如果你是一名C程序员，你肯定很熟悉宏，它们非常强大，如果正确使用可以让你的工作事半功倍。然而，如果你在定义宏时很随意没有认真检查，那么它们可能使你发狂，浪费N多时间。在很多的C程序中，你可能会看到许多看起来不是那么直接的较特殊的宏定义。下面就是一个例子：

|  |  |
| --- | --- |
| 1  2 | #define \_\_set\_task\_state(tsk, state\_value)      \      do { (tsk)->state = (state\_value); } while (0) |

**在Linux内核和其它一些著名的C库中有许多使用do{...}while(0)的宏定义。**这种宏的用途是什么？有什么好处？

Google的[Robert Love](http://www.quora.com/Robert-Love-1)（先前从事Linux内核开发）给我们解答如下：

do{...}while(0)在C中是唯一的构造程序，让你定义的宏总是以相同的方式工作，这样不管怎么使用宏（尤其在没有用大括号包围调用宏的语句），宏后面的分号也是相同的效果。

这句话听起来可能有些拗口，其实用一句话概括就是：使用do{...}while(0)构造后的宏定义不会受到大括号、分号等的影响，总是会按你期望的方式调用运行。

例如：

|  |  |
| --- | --- |
| 1 | #define foo(x) bar(x); baz(x) |

然后你可能这样调用：

|  |  |
| --- | --- |
| 1 | foo(wolf); |

这将被宏扩展为：

|  |  |
| --- | --- |
| 1 | bar(wolf); baz(wolf); |

这的确是我们期望的正确输出。下面看看如果我们这样调用：

|  |  |
| --- | --- |
| 1  2 | if (!feral)      foo(wolf); |

那么扩展后可能就不是你所期望的结果。上面语句将扩展为:

|  |  |
| --- | --- |
| 1  2  3 | if (!feral)      bar(wolf);  baz(wolf); |

显而易见，这是错误的，也是大家经常易犯的错误之一。

几乎在所有的情况下，期望写多语句宏来达到正确的结果是不可能的。你不能让宏像函数一样行为——在没有**do/while(0)**的情况下。

如果我们使用do{...}while(0)来重新定义宏，即：

|  |  |
| --- | --- |
| 1 | #define foo(x) do { bar(x); baz(x); } while (0) |

现在，该语句功能上等价于前者，do能确保大括号里的逻辑能被执行，而while(0)能确保该逻辑只被执行一次，即与没有循环时一样。

对于上面的if语句，将会被扩展为：

|  |  |
| --- | --- |
| 1  2 | if (!feral)      do { bar(wolf); baz(wolf); } while (0); |

从语义上讲，它与下面的语句是等价的：

|  |  |
| --- | --- |
| 1  2  3  4 | if (!feral) {      bar(wolf);      baz(wolf);  } |

这里你可能感到迷惑不解了，***为什么不用大括号直接把宏包围起来呢？***为什么非得使用**do/while(0)**逻辑呢？

例如，我们用大括号来定义宏如下：

|  |  |
| --- | --- |
| 1 | #define foo(x)  { bar(x); baz(x); } |

这对于上面举的if语句的确能被正确扩展，但是如果我们有下面的语句调用呢：

|  |  |
| --- | --- |
| 1  2  3  4 | if (!feral)      foo(wolf);  else      bin(wolf); |

宏扩展后将变成：

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | if (!feral) {      bar(wolf);      baz(wolf);  };  else      bin(wolf); |

大家可以看出，这就有语法错误了。

**总结：Linux和其它代码库里的宏都用do/while(0)来包围执行逻辑，因为它能确保宏的行为总是相同的，而不管在调用代码中使用了多少分号和大括号。**