从零开始编写 C 语言编译器

作者：左元

时间：May 2, 2021

ElegantLaTeX Program

最好的实践来自理论，最好的理论来自实践。——高德纳

特别声明

本书原著来自 Github 上的 chibicc 项目。

左元
May 2, 2021

目录

第 1 章 本书简介

1.1 介绍

在本书中，我们将创建一个程序，这个程序将 C 语言编写的源代码转换成 x86-64 汇编语言，编译器本身也是使用 C 语言开发的。我们的目标是编译器能够自举，也就是说我们写的编译器能够编译自己本身的源代码。

在本书中，我决定深入研究一下编译器的各个主题，这样解释一个编译器的实现的进度不会太快。原因如下：

编译器的开发在理论上可以分为多个阶段，例如词法分析，语法分析，中间表示和代码生成等等。一般的教科书会对每个主题进行章节介绍和讲解，但采用这种方法写的书往往太深了，所以读者看着看着就放弃了。

而且，每个阶段开发完成以后，都没办法运行编译器来编译写的源程序。因此，如果编译器的设计一开始就有问题，我们也发现不了，因为看不到编译出来的代码。首先，在我完成一个阶段的程序的编写之前，我是无法知道下一个阶段预期的输入是什么，也无法知道上一个阶段的输出是什么。另一个问题是，在写完整个代码之前，无法编译任何代码，所以写代码的动力严重不足。

在本书中，我们将采用增量开发的方式来开发编译器。一开始，我们的编译器只能将简单的整数编译成汇编代码，紧接着，我们的编译器能够将算术表达式编译成汇编代码，后面，我们的编译器可以编译 C 语言的一个子集，到最后，我们的编译器就能够编译完整的 C 语言了。也就是说，在开发的每一步，我们只会添加一个 C 语言中的很小的特性，而不是一下子引入一个大的 C 语言的功能。

我们还会在每个阶段讲解一下所需要的计算机的编程知识，例如数据结构，算法和一些底层的知识等等。

渐进式的开发会让我们全面了解语言的每一个微小的特性是如何开发出来的。这比在书里面去沉浸在每个主题的各种复杂细节中，要好的多。到本书结尾，你将会对每个主题都有比较深入的理解。

本书也是一本讲解如何从头开始编写大型程序的书。编写大型的复杂程序是一种独特的技能，和学习数据结构与算法不太一样。但这方面的书实在是太少了。而只有自己亲自动手开发一个大型的程序，才能真正理解各种开发方式的优缺点。本书就是想通过从零渐进式开发一个完整的 C 语言编译器来让你有对大型程序开发的真实体验。

如果你能够完整的把本书的每个步骤跟下来，那么你将不仅能够获得如何编写编译器的知识，而且还能够学习到 CPU 指令集的知识，还能够了解到如何以渐进的方式来开发大型程序，还能够学习到版本管理的知识，测试程序的编写，以及编写大型程序应该有的心态，这就够了。

在编写本书时，我将尽可能的去解释为什么要这样设计，而不是仅仅罗列一些语言规范和 CPU 指令集的规范。我还希望读者通过阅读本书，能够对编译器、CPU 以及各种计算机的历史感起兴趣来。

实现一个编译器很有意思。最开始，我只想写一个属于自己的小的编程语言，但在继续开发时，我的语言很快就发展成了 C 语言，就像变魔术一样。在实际开发的时候，我的编译器对一些比较复杂的 C 程序都能够编译，而且是正确的。编译输出的汇编代码，我甚至都无法完全理解。编译器好像比我自己都更加智能了。编译器是一种即使你知道它如何工作，你还是会怀疑它为什么能够工作的程序。非常令人着迷。

现在，以此为序，让我们开始编译器的开发吧！

专栏：为什么要编写 C 语言的编译器?
其实也不一定要选择开发 C 语言的编译器。如果想要针对一门语言开发编译器，将这门语言编译成汇编代码，那么 C 语言可能并不是最合适，但也是非常合适了。
如果选择一门脚本语言，那么我们其实无法理解很多底层的原理。而为 C 语言开发编译器的话，由于 C 语言本身和计算机底层非常的接近。所以我们可以学习很多的有关底层的知识，例如 CPU 的指令集以及程序运行的机制。
C 语言的使用非常广泛。以至于编译器写好以后，你就可以编译然后运行网上下载的第三方源代码了。例如你可以编译并运行 Unix xv6 源代码。如果编译器足够成熟，甚至可以编译 Linux 内核的源代码。而开发其他语言的编译器，是享受不了这种待遇的。
C++ 也是一种静态类型的语言，可以编译成类似于 C 语言的代码，并且应用也很广泛。但 C++ 的语言规范极其庞大和复杂，所以写一个自制的 C++ 编译器不太现实。
在提高语言的设计感方面，设计和实现一些原始的语言 (例如 Lisp) 是不错的选择，但同时也有它的缺陷。如果实现起来很麻烦，则可以通过在语言规范中规避掉这些很麻烦的部分。但对于 C 语言来说，却不一样，因为 C 语言是有标准规范的。我们直接实现 C 语言的标准规范就好了。

1.1.1 本书中的符号

我们在要输入 shell 命令时，我的代码由 $ 符号来提示。也就是输入 $ 符号后面的命令 (不要输入 $ 符号)。代码块中还会有输入，例如当我们输入 make 时，有如下表示：

```
$ make
make: Nothing to be done for `all`.
```

1.1.2 开发环境

本书在 Ubuntu 下开发，需要实现安装我们要用到的工具，例如 gcc、git 等：

```
$ sudo apt update
$ sudo apt install -y gcc make git binutils libc6-dev
```

专栏：交叉编译
在 Linux 机器上编译出能够在 Windows 上运行的汇编代码，就是交叉编译器。

1.1.3 有关编译器的自举

可以这样理解。
首先，设计 C 语言的规范。
然后，使用汇编语言来开发一个 C 语言的编译器。
第三，由于我们现在有了 C 语言的编译器了，所以可以使用这个 C 语言编译器来开发其他的 C 语言编译器。这就叫自举。

1.2 机器语言和汇编器

本章的目的是对构成计算机的组件以及应该从我们创建的 C 编译器输出什么样的代码有一个大概的了解。我还不会深入讲解具体的 CPU 指令。首先，了解概念很重要。

1.2.1 CPU 和内存

组成计算机的组件可以大致分为 CPU 和内存。内存是可以保存数据的设备，CPU 是在读取和写入内存时执行某些运算的设备。

从概念上讲，对于 CPU 来说，内存看起来就像是一个由大量可随机访问的字节组成的数组。当 CPU 访问内存时，将以数字方式指定有关要访问的内存字节数的信息，该数字值称为“地址”。例如，“从地址 16 读取 8 个字节的数据”表示从存储器的第 16 个字节开始读取 8 个字节的数据，这看起来像一个字节数组。

CPU 执行的程序以及程序读取和写入的数据都存储在内存中。CPU 将“当前正在执行的指令的地址”保存在 CPU 内，从该地址读取指令，在该地址执行写操作，然后读取并执行下一条指令。当前正在执行的指令的地址称为“程序计数器”（PC）或“指令指针”（IP）。将要由 CPU 执行的程序的形式本身称为“机器语言”（机器代码）。

程序计数器并不总是线性地前进到下一条指令。CPU“分支指令”类型的指令可用于将程序计数器设置为除了后面的指令以外的任何地址。分支指令可以实现 if 语句和循环语句。将程序计数器设置到下一条指令以外的位置称为“跳转”或“分支”。

除了程序计数器外，CPU 还具有少量的数据存储区。例如，Intel 和 AMD 处理器具有 16 个区域，可以容纳 64 位整数。该区域称为“寄存器”。内存是 CPU 的外部设备，需要花费一些时间进行读取和写入，但是寄存器位于 CPU 内部，可以无延迟地进行访问。

许多机器语言都有一种格式，其中两个寄存器的值用于执行某些操作，并将结果写回到寄存器中。因此，在执行程序时，CPU 从存储器中读取数据到寄存器中，在寄存器之间执行一些操作，然后将结果写回到存储器中，以便执行继续进行。

特定机器语言的指令统称为“指令集体系结构”（ISA）或“指令集”。指令集并不是只有一种，而且我们可以根据需要为每个 CPU 设计指令集。但是，指令集没有太多变化，因为没有机器语言兼容性就无法运行同一程序。PC 使用 Intel 及其兼容芯片制造商 AMD 的称为 x86-64 的指令集。x86-64 是主要的指令集之一，但 x86-64 并不是唯一一个主导市场的指令集。例如，iPhone 和 Android 使用称为 ARM 的指令集。

专栏：x86-64 指令集名称

x86-64 有时被称为 AMD64，Intel 64，x64 等。由于历史的原因，这套指令集有很多个名字。

x86 指令集由英特尔在 1978 年创建，但 AMD 将其扩展到 64 位。在 2000 年左右，当需要 64 位处理器时，英特尔正在全公司范围内开发一种全新的名为 Itanium 的指令集，而不敢在与之竞争的 64 位版本的 x86 上工作。借此机会，AMD 制定并发布了 64 位版本 x86 的规范。那是 x86-64。之后，AMD 可能将 x86-64 重命名为 AMD64，这可能是由于其品牌战略。

在那之后，Itanium 的失败就显而易见了，英特尔别无选择，只能制造 64 位版本的 x86，但是到那时，已经有相当数量的实际 AMD64 芯片，所以类似。扩展指令集，英特尔已决定采用与 AMD 兼容的指令集。据说微软有压力要求保持兼容性。当时，英特尔采用了与 AMD64 几乎相同的指令集，名称为 IA-32e。IA-32e（英特尔体系结构 32 扩展）的名称而不是 64，似乎是通过不成功的指令集表明 64 位 CPU 的主要外壳是 Itanium。之后，英特尔决定完全放弃 Itanium，并将 IA-32e 重命名为通常的名称 Intel 64。Microsoft 讨厌太长的名称，所以把 x86-64 指令集也叫做 x64。

由于这些原因，x86-64 具有许多不同的名称。

开源项目通常更喜欢名称 x86-64，因为其中不包含特定公司的名称。在本文档中，我们统一称为 x86-64。

1.2.2 什么是汇编器？

由于机器语言是由 CPU 直接读取的，因此只考虑了 CPU 的便利性，而不考虑人类的操作便利性。用十六进制编辑器编写这些机器语言是一项艰巨的任务。汇编器就是这样发明的，因为写 0101 实在是太难了。汇编语言是一种与机器语言一一对应的语言，但是比机器语言更容易阅读。

对于输出机器语言的二进制可执行文件的编译器（而不是解释器或者虚拟机），输出的代码通常是汇编语言程序。直接输出机器语言的编译器通常也会在输出汇编语言程序后在后台启动汇编器，而汇编器将输出的汇编代码翻译成 0101 这样的机器语言。本书编写的 C 语言编译器输出的是汇编语言程序。

将汇编代码转换为机器语言有时是“编译的”，但有时特别是“汇编的”，以强调输入是汇编语言程序。

读者可能在之前的某个地方看到过汇编器。如果你还没有看到加载器 (loader)，现在是个很好的时机。objdump 使用命令来反汇编可执行文件，并将该文件中包含的机器语言显示为汇编语言。以下对 ls 命令反汇编结果。

```
$ objdump -d /bin/ls | head -20

/bin/ls:     文件格式 elf64-x86-64


Disassembly of section .init:

0000000000004000 <.init>:
    4000:   f3 0f 1e fa             endbr64
    4004:   48 83 ec 08             sub    $0x8,%rsp
    4008:   48 8b 05 c9 ef 01 00    mov    0x1efc9(%rip),%rax        # 22fd8
    ↪  <__gmon_start__>
    400f:   48 85 c0                test   %rax,%rax
```

```
    4012:   74 02                   je     4016 <free@plt-0x6ba>
    4014:   ff d0                   callq  *%rax
    4016:   48 83 c4 08             add    $0x8,%rsp
    401a:   c3                      retq

Disassembly of section .plt:

0000000000004020 <.plt>:
    4020:   ff 35 3a ec 01 00       pushq  0x1ec3a(%rip)        # 22c60
    ↪  <quoting_style_args@@Base+0x260>
```

在我的环境中，该 ls 命令包含约 20,000 种机器语言指令，因此反汇编的结果是一个长的指令，其中包含近 20,000 行。这里仅列出前几个。

每种机器语言的程序集基本上都由一行组成。让我们以下面的代码为例。

```
4004:   48 83 ec 08             sub    $0x8,%rsp
```

这行是什么意思？4004 是包含机器语言的内存地址。换句话说，ls 执行该命令时，此行上的指令位于存储器的地址 0x4004 中，并且当程序计数器为 0x4004 时将执行该指令。接下来的四个十六进制数字是实际的机器语言。CPU 读取该数据并作为指令执行。sub $0x8,%rsp 是与该机器指令相对应的汇编代码。CPU 指令集将在不同的章节中进行说明，但是该指令是从称为 RSP 的寄存器中减去 8 的指令 (如果写成程序的话是：rsp = rsp - 8)。

1.2.3 C 语言及其对应的汇编器

1.2.3.1 一个简单的例子

为了了解 C 编译器生成的内容，让我们将 C 代码与相应的汇编代码进行比较。考虑以下 C 程序为最简单的示例。

```c
int main() {
  return 42;
}
```

给定用于编写该程序的文件，您可以 test1.c 按如下所示对其进行编译，然后看到它 main 实际上返回了 42。

```
$ cc -o test1 test1.c
$ ./test1
$ echo $?
42
```

在 C 语言中，main 函数返回的值就是整个程序的退出码。程序的退出码不显示在屏幕上，而是在 shell 中隐式设置在变量 $? 中，所以在命令行中执行 echo $? 命令，就可以看到退出码。在这里你可以看到 42 正确返回了。

现在，与此 C 程序相对应的汇编程序如下。

```
.globl main
main:
        mov $42,%rax
```

```
        ret
```

在上面的汇编代码中，main 定义了全局标签，后跟 main 函数的代码。整型数值 42 保存在寄存器 RAX 中，然后使用 ret 指令返回。总共有 16 个寄存器可以保存整数，包括 RAX，但是由于可以保证从函数返回时 RAX 中包含的值是函数的返回值，因此这里将其设置为 RAX。

让我们实际编写并运行此汇编程序。由于汇编文件的扩展名为.s，因此请将以上代码写入 test2.s 并执行以下命令。

```
$ cc -o test2 test2.s
$ ./test2
$ echo $?
42
```

与 C 一样，42 现在是退出代码。

粗略地说，C 编译器是一种程序，编译 test1.c 产生的结果就是 test2.s 汇编程序。

1.2.3.2 涉及函数调用的示例

作为一个稍微复杂的示例，让我们看一下将具有函数调用的代码转换为什么样的汇编。

函数调用不仅是跳转，还必须返回到被调用函数完成后最初执行的位置。最初执行的地址称为“返回地址”。如果只有一个函数调用，则返回地址应保存在 CPU 的相应寄存器中，但是由于可以尽可能深地进行函数调用，因此必须将返回地址保存在内存中。返回地址实际上存储在内存中的堆栈中。

只能使用一个保存栈顶地址的变量来实现栈。保持堆栈顶部的存储区域称为“堆栈指针”。x86-64 支持仅堆栈指针寄存器，以及使用这些寄存器来支持使用函数进行编程的指令。将数据堆叠在堆栈上称为“入栈”，而查看压到堆栈上的数据称为“出栈”。

现在让我们看一个函数调用的例子。考虑下面的 C 代码。

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

int main() {
  return plus(3, 4);
}
```

与此 C 代码相对应的汇编代码如下所示：

```
.globl plus, main

plus:
        add %rdi, %rsi
        mov %rsi, %rax
        ret

main:
        mov $3, %rdi
        mov $4, %rsi
```

```
        call plus
        ret
```

.globl 这一行告诉汇编语言，一共有两个函数，plus 和 main 对整个程序可见，而对文件范围不可见。暂时可以忽略这一点。

main 请首先注意它。在 C 语言中，我们 main 从 plus 函数的调用开始。在汇编器中，可以保证第一个参数将在 RDI 寄存器中，第二个参数将在 RSI 寄存器中，因此 main 值的精确设置在的前两行中。

call 那就是调用函数的指令。具体来说，call 执行以下操作：

- call 将下一条指令 ret 的地址（在这种情况下）压入堆栈
- call 跳至作为以下参数的给定地址

因此，call 执行该指令时，CPU 将 plus 开始执行该功能。

plus 注意功能。plus 该功能有三个指令。

add 是要添加的指令。在这种情况下，将 RSI 寄存器和 RDI 寄存器相加的结果写入 RSI 寄存器 (rsi = rsi + rdi)。由于 x86-64 整数算术指令通常仅接收两个寄存器，因此通过覆盖第二个参数的寄存器的值来保存结果。

该函数的返回值应该放在 RAX 中。因此，我们要将加法的结果放入 RAX 中，因此我们需要将值从 RSI 复制到 RAX。我们在这里通过 mov 指令进行操作。mov 是 move 的缩写，但并不是真正的 move，它只是一个复制指令。

plus 在函数的末尾，ret 我们从函数调用并返回。具体来说，它 ret 执行以下操作：

- 从堆栈中弹出一个地址
- 跳转到那个地址

也就是说 ret 和 call 指令撤消我们所做的操作并恢复执行调用函数的指令。以这种方式 call 和 ret 被定义为要被配对的指令。

plusmain 该 ret 命令是从返回的命令。原来的 C 代码应该按原样返回 plus 返回值 main。在这里，plus 里面的 RAX 中的返回值也在 main 函数的可见范围，因此通过按原样 main 返回，可以使它按原样返回值。

1.2.4 本章小结

本章概述了计算机内部的工作方式以及 C 编译器应该做什么。从汇编语言和机器语言的角度看，它看起来像是凌乱的数据块，与 C 语言相去甚远，但实际上，许多读者可能会认为它以一种直接的方式反映了 C 的结构。

objdump 我不知道所示的汇编代码中各个指令的含义，因为我还没有在本书中介绍很多特定的机器语言，但是每个指令都做不到，我想您可以想象。在本章的阶段，足以让人有这种感觉。

本章要点如下。

- CPU 通过读写存储器来推进程序的执行。
- CPU 执行的程序和程序处理的数据都存储在内存中，CPU 从内存中依次读取机器语言指令并执行指令。
- CPU 有一个称为寄存器的小存储区，许多机器语言被定义为寄存器之间的操作。
- 汇编语言是一种使人类更容易阅读机器语言的语言，而 C 编译器通常会输出汇编语言。
- C 函数也可以是汇编语言程序中的函数
- 使用堆栈实现函数调用

专栏：在线编译器

查看 C 代码及其编译结果是学习汇编语言的一种好方法，但是一遍又一遍地编辑和编译源代码以及检查其输出的汇编可能会令人厌烦。有很多很好的网站可以减少这种工作。那就是编译器资源管理器（俗称"godbolt"）。在 Compiler Explorer 的屏幕左半部分的文本框中输入代码，相应的汇编语言代码输出将实时显示在右半部分。如果想查看你的 C 代码将被转换为哪种汇编语言，该站点非常适合你。

第 2 章 创建一个计算器级的语言

在本章中，创建 C 编译器的第一步是支持四个算术运算和其他算术运算符，以便我们可以编译如下表达式：

```
30 + (4 - 2) * -5
```

这似乎是一个相当简单的目标，但这实际上是一个相当困难的目标。公式具有以下结构：括号中的公式优先于加法，乘法优先于加法，如果您不了解它，将无法正确计算。但是，作为输入给出的公式只是一串扁平字符，而不是结构化数据。为了正确对表达式求值，必须分析字符序列并导出其中隐藏的结构。

如果不先学习一些理论知识，这些解析问题将很难解决。实际上，这些问题曾经被认为是困难的，尤其是在 1950 和 1970 年代，当时对它们进行了积极的研究并开发了各种算法。由于有了结果，只要我们知道如何解析，解析就不再是一个难题。

在本章中，它是最常见的解析算法之一“递归下降语法分析器”（递归下降解析）。我们每天使用的 C/C++ 编译器（例如 GCC 和 Clang）也使用递归下降解析。

编程时经常需要读取具有某种结构的文本，而不仅仅是编译器需要读取这样的文本。本章中学习的技术可以直接用于此类问题。在本章中学到的解析技术并不夸张，而是正常的编程技术。阅读本章以了解算法，并以程序员的身份将解析技巧放到你的工具箱中吧。

2.1 步骤 1：创建一种语言来编译一个整数

考虑一下 C 语言的最简单子集。您想象什么语言？main 它是仅具有功能的语言吗？还是只有一种表达的语言？最后，只有一个整数的语言可能是最简单的子集。

在这一步中，我们首先实现最简单的语言。

在此步骤中创建的程序是一个编译器，该编译器从输入中读取一个数字并输出以该数字结尾的程序集作为程序的退出代码。换句话说，输入就像 42 一个字符串，当读取它时，它将创建一个输出以下程序集的编译器。

```
.globl main

main:
        mov 42, %rax
        ret
```

你们在这里可能会认为“类似这样的程序不是编译器”。老实说，我也是这样认为的。但是，该程序接受由一个数字组成的语言作为输入，并输出与该数字相对应的代码，从定义上来说，这是一个很好的编译器。修改后，即使是这样一个简单的程序也可能变得非常复杂，因此让我们首先完成此步骤。

实际上，此步骤对于整个开发过程而言非常重要。这是因为我们将在这一步中所做的工作用作未来开发的框架。在此步骤中，除了创建编译器本身之外，我们还将创建一个构建文件（Makefile），自动测试并设置一个 git 存储库。让我们逐一查看这些任务。

本文档中创建的 C 编译器名为 chibicc。cc 是 C 编译器的缩写。chibi 的意思是 small 的意思。当然，你可以给它取任何喜欢的名字。但是，不要事先考虑太多名称，否则我们将无法开始构建编译器。你可以稍后更改名称，包括 GitHub 存储库，因此可以使用合适的名称开头。

专栏：Intel 表示法和 AT & T 表示法

称为 AT&T 表示法的汇编器表示法在 Unix 上广泛使用。默认情况下，gcc 和 objdump 输出程序集采用 AT&T 表示法。

在 AT&T 表示法中，结果寄存器作为第二个参数。因此，在两个参数的指令中，参数以相反的顺序写入。用% 前缀%rax 写寄存器名称。用 $ 前缀 $42 写数字，依此类推。

同样，在引用内存时，请使用 [] 而不是 () 使用唯一的符号来编写表达式。以下是一些比较示例。

```
mov rbp, rsp   // Intel
mov %rsp, %rbp // AT&T

mov rax, 8     // Intel
mov $8, %rax   // AT&T

mov [rbp + rcx * 4 - 8], rax // Intel
mov %rax, -8(rbp, rcx, 4)    // AT&T
```

对于这次我要制作的编译器，我决定使用 AT&T 表示法会更加方便查看输出。AT&T 表示法和 Intel 表示法的表现力相同。无论使用哪种表示法，生成的机器语言指令序列都是相同的。

2.1.1 创建编译器主体程序

通常，输入是作为文件提供给编译器的，但是由于在此处打开和读取文件很麻烦，因此我们会将数值直接提供给命令行程序的第一个参数。将第一个参数读取为数字并将其嵌入固定字符串组合的 C 程序可以很容易地编写如下。

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

int main(int argc, char **argv) {
  if (argc != 2) {
    fprintf(stderr, "%s: 参数个数不正确\n", argv[0]);
    return 1;
  }

  printf("  .globl main\n");
  printf("main:\n");
  printf("  mov $%d, %%rax\n", atoi(argv[1]));
  printf("  ret\n");
  return 0;
}
```

创建名为 chibicc 的空文件夹，然后将以上代码写入 chibicc.c 文件。之后，如下所示执行 chibicc 并检查操作。

```
$ cc -o chibicc chibicc.c
$ ./chibicc 123 > tmp.s
```

我正在创建 chibicc.c 在第一行中编译 chibicc 的可执行文件。第二行将 123 输入传递给 chibicc 以生成一个汇编代码文件，并将其写入一个名为 tmp.s 的文件中。让我们检查一下 tmp.s 的内容。

```
$ cat tmp.s
.globl main
main:
  mov 123, %rax
  ret
```

如你所见，它生成的代码很好。你可以通过将生成的汇编文件传递给汇编器来创建可执行文件。

在 Unix cc（或 gcc）上，它应该是许多语言的前端，而不仅仅是 C 和 C++，并且应该根据给定文件的扩展名确定语言，然后启动编译器或汇编器。因此，在这里，我们可以通过传递扩展名为.s 的汇编代码文件进行汇编，就像编译 chibicc 时一样 cc。以下是加载和执行生成的可执行文件的示例。

```
$ cc -o tmp tmp.s
$ ./tmp
$ echo $?
123
```

在 shell 程序中，可以使用名为 $? 的变量访问上一个命令的退出码。在上面的示例中，显示的数字 123 与赋予 chibicc 的参数相同。换句话说，它运作良好。给它一个非 123 的数字，范围为 0-255（Unix 的退出代码应该为 0-255），并查看 chibicc 是否真正起作用。

2.1.2 创建自动化测试

许多读者从未在他们的业余编程中编写过测试，但是在本书中，每次扩展编译器时，我们都会编写代码以测试新代码。起初编写测试可能会很麻烦，但是您很快就会发现自己对测试很感激。如果我们不编写测试代码，则每次必须手动运行相同的测试来检查操作，但是手动进行操作要麻烦得多。

我认为编写测试很麻烦的印象来自测试框架的夸大和有时是教条式的测试思想。例如，诸如 JUnit 之类的测试框架具有许多有用的功能，但是部署和学习使用它们可能很棘手。因此，本章将不会介绍这样的测试框架。相反，我将在 Shell 脚本中编写一个非常简单的手写“测试框架”，并使用它来编写测试。

以下是用于测试的 Shell 脚本 test.sh。shell 函数 assert 接受两个参数，即输入值和期望的输出值，并实际汇编 chibicc 结果并将实际结果与期望值进行比较。assert 定义函数后，shell 脚本将使用它来验证 0 和 42 是否正确编译。

```
#!/bin/bash
assert() {
  expected="$1"
  input="$2"

  ./chibicc "$input" > tmp.s
  cc -o tmp tmp.s
  ./tmp
  actual="$?"
```

```
  if [ "$actual" = "$expected" ]; then
    echo "$input => $actual"
  else
    echo "$input => $expected expected, but got $actual"
    exit 1
  fi
}

assert 0 0
assert 42 42

echo OK
```

test.sh 使用以上内容创建 chmod a+x test.sh 并执行以使其可执行。test.sh 让我们实际运行它。如果没有错误发生，则如下所示：test.sh 最后一个 OK 并退出以显示。

```
$ ./test.sh
0 => 0
42 => 42
OK
```

如果发生错误，test.sh 将 OK 不会显示。而是 test.sh 显示失败测试的期望值和实际值，如下所示：

```
$ ./test.sh
0 => 0
42 expected, but got 123
```

如果要调试测试脚本，请-x 使用选项 bash 运行脚本。-x 使用该选项，bash 将显示执行跟踪，如下所示。

```
$ bash -x test.sh
+ assert 0 0
+ expected=0
+ input=0
+ cc -o chibicc chibicc.c
+ ./chibicc 0
+ cc -o tmp tmp.s
+ ./tmp
+ actual=0
+ '[' 0 '!=' 0 ']'
+ assert 42 42
+ expected=42
+ input=42
+ cc -o chibicc chibicc.c
+ ./chibicc 42
+ cc -o tmp tmp.s
```

```
+ ./tmp
+ actual=42
+ '[' 42 '!=' 42 ']'
+ echo OK
OK
```

本书中使用的“测试框架”只是一个类似于上面的 shell 脚本。与完善的测试框架（例如 JUnit）相比，该脚本似乎太简单了，但是此 Shell 脚本的简单性与 chibicc 本身的简单性保持了平衡，所以就这么简单。对于自动化测试，您所要做的就是一次移动代码并机械比较结果，因此重要的是不要太费力地思考并首先进行测试。

2.1.3 用 make 构建

在整本书中，读者将构建数百次甚至数千次 chibicc。每次创建 chibicc 可执行文件然后运行测试脚本的任务都是相同的，因此将其留给工具很方便。make 命令通常用于这些目的。

Makefile 执行该命令后，make 会读取当前目录中一个名为的文件，并执行在该目录中编写的命令。Makefile 由以冒号结尾的规则和该规则的一系列命令组成。以下 Makefile 是使您要在此步骤中执行的命令自动化的操作。

```
CFLAGS=-std=c11 -g -static

chibicc: chibicc.c

test: chibicc
	./test.sh

clean:
	rm -f chibicc *.o *~ tmp*

.PHONY: test clean
```

在文件所在的 chibicc.c 目录中使用 Makefile 文件名创建上述文件。然后，make 您只需 make test 执行即可创建 chibicc，并通过执行来执行测试。由于 make 可以理解文件依赖关系，因此您无需 chibicc.c 在进行更改之后且 make test 在 make 运行之前运行它。仅当 chibicc 可执行文件早于 chibicc.c 时，Make 才会在运行测试之前构建 chibicc。

make clean 这是擦除临时文件的规则。rm 您可能需要手工暂存文件，但是由于它们不方便且意外地擦除了您不想擦除的文件，即使 Makefile 您必须写入这些实用程序也是如此。

请注意，Makefile 中的缩进必须是制表符，这是一个作为编写时的警告。如果有 4 或 8 个空格，将发生错误。这只是一个不方便的语法，但是 make 是 1970 年代开发的一种旧工具，传统上就是这种方式。

确保-static 将选项传递给 cc 命令。此选项在“动态链接”一章中进行了介绍。您现在不必考虑此选项的含义。

2.1.4 用 git 进行版本控制

本书使用 git 作为版本控制系统。在本书中，您将逐步创建一个编译器，但是对于每一步，请执行 git commit 并编写一条提交消息。提交消息可以用中文显示，因此请务必在单行摘要中总结您实际更改的内容。如果要编写一个或多个详细说明，请在第一行之后打开一个空白行，然后再编写说明。

只有您手动生成的文件才使用 git 进行版本控制。通过运行同一命令可以再次生成由于运行 chibicc 而生成的文件，因此无需将它们包含在版本控制中。而是，包括这些文件会不必要地延长每次提交的更改，因此您应将它们从版本控制中删除，而不要将其放在存储库中。

在 git 中，.gitignore 您可以在文件中编写要从版本控制中删除的文件的模式。chibicc.c 在相同的目录中，.gitignore 创建以下内容，并将 git 设置为忽略临时文件，编辑器备份文件等。

```
*~
*.o
tmp*
a.out
chibicc
```

如果您不熟悉 git，请告诉 git 您的姓名和电子邮件地址。您在此处输入 git 的名称和电子邮件地址将记录在提交日志中。以下是设置作者姓名和电子邮件地址的示例。鼓励读者设置他们的姓名和电子邮件地址。

```
$ git config --global user.name "Rui Ueyama"
$ git config --global user.email "ruiu@cs.stanford.edu"
```

为了使用 git 进行提交，您首先 git add 需要使用来添加修改后的文件。由于这是第一次提交 git init，因此请首先创建一个 git 存储库，然后使用来 git add 添加到目前为止已创建的所有文件。

```
$ git init
Initialized empty Git repository in /home/ruiu/chibicc
$ git add chibicc.c test.sh Makefile .gitignore
```

然后 git commit 提交。

```
$ git commit -m " 创建一个编译一个整数的编译器"
```

-m（可选）指定提交消息。-m 如果没有选项 git，则启动编辑器。git log -p 您可以通过执行以下命令来确认提交成功。

```
$ git log -p
commit 0942e68a98a048503eadfee46add3b8b9c7ae8b1 (HEAD -> master)
Author: Rui Ueyama <ruiu@cs.stanford.edu>
Date:   Sat Aug 4 23:12:31 2018 +0000

    创建一个编译一个整数的编译器

diff --git a/chibicc.c b/chibicc.c
new file mode 100644
index 0000000..e6e4599
--- /dev/null
+++ b/chibicc.c
@@ -0,0 +1,16 @@
+#include <stdio.h>
+#include <stdlib.h>
+
```

```
+int main(int argc, char **argv) {
+  if (argc != 2) {
...
```

最后，让我们将到目前为止创建的 git 存储库上载到 GitHub。没有特别的理由上传到 GitHub，但是没有理由不上传到 GitHub，GitHub 也可以用作代码的备份。要上传到 GitHub，请创建一个新的存储库（在本示例中，我们创建了一个 rui314 使用 userchibicc 调用的存储库），并使用以下命令将该存储库添加为远程存储库：

```
$ git remote add origin git@github.com:rui314/chibicc.git
```

然后 git push 运行它以将您的存储库的内容推送到 GitHub。git push 运行后，在浏览器中打开 GitHub 并确保您的源代码已上传。

这样就完成了创建编译器的第一步。此步骤中的编译器是一个很容易调用编译器的程序，但是它是一个很好的程序，其中包含编译器的所有必要元素。从现在开始，我们将继续扩展此编译器，尽管它仍然令人难以置信，但我们会将其发展成为一个好的 C 编译器。首先，请您乐意完成第一步。

指令 mov 将整数移动到 rax 寄存器中，例如 $1 → %*rax*，就是将 1 这个整数移动到了 rax 寄存器中。

然后使用 ret 指令将 main 函数返回。

然后编译 main.c 文件。使用如下命令：

```
$ cc -std=c11 -g -fno-common -c -o main.o main.c
$ cc -o chibicc main.o
```

编译出来的 chibicc 就是可执行程序。然后使用如下命令执行并查看结果：

```
$ ./chibicc 233 > tmp.s
$ gcc -static -o tmp tmp.s
$ ./tmp
$ echo $?
```

然后会发现命令行出现了 1 这个整数，说明我们的编译器是成功的。

为了不每次这样编写命令行命令和编译命令，我们可以写一个 Makefile 和 test.sh 测试脚本来自动化我们的整个过程。

先来写 test.sh 测试脚本

```
#!/bin/bash
assert() {
    expected="$1"
    input="$2"

    ./chibicc "$input" > tmp.s || exit
    gcc -static -o tmp tmp.s
    ./tmp
    actual="$?"

    if [ "$actual" = "$expected" ]; then
        echo "$input => $actual"
    else
```

```
        echo "$input => $expected expected, but got $actual"
        exit 1
    fi
}

assert 0 0
assert 42 42

echo OK
```

上面会检测输入和输出是否相等来检验我们的编译器是否写的正确。
接下来我们写 Makefile 文件。

```
CFLAGS=-std=c11 -g -fno-common

chibicc: main.o
    $(CC) -o chibicc main.o $(LDFLAGS)

test: chibicc
    ./test.sh

clean:
    rm -f chibicc *.o *~ tmp*

.PHONY: test clean
```

第 3 章 添加加减运算符

此时我们的 main.c 变成了下面的样子：

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

int main(int argc, char **argv) {
    if (argc != 2) {
        fprintf(stderr, "%s: 无效的参数个数\n", argv[0]);
        return 1;
    }

    char *p = argv[1]; // p 指针指向输入的字符串，也就是第一个参数

    printf("  .globl main\n");
    printf("main:\n");
    // 现在输入变成了 ld 格式，长整形
    // strtol 方法从 p 指向的指针开始向后寻找
    // 找出一个完整的十进制数值
    printf("  mov $%ld, %%rax\n", strtol(p, &p, 10));

    while (*p) {
        if (*p == '+') {
            p++; // 继续移动 p 指针
            // 继续寻找十进制数值
            // 将找到的数值和 rax 中的十进制数值进行相加
            // 然后保存在 rax 中
            // rax <= rax + num
            printf("  add $%ld, %%rax\n", strtol(p, &p, 10));
            continue;
        }

        if (*p == '-') {
            p++;
            printf("  sub $%ld, %%rax\n", strtol(p, &p, 10));
            continue;
        }

        fprintf(stderr, " 未预期字符: '%c'\n", *p);
        return 1;
    }

    printf("  ret\n");
```

```
    return 0;
}
```

在 test.sh 中添加一条测试语句：

```
assert 42 42
assert 21 '5+20-4'

echo OK
```

此时 main.c 函数变成了如下的样子：

```c
#include <ctype.h>
#include <stdarg.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

// 记号的类型，用枚举来定义
typedef enum {
    TK_PUNCT, // 分隔符，例如加减号就是分隔符
    TK_NUM,   // 数值类型
    TK_EOF,   // 文件结束符
} TokenKind;

// 记号的结构体定义
typedef struct Token Token;
struct Token {
    TokenKind kind; // 记号的类型
    Token *next;    // 下一个记号的指针
    int val;        // 如果是数值类型的话，它的值
    char *loc;      // 记号的位置
    int len;        // 记号的字符串长度
};

static void error(char *fmt, ...) {
    // va_list 用于获取不确定个数的参数
    va_list ap;
    // va_start 对 va_list 变量进行初始化，将 ap 指针指向参数列表中的第一个参数
    va_start(ap, fmt);
    // 从 ap 指针开始，将 fmt 可变参数列表打印出来，打印到标准错误文件描述符
    vfprintf(stderr, fmt, ap);
    fprintf(stderr, "\n");
    exit(1);
}

// 判断记号是否是某个运算符 op
static bool equal(Token *tok, char *op) {
    // 1. 比较存储区 tok->loc 和存储区 op 的前 tok->len 个字节是否相等，
    // 也就是指针从 tok->loc 和从 op 开始的字符串
```

```
    // 2. 判断 op 的最后一个字符是否是'\0' 字符
    // 以上两个条件都必须满足
    return memcmp(tok->loc, op, tok->len) == 0 && op[tok->len] == '\0';
}

// 如果记号的字符串和 s 相等，则跳过
static Token *skip(Token *tok, char *s) {
    if (!equal(tok, s))
        error(" 预期字符串是: '%s'", s);
    return tok->next;
}

// 获取记号中的数值的值
static int get_number(Token *tok) {
    if (tok->kind != TK_NUM)
        error(" 预期是一个数值");
    return tok->val;
}

// 实例化一个记号
static Token *new_token(TokenKind kind, char *start, char *end) {
    Token *tok = calloc(1, sizeof(Token)); // 分配一块大小为 Token 结构体的内存，用
    ↪ 来存储记号
    tok->kind = kind;                      // 记号的类型
    tok->loc = start;                      // 记号的开始指针
    tok->len = end - start;                // 记号的长度
    return tok;                            // 将指针返回
}

// 将输入的字符串分割成一个一个的记号，使用链表的数据结构进行保存
static Token *tokenize(char *p) {
    Token head = {};           // 空结构体
    Token *cur = &head;        // cur 指针指向 head 结构体，或者说 cur 变量中保存了
    ↪ head 结构体的地址

    // 当指针 p 不为空时，一直循环
    while (*p) {
        // 如果 p 指向的是空白字符，跳过
        if (isspace(*p)) {
            p++;
            continue;
        }
```

```
        // 如果 p 指向的是数字，那么实例化一个包含十进制整数的记号结构体
        if (isdigit(*p)) {
            // 先实例化一个数值记号结构体，然后将 cur 移动到下一个记号
            cur = cur->next = new_token(TK_NUM, p, p);
            // q 指向 p 指向的地址
            char *q = p;
            // 从 p 指向的位置开始向后寻找一个无符号长整型数值
            // 然后将 p 指向后面第一个不是数字的位置
            cur->val = strtoul(p, &p, 10);
            // 计算整型记号的长度
            cur->len = p - q;
            continue;
        }

        // 如果 p 指向加减运算符，则实例化一个记号
        // 然后将 cur 指向下一个记号
        if (*p == '+' || *p == '-') {
            cur = cur->next = new_token(TK_PUNCT, p, p + 1);
            p++;
            continue;
        }

        error(" 无效的记号");
    }

    cur = cur->next = new_token(TK_EOF, p, p);
    return head.next;
}

int main(int argc, char **argv) {
    if (argc != 2)
        error("%s: 无效的参数个数", argv[0]);

    // 构建分割出来的记号的链表
    Token *tok = tokenize(argv[1]);

    // 样板代码
    printf(" .globl main\n");
    printf("main:\n");

    // 第一个记号必须是整型数值
    printf("  mov $%d, %%rax\n", get_number(tok));
    // 移动到下一个记号
```

```
        tok = tok->next;

        // 如果记号不是文件结束符，则一直循环
        while (tok->kind != TK_EOF) {
            // 如果记号是加号
            if (equal(tok, "+")) {
                printf("  add $%d, %%rax\n", get_number(tok->next));
                // 向后面移动两个记号
                tok = tok->next->next;
                continue;
            }

            // 如果记号是减号，则跳过记号
            tok = skip(tok, "-");
            printf("  sub $%d, %%rax\n", get_number(tok));
            tok = tok->next; // 向后移动一个记号
        }

        printf("  ret\n");
        return 0;
    }
```

在 test.sh 中添加一条测试语句：

```
    assert 0 0
    assert 42 42
    assert 21 '5+20-4'
    assert 41 ' 12 + 34 - 5 '

    echo OK
```

第 5 章 改进一下错误信息

以下是 main.c 程序：

```c
#include <ctype.h>
#include <stdarg.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef enum {
  TK_PUNCT, // 分隔符，比如加减运算符
  TK_NUM,   // 数值字面量
  TK_EOF,   // 文件结束符
} TokenKind;

// 记号的结构体
typedef struct Token Token;
struct Token {
  TokenKind kind; // 记号的类型
  Token *next;    // 下一个记号的指针
  int val;        // 如果是数值类型记号，则是数值的值
  char *loc;      // 记号的位置
  int len;        // 记号的字符串长度
};

// 输入字符串
static char *current_input;

// 报告错误然后退出
static void error(char *fmt, ...) {
  va_list ap;
  va_start(ap, fmt);
  vfprintf(stderr, fmt, ap);
  fprintf(stderr, "\n");
  exit(1);
}

// 报告错误的位置然后退出程序
static void verror_at(char *loc, char *fmt, va_list ap) {
  // 获取当前位置相对于输入开始指针的相对位置
  int pos = loc - current_input;
```

```
  fprintf(stderr, "%s\n", current_input);
  fprintf(stderr, "%*s", pos, ""); // 打印空白字符，一直打印到 pos 位置
  fprintf(stderr, "^ ");
  vfprintf(stderr, fmt, ap); // 打印错误
  fprintf(stderr, "\n");
  exit(1);
}

static void error_at(char *loc, char *fmt, ...) {
  va_list ap;
  va_start(ap, fmt);
  verror_at(loc, fmt, ap);
}

static void error_tok(Token *tok, char *fmt, ...) {
  va_list ap;
  va_start(ap, fmt);
  verror_at(tok->loc, fmt, ap);
}

// 判断当前记号和字符串 s 是否相等
static bool equal(Token *tok, char *op) {
  return memcmp(tok->loc, op, tok->len) == 0 && op[tok->len] == '\0';
}

// 跳过值为 s 的记号
static Token *skip(Token *tok, char *s) {
  if (!equal(tok, s))
    error_tok(tok, "expected '%s'", s);
  return tok->next;
}

// 返回数值记号中的数值
static int get_number(Token *tok) {
  if (tok->kind != TK_NUM)
    error_tok(tok, "expected a number");
  return tok->val;
}

// 实例化一个新的记号
static Token *new_token(TokenKind kind, char *start, char *end) {
  Token *tok = calloc(1, sizeof(Token));
  tok->kind = kind;
```

```
  tok->loc = start;
  tok->len = end - start;
  return tok;
}

// 对 `current_input` 进行词法分析，然后返回记号链表
static Token *tokenize(void) {
  char *p = current_input;
  Token head = {};
  Token *cur = &head;

  while (*p) {
    // 忽略空白符
    if (isspace(*p)) {
      p++;
      continue;
    }

    // 数值字面量
    if (isdigit(*p)) {
      cur = cur->next = new_token(TK_NUM, p, p);
      char *q = p;
      cur->val = strtoul(p, &p, 10);
      cur->len = p - q;
      continue;
    }

    // Punctuator
    if (*p == '+' || *p == '-') {
      cur = cur->next = new_token(TK_PUNCT, p, p + 1);
      p++;
      continue;
    }

    error_at(p, "invalid token");
  }

  cur = cur->next = new_token(TK_EOF, p, p);
  return head.next;
}

int main(int argc, char **argv) {
  if (argc != 2)
```

```
    error("%s: 无效参数个数", argv[0]);

  current_input = argv[1];
  Token *tok = tokenize();

  printf("  .globl main\n");
  printf("main:\n");

  // 第一个记号必须是一个数值
  printf("  mov $%d, %%rax\n", get_number(tok));
  tok = tok->next;

  // 后面跟着 `+ <number>` 或者 `- <number>`.
  while (tok->kind != TK_EOF) {
    if (equal(tok, "+")) {
      printf("  add $%d, %%rax\n", get_number(tok->next));
      tok = tok->next->next;
      continue;
    }

    tok = skip(tok, "-");
    printf("  sub $%d, %%rax\n", get_number(tok));
    tok = tok->next;
  }

  printf("  ret\n");
  return 0;
}
```