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

1 本书简介 **1**

1.1 介绍 1

1.1.1 本书中的符号 2

1.1.2 开发环境 2

2 第一步：编译整数 **3**

3 添加加减运算符 **5**

4 编写允许输入空白符的词法分析器 **7**

5 改进一下错误信息 **11**

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。

第 2 章 第一步：编译整数

先来编写 main.c 程序。

```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;
  }

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

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;
}
```