道哥的第 019 篇原创

一、前言

一直以来，我都有这样一种感觉：当我学习一个新领域的知识时，如果其中的某个知识点在刚开始接触时，我感觉比较难懂、不好理解，那么以后不论我花多长时间去研究这个知识点，心里会一直认为该知识点比较难，也就是说第一印象特别的重要。

就比如 C 语言中的宏定义，好像跟我犯冲一样，我一直觉得宏定义是 C 语言中最难的部分，就好比有有些小伙伴一直觉得指针是 C 语言中最难的部分一样。

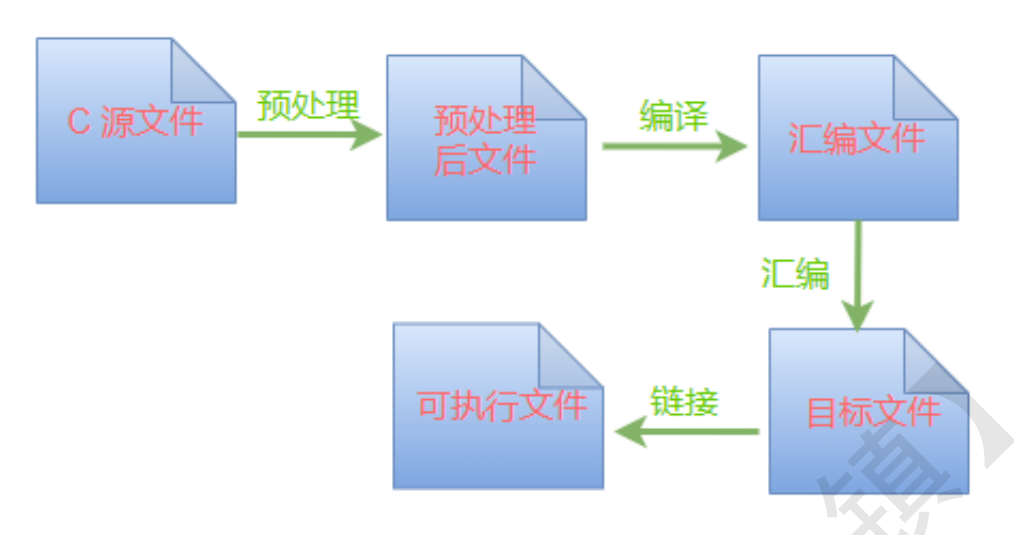
宏的本质就是代码生成器，在预处理器的支持下实现代码的动态生成，具体的操作通过条件编译和宏扩展来实现。我们先在心中建立这么一个基本的概念，然后通过实际的描述和代码来深入的体会：如何驾驭宏定义。

所以，今天我们就来把宏定义所有的知识点进行汇总、深挖，希望经过这篇文章，我能够摆脱心理的这个魔障。看完这篇总结文章后，我相信你也一定能够对宏定义有一个总体、全局的把握。

二、预处理器的操作

1. 宏的生效环节：预处理

一个 C 程序在编译的时候，从源文件开始到最后生成二进制可执行文件，一共经历 4 个阶段：



我们今天讨论的内容就是在第一个环节：预处理，由预处理器来完成这个阶段的工作，包括下面这 4 项工作：

1. 文件引入(#include);
2. 条件编译(#if..#elif..#endif);
3. 宏扩展(macro expansions);
4. 行控制(line control)。

2. 条件编译

一般情况下，C 语言文件中的每一行代码都是要被编译的，但是有时候出于对程序代码优化的考虑，希望只对其中的一部分代码进行编译，此时就需要在程序中加上条件，让编译器只对满足条件的代码进行编译，将不满足条件的代码舍弃，这就是条件编译。

简单的说：就是预处理器根据我们设置的条件，对代码进行动态的处理，把有效的代码输出到一个中间文件，然后送给编译器进行编译。

条件编译基本上在所有的项目代码中都被使用到，例如：当你需要考虑下面的几种情况时，就一定会使用条件编译：

1. 需要把程序编译成不同平台下的可执行程序；
2. 同一套代码需要运行在同一平台上的不同功能产品上；
3. 在程序中存在着一些测试目的的代码，不想污染产品级的代码，需要屏蔽掉。

这里举 3 个例子，在代码中经常看到的关于条件编译：

示例1：用来区分 C 和 C++ 代码

```
#ifdef __cplusplus
extern "C" {
#endif

void hello();

#ifdef __cplusplus
}
#endif
```

这样的代码几乎在每个开源库中都可能见到，主要的目的就是 C 和 C++ 混合编程，具体来说就是：

1. 如果使用 gcc 来编译，那么宏 __cplusplus 将不存在，其中的 extern "C" 将会被忽略；
2. 如果使用 g++ 来编译，那么宏 __cplusplus 就存在，其中的 extern "C" 就发生作用，编译出来的函数名 hello 就不会被 g++ 编译器改写，因此就可以被 C 代码来调用；

示例2：用来区分不同的平台

```
#if defined(linux) || defined(__linux) || defined(__linux__)
    sleep(1000 * 1000); // 调用 Linux 平台下的库函数
#elif defined(WIN32) || defined(_WIN32)
    Sleep(1000 * 1000); // 调用 Windows 平台下的库函数(第一个字母是大写)
#endif
```

那么，这些 linux, __linux, __linux__, WIN32, _WIN32 是从哪里来的呢？我们可以认为是编译目标平台(操作系统)为我们预先准备好的。

示例3：在编写 Windows 平台下的动态库时，声明导出和导入函数

```
#if defined(linux) || defined(__linux) || defined(__linux__)
    #define LIBA_API
#else
  #ifdef LIBA_STATIC
    #define LIBA_API
  #else
      #ifdef LIBA_API_EXPORTS
          #define LIBA_API __declspec(dllexport)
      #else
          #define LIBA_API __declspec(dllimport)
      #endif
  #endif
#endif

LIBA_API void hello();
```

这段代码是直接从我之前在 B 站录制的一个小视频里的示例拿过来的，当时主要是演示如何如何在 Linux 平台下使用 make 和 cmake 构建工具来编译，后来又小伙伴让我在 Windows 平台下也用 make 和 cmake 来构建，所以就写了上面这段宏定义。

1. 在使用 MSVC 编译动态库时，需要在编译选项(Makefle 或者 CMakeLists.txt)中定义宏

LIBA_API_EXPORTS，那么导出函数 hello 的最前面的宏 LIBA_API 就会被替换成：__declspec(dllexport)，表示导出操作；
2. 在编译应用程序的时候，使用动态库，需要 include 动态库的头文件，此时在编译选项中不需要定义宏 LIBA_API_EXPORTS，那么 hello 函数最前面的 LIBA_API 就会被替换成 __declspec(dllimport)，表示导入操作；
3. 补充一点：如果使用静态库，编译选项中不需要任何宏定义，那么宏 LIBA_API 就为空。

3. 平台预定义的宏

上面已经看到了，目标平台会为我们预先定义好一些宏，方便我们在程序中使用。除了上面的操作系统相关宏，还有另一类宏定义，在日志系统中被广泛的使用：

FILE：当前源代码文件名；
LINE：当前源代码的行号；
FUNCTION：当前执行的函数名；
DATE： 编译日期；
TIME： 编译时间；

例如：

```
printf("file name: %s, function name = %s, current line:%d \n", __FILE__, __FUNCTION__, __LINE__);
```

三、宏扩展

所谓的宏扩展就是代码替换，这部分内容也是我想表达的主要内容。宏扩展最大的好处有如下几点：

1. 减少重复的代码；
2. 完成一些通过 C 语法无法实现的功能(字符串拼接)；
3. 动态定义数据类型，实现类似 C++ 中模板的功能；
4. 程序更容易理解、修改(例如：数字、字符串常亮)；

我们在写代码的时候，所有使用宏名称的地方，都可以理解为一个占位符。在编译程序的预处理环节，这些宏名将会被替换成宏定义中的那些代码段，注意：仅仅是单纯的文本替换。

1. 最常见的宏

为了方便后面的描述，先来看几个常见的宏定义：

(1) 数据类型的定义

```
#ifndef BOOL
    typedef char BOOL;
#endif

#ifndef TRUE
    #define TRUE
#endif

#ifndef FALSE
    #define FALSE
#endif
```

在数据类型定义中，需要注意的一点是：如果你的程序需要用不同平台下的编译器来编译，那么你要去查一下所使用的编译器对这些宏定义控制的数据类型是否已经定义了。例如：在 gcc 中没有 BOOL 类型，但是在 MSVC 中，把 BOOL 类型定义为 int 型。

(2) 获取最大、最小值

```
#define MAX(a, b)    (((a) > (b)) ? (a) : (b))
#define MIN(a, b)    (((a) < (b)) ? (a) : (b))
```

(3) 计算数组中的元素个数

```
#define ARRAY_SIZE(x)    (sizeof(x) / sizeof((x)[0]))
```

(4) 位操作

```
#define BIT_MASK(x)          (1 << (x))
#define BIT_GET(x, y)        (((x) >> (y)) & 0x01u)
#define BIT_SET(x, y)        ((x) | (1 << (y)))
#define BIT_CLR(x, y)        ((x) & (~(1 << (y))))
#define BIT_INVERT(x, y)     ((x) ^ (1 << (y)))
```

2. 与函数的区别

从上面这几个宏来看，所有的这些操作都可以通过函数来实现，那么他们各有什么优缺点呢?

通过函数来实现:

1. 形参的类型需要确定，调用时对参数进行检查;
2. 调用函数时需要额外的开销：操作函数栈中的形参、返回值等;

通过宏来实现:

1. 不需要检查参数，更灵活的传参;
2. 直接对宏进行代码扩展，执行时不需要函数调用;
3. 如果同一个宏在多处调用，会增加代码体积;

还是举一个例子来说明比较好，就拿上面的比较大小来说吧:

(1) 使用宏来实现

```
#define MAX(a, b)    (((a) > (b)) ? (a) : (b))

int main()
{
    printf("max: %d \n", MAX(1, 2));
}
```

(2) 使用函数来实现

```
int max(int a, int b)
{
    if (a > b)
        return a;
    return b;
}

int main()
{
    printf("max: %d \n", max(1, 2));
}
```

除了函数调用的开销，其它看起来没有差别。这里比较的是 2 个整型数据，那么如果还需要比较 2 个浮点型数据呢?

1. 使用宏来调用：MAX(1.1， 2.2)；一切 OK;
2. 使用函数调用：max(1.1, 2.2); 编译报错：类型不匹配。

此时，使用宏来实现的优势就体现出来了：因为宏中没有类型的概念，调用者传入任何数据类型都可以，然后在后面的比较操作中，大于或小于操作都是利用了 C 语言本身的语法来执行。

如果使用函数来实现，那么就必须再定义一个用来操作浮点型的函数，以后还有可能比较：char 型、long 型数据等等。

在 C++ 中，这样的操作可以通过参数模板来实现，所谓的模板也是一种代码动态生成机制。当定义了一个函数模板后，根据调用者的实参，来动态产生多个函数。例如定义下面这个函数模板：

```
template<typename T> T max(T a, T b){
    if (a > b)
        return a;
    return b;
}

max(1, 2);      // 实参是整型
max(1.1, 2,2); // 实参是浮点型
```

当编译器看到 `max(1, 2)` 时，就会动态生成一个函数 `int max(int a, int b) { ... };`

当编译器看到 `max(1.1, 2.2)` 时，又会动态生成另一个函数 `float max(float a, float b) { ... }`。

所以，从代码的动态生成角度看，宏定义和 C++ 中的模板参数有点神似，只不过宏定义仅仅是代码扩展而已。

下面这个例子也比较不错，利用宏的类型无关，来动态生成结构体：

```
#define VEC(T)          \
    struct vector_##T { \
        T *data;        \
        size_t size;    \
    };

int main()
{
    VEC(int)   vec_1 = { .data = NULL, .size = 0 };
    VEC(float) vec_2 = { .data = NULL, .size = 0 };
}
```

这个例子中用到了 ##，下面会解释这个知识点。在前面的例子中，宏的参数传递的都是一些变量，而这里传递的宏参数是数据类型，通过宏的类型无关性，达到了“动态”创建结构体的目的：

```
struct vector_int {
    int *data;
    size_t size;
}

struct vector_float {
    float *data;
    size_t size;
}
```

这里有一个陷阱需要注意：传递的数据类型中不能有空格，如果这样使用：`VEC(long long)`，那替换之后得到：

```
struct vector_long long {  // 语法错误
    long long *data;
    size_t size;
}
```

四、符号：# 与

这两个符号在编程中的作用也是非常巧妙，夸张的说一句：在任何框架性代码中，都能见到它们的身影！

作用如下：

1. #：把参数转换成字符串；
2. ##：连接参数。

1. #：字符串化

直接看最简单的例子：

```
#define STR(x) #x

printf("string of 123: %s \n", STR(123));
```

传入的是一个数字 123，输出的结果是字符串 “123”，这就是字符串化。

2. ##： 参数连接

把宏中的参数按照字符进行拼接，从而得到一个新的标识符，例如：

```
#define MAKE_VAR(name, no) name##no

int main(void)
{
    int MAKE_VAR(a, 1) = 1;
    int MAKE_VAR(b, 2) = 2;

    printf("a1 = %d \n", a1);
    printf("b2 = %d \n", b2);
    return 0;
}
```

当调用宏 MAKE_VAR(a, 1) 后，符号 ## 把两侧的 name 和 no 首先替换为 a 和 1，然后连接得到 a1。然后在调用语句中前面的 int 数据类型就说明了 a1 是一个整型数据，最后初始化为 1。

五、可变参数的处理

1. 参数名的定义和使用

宏定义的参数个数可以是不确定的，就像调用 printf 打印函数一样，在定义的时候，可以使用三个点(...)来表示可变参数，也可以在三个点的前面加上可变参数的名称。

如果使用三个点(...)来接收可变参数，那么在使用的时候就需要使用 **VA_ARGS** 来表示可变参数，如下：

```
#define debug1(...)      printf(__VA_ARGS__)

debug1("this is debug1: %d \n", 1);
```

如果在三个点(...)的前面加上了一个参数名，那么在使用时就一定要使用这个参数名，而不能使用 **VA_ARGS** 来表示可变参数，如下：

```
#define debug2(args...)  printf(args)

debug1("this is debug2: %d \n", 2);
```

2. 可变参数个数为零的处理

看一下这个宏：

```
#define debug3(format, ...)      printf(format, __VA_ARGS__)

debug3("this is debug4: %d \n", 4);
```

编译、执行都没有问题。但是如果这样来使用宏：

```
debug3("hello \n");
```

编译的时候，会出现错误：error: expected expression before ‘)’ token。为什么呢？

看一下宏扩展之后的代码（__VA_ARGS__为空）：

```
printf("hello \n",);
```

看出问题了吧？在格式化字符串的后面多了一个逗号！为了解决问题，预处理器给我们提供了一个方法：通过 ## 符号把这个多余的逗号给自动删掉。于是宏定义改成下面这样就没有问题了。

```
#define debug3(format, ...)     printf(format, ##__VA_ARGS__)
```

类似的，如果自己定义了可变参数的名字，也在前面加上 ##，如下：

```
#define debug4(format, args...)  printf(format, ##args)
```

六、奇思妙想的宏

宏扩展的本质就是文本替换，但是一旦加上可变参数(__VA_ARGS__)和 ## 的连接功能，就能够变化出无穷的想象力。

我一直坚信，模仿是成为高手的第一步，只有见多识广、多看、多学习别人是怎么来使用宏的，然后拿来为己所用，按照“先僵化-再优化-最后固化”这个步骤来训练，总有一天你也能成为高手。

这里我们就来看几个利用宏定义的巧妙实现。

1. 日志功能

在代码中添加日志功能，几乎是每个产品的标配了，一般见到最普遍的是下面这样的用法：

```
#ifdef DEBUG
    #define LOG(...) printf(__VA_ARGS__)
#else
    #define LOG(...)
#endif

int main()
{
    LOG("name = %s, age = %d \n", "zhangsan", 20);
    return 0;
}
```

在编译的时候，如果需要输出日志功能就传入宏定义 DEBUG，这样就能打印输出调试信息，当然实际的产品中需要写入到文件中。如果不需要打印语句，通过把打印日志信息那条语句定义为空语句来达到目的。

换个思路，我们还可以通过条件判断语句来控制打印信息，如下：

```
#ifdef DEBUG
    #define debug if(1)
#else
     #define debug if(0)
#endif

int main()
{
    debug {
        printf("name = %s, age = %d \n", "zhangsan", 20);
    }
    return 0;
}
```

这样控制日志信息的看到的不多，但是也能达到目的，放在这里只是给大家开阔一下思路。

2. 利用宏来迭代每个参数

```
#define first(x, ...) #x
#define rest(x, ...)  #__VA_ARGS__

#define destructive(...)                                \
    do {                                                \
        printf("first is: %s\n", first(__VA_ARGS__)); \
        printf("rest are: %s\n", rest(__VA_ARGS__));  \
    } while (0)

int main(void)
{
    destructive(1, 2, 3);
    return 0;
}
```

主要的思想就是：每次把可变参数 VA_ARGS 中的第一个参数给分离出来，然后把后面的参数再递归处理，这样就可以分离出每一个参数了。我记得侯杰老师在 C++ 的视屏中，利用可变参数模板这个语法，也实现了类似的功能。

刚才在有道笔记中居然找到了侯杰老师演示的代码，熟悉 C++ 的小伙伴可以研究下下面这段代码：

```
// 递归的最后一次调用
void myprint()
{
}

template <typename T, typename... Types>
void myprint(const T &first, const Types&... args)
{
    std::cout << first << std::endl;
    std::cout << "remain args size = " << sizeof...(args) << std::endl;

    // 把其他参数递归调用
  myprint(args...);
}

int main()
{
    myprint("aaa", 7.5, 100);
    return 0;
}
```

3. 动态的调用不同的函数

```
// 普通的枚举类型
enum {
  ERR_One,
  ERR_Two,
  ERR_Three
};

// 利用 ## 的拼接功能，动态产生 case 中的比较值，以及函数名。
#define TEST(no) \
    case ERR_##no: \
      Func_##no(); \
      break;

void Func_One()
{
    printf("this is Func_One \n");
}

void Func_Two()
{
    printf("this is Func_Two \n");
}
```

```
void Func_Three()
{
    printf("this is Func_Three \n");
}

int main()
{
    int c = ERR_Two;
    switch (c) {
        TEST(One);
        TEST(Two);
        TEST(Three);
    };

    return 0;
}
```

在这个例子中，核心在于 TEST 宏定义，通过 ## 拼接功能，构造出 case 分支的比较目标，然后动态拼接得到对应的函数，最后调用这个函数。

4. 动态创建错误编码与对应的错误字符串

这也是一个非常巧妙的例子，利用了 #(字符串化) 和 ##(拼接) 这 2 个功能来动态生成错误编码码和相应的错误字符串：

```
#define MY_ERRORS     \
    E(TOO_SMALL)      \
    E(TOO_BIG)        \
    E(INVALID_VARS)

#define E(e) Error_## e,
typedef enum {
    MY_ERRORS
} MyEnums;
#undef E

#define E(e) #e,
const char *ErrorStrings[] = {
    MY_ERRORS
};
#undef E

int main()
{
    printf("%d - %s \n", Error_TOO_SMALL, ErrorStrings[0]);
    printf("%d - %s \n", Error_TOO_BIG, ErrorStrings[1]);
    printf("%d - %s \n", Error_INVALID_VARS, ErrorStrings[2]);

    return 0;
}
```

我们把宏展开之后，得到一个枚举类型和一个字符串常量数组：

```
typedef enum {
    Error_TOO_SMALL,
    Error_TOO_BIG,
    Error_INVALID_VARS,
} MyEnums;

const char *ErrorStrings[] = {
    "TOO_SMALL",
    "TOO_BIG",
    "INVALID_VARS",
};
```

宏扩展之后的代码是不是很简单啊。编译、执行结果如下：

```
0 - TOO_SMALL
1 - TOO_BIG
2 - INVALID_VARS
```

七、总结

有些人对宏爱之要死，多到滥用的程度；而有些人对宏恨之入骨，甚至用上了邪恶(evil)这个词！其实宏对于 C 来说，就像菜刀对于厨师和歹徒一样：用的好，可以让代码结构简洁、后期维护特别方便；用的不好，就会引入晦涩的语法、难以调试的 Bug。

对于我们开发人员来说，只要在程序的执行效率、代码的可维护性上做好平衡就可以了。

不吹嘘，不炒作，不浮夸，认真写好每一篇文章！

欢迎转发、分享给身边的技术朋友，道哥在此表示衷心的感谢！

转发的推荐语已经帮您想好了：

道哥总结的这篇总结文章，写得很用心，对我的技术提升很有帮助。好东西，要分享！

【原创声明】

公众号【IOT物联网小镇】

作者：道哥(公众号: IOT物联网小镇)
知乎：道哥
B站：道哥分享
掘金：道哥分享
CSDN：道哥分享

关注+星标公众号，不错过最新文章

推荐阅读

利用C语言中的setjmp和longjmp，来实现异常捕获和协程
C语言指针-从底层原理到花式技巧，用图文和代码帮你讲解透彻
一步步分析-如何用C实现面向对象编程
原来gdb的底层调试原理这么简单
关于加密、证书的那些事
深入LUA脚本语言，让你彻底明白调试原理