C中的预编译宏定义

2008年12月16日 03:39 来源：ChinaUnix博客 作者：knight_123 编辑：周荣茂

作者:

infobillows

在将一个C源程序转换为可执行程序的过程中, 编译预处理是最初的步骤. 这一步骤是由预处理器(preprocessor)来完成的. 在源流程序被编译器处理之前, 预处理器首先对源程序中的"宏(macro)"进行处理.

C

初学者可能对预处理器没什么概念, 这是情有可原的: 一般的C编译器都将预处理, 汇编, 编译, 连接过程集成到一起了.

编译预处理往往在后台运行. 在有的C编译器中, 这些过程统统由一个单独的程序来完成, 编译的不同阶段实现这些不同的功能.

可以指定相应的命令选项来执行这些功能. 有的C编译器使用分别的程序来完成这些步骤. 可单独调用这些程序来完成. 在gcc中,

进行编译预处理的程序被称为CPP, 它的可执行文件名为cpp.

编译预处理命令的语法与C语言的语法是完全独立的. 比如: 你可以将一个宏扩展为与C语法格格不入的内容, 但该内容与后面的语句结合在一个若能生成合法的C语句, 也是可以正确编译的.

(一) 预处理命令简介

预处理命令由#(hash字符)开头, 它独占一行, #之前只能是空白符. 以#开头的语句就是预处理命令, 不以#开头的语句为C中的代码行. 常用的预处理命令如下:

#define 定义一个预处理宏

#undef 取消宏的定义

#include 包含文件命令

#include_next 与#include相似, 但它有着特殊的用途

#if 编译预处理中的条件命令, 相当于C语法中的if语句

#ifdef 判断某个宏是否被定义, 若已定义, 执行随后的语句

#ifndef 与#ifdef相反, 判断某个宏是否未被定义

#elif 若#if, #ifdef, #ifndef或前面的#elif条件不满足, 则执行#elif之后的语句, 相当于C语法中的else-if

#else 与#if, #ifdef, #ifndef对应, 若这些条件不满足, 则执行#else之后的语句, 相当于C语法中的else

#endif #if, #ifdef, #ifndef这些条件命令的结束标志.

defined 与#if, #elif配合使用, 判断某个宏是否被定义

#line 标志该语句所在的行号

将宏参数替代为以参数值为内容的字符窜常量

将两个相邻的标记(token)连接为一个单独的标记

#pragma 说明编译器信息

#warning 显示编译警告信息

#error 显示编译错误信息

(二) 预处理的文法

预处理并不分析整个源代码文件, 它只是将源代码分割成一些标记(token), 识别语句中哪些是C语句, 哪些是预处理语句. 预处理器能够识别C标记, 文件名, 空白符, 文件结尾标志.

预处理语句格式: #command name(...) token(s)

1,

command预处理命令的名称, 它之前以#开头, #之后紧随预处理命令, 标准C允许#两边可以有空白符, 但比较老的编译器可能不允许这样.

若某行中只包含#(以及空白符), 那么在标准C中该行被理解为空白. 整个预处理语句之后只能有空白符或者注释, 不能有其它内容.

2, name代表宏名称, 它可带参数. 参数可以是可变参数列表(C99).

3, 语句中可以利用"\"来换行.

e.g.

define ONE 1 /* ONE == 1 */

等价于: #define ONE 1

#define err(flag, msg) if(flag) \ printf(msg)

等价于: #define err(flag, msg) if(flag) printf(msg)

(三) 预处理命令详述

1, #define

#define命令定义一个宏:

#define MACRO_NAME(args) tokens(opt)

之后出现的MACRO_NAME将被替代为所定义的标记(tokens). 宏可带参数, 而后面的标记也是可选的.

对象宏

不带参数的宏被称为"对象宏(objectlike macro)"

#define经常用来定义常量, 此时的宏名称一般为大写的字符串. 这样利于修改这些常量.

e.g.

#define MAX 100int a[MAX];#ifndef __FILE_H__#define __FILE_H__#include "file.h"#endif

#define __FILE_H__ 中的宏就不带任何参数, 也不扩展为任何标记. 这经常用于包含头文件.

要调用该宏, 只需在代码中指定宏名称, 该宏将被替代为它被定义的内容.

函数宏

带参数的宏也被称为"函数宏". 利用宏可以提高代码的运行效率: 子程序的调用需要压栈出栈, 这一过程如果过于频繁会耗费掉大量的CPU运算资源. 所以一些代码量小但运行频繁的代码如果采用带参数宏来实现会提高代码的运行效率.

函数宏的参数是固定的情况

函数宏的定义采用这样的方式: #define name(args) tokens

其中的args和tokens都是可选的. 它和对象宏定义上的区别在于宏名称之后不带括号.

注意, name之后的左括号(必须紧跟name, 之间不能有空格, 否则这就定义了一个对象宏, 它将被替换为 以(开始的字符串. 但在调用函数宏时, name与(之间可以有空格.

e.g.

#define mul(x,y) ((x)*(y))

注意, 函数宏之后的参数要用括号括起来, 看看这个例子:

e.g.

#define mul(x,y) x*y

"mul(1, 2+2);" 将被扩展为: 1*2 + 2

同样, 整个标记串也应该用括号引用起来:

e.g.

#define mul(x,y) (x)*(y)

sizeof mul(1,2.0) 将被扩展为 sizeof 1 * 2.0

调用函数宏时候, 传递给它的参数可以是函数的返回值, 也可以是任何有意义的语句:

e.g.

mul (f(a,b), g(c,d));

e.g.

#define insert(stmt) stmt

insert (a=1; b=2;) 相当于在代码中加入 a=1; b=2 .

insert (a=1, b=2;) 就有问题了: 预处理器会提示出错: 函数宏的参数个数不匹配. 预处理器把","视为参数间的分隔符.

insert ((a=1, b=2;)) 可解决上述问题.

在定义和调用函数宏时候, 要注意一些问题:

1, 我们经常用{}来引用函数宏被定义的内容, 这就要注意调用这个函数宏时的";"问题.

example_3.7:

#define swap(x,y) { unsigned long _temp=x; x=y; y=_tmp}

如果这样调用它: "swap(1,2);" 将被扩展为: { unsigned long _temp=1; 1=2; 2=_tmp};

明显后面的;是多余的, 我们应该这样调用: swap(1,2)

虽然这样的调用是正确的, 但它和C语法相悖, 可采用下面的方法来处理被{}括起来的内容:

#define swap(x,y) \

do { unsigned long _temp=x; x=y; y=_tmp} while (0)

swap(1,2); 将被替换为:

do { unsigned long _temp=1; 1=2; 2=_tmp} while (0);

在Linux内核源代码中对这种do-while(0)语句有这广泛的应用.

2, 有的函数宏是无法用do-while(0)来实现的, 所以在调用时不能带上";", 最好在调用后添加注释说明.

eg_3.8:

#define incr(v, low, high) \

for ((v) = (low),; (v) 函数宏中的参数包括可变参数列表的情况

C99标准中新增了可变参数列表的内容. 不光是函数, 函数宏中也可以使用可变参数列表.

#define name(args, ...) tokens

#define name(...) tokens

"..."代表可变参数列表, 如果它不是仅有的参数, 那么它只能出现在参数列表的最后. 调用这样的函数宏时, 传递给它的参数个数要不少于参数列表中参数的个数(多余的参数被丢弃).

通过__VA_ARGS__来替换函数宏中的可变参数列表. 注意__VA_ARGS__只能用于函数宏中参数中包含有"..."的情况.

e.g.

#ifdef DEBUG#define my_printf(...) fprintf(stderr, __VA_ARGS__)#else#define my_printf(...) printf(__VA_ARGS__)#endif

tokens中的__VA_ARGS__被替换为函数宏定义中的"..."可变参数列表.

注意在使用#define时候的一些常见错误:

#define MAX = 100

#define MAX 100;

=, ; 的使用要值得注意. 再就是调用函数宏是要注意, 不要多给出";".

注意: 函数宏对参数类型是不敏感的, 你不必考虑将何种数据类型传递给宏. 那么, 如何构建对参数类型敏感的宏呢? 参考本章的第九部分, 关于"##"的介绍.

关于定义宏的另外一些问题

(1) 宏可以被多次定义, 前提是这些定义必须是相同的. 这里的"相同"要求先后定义中空白符出现的位置相同, 但具体的空白符类型或数量可不同, 比如原先的空格可替换为多个其他类型的空白符: 可为tab, 注释...

e.g.

#define NULL 0#define NULL   /* null pointer */ 0

上面的重定义是相同的, 但下面的重定义不同:

#define fun(x) x+1#define fun(x) x + 1 或: #define fun(y) y+1

如果多次定义时, 再次定义的宏内容是不同的, gcc会给出"NAME redefined"警告信息.

应该避免重新定义函数宏, 不管是在预处理命令中还是C语句中, 最好对某个对象只有单一的定义. 在gcc中, 若宏出现了重定义, gcc会给出警告.

(2) 在gcc中, 可在命令行中指定对象宏的定义:

e.g.

$ gcc -Wall -DMAX=100 -o tmp tmp.c

相当于在tmp.c中添加" #define MAX 100".

那么, 如果原先tmp.c中含有MAX宏的定义, 那么再在gcc调用命令中使用-DMAX, 会出现什么情况呢?

---若-DMAX=1, 则正确编译.

---若-DMAX的值被指定为不为1的值, 那么gcc会给出MAX宏被重定义的警告, MAX的值仍为1.

注意: 若在调用gcc的命令行中不显示地给出对象宏的值, 那么gcc赋予该宏默认值(1), 如: -DVAL == -DVAL=1

(3) #define所定义的宏的作用域

宏在定义之后才生效, 若宏定义被#undef取消, 则#undef之后该宏无效. 并且字符串中的宏不会被识别

e.g.

#define ONE 1sum = ONE + TWO /* sum = 1 + TWO */#define TWO 2sum = ONE + TWO /* sum = 1 + 2 */ #undef ONEsum = ONE + TWO /* sum = ONE + 2 */char c[] = "TWO" /* c[] = "TWO", NOT "2"! */

(4) 宏的替换可以是递归的, 所以可以嵌套定义宏.

e.g.

define ONE NUMBER_1# define NUMBER_1 1int a = ONE /* a = 1 */

2, #undef

#undef用来取消宏定义, 它与#define对立:

#undef name

如够被取消的宏实际上没有被#define所定义, 针对它的#undef并不会产生错误.

当一个宏定义被取消后, 可以再度定义它.

3, #if, #elif, #else, #endif

#if, #elif, #else, #endif用于条件编译:

#if 常量表达式1 语句...#elif 常量表达式2 语句...#elif 常量表达式3 语句.....#else 语句...#endif

#if和#else分别相当于C语句中的if, else. 它们根据常量表达式的值来判别是否执行后面的语句. #elif相当于C中的else-if. 使用这些条件编译命令可以方便地实现对源代码内容的控制.

else之后不带常量表达式, 但若包含了常量表达式, gcc只是给出警告信息.

使用它们可以提升代码的可移植性---针对不同的平台使用执行不同的语句. 也经常用于大段代码注释.

e.g.

#if 0{ 一大段代码;}#endif

常量表达式可以是包含宏, 算术运算, 逻辑运算等等的合法C常量表达式, 如果常量表达式为一个未定义的宏, 那么它的值被视为0.

#if MACRO_NON_DEFINED == #if 0

在判断某个宏是否被定义时, 应当避免使用#if, 因为该宏的值可能就是被定义为0. 而应当使用下面介绍的#ifdef或#ifndef.

注意: #if, #elif, #else之后的宏只能是对象宏. 如果name为名的宏未定义, 或者该宏是函数宏. 那么在gcc中使用"-Wundef"选项会显示宏未定义的警告信息.

4, #ifdef, #ifndef, defined.

#ifdef, #ifndef, defined用来测试某个宏是否被定义

#ifdef name 或 #ifndef name

它们经常用于避免头文件的重复引用:

#ifndef __FILE_H__#define __FILE_H__#include "file.h"#endif

defined(name): 若宏被定义,则返回1, 否则返回0.

它与#if, #elif, #else结合使用来判断宏是否被定义, 乍一看好像它显得多余, 因为已经有了#ifdef和#ifndef. defined用于在一条判断语句中声明多个判别条件:

#if defined(VAX) && defined(UNIX) && !defined(DEBUG)

和#if, #elif, #else不同, #indef, #ifndef, defined测试的宏可以是对象宏, 也可以是函数宏. 在gcc中使用"-Wundef"选项不会显示宏未定义的警告信息.

5, #include , #include_next

#include用于文件包含. 在#include 命令所在的行不能含有除注释和空白符之外的其他任何内容.

#include "headfile"#include #include 预处理标记

前面两种形式大家都很熟悉, "#include 预处理标记"中, 预处理标记会被预处理器进行替换, 替换的结果必须符合前两种形式中的某一种.

实

际上, 真正被添加的头文件并不一定就是#include中所指定的文件. #include"headfile"包含的头文件当然是同一个文件,

但#include 包包含的"系统头文件"可能是另外的文件. 但这不值得被注意.

感兴趣的话可以查看宏扩展后到底引入了哪些系统头文件.

关于#include "headfile"和#include 的区别以及如何在gcc中包含头文件的详细信息, 参考本blog的GCC笔记.

相

对于#include, 我们对#include_next不太熟悉. #include_next仅用于特殊的场合.

它被用于头文件中(#include既可用于头文件中, 又可用于.c文件中)来包含其他的头文件. 而且包含头文件的路径比较特殊:

从当前头文件所在目录之后的目录来搜索头文件.

比如: 头文件的搜索路径一次为A,B,C,D,E. #include_next所在的当前头文件位于B目录, 那么#include_next使得预处理器从C,D,E目录来搜索#include_next所指定的头文件.

可参考

cpp手册

进一步了解#include_next

6, 预定义宏

标准C中定义了一些对象宏, 这些宏的名称以"__"开头和结尾, 并且都是大写字符. 这些预定义宏可以被#undef, 也可以被重定义.

下面列出一些标准C中常见的预定义对象宏(其中也包含gcc自己定义的一些预定义宏:

__LINE__ 当前语句所在的行号, 以10进制整数标注.

__FILE__ 当前源文件的文件名, 以字符串常量标注.

__DATE__ 程序被编译的日期, 以"Mmm dd yyyy"格式的字符串标注.

__TIME__ 程序被编译的时间, 以"hh:mm:ss"格式的字符串标注, 该时间由asctime返回.

__STDC__ 如果当前编译器符合ISO标准, 那么该宏的值为1

__STDC_VERSION__ 如果当前编译器符合C89, 那么它被定义为199409L, 如果符合C99, 那么被定义为199901L.

我用gcc, 如果不指定-std=c99, 其他情况都给出__STDC_VERSION__未定义的错误信息, 咋回事呢?

__STDC_HOSTED__ 如果当前系统是"本地系统(hosted)", 那么它被定义为1. 本地系统表示当前系统拥有完整的标准C库.

gcc定义的预定义宏:

__OPTMIZE__ 如果编译过程中使用了优化, 那么该宏被定义为1.

__OPTMIZE_SIZE__ 同上, 但仅在优化是针对代码大小而非速度时才被定义为1.

__VERSION__ 显示所用gcc的版本号.

可参考"GCC the complete reference".

要想看到gcc所定义的所有预定义宏, 可以运行: $ cpp -dM /dev/null

7, #line

#line用来修改__LINE__和__FILE__.

e.g.

printf("line: %d, file: %s\n", __LINE__, __FILE__);#line 100 "haha" printf("line: %d, file: %s\n", __LINE__, __FILE__); printf("line: %d, file: %s\n", __LINE__, __FILE__);

显示:

line: 34, file: 1.cline: 100, file: hahaline: 101, file: haha

8, #pragma, _Pragma

#pragma用编译器用来添加新的预处理功能或者显示一些编译信息. #pragma的格式是各编译器特定的, gcc的如下:

#pragma GCC name token(s)

#pragma之后有两个部分: GCC和特定的pragma name. 下面分别介绍gcc中常用的.

(1) #pragma GCC dependency

dependency测试当前文件(既该语句所在的程序代码)与指定文件(既#pragma语句最后列出的文件)的时间戳. 如果指定文件比当前文件新, 则给出警告信息.

e.g.

在demo.c中给出这样一句:

#pragma GCC dependency "temp-file"

然后在demo.c所在的目录新建一个更新的文件: $ touch temp-file, 编译: $ gcc demo.c 会给出这样的警告信息: warning: current file is older than temp-file

如果当前文件比指定的文件新, 则不给出任何警告信息.

还可以在在#pragma中给添加自定义的警告信息.

e.g.

#pragma GCC dependency "temp-file" "demo.c needs to be updated!"

1.c:27:38: warning: extra tokens at end of #pragma directive1.c:27:38: warning: current file is older than temp-file

注意: 后面新增的警告信息要用""引用起来, 否则gcc将给出警告信息.

(2) #pragma GCC poison token(s)

若源代码中出现了#pragma中给出的token(s), 则编译时显示警告信息. 它一般用于在调用你不想使用的函数时候给出出错信息.

e.g.

#pragma GCC poison scanf

scanf("%d", &a);

warning: extra tokens at end of #pragma directive

error: attempt to use poisoned "scanf"

注意, 如果调用了poison中给出的标记, 那么编译器会给出的是出错信息. 关于第一条警告, 我还不知道怎么避免, 用""将token(s)引用起来也不行.

(3) #pragma GCC system_header

从#pragma GCC system_header直到文件结束之间的代码会被编译器视为系统头文件之中的代码. 系统头文件中的代码往往不能完全遵循C标准, 所以头文件之中的警告信息往往不显示. (除非用 #warning显式指明).

(这条#pragma语句还没发现用什么大的用处

)

由于#pragma不能用于宏扩展, 所以gcc还提供了_Pragma:

e.g.

#define PRAGMA_DEP #pragma GCC dependency "temp-file"

由于预处理之进行一次宏扩展, 采用上面的方法会在编译时引发错误, 要将#pragma语句定义成一个宏扩展, 应该使用下面的_Pragma语句:

#define PRAGMA_DEP _Pragma("GCC dependency \"temp-file\"")

注意, ()中包含的""引用之前引该加上\转义字符.

9, #, ##

#和##用于对字符串的预处理操作, 所以他们也经常用于printf, puts之类的字符串显示函数中.

#用于在宏扩展之后将tokens转换为以tokens为内容的字符串常量.

e.g.

#define TEST(a,b) printf(#a "##用于将它前后的两个token组合在一起转换成以这两个token为内容的字符串常量. 注意##前后必须要有token.

e.g.

#define TYPE(type, n) type n

之后调用:

TYPE(int, a) = 1;

TYPE(long, b) = 1999;

将被替换为:

int a = 1;

long b = 1999;

(10) #warning, #error

#warning, #error分别用于在编译时显示警告和错误信息, 格式如下:

#warning tokens#error tokens

e.g.

#warning "some warning"

注意, #error和#warning后的token要用""引用起来!

(在gcc中, 如果给出了warning, 编译继续进行, 但若给出了error, 则编译停止. 若在命令行中指定了 -Werror, 即使只有警告信息, 也不编译.

- AIX
- HP-UNIX
- Mac OS X

Windows

- Windows7
- Windows Server

其它OS

- Android
- MeeGo

操作系统频道热议话题

- 什么时候需要创建一个nologin的用户?
- 请教一个命令的意思
- 谷歌主页可以ping通不能用浏览器打开
- centos 6.3 使用YUM更新问题

热门博客