass，你不能用 `opt` 工具运行这个 Pass，或通过命令行加载并安排它运行。机器 Pass 由后端代码管理。因此，在实践中我们修改一个已有的后端来运行并观察我们定制的 Pass。我们选择 SPARC 后端。

回想第 3 章（工具和设计）的演示插件式 Pass 接口小节，从这章的第一张图的白框中，有很多选项供我们选择决定在何处运行我们的 Pass。为了应用这些方法，我们应该找到我们的后端实现的 `TargetPassConfig` 子类。如果你用 `grep`，就会在 `SparcTargetMachine.cpp` 中找到它：

```
$ cd <llvmsource>/lib/Target/Sparc
$ vim SparcTargetMachine.cpp # 使用你喜欢的编辑器
```

观察这个从 `TargetPassConfig` 派生的 `SparcPassConfig` 类，我们看到它覆写（override）了 `addInstSelector()` 和 `addPreEmitPass()`，但是我们可以覆写很多方法，如果我们想要在其它的地方添加一个 Pass（见链接 http://llvm.org/doxygen/html/classllvm_1_1TargetPassConfig.html）。我们将在代码生成前运行我们的 Pass，因此在 `addPreEmitPass()` 中添加代码：

```
bool SparcPassConfig::addPreEmitPass() {
    addPass(createSparcDelaySlotFillerPass(
        getSparcTargetMachine()));
    addPass(createMyCustomMachinePass());
}
```

在上面的代码中，高亮的行是我们额外添加的，它通过调用函数 `createMyCustomMachinePass()` 来添加我们的 Pass。然而，这个函数还未定义。我们将增加一个新的源代码文件，编写 Pass 代码，也会定义这个函数。于是，创建一个文件，名为 `MachineCountPass.cpp`，填写下面的内容：

```
#define DEBUG_TYPE "machinecount"
#include "Sparc.h"
#include "llvm/Pass.h"
#include "llvm/CodeGen/MachineBasicBlock.h"
#include "llvm/CodeGen/MachineFunction.h"
```

(续下页)

(接上页)

```

#include "llvm/CodeGen/MachineFunctionPass.h"
#include "llvm/Support/raw_ostream.h"

using namespace llvm;

namespace {
class MachineCountPass : public MachineFunctionPass {
public:
    static char ID;
    MachineCountPass() : MachineFunctionPass(ID) {}

    virtual bool runOnMachineFunction(MachineFunction &MF) {
        unsigned num_instr = 0;
        for (MachineFunction::const_iterator I = MF.begin(), E = MF.end(); I != E; ++I)
        ↪ {
            for (MachineBasicBlock::const_iterator BBI = I->begin(), BBE = I->end(); BBI !=
        ↪ BBE; ++BBI) {
                ++num_instr;
            }
        }
        errs() << "mcount --- " << MF.getName() << " has " << num_instr << "
        ↪ instructions.\n";
        return false;
    }
};
}

FunctionPass *llvm::createMyCustomMachinePass() {
    return new MachineCountPass();
}

char MachineCountPass::ID = 0;
static RegisterPass<MachineCountPass> X("machinecount", "Machine Count Pass");

```

在第一行中，我们定义了宏 `DEBUG_TYPE`，这样以后我们就可以通过选项 `-debug-only=machinecount` 调试这个 Pass。然而，在这个例子中，没有用到调试输出。剩余的代码和我们前一章为 IR Pass 写的很相似。不同之处如下：

- 在包含文件中，我们包含了头文件 `MachineBasicBlock.h`, `MachineFunction.h`, `MachineFunctionPass.h`，它们定义了我们用于提取 `MachineFunction` 信息的类，让我们能够计数它包含的机器指令。我们还包含了头文件 `Sparc.h`，因为我们将声明 `createMyCustomMachinePass()`。
- 我们创建了一个类，从 `MachineFunctionPass` 派生，而不是从 `FunctionPass`。
- 我们覆写了 `runOnMachineFunction()` 方法，而不是 `runOnFunction()`。另外，方法的实现是相当不同的。我

们遍历了当前 `MachineFunction` 中的所有 `MachineBasicBlock` 实例。然后，对于每个 `MachineBasicBlock`，调用 `begin()/end()` 语句以计数所有的机器指令。

- 我们定义了函数 `createMyCustomMachinePass()`，让这个 `Pass` 在我们所修改的 SPARC 后端文件中被创建和添加为代码生成之前的 `Pass`。

既然已经定义了函数 `createMyCustomMachinePass()`，我们就必须在头文件中声明它。让我们编辑 `Sparc.h` 文件来做这件事。在 `createSparcDelaySlotFillerPass()` 的后面添加我们的函数声明：

```
FunctionPass *createSparcISelDag(SparcTargetMachine &TM);
FunctionPass *createSparcDelaySlotFillerPass(TargetMachine &TM);
FunctionPass *createMyCustomMachinePass();
```

下面让我们用 LLVM 编译系统编译新的 SPARC 后端。如果你还没有配置你的 LLVM 编译系统，就参考第 1 章（编译和安装 LLVM）。如果你已经有了配置项目的 `build` 文件夹，就进入这个文件夹，运行 `make` 以编译新的后端。接着，你可以安装包含修改了的 SPARC 后端的新的 LLVM，或者依你所愿，只是从你的 `build` 文件夹运行新的 `llc` 二进制程序，而不运行 `make install`：

```
$ cd <llvm-build>
$ make
$ Debug+Asserts/bin/llc -march=sparc sum.bc
mcount --- sum has 8 instructions.
```

如果我们想知道我们的 `Pass` 在 `Pass` 管线中被插入在什么位置，输入下面的命令：

```
$ Debug+Asserts/lib/llc -march=sparc sum.bc -debug-pass=Structure
(...)
Branch Probability Basic Block Placement
SPARC Delay Slot Filler
Machine Count Pass
MachineDominator Tree Construction
Sparc Assembly Printer
mcount --- sum has 8 instructions.
```

我们看到，我们的 `Pass` 恰好被安排在 SPARC Delay Slot Filler 之后，在 Sparc Assembly Printer 之前，后者是代码生成发生的地方。

6.12 总结

在这一章中，我们概要地介绍了 LLVM 后端是如何工作的。我们看到了不同的代码生成阶段，以及在编译过程中变化的内部指令表示。我们讨论了指令选择、调度、寄存器分配、代码生成，为读者演示了用 LLVM 工具对这些阶段做实验的方法。在本章结束的时候，你应该能够读懂 `llc -debug` 的输出，它打印出后端活动的详细的日志，展示了发生在后端内部的一切事情的全貌。如果你有兴趣编写自己的后端，你的下一步就是参考官方的教程：<http://llvm.org/docs/WritingAnLLVMBackend.html>。如果你有兴趣阅读更多的关于后端设计的内

容，你应该参考 <http://llvm.org/docs/CodeGenerator.html>。

在下一章中，我们将介绍 LLVM Just-in-Time 编译框架，它让你能够按需要随时地生成代码。

第 7 章 Just-in-Time 编译器

LLVM Just-in-Time (JIT) 编译器是一个基于函数的动态翻译引擎。为了理解什么是 JIT 编译器，让我们回顾原始的术语。这个术语来自 Just-in-Time 制造，一种商业策略，即工厂按需制造或者购买物资，而不引入库存。在编译过程中，这个比喻很合适，因为 JIT 编译器不会将二进制程序存储到磁盘（库存），而是在你需要它们的时候开始编译程序部分。尽管人们接受了业内行话，你可能还困惑于其它的名字，例如延时（late）或者懒惰（lazy）编译。

JIT 策略的优势在于知道将运行程序的精确的机器和微架构。这让 JIT 系统能够为特定的处理器微调代码。而且，有的编译器只有在运行时知道其输入，因而只能实现为 JIT 系统，除此之外别无选项。例如，GPU 驱动程序即时编译着色语言，互联网浏览器处理 JavaScript 也是如此。在这一章中，我们将探索 LLVM JIT 系统，讨论下列内容：

- `llvm::JIT` 类和它的基础结构
- 如何利用 `llvm::JIT` 类执行 JIT 编译
- 如何利用 `GenericValue` 简化函数调用
- `llvm::MCJIT` 类和它的基础结构
- 如何利用 `llvm::MCJIT` 类执行 JIT 编译

7.1 了解 LLVM JIT 引擎基础

LLVM JIT 编译器是基于函数的，因为它一次能够编译单个函数。这定义了编译器的工作粒度，对于 JIT 系统来说是一个重要的决定。通过按需编译函数，编译器只会处理当前程序调用中实际用到的函数。例如，你的程序有若干个函数，你在启动它的时候设置了错误的命令行参数，一个基于函数的 JIT 系统只会编译那个打印帮助消息的函数，而不是这个程序。

备注：理论上，我们可以进一步细化粒度，只编译执行踪迹（trace），就是函数的具体的线路。如此，我们已经利用了 JIT 系统的重要优势：对于给定输入的一次程序调用，知道应该尽力去编译哪个程序线路，而不是其它的。然而，LLVM JIT 系统并不支持基于踪迹的编译，一般来说，它更受研究者的关注。关于 JIT 编译的讨论没有尽头，大量不同的权衡值得仔细研究，指出哪种策略最优不是一件简单的事。目前，计算机科学社区积累了大约 20 年的对 JIT 编译的研究，这个领域仍然非常活跃，每年都有新的论文尝试解决这个开放的问题。

JIT 引擎在运行时编译并且执行 LLVM IR 函数。在编译阶段，JIT 引擎会用 LLVM 代码生成器生成由目标特定的二进制指令组成的二进制数据块。它返回一个指向所编译函数的指针，这个函数可以被执行。

小技巧：一篇有趣的博客文章对比了 JIT 编译的开源解决方案，见 <https://eli.thegreenplace.net/2014/01/15/some-thoughts-on-llvm-vs-libjit>，它分析了 LLVM 和 libjit，后者是一个小型的致力于 JIT 编译的开源项目。LLVM 作为静态编译器比 JIT 系统更加有名，因为在 JIT 编译过程中，每个 Pass 消耗的时间是很重要的，算作程序执行的开销。LLVM 基础架构更注重支持慢而强的优化，和 GCC 相似，而不是快而弱的优化，后者对构建一个有竞争力的 JIT 系统很重要。尽管如此，LLVM 已经被成功地应用于 JIT 系统来建立 Webkit JavaScript 引擎的第四级 LLVM（Fourth Tier LLVM, FTL）组件（<http://blog.llvm.org/2014/07/ftl-webkits-llvm-based-jit.html>）。因为第四级只用在长时间运行的 JavaScript 应用程序，激进的 LLVM 优化可以发挥作用，即使它们不如更低级的优化快。从理性来看，如果应用程序运行时间长，我们就可以在代价高的优化上付出更多时间。想要了解更多关于这种权衡，参看 Modeling Virtual Machines Misprediction Overhead，作者 Cesar et al.，发表于 IISWC 2013，它分析揭示了 JIT 系统在多大程度上因为对不值得的代码使用了高代价的代码生成而受损失。当你的 JIT 系统浪费大量时间去优化一个仅执行若干次的程序片段时，这种情况就发生了。

7.1.1 介绍执行引擎

LLVM JIT 系统采用了一个执行引擎来支持 LLVM 模块的执行。ExecutionEngine 类在 `<llvm_source>/include/llvm/ExecutionEngine/ExecutionEngine.h` 中定义，它被设计出来以支持执行，通过 JIT 系统或者解释器（参考后面的信息盒子）。一般来说，一个执行引擎负责管理整个宾客程序的执行，分析接下来需要运行的程序片段，采取合理的动作来执行它。要作 JIT 编译，必须有一个执行管理器来协调编译决策，运行宾客程序（一次一个片段）。就 LLVM 的 ExecutionEngine 类而言，它将执行部分抛回给你，即客户。它可以运行编译管线，产生驻留内存的代码，但是由你决定是否执行此代码。

除了接受 LLVM 模块并执行它，引擎支持下面几个场景：

- 懒惰（lazy）编译：函数被调用时，引擎才编译它。关闭懒惰编译后，一旦你请求指向函数的指针，引擎就编译它们。
- 编译外部全局变量：这包括对当前 LLVM 模块的外部实体的符号解析和内存分配。
- 通过 dlsym 查找和解析外部符号：这个过程和运行时动态共享对象（dynamic shared object, DSO）加载一样。

LLVM 实现了两个执行引擎：llvm::JIT 类和 llvm::MCJIT 类。ExecutionEngine::EngineBuilder() 方法实例化一个 ExecutionEngine 对象，根据一个 IR 模块参数。接着，ExecutionEngine::create() 方法创建一个 JIT 或者 MCJIT 实例，两者的实现截然不同，这正是这一章要讲清楚的内容。

备注：解释器实现了一种非传统的策略来执行宾客代码，就是硬件平台（宿主平台）不原生地支持此代码。例如，LLVM IR 是 x86 平台上的宾客代码，因为 x86 处理器不能之间执行 LLVM IR。不同于 JIT 编译器，解释器的任务是读取每条指令，解码它们并执行它们的行为，在软件中模仿物理处理器的功能。尽管解释器省去了启动编译器翻译宾客代码的时间，它们往往慢得多，除非编译宾客代码所需的时间不能抵消解释代码的高额开销。

7.1.2 内存管理

一般来说，JIT 引擎在运行时将二进制数据块写入内存，这是由 ExecutionManager 类完成的。随后，就可以跳转到分配的内存区域来执行这些指令了，也就是调用 ExecutionManager 返回给你的函数指针。在此上下文中，内存管理是极其重要的，处理很多常规的任务，例如分配内存，释放内存，为加载库提供空间，和内存权限管理。

JIT 和 MCJIT 类都实现了一个定制的内存管理类，从基类 RTDyldMemoryManager 派生而来。任何 ExecutionEngine 用户可能也提供定制的 RTDyldMemoryManager 派生类，来指定不同的 JIT 组件应该被放置在内存的何处。你可以在 <llvm_source>/include/llvm/ExecutionEngine/RTDyldMemoryManager.h 文件中找这个接口。

例如，RTDyldMemoryManager 类声明了如下方法：

- allocateCodeSection() 和 allocateDataSection(): 这些方法分配内存以存放给定大小和对齐的可执行代码和数据。内存管理的用户可以通过一个内部的 section 标识符追踪已分配的 section。
- getSymbolAddress(): 这个方法返回当前链接的库中可获得的 symbol 的地址。注意这不是用于获得 JIT 编译生成的 symbol。调用这个方法时，必须提供一个 std::string 实例以存放 symbol 的名字。
- finalizeMemory(): 这个方法应该在对象加载完成时被调用，然后终于可以设置内存权限了。举例来说，不能在调用这个方法之前运行生成的代码。正如这一章要进一步解释的那样，这个方法被导向到 MCJIT 用户而不是 JIT 用户。

尽管用户可以提供定制的内存管理实现，JITMemoryManager 和 SectionMemoryManager 分别是 JIT 和 MCJIT 的默认子类。

7.2 介绍 llvm::JIT 基础结构

JIT 类和它的框架代表原先的引擎，它是通过使用 LLVM 代码生成器的不同部分而实现的。LLVM 3.5 之后，它将被移除。尽管这个引擎大部分是目标无关的，每个目标必须为它的具体的指令实现二进制指令输出。

7.2.1 数据块写到内存

JIT 类通过 JITCodeEmitter 输出二进制指令，它是 MachineCodeEmitter 类的子类。MachineCodeEmitter 类用于机器代码输出，它和新的机器代码（Machine Code, MC）框架是没有联系的——尽管陈旧，它依然存在以支持 JIT 类的功能。它的局限是只支持若干个目标，对于已经支持的目标，不是所有目标特性都是可用的。

MachineCodeEmitter 类的方法使下列任务变得容易：

- 为当前将输出的函数分配空间
- 将二进制数据块写到内存缓冲区（emitByte(), emitWordLE(), emitWordBE(), emitAlignment(), 等）
- 追踪当前缓冲区地址（就是一个指针，指向下一条指令将被在何处输出的地址）
- 添加重定位，与此缓冲区内的指令地址相关联

将字节写到内存的任务是由 JITCodeEmitter 执行的，它是参与代码输出过程的另一个类。它是 JITCodeEmitter 的子类，实现具体的 JIT 功能和管理。JITCodeEmitter 是相当简单的，只是将字节写到缓冲区，而 JITEmitter 具有下列改进：

- 专用的内存管理器，JITMemoryManager，之前提到过（也是下一节的主题）。
- 解决者（JITResolver）实例，跟踪和解决未被编译的函数的调用现场。这对懒惰函数编译是至关重要的。

7.2.2 使用 JITMemoryManager

JITMemoryManager 类（见 <llvm_source>/include/llvm/ExecutionEngine/JITMemoryManager.h）实现了低层级内存处理，为前面提及的类提供缓冲区。除了来自 RTDyldMemoryManager 的方法，它提供具体的方法来协助 JIT 类，例如 allocateGlobal()，为单个全局变量分配内存；startFunctionBody()，建立 JIT 调用，分配内存并标记为读/写可执行，以输出指令。

内部地，JITMemoryManager 类使用 JITSlabAllocator slab 分配器（<llvm_source>/lib/ExecutionEngine/JIT/JITMemoryManager.cpp）和 MemoryBlock 单元（<llvm_source>/include/llvm/Support/Memory.h）。

7.2.3 目标代码输出

每个目标都实现一个机器函数 Pass, 称为 `<Target>CodeEmitter` (见 `<llvm_source>/lib/Target/<Target>CodeEmitter.cpp`), 它将指令编码为数据块, 利用 `JITCodeEmitter` 写到内存。MipsCodeEmitter, 以此为例, 遍历所有函数基本块, 对于每条机器指令 (MI), 调用 `emitInstruction()`:

```
(...)
MCE.startFunction(MF);

for (MachineFunction::iterator MBB = MF.begin(), E = MF.end(); MBB != E; ++MBB) {
    MCE.StartMachineBasicBlock(MBB);
    for (MachineBasicBlock::instr_iterator I = MBB->instr_begin(), E = MBB->instr_end();
        ↪ I != E;)
        emitInstruction(*I++, *MBB);
}
(...)
```

MIPS32 是固定 4 字节长度的 ISA, 这使得 `emitInstruction()` 的实现很简单。

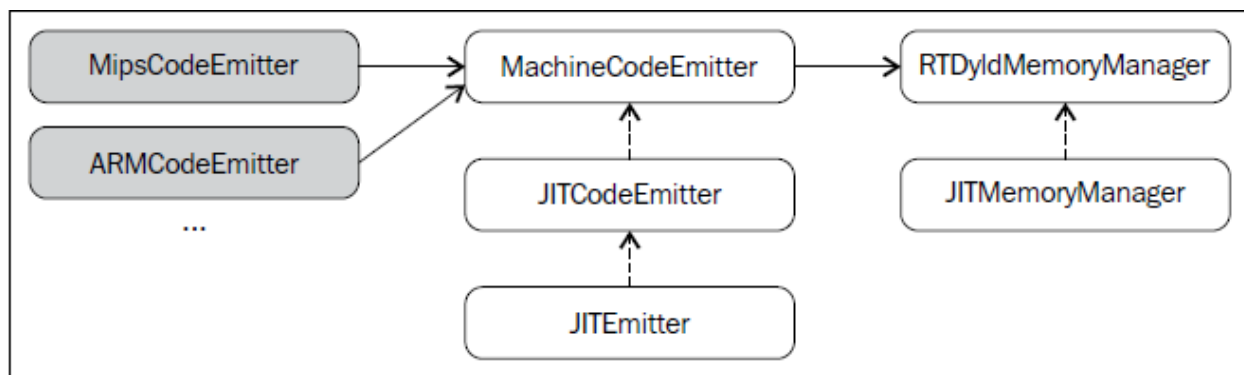
```
void MipsCodeEmitter::emitInstruction(MachineBasicBlock::instr_iterator MI, ↪
    ↪ MachineBasicBlock &MBB) {
    ...
    MCE.processDebugLoc(MI->getDebugLoc(), true);

    emitWord(getBinaryCodeForInstr(*MI));
    ++NumEmitted; // Keep track of the # of mi's emitted
    ...
}
```

`emitWord()` 方法是对 `JITCodeEmitter` 的包装, `getBinaryCodeForInstr()` 是 TableGen 为每个目标生成的, 通过解读 `.td` 文件中的指令编码描述。`<Target>CodeEmitter` 类还必须实现定制的方法以编码操作数和其它目标特定的实体。例如, 在 MIPS 中, 内存操作数必须使用 `getMemEncoding()` 放以恰当地编码 (见 `<llvm_source>/lib/Target/Mips/MipsInstrInfo.td`):

```
def mem : Operand<iPTR> {
    (...)
    let MIOperandInfo = (ops ptr_rc, simm16);
    let EncoderMethod = "getMemEncoding";
    (...)
}
```

因此, `MipsCodeEmitter` 必须实现 `MipsCodeEmitter::getMemEncoding()` 方法以符合这个 TableGen 描述。下面的示意图显示了几个代码输出器和 JIT 框架的关系:



7.2.4 目标信息

为了支持 Just-in-Time 编译，每个目标还必须提供一个 `TargetJITInfo` 的子类（见 `include/llvm/Target/TargetJITInfo.h`），例如 `MipsJITInfo` 或者 `X86JITInfo`。`TargetJITInfo` 类为通用的 JIT 功能提供了接口，需要每个目标实现它们。下面，我们来看这些功能的一些例子：

- 为了支持执行引擎重编译一个函数的需求——或许因为它被修改了——每个目标要实现 `TargetJITInfo::replaceMachineCodeForFunction()` 方法，修补原先函数的位置，用指令跳转或调用新版本函数。对于自修改代码，这是必需的。
- `TargetJITInfo::relocate()` 方法修补当前输出函数中的每个 symbol 引用，以指向正确的内存地址，这个做法和动态链接器类似。
- `TargetJITInfo::emitFunctionStub()` 方法输出一个代理：一个函数以调用给定地址的另一个函数。每个目标还要为输出的代理提供定制的 `TargetJITInfo::StubLayout` 信息，包括字节长度和对齐。`JITEmitter` 会使用这些代理信息为新的代理在输出它之前分配空间。

虽然 `TargetJITInfo` 方法的目的是不是输出常规的指令，诸如函数体生成，但是它们仍然需要为代理输出具体的指令，调用新的内存位置。然而，当 JIT 框架建立之后，没有接口可以依赖以使得输出孤立的指令变得容易，它们存在于 `MachineBasicBlock` 之外。这是今天 `MCInsts` 为 `MCJIT` 做的事情。没有 `MCInsts`，原先的 JIT 框架强制让目标手工编码指令。

为了揭示 `<Target>JITInfo` 的实现如何需要手工地输出指令，让我们来看 `MipsJITInfo::emitFunctionStub()` 的代码（见 `<llvm_source>/lib/Target/Mips/MipsJITInfo.cpp`），它用以下代码生成 4 条指令：

```

...
// lui $t9, %hi(EmittedAddr)
// addiu $t9, $t9, %lo(EmittedAddr)
// jalr $t8, $t9
// nop
if (IsLittleEndian) {
JCE.emitWordLE(0xf << 26 | 25 << 16 | Hi);
JCE.emitWordLE(9 << 26 | 25 << 21 | 25 << 16 | Lo);
JCE.emitWordLE(25 << 21 | 24 << 11 | 9);
JCE.emitWordLE(0);

```

(续下页)

(接上页)

...

7.2.5 学习如何使用 JIT 类

JIT 是一个 ExecutionEngine 子类，声明于 <llvm_source>/lib/ExecutionEngine/JIT/JIT.h。JIT 类是编译函数的入口，借助 JIT 基础结构。

ExecutionEngine::create() 方法调用 JIT::createJIT()，以一个默认的 JITMemoryManager。接着，JIT 构造器执行下面的任务：

- 创建 JITEmitter 实例
- 初始化目标信息对象
- 为代码生成添加 Pass
- 添加最后运行的 <Target>CodeEmitter Pass

引擎保存了一个 PassManager 对象，以调用所有的代码生成和 JIT 输出 Pass，每当被请求 JIT 编译一个函数的时候。

为了阐明一切是怎么发生的，我们已经描述了如何 JIT 编译 sum.bc 的一个函数，第 5 章（LLVM 中间表示）和第 6 章（后端）到处在用此 bitcode 文件。我们的目的是获取 Sum 函数，并且用 JIT 系统计算两个不同的引用运行时参数的加法运算。让我们执行下面的步骤：

1. 首先，创建一个新文件，名为 sum-jit.cpp。我们要包含 JIT 执行引擎的资源：

```
#include "llvm/ExecutionEngine/JIT.h"
```

2. 包含其它的头文件，涉及读写 LLVM bitcode、上下文接口等，并导入 LLVM namespace：

```
#include "llvm/ADT/OwningPtr.h"
#include "llvm/Bitcode/ReaderWriter.h"
#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/Module.h"
#include "llvm/Support/FileSystem.h"
#include "llvm/Support/MemoryBuffer.h"
#include "llvm/Support/ManagedStatic.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/Support/system_error.h"
#include "llvm/Support/TargetSelect.h"

using namespace llvm;
```

3. InitializeNativeTarget() 方法设置宿主目标，确保能够链接 JIT 将用到的目标库。和往常一样，每个线程需要一个上下文 LLVMContext 对象和一个 MemoryBuffer 对象，以从磁盘读取 bitcode 文件，如下面的代码所示：


```
int main() {
    InitializeNativeTarget();
    LLVMContext Context;
    std::string ErrorMessage;
    OwningPtr<MemoryBuffer> Buffer;
```

4. 用 `getFile()` 方法从磁盘读文件，如下面的代码所示：

```
if (MemoryBuffer::getFile("./sum.bc", Buffer)) {
    errs() << "sum.bc not found\n";
    return -1;
}
```

5. `ParseBitcodeFile` 函数从 `MemoryBuffer` 读取数据，生成相应的 LLVM Module 类以表示它，如下面的代码所示：

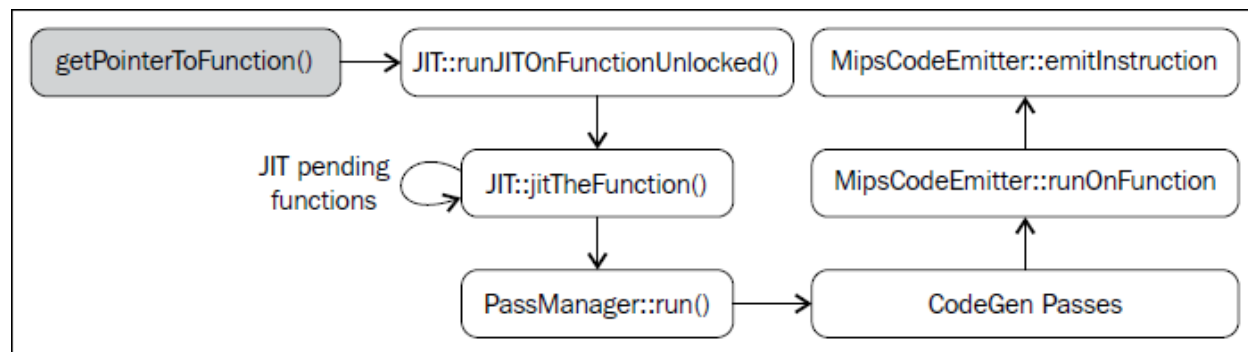
```
Module *M = ParseBitcodeFile(Buffer.get(), Context, &ErrorMessage);
if (!M) {
    errs() << ErrorMessage << "\n";
    return -1;
}
```

6. 调用 `EngineBuilder` 工厂的 `create` 方法创建一个 `ExecutionEngine` 实例，如下面的代码所示：

```
OwningPtr<ExecutionEngine> EE(EngineBuilder(M).create());
```

这个方法默认创建一个 JIT 执行引擎，是 JIT 的设置点；它直接调用 JIT 构造器来创建 `JITEmmitter`、`PassManager`，并初始化所有代码生成和目标特定的输出（emission）Pass。此刻，尽管引擎接受了一个 LLVM Module，还没有编译函数。

为了编译函数，还需要调用 `getPointerToFunction()`，它得到一个指向原生 JIT 编译的函数的指针。如果这个函数未曾 JIT 编译过，就作 JIT 编译并返回函数指针。下图阐明了此编译过程：



7. 通过 `getFunction()` 方法获取表示 `sum` 函数的函数 IR 对象：


```
Function *SumFn = M->getFunction("sum");
```

这里，JIT 编译被触发了：

```
int (*Sum)(int, int) = (int (*)(int, int)) EE->getPointerToFunction(SumFn);
```

你需要作一次恰当的类型转换，转换到匹配这个函数的函数指针类型。Sum 函数的 LLVM 定义原型是 `i32 @sum(i32 %a, i32 %b)`，因此我们用 `int (*)(int, int)` C 原型。

另一个选项是考虑懒惰编译，调用 `getPointerToFunctionOrStub()` 而不是 `getPointerToFunction()`。这个方法将生成一个代理函数，并且返回它的指针，如果目标函数还没有被编译并且懒惰编译是开启的。代理是一个简单的函数，包含一个占位符，将来修改占位符就可以跳转/调用实际的函数。

8. 接下来，根据 Sum 所指向的 JIT 编译了的函数，调用原始的 Sum 函数，如下面的代码所示：

```
int res = Sum(4, 5);
outs() << "Sum result: " << res << "\n";
```

当使用懒惰编译时，Sum 调用代理函数，它会用一个编译回调函数来 JIT 编译实际的函数。然后修改代理以重定向到实际函数并执行它。除非原始的 Module 中的 Sum 函数改变了，这个函数绝不会被再次编译。

9. 再次调用 Sum 来计算下一个结果，如下面的代码所示：

```
res = Sum(res, 6);
outs() << "Sum result: " << res << "\n";
```

在懒惰编译环境中，由于原始的函数在第一次调用 Sum 时已经编译过了，第二次调用会直接执行原生函数。

10. 我们成功地用 JIT 编译的 Sum 函数计算了两次加法。现在，释放执行引擎分配的存放函数代码的内存，调用 `llvm_shutdown()` 函数并返回：

```
EE->freeMachineCodeForFunction(SumFn);
llvm_shutdown();
return 0;
}
```

要编译并链接 `sum-jit.cpp`，可以用下面的命令行：

```
$ clang++ sum-jit.cpp -g -O3 -rdynamic -fno-rtti $(llvm-config --cppflags --ldflags --
↳libs jit native irreader) -o sum-jit
```

或者，利用第 3 章（工具和设计）的 Makefile，添加 `-rdynamic` 选项，修改 `llvm-config` 调用以使用前面的命令行指定的库。尽管这个例子没有使用外部函数，`-rdynamic` 选项是重要的，它保证外部函数在运行时被解析。

运行这个例子并查看输出：

```
$ ./sum-jit
Sum result: 9
Sum result: 15
```

通用值

在前面的例子中，我们将返回的函数指针转换为恰当的原型，为了用 C 样式的函数调用去调用这个函数。然而，当我们处理多个函数并且它们采用众多的签名和参数类型时，需要一种更灵活的方法去执行它们。

执行引擎提供了另一种调用 JIT 编译的函数的方法。`runFunction()` 方法编译并运行一个函数，函数参数由 `GenericValue` 向量决定——不需要提前调用 `getPointerToFunction()`。

`GenericValue` struct 在 `<llvm_source>/include/llvm/ExecutionEngine/GenericValue.h` 中被定义，它能够存放任何通用的类型。让我们修改前面的例子，以使用 `runFunction()` 而不是 `getPointerToFunction()` 和类型转换。

首先，创建文件 `sum-jit-gv.cpp` 以保存这个新的版本，在开头添加 `GenericValue` 头文件：

```
#include "llvm/ExecutionEngine/GenericValue.h"
```

从 `sum-jit.cpp` 复制其余的内容，让我们关注修改部分。在 `SumFn` 函数指针初始化之后，创建 `FnArgs`——`GenericValue` 向量——并用 `APInt` 接口 (`<llvm_source>/include/llvm/ADT/APInt.h`) 填充整数值。根据函数原型 `sum(i32 %a, i32 %b)`，填充两个 32 位长度的整数：

```
(...)
Function *SumFn = m->getFunction("sum");
std::vector<GenericValue> FnArgs(2);
FnArgs[0].IntVal = APInt(32, 4);
FnArgs[1].IntVal = APInt(32, 5);
```

以函数变量和参数向量调用 `runFunction()`。这样，函数会被 JIT 编译并执行。相应地，结果也是 `GenericValue`，可以被访问。

```
GenericValue Res = EE->runFunction(SumFn, FnArgs);
outs() << "Sum result: " << Res.IntVal << "\n";
```

重复相同的过程，以执行第二个加法：

```
FnArgs[0].IntVal = Res.IntVal;
FnArgs[1].IntVal = APInt(32, 6);
Res = EE->runFunction(SumFn, FnArgs);
outs() << "Sum result: " << Res.IntVal << "\n";
(...)
```

7.3 介绍 llvm::MCJIT 框架

MCJIT 类是 LLVM 新的 JIT 实现。它和原先的 JIT 实现的不同在于 MC 框架，第 6 章（后端）对此作过探索。MC 提供了统一的指令表达方式，它作为一个框架，为汇编器、反汇编器、汇编打印器和 MCJIT 所共享。

应用 MC 库的第一个优势在于，目标只需要指定一次它们的指令的编码，因为所有子系统都会得到此信息。因此，当你编写 LLVM 后端的时候，如果你实现了目标的目标代码输出功能，也就实现了 JIT 功能。

llvm::JIT 将在 LLVM 3.5 之后被去除，完全替换为 llvm::MCJIT 框架。那么，我们为何学习原先的 JIT 呢？虽然它们是不同的实现，但是 ExecutionEngine 类是通用的，大部分概念是两者共有的。最重要的是，像在 LLVM 3.4 版本中，MCJIT 的设计不支持某些特性，例如懒惰编译，它还不是原先 JIT 的完全接替者。

7.3.1 MCJIT 引擎

创建 MCJIT 引擎的方法和原先的 JIT 引擎相同，通过调用 ExecutionEngine::create()。这个方法调用 MCJIT::createJIT()，它会执行 MCJIT 构造器。MCJIT 类在文件 <llvm_source>/lib/ExecutionEngine/MCJIT/MCJIT.h 中声明。createJIT() 方法和 MCJIT 构造器在文件 <llvm_source>/lib/ExecutionEngine/MCJIT/MCJIT.cpp 中实现。

MCJIT 构造器创建一个 SectionMemoryManager 实例；将 LLVM 模块添加到它内部的模块容器，OwningModuleContainer；并且初始化目标信息。

了解模块的状态

MCJIT 类为引擎建立期间插入的初始 LLVM 模块实例指定状态。这些状态描绘了模块的编译阶段。状态如下：

- **Added:** 这些模块所包含的模块集还没有被编译但已经被添加到执行引擎了。这个状态的存在让模块能够为其它模块暴露函数定义，延迟对它们的编译直到必需之时。
- **Loaded:** 这些模块处在已 JIT 编译状态但是还未准备好执行。重定位还没有做，内存页面还需要给予恰当的权限。愿意在内存中重映射已 JIT 编译的函数的用户，也许能避免重编译，通过使用 loaded 状态的模块。
- **Finalized:** 这些模块包含已经准备好执行的函数。在此状态下，函数不能被重映射了，因为重定位已经做过了。

JIT 和 MCJIT 的一个主要区别就在于模块状态。在 MCJIT 中，引擎模块必须在请求 symbol 地址（函数和全局变量）之前就绪（finalized）。

MCJIT::finalizeObject() 函数将已添加模块转换为已加载模块，接着转换为已就绪模块。首先，它通过调用 generateCodeForModule() 生成已加载模块。接着，通过 finalizeLoadedModules() 方法，所有模块变为就绪模块。

不像原先的 JIT，MCJIT::getPointerToFunction() 函数要求模块对象在调用之前就绪。因此，必须在使用之前调用 MCJIT::finalizeObject()。

LLVM 3.4 添加的新方法消除了这种限制——当使用 MCJIT 时，`getPointerToFunction()` 方法被 `getFunctionAddress()` 淘汰了。这个新方法在请求 symbol 地址之前加载和就绪模块，而不需要调用 `finalizeObject()`。

备注：注意，在原先的 JIT 中，执行引擎单独地 JIT 编译和执行各个函数。在 MCJIT 中，整个模块（所有函数）必须在任何函数执行之前被 JIT 编译。由于编译粒度变大了，我们不能再说它是基于函数的，而是基于模块的翻译引擎。

7.3.2 理解 MCJIT 如何编译模块

代码生成发生在模块对象加载阶段，由 `MCJIT::generateCodeForModule()` 方法触发，它在 `<llvm_source>/lib/ExecutionEngine/MCJIT/MCJIT.cpp` 文件中。这个方法执行下面的任务：

- 创建一个 `ObjectBuffer` 实例以存放模块对象。如果模块对象已经被加载（编译），就用 `ObjectCache` 接口获取，避免重编译。
- 假设没有之前的缓存（cache），`MCJIT::emitObject()` 就执行 MC 代码生成。结果是一个 `ObjectBufferStream` 对象（`ObjectBuffer` 子类，支持 streaming）。
- `RuntimeDyld` 动态链接器加载结果 `ObjectBuffer` 对象，并通过 `RuntimeDyld::loadObject()` 建立符号表（symbol table）。这个方法返回一个 `ObjectImage` 对象。
- 模块被标记为已加载。

对象缓冲区，缓存，图像

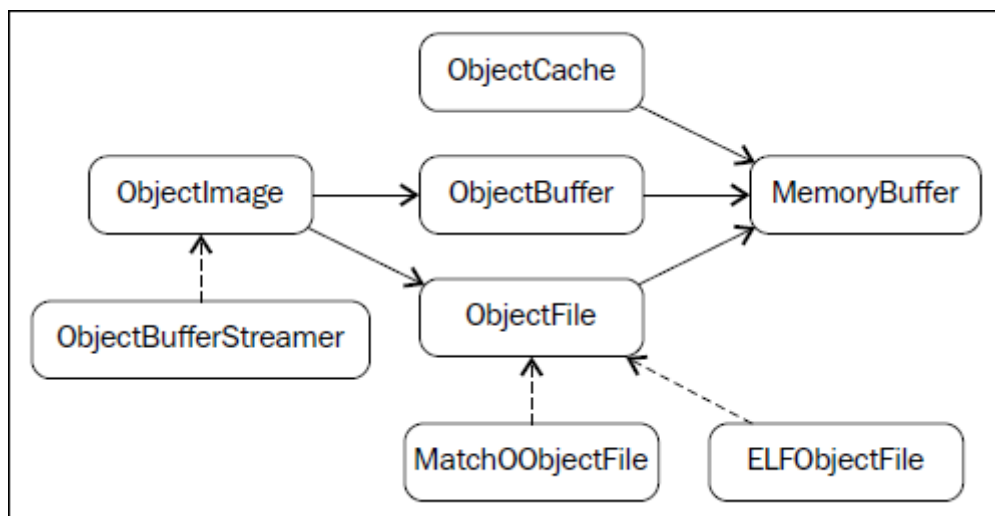
`ObjectBuffer` 类（`<llvm_source>/include/llvm/ExecutionEngine/ObjectBuffer.h`）实现了对 `MemoryBuffer` 类（`<llvm_source>/include/llvm/Support/MemoryBuffer.h`）的包装。

`MCOjectStreamer` 子类利用 `MemoryBuffer` 类输出指令和数据到内存。此外，`ObjectCache` 类直接引用 `MemoryBuffer` 实例，能从彼处获取 `ObjectBuffer`。

`ObjectBufferStream` 类是一个 `ObjectBuffer` 子类，带有附加的标准 C++ 流运算符（例如，`>>` 和 `<<`），从实现的视角来看，它让内存缓冲区的读写变得容易。

`ObjectImage` 对象（`<llvm_source>/include/llvm/ExecutionEngine/ObjectImage.h`）用于保持加载的模块，它可以直接访问 `ObjectBuffer` 和 `ObjectFile` 的引用。`ObjectFile` 对象由目标特定的目标文件类型具体化，例如 ELF、COFF、和 MachO。`ObjectFile` 对象能够从 `MemoryBuffer` 对象直接获取符号、重定位、和段。

下图说明了这些类是怎么相互关联的——实箭头表示协助，虚箭头表示继承。

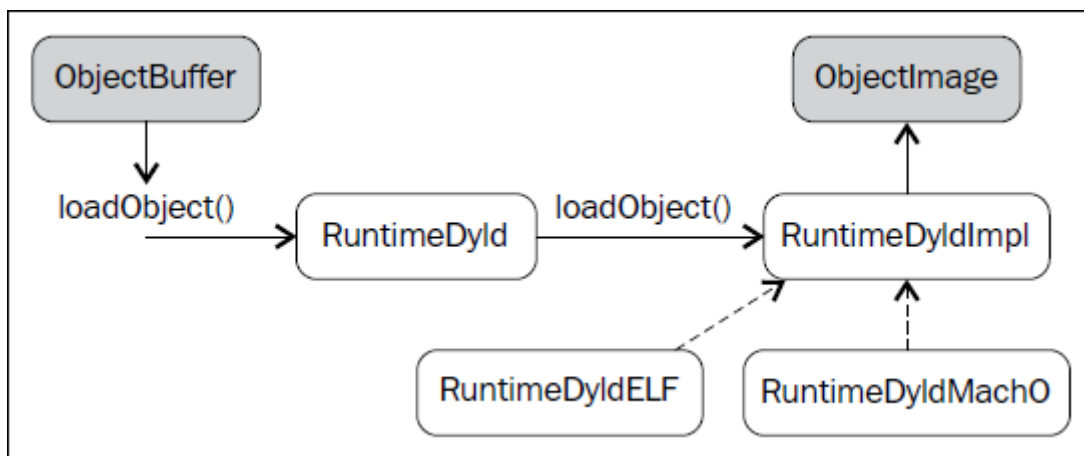


动态链接

MCJIT 加载的模块对象被表示为 `ObjectImage` 实例。如前面提到的那样，它可以透明地访问内存缓冲区，通过一个目标无关的 `ObjectFile` 接口。因此，它可以处理符号、段、和重定位。

为了生成 `ObjectImage` 对象，MCJIT 具有动态链接特性，由 `RuntimeDyld` 类提供。这个类提供了访问这些特性的公共接口，而 `RuntimeDyldImpl` 对象提供实际的实现，它由每个对象的文件类型具体化。

因此，`RuntimeDyld::loadObject()` 方法首先创建目标特定的 `RuntimeDyldImpl` 对象，然后调用 `RuntimeDyldImpl::loadObject()`。它根据 `ObjectBuffer` 生成 `ObjectImage` 对象。在这个过程中，还创建了 `ObjectFile` 对象，可以通过 `ObjectImage` 对象获取它。下图说明了这个过程：



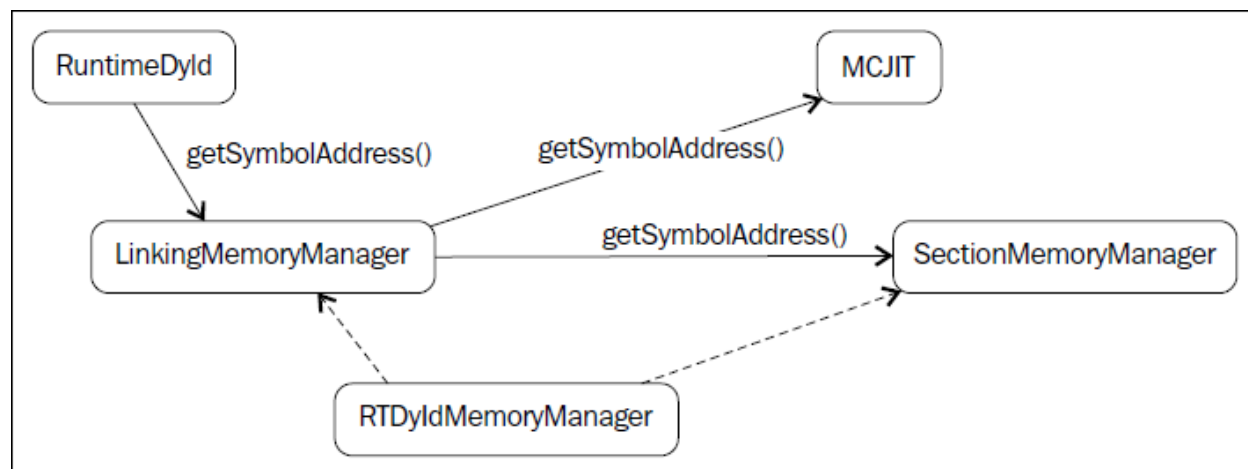
运行时 `RuntimeDyld` 动态链接器用于让模块就绪过程中解决重定位，为模块对象注册异常处理帧。回想起执行引擎方法 `getFunctionAddress()` 和 `getPointerToFunction()` 要求引擎知道符号（函数）地址。为了解决这个问题，MCJIT 还用 `RuntimeDyld` 获取任意的符号地址，通过 `RuntimeDyld::getSymbolLoadAddress()` 方法。

内存管理器

LinkingMemoryManager 类，另一个 RTDyldMemoryManager 子类，是 MCJIT 引擎所用的实际内存管理器。它聚合了一个 SectionMemoryManager 实例，向它发送委托请求。

每当 RuntimeDyld 动态链接器通过 LinkingMemoryManager::getSymbolAddress() 请求符号地址时，它有两个选择：如果符号在一个已编译的模块中是可获得的，就从 MCJIT 获取地址；否则，从外部库请求地址，它们由 SectionMemoryManager 实例加载并映射。下图说明了这个机制。参考 <llvm_source>/lib/ExecutionEngine/MCJIT/MCJIT.cpp 中的 LinkingMemoryManager::getSymbolAddress()，以了解详情。

SectionMemoryManager 实例是一个简单的管理器。作为一个 RTDyldMemoryManager 的子类，SectionMemoryManager 继承了它所有的库查询方法，但是通过直接处理低层 MemoryBlock 单元 (<llvm_source>/include/llvm/Support/Memory.h) 实现了代码和数据段的分配。



MC 代码输出

MCJIT 通过调用 MCJIT::emitObject() 执行 MC 代码输出。这个方法执行下面的任务：

- 创建一个 PassManager 对象。
- 添加一个目标布局 Pass，调用 addPassesToEmitMC() 以添加所有代码生成 Pass 和 MC 代码输出。
- 利用 PassManager::run() 方法运行所有的 Pass。结果代码存储在一个 ObjectBufferStream 对象中。
- 添加已编译的对象到 ObjectCache 实例并返回它。

MCJIT 的代码生成比原先的 JIT 更一致。不是给 JIT 提供定制的输出器和目标信息，MCJIT 透明地访问存在的 MC 基础结构的所有信息。

让对象就绪

最终，MCJIT::finalizeLoadedModules() 让模块对象就绪：重定向已解决，已加载模块被移到已就绪模块组，调用 LinkingMemoryManager::finalizeMemory() 以改变内存页面权限。对象就绪之后，MCJIT 编译的函数已准备好执行了。

7.3.3 使用 MCJIT 引擎

下面的 sum-mcjit.cpp 源文件包含了 JIT 编译 Sum 函数所必需的代码，利用 MCJIT 框架，而不是原先的 JIT。为了表明它和前面的 JIT 例子的相似之处，我们保留了原先的代码，并用布尔变量 UseMCJIT 来决定使用原先的 JIT 还是 MCJIT。因为代码和前面的 sum-jit.cpp 相当类似，我们将避免详细介绍前面的例子已经给出的代码片段。

1. 首先，包含 MCJIT 头文件，如下面的代码所示：

```
#include "llvm/ExecutionEngine/MCJIT.h"
```

2. 包含其它必需的头文件，导入 llvm 名字空间：

```
#include "llvm/ADT/OwningPtr.h"
#include "llvm/Bitcode/ReaderWriter.h"
#include "llvm/ExecutionEngine/JIT.h"
#include "llvm/IR/LLVMContext.h"
#include "llvm/IR/Module.h"
#include "llvm/Support/MemoryBuffer.h"
#include "llvm/Support/ManagedStatic.h"
#include "llvm/Support/TargetSelect.h"
#include "llvm/Support/raw_ostream.h"
#include "llvm/Support/system_error.h"
#include "llvm/Support/FileSystem.h"
using namespace llvm;
```

3. 将 UseMCJIT 设置为 true，以测试 MCJIT。设置为 false 就用原先的 JIT 运行这个例子，如下面的代码所示：

```
bool UseMCJIT = true;

int main() {
    InitializeNativeTarget();
```

4. MCJIT 需要初始化汇编解析器和打印器：

```
if (UseMCJIT) {
    InitializeNativeTargetAsmPrinter();
```

(续下页)

(接上页)

```

    InitializeNativeTargetAsmParser();
}

LLVMContext Context;
std::string ErrorMessage;
OwningPtr<MemoryBuffer> Buffer;

if (MemoryBuffer::getFile("./sum.bc", Buffer)) {
    errs() << "sum.bc not found\n";
    return -1;
}

Module *M = ParseBitcodeFile(Buffer.get(), Context, &ErrorMessage);
if (!M) {
    errs() << ErrorMessage << "\n";
    return -1;
}

```

5. 创建执行引擎，调用 `SetUseMCJIT(true)` 方法，让引擎使用 MCJIT，如下面的代码所示：

```

OwningPtr<ExecutionEngine> EE;
if (UseMCJIT)
    EE.reset(EngineBuilder(M).setUseMCJIT(true).create());
else
    EE.reset(EngineBuilder(M).create());

```

6. 原先的 JIT 需要 `Function` 引用，用于以后获取函数指针，销毁分配的内存：

```

Function* SumFn = NULL;
if (!UseMCJIT)
    SumFn = cast<Function>(M->getFunction("sum"));

```

7. 如前所述，MCJIT 淘汰了 `getPointerToFunction()`，在 MCJIT 中只能用 `getFunctionAddress()`。因此，对于各个 JIT 类别，要用正确的方法：

```

int (*Sum)(int, int) = NULL;
if (UseMCJIT)
    Sum = (int (*)(int, int)) EE->getFunctionAddress(std::string("sum"));
else
    Sum = (int (*)(int, int)) EE->getPointerToFunction(SumFn);
int res = Sum(4, 5);
outs() << "Sum result: " << res << "\n";
res = Sum(res, 6);
outs() << "Sum result: " << res << "\n";

```


8. 因为 MCJIT 一次编译整个模块，释放 Sum 函数的机器代码内存在原先的 JIT 中才有意义：

```
if (!UseMCJIT)
    EE->freeMachineCodeForFunction(SumFn);

    llvm_shutdown();
    return 0;
}
```

要编译和链接 sum-mcjit.cpp，用下面的命令：

```
$ clang++ sum-mcjit.cpp -g -O3 -rdynamic -fno-rtti $(llvm-config --cppflags --ldflags \
↳--libs jit mcjit native irreader) -o sum-mcjit
```

或者，修改第 3 章（工具和设计）的 Makefile。运行这个例子，检验输出：

```
$ ./sum-mcjit
Sum result: 9
Sum result: 15
```

7.4 使用 LLVM JIT 编译工具

LLVM 提供了一些 JIT 引擎的工具。lli 和 llvm-rtldyld 就是它们的例子。

7.4.1 使用 lli 工具

利用这一章学习的 LLVM 执行引擎，解释工具（lli）实现了一个 LLVM bitcode 解释器和 JIT 编译器。考虑下面的源文件，sum-main.c：

```
#include <stdio.h>

int sum(int a, int b) {
    return a + b;
}

int main() {
    printf("sum: %d\n", sum(2, 3) + sum(3, 4));
    return 0;
}
```

lli 工具能够运行 bitcode 文件，只要有 main 函数。用 clang 生成 sum-main.bc bitcode 文件：

```
$ clang -emit-llvm -c sum-main.c -o sum-main.bc
```

现在，通过 `lli` 利用原先的 JIT 编译引擎运行 bitcode：

```
$ lli sum-main.bc  
sum: 12
```

或者，用 MCJIT 引擎：

```
$ lli -use-mcjit sum-main.bc  
sum: 12
```

也有应用解释器的标记，它一般是很慢的：

```
$ lli -force-interpreter sum-main.bc  
sum: 12
```

7.4.2 使用 `llvm-rtdyld` 工具

`llvm-rtdyld` 工具 () 是一个非常简单的测试 MCJIT 对象加载和链接框架的工具。它能够从磁盘读取二进制目标文件，执行通过命令行指定的函数。它不作 JIT 编译和执行，但是让你能够测试和运行目标文件。

考虑下面三个 C 源代码文件：`main.c`，`add.c`，和 `sub.c`：

- `main.c`

```
int add(int a, int b);  
int sub(int a, int b);  
int main() {  
    return sub(add(3, 4), 2);  
}
```

- `add.c`

```
int add(int a, int b) {  
    return a+b;  
}
```

- `sub.c`

```
int sub(int a, int b) {  
    return a-b;  
}
```

编译它们为目标文件：

```
$ clang -c main.c -o main.o
$ clang -c add.c -o add.o
$ clang -c sub.c -o sub.o
```

利用 `llvm-rtdyld` 工具执行 `main` 函数，以 `-entry` 和 `-execute` 选项：

```
$ llvm-rtdyld -execute -entry=_main main.o add.o sub.o; echo $? loaded '_main' at:
→0x104d98000
5
```

另一个选项是，为编译了调试信息的函数打印行信息，它是 `-printline`。举例来说，看下面的命令行：

```
$ clang -g -c add.c -o add.o
$ llvm-rtdyld -printline add.o
Function: _add, Size = 20
  Line info @ 0: add.c, line: 2
  Line info @ 10: add.c, line: 3
  Line info @ 20: add.c, line: 3
```

我们看到，`llvm-rtdyld` 工具在实践中运用了 MCJIT 框架的对象抽象。`llvm-rtdyld` 工具读取一系列二进制目标文件到 `ObjectBuffer` 对象，通过 `RuntimeDyld::loadObject()` 生成 `ObjectImage` 实例。加载所有目标文件之后，由 `RuntimeDyld::resolveRelocations()` 解决重定位。接着，通过 `getSymbolAddress()` 解决入口点（entry point），并调用函数。

`llvm-rtdyld` 工具用了一个定制的内存管理器，`TrivialMemoryManager`。这是一个易于理解的简单的 `RTDyld-MemoryManager` 子类的实现。

这个了不起的概念验证工具让你理解了 MCJIT 框架涉及的基础概念。

7.5 其它的资源

通过在线文档和例子学习 LLVM JIT, 有其它资源。在 LLVM 源代码树中, `<llvm_source>/examples/HowToUseJIT` 和 `<llvm_source>/examples/ParallelJIT` 包含了简单的源代码例子，可用于学习 JIT 基础。

LLVM kaleidoscope 教程 (<http://llvm.org/docs/tutorial>) 有具体的章节介绍如何使用 JIT (<http://llvm.org/docs/tutorial/LangImpl4.html>)。

想了解更多关于 MCJIT 设计和实现的信息，请查看 <http://llvm.org/docs/MCJITDesignAndImplementation.html>。

7.6 总结

JIT 编译是一种运行时编译特性，出现于若干虚拟机环境中。在本章中，通过展示两种可得到的截然不同的实现，即原先的 JIT 和 MCJIT，我们探索了 LLVM JIT 执行引擎。此外，我们考察了两种方案的实现细节，给出了实际的例子来解释如何用 JIT 引擎编译工具。

在下一章，我们将介绍交叉编译、工具链、以及如何创建基于 LLVM 的交叉编译器。

第 8 章交叉平台编译

传统的编译器将源代码转换为本地的可执行文件。在此上下文中，本地意味着可执行文件运行的平台和编译器的平台相同，平台是如下要素的结合：硬件、操作系统、应用程序二进制接口（ABI）、系统接口的选择。这些选择定义了一种机制，用户层程序利用这种机制，与背后的系统相通信。因此，如果你使用 GNU/Linux x86 机器上的编译器，它生成的可执行文件会链接你的系统库，被定制为在完全相同的平台上运行。

交叉平台编译是一个用编译器为不同的、非本地的平台生成可执行文件的过程。如果生成的代码需要链接的库不同于你本身系统的库，一般可以通过设置编译选项来解决。然而，如果你想要部署可执行文件的目标平台和你的平台不兼容，比如用了不同的处理器架构、操作系统、ABI、或者目标文件，你需要采用交叉编译。

当为资源有限的系统开发应用程序时，交叉编译器是至关重要的。举例来说，嵌入式系统通常由低性能的处理器和有限的内存组成，由于编译过程会密集占用 CPU 和内存，在这样的系统上运行编译器，如果可能的话，是很慢的，会耽搁应用开发周期。因此，在这样的场景中，交叉编译器是极有用的工具。在这一章中，我们将讨论下面的内容：

- 对 Clang 和 GCC 交叉编译方案的比较
- 什么是工具链？
- 如何用 Clang 命令行执行交叉编译？
- 如何通过生成定制的 Clang 执行交叉编译？
- 流行的用于测试目标二进制程序的模拟器和硬件平台

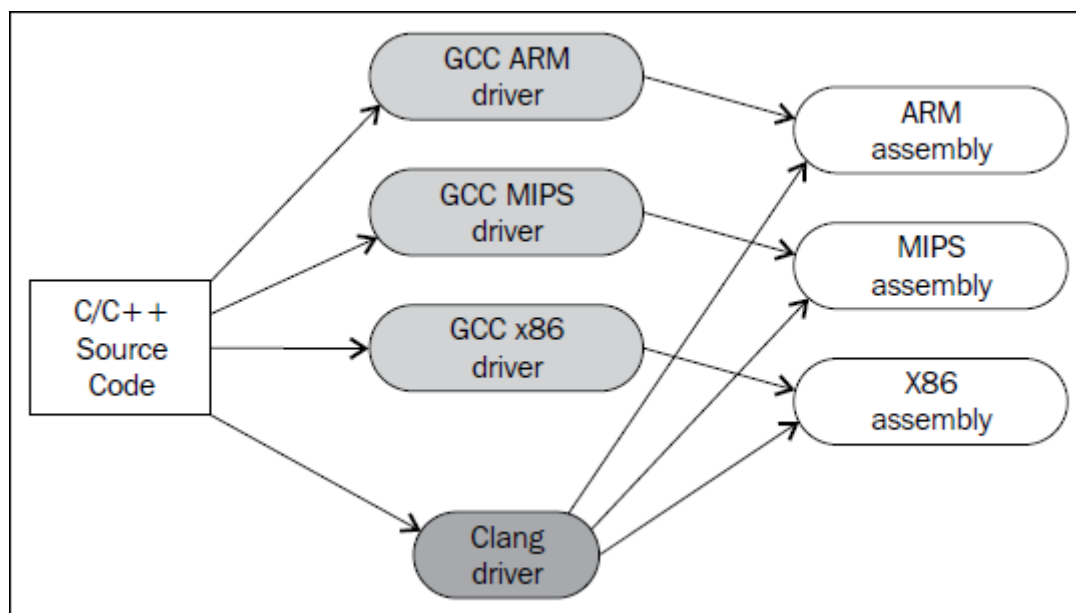
8.1 比较 GCC 和 LLVM

像 GCC 这样的编译器要支持交叉编译，必须以特别的配置编译出来，为每个目标安装不同的 GCC。通常在实践中，举例来说，会给你的 gcc 命令添加一个目标名称前缀，例如 arm-gcc，表示针对 ARM 的 GCC 交叉编译器。然而，Clang/LLVM 通过简单地开关同一个 Clang 驱动器的命令行选项，选择期望的目标、库路径、头文件、链接器、和汇编器，可以为其它目标生成代码。因此，一个 Clang 驱动器，适用所有的目标。不过，有些 LLVM 发布版不包含所有的目标，由于某种考虑，比如可执行文件的大小。另一方面，如果你自己编译 LLVM，可以选择支持哪些目标；参见第 1 章（编译和安装 LLVM）。

相比 LLVM，GCC 是一个更古老的项目，自然也是一个更成熟的项目。它支持 50 多个后端，广泛地被这些平台用作交叉编译器。然而，由于 GCC 设计的限制，它的驱动器只能以安装为单位处理单个目标库。这就是为什么，必须部署安装不同的 GCC，以为其它目标生成代码。

与此相反，Clang 驱动器默认编译和链接所有的目标库。在运行时，即使 Clang 需要知道几个目标特性，Clang/LLVM 组件可以通过目标无关的接口访问任意目标的信息，这些接口被设计用于提供任何命令行指定的目标的信息。

下图说明了 LLVM 和 GCC 是如何为不同的目标编译一份源代码的；前者动态地为截然不同的处理器生成代码，而后者需要为每个处理器安装一个不同的交叉编译器。



你也可以编译一个专用的 Clang 交叉编译器，像 GCC 那样。虽然这种选择要付出更多工夫以编译安装一个单独的 Clang/LLVM，但是它的命令行接口更易于使用。在配置的时候，用户可以提供固定的指向目标库、头文件、汇编器、和链接器的路径，避免每次执行交叉编译时都要输入大量的命令行参数。

在这一章中，我们将展示如何用 Clang 为多个平台生成代码，通过驱动器命令行参数，以及如何生成一个特殊的 Clang 交叉编译器的驱动器。

8.2 理解目标三元组

我们从三个重要的定义开始，具体如下：

- Build 表示编译交叉编译器的平台
- Host 表示交叉编译器将运行的平台
- Target 表示交叉编译器运行生成的可执行文件或者库所针对的平台

在标准的交叉编译器中，Build 和 Host 平台是相同的。目标三元组定义了 Build、Host、和 Target 平台。三元组用信息唯一地指定一个目标，此信息包括处理器架构、操作系统版本、C 库类别、和目标文件类型。

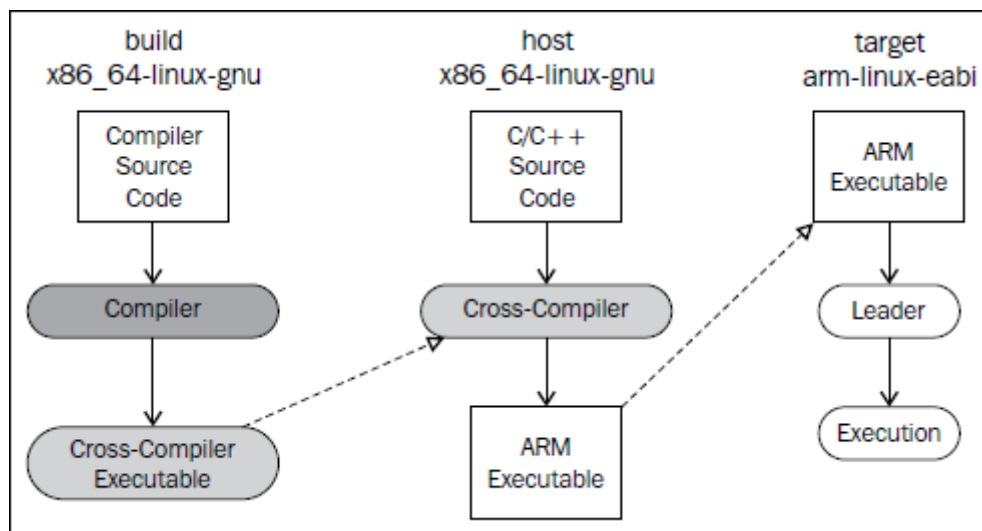
三元组的格式不是严格规定的。举例来说，GNU 工具，在格式 `<arch>-<sys/vendor>-<other>-<other>` 中，可能接受包含两个、三个、甚至四个字段的三元组，例如 `arm-linux-eabi`、`mips-linux-gnu`、`x86_64-linux-gnu`、`x86_64-apple-darwin11`、和 `sparc-elf`。Clang 努力和 GCC 保持兼容，因而认可上面的格式，但是它在内部会将任意三元组规范为自己的三元组，`<arch><sub>-<vendor>-<sys>-<abi>`。

下面的表格列出了每个 LLVM 三元组字段的可能选项；`<sub>` 字段没有包含在其中，因为它表示架构变种，例如 `armv7` 架构的 `v7`。查看 `<llvm_source>/include/llvm/ADT/Triple.h`，了解三元组详细内容。

Architecture (<arch>)	Vendor (<vendor>)	Operating system (<sys>)	Environment (<abi>)
arm, aarch64, hexagon, mips, mipsel, mips64, mips64el, msp430, ppc, ppc64, ppc64le, r600, sparc, sparcv9, systemz, tce, thumb, x86, x86_64, xcore, nvptx, nvptx64, le32, amdil, spir, and spir64	unknown, apple, pc, scei, bgp, bgq, fsl, ibm, and nvidia	unknown, auroraux, cygwin, darwin, dragonfly, freebsd, ios, kfreebsd, linux, lv2, macosx, mingw32, netbsd, openbsd, solaris, win32, haiku, minix, rtems, nacl, cnk, bitrig, aix, cuda, and nvcl	unknown, gnu, gnueabi, gnueabihf, gnueabi, gnu32, eabi, macho, android, and elf

注意，不是所有 arch、vendor、sys、和 abi 组合是有效的。每种架构支持有限数量的组合。

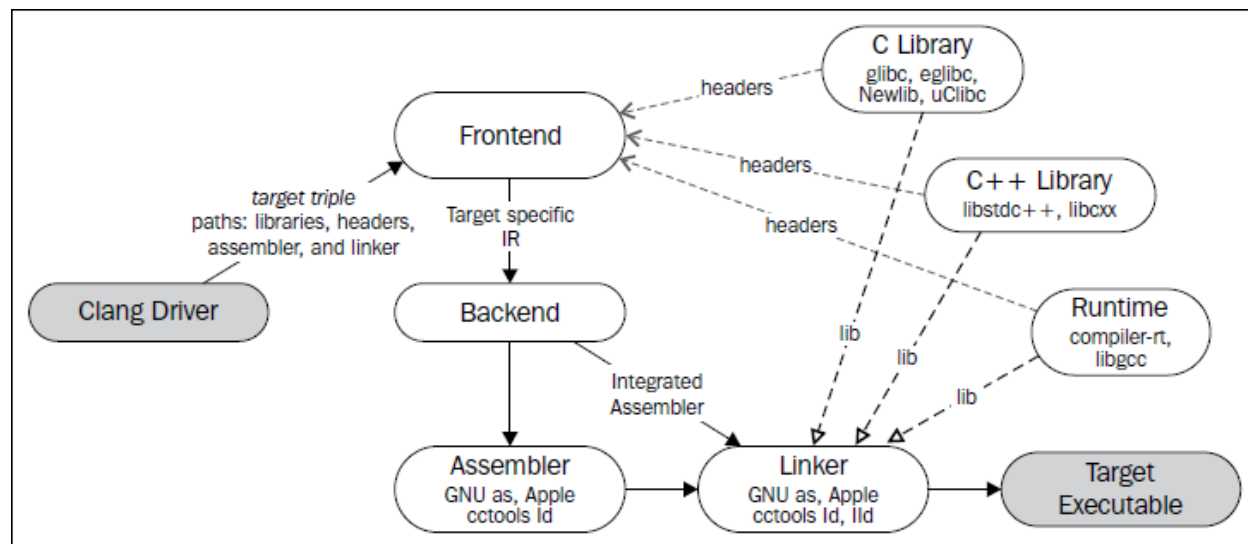
下图说明了一种 ARM 交叉编译器的概念，它在 x86 上被编译，在 x86 上运行，生成 ARM 可执行文件。好奇的读者可能想知道，如果 Host 和 Build 平台是不同的会怎样。这种组合形成加拿大型交叉编译器，过程稍微复杂一点，要求下图中的深色 Compiler 框是另一个交叉编译器，而不是本地编译器。加拿大型交叉这个名子是根据这样的事实提出的，即在当时名字提出时加拿大有三个政党，以及加拿大型交叉编译器用到三个平台。举例来说，如果你将交叉编译器发布给用户，他们期望支持的平台不同于你自己的。



8.3 准备工具链

编译器这个术语意味着一组编译相关的任务，由不同的组件来执行，例如前端、后端、汇编器、和链接器。它们之中，有些是由单独的工具实现的，而其它的是集成在一起的。然而，当为本地或者其它目标开发应用程序时，用户需要更多资源，例如平台相关的库、调试器、和执行任务的工具，举例来说，读取目标文件的工具。因此，平台制造者常常在他们的平台中为软件开发发布一批工具，从而为客户提供了一套开发工具链。

为了生成或者使用你的交叉编译器，需要知道工具链组件，以及它们之间如何交互，这是非常重要的。下图显示了成功交叉编译所必需的主要工具链组件，而后面的小节将描述每个组件：



8.3.1 标准 C 和 C++ 库

C 库是必需的，以支持标准的 C 语言功能，例如内存分配 (`malloc()/free()`)，字符串处理 (`strcmp()`)，和 IO (`printf()/scanf()`)。普遍的 C 库头文件的例子，包括 `stdio.h`、`stdlib.h`、和 `string.h`。可用的 C 库实现不止一种。GNU C 库 (`glibc`)、`newlib`、和 `uClibc` 是广为人知的例子。这些库可用于不同的目标，并且可用移植到新的目标。

类似地，C++ 标准库实现了 C++ 功能，例如输入和输出流、容器、字符串处理、和线程支持。GNU 的 `libstdc++` 和 LLVM 的 `libc++` (<http://libcxx.llvm.org>) 是实现的例子。实际上，完整的 GNU C++ 库由 `libstdc++` 和 `libsupc++` 组成。后者是一个让移植变得容易的目标相关的层级，它专门处理异常处理和 RTTI。对于除了 Mac OS X 的系统，LLVM 的 `libc++` 实现仍然依赖于第三方的 `libsupc++` 的替代物（参见第 2 章（外部项目）的 ** 介绍 `libc++` 标准库 ** 小节，以了解详情）。

交叉编译器需要知道目标 C/C++ 库和头文件的路径，这样它才能找到正确的函数原型，之后才能正确地链接。头文件要匹配已编译的库，版本和实现都有匹配，这是重要的事情。举例来说，错误配置的交叉编译器可能改为搜索本地系统的头文件，导致编译错误。

8.3.2 运行时库

每个目标都需要使用特殊的函数来模拟低层级的本地不支持的函数。例如，32 位的目标通常缺乏 64 位寄存器，无法直接处理 64 位类型。因此，目标可能使用两个 32 位寄存器并调用特殊的函数来执行简单的算术运算（加、减、乘、除）。

代码生成器生成对这些函数的调用，期望在链接的时候它们可以被找到。驱动器必须给出必需的库，而不是用户。在 GCC 中，这个功能由运行时库 `libgcc` 实现。LLVM 提供了完全替代品，称为 `compiler-rt`（见第 2 章，外部项目）。因此，Clang 驱动器在调用链接器时，使用参数 `-lgcc`，或者 `-lclang-rt`（以链接 `compiler-rt`）。再说一次，为了正确地被链接，目标特定的运行时库必须存在于路径中。

8.3.3 汇编器和链接器

汇编器和链接器通常由不同的工具提供，编译器驱动器会调用它们。举例来说，GNU Binutils 提供的汇编器和链接器支持若干个目标，对于本地目标，通常可以在系统路径中找到它们，分别命名为 `as` 和 `ld`。也有一个基于 LLVM 的链接器，但仍然是实验性的，称为 `lld` (<http://lld.llvm.org>)。

为了调用这样的工具，目标三元组被用作汇编器和链接器的名字的前缀，并且在系统的 `PATH` 变量中查找它们。举例来说，当为 `mips-linux-gnu` 生成代码时，驱动器可能会搜索 `mips-linux-gnu-as` 和 `mips-linux-gnu-ld`。根据目标三元组信息，Clang 在搜索的时候可能有所不同。

在 Clang 中，有些目标不需要调用外部的汇编器。由于 LLVM 通过 MC 层提供了直接的目标代码输出，驱动器可以使用集成的 MC 汇编器，通过选项 `integrated-as`，对于某些特定的目标，它是默认开启的。

8.3.4 Clang 前端

在第 5 章（LLVM 中间表示）中，我们解释了 Clang 输出的 LLVM IR 不是目标无关的，因为 C/C++ 语言就不是目标无关的。除了后端之外，前端也必须实现目标特定的约束。因此，你必须意识到，虽然 Clang 支持某个特定的处理器，但是，如果目标三元组不严格地匹配这个处理器，前端可能生成不完美的 LLVM IR，它可能导致 ABI 不匹配和运行时错误。

Multilib

Multilib 让用户能够在相同的平台上运行为不同的 ABI 而编译的应用程序。这个机制避免了多个交叉编译器，只要一个交叉编译器可以访问每个 ABI 变体的库和头文件的已编译的版本。举例来说，multilib 允许 soft-float 和 hard-float 库并存，就是说，一个库依赖于软件模拟浮点数算术运算，一个库依赖于处理器 FPU 处理浮点数。例如，GCC 每个 multilib 版本都有几个 libc 和 libgcc 的版本。

举例来说，在 MIPS GCC 中，multilib 库的文件夹结构的组织方式如下：

- lib/n32：这里存放 n32 库，支持 n32 MIPS ABI
- lib/n32/EL：这里存放 libgcc、libc、和 libstdc++ 的小端（little-endian）版本
- lib/n32/msoft-float：这里存放 n32 soft-float 库
- lib/n64：这里存放 n64 库，支持 n64 MIPS ABI
- lib/n64/EL：这里存放 libgcc、libc、和 libstdc++ 的小端（little-endian）版本
- lib/n64/msoft-float：这里存放 n64 soft-float 库

Clang 支持 multilib 环境，只要为库和头文件提供了正确的路径。然而，因为前端可能为有些目标的不同 ABI 生成不同的 LLVM IR，有必要核对你的路径和目标三元组，确保它们是匹配的，避免运行时错误。

8.4 Clang 命令行参数交叉编译

现在你知道了每个工具链组件，我们将展示如何将 Clang 用作交叉编译器，通过使用合适的驱动器参数。

备注：这节中的所有例子都在运行 Ubuntu 12.04 的 x86_64 机器上测试过。我们使用 Ubuntu 特定的工具下载了一些依赖软件，但是 Clang 相关的命令应该不经修改（或者稍微修改）就可以在任何其它的 OS 环境中使用。

8.4.1 驱动器的目标选项

Clang 通过 `-target=<triple>` 驱动器选项动态地选择目标三元组，而为之生成代码。除了三元组，可以用其它的选项以更精细地选择目标：

- 选项 `-march=<arch>` 选择目标的基础架构。`<arch>` 值的例子，包括 ARM 的 `armv4t`、`armv6`、`armv7`、和 `armv7f`，MIPS 的 `mips32`、`mips32r2`、`mips64`、和 `mips64r2`。这个选项还单独地选定一个默认的基础 CPU，为代码生成器所用。
- 选项 `-mcpu=<cpu>` 选择具体的 CPU。例如，`cortex-m3` 和 `cortex-a8` 是 ARM 具体的 CPU，`pentium4`、`athlon64`、和 `corei7-avx2` 是 x86 CPU。每个 CPU 有一个基础 `<arch>` 值，为目标所定义，并为驱动器所用。
- 选项 `-mfloat-abi=<abi>` 决定哪种寄存器用于存放浮点值：`soft` 或者 `hard`。如前所述，这决定了是否使用软件浮点数模拟。这还隐含了对调用惯例和其它 ABI 规范的改变。别名选项 `-msoft-float` 和 `-mhard-float` 也是可用的。注意，如果没有设定此选项，ABI 类型会遵从所选 CPU 的默认类型。

可以用 `clang-help-hidden` 参数查看其它目标特定的开关，它甚至将展示传统帮助信息所隐藏的选项。

8.4.2 依赖

我们将以 ARM 交叉编译器为活的例子演示如何用 Clang 作交叉编译。第一步是在你的系统上安装一份完整的 ARM 工具链，并识别所提供的组件。

要为拥有 `hard` 浮点数 ABI 的 ARM 安装 GCC 交叉编译器，可以用下面的命令：

```
$ apt-get install g++-4.6-arm-linux-gnueabi g++-4.6-arm-linux-gnueabi
```

要为拥有 `soft` 浮点数 ABI 的 ARM 安装 GCC 交叉编译器，可以用下面的命令：

```
$ apt-get install g++-4.6-arm-linux-gnueabi g++-4.6-arm-linux-gnueabi
```

备注：我们刚才让你安装了完整的 GCC 工具链，包括交叉编译器！为什么现在你会需要 Clang/LLVM 呢？如工具链一节解释的那样，在交叉编译期间，编译器自身充当了若干组件的组合中的一小部分，这些组件包括汇编器、链接器、和目标库。你应该寻找你的目标平台供应商准备的工具链，因为只有这个工具链才拥有正确的头文件和库，为你的目标平台所用。典型地，这份工具链也已经随 GCC 编译器发布了。我们想做的则是使用 Clang/LLVM，但是我们还依赖所有其它的工具链组件。

如果你想编译所有目标库，并自己准备整个工具链，你还需要准备操作系统 `image`，以启动目标平台。如果你自己编译系统 `image` 和工具链，你要确保两者关于目标系统所用库的版本保持一致。如果你喜欢从头编译一切，可以参考关于此的交叉 Linux 从头开始教程 (<http://trac.cross-lfs.org>)，它是一份不错的指南。

尽管 `apt-get` 会自动地安装工具链必备工具，对于基于 Clang 的 C/C++ ARM 交叉编译器，需要的和推荐的基础包如下：

- `libc6-dev-armhf-cross` 和 `libc6-dev-armel-cross`

- gcc-4.6-arm-linux-gnueabi-base 和 gcc-4.6-arm-linux-gnueabi-hf-base
- binutils-arm-linux-gnueabi 和 binutils-arm-linux-gnueabi-hf
- libgcc1-armel-cross 和 libgcc1-armhf-cross
- libstdc++6-4.6-dev-armel-cross 和 libstdc++6-4.6-dev-armhf-cross

8.4.3 交叉编译

尽管我们对于 GCC 交叉编译器本身不感兴趣，前面小节的命令安装了必需的必备工具，它们是我们的交叉编译器需要的：链接器、汇编器、库、和头文件。你可以用下面的命令为 arm-linux-gnueabi-hf 平台编译 sum.c 程序（来自第 7 章，即时编译器）：

```
$ clang --target=arm-linux-gnueabi-hf sum.c -o sum
$ file sum
sum: ELF 32-bit LSB executable, ARM, version 1 (SYSV), dynamically linked (uses
→shared libs)...
```

Clang 从 GNU arm-linux-gnueabi-hf 工具链找到了所有必需的组件，生成了最终的代码。在此例中，默认所用的架构是 armv6，但是我们可以提供更具体的 -target 参数，并且使用 -mcpu，以达到更精确的代码生成：

```
$ clang --target=armv7a-linux-gnueabi-hf -mcpu=cortex-a15 sum.c -o sum
```

安装 GCC

-target 指定的目标三元组被 Clang 用以搜索具有相同或相似前缀的 GCC 安装。如果找到了若干个候选者，Clang 会选择它认为最匹配目标的那一个：

```
$ clang --target=arm-linux-gnueabi-hf sum.c -o sum -v
clang version 3.4 (tags/RELEASE_34/final)
Target: arm-linux-gnueabi-hf
Thread model: posix
Found candidate GCC installation: /usr/lib/gcc/arm-linux-gnueabi-hf/4.6
Found candidate GCC installation: /usr/lib/gcc/arm-linux-gnueabi-hf/4.6.3
Selected GCC installation: /usr/lib/gcc/arm-linux-gnueabi-hf/4.6
(...)
```

因为一个 GCC 安装通常带有汇编器、链接器、库、和头文件，Clang 在安装中找到想要的工具链组件。通过提供系统中存在的工具链的确切名字的三元组，获得这样的路径通常是直接明了。然而，如果三元组是不同的或者不完整的，驱动器就会搜索并选择它认为最匹配的那一个：

```
$ clang --target=arm-linux sum.c -o sum -v
...
```

(续下页)

(接上页)

```
Selected GCC installation: /usr/lib/gcc/arm-linux-gnueabi/4.7
clang: Warning: unknown platform, assuming -mfloat-abi=soft
```

注意，尽管我们为 `arm-linux-gnueabi` 和 `arm-linux-gnueabihf` 安装了 GCC 工具链，驱动器选择了前者。在此例中，因为所选的平台是未知的，它假设 ABI 是 `soft-float`。

潜在的问题

如果添加 `-mfloat-abi=hard` 选项，驱动器就忽略警告信息，仍然选择 `arm-linux-gnueabi` 而不是 `arm-linux-gnueabihf`。这导致最终的可执行文件大概由于运行时错误无法运行，因为 `hard-float` 对象链接了 `soft-float` 库：

```
$ clang --target=arm-linux -mfloat-abi=hard sum.c -o sum
```

为什么不选择 `arm-linux-gnueabihf`，即使输入了 `-mfloat-abi=hard`？这是因为我们没有特别地要求 `clang` 使用 `arm-linux-gnueabihf` 工具链。如果你让驱动器作决定，它将选择找到的第一个工具链，而它可能不合乎需要。这个例子让你明白，驱动器可能会不选择最佳的选项，如果你指定的目标三元组是模糊的或不完整的，例如 `arm-linux`。

知道背后所用的工具链组件是十分重要的，以确认是否选择了正确的工具链，例如，通过使用 `###` 参数来打印 `clang` 在编译、汇编、和链接程序的过程中调用了哪些工具。

让我们尝试更模糊的目标三元组，看看究竟会发生什么。我们只使用 `-target=arm` 选项：

```
$ clang --target=arm sum.c -o sum
/tmp/sum-3bbfbc.s: Assembler message:
/tmp/sum-3bbfbc.s:1: Error: unknown pseudo-op: `.syntax'
/tmp/sum-3bbfbc.s:2: Error: unknown pseudo-op: `.cpu'
/tmp/sum-3bbfbc.s:3: Error: unknown pseudo-op: `.eabi_attribute'
(...)
```

从三元组中去除了 OS，驱动器被糊涂了，产生一个编译错误。事实上，驱动器试图用本地 (`x86_64`) 汇编器去汇编 ARM 汇编语言。由于目标三元组是相当不完整的，没有 OS 信息，对于驱动器来说，我们的 `arm-linux` 工具链不是满意的匹配，这样它就采用了系统汇编器。

8.4.4 修改系统根目录

通过查找系统中存在的具有给定三元组的 GCC 交叉编译器，在 GCC 安装目录中扫描一系列已知的前缀（参见 `<llvm_source>/tools/clang/lib/Driver/ToolChains.cpp`），驱动器能够找到支持目标的工具链。

对于某些别的情况——不正确格式的三元组或者不存在的 GCC 交叉编译器——为了使用可用的工具链组件，必须告诉驱动器特别的选项。例如，`-sysroot` 选项修改基础目录，Clang 在其中搜索工具链组件，每当目标三元组没有提供足够的信息时，就可用这个选项。类似地，可以用 `-gcc-toolchain=<value>` 指定你想用的一个具体的工具链的文件夹。

在我们的系统上安装的 ARM 工具链中，为 arm-linux-gnueabi 三元组所选的 GCC 安装路径是/usr/lib/gcc/arm-linux-gnueabi/4.6.3。从这个目录，Clang 到达其它的路径以访问库、头文件、汇编器、和链接器。一个它可到达的路径是/usr/arm-linux-gnueabi，其中包含下面的子目录：

```
$ ls /usr/arm-linux-gnueabi
bin  include  lib  usr
```

这些文件夹中的工具链组件的组织方式，跟文件系统的/bin、/include、/lib、和/usr 根文件夹中的本地工具链组件一样。考虑我们想要为带有 cortex A9 CPU 的 armv7-linux 生成代码，不依靠驱动器自动地为我们寻找组件。只要我们知道 arm-linux-gnueabi 的组件在何处，我就可以为驱动器提供-sysroot 参数：

```
$ PATH=/usr/arm-linux-gnueabi/bin:$PATH /p/cross/bin/clang --target=armv7a-linux --
→sysroot=/usr/arm-linux-gnueabi -mcpu=cortex-a9 -mfloat-abi=soft sum.c -o sum
```

再一次，这是非常有用的，当有可用的工具链组件而没有 GCC 实体安装的时候。为什么这个方法是可行的？下面列出了三个主要的理由：

- armv7a-linux：armv7a 触发为 ARM 和 linux 的代码生成。它做的事情，其中之一就是告诉驱动器使用 GNU 汇编器和链接器的调用语法。如果没有指定 OS，Clang 默认采用 Darwin 汇编器语法，导致一个汇编器错误。
- /usr、/lib、和/usr/include 文件夹是编译器搜索库和头文件的默认位置。选项-sysroot 覆盖了驱动器默认设置，查看/usr/arm-linux-gnueabi 以寻找这些目录，而不是系统根目录。
- PATH 环境变量被修改了，以避免使用 as 和 ld 的默认版本。然后我们强制驱动器首先查看路径/usr/arm-linux-gnueabi，在其中找到了 ARM 版本的 as 和 ld。

8.5 生成一个 Clang 交叉编译器

Clang 支持动态地为任意目标生成代码，如前面小节看到的那样。但是，生成一个目标专用的 Clang 交叉编译器的理由是存在的：

- 假如用户不想使用长长的命令行来调用驱动器
- 假如制造者想交付给客户一个平台特定的基于 Clang 的工具链

8.5.1 配置选项

LLVM 配置系统中，协助交叉编译器生成的选项如下：

- --target：这个选项指定了默认目标三元组，Clang 交叉编译器为之生成代码。这关联早先我们定义的 target、host、和 build 概念。选项-host 和-build 也是可用的，但是配置脚本估计了它们的值——两者都指向本地平台。
- --enable-targets：这个选项指定安装将支持的目标。如果省略了，将支持所有目标。记住，必须用前面解释的命令行选项选择不同于默认值的目标，默认值是由-target 指定的。

- `--with-c-include-dirs`: 这个选项指定目录列表，交叉编译器应在其中搜索头文件。这个选项避免了过度地使用 `-I` 来定位目标特定的库，它们可能不在规范的路径中。此外，这些目录先于系统默认目录被搜索。
- `--with-gcc-toolchain`: 这个选项指定已经存在于系统中的目标 GCC 工具链。这个选项定位了工具链组件，交叉编译器固定住它们，就像用永久的 `-gcc-toolchain` 选项。
- `--with-default-sysroot`: 这个选项为交叉编译器执行的所有编译器调用添加 `-sysroot` 选项。

用 `<llvm_source>/configure --help` 查看所有 LLVM/Clang 配置选项。额外（隐藏）的配置选项可用于考察目标特定的特性，例如 `-with-cpu`、`-with-float`、`-with-abi`、和 `-with-fpu`。

8.5.2 编译和安装你的基于 Clang 的交叉编译器

配置、编译、和安装交叉编译器的方法和编译 LLVM 和 Clang 的传统方法非常类似，后者已在第 1 章（编译和安装 LLVM）中解释过了。

因此，假设源代码已准备好，就可以用下面的命令，默认以 Cortex-A9 为目标，生成一个 LLVM ARM 交叉编译器：

```
$ cd <llvm_build_dir>
$ <PATH_TO_SOURCE>/configure --enable-target=arm --disable-optimized --prefix=/usr/
→local/llvm-arm --target=armv7a-unknown-linux-gnueabi
$ make && sudo make install
$ export PATH=$PATH:/usr/local/llvm-arm
$ armv7a-unknown-linux-gnueabi-clang sum.c -o sum
$ file sum
sum: ELF 32-bit LSB executable, ARM, version 1 (SYSV), dynamically linked (uses
→shared libs)...
```

记得在“理解目标三元组”一节，GCC 兼容的目标三元组可以有多至四个元素，但是一些工具接受元素较少的三元组。至于 LLVM 所用的配置脚本，它是由 GNU 自动工具生成的，它期望目标三元组包含全部四个元素，其中第二个元素是厂商信息。由于我们的平台没有具体的厂商，我们将我们的三元组扩展为 `armv7a-unknown-linux-gnueabi`。如果我们在此处坚持使用三个元素的三元组，配置脚本会失败。

不需要用额外的选项来检测工具链，因为 Clang 照常会查找 GCC 安装。

假设你编译并安装了另外的 ARM 库和头文件，分别位于 `/opt/arm-extra-libs/lib` 和 `/opt/arm-extra-libs/include` 目录。通过使用 `--with-c-include-dirs=/opt/arm-extra-libs/include`，可以永久地将这个目录添加到 Clang 头文件搜索路径；为了正确地链接，`-L/opt/arm-extra-libs/lib` 还是需要加的。

```
$ <PATH_TO_SOURCE>/configure --enable-target=arm --disable-optimized --prefix=/usr/
→local/llvm-arm --target=armv7a-unknown-linux-gnueabi --with-c-include-dirs=/opt/arm-
→extra-libs/include
```

类似地，我们可以添加 `sysroot` (`-sysroot`) 选项，还指定 GCC 工具链 (`--with-gcc-toolchain`)，让驱动器总是使用它们。对于所选的 ARM 三元组，这是冗余的，但是可能对其它目标有用：

```
$ <PATH_TO_SOURCE>/configure --enable-target=arm --disable-optimized --prefix=/usr/  
↪ local/llvm-arm --target=armv7a-unknown-linux-gnueabi --with-gcc-toolchain=arm-linux-  
↪ gnueabi --with-default-sysroot=/usr/arm-linux-gnueabi
```

8.5.3 别的编译方法

我们可以用其它的工具生成基于 LLVM/Clang 的工具链，或者用 LLVM 中其它的编译系统。另一个可选的方法是创建一个封装使过程变得容易。

Ninja

生成交叉编译器的一个可选方法是使用 CMake 和 Ninja。Ninja 项目的意图是成为一个小而快的编译系统。

不是以传统的配置和编译步骤来生成交叉编译器，而是用特别的 CMake 选项为 Ninja 生成适合的编译指令，然后 Ninja 为想要的目标编译并安装交叉编译器。

关于如何应用这个方法的说明和文档见 <http://llvm.org/docs/HowToCrossCompileLLVM.html>。

ELLCC

ELLCC 工具是一个基于 LLVM 的框架，用于为嵌入式目标生成工具链。

它致力于为交叉编译器的生成和使用创建简易的资源。它是可扩展的，支持新的目标配置，开发者易于用他们的程序多目标化。

ELLCC 还编译并安装若干个工具链组件，包括调试器和平台测试 QEMU（如果可用）。

ecc 工具是最终可用的交叉编译器。它在 Clang 交叉编译器上建了一层，接受 GCC 和 Clang 兼容的命令行选项，为任意支持的目标编译程序。你可以在 <http://ellcc.org> 了解更多。

EmbToolkit

嵌入式系统工具包是另一个为嵌入式系统生成工具链的框架。它支持生成基于 Clang 或 LLVM 的工具链，同时编译它的组件并提供一个根文件系统。

它为组件选择提供 ncurses 和 GUI 接口。你可以在 <https://www.embtoolkit.org> 了解更多详情。

8.6 测试

检验交叉编译是否成功的最合理的方式是在真实的目标平台上运行结果可执行文件。然而，当真实的目标不可用或承担不起时，可以采用几个仿真方法来测试你的程序。

8.6.1 开发板

有若干种开发板，适用于众多平台。如今，开发板是买得起的，可以在网上买到。例如，可以找到 ARM 开发板，从简单的 Cortex-M 系列处理器到多核 Cortex-A 系列。

外围设备组件多样，但是在这些板上通常都有网卡、Wi-Fi、USB、和内存卡。因此，交叉编译的应用程序可以通过网络、USB 传输，或者写到闪存卡上并且在裸机或者嵌入式 Linux/FreeBSD 系统上执行。

这样的开发板的部分例子如下：

Name	Features	Architecture/Processor	Link
Panda Board	Linux, Android, Ubuntu	ARM, Dual Core Cortex A9	http://pandaboard.org/
Beagle Board	Linux, Android, Ubuntu	ARM, Cortex A8	http://beagleboard.org/
SEAD-3	Linux	MIPS M14K	http://www.timesys.com/supported/processors/mips
Carambola-2	Linux	MIPS 24K	http://8devices.com/carambola-2

还有很多带有 ARM 和 MIPS 处理器的移动电话，可运行自带开发软件包的 Android。还可以尝试运行 Clang。

8.6.2 仿真器

制造商为其处理器开发仿真器是十分常见的，因为软件开发周期甚至在物理平台就绪之前就开始了。带有仿真器的工具链发布给客户，或者用于内部产品测试。

测试交叉编译的程序的一个方法，就是利用这些制造商提供的环境。然而，也有几个开源的仿真器，针对一定数量的架构和处理器。QEMU 是一个开源仿真器，支持用户和系统仿真。

在用户仿真模式，QEMU 能够仿真孤立的在当前平台上为其它目标编译的可执行文件。例如，用 Clang 编译和链接的 ARM 可执行文件，大概能在 ARM-QEMU 用户仿真器上即买即用。

系统仿真器重现了整个系统的行为，包括外围设备和多核。由于仿真了完整的启动过程，需要一个操作系统。QEMU 仿真的完整的开发板是存在的。用它测试裸机目标或者测试交互外围设备的程序也是理想的。

QEMU 支持多种架构的不同处理器变种，包括 ARM、MIPS、OpenRISC、SPARC、Alpha、和 MicroBlaze。你可以在 <http://qemu-project.org> 了解更多。

8.7 额外的资源

官方的 Clang 文档包含非常有价值的关于 Clang 作为交叉编译器的信息。见 <http://clang.llvm.org/docs/CrossCompilation.html>。

8.8 总结

对于为其它平台开发应用程序来说，交叉编译器是一个重要的工具。Clang 从设计的角度出发，让交叉编译成为可随意获得的特性，让驱动器可以动态地执行交叉编译。

在这一章中，我们介绍了构成交叉编译环境的元素，以及 Clang 如何与之交互以产生目标可执行文件。我们还看到，Clang 交叉编译器在某些场景中可能仍然是有用的。我们说明了如何编译、安装、和使用交叉编译器。

在下一章中，我们将介绍 Clang 静态编译器，展示如何搜索大型的 code base 以发现常见的漏洞。

第 9 章 Clang 静态分析器

对于策划构建抽象的装置，人类会感到困难，因为人类难于估量工作量的大小，难于量化工作量。不出意料，由于无法处理不断增加的复杂度，软件项目具有显著的失败历史。如果编译复杂的软件需要大量的协调和组织，那么维护它恐怕是更困难的挑战。

还是这样，软件越陈旧就越难维护。这典型地反映出在不同时期为之付出努力的程序员具有迥异的观点。当一个新程序员负责维护一个陈旧的软件的时候，通常的做法是简单地将不易理解的陈旧的代码部分严密地包裹起来，隔离软件，让它成为一个不可修改的程序库。

软件代码如此复杂，使得程序员需要一种新的工具以帮助他们解决隐藏的漏洞。Clang 静态分析器的目的，在于提供一种自动的方法，在编译之前分析大型软件代码，助人类一臂之力，以检测 C、C++、或者 Objective-C 项目中的各种各样的常见的漏洞。在这一章中，我们将讨论以下内容：

- 相比经典的编译器工具，Clang 静态分析器输出的警告信息有何不同
- 如何在简单的项目中使用 Clang 静态分析器
- 如何使用 scan-build 工具处理大型的真实世界的项目
- 如何扩展 Clang 静态分析器，加入你自己的漏洞检查器

9.1 理解静态分析器的角色

在总体的 LLVM 设计中，如果一个项目操作原始的源代码 (C/C++)，它就属于 Clang 前端，因为根据 LLVM IR 恢复源代码层信息是困难的。最有意思的基于 Clang 的工具之一是 Clang 静态分析器，这个项目利用一套检查器来生成详细的漏洞报告，类似于传统的编译器警告在更小的范围所做的事情。每个检查器检测对一个具体的规则的违背。

如同经典的警告，静态分析器帮助程序员在开发周期的早期发现漏洞，不需要将漏洞检测延迟到运行时。分析是在解析之后进一步编译之前做的。另一方面，这个工具可能需要很多时间处理大量代码，这就是一个很好的理由解释了为什么它没有被集成到典型的编译流程中去。举例来说，静态分析器可能独自花数个小时去处理整个 LLVM 源代码并运行所有的检查器。

Clang 静态分析器至少有两个已知的竞争者：Fortify 和 Coverity。Hewlett Packard (HP) 提供了前者，而 Synopsis 提供了后者。每个工具都有它的优势和局限，但是只有 Clang 是开源的，这允许我们 hack 它，理解它如何工作，这就是这一章的目的。

9.1.1 对比经典的警告和 Clang 静态分析器

Clang 静态分析器用到的算法具有指数级时间复杂度，这意味着，当被分析的程序单元增长时，处理它所需的时间可能变得非常大。如同在实践中用到的许多指数级时间复杂的算法，它是有界限的，这意味着能够通过应用问题特定的技巧减少执行时间和内存，尽管这不足以让它变成多项式时间复杂度。

指数级时间复杂度的本质解释了这个工具的一个最大的局限：它一次只能分析单个编译单元，不能执行模块间分析，或者处理整个程序。尽管如此，这是一个能力很强的工具，因为它依靠符号执行引擎。

为了举例说明符号执行引擎如何帮助程序员找出错综复杂的漏洞，我们先展示一个简单的漏洞，大多数编译器可以容易地检测到它并输出警告。看下面的代码：

```
#include <stdio.h>
void main() {
    int i;
    printf ("%d", i);
}
```

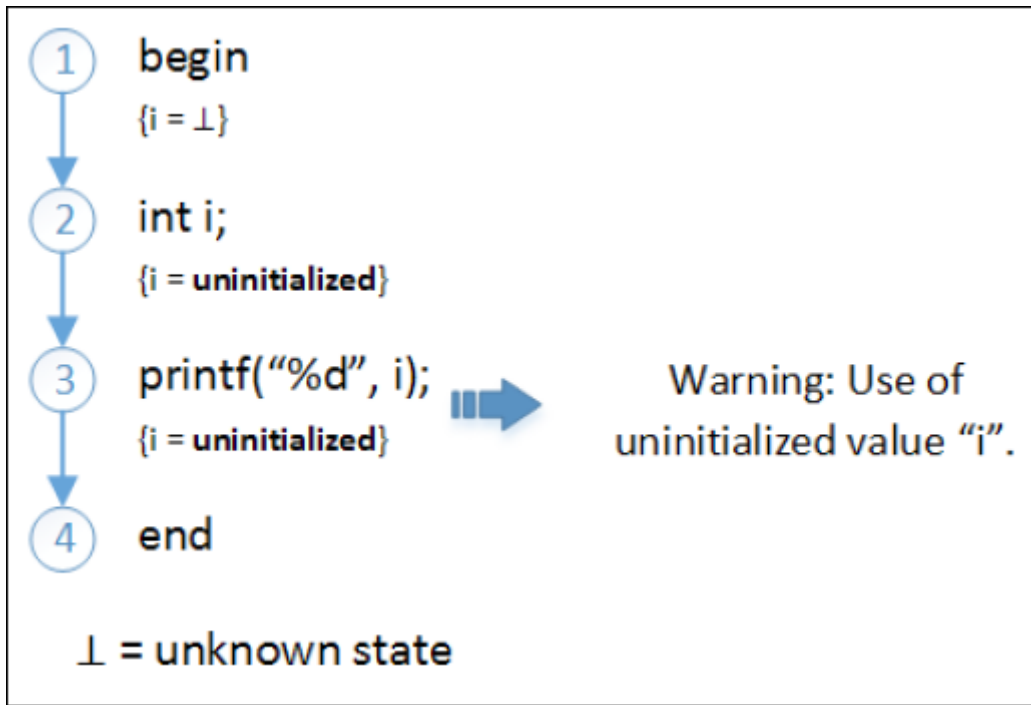
在此代码中，我们用了一个未初始化的变量，会导致程序的输出依赖于我们不能控制和预测的参数，诸如程序执行之前的内存内容，导致出乎意料的程序行为。因此，一个简单的自动检查能够避免在调试中的巨大麻烦。

如果你熟悉编译器分析技术，你可能已经注意到，我们可以运用前向数据流分析实现这种检查，它利用联合汇聚算子传播每个变量的状态，它是否被初始化。前向数据流分析传播关于每个基本块的变量的状态信息，从函数的第一个基本块开始，将此信息推向后继基本块。汇聚算子决定如何合并多个前面的基本块的信息。联合汇聚算子将设置基本块的属性，为每个前面的基本块集合的联合结果。

在此分析中，如果一个未初始化的定义到达一处使用，我们应该触发一个编译器警告。为此目的，我们的数据流框架将为程序中每个变量赋以如下状态：

- \perp 符号，当我们不知道任何关于它的信息（未知状态）
- 已初始化符号，当我们知道变量被初始化了
- 未初始化符号，当我们确定变量未初始化
- T 符号，当变量可能已初始化或者未初始化（这表示我们不确定）

下面的示意图显示了对我们给出的简单 C 程序的数据流分析：



我们看到，信息轻松地传播着，穿过代码行。当它到达使用 `i` 的 `printf` 语句时，框架检查关于这个变量我们知道什么，答案是未初始化，这为输出警告提供了充分的证据。

由于这种数据流分析依靠多项式时间复杂度算法，它非常快。

为了见识这个简单的分析如何会不准确，让我们认识 Joe，一个程序员，他精通设计不可检测的错误的艺术。Joe 可以非常轻松地迷惑检测器，聪明地在单独的程序路径中模糊实际的变量状态。让我们看一下 Joe 的一个例子。

```

#include <stdio.h>
void my_function(int unknownvalue) {
    int schroedinger_integer;
    if (unknownvalue)
        schroedinger_integer = 5;
    printf("hi");
    if (!unknownvalue)
        printf("%d", schroedinger_integer);
}
  
```

现在让我们看一下我们的数据流框架如何为这个程序计算变量的状态：



我们看到，在节点 4 处，变量第一次被初始化（粗体显示）。然而，有两条不同的路径到达节点 5：节点 3 处的 if 语句的 true 和 false 分支。在一条分支，变量 `schroedinger_integer` 未被初始化，而在另一条分支，它被初始化了。汇聚算子决定如何求和前驱的结果。我们的联合算子会尝试两份数据位，将 `schroedinger_integer` 声明为 T（任何一个）。

当检测器检查使用 `schroedinger_integer` 的节点 7 的时候，不能确定代码是否有漏洞，因为根据此数据流分析，`schroedinger_integer` 可能或可能没有被初始化。换句话说，它完全是状态重叠，已初始化或者未初始化。我们的简单检测器可以尝试警告人们一个未初始化的值被使用了，在这种情况下，它会正确地指出漏洞。但是，如果 Joe 的代码的上一次检查所用的条件变为 `if (unknownvalue)`，输出警告就是一个误报，因为现在它经过了 `schroedinger_integer` 确实被初始化的路径。

我们的检测器发生了丢失精确性，因为数据流框架不是路径敏感的，不能为每个可能执行的路径所发生的事情建模。

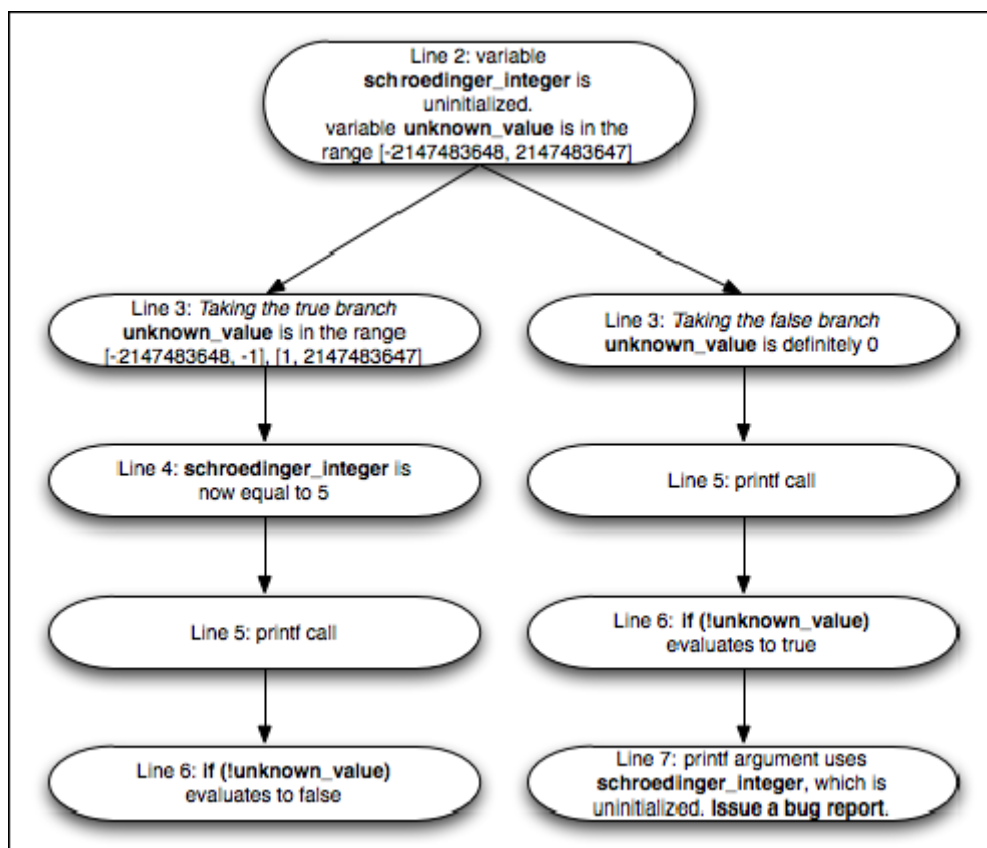
误报是非常讨厌的，因为它们迷惑了程序员，受到警告的代码并不包含实际的错误，让报告实际错误的警告变得晦涩。在现实中，如果一个检测器产生了即使少量误报的警告，程序员也很可能忽略全部警告。

9.1.2 符号化执行引擎的力量

当简单的数据流不足以提供程序的准确信息的时候，符号化执行引擎就发挥作用了。它建造一个可到达程序状态图，能够推理全部可能的代码执行路径，当程序运行时它们可能被走到。记得调试程序时，你只会练习一个路径。当你用一个强大的虚拟机调试程序寻找内存泄漏时，例如 valgrind 虚拟机，也只是练习一个路径。

相反地，符号化执行引擎能够练习所有路径，而不实际运行你的代码。这是非常强大的特性，但是需要大的运行时来处理程序。

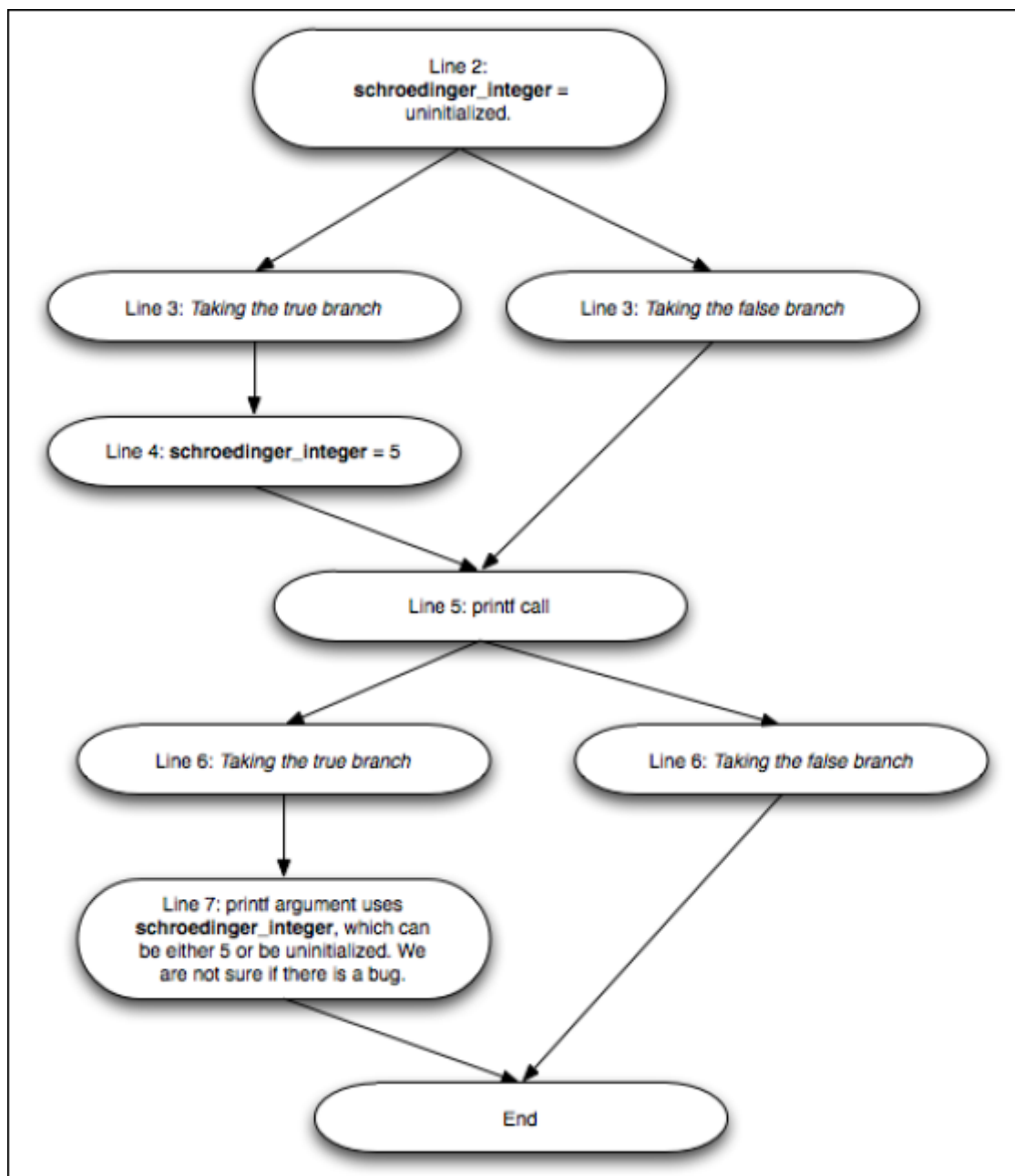
正如经典的数据流框架，引擎按照它将执行每个语句的顺序遍历程序，找到每个变量并赋给它们初始状态。当到达一个控制流改变的构造时，不同之处出现了：引擎将路径一分为二，继续对每个路径单独地分析。这个图称为可到达程序状态图，下面的示意图显示了一个简单的例子，揭示引擎会怎样推理 Joe 的代码：



在此例中，第 6 行，第一个 if 语句将可到达状态图分叉为两条不同的路径：在一条路径中，`unknown_value` 是非零，而在另一条中，`unknown_value` 肯定是零。从此处开始，引擎会处理这个关于 `unknown_value` 的重要的约束，用它决定下一步选择哪一个分支。

让我们来比较可到达程序状态图和相同代码的显示控制流的图，即控制流图，附带着数据流方程提供给我们的经典的推理。看下面的示意图：

你注意到的第一件事，是 CFG 可能分叉以表达控制流改变，但是它也合并节点以避免在可到达程序状态图中看到的组合爆炸。当它合并时，数据流分析可以用联合或者相交决定来合并来自不同路径的信息（第 5 行的节点）。如果它用联合，我们就得知 `schroedinger_integer` 既未初始化，又等于 5，如我们的上个例子。如果



它用相交，我们就无法得到关于 `schroedinger_integer` 的信息（未知状态）。

经典的数据流分析必需合并数据，这是符号化执行引擎所没有的一个限制。这让我们能够得到精确得多的结果，和用若干输入测试你的程序所得到的不相上下，但是以更多的运行时间和内存消耗为代价。

9.2 测试静态分析器

在这一节，我们将探索如何在实践中运用 Clang 静态分析器。

9.2.1 使用驱动器和使用编译器

在测试静态分析器之前，你应该始终记得，命令行 `clang -cc1` 会直接引用编译器，而使用命令 `clang` 会触发编译器驱动器。驱动器负责精心安排编译中涉及的所有其它的 LLVM 程序的执行，但是它也负责提供关于你的系统的充分的参数。

有些开发者喜欢直接使用编译器，这样有时候可能找不到系统头文件，或者不知道怎么配置其它参数，而只有 Clang 驱动器知道这些。另一方面，编译器可能设置独有的开发者选项，以让我们能够调试程序，看到内部发生的事情。让我们检验如何用两种方法检查一个源代码文件。

Compiler	<code>clang -cc1 -analyze -analyzer-checker=<package> <file></code>
Driver	<code>clang -analyze -Xanalyzer -analyzerchecker=<package> <file></code>

我们用 `<file>` 表示你想要分析的源代码文件，而 `<package>` 标签让你能够选择一批具体的头文件。

当使用驱动器时，注意 `-analyze` 参数会触发静态分析器。然而，`-Xanalyzer` 参数将下一个参数直接发送给编译器，让你能够设置具体的参数。由于驱动器是中介人，在整个示例过程中，我们将直接使用编译器。此外，在我们的简单的例子中，直接使用编译器应该满足需求了。如果你感觉你需要驱动器以官方的方式使用检查器，记得使用驱动器，并首先输入 `-Xanalyzer` 选项，后面跟着我们送给编译器的每个参数。

9.2.2 了解可用的检查器

检查器是静态分析器能够在你的代码上执行的单个分析单元。静态分析器允许你选择适合你的需求的检查器的任意子集，或者全部开启它们。

如果你没有安装 Clang，请看第 1 章（编译和安装 LLVM）的安装说明。要想得到已安装的检查器的列表，运行下面的命令：

```
$ clang -cc1 -analyzer-checker-help
```

它将打印已安装的检查器的长长的列表，显示所有你可以从 Clang 得到的即开即用的分析。现在让我们看看 `-analyzer-checker-help` 命令的输出：

OVERVIEW: Clang Static Analyzer Checkers List

USAGE: `-analyzer-checker <CHECKER or PACKAGE,...>`

CHECKERS:

`alpha.core.BoolAssignment` Warn about assigning non-`{0,1}` values to Boolean variables

检查器的名字服从规范的 `<package>.<subpackage>.<checker>` 形式，为使用者提供一种简单的方法以只运行一组特定的相关检查器。

在下面的表中，我们列出了最重要的 package，以及每个 package 的检查器例子的列表。

Package Name	Content	Examples
alpha	Checkers that are currently in development	<code>alpha.core.BoolAssignment</code> , <code>alpha.security.MallocOverflow</code> , <code>alpha.unix.cstring.NotNullTerminated</code>
core	Basic checkers that are applicable in a universal context	<code>core.NullDereference</code> , <code>core.DivideZero</code> , <code>core.StackAddressEscape</code>
cplusplus	A single checker for C++ memory allocation (others are currently in alpha)	<code>cplusplus.NewDelete</code>
debug	Checkers that output debug information of the static analyzer	<code>debug.DumpCFG</code> , <code>debug.DumpDominators</code> , <code>debug.ViewExplodedGraph</code>
llvm	A single checker that checks whether a code follows LLVM coding standards or not	<code>llvm.Conventions</code>
osx	Checkers that are specific for programs developed for Mac OS X	<code>osx.API</code> , <code>osx.cocoa.ClassRelease</code> , <code>osx.cocoa.NonNilReturnValue</code> , <code>osx.coreFoundation.CFError</code>
security	Checkers for code that introduces security vulnerabilities	<code>security.FloatLoopCounter</code> , <code>security.insecureAPI.UncheckedReturn</code> , <code>security.insecureAPI.gets</code> , <code>security.insecureAPI.strcpy</code>
unix	Checkers that are specific to programs developed for UNIX systems	<code>unix.API</code> , <code>unix.Malloc</code> , <code>unix.MallocSizeof</code> , <code>unix.MismatchedDeallocator</code>

让我们运行 Joe 的代码，它意图愚弄大多数编译器用到的简单分析过程。首先，我们试试经典的警告方法。为此，我们简单地运行 Clang 驱动器，让它不进行编译，只执行语法检查：

```
$ clang -fsyntax-only joe.c
```

选项 `syntax-only`，用于打印警告，检查语法错误，但是它没有检测到任何问题。现在，是时候测试符号化执行引擎是怎么应付的：

```
$ clang -cc1 -analyze -analyzer-checker=core joe.c
```

可选地，如果前面的命令行要求你指定头文件位置，就使用驱动器，如下：

```
$ clang --analyze -Xanalyzer -analyzer-checker=core joe.c
./joe.c:10:5: warning: Function call argument is an uninitialized value
printf("%d", schroedinger_integer);
^~~~~~
1 warning generated.
```

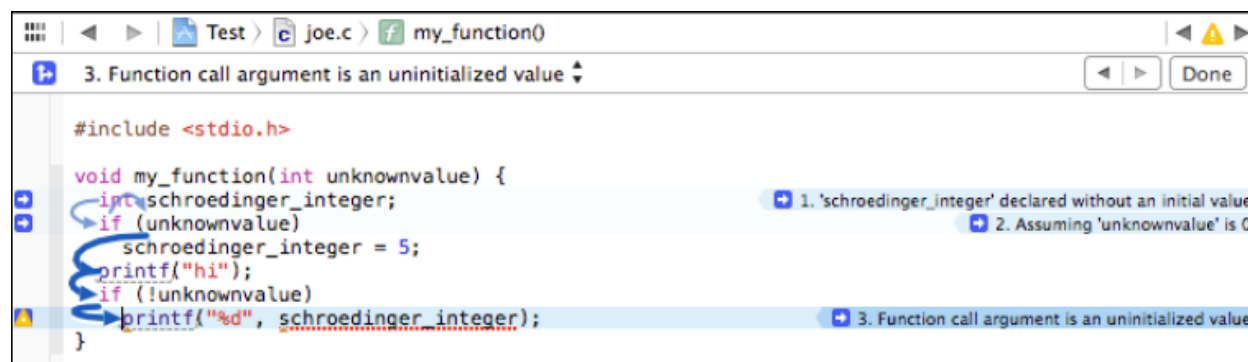
就在当场！记住，`analyzer-checker` 选项期待检查器的全称，或者检查器的整个 `package` 的名字。我们选择使用了 `core` 检查器的整个 `package`，但是我们可以只用具体的检查器 `core.CallAndMessage`，它检查函数调用的参数。

注意，所有静态分析器命令都以 `clang -cc1 -analyzer` 开始；因此，如果你想知道分析器支持的所有命令，可以用下面的命令：

```
$ clang -cc1 -help | grep analyzer
```

9.2.3 在 Xcode IDE 中使用静态分析器

如果你使用 Apple Xcode IDE，你可以在其中使用静态分析器。首先你需要打开一个项目，在 `Product` 菜单中选择菜单项 `Analyze`。你将看到，Clang 静态分析器给出了漏洞发生的确切路径，让 IDE 能够为程序员将它高亮出来，如下面的截屏所示：



分析器能够以 `plist` 格式导出信息，然后 Xcode 解释此信息，并以用户友好的方式将它显示出来。

9.2.4 在 HTML 中生成图形化报告

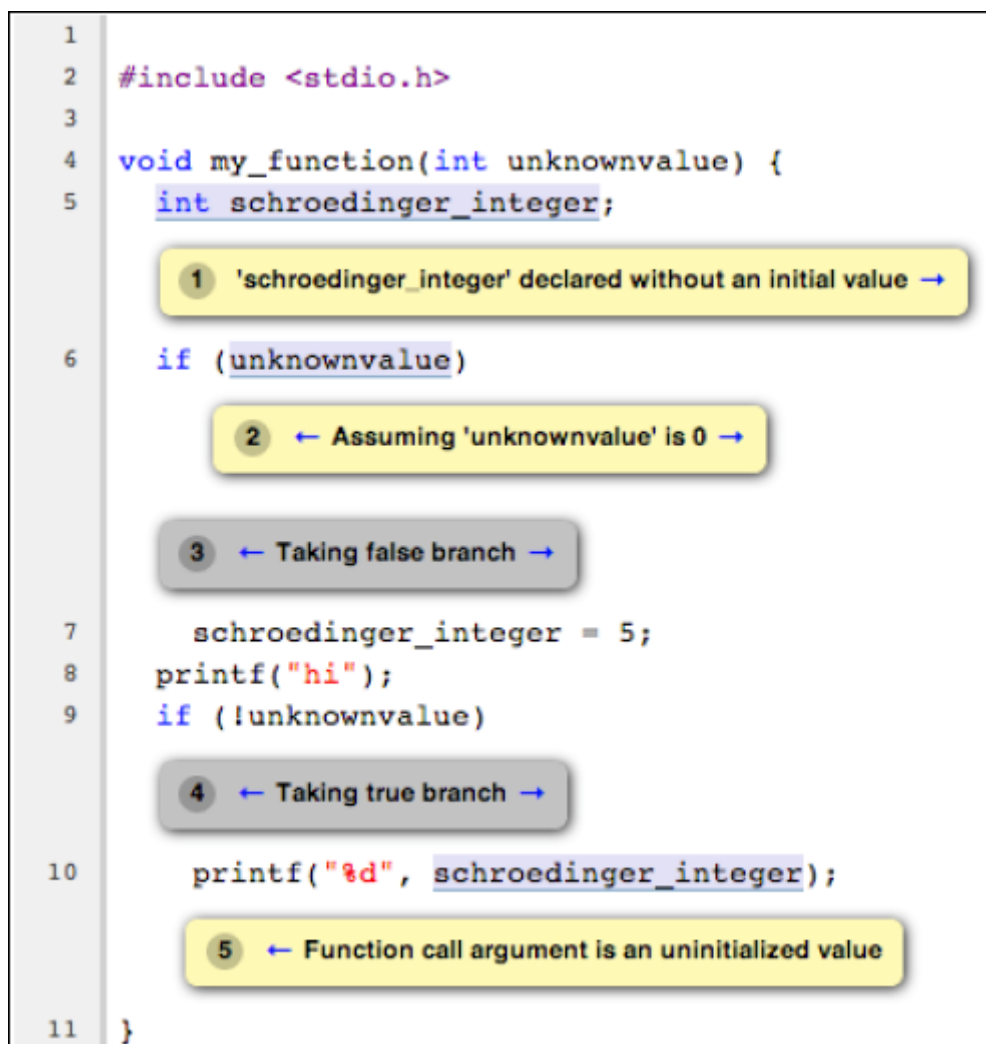
静态分析器还能够导出一个 HTML 文件，它图形化地指出代码中存在危险行为的程序路径，如 Xcode 所做的那样。我们还用参数 `-o` 指定一个文件夹名字，指示报告存储的地方。例如，看下面的命令行：

```
$ clang -cc1 -analyze -analyzer-checker=core joe.c -o report
```

可选地，你可以调用驱动器，如下：

```
$ clang --analyze -Xanalyzer -analyzer-checker=core joe.c -o report
```

根据这个命令行，分析器将处理 `joe.c`，并生成一个与 Xcode 中所看到的类似的报告，HTML 文件，放置在 `report` 文件夹中。命令完成之后，查看此文件夹并打开 HTML 文件，以阅读漏洞报告。你应该看到一个类似于如下截图的报告：



9.2.5 处理大型项目

如果你想用静态分析器检查一个大型项目，你大概不愿意写一个 Makefile 或者 bash 脚本，对项目的每个源文件调用分析器。静态分析器为此给出了一个便利的工具，称为 scan-build。

scan-build 替换 CC 或 CXX 环境变量，它们定义了 C/C++ 编译器命令，如此就介入了项目常规的 build 过程。它在编译之前分析每个文件，然后编译它，使得 build 过程或脚本能够如期望的那样继续工作。最终，它会生成 HTML 报告，你可以在浏览器中查看之。基本的命令行结构是很简单的：

```
$ scan-build <your build command>
```

你可以自由地在 scan-build 之后运行任意的 build 命令，例如 make。要想 build Joe 的程序，举例来说，我们不需要 Makefile，可以直接提供编译命令：

```
$ scan-build gcc -c joe.c -o joe.o
```

它完成之后，你可以运行 scan-view 以查看漏洞报告：

```
$ scan-view <output directory given by scan-build>
```

scan-build 所打印的最后一行，给出了运行 scan-view 所需要的参数。它会引用一个临时文件夹，那里存放着所有生成的报告。你应该看到一个格式优美的网页，列出了每个源文件的错误报告，如下面的截屏所示：

Bug Summary

Bug Type	Quantity	Display?
All Bugs	1	<input checked="" type="checkbox"/>
Logic error		
Uninitialized argument value	1	<input checked="" type="checkbox"/>

Reports

Bug Group	Bug Type ▾	File	Line	Path Length	
Logic error	Uninitialized argument value	joe.c	10	5	View Report Report Bug Open File

真实世界的例子——找到 Apache 的漏洞

在此例中，我们将检验在大型项目中检查漏洞是何等容易。为此，在 <http://httpd.apache.org/download.cgi> 下载最新的 Apache HTTP Server 源代码包。在写作的时候，它的版本是 2.4.9。在我们的例子中，我们将通过控制台下载它，并在当前文件夹解压文件：

```
$ wget http://archive.apache.org/dist/httpd/httpd-2.4.9.tar.bz2
$ tar -xjvf httpd-2.4.9.tar.bz2
```

我们将利用 `scan-build` 检查这个源代码库。为此，我们需要重复生成 `build` 脚本的步骤。注意，你需要所有必需的依赖库，以编译 `Apache` 项目。确认已经有了所有依赖库之后，执行下面的命令序列：

```
$ mkdir obj
$ cd obj
$ scan-build ../httpd-2.4.9/configure -prefix=$(pwd)/../install
```

我们用 `prefix` 参数指示这个项目新的安装路径，如此就不需要这台机器的管理员权限了。不过，如果你不打算实际安装 `Apache`，就不需要提供额外的参数，只要你不运行 `make install`。在我们的例子中，我们将安装路径定义为文件夹 `install`，它将在我们下载压缩源文件的相同目录中被创建。注意，我们还在命令前面加上 `scan-build`，它会覆写 `CC` 和 `CXX` 环境变量。

在 `configure` 脚本创建所有 `Makefile` 之后，就是启动实际的 `build` 过程的时候了。我们用 `scan-build` 拦截 `make` 命令，而不是单独执行它：

```
$ scan-build make
```

由于 `Apache` 代码非常多，完成分析花了几分钟，找到了 82 个漏洞。下面是 `scan-view` 报告的一个例子：

心脏击穿漏洞臭名昭著，在它击中了所有 `OpenSSL` 实现之后——这个问题引起了极大的关注——有趣的是，我们看到静态分析器仍然能够在 `Apache SSL` 的实现文件 `modules/ssl/ssl_util.c` 和 `modules/ssl/ssl_engine_config.c` 中找到六个疑似漏洞。请注意这些点可能存在于实践中从未被执行的路径内，可能不是真正的漏洞，因为静态分析器工作在一个有限的强度范围，为了在可接受的时间帧内完成分析。因此，我们没有断言它们是真正的漏洞。我们只是在此给出了一个例子来说明一个赋值是垃圾或者未定义的情况：

在这个例子中，静态分析器向我们表明，有一个执行路径最后给 `dc->nVerifyClient` 赋了一个未定义的值。这个路径的部分经历了对 `ssl_cmd_verify_parse()` 函数的调用，这显示出分析器在一个相同的编译模块内检查复杂的函数间路径的能力。在这个辅助函数中，静态分析器显示了在一个路径中 `mode` 没有被赋以任何值，因而它是未初始化的。

备注：之所以这可能不是一个真正的漏洞，是因为 `ssl_cmd_verify_parse()` 的代码可能处理了 `cmd_parms` 输入的所有情况，为它们正确地初始化了 `mode`，这些情况在实际的程序中发生了（注意上下文依赖）。`scan-build` 所发现的是，这个模块在孤立状态下可能会执行有漏洞的路径，但是我们没有证据得知这个模块的使用者会用到有漏洞的输入。静态分析器不够强大，无法在整个项目的上下文中分析这个模块，因为这样的分析需要花费不切实际的时间（记得算法的指数复杂度）。

这个路径有 11 步，而我们在 `Apache` 中发现的最长的路径有 42 步。这个路径出现在 `modules/generators/mod_cgid.c` 模块中，它违反了一个标准 C API 调用：它以一个 `null` 指针参数调用 `strlen()` 函数。

如果你好奇到想看所有这些报告的细节，不要犹豫亲自运行命令。

Bug Summary

Bug Type	Quantity	Display?
All Bugs	82	<input checked="" type="checkbox"/>
Dead store		
Dead assignment	12	<input checked="" type="checkbox"/>
Dead initialization	1	<input checked="" type="checkbox"/>
Logic error		
Assigned value is garbage or undefined	5	<input checked="" type="checkbox"/>
Branch condition evaluates to a garbage value	1	<input checked="" type="checkbox"/>
Called function pointer is null (null dereference)	1	<input checked="" type="checkbox"/>
Dereference of null pointer	20	<input checked="" type="checkbox"/>
Dereference of undefined pointer value	18	<input checked="" type="checkbox"/>
Division by zero	1	<input checked="" type="checkbox"/>
Result of operation is garbage or undefined	3	<input checked="" type="checkbox"/>
Uninitialized argument value	12	<input checked="" type="checkbox"/>
Unix API	7	<input checked="" type="checkbox"/>
Memory Error		
Memory leak	1	<input checked="" type="checkbox"/>


```

996  const char *ssl_cmd_SSLVerifyClient(cmd_parms *cmd,
997                                     void *dcfg,
998                                     const char *arg)
999  {
1000      SSLDirConfigRec *dc = (SSLDDirConfigRec *)dcfg;
1001      SSLSrvConfigRec *sc = mySrvConfig(cmd->server);
1002      ssl_verify_t mode;

1003      const char *err;
1004
1005      if ((err = ssl_cmd_verify_parse(cmd, arg, &mode))) {
1006          return err;
1007      }
1008
1009      if (cmd->path) {
1010          dc->nVerifyClient = mode;

```

1 'mode' declared without an initial value →

2 ← Calling 'ssl_cmd_verify_parse' →

7 ← Returning from 'ssl_cmd_verify_parse' →

8 ← Assuming 'err' is null →

9 ← Taking false branch →

10 ← Taking true branch →

11 ← Assigned value is garbage or undefined

9.3 用你自己的检查器扩展静态分析器

由于它的设计，我们可以轻易地以定制的检查器扩展静态分析器。记住静态分析器和它的检查器一样好，如果你想分析是否有代码以非预期的方式使用你的某个 API，你需要学习如何将这个域特定的知识嵌入到 Clang 静态分析器中。

9.3.1 熟悉项目的架构

Clang 静态分析器的源代码在 `llvm/tools/clang` 中。头文件在 `include/clang/StaticAnalyzer` 中，源代码在 `lib/StaticAnalyzer` 中。查看文件夹的内容，你会发现项目被划分为三个不同的子文件夹：Checkers，Core，和 Frontend。

Core 的任务是在源代码层次模拟程序的执行，利用一个 `visitor pattern`，并在每个程序点（在重要的语句之前或之后）调用注册的检查器，以强制一个给定的不变量。例如，如果你的检查器确认同一分配的内存区域不会被释放两次，它会观察 `malloc()` 和 `free()`，当它检测到重复释放时会生成一个漏洞报告。

符号引擎不能以精确的程序值模拟程序，如你在一个程序运行时看到的值。如果你让使用者输入一个整数值，你肯定会知道，在一次给定的运行中，举例来说，这个值是 5。符号引擎的威力在于对程序的每个可能的结果推断发生了什么，为了完成这个宏伟的目标，它考察符号（SVals）而不是具体的值。一个符号可能代表任意的整数、浮点数或者甚至一个完全未知的数。它对值知道得越多，它就越强大。

有三个重要的数据结构：`ProgramState`，`ProgramPoint`，和 `ExplodedGraph`；它们是理解项目实现的钥匙。第一个代表当前执行的关于当前状态的上下文。例如，当分析 Joe 的代码时，它会注明某个给定的变量的数值是 5。第二个代表程序流中的一个具体的点，在一个语句的前面或者后面，例如，在给一个整数变量赋值 5 的后面。最后一个代表整个可达程序状态的图。另外，这个图的节点是由 `ProgramState` 和 `ProgramPoint` 的元组表示的，这意味着，每个程序点都有一个具体的状态和它相关联。例如，给一个整数变量赋值 5 之后的点，由一个状态将这个变量和数字 5 联系起来。

正如本章的开头已经指出的那样，`ExplodedGraph`，或者说，可达状态图，表示对经典 CFG 的一个重要的展开。注意，一个具有两个串联的而不是嵌套的 if 的小的 CFG，在可达状态图的表示中，会爆炸成四个不同的路径——组合的扩展。为了节省空间，这个图会被折叠，这意味着，如果你创建一个节点，它表示的程序点以及状态和另一个节点的相同，就不会分配新的节点，而是重用这个已有的节点，可能建造回路。为了实现这个行为，`ExplodedNode` 继承了 LLVM 库的超类 `llvm::FoldingSetNode`。LLVM 库已经为这种情形引入了一个公共的类，因为在表示程序时，折叠在编译器的中间端和后端中被广泛使用。

静态分析器的总体设计可以被划分成以下部分：引擎，它跟随仿真路径并管理其它组件；状态管理器，管理 `ProgramState` 对象；约束管理器，负责推断由跟随给定程序路径引起的对 `ProgramState` 的约束；以及存储管理器，管理程序存储模型。

分析器的另一个重要的方面是，如何建模内存的行为，当它沿着每条路径模拟程序的执行时。对于如 C 和 C++ 这样的语言，这是相当具有挑战的，因为它们为程序员提供了多种访问相同内存片段的方式，从而产生别名。

分析器实现了一种由 Xu 等人的论文所描述的区域内存模型（查看本章末尾的引用），它甚至能够区分一个数组的每个元素的状态。Xu 等人提出了一种内存区域的层级结构，在其中，举例来说，数组元素是数组的子区

域，数组是堆栈的子区域。C 中的每个 lvalue，或者换句话说，每个变量或者引用，有一个对应的区域建模了它们所作用的内存片段。另一方面，每个内存区域的内容，是通过绑定建模的。每个绑定将一个符号值和一个内存区域关联起来。这里有太多的信息需要吸收，所以让我们以一种可能的最佳方式消化它——编写代码。

9.3.2 编写你自己的检查器

考虑你在开发一个特定的嵌入式软件，它控制着一个核反应堆，依靠具有两个基本调用的 API: `turnReactorOn()` 和 `SCRAM()`（关闭核反应堆）。核反应堆包含燃料和控制杆，前者是核反应发生的地方，后者包含中子吸收器，它能减缓核反应，使核反应堆保持发电厂的规模，而不是变成原子弹。

你的客户告知你，调用 `SCRAM()` 两次可能导致控制杆被卡住，调用 `turnReactorOn()` 两次会导致核反应失去控制。这个 API 具有严格的使用规则，你的任务是，在代码成为产品之前，审查一个大型的代码库，确保它没有违反这些规则：

- 不存在代码路径在不介入 `turnReactorOn()` 的情况下调用 `SCRAM()` 超过一次
- 不存在代码路径在不介入 `SCRAM()` 的情况下调用 `trunRactionOn()` 超过一次

作为一个例子，考虑下面的代码：

```
int SCRAM();
int turnRactionOn();

void test_loop(int wrongTemperature, int restart) {
    turnRactionOn();
    if (wrongTemperature) {
        SCRAM();
    }
    if (restart) {
        SCRAM();
    }
    turnReactorOn();
    // code to keep the reactor working
    SCRAM();
}
```

如果 `wrongTemperature` 和 `restart` 都不是 0，这份代码违反了 API，导致调用 `SCRAM()` 两次，其间没有介入 `trunReactorOn()`。如果这两个参数都是 0，它也违反了 API，因为这样的话，代码会调用 `turnReactorOn()` 两次，其间没有介入 `SCRAM()` 调用。

用定制的检查器解决问题

你要么可以尝试用肉眼检查代码，这是非常枯燥且容易出错的，要么使用一个像 Clang 静态分析器这样的工具。问题在于，它不理解核电站 API。我们将通过实现一个特殊的检查器来克服它。

第一步，我们要为我们的状态模型建立概念，关于我们想要在不同程序状态间传播的信息。在这个问题中，我们关切反应堆是开启的还是关闭的。我们可能不知道它是开启的还是关闭的；因此，我们的状态模型包含三个可能的状态：未知，开启，和关闭。

现在，关于我们的检查器如何处理状态，我们有一个优雅的主意。

编写状态类

让我们付诸实践。我们的代码将会以 SimpleStreamChecker.cpp 为基础，这是 Clang 代码树中可找到的一个简单的检查器。

在 lib/StaticAnalyzer/Checkers 中，我们应该创建一个新的文件，ReactorChecker.cpp，并开始编写我们自己的类，这个类表示我们在跟踪的时候所关心的状态：

```
#include "ClangSACheckers.h"
#include "clang/StaticAnalyzer/Core/BugReporter/BugType.h"
#include "clang/StaticAnalyzer/Core/Checker.h"
#include "clang/StaticAnalyzer/Core/PathSensitive/CallEvent.h"
#include "clang/StaticAnalyzer/Core/PathSensitive/CheckerContext.h"
using namespace clang;
using namespace ento;
class ReactorState {
private:
    enum Kind {On, Off} K;
public:
    ReactorState(unsigned InK) : K((Kind) InK) {}
    bool isOn() const { return K == On; }
    bool isOff() const { return K == Off; }
    static unsigned getOn() { return (unsigned) On; }
    static unsigned getOff() { return (unsigned) Off; }
    bool operator == (const ReactorState &X) const {
        return K == X.K;
    }
    void Profile(llvm::FoldingSetNodeID &ID) const {
        ID.AddInteger(K);
    }
};
```

我们的类的数据部分限制为 Kind 的单个实例。注意 ProgramState 类会管理我们编写的状态信息。

理解 ProgramState 的不变性

关于 ProgramState 的一个有趣的经验是，它生来就是不可变的。一旦建造出来，它就应该绝不改变：它代表在一个给定的执行路径中的一个给定的程序点的被计算出来的状态。不同于处理 CFG 的数据流分析，在这种情况下，我们处理可达程序状态图，对于不同的一对程序点和状态，它都有不同的节点。以这种方式，如果程序发生循环，引擎会创建一个完全新的路径，这个路径记录了关于这次新的迭代的关联信息。相反地，在数据流分析中，一个循环会导致循环体的状态被新的信息更新，直到到达一个固定的点。

然而，正如之前强调的那样，一旦符号引擎到达一个表示一个给定循环体的相同程序点的节点，这个点具有相同的状态，它会认为在这个路径中没有新的信息需要处理，就重用这个节点而不是新建一个。另一方面，如果你的循环有一个循环体在不断地以新的信息更新状态，你就很快会达到符号引擎的限度：它会在模拟预定数目的迭代后放弃这个路径，这是一个可配置的数目，你可以在启动这个工具时设置它。

剖析代码

由于状态一旦创建就不可变，我们的 ReactorState 类不需要 setter，或者用于修改其状态的类成员函数，但是我们确实需要构造器。这就是 ReactorState(unsigned InK) 构造器的目的，它接受一个编码当前反应器状态的整数作为输入。

最后，Profile 函数是 ExplodeNode 的结果，它是 FoldingSetNode 的子类。所有子类必须提供这样的方法，以协助 LLVM 折叠追踪节点的状态并判断两个节点是否相同（这时它们会被折叠）。因此，我们的 Profile 函数会说明 K，一个数字，给出我们的状态。

你可以用任何以 Add 开头的 FoldingSetNodeID 成员函数来告知独特的位，这些位用于识别这个对象的实例（查看 llvm/ADT/FoldingSet.h）。在我们的例子中，我用了 AddInteger()。

定义检查器子类

现在，是时候声明我们的 Checker 子类了：

```
class ReactorChecker : public Checker<check::PostCall> {
    mutable IdentifierInfo *IITurnReactorOn, *IISCRAM;
    OwingPtr<BugType> DoubleSCRAMBugType;
    OwingPtr<BugType> DoubleONBugType;
    void initIdentifierInfo(ASTContext &Ctx) const;
    void reportDoubleSCRAM(const CallEvent &Call, CheckerContext &C) const;
    void reportDoubleON(const CallEvent &Call, CheckerContext &C) const;
public:
    ReactorChecker();
    /// Process turnReactorOn and SCRAM
    void checkPostCall(const CallEvent &Call, CheckerContext &C) const;
};
```

备注：注意 Clang 版本——从 Clang 3.5 开始，`OwningPtr<>` 模板被淘汰，而采用标准的 C++ `std::unique_ptr<>` 模板。这两个模板都提供了智能指针的实现。

我们的类的第一行表明，它是一个指定了模板参数的 `Checker` 的子类。对于这个类，可以使用多个模板参数，它们表示你的检查器在巡查时所感兴趣的程序点。技术上来说，这些模板参数用于派生一个定制的 `Checker` 类，这个类是所有被指定为参数的类的子类。这意味着，对于我们的案例，我们的检查器会从基类继承 `PostCall`。如此继承是用于实现巡查模式，它只会为我们感兴趣的对象调用我们，因此，我们的类必须实现成员函数 `checkPostCall`。

你也许对登记你的检查器感兴趣，以巡查广泛多样的程序点类型（查看 `CheckerDocumentation.cpp`）。在我们的案例中，我们关注在调用到达之后立即访问程序点，因为我们想在某个核电厂 API 函数被调用之后，记录状态的变化。

这些成员函数使用了 `const` 关键字，这遵从了其设计，它依赖无状态的检查器。然而，我们确实想贮存获取 `IdentifierInfo` 对象的结果，它们代表符号 `turnReactorOn()` 和 `SCRAM()`。这样，我们使用 `mutable` 关键字，它被创造出来以绕过 `const` 的限制。

备注：谨慎使用 `mutable` 关键字。我们不是在损害检查器的设计，因为我们只是贮存结果以加速第二次调用到达我们的检查器之后的计算，但是概念上我们的检查器仍然是无状态的。`mutable` 关键字应该只用于互斥或者像这样的贮存的场景。

我们还想告知 Clang 基础设施，我们在处理一种新的漏洞类型。为此，我们必须保存新的 `BugType` 实例，新的漏洞各保存一个，我们打算报告这些漏洞：程序员调用 `SCRAM()` 两次时发生的漏洞，以及程序员调用 `turnReactorOn()` 两次时发生的漏洞。我们还用 `OwningPtr LLVM` 类封装我们的对象，它是一种自动指针的实现，用于自动地释放我们的对象，一旦我们的 `ReactorChecker` 对象被销毁。

你应该封装我们刚编写的两个类，`ReactorState` 和 `ReactorChecker`，封装在一个匿名名字空间中。这会阻止我们的链接器导出这两个数据结构，我们知道它们只在本地使用。

编写寄存器宏

在深入学习类的实现之前，我们必须调用一个宏来展开 `ProgramState` 实例，分析器引擎用它处理我们定制的状态：

```
REGISTER_MAP_WITH_PROGRAMSTATE(RS, int, ReactorState)
```

注意，这个宏的末尾没有分号。这为每个 `ProgramState` 实例关联一个新的 `map`。第一个参数可以是任意名字，此后你将用它引用这个数据，第二个参数是 `map` 键值的类型，第三个参数是我们要存储的对象的类型（此处它是 `ReactorState` 类）。

检查器常常用 `map` 存储它们的状态，因为给特定的资源关联新的状态是常见的，例如，在本章开头的检测器中，每个变量的状态，初始化的或未初始化的。在这种情况下，`map` 的键值会是变量的名字，存储的值会

是一个定制类，这个类建模了状态的未初始化或初始化。对于另外的向程序状态登记信息的方式，查看 `CheckerContext.h` 中的宏定义。

注意，我们并不真正地需要一个 `map`，因为我们会总是为每个程序点只存储一个状态。因此，我们会总是用键值 1 访问我们的 `map`。

实现检查器子类

我们的检查器类的构造器实现如下：

```
ReactorChecker::ReactorChecker() : IITurnReactorOn(0), IISCRAM(0) {
    // Initialize the bug types.
    DoubleSCRAMBugType.reset(new BugType( "Double SCRAM" , "Nuclear Reactor API Error" ));
    DoubleONBugType.reset(new BugType( "Double ON" , "Nuclear Reactor API Error" ));
}
```

备注：注意 Clang 版本——从 Clang 3.5 开始，我们的 `BugType` 构造器调用需要变为 `BugType(this, ("Double SCRAM" , "Nuclear Reactor API Error")` 和 `BugType(this, "Double ON" , "Nuclear Reactor API Error")`，就是添加 `this` 关键字作为第一个参数。

我们的构造器实例化了一个新的 `BugType` 对象，利用 `OwningPtr` 的 `reset()` 成员函数，我们给出了关于新的漏洞种类的描述。我们还初始化了 `IdentifierInfo` 指针。接着，是时候定义我们的辅助函数以贮存这些指针的结果：

```
void ReactorChecker::initIdentifierInfo(ASTContext &Ctx) const {
    if (IITurnReactorOn)
        return;
    IITurnReactorOn = &Ctx.Idents.get("turnReactorOn");
    IISCRAM = &Ctx.Idents.get("SCRAM");
}
```

`ASTContext` 对象保存了特定的 `AST` 节点，这些节点包含用户程序用到的类型和声明，我们可以用它找到我们在监听时所感兴趣的函数的准确的标识符。现在，我们实现巡查器模式函数，`checkPostCall`。记住，它是一个 `const` 函数，应该不修改检查器的状态：

```
void ReactorChecker::checkPostCall(const CallEvent &Call,
                                   CheckerContext &C) const {
    initIdentifierInfo(C.getASTContext());
    if (!Call.isGlobalCFunction())
        return;
    if (Call.getCalleeIdentifier() == IITurnReactorOn) {
```

(续下页)

(接上页)

```

ProgramStateRef State = C.getState();
const ReactorState *S = State->get<RS>(1);
if (S && S->isOn()) {
    reportDoubleON(Call, C);
    return;
}
State = State->set<RS>(1, ReactorState::getOn());
C.addTransition(State);
return;
}
if (Call.getCalleeIdentifier() == IISCRAM) {
    ProgramStateRef State = C.getState();
    const ReactorState *S = State->get<RS>(1);
    if (S && S->isOff()) {
        reportDoubleSCRAM(Call, C);
        return;
    }
    State = State->set<RS>(1, ReactorState::getOff());
    C.addTransition(State);
    return;
}
}
}

```

第一个参数是 `CallEvent` 类型，它持有一个函数的信息，程序就在这个程序点之前调用了这个函数（查看 `CallEvent.h`），因为我们登记了一个调用后巡查器。第二个参数是 `CheckerContext` 类型，它是在这个程序点的当前状态的唯一信息来源，因为我们的检查器必须是无状态的。我们用它获取 `ASTContext`，初始化 `Identifier` 对象，检查我们监听的函数有赖于它们。我们询问 `CallEvent` 对象，以检查它是否调用了 `trunReactorOn()` 函数。如果是，我们需要进行状态转移，转移到开启状态。

在转移状态之前，我们首先检查状态是否已经是开启的，在这种情况下，就存在漏洞。注意在 `State->get<RS>(1)` 语句中，`RS` 只是我们在登记程序状态的新特征时所给的名字，`1` 是固定的整数，总是用它访问 `map` 的位置。虽然在这种情况下我们实际上不需要 `map`，但是通过使用 `map`，你将能够轻松地扩展我们的检查器以监听更加复杂的多个状态，如果你想的话。

我们将我们存储的状态恢复为一个 `const` 指针，因为我们在处理的到达这个程序点的信息是不可变的。首先，有必要检查它是否为空的引用，这表示我们不知道反应堆是开启的还是关闭的。如果它不是空的，我们检查它是否为开启的并且处于阳性的状况，这样就放弃进一步的分析而报告一个漏洞。对于其它情况，我们通过 `ProgramStateRef set` 成员函数新建一个状态，并将这个新的状态传送给 `addTransition()` 成员函数，它会记录信息以在 `ExplodedGraph` 中创建一条新的边。只有在状态实际改变时，才会创建这样的边。在处理 `SCRAM` 的时候，我们用了类似的逻辑。

漏洞报告成员函数的代码如下所示：

```

void ReactorChecker::reportDoubleON(const CallEvent &Call,
                                   CheckerContext &C) const {
    ExplodedNode *ErrNode = C.generateSink();
    if (!ErrNode)
        return;
    BugReport *R = new BugReport(*DoubleONBugType,
                                "Turned on the reactor two times", ErrNode);
    R->addRange(Call.getSourceRange());
    C.emitReport(R);
}

void ReactorChecker::reportDoubleSCRAM(const CallEvent &Call,
                                       CheckerContext &C) const {
    ExplodedNode *ErrNode = C.generateSink();
    if (!ErrNode)
        return;
    BugReport *R = new BugReport(*DoubleSCRAMBugType,
                                "Called a SCRAM procedure twice", ErrNode);
    R->addRange(Call.getSourceRange());
    C.emitReport(R);
}

```

我们的第一个动作是生成一个 sink 节点，在可达程序状态中，它意味着我们在这个路径上遇到一个严重的漏洞，我们不想继续分析这个路径。下面几行创建一个 **BugReport** 对象，报告我们找到了一个新的漏洞，漏洞的类型是 **DoubleOnBugType**，漏洞描述可以任意写，提供我们刚刚建造的出错节点。我们还用到了 **addRange()** 成员函数，它会高亮出现漏洞的代码，显示给用户。

添加登记代码

为了让静态分析器工具认出我们的新检查器，我们需要在我们的源代码中定义一个登记函数，然后在一个 **TableGen** 文件中添加我们的检查器的描述。登记函数如下所示：

```

void ento::registerReactorChecker(CheckerManager &mgr) {
    mgr.registerChecker<ReactorChecker>();
}

```

TableGen 文件有一个检查器的表。它位于 **lib/StaticAnalyzer/Checkers/Checkers.td**，相对于 Clang 源代码文件夹。在编辑这个文件之前，我们需要选择一个包以放置我们的检查器。我们会把它放在 **alpha.powerplant** 中。这个包还不存在，因此我们要创建它。打开 **Checkers.td**，在所有已存在的包定义之后添加一个新的定义：

```
def PowerPlantAlpha : Package<" powerplant" >, InPackage<Alpha>;
```

下面，添加我们新写的检查器：


```
let ParentPackage = PowerPlantAlpha in {

def ReactorChecker : Checker<" ReactorChecker" >,
  HelperText<" Check for misuses of the nuclear power plant API" >,
  DescFile<" ReactorChecker.cpp" >;

} // end "alpha.powerplant"
```

如果你用 CMake build Clang，你应该将你的新源文件添加到 lib/StaticAnalyzer/Checkers/CMakeLists.txt。如果你用 GNU 自动工具配置脚本以 build Clang，你就不需要修改任何其它文件，因为 LLVM Makefile 会扫描 Checkers 文件夹中的新源代码文件，并在静态分析器的检查器库中链接它们。

编译和测试

进入你 build LLVM 和 Clang 的文件夹，运行 make。现在 build 系统会检测到你的新代码，build 它，并向 Clang 静态分析器链接它。当你完成 build 之后，命令行 clang -cc1 -analyzer-checker-help 就应该列出我们的新检查器为一个合法的选项。

下面给出了一个我们的检查器的测试案例，managereactor.c（和前面给出的相同）：

```
int SCRAM();
int turnReactorOn();

void test_loop(int wrongTemperature, int restart) {
    turnReactorOn();
    if (wrongTemperature) {
        SCRAM();
    }
    if (restart) {
        SCRAM();
    }
    turnReactorOn();
    // code to keep the reactor working
    SCRAM();
}
```

要用我们的新检查器分析以上代码，我们使用下面的命令：

```
$ clang -analyze -Xanalyzer -analyzer-check=alpha.powerplant managereactor.c
```

检查器会显示它能发现为错误的路径并退出。如果你请求一个 HTML 报告，你就会看到一个漏洞报告，类似下面的截屏所示：

现在你的任务完成了：你成功地开发了一个程序来自动检查对一个特定的路径敏感的 API 的违规。如果你愿意，你可以查看其它检查器的实现，学习更多处理更复杂场景的知识，或者查看下一节列出的资源以获得更

```
1  int SCRAM();
2  int turnReactorOn();
3
4  void test_loop(int wrongTemperature, int restart) {
5      turnReactorOn();
6      if (wrongTemperature) {
7          SCRAM();
8      }
9      if (restart) {
10         SCRAM();
11     }
12     turnReactorOn();
13     // code to keep the reactor working
14     SCRAM();
15 }
16
```

1 Assuming 'wrongTemperature' is 0 →

2 ← Taking false branch →

3 ← Assuming 'restart' is 0 →

4 ← Taking false branch →

5 ← Turned on the reactor two times

多信息。

9.4 更多资源

你可以查看下面的资源以了解更多的项目和其它的信息：

- <http://clang-analyzer.llvm.org>：Clang 静态分析器项目的网页。
- http://clang-analyzer.llvm.org/checker_dev_manual.html：为想要开发新的检查器的人准备的有用的手册。
- <http://lcs.ios.ac.cn/~xzx/memmodel.pdf>：论文 A Memory Model for Static Analysis of C，作者 Zhongxing Xu, Ted Kremenek, Jian Zhang。它从理论层面详细解释了分析器核心所实现的内存模型。
- <http://clang.llvm.org/doxygen/annotated.html>：Clang doxygen 文档。
- <http://llvm.org/devmtg/2012-11/videos/Zaks-Rose-Checker24Hours.mp4>：由 Anna Zaks 和 Jordan Rose 在 2012 LLVM 开发者会议上作的一个讲座，解释如何快速建造检查器，他们是分析器开发者。

9.5 总结

在本章中，我们探讨了 Clang 静态分析器如何不同于运行在编译器前端的简单漏洞检测工具。我们以例子说明了静态分析器是更精确的，解释了在精确性和计算时间之间的权衡，需要指数级时间的静态分析算法是不适合集成到常规的编译器管线的，因为它完成分析所需的时间是不可接受的。我们还介绍了如何用命令行接口对简单项目运行静态分析器，以及用辅助工具 scan-build 来分析大型的项目。最后我们介绍了如何用我们自己的路径敏感的漏洞检查器扩展静态分析器。

在下一章，我们将介绍建造在 LibTooling 基础之上的 Clang 工具，它简化了建造代码重构工具的过程。

第 10 章 Clang 工具和 LibTooling

在这一章，我们将看到有多少工具以库的形式利用 Clang 前端，为了不同的目的而操作 C/C++ 程序。特别地，它们都依赖 LibTooling，一个 Clang 库，它使人们可以编写独立的工具。在这种情况下，你可以设计一个完全属于你自己的工具，利用 Clang 的解析能力，让你的用户可以直接调用你的工具，而不是编写一个插件以适应 Clang 编译管线。这一章所展示的工具可以在 Clang 额外工具包中找到；参考第 2 章（外部项目）了解如何安装它们。我们将以一个可用的例子结束这一章，演示如何创建你自己的代码重构工具。我们将讨论以下话题：

- 生成编译命令 database
- 理解并使用若干 Clang 工具，它们依赖 LibTooling，例如 Clang Tidy、Clang Modernizer、Clang Apply Replacements、ClangFormat、Modularize、PPTrace、和 Clang Query
- 建造你自己的基于 LibTooling 的代码重构工具

10.1 生成编译命令 database

一般来说，编译器被 build 脚本调用，例如 Makefile，用一系列参数配置它，使之恰当地使用项目头文件和定义。这些参数让前端能够正确地分词和解析输入的源代码文件。然而，在这一章，我们将学习独立工具，它们将独立运行，而不是作为 Clang 编译管线的一部分。因此，理论上，我们会需要一个具体的脚本，以正确的参数对每个源代码文件运行我们的工具。举例来说，下面的命令显示了 Make 所用的完整的命令行，它调用编译器以 build 来自 LLVM 库的一个典型的文件：

```
$ /usr/bin/c++ -DNDEBUG -D__STDC_CONSTANT_MACROS -D__STDC_FORMAT_MACROS  
-D__STDC_LIMIT_MACROS -fPIC -fvisibility-inlines-hidden -Wall -W -Wnunused-
```

(续下页)

(接上页)

```
parameter -Wwrite-strings -Wmissing-field-initializers -pedantic
-Wno-long-long -Wcovered-switch-default -Wnon-virtual-dtor -fno-rtti
-I/Users/user/p/llvm/llvm-3.4/cmake-scripts/utils/TableGen -I/Users/
user/p/llvm/llvm-3.4/llvm/utils/TableGen -I/Users/user/p/llvm/llvm-3.4/
cmake-scripts/include -I/Users/user/p/llvm/llvm-3.4/llvm/include -fnoexceptions
-o CMakeFiles/llvm-tblgen.dir/DAGISelMatcher.cpp.o -c /Users/
user/p/llvm/llvm-3.4/llvm/utils/TableGen/DAGISelMatcher.cpp
```

当你在使用这个库，你会相当不开心，如果你不得不输入如此长的命令，它占据终端 10 行，以分析每个源代码文件，不能丢弃一个字符，因为前端将使用此信息的全部。

为了让工具易于处理源代码文件，任意使用 LibTooling 的项目都接受命令 `database` 作为输入。这个命令 `database` 为一个具体项目的每个源文件设置正确的编译器参数。为了让事情变得更容易，如果以 `DCMAKE_EXPORT_COMPILE_COMMANDS` 参数调用 CMake，它就会为你生成这个 `database` 文件。举例来说，假设你期望对来自 Apache 项目的一个具体源代码文件运行基于 LibTooling 的工具。为了让你无需输入准确的编译器参数以正确地解析这个文件，你可以用 CMake 生成一个命令 `database`，如下所示：

```
$ cd httpd-2.4.9
$ mkdir obj
$ cd obj
$ cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=ON ../
$ ln -s $(pwd)/compile_commands.json ../
```

这和你用 CMake build Apache 所用的 `build` 命令类似，但是并不实际 build 它，`-DCMAKE_EXPORT_COMPILE_COMMANDS=ON` 参数指示 CMake 用编译器参数生成一个 JSON 文件，它将会用这些参数去编译每个 Apache 源文件。我们需要创建一个链接到这个 JSON 文件，让它出现在 Apache 源代码的根目录中。然后，当我们运行任何 LibTooling 程序去解析一个 Apache 源文件的时候，它将搜索父目录直到在其中找到 `compile_commands.json`，以得到恰当的参数去解析这个文件。

可选地，如果你不想在运行你的工具之前 build 编译命令 `database`，你可以用双短线 (-) 直接传递编译器命令，你将会用它处理这个文件。当你的项目不需要很多参数来编译时，这是有用的。举例来说，看下面的命令行：

```
$ my_libtooling_tool test.c -- -Iyour_include_dir -Dyour_define
```

10.2 clang-tidy 工具

在这小节，我们将介绍 clang-tidy，作为 LibTooling 工具的一个例子，解释如何使用它。所有其它 Clang 工具具有类似的样子和感觉，从而让你能够愉快地探索它们。

clang-tidy 是一个 linter，基于 Clang。一般来说，linter 是一种分析代码的工具，它暴露不符合最优形式的代码。它可以检查具体的特征，例如：

- 代码是否适应不同的编译器

- 代码是否遵循特定的习语或编码惯例
- 代码是否可能由于滥用语言特性而导致漏洞

就 clang-tidy 的具体情况而言，这个工具能够运行两种类型的检测器：来自原始的 Clang 静态分析器的检查器和专门为 clang-tidy 编写的检查器。尽管能够运行静态分析器检查，注意 clang-tidy 和其它基于 LibTooling 的工具是基于源代码分析的，这和前面章节描述的复杂的静态分析引擎是相当不同的。这些检查只是遍历 Clang AST，而不是模拟程序运行，它们也快得多。不同于 Clang 静态分析器的检查，为 clang-tidy 编写的检查一般以检查是否符合特定的编码惯例为目标。特别地，它们检查 LLVM 编码惯例和 Google 编码惯例，还有其它一般的检查。

如果你遵循特定的编码惯例，你会发现 clang-tidy 非常有用，用它定期地检查你的代码。花点工夫，你甚至可以配置它，让它从一些文本编辑器里直接运行。但是，目前这个工具还未成熟，只实现了少量测试。

10.2.1 利用 clang-tidy 检查你的代码

在此例中，我们将演示如何用 clang-tidy 检查我们在第 9 章（Clang 静态分析器）写的代码。我们为静态分析器写了一个插件，如果我们想把这个检查器提交到官方的 Clang 源代码树，我们需要严格地遵循 LLVM 编码惯例。是时候检查我们是否真的遵循它了。一般的 clang-tidy 命令行接口如下：

```
$ clang-tidy [options] <source0> [... <sourceN>] [-- <compiler command>]
```

你可以小心地通过 -checks 参数中的名字激活每个检查器，但是你也可以利用通配符 * 选择许多具有相同开始子字符串的检查器。当你需要关闭一个检查器，就用带短划线前缀的检查器名字。举例来说，如果你想运行所有属于 LLVM 编码惯例的检查器，就应该用下面的命令：

```
$ clang-tidy -checks="llvm-*" file.cpp
```

备注：只有安装了 Clang 连同 Clang 额外工具代码仓库，所有本章中描述的工具才能运行，后者跟 Clang 树是分开的。如果你还没有安装 clang-tidy，请阅读第 2 章（外部项目），了解如何编译并安装 Clang 外部工具。

因为我们的代码是和 Clang 一起编译的，我们需要一个编译器 database。我们将开始生成它。进入你的 LLVM 源代码所在的文件夹，用下面的命令创建一个兄弟文件夹以存放 CMake 文件：

```
$ mkdir cmake-scripts
$ cd cmake-scripts
$ cmake -DCMAKE_EXPORT_COMPILE_COMMANDS=ON ../llvm
```

备注：如果你遇到一个 unknown-source-file 的错误，指向前一章所创建的检查器的代码，你需要以你的检查器源文件的名字更新 CMakeLists.txt 文件。用下面的命令行编辑这个文件，然后再次运行 CMake：

```
$ vim ../llvm/tools/clang/lib/StaticAnalyzer/Checkers/CMakeLists.txt
```

然后，在 LLVM 根文件夹中创建一个链接，指向编译器命令 database 文件。

```
$ ln -s $(pwd)/compile_commands.json ../llvm
```

现在，我们终于可以运行 clang-tidy 了：

```
$ cd ../llvm/tools/clang/lib/StaticAnalyzer/Checkers
$ clang-tidy -checks="llvm-*" ReactorChecker.cpp
```

你应该看到许多关于我们的检查器所包含的头文件的抱怨，它们没有严格地遵循 LLVM 规则，它要求每个 namespace 结尾的大括号有注释（见 <http://llvm.org/docs/CodingStandards.html#namespace-indentation>）。好消息是，我们的工具的代码，包括头文件，没有违反这些规则。

10.3 重构工具

在这一小节，我们将介绍许多其它的工具，它们利用 Clang 的解析能力，执行代码分析和源到源的转换。以一种类似 clang-tidy 的方式使用它们，依靠你的命令 database 来简化用法，这会让你感到舒服。

10.3.1 Clang Modernizer

Clang Modernizer 是一个革命性的独立工具，它帮助人们改写陈旧的 C++ 代码以使用最新的标准，例如，C++11。它通过执行下面的变换以达到这个目标：

- 循环转变变换：将陈旧的 C-风格的 for(;;) 循环转变为更新的基于范围的 for(auto &...;) 形式的循环
- 使用 nullptr 变换：将陈旧的 C-风格的表示空指针的 NULL 或常数 0 转变为更新的 nullptr C++11 关键字
- 使用 auto 变换：将一些类型声明在特定的情况下转变为使用 auto 关键字，这提高了代码可读性
- 添加 override 变换：为重写基类函数的虚拟成员函数声明添加 override 修饰
- 值转递变换：用值传递成语替换被复制的 const 引用
- 替换 auto_ptr 变换：用 std::unique_ptr 替换已过时的 std::auto_ptr

源到源的变换工具利用了 Clang LibTooling 基础设施，Clang Modernizer 是其中的一个引人入胜的例子。要想使用它，观察下面的模板：

```
$ clang-modernize [<options>] <source0> [... <sourceN>] [-- <compiler command>]
```

注意，如果你不提供任何额外的选项，除了源代码文件名，这个工具就会直接对源文件付诸全部变换。用参数 -serialize-replacements 强制将提议的补丁写到磁盘，这让你能够先阅读它们，再应用它们。有特别的工具可以应用在磁盘上的补丁，我们将在后面介绍它们。

10.3.2 Clang Apply Replacements

Clang Modernizer（之前的 C++ 迁移器）的开发引发了讨论，关于如何协调对大型代码库的源到源的变换。例如，当分析不同的翻译单元时，同一个头文件可能被分析多次。

处理这个问题的一个可选方法是，序列化替换提议，将它们写到文件。第二个工具将负责读入这些提议的文件，丢弃冲突的和重复的提议，并对源文件应用这些替换提议。这是 Clang Apply Replacements 的目的，它生来就是用于帮助 Clang Modernizer 修正大型的代码库的。

Clang Modernizer 和 Clang Apply Replacements，前者产生替换提议，后者实施这些提议，它们都会利用 clang::tooling::Replacement 类的一个序列化版本。此序列化用到了 YAML 格式，它可以被定义为 JSON 的超集，易于人们阅读。

代码版本工具所用的补丁文件，正好是一种修改提议的序列化格式，但是 Clang 开发者选择使用 YAML，直接利用 Replacement 类的序列化，避免解析补丁文件。

因此，Clang Apply Replacements 工具不打算成为一个通用的代码补丁工具，而是一个专用的工具，致力于处理依赖于工具化 API 的 Clang 工具所作出的修改。注意，如果你在编写一个源到源的变换工具，只有当你希望协调多个修改提议以消除重复修改时，才需要使用 Clang Apply Replacements 工具。否则，你就直接简单地修改源文件。

为了看清 Clang Apply Replacements 如何工作，我们首先需要使用 Clang Modernizer，强制它序列化它的修改提议。假设我们想要转换下面的 C++ 源文件，让它使用新的 C++ 标准：

```
int main() {
    const int size = 5;
    int arr[] = {1,2,3,4,5};
    for (int i = 0; i < size; ++i) {
        arr[i] += 5;
    }
    return 0;
}
```

根据 Clang Modernizer 的用户手册，转换这个循环让它使用新的 auto 迭代器是安全的。为此，我们需要使用 Clang Modernizer 的循环转换：

```
$ clang-modernize -loop-convert -serialize-replacements test.cpp
--serialize-dir=.
```

最后一个参数是可选的，它指定当前文件夹将用于存放替换文件。如果我们不指定它，这个工具会创建一个临时文件夹，之后让 Clang Apply Replacements 使用。由于我们将所有替换文件输出到当前文件夹，你可以直接分析生成的 YAML 文件。付诸实践，简单地运行 clang-apply-replacements，以当前文件夹作为它唯一的参数：

```
$ clang-apply-replacements .
```

备注： 运行这个命令之后，如果你得到这样的错误信息：“trouble iterating over directory ./: too many levels of symbolic links”，你可以通过使用/tmp 作为存储替换文件的文件夹，重试最后两个命令。或者，你可以创建一个新的文件夹以存放这些文件，让你易于分析它们。

不止于这个简单的例子，这些工具通常被设计成用于处理大型代码库。因此，Clang Apply Replacements 不会问任何问题，只是直接开始解析所指定文件夹中存在的所有 YAML 文件，分析并实行转换。

你甚至可以指定具体的编码标准，要求这个工具在打补丁（向源文件写入新的代码）的时候必须遵从之。这就是参数-style=<LLVM|Google|Chromium|Mozilla|Webkit> 的目的。这项功能是 LibFormat 库提供的便利，它让任意重构工具能够以某种具体的格式或编码惯例编写新代码。我们将在下一小节给出关于这个著名的特性的更多细节。

10.3.3 ClangFormat

想象你是一项竞赛的评审员，类似于国际模糊 C 代码竞赛（IOCCC: International Obfuscated C Code Contest）。为了给你一种竞赛的感觉，我们将再次产生 22 期胜者之一 Michael Birken 的代码。记住，这份代码在 Creative Commons

Attribution-ShareAlike 3.0 许可证下获得许可，这意味着你可以任意地修改它，只要你保留此许可证，并把荣誉归于 IOCCC。

免得你想问，这是正确的代码吗？告诉你，是的。访问 <http://www.ioccc.org/2013/birken> 可下载它。现在，让我们演示 ClangFormat 会怎么处理此代码。

```
$ clang-format -style=llvm obf.c --
```

下面的截屏显示了结果：

变好了，对吗？在实际中，你将幸运地不需要检查模糊不清的代码，但是调整格式以遵循特别的编码惯例不是人类特别梦想的工作。这就是 ClangFormat 的目的。它不只是一个工具，还是一个库，LibFormat，它格式化代码以适应某种编码惯例。这样，如果你新建的工具恰好会生成 C 或 C++ 代码，你可以专注于你的项目，而把格式的事情留给 ClangFormat。

这个例子明显是人造的，除了展开它并执行代码缩进，ClangFormat 这个巧妙的工具被细心地开发出来，用以寻找将代码调整为每行 80 列的格式的最好的方法，提高代码的可读性。如果你曾经停留于考虑如何最好地分解一个长句，你会感激 ClangFormat 是多么善于处理这样的任务。尝试在你最喜欢的编辑器中将它设置为一个外部工具，配置一个启动它的热键。如果你在使用著名的编辑器，例如 Vim 或者 Emacs，请确认有人已经写了定制脚本来集成 ClangFormat。

代码格式化、组织和澄清等话题，还引出了 C 和 C++ 代码令人讨厌的问题：滥用头文件，以及怎么协调它们。下一节将专注于讨论针对此问题的在进行中的方案，以及 Clang 工具怎么帮助你采用此新方法。

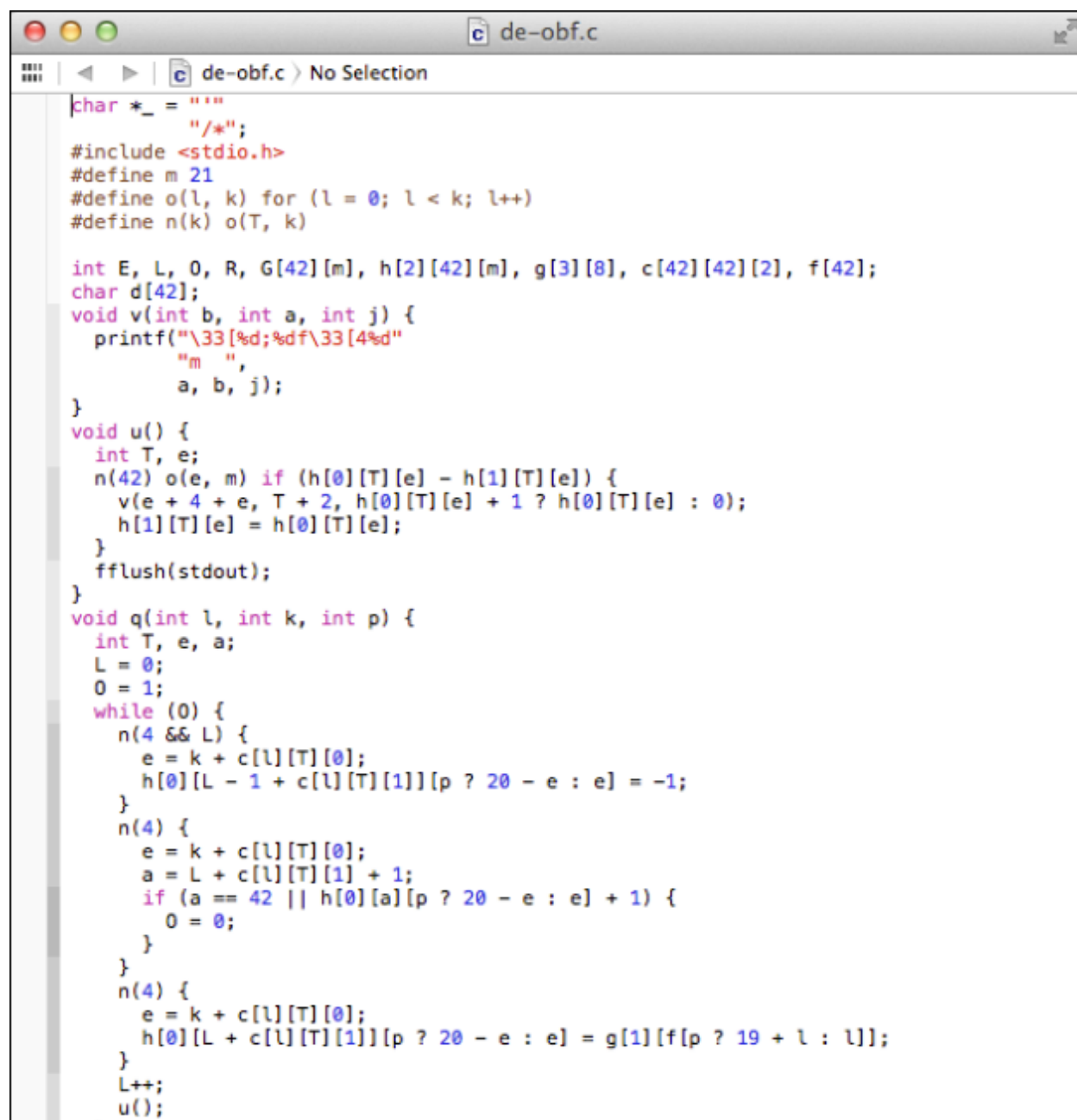


```

char* _ = ""/*";
#include <stdio.h>
#define m 21
#define o(l, k) for(l=0; l<k; l++)
#define n(k) o(T, k)

int E,L,O,R,G[42][m],h[2][42][m],g[3][8],c
[42][42][2],f[42]; char d[42]; void v( int
b,int a,int j){ printf("\33[%d;%df\33[4%d"
"m ",a,b,j); } void u(){ int T,e; n(42)o(
e,m)if(h[0][T][e]-h[1][T][e]){ v(e+4+e,T+2
,h[0][T][e]+17h[0][T][e]:0); h[1][T][e]=h[
0][T][e]; } fflush(stdout); } void q(int l
,int k,int p){
int T,e,a; L=0
; 0=1; while(0
){ n(4&&L){ e=
k+c[l][T][0];
h[0][L-1+c[l][
T][1]][p?20-e:
e]=-1; } n(4){
1}+1; if(a==42
; } } n(4){ e=
T[1]][p?20-e:
u(); } n(42){
o(a, m&&e==m){
]=h[0][L-1][a
}int main(){ int T,e,t,r,i,s
1][T]-7-T; R--; n(42) o(e,m)
R; n(17){ e=d[T]-48; d[T]=0;
} } n(8)if(g[0][7-T]){ t=g[i
g[2][g[i][T]]=T; n(R+i)o(e,m
)o(t,2){ f[T+t+T]-t["%#",1"
[e][t]="5'<$=$8)Ih$=h9i8'9"
} } n(15) { s=T>9?n:(T&3)-3?15:36;o(e,s)o(t,2)c[T+19][e][t]="6*6,8*6.608.6264826668\
865::(+;0(6+6-6/8,61638065678469.;88))3(6,8*6.608.6264826668865:++;4)-*6-6/616365,\
-6715690.5; ,89,81+, (023096/:40(8-7751)2)65;695(855(+*8)+;4**+4((6.608.626482666886\
5:++;4+4)0(8)6/61638065678469.;88)-4,4*8+4((60(/6264826668865:++;4-616365676993-9:54\
+-14).:/347.+18*):1;-*0-975/)936.+:4*,80987(887(0(*)4.*""/4,4*8+4((6264826668865:\
+;4/4-4+8-4)0(8)6365678469.;88)1/(6*6,6.60626466686:8)-8*818.8582/9863(+;/"*6,6.6\
0626466686:4(8)8-8*818.8582/9863(+;/,6.60626466686:8-818.8582/9864*4+4(0())+;/.6062\
6466686:8/8380/7844,4-4*4+4(0())69+;/0626466686:818582/9864,4/4,4-4*4+4(0())+;" [e+E
+e+t]-40; E+=s+s; } n(45){ if(T>i) { v(2,T,7); v(46,T,7); } v(2+T,44,7); } T=0; o(c,
42)o(t,m)h[T][e][t]--; while(R+i) { s = D=0; if (r-R) { n(19) if (G[R+i][T]+i) V=T/2
; else if(G[R][T]+i) s++; if(s) { if(V>4){ V=9-V; D++; } V+=29; n(20) q(c[V][T][0],c
[V][T][i],D); } } n(19) if((L=G[R][T])+i) { 0=T-L; e=0>9; t=e?18-0 :0; o(K,((t&3)-3?
16:37)){ if(K){ L=c[t+19][K-1][0]; 0=c[t+19][K-1][1]; } q(L,0,K && e); } } if(s) q(
c[V][20][0], c[V][20][i], D); R--; } printf("\33[47;1f\33[725h\33[40m"); return 0; }

```



```

char *_ = ""
    /*
#include <stdio.h>
#define m 21
#define o(l, k) for (l = 0; l < k; l++)
#define n(k) o(T, k)

int E, L, O, R, G[42][m], h[2][42][m], g[3][8], c[42][42][2], f[42];
char d[42];
void v(int b, int a, int j) {
    printf("\33[%d;%df\33[4%d"
           "m ",
           a, b, j);
}
void u() {
    int T, e;
    n(42) o(e, m) if (h[0][T][e] - h[1][T][e]) {
        v(e + 4 + e, T + 2, h[0][T][e] + 1 ? h[0][T][e] : 0);
        h[1][T][e] = h[0][T][e];
    }
    fflush(stdout);
}
void q(int l, int k, int p) {
    int T, e, a;
    L = 0;
    O = 1;
    while (0) {
        n(4 && L) {
            e = k + c[l][T][0];
            h[0][L - 1 + c[l][T][1]][p ? 20 - e : e] = -1;
        }
        n(4) {
            e = k + c[l][T][0];
            a = L + c[l][T][1] + 1;
            if (a == 42 || h[0][a][p ? 20 - e : e] + 1) {
                O = 0;
            }
        }
        n(4) {
            e = k + c[l][T][0];
            h[0][L + c[l][T][1]][p ? 20 - e : e] = g[1][f[p ? 19 + l : l]];
        }
        L++;
        u();
    }
}

```

10.3.4 Modularize

为了解理解 Modularize 项目的目标，我们首先需要介绍 C++ 中的模块概念，这是偏离本章主题的闲谈。在写作此文的时候，模块还没有正式地标准化。对于 Clang 怎么为 C/C++ 项目实现新想法不感兴趣的读者，鼓励你跳过这个小节，跳到下一个工具。

理解 C/C++ API 的定义

目前，C 和 C++ 程序被分成头文件，例如扩展名为.h 的文件，和实现文件，例如扩展名为.c 或者.cpp 的文件。编译器把每个实现文件和包含文件的结合诠释为单独的翻译单元。

当以 C 或 C++ 编程的时候，如果你在一个特定的实现文件上工作，你需要考虑哪些实体属于局部作用域，哪些属于全局作用域。例如，不被不同实现文件共享的函数和数据，在 C 中应该以关键字 static 声明，或者在 C++ 中声明在匿名 namespace 中。这告诉链接器这个翻译单元不暴露局部实体，因而其它单元无法使用它们。

然而，如果你不想在不同翻译单元之间共享实体，会出现问题。为了清楚起见，让我们将导出实体的翻译单元称为 exporter，将使用这些实体的翻译单元称为 importer。我们还假设，一个名为 gamelogic.c 的 exporter 想要向名为 screen.c 的 importer 导出一个简单的整数变量，名为 num_lives。

链接器职责

首先，我们将介绍在我们的例子中链接器如何处理符号导入。在编译并汇编 gamelogic.c 之后，我们将得到一个名为 gamelogic.o 的目标文件，它的符号表显示，符号 num_lives 占用 4 个字节，其它翻译单元可以使用它。

```
$ gcc -c gamelogic.c -o gamelogic.o
$ readelf -s gamelogic.o
```

Num	Value	Size	Type	Bind	Vis	Index	Name
7	00000000	4	OBJECT	GLOBAL	DEFAULT	3	num_lives

这个表只显示了我们关注的符号，省略了其它符号。readelf 工具仅在 Linux 平台上可用，它依赖 ELF，被广泛采用的 Executable and Linkable Format。如果你使用其它平台，可以用 objdump -t 打印符号表。我们这样理解这个表：在表中符号 num_lives 被分配为第 7 个位置，占用相对于索引为 3 的段（.bss 段）的首地址（零）。反过来，.bss 段持有数据实体，它被初始化为零。为了验证段名和其索引的对应关系，用 readelf -S 或者 objdump -h 打印段的头信息。从这个表，我们还知道，符号 num_lives 是一个（数据）object，包含 4 个字节，是全局可见的（global bind）。

类似地，screen.o 文件的符号表会显示这个翻译单元依赖符号 num_lives，它属于另一个翻译单元。要想分析 screen.o，可用之前用在 gamelogic.o 上的相同命令：

```
$ gcc -c screen.c -o screen.o
$ readelf -s screen.o
```

Num	Value	Size	Type	Bind	Vis	Index	Name
10	00000000	0	NOTYPE	GLOBAL	DEFAULT	UND	num_lives

这个表项类似于 `exporter` 中的那个，只是它的信息少。它没有 `size` 和 `type`，显示哪个 ELF 段包含这个符号的 `index` 被标记为 `UND`（未定义），这标志这个翻译单元是 `importer`。如果这个翻译单元被选择编入最终的程序，链接必须解决这个依赖关系，否则就不成功。

链接器收到这两个文件作为输入，用 `importer` 请求的符号的地址对 `importer` 打补丁，这个符号在 `exporter` 中。

```
$ gcc screen.o gamelogic.o -o game
$ readelf -s game
```

Num	Value	Size	Type	Bind	Vis	Index	Name
60	0804a01c	4	OBJECT	GLOBAL	DEFAULT	25	num_lives

现在，这个值反映了程序被加载时变量的完整虚拟内存地址，向 `importer` 的代码段提供了符号的位置，完成了不同翻译单元之间的导出-导入协议。

我们得出结论，在链接器这边，在多个翻译单元之间共享实体是简单而高效的。

前端对应部分

处理目标文件是简单的，但是这并不反映在语言中。不同于链接器，在导入的实现中，编译器不能仅仅依靠导入实体的名字，因为它需要验证这个翻译单元的语义没有违反语言的类型系统，即它需要知道 `num_lives` 是一个整数。因此，编译器期望得到导入实体的名字连同类型信息。回顾历史可知，C 通过引入头文件解决这个问题。

头文件包含实体的名字连同类型信息，它们被不同的翻译单元使用。在这个模型中，导入者用 `include` 指令加载它要导入的实体的类型信息。然而，头文件的用法不止于此，事实上，它还可以带入任意的 C 或 C++ 代码，不只是声明。

依赖 C/C++ 预处理器的的问题

和如 Java 中的语言指令 `import` 不同，`Include` 指令的语义不要求为编译器提供导入符号的必要信息，而是展开成更多需要被解析的 C 或 C++ 代码。这个机制由预处理器实现，它不加思考地在实际编译前复制并修补代码，相当于一个文本处理工具。

代码量的膨胀在 C++ 代码中更复杂，C++ 模板鼓励在头文件中实现完整的类，它之后变成大量额外的 C++ 代码被注入到所有使用头文件的导入者中。

这使得 C 或 C++ 项目的编译增加沉重的负担，因为它们依赖于很多库（或者外部定义的实体），编译器需要多次解析很多头文件，为每个编译单元解析一次它所用到的头文件。

备注：回顾历史，实体的导入和导出，曾经可以由扩展的符号表解决，如今需要仔细地解析人类编写的成千上万行代码。

大型的编译器项目往往用一个预编译头文件方法来避免重复词法解析每个头文件，例如，Clang 的 PCH 文件。然而，这仅仅缓解了问题，因为编译仍然需要重新解释整个头文件，鉴于可能存在新的宏定义，这影响当前翻译单元如何解释这个头文件。

举例来说，假设我们的游戏以下面的方式实现 gamelogic.h：

```
#ifndef PLATFORM_A
extern uint32_t num_lives;
#else
extern uint16_t num_lives;
#endif
```

当 screen.c 包含这个文件时，导入的实体 num_lives 的类型依赖于是否在翻译单元 screen.c 的上下文中定义了宏 PLATFORM_A。而且，对于另一个翻译单元，这个上下文不是必须相同的。这强制编译器加载头文件的额外的代码，每当不同的翻译单元包含头文件时。

为了控制 C/C++ 导入以及如何编写库接口，模块提出了一个描述此接口的新的方法，它是讨论中的标准的一部分。此外，Clang 已经在实现对模块的支持了。

理解模块的工作方式

你的翻译单元可以导入一个模块，它定义一个清晰无歧义的接口以使用一个具体的库，而不是包含头文件。import 指令会加载由一个给定的库导出的实体，无需向你的翻译单元注入额外的 C 或 C++ 代码。

然而，目前没有已定义的导入语法，C++ 标准委员会还在讨论此特性。目前，Clang 提供了一个额外的标记，称为 fmodules，它直接将 include 解释为模块的 import 指令，当你包含一个属于模块化的库的头文件的时候。

当解析属于模块的头文件时，Clang 会生成一个它自己的实例，它的预处理器状态是干净的，以编译这些头文件，并以二进制形式缓存编译结果，以加速后续翻译单元的编译，它们依赖于相同的头文件，此头文件定义了一个特定的模块。因此，这些已成为模块一部分的头文件，不可依赖于先前定义的宏，或者其它预处理器先前的状态。

使用模块

为了将一组头文件映射为一个具体的模块，可以定义一个单独的文件，称为 module.modulemap，它提供此信息。这个文件被放置的目录，应该和定义库的 API 的头文件的目录相同。如果这个文件存在，并且以 fmodules 调用 Clang，编译就会使用模块。

让我们扩展简单游戏例子以使用模块。假设游戏 API 是由两个头文件定义的，gamelogic.h 和 screenlogic.h。主文件 game.c 从这两个文件导入实体。游戏 API 源代码的内容如下：

- gamelogic.h 文件的内容: extern int num_lives;
- screenlogic.h 文件的内容: extern int num_lines;
- gamelogic.c 文件的内容: int num_lives = 3;
- screenlogic.c 文件的内容: int num_lines = 24;

还有, 在我们的游戏 API 中, 每当用户包含 gamelogic.h 头文件时, 他也会想要包含 screenlogic.h 以在屏幕上打印游戏数据。从而, 我们将结构化我们的逻辑模块以表达这种依赖。因此, 项目的 module.modulemap 文件定义如下:

```
module MyGameLib {
    explicit module ScreenLogic {
        header "screenlogic.h"
    }
    explicit module GameLogic {
        header "gamelogic.h"
        export ScreenLogic
    }
}
```

关键字 `module` 后面跟着名字, 你期望用这个名字来识别它。在我们的例子中, 我们把它命名为 `MyGameLib`。每个模块可以有一列封闭的子模块。关键字 `explicit` 用来告诉 Clang, 当且仅当它的其中一个头文件被显式地包含时, 这个子模块被导入。你可以列出很多头文件来表示单个子模块, 但是这里我们的每个子模块只用到一个头文件。

由于我们在使用模块, 我们可以利用它们让事情变得更简单, 让 `include` 指令更简单。注意, 在 `GameLogic` 子模块的作用域, 通过在 `ScreenLogic` 子模块的名字后面使用 `export` 关键字, 我们声明, 每当用户导入 `GameLogic` 子模块时, 我们也让 `ScreenLogic` 的符号可见。

为了说明上述内容, 我们会编写 `game.c`, 即这个 API 的用户, 如下

```
// File: game.c
#include "gamelogic.h"
#include <stdio.h>
int main() {
    printf("lives= %d\nlines=%d\n", num_lives, num_lines);
    return 0;
}
```

注意, 我们用到了在 gamelogic.h 中定义的 `num_lives`, 和在 screenlogic.h 中定义的 `num_lines`, 它们不是显式包含的。然而, 当 clang 以 `-fmodules` 参数解析这个文件时, 它会转换第一个 `include` 指令, 达到 `import GameLogic` 子模块的效果, 使得在 `ScreenLogic` 中定义的符号可见。因此, 下面的命令可以正确地编译这个项目:

```
$ clang -fmodules game.c gamelogic.c screenlogic.c -o game
```

另一方面, 调用无模块系统的 Clang 将导致报告缺失符号定义:


```
$ clang game.c gamelogic.c screenlogic.c -o game
screen.c:4:50: error: use of undeclared identifier 'num_lines'; did you
mean 'num_lives'?
printf("lives= %d\nlines=%d\n", num_lives, num_lines);
^~~~~~
num_lives
```

然而，记住你希望你的项目尽可能地可移植，因此避免如下情况是令人感兴趣的，即支持模块时能正确编译，不支持时则不能。最适合采用模块的场景，是为简化库 API 的使用，和加速依赖很多公共头文件的翻译单元的编译。

理解 Modularize

一个好的示例，是改编一个已有的大型项目，让它使用模块而不是包含头文件。记住，在模块框架中，附属每个子模块的头文件是独立编译的。例如，很多项目依赖于这样的宏，它们在包含指令之前的其它文件中被定义，大概不能移植为使用模块。

`modularize` 的目的就是帮助你完成此任务。它分析一系列头文件，报告它们是否具有重复的变量定义、宏定义，或者这样的宏定义，它们可能被评估为不同的结果，依赖于预处理器的状态。它会帮助你诊断根据一系列头文件创建模块时遇到的常见的阻碍。它还检测项目是否在名字空间区域中使用 `include` 指令，这也会强制编译器在不同的作用域中解释包含的文件，此作用域和模块的概念不兼容。如此，在头文件中定义的符号必须不依赖于头文件被包含处的上下文。

使用 Modularize

要使用 `modularize`，你必须提供一个头文件的列表，它们将被逐个比较检查。继续我们的游戏项目的例子，我们会写一个新的文本文件，称为 `list.txt`，如下：

```
gamelogic.h
screenlogic.h
```

然后，简单地运行 `modularize`，以这个列表为参数：

```
$ modularize list.txt
```

如果你改变其中一个头文件，定义相同的符号，`modularize` 会报告存在不安全的模块行为，在为你的项目写入 `module.modulemap` 文件之前，你应该修正头文件。在修正头文件时，记住每个头文件应该尽可能地独立，它不应该修改它定义的符号，依赖于包含头文件的文件所定义的值。如果依赖于这种行为，你应该将这个头文件分成两个或更多，每个定义编译看到的符号，当使用一组特定的宏时。

模块映射检查器

Clang 工具模块映射检查器检查 `module.modulemap` 文件，确保它涵盖了一个目录中的所有头文件。对于前面小节的例子，用下面的命令调用它：

```
$ module-map-checker module.modulemap
```

我们讨论了使用 `include` 指令对比模块，预处理器是其中的症结。在下一节，我们会推介一个工具，它帮助你跟踪这个独特的前端组件的活动。

10.3.5 PPTrace

请看下面的引文，它来自关于 `clang::preprocessor` 的 Clang 文档，参见 http://clang.llvm.org/doxygen/classclang_1_1Preprocessor.html：

Engages in a tight little dance with the lexer to efficiently preprocess tokens. (与词法分析器紧密协作以高效地预处理标记。)

如第 4 章（前端）已经指出的那样，Clang 中的 `lexer` 类执行源代码文件分析的第一步。它将大块的文本识别成词汇，之后由解析器作解释。`lexer` 没有语义的信息，语义分析是解析器的责任，也不关心包含的头文件和宏展开，这是预处理器的责任。

Clang 的 `pp-trace` 独立工具输出预处理过程的踪迹。它实现此功能的方法是实现 `clang::PPCallbacks` 接口的回调函数。它首先将自己注册为预处理器的观察员，然后启动 Clang 以分析输入文件。对于预处理器的每个动作，例如解释 `#if` 指令，导入模块，包含头文件，等等，这个工具会在屏幕上打印消息。

考虑下面的特意编写的“hello world” C 程序：

```
#if 0
#include <stdio.h>
#endif
#ifdef CAPITALIZE
#define WORLD "WORLD"
#else
#define WORLD "world"
#endif
extern int write(int, const char*, unsigned long);
int main() {
    write(1, "Hello, ", 7);
    write(1, WORLD, 5);
    write(1, "!\n", 2);
    return 0;
}
```

在前面的代码的第一行，我们用了预处理指令 `#if`，它总是取值为假，强制编译器忽略源代码块的内容，直到下一个 `#endif` 指令。接着，我们用 `#ifdef` 指令检查是否定义了 `CAPITALIZE` 宏。根据是否定义了这个宏，宏

WORD 会被定义为大写的 WORD 字符串，或者小写的 word 字符串。最后，代码调用了一系列 write 系统调用，以在屏幕上输出消息。

运行 pp-trace，就像我们运行其它类似的 Clang 源代码分析独立工具：

```
$ pp-trace hello.c
```

结果是一系列关于宏定义的预处理器事件，甚至发生在实际的源代码被处理之前。最后的事件涉及我们的具体文件，如下：

```
- Callback: If
Loc: "hello.c:1:2"
ConditionRange: ["hello.c:1:4", "hello.c:2:1"]
ConditionValue: CVK_False
- Callback: Endif
Loc: "hello.c:3:2"
IfLoc: "hello.c:1:2"
- Callback: SourceRangeSkipped
Range: ["hello.c:1:2", "hello.c:3:2"]
- Callback: Ifdef
Loc: "hello.c:5:2"
MacroNameTok: CAPITALIZE
MacroDirective: (null)
- Callback: Else
Loc: "hello.c:7:2"
IfLoc: "hello.c:5:2"
- Callback: SourceRangeSkipped
Range: ["hello.c:5:2", "hello.c:7:2"]
- Callback: MacroDefined
MacroNameTok: WORLD
MacroDirective: MD_Define
- Callback: Endif
Loc: "hello.c:9:2"
IfLoc: "hello.c:5:2"
- Callback: MacroExpands
MacroNameTok: WORLD
MacroDirective: MD_Define
Range: ["hello.c:13:14", "hello.c:13:14"]
Args: (null)
- Callback: EndOfFile
```

第一个事件涉及我们的第一个 #if 预处理器指令。这个区域触发了三次回调：If，Endf，和 SourceRangeSkipped。注意到里面的 #include 指令是不处理的，它被跳过了。类似地，我们看到宏 WORD 相关的事件：IfDef，Else，MacroDefined，和 Endif。最后，pp-trace 通过 MacroExpands 事件报告我们用到了宏 WORD，然后到达了文件末尾，调用了回到函数 EndOfFile。

预处理之后，前端的下一步是词法分析和解析。在下一节，我们将介绍一个工具，它研究解析器的结果，即 AST 节点。

10.3.6 Clang Query

Clang Query 工具是在 LLVM 3.5 中引入的，它能够读入一个源文件，交互地查询它所关联的 Clang AST 节点。这是一个很好的工具，帮助我们查看并学习前端如何表达每行代码。然而，它的主要目标是，让你不但能够查看程序的 AST，而且能够测试 AST 匹配器。

当编写一个重构工具时，你会对使用 AST 匹配器库感兴趣，它包含若干匹配你所感兴趣的 Clang AST 片段的断言 (predicate)。Clang Query 工具可以在开发的这个部分帮助你，因为它让你能够查看哪个 AST 节点匹配一个具体的 AST 匹配器。你可以在 `ASTMatchers.h` 中查看可用的 AST 匹配器的列表，但是你也可以根据驼峰大小写的名字，猜测表示你所感兴趣的 AST 节点的类。例如，`functionDecl` 会匹配所有 `FunctionDecl` 节点，它们表示函数声明。在你试验了哪个匹配器确切地返回你所感兴趣的节点之后，你可以在你的重构工具中用它们实现一个自动转换的方法，为了某个特定的目的。在本章的后面，我们会解释如何使用 AST 匹配器库。

作为一个查看 AST 的例子，我们会对上次 PPTrace 中用到的“hello world”代码运行 `clang-query`。Clang Query 期望你有一个编译命令 `database`。如果你在查看一个文件，它没有编译命令 `database`，就在双短划线之后给出编译命令，或者空着它，如果不需要特别的编译器选项，如下面的命令行所示：

```
$ clang-query hello.c --
```

发出这个命令之后，`clang-query` 会显示一个交互提示，等待你输入命令。你可以输入 `match` 命令和任意 AST 匹配器的名字。例如，在下面的命令中，我们让 `clang-query` 显示所有 `CallExpr` 节点：

```
clang-query> match callExpr()

Match #1:
hello.c:12:5: note: "root" node binds here
write(1, "Hello, ", 7);
^~~~~~
...
```

这个工具会突出程序中一个确切的位置，它对应于关联 `CallExpr` AST 节点的第一个标记。Clang Query 能够接受的命令列表如下：

- `help`：打印命令列表。
- `match <matcher name>` 或 `m <matcher name>`：这个命令以要求的匹配器遍历 AST。
- `set output <(diag | print | dump)>`：这个命令修改如何打印节点信息，一旦它被成功地匹配。第一个选项会打印一个 Clang 诊断消息，突出节点，这是默认选项。第二个选项会简单地打印匹配到的对应源代码的摘要，而最后的选项会调用类成员函数 `dump()`，它具有相当精妙的调试功能，还会显示所有子节点。

了解一个程序的 Clang AST 的结构的一个重要方法，是修改 `dump` 输出，匹配高层级节点。试一试：

```
lang-query> set output dump
clang-query> match functionDecl()
```

它会显示某些类的所有实例，这些类制作了所有函数体的语句和表达式，这些函数来自你所打开的 C 源代码。另一方面，记住，这种完全的 AST dump，利用 Clang Check 是更容易得到的，我们会在下一节介绍它。Clang Query 更适用于制作 AST 匹配器表达式和检查它们的结果。后面你会见证 Clang Query 如何是一个极其有用的工具，当它帮助我们制作我们的第一个代码重构工具的时候，那时我们会讲到如何产生更复杂的查询。

10.3.7 Clang Check

Clang Check 是一个非常基础的工具，它只有几百行代码，这让它易于学习。然而，它具备整个 Clang 的解析能力，因为它链接了 LibTooling。

Clang Check 让你能够解析 C/C++ 源代码文件，打印 Clang AST，或者执行基础的检查。它还可以应用 Clang 给出的“fix it”修改建议，利用为 Clang Modernizer 建造的重写器设施。

例如，假设你想要打印 program.c 的 AST，就输入下面的命令：

```
$ clang-check program.c -ast-dump --
```

注意，Clang Check 遵从 LibTooling 读取源文件的方式，你可以用一个命令 database 文件，或者在双短划线 (--) 之后输入适当的参数。

Clang Check 是一个小工具，当编写你自己的工具时，将它当作一个例子来学习。在下一小节，我们将介绍另一个小工具，让你了解小的代码重构工具能做什么。

10.3.8 去除 c_str() 调用

remove-cstr-calls 工具是一个简单的源到源转换工具的例子，也就是一个重构工具。它在工作时会识别冗余的对 std::string 对象的 c_str() 调用，并重写代码使得在特定情况下避免之。这种冗余的调用可能会出现，首先，当建造一个新的 string 对象时，通过另一个 string 对象的 c_str() 的结果，例如，std::string(myString.c_str())。这可以简化为直接使用 string 拷贝构造器，例如，str::string(myString)。其次，当建造 LLVM 的具体的 StringRef 和 Twine 类的实例时，根据 string 对象来建造。在此情况下，更优的是使用 string 对象本身，而不是其 c_str() 的结果，使用 StringRef(myString)，而不是 StringRef(myString.c_str())。

这个工具可以被完整地写在单个 C++ 文件里，它是另一个优秀的易于学习的例子，演示如何使用 LibTooling 建造重构工具。这就是我们下一个话题的主题。

10.4 编写你自己的工具

Clang 项目为使用者提供了三种接口，以利用 Clang 的特性和它的解析能力，包括语法和语义分析。首先，libclang 是和 Clang 交互的主要方式，它提供了稳定的 C API，允许外部项目将它嵌入其中，获得对整个框架的高层级的访问。这个稳定的接口试图保持对旧版本的向后兼容，避免由于发布新版的 libclang 而破坏你的软件。从其它语言使用 libclang 也是可能的，例如，使用 Clang Python 绑定。Apple Xcode，举个例子，它通过 libclang 和 Clang 交互。

其次，Clang 插件，它允许你在编译过程中添加你自己的 Pass，而不是由工具执行离线的分析，比如 Clang 静态分析器。当你每次编译一个翻译单元都要执行它时，这是有用的。因此，你需要考虑执行这种分析所需的时间，是否适合频繁地运行。另一方面，将你的分析集成到 build 系统是如此容易，就像给编译器命令增加选项。

最后的方式是我们将要探索的，就是通过 LibTooling 利用 Clang。这是一个令人激动的库，它让我们能够轻松地建造独立的工具，类似于本章中所介绍的，以代码重构或者语义检查为目标。和 LibClang 相比，LibTooling 较少为了向后兼容而妥协，让你能够完全地访问 Clang AST 结构。

10.4.1 问题定义-编写一个 C++ 代码重构工具

在本章的剩余部分，我们将介绍一个例子。假设你发起了一个虚构的创业项目，创立一种新的 C++ IDE，称为 IzzyC++。你的商业计划是吸引特定的用户，他们厌烦 IDE 不能自动地重构他们的代码。你将利用 LibTooling 制作一个简单而好用的 C++ 代码重构工具；它接受如下参数，一个 C++ 成员函数，它是完全限定的名字，和一个替换的名字。它的任务，就是找到这个成员函数的定义，将它修改为替换的名字，并且相应地修改所有对这个函数的调用。

10.4.2 配置你的源代码的位置

第一步是决定在何处存放你的工具的代码。在 LLVM 的源代码文件夹中，我们将新建一个文件夹，称为 izzyrefactor，在 tools/clang/tools/extra 中，以存放我们项目的所有文件。之后，扩展 extra 文件夹中的 Makefile，以包含你的项目。简单地，找到 DIRS 变量，并在其它 Clang 工具项目的旁边添加名字 izzyrefactor。或许你还想编辑 CMakeLists.txt 文件，假如你使用 CMake，添加新的一行：

```
add_subdirectory(izzyrefactor)
```

去到 izzyrefactor 文件夹，创建一个新的 Makefile，以标记 LLVM-build 系统你要建造一个独立的工具，它会独立于其它二进制文件而存在。使用下面的内容：

```
CLANG_LEVEL := ../../..
TOOLNAME = izzyrefactor
TOOL_NO_EXPORTS = 1
include $(CLANG_LEVEL)/../../Makefile.config
LINK_COMPONENTS := $(TARGETS_TO_BUILD) asmparser bitreader support\
```

(续下页)

(接上页)

```

mc option
USEDLIBS = clangTooling.a clangFrontend.a clangSerialization.a \
           clangDriver.a clangRewriteFrontend.a clangRewriteCore.a \
           clangParse.a clangSema.a clangAnalysis.a clangAST.a \
           clangASTMatchers.a clangEdit.a clangLex.a clangBasic.a
include $(CLANG_LEVEL)/Makefile

```

这是一个重要的文件，它指定了所有需要和你的代码链接到一起的库，这样你才能建造这个工具。可选地，你可以添加一行 `NO_INSTALL = 1`，就在设置 `TOOL_NO_EXPORTS` 这行之后，如果你不想你的新工具和其它 LLVM 工具那样被安装，当你运行 `make install` 的时候。

我们设置 `TOOL_NO_EXPORTS = 1`，因为你的工具不会使用任何插件，因此，它不需要导出符号，减小了最终程序的动态符号表的尺寸，这样也减少了动态链接并加载程序的时间。注意我们通过包含 Clang 总的 Makefile 完成了工作，它定义了编译这个项目所需的所有规则。

如果你使用 CMake 而不是自动工具配置脚本，就创建一个新的 CMakeLists.txt 文件，写入如下内容：

```

add_clang_executable(izzyrefactor
    IzzyRefactor.cpp
)
target_link_libraries(izzyrefactor
    clangEdit clangTooling clangBasic clangAST clangASTMatchers)

```

此外，如果你不想在 Clang 源代码树中 build 这个工具，你也可以将它 build 为一个独立的工具。只要使用第 4 章（前端）的末尾为驱动器工具介绍的同样的 Makefile，作稍微修改。注意我们在前面的 Makefile 中用了哪些库，在 USEDLIBS 变量中，以及我们在第 4 章（前端）的 Makefile 中用了哪些库，在 CLANGLIBS 变量中。它们引用了相同的库，除了 USEDLIBS 有 clangTooling，它包含 LibTooling。因此，在第 4 章（前端）的 Makefile 中，在 `-lclang` 这行之后，添加一行 `-lclangTooling`，就大功告成了。

10.4.3 剖析工具样板代码

你的所有代码会写在 IzzyRefactor.cpp 中。新建这个文件并开始添加初始的样板代码，如下所示：

```

int main(int argc, char **argv) {
    cl::ParseCommandLineOptions(argc, argv);
    string ErrorMessage;
    OwningPtr<CompilationDatabase> Compilations_
    ↪ (CompilationDatabase::loadFromDirectory(BuildPath, ErrorMessage));
    if (!Compilations)
        report_fatal_error(ErrorMessage);
    // ...
}

```


你的主要代码从 `ParseCommandLineOptions` 函数开始，它来自 `llvm::cl` 名字空间（command-line 实用程序）。这个函数为你不厌其烦地解析 `argv` 中的每个选项。

备注：典型地，基于 `LibTooling` 的工具会使用 `CommonOptionsParser` 对象，以轻松解析通用的选项，它们为所有重构工具所共用（参见 http://clang.llvm.org/doxygen/classclang_1_1tooling_1_1CommonOptionsParser.html 作为一个代码示例）。在这个例子中，我们用低层级的 `ParseCommandLineOptions()` 函数来确切地说明我们打算解析哪些参数，并训练你在其它不使用 `LibTooling` 的工具中使用它。然而，自由地去使用 `CommonOptionsParser`，让你的工作变得轻松（以不同的方式编写此工具，作为练习）。

你将证实，所有的 LLVM 工具都会使用 `cl` 名字空间提供的功能（http://llvm.org/docs/doxygen/html/namespacellvm_1_1cl.html），定义我们的工具在命令行中识别哪些参数，实在是简单。为此，我们声明新的模板类型 `opt` 和 `list` 的变量：

```
cl::opt<string> BuildPath(
    cl::Positional,
    cl::desc("<build-path>"));
cl::list<string> SourcePaths(
    cl::Positional,
    cl::desc("<source0> [... <sourceN>]"),
    cl::OneOrMore);
cl::opt<string> OriginalMethodName("method",
    cl::desc("Method name to replace"),
    cl::ValueRequired);
cl::opt<string> ClassName("class",
    cl::desc("Name of the class that has this method"),
    cl::ValueRequired);
cl::opt<string> NewMethodName("newname",
    cl::desc("New method name"),
    cl::ValueRequired);
```

在定义 `main` 函数前声明这五个全局变量。我们具体化了类型 `opt`，根据我们期望读取什么样的数据作为参数。例如，如果你需要读取一个数字，你会声明一个新的 `cl::opt<int>` 全局变量。

为了读取这些参数的数值，你首先需要调用 `ParseCommandLineOptions`。之后，你只需要引用关联变量的全局变量的名字，在你期望得到所关联的数据类型的代码处。例如，`NewMethodName` 会为此参数估值使用者所提供的字符串，如果你的代码期望一个字符串的话，像 `std::out << NewMethodName`。

这是怎么工作的？`opt_storage<>` 模板，就是 `opt<>` 的父类，定义了一个类，此类继承自它所管理的数据类型（此处为 `string`）。通过继承，`opt<string>` 变量也是可以被如此使用的字符串。如果 `opt<>` 类模板不能继承自被包裹的数据类型（例如，不存在 `int` 类），它会定义一个类型转换操作符，例如为 `int` 数据类型定义 `operator int()`。在你的代码中，效果是一样的；当你引用一个 `cl::opt<int>` 变量时，它会自动地转换为一个整数，并返回它所存储的数字，就是使用者在命令行中提供的数字。

我们还可以为参数指定不同的特征。在我们的例子中，我们通过指定 `cl::Positional` 使用了位置参数，这意味

着使用者不会显示地以它的名字指定参数，而是会根据它在命令行中的相对位置推断出来。我们还向 `opt` 构造器传递了一个 `desc` 对象，它定义了一段描述，当使用者在命令行中输入 `-help` 参数以打印帮助信息时，此描述信息会展示给使用者。

我们还有一个使用类型 `cl::list` 的参数，不同于 `opt`，它允许传递多个参数，在这种情况下，要处理一系列源代码文件。这些用法要求包含下面的头文件：

```
#include "llvm/Support/CommandLine.h"
```

备注：作为 LLVM 编码标准的一部分，你应该组织你的 `include` 语句，首先包含本地头文件，随后包含 Clang 和 LLVM API 头文件。当两个头文件属于相同的类别时，按字母顺序安排它们。写一个新的独立工具，它自动为你整理头文件顺序，这将是一个有趣的项目。

最后三个全局变量设定所需选项以使用我们的重构工具。第一个是名字参数 `-method`。紧随的第一个字符串指定参数名字，没有短线，而 `cl::RequiredValues` 会通知命令行解析器，指示这个值是运行这个程序所需要的。这个参数会给出方法的名字，我们的工具会去寻找这个方法，然后将它的名字修改为由 `-newname` 给出的名字。参数 `-class` 给出拥有这个方法的类的名字。

下一段来自模板代码的代码摘要管理一个新的 `CompilationDatabase` 对象。首先，我们需要包含定义 `OwningPtr` 类的头文件，它是 LLVM 库用到的智能指针，就是说，它会自动地释放所包含的指针，当它到达作用域的末尾时。

```
#include "llvm/ADT/OwningPtr.h"
```

备注：注意 Clang 版本

从 Clang/LLVM 版本 3.5 开始，人们弃用了 `OwningPtr<>` 模板，而是转向 C++ 标准的 `std::unique_ptr<>` 模板。

其次，我们需要包含 `CompilationDatabase` 类的头文件，它是我们第一次用到的正式属于 `LibTooling` 的文件：

```
#include "clang/Tooling/CompilationDatabase.h"
```

这个类负责管理编译 `database`，本章的开头解释了对它的配置。它是一个强大的编译命令的列表，这些命令是处理每个源文件所必需的，使用者用你的工具分析这些文件，这是他们感兴趣的。为了初始化这个对象，我们用到一个工厂方法，称为 `loadFromDirectory`，它会从一个特定的 `build` 目录加载编译 `database` 文件。这就是将 `build` 路径声明为输入工具的参数的目的；使用者需要指定从哪里加载他们的源文件以及编译 `database` 文件。

注意，我们给这个工厂成员函数输入两个参数：`BuildPath`，我们的 `cl::opt` 对象，它代表一个命令行对象，以及一个近期声明的 `ErrorMessage` 字符串。`ErrorMessage` 字符串会被填充一个消息，假如引擎加载编译 `database` 失败了，即工厂成员函数没有返回任何 `CompilationDatabase` 对象，这时我们会马上显示这个消息。`llvm::report_fatal_error()` 函数会触发任何已配置的 LLVM 错误处理例程，并以错误码 1 退出我们的工具。它要求包含下面的头文件：

```
#include "llvm/Support/ErrorHandler.h"
```

在我们的例子中，我们缩写了很多类的完全修饰名字，因此还需要在全局作用域添加若干个 `using` 声明，但是只要你喜欢，你可以使用完全修饰名字：

```
using namespace clang;
using namespace std;
using namespace llvm;
using clang::tooling::RefactoringTool;
using clang::tooling::Replacement;
using clang::tooling::CompilationDatabase;
using clang::tooling::newFrontendActionFactory;
```

10.4.4 使用 AST 匹配器

本章的 Clang Query 小节简单地介绍过了 AST 匹配器，但是我们在这里会深入分析其细节，因为它们对于编写基于 Clang 的代码重构工具是非常重要的。

AST 匹配器库让它的使用者能够轻松地匹配符合特定断言的 Clang AST 的子树，例如，表示对一个函数的调用的所有 AST 节点，它的名字为 `clang::CallExpr`，并且有两个参数。查找特定的 Clang AST 节点并修改它们，这是每个代码重构工具共同的基本任务，对这个库的利用极大地减轻了编写此类工具的任务。

为了帮助我们找到正确的匹配器，我们会依靠 Clang Query 和 AST 匹配器文档，文档在此处可获得：<http://clang.llvm.org/docs/LibASTMatchersReference.html>。

我们先为你的工具编写一个名为 `wildlifesim.cpp` 的测试案例。这是一个复杂的一维动物生活模拟器，其中的动物可以沿着直线向任何方向行走：

```
class Animal {
    int position;
public:
    Animal(int pos) : position(pos) {}
    // Return new position
    int walk(int quantity) {
        return position += quantity;
    }
};

class Cat : public Animal {
public:
    Cat(int pos) : Animal(pos) {}
    void meow() {}
    void destroySofa() {}
    bool wildMood() {return true;}
};
```

(续下页)

(接上页)

```
int main() {
    Cat c(50); c.meow();
    if (c.wildMood())
        c.destroySofa();
    c.walk(2);
    return 0;
}
```

我们要求你的工具能够将成员函数比如 `walk` 重命名为 `run`。让我们运行 Clang Query，研究在此例子中 AST 看起来是什么样子。我们会用 `recordDecl` 匹配器，输出所有 `RecordDecl` AST 节点的内容，它们负责表示 C 结构和 C++ 类：

```
$ clang-query wilddanimal-sim.cpp --
clang-query> set output dump
clang-query> match recordDecl()
(...)
|-CXXMethodDecl 0x(...) <line:6:3, line 8:3> line 6:7 walk 'int (int)'
(...)
```

在表示 `Animal` 类的 `RecordDecl` 对象的内部，我们观察到 `walk` 被表示为一个 `CXXMethodDecl` AST 节点。通过查看 AST 匹配器文档，我们发现它是由 `methodDecl` AST 匹配器匹配的。

组合匹配器

AST 匹配器的强大在于它们能被组合。如果我们只想要 `MethodDecl` 节点，它们声明了一个称为 `walk` 的成员函数，就可以先匹配所有名为 `walk` 的有名字声明，然后精炼之使之只匹配那些又是方法声明的节点。`hasName("input")` 匹配器返回所有名为 “input” 的有名字声明。你可以在 Clang Query 中测试 `methodDecl` 和 `hasName` 的组合：

```
clang-query> match methodDecl(hasName("walk"))
```

你将看到它只返回了一个声明，`walk` 的声明，而不是代码中存在的所有八个不同方法的声明。太好了！

尽管如此，观察到仅修改 `Animal` 类的 `walk` 方法的定义是不够的，因为派生的类可能重载它。我们不希望我们的重构工具重写了基类的一个方法而不重写派生类中重载的其它方法。

我们需要找到所有定义了 `walk` 方法的类，它们是 `Animal` 类或者其派生类。为了找到所有 `Animal` 类或者其派生类，我们使用匹配器 `isSameOrDerivedFrom()`，它期望一个 `NamedDecl` 参数。这个参数将通过和一个匹配器的组合来提供，这个匹配器选择具有特定名字的所有 `NamedDecl`，`hasName()`。因此，我们的查询看起来是这样的：

```
clang-query> match recordDecl(isSameOrDerivedFrom(hasName("Animal")))
```

我们还需要选择那些重载了 `walk` 方法的派生类。`hasMethod()` 断言返回包含具体方法的类声明。我们将它和第一个查询组合成如下查询：

```
clang-query> match recordDecl(hasMethod(methodDecl(hasName("walk"))))
```

为了用 `and` 操作符语义（所有的断言必须成立）连结两个断言，我们使用 `allOf()` 匹配器。它规定所有作为操作数输入的匹配器必须成立。此时我们准备好了建造我们最终的查询，以找到我们将重写的所有声明：

```
clang-query> match recordDecl(allOf(hasMethod(methodDecl(hasName("walk"))),  
→ isSameOrDerivedFrom(hasName("Animal"))))
```

利用这个查询，我们能够精确地找到 `Animal` 类或者其派生类的所有 `walk` 方法的声明。

这允许我们修改所有这些声明的名字，但是我们还需要修改方法的调用。为此，我们先来考察 `CXXMemberCallExpr` 节点和它的匹配器 `memberCallExpr`。试一下：

```
clang-query> match memberCallExpr()
```

`Clang Query` 返回四个匹配，因为我们的代码确实含有四个方法调用：`meow`，`wildMood`，`destroySofa`，和 `walk`。我们只对定位最后一个感兴趣。我们已经知道如何利用 `hasName()` 匹配器来选择具有特定名字的声明，但是如何将具有名字的声明映射到成员函数调用的表达式呢？答案是使用 `member()` 匹配器来只选择具有名字且和一个方法名字相链接的声明，然后使用 `callee()` 匹配器将它们和调用表达式链接起来。完整的表达式如下：

```
clang-query> match memberCallExpr(callee(memberExpr(member(hasName("walk")))))
```

然而，这样的做法，我们盲目地选择了所有对 `walk()` 方法的调用。我们只想选择那些确实指向 `Animal` 类或者其派生类的 `walk` 调用。`memberCallExpr()` 匹配器接受第二个匹配器作为参数。我们会使用 `thisPointerType()` 匹配器以只选择那些方法调用，其被调用的对象是特定的类。利用这个规则，我们构建了完整的表达式：

```
clang-query> match memberCallExpr(callee(memberExpr(member(hasName("walk"))),  
→ thisPointerType(recordDecl(isSameOrDerivedFrom(hasName("Animal")))))
```

在代码中运用 AST 匹配器断言

我们已经决定了用哪些断言来捕获正确的 AST 节点，是时候在我们的工具的代码中运用它们了。首先，为了使用 AST 匹配器，我们需要添加新的 `include` 指令：

```
#include "clang/ASTMatchers/ASTMatchers.h"  
#include "clang/ASTMatchers/ASTMatchFinder.h"
```

我们还需要添加新的 `using` 指令，使得易于引用这些类（写在其它的 `using` 指令后面）：

```
using namespace clang::ast_matchers;
```

第二个头文件是使用实际的查找器机制所必须的，马上我们会介绍它。从之前停止的地方继续编写 `main` 函数，我们开始添加剩余的代码：

```
RefactoringTool Tool(*Compilations, SourcePaths);
ast_matchers::MatchFinder Finder;
ChangeMemberDecl DeclCallback(&Tool.getReplacements());
ChangeMemberCall CallCallback(&Tool.getReplacements());
Finder.addMatcher(recordDecl(allOf(hasMethod(id("methodDecl", ↵
↵methodDecl(hasName(OriginalMethodName)))),
    isSameOrDerivedFrom(hasName(Classname)))), &DeclCallback);
Finder.addMatcher(memberCallExpr(callee(id("member", ↵
↵memberExpr(hasName(OriginalMethodName)))),
    thisPointerType(recordDecl(isSameOrDerivedFrom(hasName(Classname)))), &
↵CallCallback);
return Tool.runAndSave(newFrontendActionFactory(&Finder));
```

备注： 注意 Clang 版本：在版本 3.5 中，你需要将以上代码的最后一行修改为 `return Tool.runAndSave(newFrontendActionFactory(&Finder.get()));` 为了使它能工作。

这完成了 `main` 函数的整个代码。之后我们会介绍回调函数的代码。

第一行代码实例化了一个新的 `RefactoringTool` 对象。这是我们用到的 `LibTooling` 的第二个类，它需要一个另外的语句：

```
#include "clang/Tooling/Refactoring.h"
```

`RefactoringTool` 类为你的工具实现了协调基本任务的所有逻辑，例如打开源文件，解析它们，运行 AST 匹配器，当匹配发生时调用你的回调函数以执行一个动作，并按照你的工具的要求修改源代码。这就回答了为什么在初始化所有需要的对象之后，我们要调用 `RefactoringTool::runAndSave()`，然后才结束 `main` 函数。我们将控制转移到这个类，让它执行所有基本任务。

接下来，我们声明了一个 `MatchFinder` 对象，其头文件已经包含了。这个类负责对 Clang AST 执行匹配，这是你已经用 Clang Query 练习过的。`MatchFinder` 要求配置 AST 匹配器和回调函数，当所提供的 AST 匹配器匹配一个 AST 节点时，回调函数就会被调用。在这个回调函数中，你将有机会修改源代码。回调函数会被实现为一个 `MatchCallback` 的子类，之后我们会探讨它。

然后，我们接着声明回调函数对象，并且用 `MatchFinder::addFinder()` 方法将一个具体的 AST 匹配器关联到一个回调函数。我们声明两个单独的回调函数，一个用于重写方法声明，另一个用于重写方法调用。我们将这两个回调函数命名为 `DeclCallback` 和 `CallCallback`。我们使用前面小节设计的两个 AST 匹配器组合，但是我们用 `ClassName` 替换类名字 `Animal`，这是命令行参数，使用者会用它提供他们的要被重构的类名字。还有，我们用 `OriginalMethodName` 替换 `walk`，这也是命令行参数。

我们还战略性地引入了新的匹配器，称为 `id()`，它不修改表达式所匹配的节点，只是将一个名字绑定到一个具体的节点。这是非常重要的，使得回调函数能够产生替换内容。`id()` 匹配器接受两个参数，第一个是节点的名字，你会利用它获取节点，第二个是匹配器，它会捕获带名字的 AST。

第一个 AST 组合负责定位成员方法声明，在其中我们命名了 `MethodDecl` 节点，它识别方法。第二个 AST 组合负责定位对成员函数的调用，在其中我们命名了 `CXXMemberExpr` 节点，它和被调用的成员函数相链接。

10.4.5 编写回调函数

你需要定义当 AST 节点被匹配时要执行的动作。我们为此创建两个新的类，它们派生自 `MatchCallback`，每个匹配行为各有一个类。

```
class ChangeMemberDecl : public ast_matchers::MatchFinder::MatchCallback {
    tooling::Replacements *Replace;
public:
    ChangeMemberDecl(tooling::Replacements *Replace) :
        Replace(Replace) {}
    virtual void run(const ast_matchers::MatchFinder::MatchResult &Result) {
        const CXXMethodDecl *method = Result.Nodes.getNodeAs<CXXMethodDecl>
↪ ( "methodDecl" );
        Replace->insert(Replacement(*Result.SourceManager, ↪
↪ CharSourceRange::getTokenRange(SourceRange(method->getLocation()))), NewMethodName));
    }
};

class ChangeMemberCall : public ast_matchers::MatchFinder::MatchCallback {
    tooling::Replacements *Replace;
public:
    ChangeMemberCall(tooling::Replacements *Replace) :
        Replace(Replace) {}
    virtual void run(const ast_matchers::MatchFinder::MatchResult &Result) {
        const MemberExpr *member = Result.Nodes.getNodeAs<MemberExpr>( "member" );
        Replace->insert(Replacement(*Result.SourceManager, ↪
↪ CharSourceRange::getTokenRange(SourceRange(member->getMemberLoc()))), ↪
↪ NewMethodName));
    }
};
```

这两个类都存储了对 `Replacements` 对象的私有的引用，它只是一个对 `std::set<Replacement>` 的 typedef。 `Replacement` 类存储的信息包括，哪些行需要打补丁，在哪个文件，以及用哪部分文本。它的序列化，我们在介绍 Clang Apply Replacements 时讨论过了。 `RefactoringTool` 类在内部管理 `Replacement` 对象的集合，这解释了为什么我们在 `main` 函数中利用 `RefactoringTool::getReplacements()` 方法获得这个集合，并且初始化我们的回调函数。

我们定义了一个基本的构造器，它的参数是一个指向 `Replacements` 对象的指针，我们会存储它，为以后的使用。我们会通过重载 `run()` 方法，来实现回调函数的动作，又一次地，它的代码出奇地简单。我们的函数接受一个 `MatchResult` 对象作为参数。对于一个给定的匹配， `MatchResult` 类存储了绑定一个名字的所有节点，如我们的 `id()` 匹配器要求的那样。

这些节点由 `BoundNodes` 类管理，它们在 `MatchResult` 对象中是公开可见的，可通过节点名字访问。因此，我们在 `run()` 函数中的第一个动作是得到我们感兴趣的节点，通过调用专门的方法 `BoundNodes::getNodeAs<CXXMethodDecl>`。结果，我得到了一个指向 `CXXMethodDecl` AST 节点的只读版本的引用。

获得了对这个节点的访问之后，为了决定如何给代码打补丁，我们需要一个 `SourceLocation` 对象，它告诉我们关联的标记在源文件中所占据的确切的行和列。`CXXMethodDecl` 继承自基类 `Decl`，它表示通用的声明。这个通用的类提供了 `Decl::getLocation()` 方法，它返回的正是我们想要的 `SourceLocation` 对象。有了这个信息，我们就可以创建我们的第一个 `Replacement` 对象，并将它插入到我们的工具所建议的源代码修改的列表中。

我们用到的 `Replacement` 构造器需要三个参数：一个 `SourceManager` 对象的引用，一个 `CharSourceRange` 对象的引用，和一个包含新的文本的字符串，这个字符串将被写入到由头两个参数指定的位置。`SourceManager` 类是一个普通的 Clang 组件，它管理加载到内存的源代码。`CharSourceRange` 类包含有用的分析程序，它分析标记并推导出组成这个标记的源代码范围（文件中的两个点），从而决定需要从源代码文件中删除的确切的字符，而为新的文本空出位置。

我们用这个信息创建一个新的 `Replacement` 对象，并且将它存储在由 `RefactoringTool` 管理的 `set` 中，就完成任务了。实际上，`RefactoringTool` 会应用这些补丁，或者去除冲突的补丁。不要忘记将所有本地的声明包裹在一个匿名的名字空间里；这是一个不让这个翻译单元导出本地符号的好办法。

10.4.6 测试你的新重构工具

我们将用我们的野生动物模拟器代码例子作为一个测试案例，来测试你新创建的工具。现在你应该运行 `make`，然后等待 LLVM 完成对你的新工具的编译和链接。工具生成之后，尽情地试用一番。看看我们声明为 `cl::opt` 对象的参数在命令行接口中是什么样子：

```
$ izzyrefactor -help
```

为了使用这个工具，我们还需要一个编译命令 `database`。为了避免要创建并运行一个 CMake 配置文件，我们将手动地创建一个。将它命名为 `compile_commands.json`，写入下面的代码。将标签 `<FULLPATHTOFILE>` 替换为完整的路径，指向你放置野生动物模拟器源代码的文件夹：

```
[
{
  "directory": "<FULLPATHTOFILE>",
  "command": "/usr/bin/c++ -o wildlifesim.cpp.o -c <FULLPATHTOFILE>/ wildlifesim.cpp",
  "file": "<FULLPATHTOFILE>/wildlifesim.cpp"
}
]
```

保存这个编译命令 `database` 之后，就可以测试工具了：

```
$ izzyrefactor -class=Animal -method=walk -newname=run ./ wildfilesim.cpp
```

现在你可以检查野生动物模拟器源代码，会看到这个工具重命名了所有方法的定义和调用。这结束了我们的指导，但是你可以在下一节查看更多的资源，并进一步扩展你的知识。

10.5 更多资源

你可以从下面的链接找到更多的资源：

- <http://clang.llvm.org/docs/HowToSetupToolingForLLVM.html>: 这个链接包含关于如何设置命令 `database` 的指令。一旦你有了这个文件，你甚至可以配置你喜欢的文本编辑器，来运行一个工具以按需检查代码。
- <http://clang.llvm.org/docs/Modules.html>: 这个链接给出了关于实现 Clang C/C++ 模块的更多信息。
- <http://clang.llvm.org/docs/LibASTMatchersTutorial>: 这是另一个关于使用 AST 匹配器和 LibTooling 的教程。
- <http://clang.llvm.org/extra/clang-tidy.html>: 这里有 Clang Tidy 的用户手册，伴随其它工具的用户手册。
- <http://clang.llvm.org/docs/ClangFormat.html>: 这里包含了 ClangFormat 的用户手册。
- <http://www.youtube.com/watch?v=yuIOGfcOH0k>: 这里包含了 Chandler Carruth 对 C++Now 的介绍，解释了如何建造一个重构工具。

10.6 总结

在这一章中，我们介绍了建立在 LibTooling 基础之上的 Clang 工具，它们让你能够轻松地编写操作 C/C++ 源代码的工具。我们介绍了如下工具：Clang Tidy，Clang 的剥绒机；Clang Modernizer，它自动地将旧的 C++ 编程方式替换为新的；Clang Apply Replacements，它负责应用由其它工具创建的补丁；ClangFormat，它自动地缩进和格式化 C++ 代码；Modularize，它让使用未标准化的 C++ 模块框架变得容易；PPTrace，它文档化预处理器的活动；Clang Query，它让你能够测试 AST 匹配器。最后，我们通过演示如何创建你自己的工具结束了这一章。

到此本书该结束了，但是这绝对不应该是你学习旅途的终点。网络上有很多关于 Clang 和 LLVM 的额外资料，不是教程就是正式文档。还有，Clang/LLVM 总是在进化并引入新的特性，值得我们继续学习。要了解这些内容，请访问 LLVM 的博客：<http://blog.llvm.org>。

黑客快乐！

CHAPTER 11

Indices and tables

- `genindex`
- `modindex`
- `search`