```
比如一下宏， 如果需要使用则需要包含它所在的头文件
/usr/include/linux/input-event-codes.h
....
#define ABS_X                   0x00
#define ABS_Y                   0x01
#define ABS_Z                   0x02
#define ABS_RX                  0x03
#define ABS_RY                  0x04
#define ABS_RZ                  0x05
...
```

编译器：

概念： 编译器是一个用来帮助我们把原码.c翻译成计算机能够之直接识别的二进制编码。使用不同的编译器可以翻译出来不同机器的二进制编码。

gcc 编译器：

```
gcc hello.c -o hello

gcc         -->  C语言编译器
hello.c     -->  需要编译的原码
-o          -->  指定输出文件名
hello       -->  可执行文件的名字
```

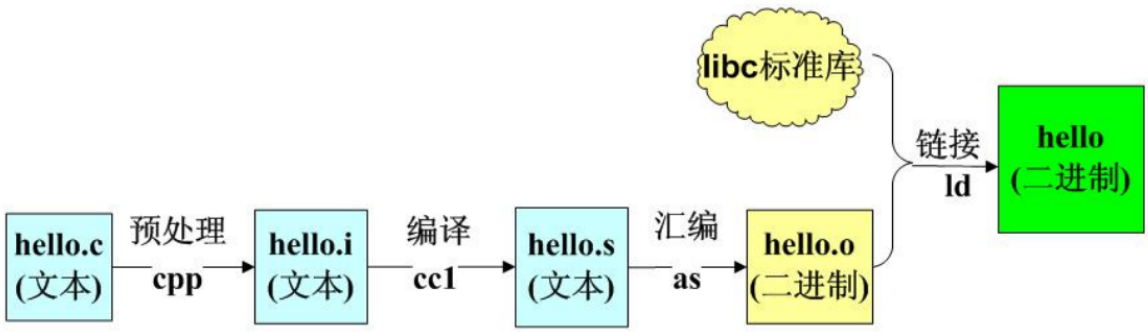

图 1-35 编译过程

1 预处理

```
gcc hello.c -o hello.i -E
加上一个编译选项 -E 就可以使得 GCC 在进行完第一阶段的预处理之后停下来，生成
一个默认后缀名为.i 的文本文件
```

预处理是指在编译代码之前先进行预先的处理工作，这些工作包含哪些内容：

- 头文件被包含进来（复制）： #include
- 宏定义会被替换：#define
- 取消宏定义： #undef
- 条件编译： #if #ifdef #ifndef #else #elif #endif
- 修改行号以及文件名： #line 998 "Hello.c"
- 清除注释

预处理大部分的工作是在处理以 # 开头的一些语句，从严格意义来讲这些语句并不属于C语言的范畴， 它们在编译的第一个阶段被所谓的 预处理器 来处理。

2 编译

```
gcc hello.i -o hello.s -S
```

加上一个编译选项 -S 就可以使得 gcc 在进行完第一和第二阶段之后停下来，生成一个默认后缀名为.s 的文本文件。打开此文件看一看，你会发现这是一个符合 x86 汇编语言的源程序文件。

经过预处理之后生成的.i 文件依然是一个文本文件，不能被处理器直接解释，我们需要进一步的翻译。接下来的编译阶段是四个阶段中最为复杂的阶段，它包括词法和语法的分析，最终生成对应硬件平台的汇编语言（不同的处理器有不同的汇编格式）。

汇编文件取决于所采用的编译器，如果用的是 GCC，那么将会生成 x86 格式的汇编文件，如果用的是针对 ARM 平台的交叉编译器，那么将会生成 ARM 格式的汇编文件。

3 汇编

```
gcc hello.s -o hello.o -c
-c 则是让编译器在对汇编语言文件进行编译后停下来，
    这里会生成一个待链接的可执行文件
```

则会生成一个扩展名为.o 的文件，这个文件是一个 ELF 格式的可重定位(relocatable)文件，所谓的可重定位，指的是该文件虽然已经包含可以让处理器直接运行的指令流，但是程序中的所有的全局符号尚未定位，所谓的全局符号，就是指函数和全局变量，函数和全局变量默认情况下是可以被外部文件引用的，由于定义和调用可以出现在不同的文件当中，因此他们在编译的过程中需要确定其入口地址，比如 a.c 文件里面定义了一个函数 func( )，b.c 文件里面调用了该函数，那么在完成第三阶段汇编之后，b.o 文件里面的函数 func( )的地址将是 0，显然这是不能运行的，必须要找到 a.c 文件里面函数 func( )的确切的入口地址，然后将 b.c 中的“全局符号” func重新定位为这个地址，程序才能正确运行。因此，接下来需要进行第四个阶段：链接。

可以尝试使用命令readelf 来查看这个可重定位文件

```
$ readelf demo.o -a
```

```
Section Headers:
  [Nr] Name              Type             Address           Offset
       Size              EntSize          Flags  Link  Info  Align
  [ 0]                   NULL             0000000000000000  00000000
       0000000000000000  0000000000000000           0     0     0
  [ 1] .text             PROGBITS         0000000000000000  00000040
       00000000000002ef  0000000000000000  AX       0     0     1
  [ 2] .rela.text        RELA             0000000000000000  00000680
       0000000000000210  0000000000000018   I      10     1     8
  [ 3] .data             PROGBITS         0000000000000000  0000032f
       0000000000000000  0000000000000000  WA       0     0     1
  [ 4] .bss              NOBITS           0000000000000000  0000032f
       0000000000000000  0000000000000000  WA       0     0     1
  [ 5] .rodata           PROGBITS         0000000000000000  00000330
       000000000000012c  0000000000000000   A       0     0     8
  [ 6] .comment          PROGBITS         0000000000000000  0000045c
       000000000000002a  0000000000000001  MS       0     0     1
  [ 7] .note.GNU-stack   PROGBITS         0000000000000000  00000486
       0000000000000000  0000000000000000           0     0     1
  [ 8] .eh_frame         PROGBITS         0000000000000000  00000488
       0000000000000038  0000000000000000   A       0     0     8
  [ 9] .rela.eh_frame    RELA             0000000000000000  00000890
       0000000000000018  0000000000000018   I      10     8     8
  [10] .symtab           SYMTAB           0000000000000000  000004c0
       0000000000000168  0000000000000018          11    10     8
  [11] .strtab           STRTAB           0000000000000000  00000628
       0000000000000055  0000000000000000           0     0     1
  [12] .shstrtab         STRTAB           0000000000000000  000008a8
       0000000000000061  0000000000000000           0     0     1
```

4 链接

```
gcc hello.o -o hello -lc -lgcc
gcc         .....
hello.o     ....
-o          ......
hello       .....
-lc          --> -l 链接      c   标准C库
-lgcc       --> -l 链接    gcc 链接GCC 的库
```

有两个很重要的工作没完成，首先是重定位，其次是合并相同权限的段

一个可执行镜像文件可以由多个可重定位文件链接而成，比如 a.o，b.o，c.o 这三个可重定位文件链接生成一个叫做 x 的可执行文件，这些文件不管是可重定位的，还是可执行的，它们都是 ELF 格式

的，ELF 格式是是符合一定规范的文件格式，里面包含很多段(section)，比如我们上面所述的 hello.c 编译生成的 hello.o 有如下的格式

```
[10] .rela.plt         RELA            0000000000000540  00000540
     0000000000000048  0000000000000018  AI       5    22     8
[11] .init             PROGBITS        0000000000000588  00000588
     0000000000000017  0000000000000000  AX       0     0     4
[12] .plt              PROGBITS        00000000000005a0  000005a0
     0000000000000040  0000000000000010  AX       0     0     16
[13] .plt.got          PROGBITS        00000000000005e0  000005e0
     0000000000000008  0000000000000008  AX       0     0     8
[14] .text             PROGBITS        00000000000005f0  000005f0
     0000000000000472  0000000000000000  AX       0     0     16
[15] .fini             PROGBITS        0000000000000a64  00000a64
     0000000000000009  0000000000000000  AX       0     0     4
[16] .rodata           PROGBITS        0000000000000a70  00000a70
     0000000000000134  0000000000000000   A       0     0     8
[17] .eh_frame_hdr     PROGBITS        0000000000000ba4  00000ba4
     000000000000003c  0000000000000000   A       0     0     4
[18] .eh_frame         PROGBITS        0000000000000be0  00000be0
```

宏的概念：

宏（macro）其实就是一个特定的字符串，用来直接替换比如：

```
#define  PI    3.14
```

上面定义了一个宏 名为 PI ，在下面代码的引用过程中 PI 将会被直接替换为实际的值

```
int main(int argc, char const *argv[])
{
    printf("%f\n" , PI );
    printf("%f\n" , 3.14 );
    return 0;
}
```

宏的作用：

- 使得程序的可读取有所提高，使用一个又有意义的单词来表示一个无意义数字（某个值）
- 方便对代码进行迭代更新， 如果代码中有多处使用到该值， 则只需要修改一处即可（定义）
- 提高程序的执行效率，可以使用宏来实现一个比较简单的操作。用来替代函数的调用

无参宏

意味着我们在使用该宏的时候不需要携带额外的参数

```
// 定义一个宏用来表示 人数
#define     PEOPLE      10
```

系统中有预定义一些宏：

```
比如一下宏， 如果需要使用则需要包含它所在的头文件
/usr/include/linux/input-event-codes.h
....
#define ABS_X                   0x00
#define ABS_Y                   0x01
#define ABS_Z                   0x02
#define ABS_RX                  0x03
#define ABS_RY                  0x04
#define ABS_RZ                  0x05
...
```

带参宏：

意味着我们在使用这些宏的时候，需要额外传递参数

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


int main(int argc, char const *argv[])
{

   printf("%d\n" , MAX(198,288) );
   //printf("%d\n" , 198>288? 198:288 );  --> 预处理后

    return 0;
}
```

以上MAX 的宏在使用的时候， 把198 以及 288 分别作为 a ,和b 的值， 然后进行替换。

注意：

宏只是直接文本替换，没有任何的判断以及语法检查的操作甚至运算

宏在编译的第一个阶段， 被处理完成，运行的过程中不占用时间（宏已经不存在）

宏在预处理的时候直接展开。

带参宏的副作用

由于宏只是一个简单的文本替换，中间不涉及任何的计算以及语法检查（类型），因此在使用复杂宏的时候需要小心注意

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


int x = 100 ;
int y = 200 ;
printf("%d\n" , MAX( x, y==200 ? 988:1024 ) );
// printf("%d\n" , x>y==200 ? 988:1024? x:y==200 ? 988:1024 );
```

从直观上来看不管 y==200？988:1024 结果如何都应该比100 大， 但是由于宏是直接的文本替换，导致替换之后三目运算符的运算结果出现了偏差。

由于带参宏的这个偏差出现的原因主要是优先级的问题，因此可添加括号来解决：

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

printf("%d\n" , (x)>(y==200 ? 988:1024)? (x):(y==200 ? 988:1024) );
```

注意：

宏只能写在同一行中（逻辑行），如果迫不得已需要使用多行来写一个宏则可以使用转义字符\把换行符转义掉

```
#define LIST_HEAD(name) \
    struct list_head name = \
    LIST_HEAD_INIT(name)
```

如上 41 42 43 为3个物理行， 但是通过转义字符让这三个物理行被看作同一个逻辑行

操作：

编译一个宏来求最小值（3个参数）

MIN(45 , 65 , 2) ;