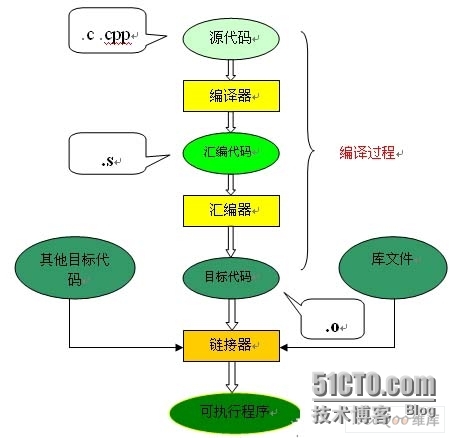
<https://blog.csdn.net/q_l_s/article/details/51314663>

学过C语言的人都应该知道，我们所编辑的C语言程序是不能直接放到机器上运行的，它只不过是一个带".c"后缀的文件（也称为源代码）而已，需要经过一定的处理才能转换成机器上可运行的可执行文件。我们将对C语言的这种处理过程称为编译与链接。

编译就是把文本形式源代码翻译为机器语言形式的目标文件过程。

链接是把目标文件、操作系统的启动代码和用到的库文件进行组织最终形成可执行代码的过程。编译和链接的过程图解如下：

[](http://img1.51cto.com/attachment/201309/161005463.jpg)

从图上可知，整个代码的编译过程分为编译和链接两个过程，其中**编译**对应图中的大括号括起来部分，其余则为**链接**过程。

其中编译过程又分为两个阶段：**编译和汇编。**

**编译**是读取源程序（字符流），对之进行词法和语法的分析，将高级语言指令转换为功能等效的汇编代码，源文件的编译过程主要包含两个阶段：

第一个阶段是：预处理阶段，在正式的编译阶段之前进行。预处理阶段将根据已放置的文件中的预处理指令来修改源文件的内容。如#include指令就是一个预处理指令，他把文件的内容添加到.cpp文件中。这个在编译之前修改源文件的方式提供了很大的灵活性，以适应不同的计算机和操作系统环境的限制。一个环境所需要的可执行代码跟另一个环境所需要的可执行代码可能有所不同，因为可用的硬件体系结构和操作系统不同所致。在许多情况下（特别是在嵌入式开发中），可以把用于不同环境的代码放在同一个文件中，再在预处理阶段修改代码，使之适应环境。

预处理阶段主要是以下几方面的处理：

1）、宏定义指令，如#define、M a；

对于这种伪指令，预编译所要做的是将程序中的所有M用a来替换，一定要注意作为字符常量a则不被替换（因为已经是常量，其值已经是确定的）。与之相对应的还有#undef，则是将取消对某个宏的定义，使之在后面出现时再不被替换。

2）、条件编译指令，如#ifdef、 #ifndef、#else、#elif、#endif等。

这些伪指令的引入使得程序员可以通过定义不同的宏来决定编译程序对哪些代码进行处理。换言而之也就是预编译程序将根据有关的文件，将哪些不必要的代码过滤掉。

3）、头文件包含指令，如#include等。

在头文件中一般用伪指令#define定义了大量的宏（最常见的是字符常量），同时包含各种外部符号的声明。采用头文件的主要目的是为了使某些定义可以供多个不同的C源程序使用。因为当需要使用到这些定义的C源程序中，只需要加上一条#include语句即可，而不必在此文件中将这些定义重复一遍。预编译程序将把头文件中定义统统加入到它所产生的输出文件中，以供编译程序对之处理。在linux操作系统中包含到C源程序中的头文件可以是系统提供的，这些头文件一般被放在/usr/include/目录下。在程序中使用它们，#include要使用尖括号<>；另外开发人员也可以定义自己的头文件，这些文件一般与C源程序放在同一目录下，此时在#include中要使用""。

4）、特殊符号，预编译程序可以识别一些特殊的符号。

例如，在源程序中出现的LINE标识符将被解释为当前行号（十进制），FILE则被解释为当前被编译的C源程序的文件名称，FUNCTION则被解释为当前被编译的C源程序中的函数名称。预编译程序对于在源程序中出现的这些串将用合适的值进行替换，这些常常是配套使用，用来进行对程序的调试。

最后需要着重强调一点的是预处理阶段并不属于预编译过程，这经常是初学者容易搞错的。

预编译程序所完成的基本上是对源程序的"替换"工作。经过此替换后，生成一个没有宏定义、没有条件编译指令、没有特殊符号的输出文件。这个文件的含义同没有经过预处理的源文件是相同的，但内容有所不同。下一步，此输出文件将作为编译程序的输出而被翻译成为机器指令。

第二个阶段是：编译、优化，经过预编译得到的输出文件中只有常量，一般都是一些指令。

编译程序所要做的工作就是通过词法分析和语法分析，在确认所有指令都是符合语法规则之后，将其翻译成等价的中间代码表示或汇编代码。

优化处理是编译系统中一项比较复杂高深的技术。它涉及到的问题不仅同编译技术有关，而且跟机器的硬件环境也有关。优化一部分是对中间代码的优化，这种优化不依赖于具体的计算机。另一种优化则主要针对目标代码的生成而进行的,这种优化与硬件环境有莫大的关系。

对于前一种优化，主要的工作是删除公共表达式、循环优化（代码外提、强度消弱、变换循环控制条件、已知量的合并等）、复写传播及无用赋值的删去等等。

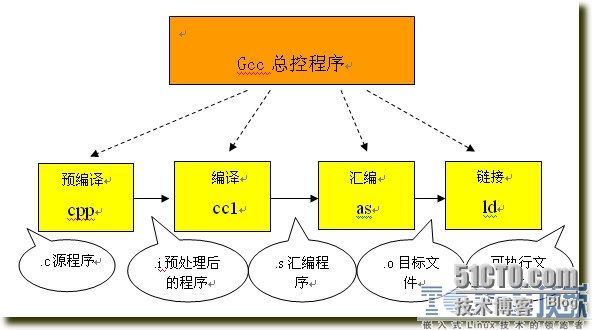
后一种类型的优化同机器的硬件结构密切相关，最主要的是考虑是如何充分利用机器的各个硬件寄存器存放的有关变量的值，以减少对于内存的访问次数（要知道内存资源对于计算机至关重要，控制好它可以大幅提高计算机的运算速度）。另外，如何根据机器硬件执行指令的特点（如流水线、RISC、CISC、VLIW等）而对指令进行一些调整使目标代码比较短，执行的效率比较高，也是一个重要的研究课题。

**汇编**实际上指把汇编语言代码翻译成目标机器指令的过程。对于被翻译系统处理的每一个C语言源程序，都将最终经过这一处理而得到相应的目标文件。目标文件中所存放的也就是与源程序等效的目标的机器语言代码。目标文件由段组成。通常一个目标文件中至少有两个段：代码段和数据段；

代码段：该段中所包含的主要是程序的指令。该段一般是可读和可执行的，但一般却不可写。  
数据段：主要存放程序中要用到的各种全局变量或静态的数据。一般数据段都是可读，可写，可执行的。

UNIX环境下主要有三种类型的目标文件：  
1）可重定位文件：其中包含有适合于其它目标文件链接来创建一个可执行的或者共享的目标文件的代码和数据。  
2）共享的目标文件：这种文件存放了适合于在两种上下文里链接的代码和数据。第一种是链接程序可把它与其它可重定位文件及共享的目标文件一起处理来创建另一个目标文件；第二种是动态链接程序将它与另一个可执行文件及其它的共享目标文件结合到一起，创建一个进程映象。  
3）可执行文件：它包含了一个可以被操作系统创建一个进程来执行的文件。汇编程序生成的实际上是第一种类型的目标文件。对于后两种还需要其他的一些处理方能得到，这个就是链接程序的工作了。

**链接**过程是由汇编程序生成的目标文件并不能立即就被执行，其中可能还有许多没有解决的问题。 例如，某个源文件中的函数可能引用了另一个源文件中定义的某个符号（如变量或者函数调用等）；在程序中可能调用了某个库文件中的函数，等等。所有的这些问题，都需要经链接程序的处理方能得以解决。  
链接程序的主要工作就是将有关的目标文件彼此相连接，也即将在一个文件中引用的符号同该符号在另外一个文件中的定义连接起来，使得所有的这些目标文件成为一个能够按操作系统装入执行的统一整体。  
根据开发人员指定的同库函数的链接方式的不同，链接处理可分为两种：  
1）静态链接  
在这种链接方式下，函数的代码将从其所在地静态链接库中被拷贝到最终的可执行程序中。这样该程序在被执行时这些代码将被装入到该进程的虚拟地址空间中。静态链接库实际上是一个目标文件的集合，其中的每个文件含有库中的一个或者一组相关函数的代码。  
2） 动态链接  
在此种方式下，函数的代码被放到称作是动态链接库或共享对象的某个目标文件中。链接程序此时所作的只是在最终的可执行程序中记录下共享对象的名字以及其它少量的登记信息。在此可执行文件被执行时，动态链接库的全部内容将被映射到运行时相应进程的虚地址空间。动态链接程序将根据可执行程序中记录的信息找到相应的函数代码。  
对于可执行文件中的函数调用，可分别采用动态链接或静态链接的方法。使用动态链接能够使最终的可执行文件比较短小（没有将函数部分拷贝），并且当共享对象被多个进程使用时能节约一些内存，因为在内存中只需要保存一份此共享对象的代码。但并不是使用动态链接就一定比使用静态链接要优越。在某些情况下动态链接可能带来一些性能上损害，例如移植性将大大降低。  
我们在linux使用的gcc编译器便是把以上的几个过程进行捆绑，使用户只使用一次命令就把编译工作完成，这的确方便了编译工作，但为了初学者了解编译过程，下图便给出了gcc代理的编译过程：

[](http://img1.51cto.com/attachment/201309/224935190.jpg)

从上图可以看到：  
a、预编译  
将.c 文件转化成 .i文件  
使用的gcc命令是：gcc –E  
对应于预处理命令cpp

b、编译  
将.c/.h文件转换成.s文件  
使用的gcc命令是：gcc –S  
对应于编译命令 cc –S  
  
c、汇编  
将.s 文件转化成 .o文件  
使用的gcc 命令是：gcc –c  
对应于汇编命令是 as  
  
d、链接  
将.o文件转化成可执行程序  
使用的gcc 命令是： gcc  
对应于链接命令是 ld

总结起来编译过程就上面的四个过程：预编译、编译、汇编、链接。了解了这四个过程中所做的工作，对我们理解头文件、库等的工作过程是有帮助的，而且清楚的了解编译链接过程还对我们在编程时定位错误，以及编程时尽量调动编译器的检测错误会有很大的帮助的。

呵呵，看完这些是不是对C语言的整个运行过程又有了进一步的了解呀！其实机器比人笨得多哦，进行一个进程的执行机器都需要无跳跃的一步一步的进行，不能像人的思维那样有跳跃性。^-^说到这里相信你有一定的感悟哦！呵呵...