# daily-code

主要完成C语言的预处理功能

C语言预处理主要针对如下关键字进行特定的动作

主要完成对源文件的扩展，#include #if 递归包含

|  |  |
| --- | --- |
| command | explain |
| #define | 宏定义 |
| #error | 用于调试，输出错误信息 |
| #include | 包含其他文件（递归包含） |
| #if | 后面接表达式 |
| #else |  |
| #elif |  |
| #endif |  |
| #ifdef | 单个宏判断，#ifdef AAA |
| #ifndef |  |
| #undef | 取消宏定义 |
| #line | 改变预先定义的符号，\_\_LINE\_\_ , \_\_FILE\_\_ |
| #pragma | 设定编译器状态 |
| #if defined | 用于多个宏判断#if defined(AAA) && defined(BBB) |
| \_\_LINE\_\_ | 内置宏：代表该行代码所在行号 |
| \_\_FILE\_\_ | 内置宏：代表源文件名 |
| \_\_DATE\_\_ | 内置宏：含有 月/日/年的串，表示源文件翻译到代码时的日期 |
| \_\_TIME\_\_ | 内置宏：时：分：秒，作用同\_\_DATE\_\_ |
| \_\_STDC\_\_ | 如果实现是标准C，则\_\_STDC\_\_含有10进制1 |

词法分析过程:

字符分类：

字母(letter)： a-z, A-Z

数字(digit)： 0-9

空格(ws)： tab space

其他字符(other): 其他非以上三种字符

所有字符(all)： 表示所有字符

有限自动机定义

state，遇见的字符，next\_state

通过逐步定义所有state形成一个状态表，其中START表示起始状态，END表示终止状态，任何非START状态的第一条规则都是all字符，表示一个默认状态，然后对该状态可接受的字符定义。

策略是：当前状态下查看字符，如果非END状态，则”eat”该字符，进入下一个状态，

如果是END状态，则直接返回该字符的内部编码。字符有可能会向前查看来达到最佳匹配，比如 “==”比”=”更匹配（贪婪匹配），所以需要多向前查看一个字符来达到是否已经终结，用END\_FORWARD状态表示遇到字符后，直接”eat”然后返回一个内部编码

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 所有状态 | 基本状态： START , ID1,NUM1,NUM2,NUM3,DOT1,STR1,STR2,STR3,CC1,  CC2,WS1,PLUS1,MINUS1,COM1,COM2,COM3,COM4,STAR1,PERSENT1,  GT1,GT2,LT1,LT2,OR1,AND1,NOT1,CIRC1,ASG1,SHARP1  一行10个状态，共有30个状态  需要进行特别的action  特殊状态： END (token结束)  END\_FORWARD (token需包含当前查看的字符)  S\_EOF(遇见文件结束)  S\_NL(遇见换行符)  STR\_EOF(字符串遇见文件结束)  STR\_NL(字符串遇见换行符)  S\_COMNL(注释遇见换行符)  S\_EOFCOM(注释遇见文件结束)  S\_COMMENT(注释结束)  S\_WS(空格需要跳过)  S\_NAME(可能是宏定义，需要进行宏check)  S\_EOB(需要从文件将字符读入到缓冲区中)  内部编码（简写：D---->double，L---->logical）：  共有编码：54个  NAME,（标识符）  NUMBER,（数字）  ELLIPS,（省略号函数可变参数运算符：…）  STRING,（字符串）  CHAR\_CONST,（字符）  NL,（换行符）  WS,（空格，制表符）  DOT,（点运算符：‘.’）  PLUS,（算数运算符：+）  D\_PLUS,（算数运算符：++）  ASSIGN\_PLUS, （赋值运算符：+=）  MINUS,（算数运算符：-）  D\_MINUS,（算数运算符：--）  ASSIGN\_MINUS,（赋值运算符：-=）  ARRAW,（指针运算符：->）  STAR,（星号：\*）  ASSIGN\_STAR,（赋值运算符：\*=）  SLASH,（除号：/）  ASSIGN\_SHASH,（赋值运算符：/=）  PERSENT,（取余运算符：%）  ASSIGN\_PERSENT,（赋值运算符：%=）  GT,（比较运算符：>）  GEQ,（比较符号：>=）  LT,（比较运算符：<）  LEQ,（比较符号：<=）  RSH,（移位符：>>）  ASSIGN\_RSH,（赋值运算符：>>=）  LSH,（移位符：<<）  ASSIGN\_LSH,（赋值运算符：=<<  OR,（或运算符：|）  ASSIGN\_OR, （赋值运算符：|=）  LOR,（逻辑运算符：LOR）  AND,（按位与：&）  ASSIGN\_AND,（赋值运算符：&=）  LAND,（逻辑运算符：&&）  NOT,（非运算符：!）  NEQ,（比较符号：!=）  TITDE,（位取反：~）  ASSIGN,（赋值运算符：=）  EQ,（比较符号：==）  CIRC,（异或运算符：^）  ASSIGN\_CIRC,（赋值运算符：^=）  SHARP,（#）  D\_SHARP,（##宏运算符）  LEFT\_BRAKET,（左方括号：‘[’）  RIGHT\_BRAKET,（右方括号：‘]’）  LEFT\_PARENTHESIS,（左大括号：‘(’}  RIGHT\_PARENTHESIS,（右大括号：‘)’）  LEFT\_BRACE,（左括号：‘{’）  RIGHT\_BRACE,（右括号：‘}’）  COMMA,（逗号：‘,’）  COLON,（三目冒号运算符：’:’）  SEMIC,（分号：‘;’）  QUEST,（三目问号运算符：?） | | | | |
| 符号含义 | 起始状态 | 遇见字符 | 下一个状态 | END返回token类别 | 解释 |
| ID | START | letter | \_ | ID1 |  |  |
| ID1 | all | END | NAME | 标识符 |
| ID1 | letter| \_ |digit | ID1 |  |  |
| num | START | digit | NUM1 |  |  |
| START | ‘.’ | NUM2 |  | 浮点型 |
| NUM1 | all | END | NUMBER |  |
| NUM1 | digit | letter | ‘.’ | NUM1 |  | 遇见数字，字符，或者小数点儿，还是NUM1，至于letter是否是合法的由编译器判断 |
| NUM1 | ‘E’ | ’e’ | NUM3 |  | 科学记数法 |
| NUM2 | all | END | DOT | 点运算符 |
| NUM2 | digit | NUM1 |  |  |
| NUM2 | . | DOT1 |  | 遇见两个小数点，有可能是 …运算符（函数定义成可变参数的运算符） |
| NUM3 | all | END | NUMBER |  |
| NUM3 | + | - | NUM1 |  | 正负号 |
| NUM3 | digit | letter | NUM1 |  | e后面可以直接接数字 |
| … | DOT1 | all | END |  | 错误符号 |
| DOT1 | . | END\_FORWARD | ELLIPS(省略号) | …运算符(函数可变参数) |
| string | START | “L” | STR1 |  | 宽字符，一个字符占两个字节 |
| START | “ | STR2 |  | 双引号 |
| STR1 | all | END | NAME | 标识符 |
| STR1 | letter | digit | \_ | ID1 |  |  |
| STR1 | “ | STR2 |  | 双引号 |
| STR1 | ‘ | CC1 |  | 单引号 |
| STR2 | all | STR2 |  | 引号内所有字符都属于字符串 |
| STR2 | “ | END | STRING | 遇见结尾双引号 |
| STR2 | \ | STR3 |  | 字符串换行符 |
| STR2 | \n | STR\_NL |  | 字符串不能跨行定义 |
| STR2 | EOF | STR\_EOF |  | 字符串定义错误 |
| STR3 | all | STR2 |  |  |
| STR3 | \n | STR\_NL |  |  |
| STR3 | EOF | STR\_EOF |  |  |
| character const | CC1 | all | CC1 |  | 单引号内的字符 |
| CC1 | ‘ | END\_FORWARD | CHAR\_CONST | 语句结束单引号 |
| CC1 | \ | CC2 |  | 跨行定义 |
| CC1 | \n | STR\_NL |  |  |
| CC1 | EOF | STR\_EOF |  |  |
| CC2 | all | CC1 |  |  |
| CC2 | \n | STR\_NL |  |  |
| CC2 | EOF | STR\_EOF |  |  |
| 空白符 | START | 空格 | WS1 |  |  |
| START | 水平制表符 | WS1 |  |  |
| START | 垂直制表符 | WS1 |  |  |
| WS1 | all | END | WS |  |
| WS1 | \t | \v | ‘ ‘ | WS1 |  |  |
| 算数运算符：  +  -  /  \*  % | START | + | PLUS1 |  | 遇到加号 |
| PLUS1 | all | END | PLUS | 单木运算符 |
| PLUS1 | + | END\_FORWARD | D\_PLUS | ++运算符 |
| PLUS1 | = | END\_FORWARD | ASSIGN\_PLUS | +=运算符 |
| START | - | MINUS1 |  | 遇见减号 |
| MINUS1 | all | END | MINUS | 单木运算 |
| MINUS1 | - | END\_FORWARD | D\_MINUS | -- |
| MINUS1 | = | END\_FORWARD | ASSIGN\_MINUS | -= |
| MINUS1 | > | END\_FORWARD | ARROW | -> 指针运算符 |
| START | / | COM1 |  | 有可能是注释 |
| COM1 | all | END | SLASH | 除号 |
| COM1 | = | END\_FORWARD | ASSIGN\_SLASH | /= 运算符 |
| COM1 | \* | COM2 |  | 多行注释开始 |
| COM1 | / | COM4 |  | 单行注释开始 |
| COM2 | all | COM2 |  | eat up comment |
| COM2 | \* | COM3 |  | 可能是comment 结束 |
| COM2 | \n | S\_COMNL |  | 多行注释 |
| COM2 | EOF | S\_EOFCOM |  | 文件结尾字符 |
| COM3 | all | COM2 |  | 并不是多行结束符 |
| COM3 | / | S\_COMMENT |  | 注释结束位置 |
| COM3 | \n | S\_COMNL |  | 注释换行符 |
| COM3 | \* | COM3 |  | 遇见多个\* |
| COM3 | EOF | S\_EOFCOM |  | comment遇到文件结束 |
| COM4 | all | COM4 |  | 单行注释 |
| COM4 | \n | S\_NL |  |  |
| COM4 | EOF | S\_EOFCOM |  | 注释结束 |
| START | \* | STAR1 |  | 语句\*，可能是乘法也可能是指针运算符 |
| STAR1 | all | END | STAR | 单个\*运算符 |
| STAR1 | = | END\_FORWARD | ASSIGN\_STAR | \*= |
| START | % | PERSENT1 |  | 百分号 |
| PERSENT1 | all | END | PERSENT | % |
| PERSENT1 | = | END\_FORWARD | ASSIGN\_PERSENT | %= |
| 比较运算符：  >  <  |  &  !  ~  =  ^ | START | > | GT1 |  | 遇见大于号 |
| GT1 | all | END | GT | 大于号 |
| GT1 | = | END\_FORWARD | GEQ | >= |
| GT1 | > | GT2 |  | >> 移位运算符 |
| GT2 | all | END | RSH | 右移位运算符 |
| GT2 | = | END\_FORWARD | ASSIGN\_RSH | >>= |
| START | < | LT1 |  | 小于号 |
| LT1 | all | END | LT | < |
| LT1 | = | END\_FORWARD | LEQ | <= |
| LT1 | < | LT2 |  | 左移位运算符 |
| LT2 | all | END | LSH | 左移位运算符 |
| LT2 | = | END\_FORWARD |  | <<= |
| START | | | OR1 |  | 或运算符 |
| OR1 | all | END | OR | | |
| OR1 | = | END\_FORWARD | ASSIGN\_OR | |= |
| OR1 | | | END\_FORWARD | LOR | || 逻辑运算符(logical\_or) |
| START | & | AND1 |  |  |
| AND1 | all | END | AND | & |
| AND1 | = | END\_FORWARD | ASSIGN\_AND | &= |
| AND1 | & | END\_FORWARD | LAND | && logical and |
| START | ! | NOT1 |  | 非运算符 |
| NOT1 | all | END | NOT | ! |
| NOT1 | = | END\_FORWARD | NEQ | != |
| START | ~ | END\_FORWARD | TITDE | 波浪符，位取反运算符 |
| START | = | ASG1 |  | 赋值运算符 |
| ASG1 | all | END | ASGN | assign 赋值运算符 |
| ASG1 | = | END\_FORWARD | EQ | 比较运算符 == |
| START | ^ | CIRC1 |  | 异或运算符 |
| CIRC1 | all | END | CIRC | ^ |
| CIRC1 | = | END\_FORWARD | ASSIGN\_CIRC | ^= |
| 其他符号 | START | # | SHARP1 |  | #号，可构成##运算符，##用于宏拼接 |
|  | SHARP1 | all | END | SHARP | # |
|  | SHARP1 | # | END\_FORWARD | DSHARP | double sharp ## |
| [ | START | [ | END\_FORWARD | LEFT\_BRAKET | 左方括号 |
| ] | START | ] | END\_FORWARD | RIGHT\_BRAKET | 右方括号 |
| ( | START | ( | END\_FORWARD | LEFT\_PARENTHESIS | left parenthesis |
| ) | START | ) | END\_FORWARD | RIGHT\_PARENTHESIS | right parenthesis |
| { | START | { | END\_FORWARD | LEFT\_BRACE | left brace |
| } | START | } | END\_FORWARD | RIGHT\_BRACE | right brace |
| , | START | , | END\_FORWARD | COMMA | , |
| : | START | : | END\_FORWARD | COLON | 冒号运算符 |
| ; | START | ; | END\_FORWARD | SEMIC | 分号 |
| ? | START | ? | END\_FORWARD | QUEST | 问号运算符 |
| \n | START | \n | S\_NL | NL | 换行符 |
| EOF | START | EOF | S\_EOF |  | 文件结束符 |

在解析的时候，需要将token type信息携带上，所以需要将token type和end state进行结合编码，因state总共有41个，可以采用state的低7位存储该信息

ACT(token\_type,state) (token\_type<<7|state)

GETACT(state) state>>7

文件读取采用缓冲区形式，即：从文件读取一定的字节到buffer中，然后再buffer最后设置“buffer结束字符（该字符是非显示字符，不会再源文件中出现）” ，当读到文件末尾后再buffer最后设置“文件结束字符”这样在有限自动机分析的时候会碰到“buffer结束字符”然后读取源文件，这样需要有一个结构体记录文件信息。

#define EOB 0xfe

#define EOFC 0xff

特殊字符，

需要在遇见字符：\ ，?做特别处理，所以状态中需要进行特殊的编码标识这种状态

int state占32位，0~5位存储状态，6位存储flag，7~15存储token type

#define MARKER 0100

1. 当在字符串遇见字符“\”的时候，可能是行折叠，即可能mutil-line string，所以需要进行特别处理
2. 当遇见字符“?”的时候，可能是三字母词

背景：原始的键盘上有如下九个字符无法输入：#，\，^，[，]，{，}，|，~

于是在C语言中允许 ??后面接个特殊字符进行转换，

gcc可以用-trigraphs 来打开三字母词

转换表如下：

|  |  |
| --- | --- |
| ??= | # |
| ??( | [ |
| ??/ | \ |
| ??) | ] |
| ??’ | ^ |
| ??< | { |
| ??! | | |
| ??> | } |
| ??- | ~ |

因为在处理include时候会与多个文件进行交互，因此需要一个结构体保存文件信息，例如：

文件名，当前行等

struct Source{

char \*filename;//文件名

int line; //当前行号

int lineinc; //文件增加行数

char \*in\_buffer;//指向缓冲区

char \*in\_current;//指向下一个待读入的字符

char \*in\_last; //指向缓冲区最后一个字符

FILE \*fd；//文件指针

struct Source \*next//待处理的文件链

}

预处理只管从当前source中读取token，遇见#include的时候需要调整source 链

因为需要将文件内容读到缓冲区中，所以需要在source中设定3个指针，

\*in\_buffer，\*in\_current，\*in\_last，每次都是从当前source中读取字符，所以需要有一个全局变量current\_source指向当前需要读取的文件，每次设定current\_source的时候，插入到source链的头部。

关于buffer的处理：

buffer并不是一次性将所有字节读入，所以在处理buffer的时候需要判断当前指针是否到达in\_last，如果到达需要读取file内容到缓冲区，并将当前缓存区内容往buffer前面移动

因此可以在buffer中设定一个特殊字符EOB，当读取到该字符后进入S\_EOB状态，在该状态下进行special action，也就是说在有限自动机处理的时候，我们可以自己定义一些状态便于特别处理。

token的处理相当于在一个buffer中划分，每个token由两部分组成，token\_start,token\_len，通过该方法能在buffer中定位token，每次get\_token总是获取一行

目前设定的结构只有两个

struct Token{

char \*start;

int len；

int type；//token 类型

struct Token \*next;//形成一个token链

}

struct Token\_row{

int len; //有几个token

struct Token \*t\_last;//最后一个token

struct Token \*t\_current;//当前token

}