[**《程序员的自我修养——链接、装载与库》——链接**](http://www.cnblogs.com/iyoyos/p/4241096.html)

**导读**

　　对于平常的应用程序开发，我们很少需要关注编译和链接过程，因为通常的开发环境都是流行的集成开发环境（IDE），比如Visual Studio、Myeclipse等。这样的IDE一般都将编译和链接的过程一步完成，通常将这种编译和链接合并在一起的过程称为构建，即使使用命令行来编译一个源代码文件，简单的一句”gcc hello.c”命令就包含了非常复杂的过程。然而，正是因为集成开发环境的强大，很多系统软件的运行机制与机理被掩盖，其程序的很多莫名其妙的错误让我们无所适从，面对程序运行时种种性能瓶颈我们束手无策。我们看到的是这些问题的现象，但是却很难看清本质，所有这些问题的本质就是软件运行背后的机理及支撑软件运行的各种平台和工具，如果能深入了解这些机制，那么解决这些问题就能够游刃有余，收放自如了。

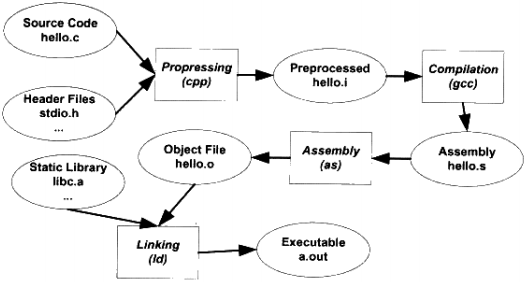
　　现在我们通过一个C语言的经典例子，来具体了解一下这些机制：

|  |  |
| --- | --- |
| 1  2  3  4  5 | #include <stdio.h>  int main(){      printf("Hello World");      return 0;  } |

在linux下只需要一个简单的命令（假设源代码文件名为hello.c）：

|  |  |
| --- | --- |
| 1  2  3 | $ gcc hello.c  $ ./a.out  Hello World |

　　事实上，上述gcc过程可分解为4个步骤：预处理、编译、汇编、链接，如图所示：



**目录**

* 预处理（Prepressing）
* 编译（Compliation）
* 汇编（Assembly）
* 链接（Linking）

**正文**

**1、  预编译（prepressing）**

　　首先是源代码文件hello.c和相关的头文件（如stdio.h等）被预编译器cpp预编译成一个.i文件。第一步预编译的过程相当于如下命令（-E 表示只进行预编译）：

|  |  |
| --- | --- |
| 1 | $ gcc –E hello.c –o hello.i |

或者，

|  |  |
| --- | --- |
| 1 | $ cpp hello.c > hello.i |

　　预编译过程主要处理源代码文件中以”#”开头的预编译指令。比如”#include”、”#define”等，主要处理规则如下：

* 将所有的”#define”删除，并展开所有的宏定义
* 处理所有条件预编译指令，比如”#if”，”#ifdef”，”#elif” ”，#else”，”#endif”
* 处理”#include”预编译指令，将被包含的文件插入到该预编译指令的位置。
* 删除所有的注释“//”和“/\*\*/”
* 添加行号和文件名标识，比如#2 “hello.c” 2。
* 保留所有的#pragma编译器指令

**2、  编译（compliation）**

　　编译过程就是把预处理完的文件进行一系列词法分析、语法分析、语义分析及优化后生产相应的汇编代码文件，此过程是整个程序构建的核心部分，也是最复杂的部分之一。其编译过程相当于如下命令：

|  |  |
| --- | --- |
| 1  2 | $ gcc –S hello.i –o hello.s |

Gcc是好多后台程序的包装，它会根据不同的参数要求去调用预编译程序cc1、汇编器as、链接器ld。

**3、  汇编（assembly）**

　　汇编器是将汇编代码转变成机器可以执行的指令，每一个汇编语句几乎对应一条机器令。所以汇编器的汇编过程相对于编译器来讲比较简单，它没复杂的语法，也没有语义，也不需要做指令优化，只是根据汇编指令和机器指令的对照表一一翻译就可以了。其汇编过程相当于如下命令：

|  |  |
| --- | --- |
| 1 | as hello.s –o hello.o |

或者，

|  |  |
| --- | --- |
| 1 | gcc –c hello.s –o hello.o |

或者使用gcc命令从C源代码文件开始，经过预编译、编译和汇编直接输出目标文件：

|  |  |
| --- | --- |
| 1 | gcc –c hello.c –o hello.o |

**4、  链接（linking）**

　　链接通常是一个让人比较费解的过程，为什么汇编器不直接输出可执行文件而是输出一个目标文件呢？为什么要链接？下面让我们来看看怎么样调用ld才可以产生一个能够正常运行的Hello World程序：

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | $ ld –static /usr/lib/crt1.o /usr/lib/crti.o    /usr/lib/gcc/i486-linux-gnu/4.3.1/crtbeginT.o    -L/usr/lib/gcc/i486-linux-gnu/4.3.1 –L/usr/lib/ -L/lib hello.o –start-group    -lgcc –lgcc\_eh –lc –end-group /usr/lib/gcc/i486-linux-gnu/4.3.1/crtend.o    /usr/lib/ctrn.o |

如果把所有的路径都省略掉，那么上面的命令：

|  |  |
| --- | --- |
| 1 | ld –static crti.o crtbeginT.o hello.o –start-group -lgcc –lgcc\_eh –lc –end-group crtend.o ctrn.o |

可以看到，我们需要将一大堆文件链接起来才可以得到“a.out”，即最终的可执行文件。看到这么复杂的命令，你可能会问，这些.o文件是什么？它们做什么用的？–lgcc\_eh –lc –end-group这些又是些什么参数？为什么要使用它们？为什么要将它们和hello.o链接起来才可能得到可执行文件？等等。

　　现在我们一起在探索他们的条案吧。

　　在现代软件开发过程中，软件的规模往往都很大，动辄数百万行代码，如果都放在一个模块肯定是无法想象的。所以现代的大型软件往往拥有成千上万个模块，这些模块之间相互依赖又相对独立。这种层次化及模块化存储和组织源代码有很多好处，比如代码更容易阅读、理解、重用，每个模块可以单独开发、编译、测试，改变部分代码不需要编译整个程序等。

　　在一个程序被分割成多个模块以后，这些模块之间最后如何组合形成一个单一的程序是须解决的问题。模块之间如何组合的问题可以归结为模块之间如何通信的问题，最常见的属于静态语言的C/C++模拟之间通信有两种方式：

* 模块间的函数调用
* 模块间的变量访问

　　函数访问须知道目标函数的地址，变量访问也须知道目标变量的地址，所以，这两种方式都可以归结为一种方式，那就是模块间符号的引用。模块间依靠符号来通信类似于拼图版，定义符号的模块多出一块区域，引用该符号的模块刚好少了那一块区域，两者一拼接刚好完美组合。这个模块的拼接过程就是链接。链接的主要内容就是把各个模块之间相互引用的部分都处理好，使得各个模块之间都能够正确的衔接，也就是把一些指令对其他符号地址的引用加以修正。链接过程主要包括了地址和空间分配（Address and storage allocation）、符号决议（symbol resolution）和重定位（relocation）等这些步骤。

　　现在我们举例解释一下编译和链接的概念。比如我们在程序模块main.c中使用另外一个模块func.c中的函数foo()。我们在main.c模块中每一处调用foo()的时候都必须确切知道foo()这个函数的地址，但是由于每个模块都是单独编译的，在编译器编译main.c的时候，它并不知道foo()函数的地址，所以它暂时把这些调用foo()的指令的目标地址搁置，等待最后链接的时候由链接器去将这些指令的目标地址修正。如果没有链接器，须要我们手工把每个调用foo()的指令进行修正，则填入正确的foo()函数地址。当func.c模块被重新编译，foo()函数的地址有可能改变时，那么我们在main.c中所有使用到foo()的地址的指令将要全部重新调整，这些繁琐的工作将成为程序员的噩梦。使用链接器，我们可以直接引用其他模块的函数和全局变量而无须知道它们的地址，因为链接器在链接的时候，会根据你所引用的符号foo()，自动去相应的func.c模块查找foo()地址，然后将main.c模块中所有引用到foo()的指令重新修正，让它们的目标地址为真正的foo()的指令重新修正，让它们的目标地址为真正的foo()函数地址。这也就是静态链接的最基本的过程和作用。

　　在链接过程中，对其他定义在目标文件中的函数调用的指令须要被重新调整，对使用其他定义在其他目标文件的变量来说，也存在同样的问题。让我们结合具体的CPU指令来了解这个过程。假设我们有个全局变量叫做var，它在目标文件A里面。我们在目标文件B里面要访问这个全局变量，比如我们在目标文件B里面有这么一条指令：

|  |  |
| --- | --- |
| 1 | mov1 $0x2a, var |

这条指令就是给这个var变量赋值0x2a，相当于C语言里面的语句var = 42。然后我们编译目标文件B，得到这条指令机器码，如图：



由于在编译文件B的时候，编译器并不知道变量var的目标地址，所以编译器在没法确定地址的情况下，将这条mov指令的目标地址置为0，等待链接器在将目标文件A和B链接起来的时候，再将其修正。我们假定A和B链接后，变量var的地址确定下来为0x1000，那么链接器将会把这个指令的目标地址部分修改成0x1000。这个地址修正的过程也被叫做重定位（relocation），每一个要被修正的地方叫一个重定位入口（relocation entry）。重定位所做的就是给程序中每个这样的绝对地址引用的位置“打补丁”，使它们指向正确的地址。