**编译和链接那点事**

原创： 作者：刘欢 [Linux内核之旅](javascript:void(0);)  4月24日

　　有位学弟想让我说说编译和链接的简单过程，我觉得几句话简单说的话也没什么意思，索性写篇博文稍微详细的解释一下吧。其实详细的流程在经典的《Linkers and Loaders》和《深入理解计算机系统》中均有描述，也有国产的诸如《程序员的自我修养——链接、装载与库》等大牛著作。不过，我想大家恐怕很难有足够的时间去研读这些厚如词典的书籍。正巧我大致翻阅过其中的部分章节，干脆也融入这篇文章作为补充吧。

　　我的环境：Fedora 16 i686 kernel-3.6.11-4 gcc 4.6.3

　　其实MSVC的编译器在编译过程中的流程是差不多的，只是具体调用的程序和使用的参数不同罢了。不过为了描述的流畅性，我在行文中不会涉及MSVC的具体操作，使用Windows的同学可以自行搜索相关指令和参数。但是作为Linuxer，我还是欢迎大家使用Linux系统。如果大家确实需要，我会挤时间在附言中给出MSVC中相对应的试验方法。

　　闲话不多说了，我们进入正题。在正式开始我们的描述前，我们先来引出几个问题：

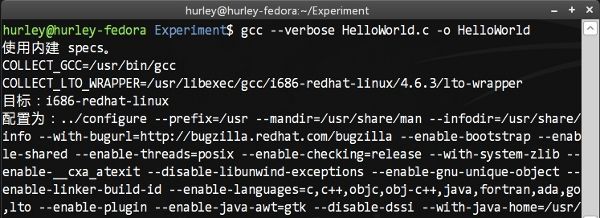
1. C语言代码为什么要编译后才能执行？整个过程中编译器都做了什么？
2. C代码中经常会包含头文件，那头文件是什么？C语言库又是什么？
3. 有人说main函数是C语言程序的入口，是这样吗？难道就不能把其它函数当入口？
4. 不同的操作系统上编译好的程序可以直接拷贝过去运行吗？

　　如果上面的问题你都能回答的话，那么后文就不用再看下去了。因为本文是纯粹的面向新手，所以注定了不会写的多么详细和深刻。如果你不知道或者不是很清楚，那么我们就一起继续研究吧。

　　我们就以最经典的HelloWorld程序为例开始吧。我们先使用vim等文本编辑器写好代码，接着在终端执行命令 gcc HelloWorld.c -o HelloWorld 输出了可执行文件HelloWorld，最后我们在终端执行 ./HelloWorld，顺利地显示了输出结果。

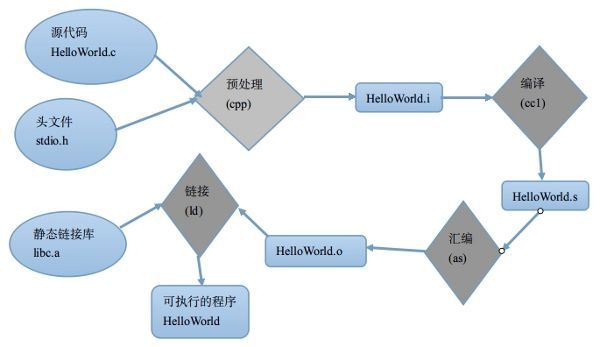


　　可是，简单的命令背后经过了什么样的处理过程呢？gcc真的就“直接”生成了最后的可执行文件了吗？当然不是，我们在gcc编译命令行加上参数 –verbose要求gcc输出完整的处理过程(命令行加上 -v 也行)，我们看到了一段较长的过程输出。



　　输出结果我们就不完整截图了，大家有兴趣可以自己试验然后试着分析整个流程。

　　一图胜千言，我们先上一张图吧。这是gcc编译过程的分解图，我在网上找不到满意的，就自己画了一张简单的，大家将就着看吧。



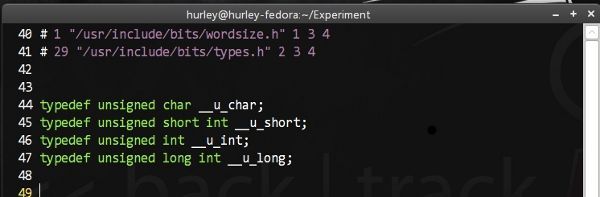
　　从图中我们大致可以看出gcc处理HelloWorld.c的大致过程：

　　预处理(Prepressing)—>编译(Compilation)—>汇编(Assembly)—>链接(Linking)

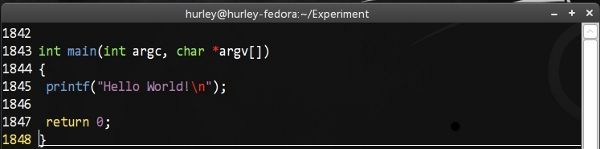
　　括号中我注明了各个过程中实际执行任务的程序名称：预处理器cpp、编译器cc1、汇编器as以及最后的链接器ld。

　　我们一步一步来看，首先是预处理，我们看看预处理阶段对代码进行了哪些处理。

　　我们在终端输入指令 gcc -E HelloWorld.c -o HelloWorld.i，然后我们打开输出文件。



　　首先是大段大段的变量和函数的声明，汗..我们的代码哪里去了？我们在vim的普通模式中按下shift+g(大写G)来到最后，终于在几千行以后看到了我们可怜兮兮的几行代码。



　　前面几千行是什么呢？其实它就是 /usr/include/stdio.h 文件的所有内容，预处理器把所有的#include替换为实际文件的内容了。这个过程是递归进行的，所以stdio.h里面的#include也被实际内容所替换了。

　　而且我在HelloWorld.c里面的所有注释被预处理器全部删除了。就连printf语句前的Tab缩进也被替换为一个空格了，显得代码都不美观了。

　　时间关系，我们就不一一试验处理的内容了，我直接给出预处理器处理的大致范围吧。

* 展开所有的宏定义并删除 #define
* 处理所有的条件编译指令，例如 #if #else #endif #ifndef …
* 把所有的 #include 替换为头文件实际内容，递归进行
* 把所有的注释 // 和 / / 替换为空格
* 添加行号和文件名标识以供编译器使用
* 保留所有的 #pragma 指令，因为编译器要使用 ……

　　基本上就是这些了。在这里我顺便插播一个小技巧，在代码中有时候宏定义比较复杂的时候我们很难判断其处理后的结构是否正确。这个时候我们呢就可以使用gcc的-E参数输出处理结果来判断了。

　　前文中我们提到了头文件中放置的是变量定义和函数声明等等内容。这些到底是什么东西呢？其实在比较早的时候调用函数并不需要声明，后来因为“笔误”之类的错误实在太多，造成了链接期间的错误过多，所有编译器开始要求对所有使用的变量或者函数给出声明，以支持编译器进行参数检查和类型匹配。头文件包含的基本上就是这些东西和一些预先的宏定义来方便程序员编程。其实对于我们的HelloWorld.c程序来说不需要这个庞大的头文件，只需要在main函数前声明printf函数，不需要#include 即可通过编译。

　　声明如下：

|  |  |
| --- | --- |
| 1 | intprintf(constchar\*format, ...); |

　　这个大家就自行测试吧。另外再补充一点，gcc其实并不要求函数一定要在被调用之前定义或者声明（MSVC不允许），因为gcc在处理到某个未知类型的函数时，会为其创建一个隐式声明，并假设该函数返回值类型为int。但gcc此时无法检查传递给该函数的实参类型和个数是否正确，不利于编译器为我们排除错误（而且如果该函数的返回值不是int的话也会出错）。所以还是建议大家在函数调用前，先对其定义或声明。

　　预处理部分说完了，我们接着看编译和汇编。那么什么是编译？一句话描述：编译就是把预处理之后的文件进行一系列词法分析、语法分析、语义分析以及优化后生成的相应汇编代码文件。这一部分我们不能展开说了，一来我没有系统学习过编译原理的内容不敢信口开河，二来这部分要是展开去说需要很厚很厚的一本书了，细节大家就自己学习《编译原理》吧，相关的资料自然就是经典的龙书、虎书和鲸书了。

　　gcc怎么查看编译后的汇编代码呢？命令是 gcc -S HelloWorld.c -o HelloWorld.s，这样输出了汇编代码文件HelloWorld.s，其实输出的文件名可以随意，我是习惯使然。顺便说一句，这里生成的汇编是AT&T风格的汇编代码，如果大家更熟悉Intel风格，可以在命令行加上参数 -masm=intel ,这样gcc就会生成Intel风格的汇编代码了（如图，这个好多人不知道哦）。不过gcc的内联汇编只支持AT&T风格，大家还是找找资料学学AT&T风格吧。



　　再下来是汇编步骤，我们继续用一句话来描述：**汇编就是将编译后的汇编代码翻译为机器码，几乎每一条汇编指令对应一句机器码。**

　　这里其实也没有什么好说的了，命令行 gcc -c HelloWorld.c 可以让编译器只进行到生成目标文件这一步，这样我们就能在目录下看到HelloWorld.o文件了。

　　Linux下的可执行文件以及目标文件的格式叫作ELF(Executable Linkable Format)。其实Windows下的PE(Portable Executable)也好，ELF也罢，都是COFF(Common file format)格式的一种变种，甚至Windows下的目标文件就是以COFF格式去存储的。不同的操作系统之间的可执行文件的格式通常是不一样的，所以造成了编译好的HelloWorld没有办法直接复制执行，而需要在相关平台上重新编译。当然了，不能运行的原因自然不是这一点点，不同的操作系统接口（windows API和Linux的System Call）以及相关的类库不同也是原因之一。

　　由于本文的读者定位，我们不能详细展开说了，有相关需求的同学可以去看《Windows PE权威指南》和《程序员的自我修养》去详细了解。

　　我们接下来看最后的链接过程。这一步是将汇编产生的目标文件和所使用的库函数的目标文件链接生成一个可执行文件的过程。我想在这里稍微的扩展一下篇幅，稍微详细的说一说链接，一来这里造成的错误通常难以理解和处理，二来使用第三方库在开发中越来越常见了，想着大家可能更需要稍微了解一些细节了。

　　我们先介绍gnu binutils工具包，这是一整套的二进制分析处理工具包。详细介绍请大家参考喂鸡百科：http://zh.wikipedia.org/wiki/GNU\_Binutils

　　我的fedora已经自带了这套工具包，如果你的发行版没有，请自行搜索安装方法。

　　这套工具包含了足够多的工具，我们甚至可以用来研究ELF文件的格式等内容。不过本文只是抛砖引玉，更多的使用方法和技巧还是需要大家自己去学习和研究。

　　由于时间关系，上篇到此就告一段落了，我们的问题2和3还没有给出完整的答案，而且链接还没有详细去解释和说明。这些内容我们将在下篇中解决，当然，大家也可以先行研究，到时候我们相互学习补充。

　　上回书我们说到了链接以前，今天我们来研究最后的链接问题。

　　链接这个话题延伸之后完全可以跑到九霄云外去，为了避免本文牵扯到过多的话题导致言之泛泛，我们先设定本文涉及的范围。我们今天讨论只链接进行的大致步骤及其规则、静态链接库与动态链接库的创建和使用这两大块的问题。至于可执行文件的加载、可执行文件的运行时储存器映像之类的内容我们暂时不讨论。

　　首先，什么是链接？我们引用CSAPP的定义：**链接（linking）是将各种代码和数据部分收集起来并组合成为一个单一文件的过程，这个文件可被加载（或被拷贝）到存储器并执行。**

　　需要强调的是，链接可以执行于编译时（compile time），也就是在源代码被翻译成机器代码时；也可以执行于加载时，也就是在程序被加载器（loader）加载到存储器并执行时；甚至执行于运行时（run time），由应用程序来执行。

　　说了这么多，了解链接有什么用呢？生命这么短暂，我们干嘛要去学习一些根本用不到的东西。当然有用了，继续引用CSAPP的说法，如下：

1. 理解链接器将帮助你构造大型程序。
2. 理解链接器将帮助你避免一些危险的编程错误。
3. 理解链接将帮助你理解语言的作用域是如何实现的。
4. 理解链接将帮助你理解其他重要的系统概念。
5. 理解链接将使你能够利用共享库。 ……

　　言归正传，我们开始吧。为了避免我们的描述过于枯燥，我们还是以C语言为例吧。想必大家通过我们在上篇中的描述，已经知道C代码编译后的目标文件了吧。目标文件最终要和标准库进行链接生成最后的可执行文件。那么，标准库和我们生成的目标文件是什么关系呢？

　　其实，任何一个程序，它的背后都有一套庞大的代码在支撑着它，以使得该程序能够正常运行。这套代码至少包括入口函数、以及其所依赖的函数构成的函数集合。当然，它还包含了各种标准库函数的实现。

**这个“支撑模块”就叫做运行时库（Runtime Library）。而C语言的运行库，即被称为C运行时库（CRT）。**

　　CRT大致包括：启动与退出相关的代码（包括入口函数及入口函数所依赖的其他函数）、标准库函数（ANSI C标准规定的函数实现）、I/O相关、堆的封装实现、语言特殊功能的实现以及调试相关。其中标准库函数的实现占据了主要地位。标准库函数大家想必很熟悉了，而我们平时常用的printf，scanf函数就是标准库函数的成员。C语言标准库在不同的平台上实现了不同的版本，我们只要依赖其接口定义，就能保证程序在不同平台上的一致行为。C语言标准库有24个，囊括标准输入输出、文件操作、字符串操作、数学函数以及日期等等内容。大家有兴趣的可以自行搜索。

　　既然C语言提供了标准库函数供我们使用，那么以什么形式提供呢？源代码吗？当然不是了。下面我们引入静态链接库的概念。我们几乎每一次写程序都难免去使用库函数，那么每一次去编译岂不是太麻烦了。干嘛不把标准库函数提前编译好，需要的时候直接链接呢？我很负责任的说，我们就是这么做的。

　　那么，标准库以什么形式存在呢？一个目标文件？我们知道，链接的最小单位就是一个个目标文件，如果我们只用到一个printf函数，就需要和整个库链接的话岂不是太浪费资源了么？但是，如果把库函数分别定义在彼此独立的代码文件里，这样编译出来的可是一大堆目标文件，有点混乱吧？所以，**编辑器系统提供了一种机制，将所有的编译出来的目标文件打包成一个单独的文件，叫做静态库（static library）。当链接器和静态库链接的时候，链接器会从这个打包的文件中“解压缩”出需要的部分目标文件进行链接。这样就解决了资源浪费的问题。**

　　Linux/Unix系统下ANSI C的库名叫做libc.a，另外数学函数单独在libm.a库里。静态库采用一种称为存档（archive）的特殊文件格式来保存。其实就是一个目标文件的集合，文件头描述了每个成员目标文件的位置和大小。

　　光说不练是假把式，我们自己做个静态库试试。为了简单起见我们就做一个只有两个函数的私有库吧。

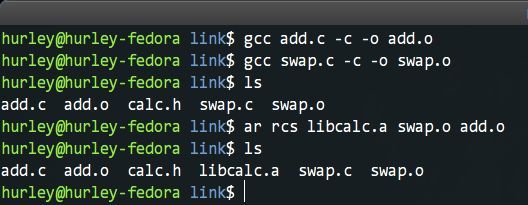
　　我们在swap.c里定义一个swap函数，在add.c里定义了一个add函数。最后还有含有它们声明的calc.h头文件。

|  |  |
| --- | --- |
| 1  　　2  　　3  　　4  　　5  　　6  　　7 | // swap.c  　　voidswap(int\*num1, int\*num2)  　　{  　　inttmp = \*num1;  　　\*num1 = \*num2;  　　\*num2 = tmp;  　　} |

|  |  |
| --- | --- |
| 1  　　2  　　3  　　4  　　5 | // add.c  　　intadd(inta, intb)  　　{  　　returna + b;  　　} |

|  |  |
| --- | --- |
| 1  　　2  　　3  　　4  　　5  　　6  　　7  　　8  　　9  　　10  　　11  　　12  　　13  　　14  　　15  　　16  　　17 | // calc.h  　　#ifndefCALC\_H\_  　　#defineCALC\_H\_  　　#ifdef\_cplusplus  　　extern"C"  　　{  　　#endif  　　voidswap(int\*, int\*);  　　intadd(int, int);  　　#ifdef\_cplusplus  　　}  　　#endif  　　#endif// CALC\_H\_ |

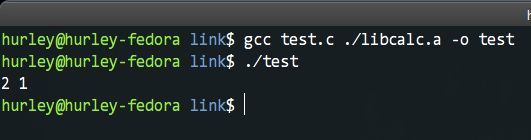
　　我们分别编译它们得到了swap.o和add.o这两个目标文件，最后使用ar命令将其打包为一个静态库。



　　现在我们怎么使用这个静态库呢？我们写一个test.c使用这个库中的swap函数吧。代码如下：

|  |  |
| --- | --- |
| 1  　　2  　　3  　　4  　　5  　　6  　　7  　　8  　　9  　　10  　　11  　　12  　　13  　　14  　　15 | #include<stdio.h>  　　#include<stdlib.h>  　　#include"calc.h"  　　intmain(intargc, char\*argv[])  　　{  　　inta = 1, b = 2;  　　swap(&a, &b);  　　printf("%d %dn", a, b);  　　returnEXIT\_SUCCESS;  　　} |

　　下来是编译执行，命令行执行gcc test.c ./libcalc.a -o test编译，执行。如图，我们输出了预期的结果。



　　可能你会问，我们使用C语言标准库的时候，编译并不需要加什么库名啊。是的，我们不需要。因为标准库已经是标准了，所以会被默认链接。不过因为数学函数库libm.a没有默认链接，所以我们使用了数学函数的代码在编译时需要在命令行指定 -lm 链接（-l是制定链接库，m是去掉lib之后的库名），不过现在好多gcc都默认链接libm.c库了，比如我机子上的gcc 4.6.3会默认链接的。

　　正如我们所看到的，静态链接库解决了一些问题，但是它同时带来了另一些问题。比如说每一个使用了相同的C标准函数的程序都需要和相关目标文件进行链接，浪费磁盘空间；当一个程序有多个副本执行时，相同的库代码部分被载入内存，浪费内存；当库代码更新之后，使用这些库的函数必须全部重新编译……

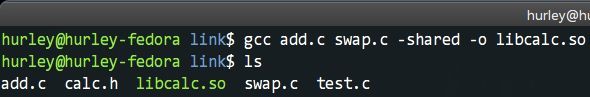
　　有更好的办法吗？当然有。我们接下来引入动态链接库/共享库（shared library）。

**动态链接库/共享库是一个目标模块，在运行时可以加载到任意的存储器地址，并和一个正在运行的程序链接起来。这个过程就是动态链接（dynamic linking），是由一个叫做动态链接器（dynamic linker）的程序完成的。**

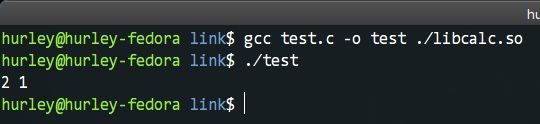
　　Unix/Linux中共享库的后缀名通常是.so（微软那个估计大家很熟悉，就是DLL文件）。怎么建立一个动态链接库呢？

　　我们还是以上面的代码为例，我们先删除之前的静态库和目标文件。首先是建立动态链接库，我们执行gcc swap.c add.c -shared -o libcalc.so 就可以了，就这么简单（微软那个有所区别，我们在这里只为说明概念，有兴趣的同学请自行搜索）。

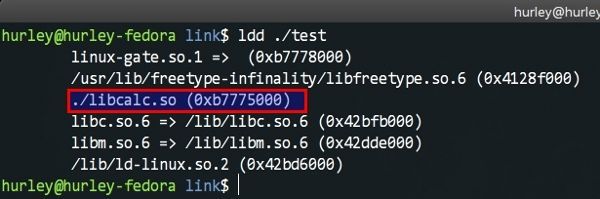
　　顺便说一下，最好在gcc命令行加上一句-fPIC让其生成与位置无关的代码（PIC），具体原因超出本文范围，故不予讨论。



　　如何使用呢？我们继续编译测试代码，执行gcc test.c -o test ./libcalc.so即可。运行后我们仍旧得到了预期的结果。



　　这看起来也没啥不一样的啊。其实不然，我们用ldd命令（ldd是我们在上篇中推荐的GNU binutils工具包的组成之一）检查test文件的依赖。



　　我们看到这个文件能顺利运行需要依赖libcalc.so这个动态库，我们还能看到C语言的标准库默认也是动态链接的（在gcc编译的命令行加上 -static 可以要求静态链接）。

　　好处在哪？第一，库更新之后，只需要替换掉动态库文件即可，无需编译所有依赖库的可执行文件。第二，程序有多个副本执行时，内存中只需要一份库代码，节省空间。

　　大家想想，C语言标准库好多程序都在用，但内存只有一份代码，这样节省的空间很可观吧，而且假如库代码发现bug，只需要更新libc.so即可，所有程序即可使用新的代码，岂不是很Cool。

　　好了，关于库我们就说到这里了，再说下去就没法子结束了。

　　我们来看看链接过程中具体做的事情。**链接的步骤大致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）等主要步骤。**

　　首先是地址和空间分配，我们之前提到的目标文件其实全称叫做可重定位目标文件（这只是一种翻译，叫法很多…）。目标文件的格式已经无限度接近可执行文件了，Unix/Linux下的目标文件的格式叫做ELF（Executable and Linkable Format，可执行连接格式）。详细的讨论可执行文件的格式超出了本文范围，我们只需要知道可执行文件中代码，数据，符号等内容分别存储在不同的段中就可以了，这也和保护模式下的内存分段是有一定关系的，但是这个又会扯远就不详谈了……

　　地址和空间分配以及重定位我们简单叙述一下就好，但是符号决议这里我想稍微展开描述一下。

**什么是符号（symbol）？简单说我们在代码中定义的函数和变量可以统称为符号。符号名（symbol name）就是函数名和变量名了。**

　　目标文件的拼合其实也就是对目标文件之间相互的符号引用的一个修正。我们知道一个C语言代码文件只要所有的符号被声明过就可以通过编译了，可是对某符号的引用怎么知道位置呢？比如我们调用了printf函数，编译时留下了要填入的函数地址，那么printf函数的实际地址在那呢？这个空位什么时候修正呢？当然是链接的时候，重定位那一步就是做这个的。但是在修改地址之前需要做符号决议，那什么是符号决议呢？正如前文所说，编译期间留下了很多需要重新定位的符号，所以目标文件中会有一块区域专门保存符号表。那链接器如何知道具体位置呢？其实链接器不知道，所以链接器会搜索全部的待链接的目标文件，寻找这个符号的位置，然后修正每一个符号的地址。

**这时候我们可以隆重介绍一个几乎所有人在编译程序的时候会遇见的问题——符号查找问题。这个通常有两种错误形式，即找不到某符号或者符号重定义。**

　　首先是找不到符号，比如，当我们声明了一个swap函数却没有定义它的时候，我们调用这个函数的代码可以通过编译，但是在链接期间却会遇到错误。形如“test.c:(.text+0x29): undefined reference to ‘swap’”这样，特别的，MSVC编译器报错是找不到符号\_swap。咦？那个下划线哪里来的？这得从C语言刚诞生说起。当C语言刚面世的时候，已经存在不少用汇编语言写好的库了，因为链接器的符号唯一规则，假如该库中存在main函数，我们就不能在C代码中出现main函数了，因为会遭遇符号重定义错误，倘若放弃这些库又是一大损失。所以当时的编译器会对代码中的符号进行修饰（name decoration），C语言的代码会在符号前加下划线，fortran语言在符号前后都加下划线，这样各个目标文件就不会同名了，就解决了符号冲突的问题。随着时间的流逝，操作系统和编译器都被重写了好多遍了，当前的这个问题已经可以无视了。所以新版的gcc一般不会再加下划线做符号修饰了（也可以在编译的命令行加上-fleading-underscore/-fno-fleading-underscore开打开/关闭这个是否加下划线）。而MSVC依旧保留了这个传统，所以我们可以看到\_swap这样的修饰。

　　顺便说一下，符号冲突是很常见的事情，特别是在大型项目的开发中，所以我们需要一个约定良好的命名规则。C++也引入了命名空间来帮助我们解决这些问题，因为C++中存在函数重载这些东西，所以C++的符号修饰更加复杂难懂（Linux下有c++filt命令帮助我们翻译一个被C++编译器修饰过的符号）。

　　说了这么多，该到了我们变成中需要注意的一个大问题了。即存在同名符号时链接器如何处理。不是刚刚说了会报告重名错误吗？怎么又要研究这个？很可惜，不仅仅这么简单。在编译时，编译器会向汇编器输出每个全局符号，分为强（strong）符号和弱符号（weak），汇编器把这个信息隐含的编码在可重定位目标文件的符号表里。其中函数和已初始化过的全局变量是强符号，未初始化的全局变量是弱符号。根据强弱符号的定义，GNU链接器采用的规则如下：

1. 不允许多个强符号
2. 如果有一个强符号和一个或多个弱符号，则选择强符号
3. 如果有多个弱符号，则随机选择一个

　　好了，就三条，第一条会报符号重名错误的，而后两条默认情况下甚至连警告都不会有。关键就在这里，默认甚至连警告都没有。

　　我们来个实验具体说一下：

|  |  |
| --- | --- |
| 1  　　2  　　3  　　4  　　5  　　6  　　7  　　8  　　9  　　10  　　11 | // link1.c  　　#include<stdio.h>  　　intn;  　　intmain(intargc, char\*argv[])  　　{  　　printf("It is %dn", n);  　　return0;  　　} |

|  |  |
| --- | --- |
| 1  　　2 | // link2.c  　　intn = 5; |

　　这两个文件编译运行会输出什么呢？聪明的你想必已经知道了吧？没错，就是5。

　　初始化过的n是强符号，被优先选择了。但是，在很复杂的项目代码，这样的错误很难发现，特别是多线程的……不过当我们怀疑代码中的bug可能是因为此原因引起的时候，我们可以在gcc命令行加上-fno-common这个参数，这样链接器在遇到多重定义的符号时，都会给出一条警告信息，而无关强弱符号。如图所示：

　　好了，到这里我们的下篇到此也该结束了，不过关于编译链接其实远比这深奥复杂的多，我权当抛砖引玉，各位看官自可深入研究。