内核中有很多的宏定义，在宏定义 define 中经常看到两个字符串##和#，这里把它的用法做一下说明：

##是一个连接符号，用于把参数连在一起

例如：

> #define FOO(arg) my##arg

则

> FOO(abc)

相当于 myabc

#是“字符串化”的意思。出现在宏定义中的#是把跟在后面的参数转换成一个字符串

例如：

> #define STRCPY(dst, src) strcpy(dst, #src)

则

> STRCPY(buff, abc)

相当于 strcpy(buff, "abc")

另外，如果##后的参数本身也是一个宏的话，##会阻止这个宏的展开 。

```
#define STRCPY(a, b)       strcpy(a ## _p, #b)
int main()
{
      char var1_p[20];
      char var2_p[30];
      strcpy(var1_p, "aaaa");
      strcpy(var2_p, "bbbb");
      STRCPY(var1, var2);
      STRCPY(var2, var1);
      printf("var1 = %s\n", var1_p);
      printf("var2 = %s\n", var2_p);
      return 0;

      /* 注意这里 */
      STRCPY(STRCPY(var1,var2),var2);
      /* 这里是否会展开为： strcpy(strcpy(var1_p,"var2")_p,"var2“）？
       * 答案是否定的：
       * 展开结果将是： strcpy(STRCPY(var1,var2)_p,"var2")
       *## 阻止了参数的宏展开!
       * 如果宏定义里没有用到 # 和 ##, 宏将会完全展开
       */
}
```

//

tell you about ## in common text

关于记号粘贴操作符(token paste operator): ##

1. 简单的说，“##”是一种分隔连接方式，它的作用是先分隔，然后进行强制连接。

其中，分隔的作用类似于空格。我们知道在普通的宏定义中，预处理器一般把空格解释成分段标志，对于每一段和前面比较，相同的就被替换。但是这样做的结果是，被替换段之间存在一些空格。如果我们不希望出现这些空格，就可以通过添加一些##来替代空格。

另外一些分隔标志是，包括操作符，比如 +, -, *, /, [,], ...，所以尽管下面的宏定义没有空格，但是依然表达有意义的定义： define add(a, b) a+b

而其强制连接的作用是，去掉和前面的字符串之间的空格，而把两者连接起来。

2. 举列 -- 试比较下述几个宏定义的区别

```
#define A1(name, type)    type name_##type##_type  或
#define A2(name, type)    type name##_##type##_type

A1(a1, int);   /* 等价于: int name_int_type; */
A2(a1, int);   /* 等价于: int a1_int_type;     */
```

解释：

1) 在第一个宏定义中，"name"和第一个"_"之间，以及第 2 个"_"和第二个"type"之间没有被分隔，所以预处理器会把 name_##type##_type 解释成 3 段：“name_”、“type”、以及“_type”，这中间只有“type”是在宏前面出现过的，所以它可以被宏替换。

2) 而在第二个宏定义中，“name”和第一个“_”之间也被分隔了，所以预处理器会把 name##_##type##_type 解释成 4 段：“name”、“_”、“type”以及“_type”，这其间，就有两个可以被宏替换了。

3) A1 和 A2 的定义也可以如下：

```
#define A1(name, type)    type name_    ##type ##_type
                                        <##前面随意加上一些空格>
#define A2(name, type)    type name ##_ ##type ##_type
```

结果是## 会把前面的空格去掉完成强连接，得到和上面结果相同的宏定义

3. 其他相关 -- 单独的一个 #

至于单独一个#，则表示 对这个变量替换后，再加双引号引起来。比如

```
#define   __stringify_1(x)     #x
```

那么

```
__stringify_1(linux)     <==>   "linux"
```

所以，对于 MODULE_DEVICE_TABLE

```
1) #define MODULE_DEVICE_TABLE(type,name)
        MODULE_GENERIC_TABLE(type##_device,name)
2) #define MODULE_GENERIC_TABLE(gtype,name)
        extern const struct gtype##_id __mod_##gtype##_table
        __attribute__ ((unused, alias(__stringify(name))))
```

得到

```
MODULE_DEVICE_TABLE(usb, products)
                                /*notes: struct usb_device_id products; */
<==> MODULE_GENERIC_TABLE(usb_device,products)
<==> extern const struct usb_device_id __mod_usb_device_table
        __attribute__ ((unused, alias("products")))
```

注意到 alias attribute 需要一个双引号，所以在这里使用了__stringify(name)来给 name 加上双引号。另外，还注意到一个外部变量"__mod_usb_device_table"被 alias 到了本驱动专用的由用户自定义的变量 products<usb_device_id 类型>。这个外部变量是如何使用的，更多的信息请参看《probe()过程分析》。

4. 分析方法和验证方式 -- 编写一个简单的 C 程序

用宏定义一个变量，同时用直接方式定义一个相同的变量，编译报告重复定义；
用宏定义一个变量，直接使用该宏定义的变量名称，编译通过且运行结果正确；
使用 printf 打印字符串数据。printf("token macro is %s", __stringify_1(a1));