

# 自制编译器

How to Develop a Compiler

[日] 青木峰郎 / 著 严圣逸 绝云 / 译

TURING

图灵程序  
设计丛书



从零开始制作真正的编译器

贯穿编译、汇编、链接、加载的全过程！

比“龙书”更具实践性！



中国工信出版集团



人民邮电出版社  
POSTS & TELECOM PRESS

---

提供各种书籍的pd电子版代找服务，如果你找不到自己想要的书的pdf电子版，我们可以帮您找到，如有需要，请联系QQ或微信2028969416.

PDF代找说明：

本人可以帮助你找到你要的PDF电子书，计算机类，文学，艺术，设计，医学，理学，经济，金融，等等。质量都很清晰，而且每本100%都带书签索引和目录，方便读者阅读观看，只要您提供给我书的相关信息，一般我都能找到，如果您有需求，请联系我微信或QQ2028969416。

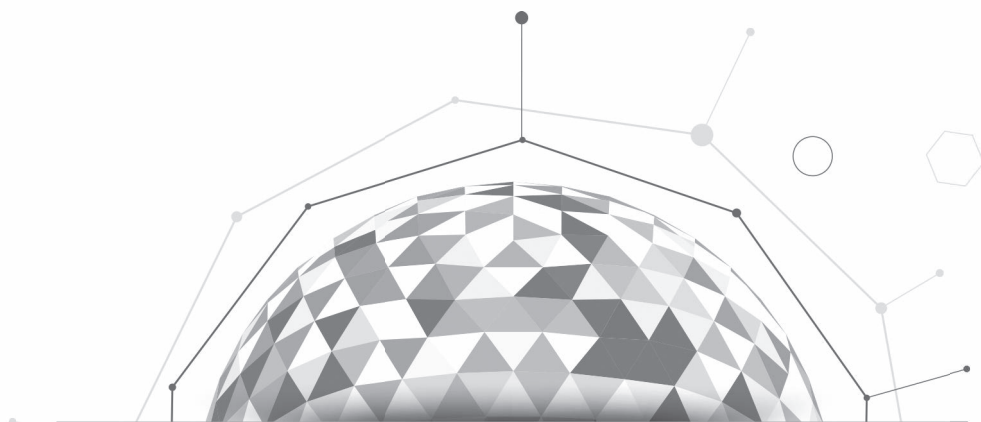
本人已经帮助了上万人找到了他们需要的PDF，其实网上有很多PDF,大家如果在网上不到的话，可以联系我QQ，大部分我都可以找到，而且每本100%带书签索引目录。因PDF电子书都有版权，请不要随意传播，如果您有经济购买能力，请尽量购买正版。

**声明：本人只提供代找服务，每本100%索引书签和目录，因寻找pdf电子书有一定难度，仅收取代找费用。如因PDF产生的版权纠纷，本人概不负责，我们仅仅只是帮助你寻找到你要的pdf而已。**

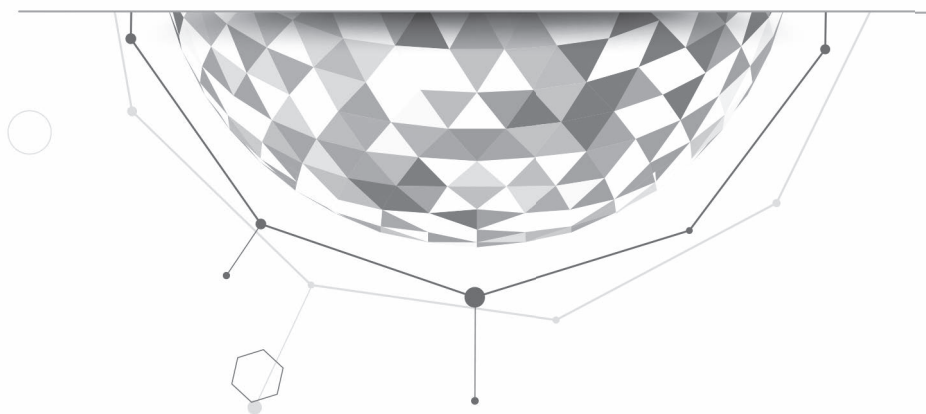
**TURING**  
图灵程序  
设计丛书

# 自制编译器

[日] 青木峰郎 / 著 严圣逸 绝云 / 译



How to Develop a Compiler



人民邮电出版社  
北 京

## 图书在版编目(CIP)数据

自制编译器 / (日) 青木峰郎著; 严圣逸, 绝云译

. -- 北京: 人民邮电出版社, 2016.6

(图灵程序设计丛书)

ISBN 978-7-115-42218-7

I. ①自… II. ①青… ②严… ③绝… III. ①C语言  
—编译器—程序设计 IV. ①TP312

中国版本图书馆CIP数据核字(2016)第083048号

## 内 容 提 要

本书将带领读者从头开始制作一门语言的编译器。笔者特意为本书设计了Cb语言,Cb可以说是C语言的子集,实现了包括指针运算等在内的C语言的主要部分。本书所实现的编译器就是Cb语言的编译器,是实实在在的编译器,而非有诸多限制的玩具。另外,除编译器之外,本书对以编译器为中心的编程语言的运行环境,即编译器、汇编器、链接器、硬件、运行时环境等都有所提及,介绍了程序运行的所有环节。

从单纯对编译器感兴趣的读者到以实用为目的的读者,都适合阅读本书。

---

◆ 著 [日] 青木峰郎

译 严圣逸 绝云

责任编辑 乐 馨

执行编辑 杜晓静

责任印制 彭志环

◆ 人民邮电出版社出版发行 北京市丰台区成寿寺路11号

邮编 100164 电子邮件 315@ptpress.com.cn

网址 <http://www.ptpress.com.cn>

北京 印刷

◆ 开本: 800×1000 1/16

印张: 29.5

字数: 605千字

2016年6月第1版

印数: 1-3 000册

2016年6月北京第1次印刷

著作权合同登记号 图字: 01-2014-5502号

---

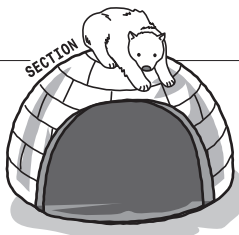
定价: 99.00元

读者服务热线: (010)51095186转600 印装质量热线: (010)81055316

反盗版热线: (010)81055315

广告经营许可证: 京东工商广字第8052号





## 译者序

算上这本《自制编译器》，图灵的“自制”系列应该已经有 6 本了。从 CPU 到操作系统，从编译器到编程语言，再到搜索引擎等具体的应用，俨然已经可以自制一套完整的计算机体系了。

“自制”系列图书都是从日本引进并翻译出版的，本人也有幸读过其中几本。可能有很多读者和曾经的我一样对“自制”抱有疑惑：“在时间就是金钱、时间就是生命的 IT 行业，为什么会有这样的自制风潮？为什么要自制呢？CPU 可以用 Intel、AMD，操作系统已经有了 Windows、Linux，搜索引擎已经有了 Google、Yahoo，编程语言及其对应的编译器、解释器更是已经百花齐放、百家争鸣……”直到翻译完本书，我才逐渐体会到自制是最好的结合实践学习的方式之一。拿来的始终是别人的，要吃透某项技术、打破技术垄断，最好的方法就是自制。并且从某种程度上来说，自制也是一种创新，可能下一个 Google 或 Linux 就孕育在某次自制之中。

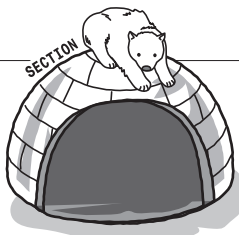
自制编译器的目的是了解当前实用的编程语言、编译器和 OS 的相关知识，绝对不能闭门造车。因此作者使用的 Cb 语言是 C 语言的子集，实现了包括指针运算在内的 C 语言的主要部分，通过自制一个 Cb 语言的编译器，能够让我们了解 C 语言程序编译、运行背后的细节；OS 选用 Linux，能够让我们知晓 Linux 上的链接、加载和程序库；汇编部分采用最常见的 x86 系统架构。作者自制的编译器 cbc 能够运行在 x86 架构的任何发行版本的 Linux 上，编译 Cb 代码并生成可执行的 ELF 文件。

作者青木先生在致谢中提到了 Linux 和 GNU 工具等开源软件的开发者。这也是本书的另一大特色：充分利用开源软件和工具。从 GCC 到 GNU Assembler 再到 JavaCC 以及 Linux，并非每一行代码都是自己写的才算自制，根据自己的设计合理有效地利用开源软件，既可以让我们更快地看到自制的成果，又能向优秀的开源软件学习。如果要深入学习、研究，那么开源软件的源代码以及活跃的社区等都是非常有帮助的。而如果把自制的软件也作为开源软件上传到 Github 上供大家使用，并根据其他开发者提出的 Pull Request 不断改进软件，那就更好了。

最后我要由衷地感谢本书的另一位译者绝云老师以及图灵的编辑。还要特别感谢我的外公，一位毕生耕耘于教育出版行业的老编辑。自己能有幸参加翻译，和从小对出版工作的耳闻目染是密不可分的。

严圣逸

2016 年 4 月于上海



## 前言

本书有两大特征：第一，实际动手实现了真正的编译器；第二，涉及了以往编译器相关书籍所不曾涉及的内容。

先说第一点。

本书通篇讲述了“Cb”这种语言的编译器的制作。Cb 基本上是 C 语言的子集，并实现了包括指针运算等在内的 C 语言的主要部分。因此可以说，本书实现的是实实在在的编译器，而并非有诸多限制的玩具。

更具体地说，本书实现的 Cb 编译器是以运行在 x86 系列 CPU 上的 Linux 为平台的。之所以选择 x86 系列的 CPU，是因为这是最普及的 CPU，相应的硬件非常容易找到。选择 Linux 是因为从标准库到程序运行环境的代码都是公开的，只要你有心，完全可以自己分析程序的结构。

可能有些作者不喜欢把话题局限于特定的语言或者 OS，而笔者却恰恰更倾向于在一开始就对环境进行限定。因为比起一般化的说明，从具体的环境出发，再向一般化扩展的做法要简单、直观得多。笔者赞成最终把话题往一般化的方向扩展，但并不赞成一开始就一定要做到一般化。

再说第二点。

本书并不局限于书名中的“编译器”，对以编译器为中心的编程语言的运行环境，即编译器、汇编器、链接器、硬件、运行时环境都有所涉及。

编译器生成的程序的运行不仅和编译器相关，和汇编器、链接器等软件以及硬件都密切相关。因此，如果了解编译器以及程序的运行结果，对上述几部分内容的了解当然是必不可少的。不过这里的“当然”现在看起来也逐渐变得没那么绝对了。

只讲编译器或者只讲汇编语言的书已经多得烂大街了，只讲链接器的书也有一些，但是贯穿上述所有内容的书至今还没有。写编译器的书，一涉及具体的汇编语言，就会注上“请参考其他书籍”；写汇编语言的书，对于 OS 的运行环境问题却只字不提；写链接器的书，如果读者不了解编译器等相关知识，也就只能被束之高阁了。

难道就不可能完整地记述编程语言的运行环境吗？笔者认为可能的。只要专注于具体的语言、具体的 OS 以及具体的硬件，就可以对程序运行的所有环节进行说明了。基于这样的想法，笔者进行了稍显鲁莽的尝试，并最终写成了本书。

以上就是本书的基本原则。下面是本书的读者对象。

- 想了解编译器和解释器内部结构的人
- 想了解 C 语言程序运行机制的人
- 想了解 x86 CPU（Pentium 或 Intel Core、Operon 等）的结构的人
- 想了解 Linux 上的链接、加载和程序库的人
- 想学习语法分析的人
- 想设计新的编程语言的人

综上，本书是一本基于具体的编程语言、具体的硬件平台以及具体的 OS 环境，介绍程序运行的所有环节的书。因此，从单纯对编译器感兴趣的读者到以实用为目的的读者，都适合阅读本书。



## 必要的知识

---

本书的读者需要具备以下知识。

- Java 语言的基础知识
- C 语言的基础知识
- Linux 的基础知识

本书中制作的 Cb 编译器是用 Java 来实现的，所以能读懂 Java 代码是阅读本书的前提条件。不只是语言，书中对集合等基本库也都没有任何说明，因此需要读者具备相关的知识储备。

本书所使用的 Java 版本是 5.0。关于泛化（generics）和 foreach 语句等 Java 5 特有的功能，在第一次出现时会进行简单的说明。

另外，之所以需要读者具有 C 语言的基础知识，是因为 Cb 语言是 C 语言的子集，另外，以 C 语言的知识为基础，对汇编器的理解也将变得容易得多。不过读者不需要深究细节，只要能够理解指针和结构体可以组合使用这种程度就足够了。

最后，关于 shell 的使用方法以及 Linux 方面的常识，这里也不作介绍。例如 cd、ls、cp、mv 等基本命令的用法，都不会进行说明。



## 不必要的知识

---

本书的读者不需要具备以下知识。

- 编译器和解释器的构造
- 解析器生成器的使用方法
- 操作系统的详细知识
- 汇编语言
- 硬件知识

即使读者对编译器和解释器的构造一无所知，也没有关系，本书会对此进行详尽的说明。

另外，OS 及 CPU 相关的前提知识也基本不需要。能用 Linux 的 shell 进行文件操作，用 gcc 命令编译 C 语言的“Hello,World”程序，这样就足够了。



## 本书的结构

本书由以下章节构成。

章		内容
第 1 章	开始制作编译器	本书概要以及了解编译器所需要的基础知识
第 2 章	Cb 和 cbc	本书制作的 Cb 编译器的概要
第 1 部分	代码分析	
第 3 章	语法分析的概要	语法分析的概念及方法
第 4 章	词法分析	cbc 的词法分析（扫描）
第 5 章	基于 JavaCC 的解析器的描述	JavaCC 的使用方法（语法部分）
第 6 章	语法分析	cbc 的语法分析
第 2 部分	抽象语法树和中间代码	
第 7 章	JavaCC 的 action 和抽象语法树	JavaCC 的使用方法（action 部分）
第 8 章	抽象语法树的生成	根据语法分析的结果生成语法树的方法
第 9 章	语义分析（1）引用的消解	变量的引用和具体定义之间的消解
第 10 章	语义分析（2）静态类型检查	编译时的类型检查
第 11 章	中间代码的转换	从抽象语法树生成中间代码
第 3 部分	汇编代码	
第 12 章	x86 架构的概要	使用 Intel 系列 CPU 的系统的构造
第 13 章	x86 汇编器编程	x86 CPU 的汇编语言的读法
第 14 章	函数和变量	x86 CPU 架构中函数调用的形式
第 15 章	编译表达式和语句	和栈帧无关的汇编代码的生成
第 16 章	分配栈帧	和栈帧相关的汇编代码的生成
第 17 章	优化的方法	优化程序的方法的概要
第 4 部分	链接和加载	
第 18 章	生成目标文件	ELF 文件的构造和生成
第 19 章	链接和库	链接的种类和库
第 20 章	加载程序	内存中程序的加载及动态链接
第 21 章	生成地址无关代码	地址无关代码及共享库的生成
第 22 章	扩展阅读	为读者的后续学习介绍相关知识

编译器自身也是一款程序，它将程序的代码逐次进行转换，最终生成可以运行的文件。因此前面章节的内容会成为后续章节的前提，推荐从头开始依次阅读本书的所有章节。

但是，如果你对编译器有一定程度的了解，并且只对特定的话题感兴趣，也可以选取相应的章节来阅读。本书做成的 Cb 编译器可以显示每个阶段生成的数据结构，因此你也可以实际运行一下 Cb 编译器，一边确认前一阶段生成的结果，一边往下阅读。

例如，即使跳过语法分析的章节，只要用 `--dump-ast` 选项显示前一阶段生成的抽象语法

树，就可以理解下一阶段的语义分析和中间代码的相关内容。同样，还可以用 `--dump-ir` 选项显示中间代码，用 `--dump-sam` 选项显示汇编代码。

## 致谢

---

首先感谢 RHG 读书会的成员阅读了第 2 部分之前的草稿，并提出了很多宝贵意见。感谢笹田、山下、酒井、向井、shelarcy、志村、岸本、丰福、佐野。

还要感谢 3 年来一直耐心地等待笔者交稿的 SB Creative 株式会社的杉山，以及在短时间内对本书 600 余页的稿件进行编辑的 Top Studio Corporation 株式会社的武藤。非常感谢！

最后，感谢为本书出版付出努力的各位，以及所有维护 Linux 和 GNU 工具等自由软件的人。正是因为有了你们，本书才得以出版。

青木峰郎

# 目 录

## 第 1 章

### 开始制作编译器

1

1.1	本书的概要	2
	本书的主题	2
	本书制作的编译器	2
	编译示例	2
	可执行文件	3
	编译	4
	程序运行环境	6
1.2	编译过程	8
	编译的 4 个阶段	8
	语法分析	8
	语义分析	9
	生成中间代码	9
	代码生成	10
	优化	10
	总结	10
1.3	使用 Cb 编译器进行编译	11
	Cb 编译器的必要环境	11
	安装 Cb 编译器	11
	Cb 的 Hello, World!	12

## 第 2 章

### Cb 和 cbc

13

2.1	Cb 语言的概要	14
	Cb 的 Hello, World !	14
	Cb 中删减的功能	14
	import 关键字	15
	导入文件的规范	16
2.2	Cb 编译器 cbc 的构成	17
	cbc 的代码树	17

cbc 的包 .....	18
compiler 包中的类群 .....	18
main 函数的实现 .....	19
commandMain 函数的实现 .....	19
Java5 泛型 .....	20
build 函数的实现 .....	20
Java 5 的 foreach 语句 .....	21
compile 函数的实现 .....	21

---

# 第 1 部分 代码分析

---

## 第 3 章

语法分析的概要 .....	24
---------------	----

3.1 语法分析的方法 .....	25
代码分析中的问题点 .....	25
代码分析的一般规律 .....	25
词法分析、语法分析、语义分析 .....	25
扫描器的动作 .....	26
单词的种类和语义值 .....	27
token .....	28
抽象语法树和节点 .....	29
3.2 解析器生成器 .....	30
什么是解析器生成器 .....	30
解析器生成器的种类 .....	30
解析器生成器的选择 .....	31
3.3 JavaCC 的概要 .....	33
什么是 JavaCC .....	33
语法描述文件 .....	33
语法描述文件的例子 .....	34
运行 JavaCC .....	35
启动 JavaCC 所生成的解析器 .....	36
中文的处理 .....	37

## 第 4 章

词法分析 .....	39
------------	----

4.1 基于 JavaCC 的扫描器的描述 .....	40
-----------------------------	----

本章的目的 .....	40
JavaCC 的正则表达式 .....	40
固定字符串 .....	41
连接 .....	41
字符组 .....	41
排除型字符组 .....	41
重复 1 次或多次 .....	42
重复 0 次或多次 .....	42
重复 n 次到 m 次 .....	42
正好重复 n 次 .....	43
可以省略 .....	43
选择 .....	43
<b>4.2 扫描没有结构的单词 .....</b>	<b>44</b>
TOKEN 命令 .....	44
扫描标识符和保留字 .....	44
选择匹配规则 .....	45
扫描数值 .....	46
<b>4.3 扫描不生成 token 的单词 .....</b>	<b>48</b>
SKIP 命令和 SPECIAL_TOKEN 命令 .....	48
跳过空白符 .....	48
跳过行注释 .....	49
<b>4.4 扫描具有结构的单词 .....</b>	<b>50</b>
最长匹配原则和它的问题 .....	50
基于状态迁移的扫描 .....	50
MORE 命令 .....	51
跳过块注释 .....	52
扫描字符串字面量 .....	53
扫描字符字面量 .....	53

## 第 5 章

### 基于 JavaCC 的解析器的描述

55

<b>5.1 基于 EBNF 语法的描述 .....</b>	<b>56</b>
本章的目的 .....	56
基于 JavaCC 的语法描述 .....	56
终端符和非终端符 .....	57
JavaCC 的 EBNF 表示法 .....	58
连接 .....	58
重复 0 次或多次 .....	59
重复 1 次或多次 .....	59



选择 .....	60
可以省略 .....	60
5.2 语法的二义性和 token 的超前扫描 .....	61
语法的二义性 .....	61
JavaCC 的局限性 .....	62
提取左侧共通部分 .....	63
token 的超前扫描 .....	63
可以省略的规则和冲突 .....	64
重复和冲突 .....	65
更灵活的超前扫描 .....	66
超前扫描的相关注意事项 .....	66

**第 6 章**

<b>语法分析</b>	<b>68</b>
-------------	-----------

6.1 定义的分析 .....	69
表示程序整体的符号 .....	69
语法的单位 .....	69
import 声明的语法 .....	70
各类定义的语法 .....	71
变量定义的语法 .....	72
函数定义的语法 .....	73
结构体定义和联合体定义的语法 .....	74
结构体成员和联合体成员的语法 .....	75
typedef 语句的语法 .....	76
类型的语法 .....	76
C 语言和 Cb 在变量定义上的区别 .....	77
基本类型的语法 .....	77
6.2 语句的分析 .....	79
语句的语法 .....	79
if 语句的语法 .....	80
省略 if 语句和大括号 .....	80
while 语句的语法 .....	81
for 语句的语法 .....	81
各类跳转语句的语法 .....	82
6.3 表达式的分析 .....	83
表达式的整体结构 .....	83
expr 的规则 .....	83
条件表达式 .....	84
二元运算符 .....	85

6.4	项的分析.....	88
	项的规则 .....	88
	前置运算符的规则 .....	88
	后置运算符的规则 .....	89
	字面量的规则 .....	89

---

## 第 2 部分 抽象语法树和中间代码

---

### 第 7 章

JavaCC 的 action 和抽象语法树	92
------------------------	----

7.1	JavaCC 的 action.....	93
	本章的目的 .....	93
	简单的 action.....	93
	执行 action 的时间点 .....	93
	返回语义值的 action.....	95
	获取终端符号的语义值.....	95
	Token 类的属性.....	96
	获取非终端符号的语义值 .....	98
	语法树的结构 .....	99
	选择和 action.....	99
	重复和 action.....	100
	本节总结 .....	102
7.2	抽象语法树和节点.....	103
	Node 类群 .....	103
	Node 类的定义 .....	105
	抽象语法树的表示 .....	105
	基于节点表示表达式的例子.....	107

### 第 8 章

抽象语法树的生成	110
----------	-----

8.1	表达式的抽象语法树.....	111
	字面量的抽象语法树 .....	111
	类型的表示 .....	112
	为什么需要 TypeRef 类 .....	113
	一元运算的抽象语法树 .....	114
	二元运算的抽象语法树 .....	116

条件表达式的抽象语法树 .....	117
赋值表达式的抽象语法树 .....	118
<b>8.2 语句的抽象语法树 .....</b>	<b>121</b>
if 语句的抽象语法树 .....	121
while 语句的抽象语法树 .....	122
程序块的抽象语法树 .....	123
<b>8.3 声明的抽象语法树 .....</b>	<b>125</b>
变量声明列表的抽象语法树 .....	125
函数定义的抽象语法树 .....	126
表示声明列表的抽象语法树 .....	127
表示程序整体的抽象语法树 .....	128
外部符号的 import .....	128
总结 .....	129
<b>8.4 cbc 的解析器的启动 .....</b>	<b>132</b>
Parser 对象的生成 .....	132
文件的解析 .....	133
解析器的启动 .....	134

## 第 9 章

### 语义分析 (1) 引用的消解 135

<b>9.1 语义分析的概要 .....</b>	<b>136</b>
本章目的 .....	136
抽象语法树的遍历 .....	137
不使用 Visitor 模式的抽象语法树的处理 .....	137
基于 Visitor 模式的抽象语法树的处理 .....	138
Visitor 模式的一般化 .....	140
cbc 中 Visitor 模式的实现 .....	141
语义分析相关的 cbc 的类 .....	142
<b>9.2 变量引用的消解 .....</b>	<b>144</b>
问题概要 .....	144
实现的概要 .....	144
Scope 树的结构 .....	145
LocalResolver 类的属性 .....	146
LocalResolver 类的启动 .....	146
变量定义的添加 .....	147
函数定义的处理 .....	148
pushScope 方法 .....	149
currentScope 方法 .....	149

popScope 方法	150
添加临时作用域	150
建立 VariableNode 和变量定义的关联	151
从作用域树取得变量定义	151
<b>9.3 类型名称的消解</b>	<b>153</b>
问题概要	153
实现的概要	153
TypeResolver 类的属性	153
TypeResolver 类的启动	154
类型的声明	154
类型和抽象语法树的遍历	155
变量定义的类型消解	156
函数定义的类型消解	157

**第 10 章**

**语义分析（2）静态类型检查**

**159**

<b>10.1 类型定义的检查</b>	<b>160</b>
问题概要	160
实现的概要	161
检测有向图中的闭环的算法	162
结构体、联合体的循环定义检查	163
<b>10.2 表达式的有效性检查</b>	<b>165</b>
问题概要	165
实现的概要	165
DereferenceChecker 类的启动	166
SemanticError 异常的捕获	167
非指针类型取值操作的检查	167
获取非左值表达式地址的检查	168
隐式的指针生成	169
<b>10.3 静态类型检查</b>	<b>170</b>
问题概要	170
实现的概要	170
Cb 中操作数的类型	171
隐式类型转换	172
TyperChecker 类的启动	173
二元运算符的类型检查	174
隐式类型转换的实现	175

**第 11 章****中间代码的转换****178**

11.1 cbc 的中间代码.....	179
组成中间代码的类 .....	180
中间代码节点类的属性 .....	181
中间代码的运算符和类型 .....	182
各类中间代码 .....	183
中间代码的意义 .....	184
11.2 IRGenerator 类的概要 .....	185
抽象语法树的遍历和返回值 .....	185
IRGenerator 类的启动 .....	185
函数本体的转换 .....	186
作为语句的表达式的判别 .....	187
11.3 流程控制语句的转换 .....	189
if 语句的转换 (1) 概要 .....	189
if 语句的转换 (2) 没有 else 部分的情况 .....	190
if 语句的转换 (3) 存在 else 部分的情况 .....	191
while 语句的转换 .....	191
break 语句的转换 (1) 问题的定义 .....	192
break 语句的转换 (2) 实现的方针 .....	193
break 语句的转换 (3) 实现 .....	194
11.4 没有副作用的表达式的转换 .....	196
UnaryOpNode 对象的转换 .....	196
BinaryOpNode 对象的转换 .....	197
指针加减运算的转换 .....	198
11.5 左值的转换 .....	200
左边和右边 .....	200
左值和右值 .....	200
cbc 中左值的表现 .....	201
结构体成员的偏移 .....	202
成员引用 (expr.memb) 的转换 .....	203
左值转换的例外: 数组和函数 .....	204
成员引用的表达式 (ptr->memb) 的转换 .....	205
11.6 存在副作用的表达式的转换 .....	206
表达式的副作用 .....	206
有副作用的表达式的转换方针 .....	206
简单赋值表达式的转换 (1) 语句 .....	207
临时变量的引入 .....	208

简单赋值表达式的转换（2）表达式..... 209

后置自增的转换..... 210

第 3 部分 汇编代码

第 12 章

x86 架构的概要..... 214

12.1 计算机的系统结构..... 215

    CPU 和存储器 ..... 215

    寄存器 ..... 215

    地址..... 216

    物理地址和虚拟地址 ..... 216

    各类设备 ..... 217

    缓存..... 218

12.2 x86 系列 CPU 的历史..... 220

    x86 系列 CPU ..... 220

    32 位 CPU ..... 220

    指令集 ..... 221

    IA-32 的变迁 ..... 222

    IA-32 的 64 位扩展——AMD64 ..... 222

12.3 IA-32 的概要 ..... 224

    IA-32 的寄存器..... 224

    通用寄存器..... 225

    机器栈 ..... 226

    机器栈的操作 ..... 227

    机器栈的用途 ..... 227

    栈帧..... 228

    指令指针 ..... 229

    标志寄存器..... 229

12.4 数据的表现形式和格式..... 231

    无符号整数的表现形式..... 231

    有符号整数的表现形式..... 231

    负整数的表现形式和二进制补码 ..... 232

    字节序 ..... 233

    对齐..... 233

    结构体的表现形式 ..... 234

**第 13 章****x86 汇编器编程****236**

13.1 基于 GNU 汇编器的编程 .....	237
GNU 汇编器 .....	237
汇编语言的 Hello, World! .....	237
基于 GNU 汇编器的汇编代码 .....	238
13.2 GNU 汇编器的语法 .....	240
汇编版的 Hello, World! .....	240
指令 .....	241
汇编伪操作 .....	241
标签 .....	241
注释 .....	242
助记符后缀 .....	242
各种各样的操作数 .....	243
间接内存引用 .....	244
x86 指令集的概要 .....	245
13.3 传输指令 .....	246
mov 指令 .....	246
push 指令和 pop 指令 .....	247
lea 指令 .....	248
movsx 指令和 movzx 指令 .....	249
符号扩展和零扩展 .....	250
13.4 算术运算指令 .....	251
add 指令 .....	251
进位标志 .....	252
sub 指令 .....	252
imul 指令 .....	252
idiv 指令和 div 指令 .....	253
inc 指令 .....	254
dec 指令 .....	255
neg 指令 .....	255
13.5 位运算指令 .....	256
and 指令 .....	256
or 指令 .....	257
xor 指令 .....	257
not 指令 .....	257
sal 指令 .....	258
sar 指令 .....	258

shr 指令 .....	259
<b>13.6 流程的控制 .....</b>	<b>260</b>
jmp 指令 .....	260
条件跳转指令 ( jz、jnz、je、jne、..... ) .....	261
cmp 指令 .....	262
test 指令 .....	263
标志位获取指令 ( SETcc ) .....	263
call 指令 .....	264
ret 指令 .....	265

## 第 14 章

### 函数和变量

266

<b>14.1 程序调用约定 .....</b>	<b>267</b>
什么是程序调用约定 .....	267
Linux/x86 下的程序调用约定 .....	267
<b>14.2 Linux/x86 下的函数调用 .....</b>	<b>269</b>
到函数调用完成为止 .....	269
到函数开始执行为止 .....	270
到返回原处理流程为止 .....	271
到清理操作完成为止 .....	271
函数调用总结 .....	272
<b>14.3 Linux/x86 下函数调用的细节 .....</b>	<b>274</b>
寄存器的保存和复原 .....	274
caller-save 寄存器和 callee-save 寄存器 .....	274
caller-save 寄存器和 callee-save 寄存器的灵活应用 .....	275
大数值和浮点数的返回方法 .....	276
其他平台的程序调用约定 .....	277

## 第 15 章

### 编译表达式和语句

278

<b>15.1 确认编译结果 .....</b>	<b>279</b>
利用 cbc 进行确认的方法 .....	279
利用 gcc 进行确认的方法 .....	280
<b>15.2 x86 汇编的对象与 DSL .....</b>	<b>282</b>
表示汇编的类 .....	282
表示汇编对象 .....	283
<b>15.3 cbc 的 x86 汇编 DSL .....</b>	<b>285</b>



- 利用 DSL 生成汇编对象 ..... 285
- 表示寄存器 ..... 286
- 表示立即数和内存引用 ..... 287
- 表示指令 ..... 287
- 表示汇编伪操作、标签和注释 ..... 288
- 15.4 CodeGenerator 类的概要 ..... 290
  - CodeGenerator 类的字段 ..... 290
  - CodeGenerator 类的处理概述 ..... 290
  - 实现 compileStmts 方法 ..... 291
  - cbc 的编译策略 ..... 292
- 15.5 编译单纯的表达式 ..... 294
  - 编译 Int 节点 ..... 294
  - 编译 Str 节点 ..... 294
  - 编译 Uni 节点 (1) 按位取反 ..... 295
  - 编译 Uni 节点 (2) 逻辑非 ..... 297
- 15.6 编译二元运算 ..... 298
  - 编译 Bin 节点 ..... 298
  - 实现 compileBinaryOp 方法 ..... 299
  - 实现除法和余数 ..... 300
  - 实现比较运算 ..... 300
- 15.7 引用变量和赋值 ..... 301
  - 编译 Var 节点 ..... 301
  - 编译 Addr 节点 ..... 302
  - 编译 Mem 节点 ..... 303
  - 编译 Assign 节点 ..... 303
- 15.8 编译 jump 语句 ..... 305
  - 编译 LabelStmt 节点 ..... 305
  - 编译 Jump 节点 ..... 305
  - 编译 CJump 节点 ..... 305
  - 编译 Call 节点 ..... 306
  - 编译 Return 节点 ..... 307

**第 16 章**

**分配栈帧**

**308**

- 16.1 操作栈 ..... 309
  - cbc 中的栈帧 ..... 309
  - 栈指针操作原则 ..... 310
  - 函数体编译顺序 ..... 310

16.2 参数和局部变量的内存分配..... 312

    本节概述 ..... 312

    参数的内存分配..... 312

    局部变量的内存分配：原则 ..... 313

    局部变量的内存分配 ..... 314

    处理作用域内的局部变量 ..... 315

    对齐的计算 ..... 316

    子作用域变量的内存分配 ..... 316

16.3 利用虚拟栈分配临时变量..... 318

    虚拟栈的作用 ..... 318

    虚拟栈的接口 ..... 319

    虚拟栈的结构 ..... 319

    virtualPush 方法的实现 ..... 320

    VirtualStack#extend 方法的实现 ..... 320

    VirtualStack#top 方法的实现 ..... 321

    virtualPop 方法的实现 ..... 321

    VirtualStack#rewind 方法的实现 ..... 321

    虚拟栈的运作 ..... 322

16.4 调整栈访问的偏移量..... 323

    本节概要 ..... 323

    StackFrameInfo 类 ..... 323

    计算正在使用的 callee-save 寄存器 ..... 324

    计算临时变量区域的大小 ..... 325

    调整局部变量的偏移量 ..... 325

    调整临时变量的偏移量 ..... 326

16.5 生成函数序言和尾声..... 327

    本节概要 ..... 327

    生成函数序言 ..... 327

    生成函数尾声 ..... 328

16.6 alloca 函数的实现 ..... 330

    什么是 alloca 函数 ..... 330

    实现原则 ..... 330

    alloca 函数的影响 ..... 331

    alloca 函数的实现 ..... 331

**第 17 章**

**优化的方法**

17.1 什么是优化..... 334

各种各样的优化.....	334
优化的案例.....	334
常量折叠.....	334
代数简化.....	335
降低运算强度.....	335
削除共同子表达式.....	335
消除无效语句.....	336
函数内联.....	336
<b>17.2 优化的分类.....</b>	<b>337</b>
基于方法的优化分类.....	337
基于作用范围的优化分类.....	337
基于作用阶段的优化分类.....	338
<b>17.3 cbc 中的优化.....</b>	<b>339</b>
cbc 中的优化原则.....	339
cbc 中实现的优化.....	339
cbc 中优化的实现.....	339
<b>17.4 更深层的优化.....</b>	<b>341</b>
基于模式匹配选择指令.....	341
分配寄存器.....	342
控制流分析.....	342
大规模的数据流分析和 SSA 形式.....	342
总结.....	343

---

## 第 4 部分 链接和加载

---

### 第 18 章

<b>生成目标文件</b>	<b>346</b>
---------------	------------

<b>18.1 ELF 文件的结构.....</b>	<b>347</b>
ELF 的目的.....	347
ELF 的节和段.....	348
目标文件的主要 ELF 节.....	348
使用 readelf 命令输出节头.....	349
使用 readelf 命令输出程序头.....	350
使用 readelf 命令输出符号表.....	351
readelf 命令的选项.....	351
什么是 DWARF 格式.....	352

18.2	全局变量及其在 ELF 文件中的表示 .....	354
	分配给任意 ELF 节 .....	354
	分配给通用 ELF 节 .....	354
	分配 .bss 节 .....	355
	通用符号 .....	355
	记录全局变量对应的符号 .....	357
	记录符号的附加信息 .....	357
	记录通用符号的附加信息 .....	358
	总结 .....	358
18.3	编译全局变量 .....	360
	generate 方法的实现 .....	360
	generateAssemblyCode 方法的实现 .....	360
	编译全局变量 .....	361
	编译立即数 .....	362
	编译通用符号 .....	363
	编译字符串字面量 .....	364
	生成函数头 .....	365
	计算函数的代码大小 .....	366
	总结 .....	366
18.4	生成目标文件 .....	367
	as 命令调用的概要 .....	367
	引用 GNUAssembler 类 .....	367
	调用 as 命令 .....	367

## 第 19 章

### 链接和库

369

19.1	链接的概要 .....	370
	链接的执行示例 .....	370
	gcc 和 GNU ld .....	371
	链接器处理的文件 .....	372
	常用库 .....	374
	链接器的输入和输出 .....	374
19.2	什么是链接 .....	375
	链接时进行的处理 .....	375
	合并节 .....	375
	重定位 .....	376
	符号消解 .....	377
19.3	动态链接和静态链接 .....	379

两种链接方法 .....	379
动态链接的优点 .....	379
动态链接的缺点 .....	380
动态链接示例 .....	380
静态链接示例 .....	381
库的检索规则 .....	381
<b>19.4 生成库 .....</b>	<b>383</b>
生成静态库 .....	383
Linux 中共享库的管理 .....	383
生成共享库 .....	384
链接生成的共享库 .....	385

## 第 20 章

### 加载程序

**387**

<b>20.1 加载 ELF 段 .....</b>	<b>388</b>
利用 mmap 系统调用进行文件映射 .....	388
进程的内存镜像 .....	389
内存空间的属性 .....	390
ELF 段对应的内存空间 .....	390
和 ELF 文件不对应的内存空间 .....	392
ELF 文件加载的实现 .....	393
<b>20.2 动态链接过程 .....</b>	<b>395</b>
动态链接加载器 .....	395
程序从启动到终止的过程 .....	395
启动 ld.so .....	396
系统内核传递的信息 .....	397
AUX 矢量 .....	397
读入共享库 .....	398
符号消解和重定位 .....	399
运行初始化代码 .....	400
执行主程序 .....	401
执行终止处理 .....	402
ld.so 解析的环境变量 .....	402
<b>20.3 动态加载 .....</b>	<b>404</b>
所谓动态加载 .....	404
Linux 下的动态加载 .....	404
动态加载的架构 .....	405
<b>20.4 GNU ld 的链接 .....</b>	<b>406</b>
用于 cbc 的 ld 选项的结构 .....	406

C 运行时 ..... 407

生成可执行文件 ..... 408

生成共享库 ..... 408

**第 21 章**

**生成地址无关代码** **410**

21.1 地址无关代码 ..... 411

    什么是地址无关代码 ..... 411

    全局偏移表 ( GOT ) ..... 412

    获取 GOT 地址 ..... 412

    使用 GOT 地址访问全局变量 ..... 413

    访问使用 GOT 地址的文件内部的全局变量 ..... 414

    过程链接表 ( PLT ) ..... 414

    调用 PLT 入口 ..... 416

    地址无关的可执行文件: PIE ..... 416

21.2 全局变量引用的实现 ..... 418

    获取 GOT 地址 ..... 418

    PICThunk 函数的实现 ..... 418

    删除重复函数并设置不可见属性 ..... 419

    加载 GOT 地址 ..... 420

    locateSymbols 函数的实现 ..... 421

    全局变量的引用 ..... 421

    访问全局变量: 地址无关代码的情况下 ..... 422

    函数的符号 ..... 423

    字符串常量的引用 ..... 424

21.3 链接器调用的实现 ..... 425

    生成可执行文件 ..... 425

    generateSharedLibrary 方法 ..... 426

21.4 从程序解析到执行 ..... 428

    build 和加载的过程 ..... 428

    词法分析 ..... 429

    语法分析 ..... 429

    生成中间代码 ..... 430

    生成代码 ..... 431

    汇编 ..... 432

    生成共享库 ..... 432

    生成可执行文件 ..... 433

    加载 ..... 433

**第 22 章**  
**扩展阅读**

**434**

22.1 参考书推荐 ..... 435

    编译器相关 ..... 435

    语法分析相关 ..... 435

    汇编语言相关 ..... 436

22.2 链接、加载相关 ..... 437

22.3 各种编程语言的功能 ..... 438

    异常封装相关的图书 ..... 438

    垃圾回收 ..... 438

    垃圾回收相关的图书 ..... 439

    面向对象编程语言的实现 ..... 439

    函数式语言 ..... 440

**附 录**

**441**

A.1 参考文献 ..... 442

A.2 在线资料 ..... 444

A.3 源代码 ..... 445

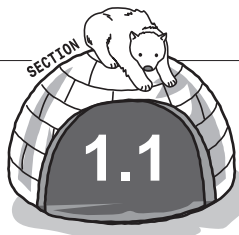
# 第 1 章

## 开始制作编译器

本章先讲述本书以及编译器的概要，之后说明本书的示例程序 Cb 的安装方法。







## 本书的概要

这节将对本书的概要进行说明。



### 本书的主题

本书的主题是编译器。编译器（compiler）是将编程语言的代码转换为其他形式的软件。这种转换操作就称为编译（compile）。

实际的编译器有 C 语言的编译器 GCC（GNU Compiler Collection）、Java 语言的编译器 javac（Sun 公司）等。

像编译器这样复杂的软件，仅仅笼统地介绍一下是很难让人理解的，所以本书将从头开始制作一门语言的编译器。通过实际地设计、制作编译器，使读者对编译器产生具体、深刻的认识。这样通过实践获得的知识，在其他语言的编译器上也是通用的。



### 本书制作的编译器

本书将从头开始制作 Cb<sup>①</sup> 这门语言的编译器。

Cb 是笔者为本书设计的语言，基本上可以说是 C 语言的子集。它在 C 语言的基础上进行了简化，并加入了一些时兴的功能，使得与之配套的编译器制作起来比较容易。笔者最初想直接使用 C 语言的，但是 C 语言的编译器无论写起来还是读起来都非常难，所以最终放弃了。关于 Cb 的标准，第 2 章会详细说明。

使用本书的 Cb 编译器编译出的程序是在 PC 的 Linux 平台上运行的。最近，借助虚拟机以及 KNOPPIX 等，Linux 环境已经很容易搭建了。请读者一定要实际用 Cb 编译器编译程序，并尝试运行一下。



### 编译示例

接着让我们赶紧进入编译器的正题。

首先我们来思考一下编译究竟是一种什么样的处理。这里以使用 GCC 处理代码清单 1.1 中

① b 为降调符号（读音同“降”，表示把基本音符音高降低半音。——译者注

的 C 语言程序为例进行说明。实际编译下面的程序时，需要重新安装 GCC。

代码清单 1.1 hello.c

```
#include <stdio.h>

int
main(int argc, char **argv)
{
    printf("Hello, World!\n"); /* 打个招呼 */
    return 0;
}
```

本书的读者对象是已经掌握 C 语言知识的人，所以理应编译过 C 语言程序。但保险起见，还是确认一下编译的步骤。使用 GCC 处理上述程序，需要输入如下命令。

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

这样便生成了名为 `hello` 的文件，这是个可执行文件（executable file）。

接着输入下面的命令，运行刚才生成的 `hello` 命令。

```
$ ./hello
Hello, World!
```

通过这样操作来运行程序本身没有问题，但从过程来看，还是有一些不明确的地方。

- 可执行文件是怎样的文件
- gcc 命令是如何生成可执行文件的
- 可执行文件 `hello` 是经过哪些步骤运行起来的

让我们依次看一下上述疑问。



## 可执行文件

首先从 GCC 生成的可执行文件是什么说起。

说到现代的 Linux 上的可执行文件，通常是指符合 ELF（Executable and Linking Format）这种特定形式的文件。`ls`、`cp` 这些命令（command）对应的实体文件都是可执行文件，例如 `/bin/ls` 和 `/bin/cp` 等。

使用 `file` 命令能够查看文件是否符合 ELF 的形式。例如，要查看 `/bin/ls` 文件是不是 ELF，在 shell 中输入如下命令即可。

```
$ file /bin/ls
/bin/ls: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/Linux 2.4.1, dynamically linked (uses shared libs), for GNU/Linux 2.4.1, stripped
```

如果像这样显示 `ELF.....executable`，就表示该文件为 ELF 的可执行文件。根据所使用的 Linux 机器的不同，可能显示 `ELF 64-bit`，也可能显示 `ELF 32-bit MSB`，这些都是 ELF 的可执行文件。

ELF 文件中包含了程序（代码）以及如何运行该程序的相关信息（元数据）。程序（代码）就是**机器语言**（machine language）的列表。机器语言是唯一一种 CPU 能够直接执行的语言，不同种类的 CPU 使用不同的机器语言。

例如，现在基本上所有的个人计算机使用的都是 Intel 公司的 486 这款 CPU 的后续产品，486 有着自己专用的机器语言。Sun 公司的 SPARC 系列 CPU 使用的是其他机器语言。IBM 公司的 PowerPC 系列 CPU 使用的又是不一样的机器语言。486 的机器语言不能在 SPARC 上运行，反过来 SPARC 的机器语言也不能在 486 上运行。这点在 SPARC 和 PowerPC、486 和 PowerPC 上也一样。

GCC 将 C 语言的程序转化为用机器语言（例如 486 的机器语言）描述的程序。将机器语言的程序按照 ELF 这种特定的文件格式注入文件，得到的就是可执行文件。



## 编译

---

那么 gcc 命令是如何将 `hello.c` 转换为可执行文件的呢？

由 `hello.c` 这样的单个文件来生成可执行文件时，虽然只需要执行一次 gcc 命令，但实际上其内部经历了如下 4 个阶段的处理。

1. 预处理
2. （狭义的）编译
3. 汇编
4. 链接

上述处理也可以统称为编译，但严谨地说第 2 阶段进行的狭义的编译才是真正意义上的编译。本书中之后所提到的编译，指的就是狭义的编译。这 4 个阶段的处理我们统称为 **build**。

下面对这 4 个阶段的处理的作用进行简单的说明。

### 预处理

C 语言的代码首先由**预处理器**（preprocessor）对 `#include` 和 `#define` 进行处理。具体来说，读入头文件，将所有的宏展开，这就是**预处理**（preprocess）。预处理的英文是 `pre-process`，就是前处理的意思。这里的“前”是在什么之前呢？当然是编译之前了。

预处理的内容近似于 `sed` 命令和 `awk` 命令这样的纯文本操作，不考虑 C 语言语法的含义。

## 狭义的编译

接着，编译器对预处理器的输出进行编译，生成**汇编语言**（assemble language）的代码。一般来说，汇编语言的代码的文件扩展名是“.s”。

汇编语言是由机器语言转换过来的人类较易阅读的文本形式的语言。机器语言是以 CPU 的执行效率为第一要素设计的，用二进制代码表示，每一个 bit 都有自己的含义，人类很难理解。因此，一般要使用与机器语言直接对应的汇编语言，以方便人们理解。

## 汇编

然后，汇编语言的代码由**汇编器**（assembler）转换为机器语言，这个处理过程称为**汇编**（assemble）。

汇编器的输出称为**目标文件**（object file）。一般来说，目标文件的扩展名是“.o”。

Linux 中，目标文件也是 ELF 文件。既然都是 ELF 文件，那么究竟是目标文件还是可执行文件呢？这不是区分不了了吗？这个不用担心。ELF 文件中有用于提示文件种类的标志。例如，用 file 命令来查看目标文件，会像下面这样显示 ELF...relocatable，据此就能够将其和可执行文件区分开。

```
$ file t.o
t.o: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped
```

## 链接

目标文件本身还不能直接使用，无论是直接运行还是作为**程序库**（library）文件调用都不可以。将目标文件转换为最终可以使用的形式的处理称为**链接**（link）。使用程序库的情况下，会在这个阶段处理程序库的加载

例如，假设 Hello,World! 程序经过编译和汇编生成了目标文件 hello.o，链接 hello.o 即可生成可执行文件。生成的可执行文件的默认文件名为 a.out，可以使用 gcc 命令的 -o 选项来修改输出的文件名。

顺便提一下，通过链接处理生成的并不一定是可执行文件，也可以是程序库文件。程序库文件相关的话题将在第 19 章中详细说明。

## build 过程总结

如上所述，C 语言的代码经过预处理、编译、汇编、链接这 4 个阶段的处理，最终生成可执行文件。图 1.1 中总结了各个阶段的输出文件，我们再来确认一下。

本书将对这 4 个处理阶段中除预处理之外的编译、汇编和链接进行说明。

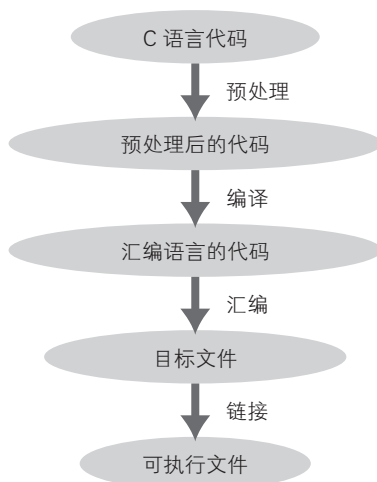


图 1.1 生成可执行文件的过程



## 程序运行环境

build<sup>①</sup>的过程以链接为终点，但本书并不仅仅局限于 build 的过程，还会涉及 build 之后的程序运行环境相关的话题。从代码的编写、编译、运行到运行结束，理解上述全部过程是我们的目标。换言之，从编写完程序到该程序被运行，所有环节本书都会涉及（图 1.2）。

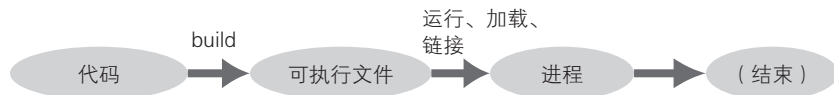


图 1.2 程序运行的全过程

为何除了 build 的过程之外，本书还要涉及程序运行的环节呢？这是因为在现代编程语言的运行过程中，运行环境所起的作用越来越大。

首先，链接的话题并非仅仅出现在 build 的过程中。如果使用了共享库，那么在开始运行程序时，链接才会发生。最近广泛使用的**动态加载**（dynamic load），就是一种将所有链接处理放到程序运行时进行的手法。

其次，像 Java 和 C# 这种语言的运行环境中都有**垃圾回收**（Garbage Collection，GC）这一强大的功能，该功能对程序的运行有着很大的影响。

再次，在 Sun 的 Java VM 等具有代表性的 Java 的运行环境中，为了提高运行速度，采用了**JIT 编译器**（Just In Time compiler）。JIT 编译器是在程序运行时进行处理，将程序转换为机器语言的编译器。也就是说，Java 语言是在运行时进行编译的。

<sup>①</sup> build 有“构建”“生成”等译法，但似乎都不能表达出其全意，因此本书保留了英文用法。——译者注

既然涉及了这样的话题，仅了解 build 的过程是不够的，还必须了解程序的运行环境。不掌握包含运行环境在内的整个流程，就不能说完全理解了程序的动作。今后，无论是理解程序还是制作编译器，都需要了解从 build 到运行环境的整体流程。



## 编程语言的运行方式

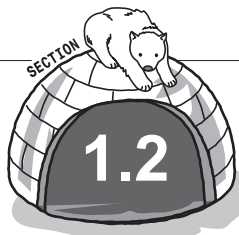
编译器会对程序进行编译，将其转换为可执行的形式。另外也有不进行编译，直接运行编程语言的方法。**解释器**（interpreter）就是这样一个例子。解释器不将程序转换为别的语言，而是直接运行。例如 Ruby 和 Perl 的语言处理器就是用解释器来实现的。

运行语言的手段不只一种。例如，C 语言也可以用解释器来解释执行，Ruby 也可以编译成机器语言或者 Java 的二进制码。也就是说，编程语言与其运行方式可以自由搭配。因此，编译器也好，解释器也罢，都是处理并运行编程语言的手段之一，统称为**编程语言处理器**（programming language processor）。

但是，根据语言的特点，其运行方式有适合、不适合该语言之说。一般来说，有**静态类型检查**（static type checking）、要求较高可靠性的情况下使用编译的方式；相反，没有静态类型检查、对灵活性的要求高于严密性的情况下，则使用解释的方式。

静态类型检查是指在程序开始运行之前，对函数的返回值以及参数的类型进行检查的功能。与之相对，在程序运行过程中随时进行类型检查的方式称为**动态类型检查**（dynamic type checking）。

这里提到的“动态”“静态”在语言处理器的话题中经常出现，所以最好记住。说到“静态”，就是指不运行程序而进行某些处理；说到“动态”，就是指一边运行程序一边进行某些处理。



## 编译过程

这一节将对狭义的编译的内部处理过程进行介绍。



### 编译的 4 个阶段

狭义的编译大致可分为下面 4 个阶段。

1. 语法分析
2. 语义分析
3. 生成中间代码
4. 代码生成

下面就依次对这 4 个阶段进行说明。



### 语法分析

一般我们所说的编写程序，就是把代码写成人可读的文本文件的形式。像 C 和 Java 这样，以文本形式编写的代码对人类来说的确易于阅读，但并不是易于计算机理解的形式。因此，为了运行 C 和 Java 的程序，首先要对代码进行解析，将其转换为计算机易于理解的形式。这里的解析（parse）也称为语法分析（syntax analyzing）。解析代码的程序模块称为解析器（parser）或语法分析器（syntax analyzer）。

那么“易于计算机理解的形式”究竟是怎样的形式呢？那就是称为语法树（syntax tree）的形式。顾名思义，语法树是树状的构造。将代码转化为语法树形式的过程如图 1.3 所示。

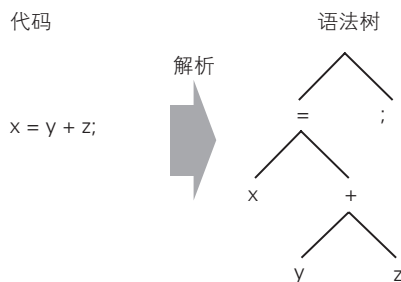


图 1.3 语法树

## 语义分析

通过解析代码获得语法树后，接着就要解析语法树，除去多余的内容，添加必要的信息，生成**抽象语法树**（Abstract Syntax Tree，AST）这样一种数据结构。上述处理就是**语义分析**（semantic analysis）。

语法分析只是对代码的表象进行分析，语义分析则是对表象之外的部分进行分析。举例来说，语义分析包括以下这些处理。

- 区分变量为局部变量还是全局变量
- 解析变量的声明和引用
- 变量和表达式的类型检查
- 检查在引用变量之前是否进行了初始化
- 检查函数是否按照定义返回了结果

上述处理的结果都会反映到抽象语法树中。语法分析生成的语法树只是将代码的构造照搬了过来，而语义分析生成的抽象语法树中还包含了语义信息。例如，在变量的引用和定义之间添加链接，适当地增加类型转换等命令，使表达式的类型一致。另外，语法树中的表达式外侧的括号、行末的分号等，在抽象语法树中都被省略了。

## 生成中间代码

生成抽象语法树后，接着将抽象语法树转化为只在编译器内部使用的**中间代码**（Intermediate Representation，IR）。

之所以特地转化为中间代码，主要是为了支持多种编程语言或者机器语言。

例如，GCC 不仅支持 C 语言，还可以用来编译 C++ 和 Fortran。CPU 方面，不仅是 Intel 的 CPU，还可以生成面向 Alpha、SPARC、MIPS 等各类 CPU 的机器语言。如果要为这些语言和 CPU 的各种组合单独制作编译器，将耗费大量的时间和精力。Intel CPU 用的 C 编译器、Intel CPU 用的 C++ 编译器、Intel CPU 用的 Fortran 编译器、Alpha 用的 C 编译器……要制作的编译器的数量将非常庞大（图 1.4）。

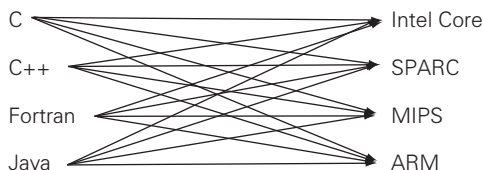


图 1.4 不使用中间代码的情况

而如果将所有的编程语言先转化为共同的中间代码，那么对应一种语言或一种 CPU，只要



添加一份处理就够了（图 1.5）。因此支持多种语言或 CPU 的编译器使用中间代码是比较合适的。例如 GCC 使用的是一种名为 RTL（Register Transfer Language）的中间代码。

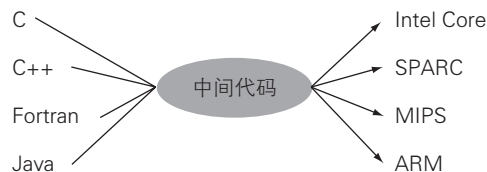


图 1.5 使用中间代码的情况

根据编译器的不同，也存在不经过中间代码，直接从抽象语法树生成机器语言的情况。本书制作的 Cb 编译器最初并没有使用中间代码，后来发现使用中间代码的话，代码的可读性和简洁性都要更胜一筹，所以才决定使用中间代码。

解析代码转化为中间代码为止的这部分内容，称为编译器的**前端**（front-end）。

## 代码生成

最后把中间代码转换为汇编语言，这个阶段称为**代码生成**（code generation）。负责代码生成的程序模块称为**代码生成器**（code generator）。

代码生成的关键在于如何来填补编程语言和汇编语言之间的差异。一般而言，比起编程语言，汇编语言在使用上面的限制要多一些。例如，C 和 Java 可以随心所欲地定义局部变量，而汇编语言中能够分配给局部变量的寄存器只有不到 30 个而已。处理流程控制方面也只有和 goto 语句功能类似的跳转指令。在这样的限制下，还必须以不改变程序的原有语义为前提进行转换。

## 优化

除了之前讲述的 4 个阶段之外，现实的编译器还包括**优化**（optimization）阶段。

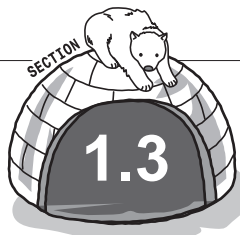
现在的计算机，即便是同样的代码，根据编译器优化性能的不同，运行速度也会有数倍的差距。由于编译器要处理相当多的程序，因此在制作编译器时，最重要的一点就是要尽可能地提高编译出来的程序的性能。

优化可以在编译器的各个环节进行。可以对抽象语法树进行优化，可以对中间代码的代码进行优化，也可以对转换后的机器语言进行优化。进一步来说，不仅是编译器，对链接以及运行时调用的程序库的代码也都可以进行优化。

## 总结

经过上述 4 个阶段，以文本形式编写的代码就被转换为了汇编语言。之后就是汇编器和链接器的工作了。

本书中所制作的编译器主要实现上述 4 个阶段的处理。



## 使用 Cb 编译器进行编译

本节我们来了解一下 Cb 编译器的使用方法。

### Cb 编译器的必要环境

使用 Cb 编译器所需要的软件有如下 3 项。

1. Linux
2. JRE ( Java Runtime Environment ) 1.5 以上版本
3. Java 编译器 ( 非必需 )

首先,要想运行 Cb 编译器 build 的程序,需要运行在 Intel CPU ( 包括 AMD 等的同架构 CPU ) 上的 Linux。这里对 Linux 的发行版本没有特别的要求,大家可以选择喜欢的 Linux 发行版本来安装。本书不对 Linux 的安装方法进行说明。

另外,虽然这里以在 32 位版本的 Linux 上运行为前提,但通过使用兼容模式,64 位的 Linux 也可以运行 32 位的程序<sup>①</sup>。

运行 Cb 编译器需要 JRE ( Java 运行时环境 )。本书不对 JRE 的安装进行说明,请根据所使用的 Linux 发行版本的软件安装方法进行安装。

最后,本书制作的 Cb 编译器是用 Java 实现的。因此 build Cb 编译器本身需要 Java 的编译器。如果只是使用 Cb 编译器的话,则不需要 Java 编译器。

### 安装 Cb 编译器

接着说一下 Cb 编译器的安装方法,在此之前请先安装好 Linux 和 Java 运行环境。

首先下载 Cb 编译器的 jar 文件<sup>②</sup>。

下载的文件是用 tar 和 gzip 打包压缩的,请使用如下命令进行解压。

```
$ tar xzf cbc-1.0.tar.gz
```

① 关于 Linux 的兼容模式,请参考 <http://www.ituring.com.cn/book/1308>。另外,也可以参考 ubuntu 64 位系统下的 cbc 版本: <https://github.com/leungwensen/cbc-ubuntu-64bit> ( 提供 docker 镜像 )。——译者注

② 打开 <http://www.ituring.com.cn/book/1308>, 点击“随书下载”, 下载 Cb 编译器。

解压后会生成名为 `cbc-1.0` 的目录，进入该目录。接着，如下切换到超级用户（`root`），运行 `install.sh`，这样安装就完成了。所有的文件都会被安装到 `/usr/local` 的目录下。

```
$ cd cbc-1.0
$ su
# ./install.sh
```

没有 `root` 权限的用户，也可以安装到自己的 `home` 目录下面。如下运行 `install.sh`，就可以把文件安装到 `$HOME/cbc` 目录下面。

```
$ prefix=$HOME/cbc ./install.sh
```



## Cb 的 Hello, World!

安装完 Cb 的编译器后，让我们来试着 build 一下 Cb 的 Hello,World! 程序吧。Cb 的 Hello,World! 程序如代码清单 1.2 所示。

代码清单 1.2 Cb 的 Hello,World! ( `hello.cb` )

```
import stdio;

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

build 文件时，先进入 `hello.cb` 所在的目录，然后在 shell 中输入如下命令即可。

```
$ cbc hello.cb
```

和 `gcc` 不同的是，`cbc` 不需要输入任何选项，输出的文件名就为 `hello`。因此，只要 `cbc` 命令正常结束，应该就能生成可执行文件 `hello`。确认 `hello` 已经生成后，如下运行该文件。

```
$ ./hello
Hello, World!
```

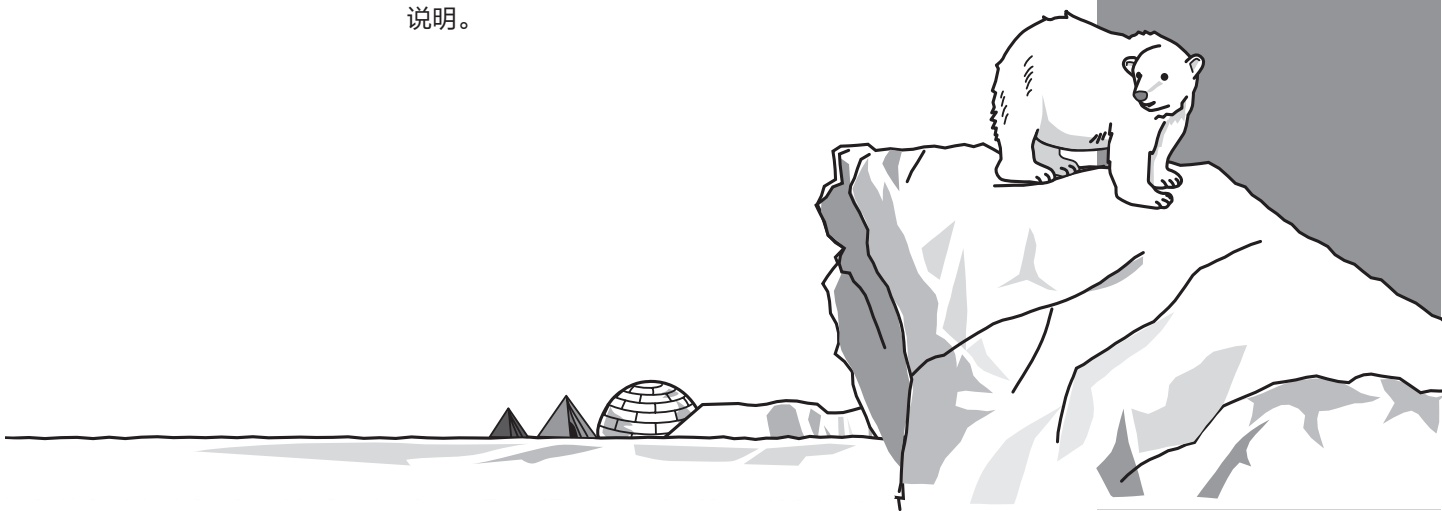
如果像这样显示了 `Hello,World!`，就说明 `cbc` 编译器运行正常。并且上述 `hello` 命令是纯粹的 Linux 原生应用程序，在没有安装 `cbc` 的 Linux 机器上也可以正常运行。

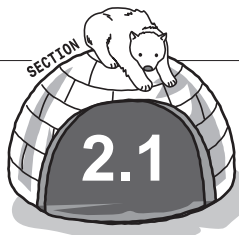
下一章将对 Cb 语言和 `cbc` 进行说明。

# 第2章

## Cb和 cbc

本章将对本书制作的编译器及其实现的概要进行说明。





## Cb语言的概要

本书制作的编译器可将 Cb 这种语言编译为机器语言。本节首先对 Cb 语言的概要进行简单的说明。

### Cb的 Hello, World !

Cb 是 C 语言的简化版，省略了 C 语言中琐碎的部分以及难以实现、容易混淆的功能，实现起来条理更加清晰。虽然如此，Cb 仍保留了包括指针等在内的 C 语言的重要部分。因此，理解了 Cb 的编译过程，也就相当于理解了 C 程序的编译过程。

让我们再来看一下用 Cb 语言编写的 Hello,World! 程序，如代码清单 2.1 所示。

代码清单 2.1 用 Cb语言编写的 Hello,World! 程序

```
import stdio;

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

可见该程序和 C 语言几乎没有差别，不同之处只是用 `import` 替代了 `#include`，仅此而已。

本书的目的是让读者理解“在现有的 OS 上，现有的程序是如何编译及运行的”。那些有着诸多不切实际的限制，仅能作为书中示例的“玩具”语言，对其进行编译丝毫没有意义。从这个角度来说，C 语言作为编程语言是非常具有现实意义的，而 Cb 则十分接近于 C 语言。因此，理解了 Cb，对于现实的程序就会有更深刻的认识。

### Cb中删减的功能

为了使编译器的处理简明扼要，下面这些 C 语言的功能不会出现在 Cb 中。

- 预处理器
- K&R 语法
- 浮点数

- enum
- 结构体（struct）的位域（bit field）
- 结构体和联合体（union）的赋值
- 结构体和联合体的返回值
- 逗号表达式
- const
- volatile
- auto
- register

简单地说一下删除上述功能的原因。

首先，Cb 同 C 语言最大的差异在于 Cb 没有预处理器。认真地制作 C 语言的预处理器会花费过多的时间和精力，进而无法专注于本书的主题——编译器。

但是，因为省略了预处理器，所以 Cb 无法使用 `#define` 和 `#include`。特别是不能使用 `#include`，将无法导入类型定义和函数原型，这是有问题的。为了解决该问题，Cb 使用了与 Java 类似的 `import` 关键字。`import` 关键字的用法将稍后说明。

数据类型方面也做了一些变化。

首先，删除了和浮点数相关的所有功能。浮点数的计算是比较重要的功能，笔者也想对此进行实现，但由于本书页数的限制，最后也只能放弃。

其次，由于 C 语言的 `enum` 和生成名称连续的 `int` 型变量的功能本质上无太大区别，因此为了降低编译器实现的复杂度，这里将其删除。至于结构体和联合体，主要也是考虑到编译器的复杂度，才删除了类似的使用频率不高或非核心的功能。

`volatile` 和 `const` 还是比较常用的，但因为 `cbc` 几乎不进行优化，所以 `volatile` 本身并没有太大意义。`const` 可以有条件地用数字字面量和字符串字面量来实现。

最后，`auto` 和 `register` 不仅使用频率低，而且并非必要，所以将其也删除了。



## import 关键字

下面对 Cb 中新增的 `import` 关键字进行说明。

Cb 在语法上和 C 语言稍有差异，而且没有预处理器，所以不能直接使用 C 语言的头文件。为了能够从外部程序库导入定义，Cb 提供了 `import` 关键字。`import` 的语法如下所示。

```
import 导入文件 ID;
```

下面是具体的示例。

```
import stdio;
import sys.params;
```

导入文件类似于C语言中的头文件，记载了其他程序库中的函数、变量以及类型的定义。cbc中有stdio.hb、stdlib.hb、sys/params.hb等导入文件，当然也可以自己编写导入文件。

导入文件的ID是去掉文件名后的“.hb”，并用“.”取代路径标识中的“\”后得到的。例如导入文件stdio.hb的ID为stdio，导入文件sys/params.hb的ID为sys.params。



## 导入文件的规范

下面让我们看一个导入文件的例子，cbc中的stdio.hb的内容如代码清单2.2所示。

代码清单 2.2 导入文件stdio.hb

```
// stdio.hb

import stddef; // for NULL and size_t
import stdarg;

typedef unsigned long FILE; // dummy

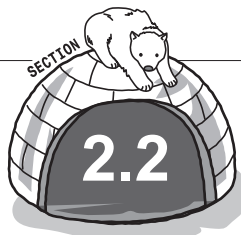
extern FILE* stdin;
extern FILE* stdout;
extern FILE* stderr;

extern FILE* fopen(char* path, char* mode);
extern FILE* fdopen(int fd, char* mode);
extern FILE* freopen(char* path, char* mode, FILE* stream);
extern int fclose(FILE* stream);
:
:
```

只有下面这些声明能够记述在导入文件中。

- 函数声明
- 变量声明（不可包含初始值的定义）
- 常量定义（这里必须有初始值）
- 结构体定义
- 联合体定义
- typedef

函数及变量的声明必须添加关键字extern。并且在Cb中，函数返回值的类型、参数的类型、参数名均不能省略。



## Cb 编译器 cbc 的构成

阅读有一定数量的代码时，首先要做的就是把握代码目录以及文件的构成。这一节将对本书制作的 Cb 编译器 cbc 的代码构成进行说明。

### cbc 的代码树

cbc 采用 Java 标准的目录结构，即将作者的域名倒序，将倒序后的域名作为包（package）名的前缀，按层次排列。比如，笔者的个人主页的域名是 loveruby.net，则包名以 net.loveruby 开头，接着是程序的名称 cflat，其下面排列着 cbc 所用的包。代码的目录结构如图 2.1 所示。

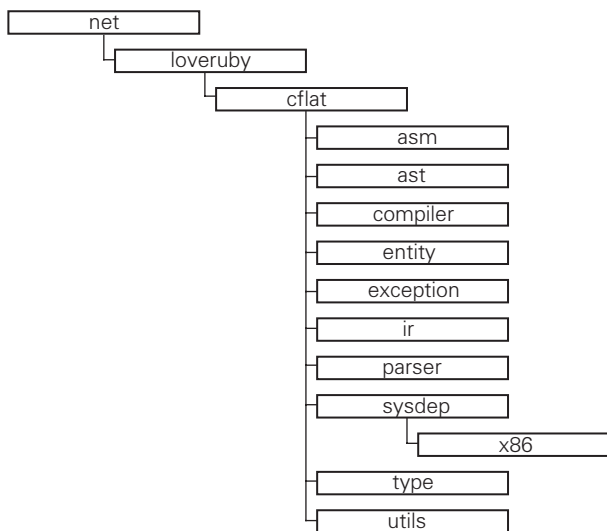


图 2.1 cbc 中包的层次

从 asm 到 utils 的 11 个目录，各自对应着同名的包。也就是说，cbc 有 11 个包，所有 cbc 的类都属于这 11 个包中的某一个。cbc 不直接在 net.loveruby 和 net.loveruby.cflat 下面放置类。



---

提供各种书籍的pd电子版代找服务，如果你找不到自己想要的书的pdf电子版，我们可以帮您找到，如有需要，请联系QQ或微信2028969416.

PDF代找说明：

本人可以帮助你找到你要的PDF电子书，计算机类，文学，艺术，设计，医学，理学，经济，金融，等等。质量都很清晰，而且每本100%都带书签索引和目录，方便读者阅读观看，只要您提供给我书的相关信息，一般我都能找到，如果您有需求，请联系我微信或QQ2028969416。

本人已经帮助了上万人找到了他们需要的PDF，其实网上有很多PDF,大家如果在网上不到的话，可以联系我QQ，大部分我都可以找到，而且每本100%带书签索引目录。因PDF电子书都有版权，请不要随意传播，如果您有经济购买能力，请尽量购买正版。

**声明：本人只提供代找服务，每本100%索引书签和目录，因寻找pdf电子书有一定难度，仅收取代找费用。如因PDF产生的版权纠纷，本人概不负责，我们仅仅只是帮助你寻找到你要的pdf而已。**