**C语言陷阱与技巧第7节，define函数式宏定义不能用普通函数代替吗？为什么要使用do{}while(0)包裹代码**

上一节基于 usleep() 函数，使用若干行代码，简单实现了用于避免C语言程序陷入死循环的“超时”功能，并且为了方便之后的调用，我们还使用了 define 宏定义将“超时”代码封装成一个方法。相信读者已经发现 C语言中的 define 宏定义的强大了，它远远不止仅提供常数替换的功能。

#define N 5，只是 define 非常基础的用法之一。



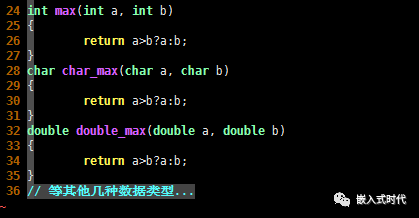
C语言中的“函数式宏定义”

C语言中的 define 宏定义可以像函数那样接收参数（这种宏定义常被称作“函数式宏定义”），不过不能像函数那样提供参数的类型检查，这个特点在有些程序员看来是不安全的。但是，函数式宏定义不关心参数类型这个特点，有时候也会被利用起来，写出一些适用性更广的C语言代码，例如：

#define max(\_\_a, \_\_b) ( (a)>(b)?(a):(b) )上面这段C语言宏定义代码实现了一个 max() 方法，它接收两个参数，并返回较大的那个参数，max() 方法不关心参数的类型，因此 \_\_a 和 \_\_b 可以是 int 型的，也可以是 char 型或者 double 型以及其他数据类型的。

如果使用 max() 方法提供的功能以C语言函数的方式来写，就稍显麻烦些了，程序员不得不为每一种数据类型实现一个 max() 函数。更加糟糕的是，C语言并不支持函数的重载，因此 max() 这个函数名一旦被使用，其他函数就不能再使用了，因此相关的C语言代码可能是下面这样的：

int max(int a, int b){return a>b?a:b;}char char\_max(char a, char b){return a>b?a:b;}double double\_max(double a, double b){return a>b?a:b;}// 等其他几种数据类型...



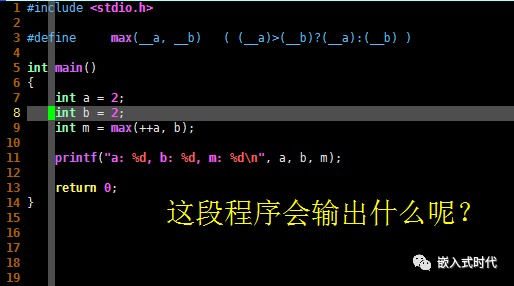
这样对比起来，显然使用 define 宏来定义 max() 方法更加方便一些。不过，C语言中的宏定义**不提供参数类型检查的确也是一个缺点**，它可能会导致程序的不安全，读者不应忽视这一点。因此如果不是必须要使用 define 宏定义才能解决问题，应该尽可能的使用函数，若是希望能够得到较高效率的代码，可以使用 inline 函数。

关于 inline 函数，在第 4 节已经较为详细的讨论过，这里就不再赘述了。

C语言中 define 宏定义的“陷阱”

C语言中的“函数式宏定义”虽然使用起来很像函数，但它实际上并不是函数，读者千万不能忽视这一点，不然可能会写出具有隐患，甚至严重错误的C语言程序。请看下面这个例子：

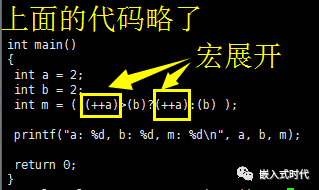
#include <stdio.h>#define max(\_\_a, \_\_b) ( (\_\_a)>(\_\_b)?(\_\_a):(\_\_b) )int main(){int a = 2;int b = 2;int m = max(++a, b);printf("a: %d, b: %d, m: %d\n", a, b, m);return0;}



上面这段C语言代码编译并执行，会输出什么呢？

在 main() 函数中，变量 a 和 b 都被初始化为 2。接着调用了 max() 宏，传递的参数分别是 ++a 和 b，粗略来看，此时执行 max(++a, b)，就相当于执行 max(3, 2)，那上面这段C语言程序会输出 3, 2, 3 了？得到答案最简单粗暴的方法就是编译并执行这段代码，请看：

# gcc t.c# ./a.outa: 4, b: 2, m: 4没有经验的读者看到实际输出估计会大吃一惊，a 和 m 怎么不是 3 而是 4 呢？并没有第二处给 a 再加一啊？上一节曾讨论，编译器会将C语言中的宏定义展开到被调用处，而不是像函数那样编译后，再通过 call 指令调用。使用 gcc -E 命令查看编译器将上述C语言代码预处理后的代码，得到如下结果，请看：



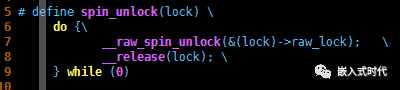
显然，这里就是C语言中“函数式宏定义”的“陷阱”了，传递给 max() 的参数 ++a 会被展开到宏定义中所有的 \_\_a 处，这就解释了为何 a 和 m 最后都等于 4 而不是 3 了。

“函数式宏定义”还有其他与真正函数不同的地方，例如“函数式宏定义”就不适合用于递归等。

使用 do{}while(0)包裹代码

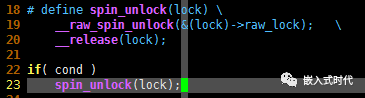
尽管C语言中的“函数式宏定义”和真正的函数相比有一些缺点，但只要小心使用还是会显著提高代码的执行效率的，毕竟省去了分配和释放栈帧、传参、传返回值等一系列工作。正因为如此，Linux 内核中有相当多的方法是使用 define 宏定义实现的，并且，在内核C语言代码中，“函数式宏定义”经常借助 do{}while(0) 实现，例如：

# define spin\_unlock(lock) \do {\ \_\_raw\_spin\_unlock(&(lock)->raw\_lock); \ \_\_release(lock); \ } while (0)



为什么要用 do{}while(0) 包裹C语言代码呢？不使用 do{}while(0) 包裹起来有什么不好吗？请看下面这几行代码：

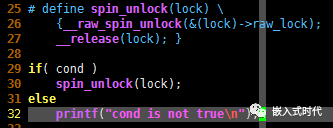
# define spin\_unlock(lock) \ \_\_raw\_spin\_unlock(&(lock)->raw\_lock); \ \_\_release(lock); if( cond ) spin\_unlock(lock);



宏定义被编译器展开后，会产生下面这样的C语言代码：

if(cond) \_\_raw\_spin\_unlock(&(lock)->raw\_lock);\_\_release(lock);这可能就与程序员的意图不一致了，这种情况下 \_\_release(lock); 并没有在 if(cond) 的作用范围内。可能读者会说，那像函数一样，使用 {} 包裹代码不就可以了吗？请再来看看下面这几行代码：

# define spin\_unlock(lock) \ {\_\_raw\_spin\_unlock(&(lock)->raw\_lock); \ \_\_release(lock); }if( cond ) spin\_unlock(lock);elseprintf("cond is not true\n");



问题就出在 spin\_unlock(lock); 后面的这个分号“;”，如果不写就不像函数调用，如果写了就会引发语法错误——if 语句会被这个“;”提前结束，else 无法与其配对。这么看来，在C语言的“函数式宏定义”中使用 do{}while(0) 包裹C语言代码显然就是一个不错的方法了。

小结

本节主要讨论了C语言中 define “函数式宏定义”的重要性，不过读者也应该明白，“函数式宏定义”并不是真正的函数，它与真正的函数是有区别的，如果弄不清楚这一点，很容易被“陷阱”迷惑。在最后，我们还一起分析了 Linux 内核中常用 do{}while(0) 包裹宏定义的代码的原因，读者今后在C语言程序开发中，也可以使用该技巧。