大咖助阵丨海纳：C 语言是如何编译执行的？（一）

2022-03-04 海纳

《深入C语言和程序运行原理》 课程介绍 >

讲述：海纳

时长 14:17　大小 13.09M

> 你好，我是于航。这一讲是一期大咖加餐，我们邀请到了海纳老师，来跟你聊聊与 C 程序编译相关的内容。C 语言是一门语法简单，且被广泛使用的编程语言，通过观察其代码的编译流程，你能够清楚地了解一个传统编译器的基本运作原理。海纳老师会用三到四讲的篇幅，来帮助你深刻理解 C 程序的编译全过程，这也是对我们专栏内容的很好补充。感谢海纳老师，也希望你能够有所收获，对 C 语言了解得更加透彻。

你好，我是海纳，是极客时间《编程高手必学的内存知识》的专栏作者。

作为一名编译器开发工程师，在这里我想和你聊一下 C 语言的编译过程。对于 C 语言的开发者来说，深刻理解编译的过程是十分必要的。由于 C 语言非常接近底层，所以它是一门用于构建基础设施的语言。很多时候，C 语言的开发者要理解每一行代码在 CPU 上是如何执行的。所以，有经验的开发者在看到 C 的代码时，基本都能够判断它对应的汇编语句是什么。

在接下来的几篇加餐里，我会通过一个简单的例子，来说明一个 C 编译器有哪些基本步骤。在这个过程中，你也可以进一步通过操作 gcc 的相关工具，来掌握如何查看 C 编译过程的每一步的中间结果。

接下来，我们就先从对 C 编译器基本步骤的整体了解开始吧。

编译的基本步骤

一个 C 语言的源代码文件，一般要经过编译和链接两个大的步骤才能变成可执行程序。其中，编译的过程是将单个 C 源码文件翻译成中间文件。而链接器主要用于符号解析，它负责将中间文件中的符号进行跨文件解析，进而把中间文件组成一个二进制文件。关于链接的知识，于航老师已经在这个专栏的第 27~28 讲中深入地介绍过了，所以在这里我就不赘述了。

我们只聚焦于编译的过程，编译主要可以分为以下几个步骤：

1. 预处理，主要是处理宏定义，将宏定义展开，这一步所使用的技术一般只涉及字符串替换；
2. 词法分析，将文本转成 token 序列；
3. 文法分析，将 token 序列转成抽象语法树；
4. 语义分析，文法只能检查局部的信息，有一些语义信息需要在这一步检查，例如非 void 的函数漏写 return 语句；
5. 平台无关优化，与具体的平台（体系结构）无关的结构优化，往往与语义相关；
6. 平台相关优化，与具体的体系结构相关的优化，例如考虑平台的缓存和流水线设计而做出的优化；
7. 指令选择，调度和寄存器分配，主要是为目标平台生成代码；
8. 中间文件生成，编译过程结束，一个编译单元会生成一个中间文件。

接下来的几节课，我们就按照先后顺序依次介绍编译的每一个步骤。这节课我主要介绍预处理和词法分析。下面来看第一个步骤，预处理。

预处理

预处理最重要的工作步骤是对宏进行处理。宏的概念比较简单，但想要精通却很难，所以大多数时候，我们需要依赖 gcc 的预处理命令对宏进行展开，以观察它的效果。这个命令如下：

```
$ gcc -E file.c
```

接下来，我用一个具体的例子来说明预处理的工作原理，例子代码如下所示：

```
#include <stdio.h>

#ifdef USE_MACRO
#define square(x) (x)*(x)
#else
inline int square(int x) {
    return x * x;
}
#endif

int main() {
    int a = 3;
    int b = square(++a);
    printf("%d\n", b);
    return 0;
}
```

在这个例子中，对 square 的调用（第 13 行）究竟是一个宏，还是一个内联函数调用，取决于“USE_MACRO”这个宏是否被定义。

我们分别使用“gcc -E -DUSE_MACRO”命令和“gcc -E”命令处理这个文件，就会得到不同的结果。先来看定义了“USE_MACRO”的情况：

```
// gcc -E macro_def.c -DUSE_MACRO
int main() {
    int a = 3;
    int b = (++a)*(++a);
    printf("%d\n", b);
    return 0;
}
```

从以上代码中可以看到，square 是一个宏，而这个宏的定义已经消失了，而对 square 的调用也被替换为一个乘法（第 4 行）。

不知道这个结果有没有让你感到吃惊。因为我们的本意是想让变量 a 自增 1，然后再求变量 a 的平方，但从宏展开的结果来看，显然是做了两次自增运算，所以最终的运行结果是 20，而不是 16。从这个例子中，我们也可以看出，宏的替换本质上是字符串替换。也就是说，这里在宏展开的过程中，预处理器仅仅是把字符串“x”简单地替换成了“++a”而已。

如果没有定义“USE_MACRO”，那么预处理的结果就是这样的：

复制代码

```
// gcc -E macro_def.c
inline int square(int x) {
    return x * x;
}

int main() {
    int a = 3;
    int b = square(++a);
    printf("%d\n", b);
    return 0;
}
```

这一次的结果显然就是 16 了。这个结果是比较简单而且直观的，所以我就不再过多解释了，你可以自己动手试验并解释它的执行结果。

这里再留一个小练习：请你自己动手，使用“gcc -E”命令对以下程序进行预处理，并解释预处理的结果，以此来掌握井号和双井号在宏定义中的作用。如果这个过程中遇到什么问题，欢迎在评论区交流讨论。

复制代码

```
#include <stdio.h>

#define TYPE_Apple 1
#define TYPE_Pear  2

struct Fruit {
    int _type;
    char* _name;
};

#define DECLARE(x) \
struct Fruit x = {      \
    TYPE_##x,           \
    #x,                 \
```

```
};

DECLARE(Apple)
DECLARE(Pear)

int main() {
    printf("%d, %s\n", Apple._type, Apple._name);
    printf("%d, %s\n", Pear._type, Pear._name);
    return 0;
}
```

预处理的核心工作就是对宏定义进行展开，展开的时候主要是进行字符串的直接替换，尤其是对于宏函数，不能把它真的当成函数进行处理。在理解了预处理器的工作原理之后，我们再来看下一个步骤，那就是词法分析。

词法分析

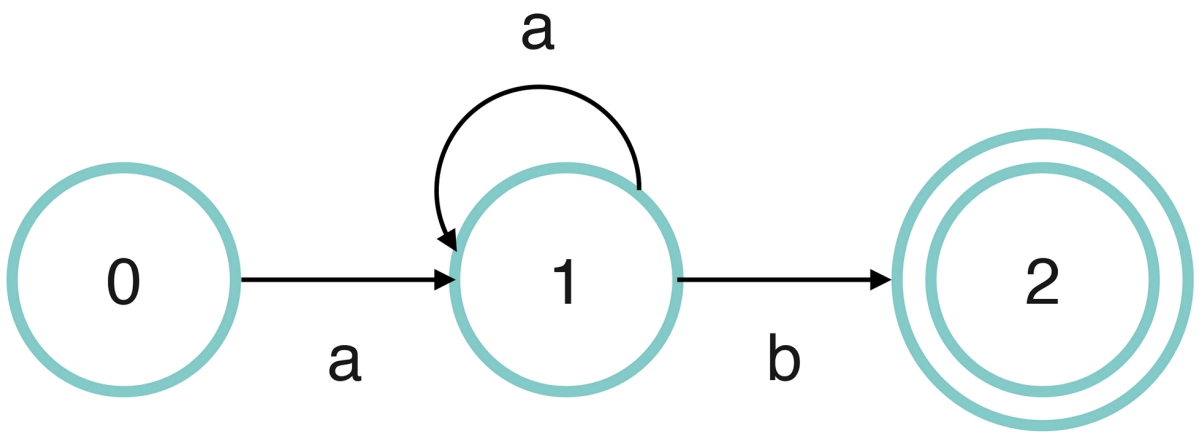
词法分析的作用是把字符进行分组，将有意义关联的字符分到同一个组里，每个组就是一个词。词法分析主要由词法分析器来完成。

例如，“double PI = 3.1415926”这句 C 语言的代码包含了多个字符，但人们在理解它的时候是先把它分成了这四个小组：（“double”，类型声明）、（“PI”，变量名）、（“=”，赋值操作符）、（“3.1415926”，浮点立即数）。

而词法分析的过程和人的理解过程是一致的，也是把字符串进行同样的分组。在编译的过程中，这种分组有一个专门的名称叫做 Token。Token 这个术语在计算机学科中经常出现，在不同的场景中代表不同的含义。例如在网络中，它被翻译成“令牌”，在客户端和服务端通信的场景中，token 又被作为授权验证的加速手段。所以，为了避免歧义，我们这里把 Token 称为词法单元，就代表一个词的意思。

词法分析的主要手段有两种，分别是正则表达式和**有限状态自动机，简称为自动机**。在这一讲中，我主要介绍自动机的方法，这是因为这种方法比较灵活，易于编写和理解。那什么是有限状态自动机呢？

有限状态自动机由一个有限的内部状态集合和一组控制规则组成，这些规则是用来控制在当前状态下可以接受什么输入，以及接受这些输入以后应转向什么状态。例如下图中的有限状态机就包含了 3 个状态：



在这张图片中，状态 0 是自动机的初始状态，从状态 0 出发的箭头号标上了字母 a，表示这个状态可以接受字母 a，进入状态 1 。

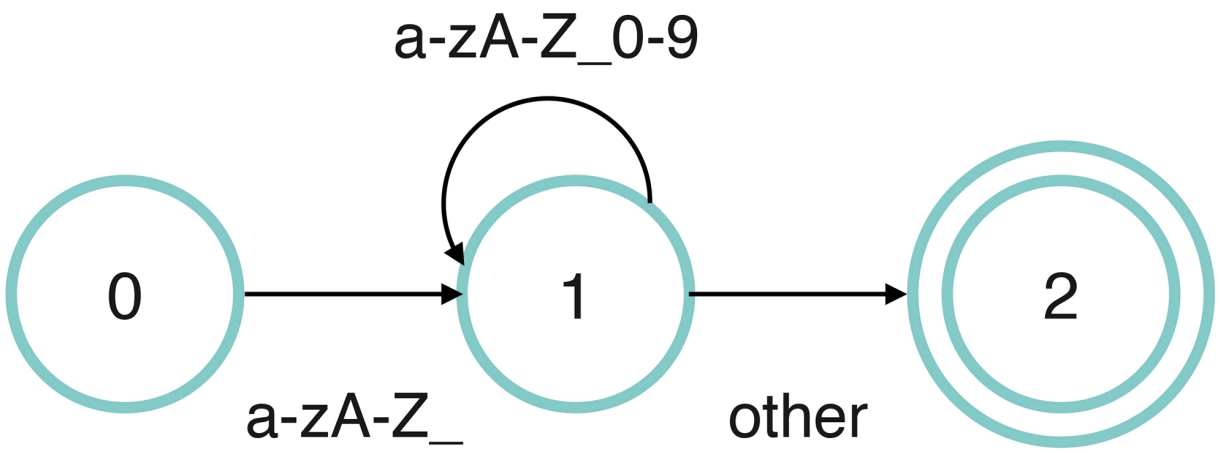
从状态 1 出发的箭头有两条，分别是指回自己的箭头 a，和指向状态 2 的箭头 b。也就是说，状态 1 接受字母 a，仍然回到了状态 1，这就意味着自动机可以在状态 1 的情况下，接受无穷多个字母 a。而箭头 b 则意味着状态 1 还可以接受字母 b，变成状态 2。

状态 2 是比较特殊的一个状态，我们使用两个圈来绘制它，这代表它是一个终态。如果自动机进入到终态以后，就表示自动机完成了一次匹配。

实际上，这个自动机代表了这样一种模式：“a+b”，其中的加号表示，至少包含一个 a，并且以 b 结尾的字符串。例如“aab”、“ab”，都符合“a+b”这个模式。这里你可以自己练习一下，当输入是这两个字符的时候，自动机的状态是如何变化的。

在理解了自动机的概念以后，我们再来看如何通过代码实现一个自动机。我们可以使用一个整型变量 state 来代表自动机的状态，然后根据输入，不断地改变这个变量。

我先举一个处理变量名的例子。C 语言中的变量名可以使用字母和下划线开头，后面可以跟着数字、字母和下划线。所以变量名的正则表达式可以表示为“[a-zA-Z_][a-zA-Z_0-9]*”，其中，中括号表示待匹配的字符范围。这个规则表达成自动机，可以用下面的图片表示：



将这个自动机转换成代码是比较容易的，请你跟着下面的四个步骤来实现它。

第一步，先创建代码所需要的数据结构：


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

enum State {
    STATE_INIT,    /* 有限状态自动机的初始状态 */
    STATE_VAR,     /* 接受字符的状态 */
};

enum TokenType {
    TT_VAR,        /* 标示Token类型是变量 */
    TT_INTEGER,    /* token类型是整数 */
    TT_STRING,     /* token类型是字符串 */
};

union TokenValue {
    char* _str;    /* 这里使用了一个union，即可以用于指向字符串，*/
    int   _int;    /* 也可以是一个整数值。 */
};

struct Token {
    enum TokenType _type;
    union TokenValue _value;
};
```

这段代码中，分别声明了代表自动机状态的枚举值 State（第 5 行），代表 token 类型的枚举值 TokenType（第 10 行），代表 token 的结构体 Token（第 16 行至第 21 行）。这些类型的声明是比较直接的，我就不再多花篇幅一一介绍了，请你阅读代码进行理解，可以参考我加的注释。

第二步，实现创建 token 和销毁 token 的函数：

复制代码

```
1  enum State state = STATE_INIT;
2  char* cur;
3
4  struct Token* create_token(enum TokenType tt, char* begin, char* cur) {
5      struct Token* nt = (struct Token*)malloc(sizeof(struct Token));
6      nt->_type = tt;
7
8      /* 这里只需要对变量进行处理，等号、分号等符号只需要类型就够了。 */
9      if (tt == TT_VAR) {
10         nt->_value._str = (char*)malloc(cur - begin + 1);
11         strncpy(nt->_value._str, begin, cur - begin);
12         nt->_value._str[cur-begin] = 0;
13     }
14
15     return nt;
16 }
17
18 void destroy_token(struct Token* t) {
19     /* 释放空间是和malloc对应的，也在变量的情况下才需要。 */
20     if (t->_type == TT_VAR) {
21         free(t->_value._str);
22         t->_value._str = NULL;
23     }
24
25     free(t);
26 }
```

这段代码先定义了两个全局变量，state 代表自动机的内部状态（第 1 行），指针 cur 代表词法分析器当前的输入字符（第 2 行）。然后又定义了两个辅助函数，分别用于创建 token（第 4 行至第 16 行），以及销毁过期的 token（第 18 行至 26 行）。这里需要注意，create_token 中的 malloc 和 destroy_token 中的 free 是成对出现的，否则就容易造成内存泄漏。

接下来的第三步，我们就可以实现 next_token 函数，用于从字符串中逐个分割单词，每调用一次 next_token 就会得到一个 token：

复制代码

```
1  struct Token* next_token() {
2      state = STATE_INIT;
3      char* begin = 0;
4
5      while (*cur) {
6          char c = *cur;
7          if (state == STATE_INIT) {
8              /* 在初态下，遇到空白字符都可以跳过。 */
9              if (c == ' ' || c == '\n' || c == '\t') {
10                 cur++;
11                 continue;
12             }
13
14             /* 遇到字符则认为是一个变量的开始。 */
15             if ((c <= 'Z' && c >= 'A') ||
16                 (c <= 'z' && c >= 'a') || c == '_') {
17                 begin = cur;
18                 state = STATE_VAR;
19                 cur++;
20             }
21         }
22         else if (state == STATE_VAR) {
23             /* 当前状态机处于分析变量的阶段，所以可以继续接受字母和数字。 */
24             if ((c <= 'Z' && c >= 'A') ||
25                 (c <= 'z' && c >= 'a') ||
26                 (c <= '9' && c >= '0') || c == '_') {
27                 cur++;
28             }
29             else { /* 否则的话，就说明这个变量已经分析完了。 */
30                 return create_token(TT_VAR, begin, cur);
31             }
32         }
33     }
34
35     return NULL;
36 }
```


上述代码就是一个自动机的典型实现：一开始，自动机的状态是 INIT（第 2 行）；在初始状态下，如果遇到空格，制表符和换行符就可以自动忽略（第 8 行至第 12 行）；如果遇到字母，自动机的状态就转换为 STATE_VAR，代表自动机当前正在分析的是一个变量名（第 14 行至第 20 行）。

如果自动机的状态是 VAR，那么当前输入如果是字母或者数字，则状态不变，将字符直接移进即可（第 23 行至第 28 行）；否则就说明当前 token 已经结束了，可以把这个 token 直接返回出去了（第 30 行）。

如果分析到了整个字符串的最后，那控制流就跳出了 while 循环，通过第 35 行返回空值。

最后一步，我们在 main 函数中添加测试程序：

复制代码

```
int main() {
    cur = "int val1 = 1;";

    struct Token* t = next_token();
    printf("%d, %s\n", t->_type, t->_value._str);
    destroy_token(t);

    t = next_token();
    printf("%d, %s\n", t->_type, t->_value._str);
    destroy_token(t);

    return 0;
}
```

编译并执行这个程序，我们就会发现程序正确地打印了前两个 token 的类型和值。

通过这个例子，相信你已经掌握如何使用自动机进行词法分析了，这里我再总体概括一下：我们先根据词法规则写出 token 的正则表达式，然后将正则表达式手绘成由圆圈和箭头组成的图形化的状态机，最后将这个状态机翻译成代码即可。从自动机的图形到代码，这个翻译过程是直接而简明的。

不过，当前的这个词法分析器只能处理变量名，而 main 函数中提供的例子，还要处理等号和整数，以及行尾的分号。接下来，我们就来一起完善它。

完善词法分析器

首先，我们在程序中增加对等号的处理。在 C 语言中，自动机遇到一个等号时，还不知道它是一个赋值操作，还是一个判断相等的操作，而前者是一个等号，后者是两个等号。所以我们只能继续向后看一个字符，把这个过程转化为代码，如下所示：

复制代码

```
struct Token* next_token() {
    state = STATE_INIT;
    char* begin = 0;
    while (*cur) {
        char c = *cur;
        if (state == STATE_INIT) {
            // ....
            else if (c == '=') {
                /* 在初始状态下遇到等号，并不能立即确定这就是一个赋值操作，
                 * 还需要再往后看一个字符。如果后面的字符也是等号，说明这
                 * 是一个"=="操作符。如果不是等号，才说明当前的等号是赋值。*/
                begin = cur;
                state = STATE_EQU;
                cur++;
            }
        }
        //....
        else if (state == STATE_EQU) {
            if (c == '=') { /* "==" 操作符，先不处理 */
            }
            else { /* 赋值操作符 */
                return create_token(TT_ASSIGN, begin, cur);
            }
        }
    }

    return NULL;
}
```

如果只有一个等号，我们就可以判定当前 token 是一个赋值。如果有两个等号，就是“==”操作符，这段程序中先不支持（第 19、20 行），所以这个分支，可以先不实现。这样一来，赋值操作就可以支持了。

接下来，我们再来支持识别整数类型的 token。作为练习，请你自己动手画出整数的自动机，然后再将它转换成代码。因为想鼓励你动手操作，我就不再给出中间步骤了，这里只把最终代码展示给你。你可以将这份代码与自己的代码进行对比，以检查自己的学习效果，也欢迎你在评论区分享自己的操作步骤和实践经验。

复制代码

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

enum State {
```

```
    STATE_INIT,  /* 初始状态 */
    STATE_VAR,   /* 判定变量 */
    STATE_EQU,   /* 判定等号 */
    STATE_NUM,   /* 判定整数 */
};

enum TokenType {
    TT_VAR,      /* token类型为变量 */
    TT_INTEGER,  /* 整数 */
    TT_STRING,   /* 字符串 */
    TT_ASSIGN,   /* 赋值操作符 */
    TT_SEMICON,  /* 行尾分号 */
};

union TokenValue {
    char* _str;
    int   _int;
};

struct Token {
    enum TokenType _type;
    union TokenValue _value;
};

enum State state = STATE_INIT;
char* cur;

/*
 * 创建一个 token。token类型由 tt 指定，它的值由 begin 到 cur 的这一段
 * 字符串决定。如果类型是整型，还要把它的值从字符串转换成整数。
 */
struct Token* create_token(enum TokenType tt, char* begin, char* cur) {
    struct Token* nt = (struct Token*)malloc(sizeof(struct Token));
    nt->_type = tt;

    if (tt == TT_VAR) {
        nt->_value._str = (char*)malloc(cur - begin + 1);
        strncpy(nt->_value._str, begin, cur - begin);
        nt->_value._str[cur-begin] = 0;
    }
    else if (tt == TT_INTEGER) {
        int sum = 0;
        for (char* p = begin; p < cur; p++) {
            sum *= 10;
            sum += (*p - '0');
        }
        nt->_value._int = sum;
    }

    return nt;
}

```

```c
void destroy_token(struct Token* t) {
    if (t->_type == TT_VAR) {
        free(t->_value._str);
        t->_value._str = NULL;
    }

    free(t);
}

void log_token(struct Token* t) {
    printf("%d", t->_type);

    if (t->_type == TT_VAR) {
        printf(", %s\n", t->_value._str);
    }
    else if (t->_type == TT_INTEGER) {
        printf(", %d\n", t->_value._int);
    }
    else {
        printf("\n");
    }
}

char is_alpha(char c) {
    return (c <= 'Z' && c >= 'A') || (c <= 'z' && c >= 'a') || c == '_';
}

char is_num(char c) {
    return c <= '9' && c >= '0';
}

struct Token* next_token() {
    state = STATE_INIT;
    char* begin = 0;

    while (*cur) {
        char c = *cur;
        if (state == STATE_INIT) {
            if (c == ' ' || c == '\n' || c == '\t') {
                cur++;
                continue;
            }

            if (is_alpha(c)) { /* 初始状态下遇到字符 */
                begin = cur;
                state = STATE_VAR;
                cur++;
            }
            else if (is_num(c)) { /* 初始状态下遇到数字 */
                begin = cur;
                state = STATE_NUM;
                cur++;
```

```
            }
            else if (c == '=') { /* 初始状态下遇到等号，需要向后再看一位 */
                begin = cur;
                state = STATE_EQU;
                cur++;
            }
            else if (c == ';') { /* 遇到分号则可以直接返回 */
                begin = cur;
                cur++;
                return create_token(TT_SEMICON, begin, cur);
            }
        }
        else if (state == STATE_VAR) {
            if (is_alpha(c) || is_num(c)) {
                cur++;
            }
            else {
                return create_token(TT_VAR, begin, cur);
            }
        }
        else if (state == STATE_NUM) {
            if (is_num(c)) {
                cur++;
            }
            else {
                return create_token(TT_INTEGER, begin, cur);
            }
        }
        else if (state == STATE_EQU) {
            if (c == '=') { /* "==" 操作符 */
            }
            else { /* 赋值操作符 */
                return create_token(TT_ASSIGN, begin, cur);
            }
        }
    }

    return NULL;
}

int main() {
    cur = "int val1 = 12;";

    struct Token* t = next_token();
    while (t) {
        log_token(t);
        destroy_token(t);
        t = next_token();
    }

    return 0;
```

到这里，我们就对上述代码中第 151 行所展示的那一行 C 代码进行了正确的词法分析。那么下一节课，我们就把注意力放到文法分析上了。

总结

在这节课里，我先介绍了 C 语言编译的基本过程。把一个 C 语言源文件编译成可执行程序，最基本的步骤包括编译和链接两部分。而编译又可以细分为预处理、词法分析、文法分析、语义分析、中间代码生成、平台无关优化、平台相关优化、指令选择与调度、寄存器分配等等。

按照顺序，我在这节课里先展示了预处理和词法分析是如何工作的。

预处理的核心任务是进行宏展开，而宏展开的主要手段就是使用字符串替换，这一点在宏函数定义展开时往往会带来意想不到的问题，所以在使用宏的时候一定要非常慎重。如果自己对宏展开的结果没有把握，我们可以通过“gcc -E”命令来查看预处理的结果。

而词法分析任务主要是把输入的源代码字符串进行合理的分组，将字符串分割成一个个的 token，每一种 token 有自己的类型和值。而词法分析的主要手段有正则表达式和有限状态自动机两种，这节课里我重点介绍了有限自动机的生成方式。

有限自动机是一种包括了状态和转换规则的数据结构，它的状态可以根据输入而不断地发生变化。如果有一种输入序列可以使得自动机从初态转移到终状，我们就称这种输入被自动机所接受。而词法分析的过程，就是不断地令自动机接受输入字符串并且识别 token 的过程。

最后，我通过一个实际的例子，向你展示了词法分析器如何正确地识别变量名和数字，以及赋值操作符、分号等等。

这节课就到这里了，如果今天的内容让你有所收获，欢迎把它分享给你的朋友。下一次的加餐，我将继续讲解 C 语言程序编译的下一个步骤，文法分析。我们到时候见！

上一篇 大咖助阵丨罗剑锋：为什么 NGINX 是 C 编程的经典范本？

下一篇 大咖助阵丨海纳：C 语言是如何编译执行的？（二）

精选留言 (1)

写留言

李慧文
2022-03-14

居然在这里见到了海纳老师，凡代码存在处，皆可学“海”课~太棒了~

2