Linux 下动态链接库的使用

编译、链接和装载

崔利伟

电子科技大学计算机科学与工程学院

cui@hellolw.com

Abstract

本文主要叙述了动态链接库的编译、链接和装载过程并讲解了其中的一些细节问题，并试图展示动态链接库使用相关的概念和技术。

Keywords 动态链接库，编译，链接，装载

1. 动态链接库概述

1.1. 库的概要回顾

我们所说的“库”，指的是为了复用代码或隐藏底层细节而制成的模块。开发者便可以方便地通过接口调用模块功能而不必获得源码或考虑实现，所以可以提高复用性并降低耦合度，非常适合模块化编程。我们常见的动态链接库有 GNU 的标准库 (libstdc++.so)，Intel 公司的数学库 (libmkl.so) 等。

Note. 任何代码均可以制成库，但由于 C++ 语言标准对于类、多态等概念并没有实现上的约束，导致编译期多态（模板）和运行时多态（虚表）的实现均可以由编译器厂商自行决定，所以造成二进制接口 (application binary interface) 互不兼容，或者说不可保证兼容。因此多数库的代码均由 extern "C" 关键字进行声明以避免编译器的名字混淆 (name mangling)。

1.2. 与静态库的联系

目前的主流操作系统均支持动态链接库和静态链接库。这里的”动态”“静态”之分，主要指的是代码是如何被链接的。静态库本质上是一组目标文件的集合，在链接过程中与其余的文件进行符号解析 (symbol resolution) 及地址重定位 (relocation) 操作。这样将会在每一个最终生成的结果中保留一份库的副本（一般来说是库的一部分，即满足其他文件所缺少的那部分符号表的目标文件）。这样的优点是不必对执行环境有着过多的依赖，但缺点同样明显：常用的库可能会有上千份同样的副本同时存在于内存和外存中；版本迭代将会是巨大的负担，新版本的库会要求所有使用旧版本的程序进行重新链接。为了解决这个两个问题，我们希望能把功能模块化，变成多个独立的文件而非一个。

1.3. 动态链接库概述

使用模块的思想，我们可以用另一种方式使用库：在内存中只保留一份库的副本并让所有有需要的程序共用。这意味着我们要将链接操作推迟到加载甚至运行时再进行。现代的库和“插件”功能都是作为动态库来被使用的。这样就可以解决对于内存外存的浪费以及版本迭代问题。进一步说，动态链接的性能损失也可以会通过物理页面的换入换出减少，CPU 缓存的命中率的提高来弥补。

2. 编译链接动态库的技术

我们知道动态库最终也会被载入内存并进行重定位。但程序又可能依赖多个动态库，造成每个动态库都不能假设自己在进程虚拟地址空间中的位置，所以为重定位造成了困难。而一般的程序指令和数据又会或多或少地使用了绝对地址引用，所以早期操作系统是通过给已知的模块预留空间来解决问题的。但随着程序的日益复杂这个方案逐渐捉襟见肘。为了解决这个问题，出现了称作“装载时重定

位 (load time relocation)”和“地址无关代码 (position-independent code)”的技术。

2.1. 装载时重定位

“装载时重定位”的基本思想是在链接时对于所有需要绝对地址的引用均不作重定位，将这个过程推迟到装载时进行。由于每个进程都拥有一份动态库可修改数据部分的副本，通过在装载时简单地加上偏移量可以解决数据部分的重定位的问题。但由于动态库的指令部分在整个内存中仅有一份副本，因此不能像数据段一样直接修改来完成重定位。所以之后出现了“地址无关代码”的技术。

2.2. 地址无关代码

事实上由于模块内的函数调用可以通过相对位置计算出来，通过偏移量进行调用事实上就是一种“地址无关代码”技术。但对于外部模块的调用和全局变量的引用是无法通过偏移量得出的，所以我们需要一些其他结构来实现这一技术。我们把指令中需要变动的地址提出来作为全局偏移量表 (global offset table) 和过程链接表 (Procedure Linkage Table) 并作为数据的一部分以供动态调整。这样其他指令无需变化（也即没有代码重定向表），只需让不同进程通过修改自己的副本部分就可达到复用库的目的。

Note. 我们可以通过编译选项-fPIC 来使用“地址无关代码”技术，通过 readelf -r 来查看需要进行修正的引用。

2.2.1. 全局偏移量表

由于加载文件到内存中时数据段总是紧随在代码段后面，因此代码段的任意指令到数据段的指令的距离是不会随着加载位置而改变的，它们都有一个自己的“运行时常量”来索引全局偏移量表。因此编译器利用这个事实，在数据段的开始创建了一个全局偏移量表，每个需要“地址无关代码”技术的引用（如全局变量、其他模块的函数）都在这个表中有一个条目。加载时动态链接器会重定位其中的每一个条目使其包含正确的绝对地址。

代码中使用全局变量的地方都可以通过当前代码的地址加上自己的“运行时常量”来拿到全局偏移量表，再间接获得变量的值。这样做会有相应的性能损失，所以对于函数调用，我们还需要过程链接表的配合。

2.2.2. 过程链接表

使用全局偏移量表和过程链接表来处理外部模块的函数引用的技术又称为延迟绑定 (lazy binding)。这个技术将真正的绑定推迟到该函数第一次被调用时。第一次调用时开销会很大，但之后的调用仅需多花费一条指令。

过程链接表在.data 段中，因此是不可变的。一开始动态链接器会将对外部函数的调用绑定在这个表中，而这个表又会去引用全局偏移量表中的适当条目；而该函数的全局偏移量表又跳转至过程链接表。控制权第二次回到过程链接表时它将向后执行，调用动态链接器来真正地找到这个外部函数的地址，并重写全局偏移量表的条目。这样，下次再通过过程链接表调用函数时全局偏移量表中就有正确的地址了，也就可以直接找到该外部函数的绝对地址。

2.3. 动态链接库的相关结构

2.3.1. .dynamic 段

.dynamic 段可以认为是库的“头”，保存了一些动态链接器所需的基本信息。比如动态库的 soname、依所赖的其他库、符号表 (symbol table) 的位置、重定向表 (relocation table) 的位置等。

Note. 事实上目标文件、可执行文件和库均为 ELF 文件格式 (Extensible Linking Format)，因此都具有同样的结构。我们可以通过 readelf -d 参看 ELF 文件头。

2.3.2. 动态符号表

所有的 ELF 文件均维护了段.symtab，也即符号表。它保存了所有关于该文件的符号的定义和引用。同时动态链接库又维护了段.dynsym，也即动态符号表 (dynamic symbol table)。动态符号表只保存与动态链接相关的符号，不保存只用于模块内部的符号。我们可以认为动态符号表是符号表的一个子集，并且不会被 strip 命令删去。通过阅读动态符号表我们可以了解到该文件有哪些符号未定义需要外部导入，那些符号已有定义可以向外导出。当经过静态链接

或者动态链接后均不可解析出某个未定义的符号时程序会在运行时报错。

Note. 我们可以通过编译选项-shared 生成动态符号表，使用 readelf -s 来查看动态符号表。

2.3.3. 动态链接重定位表 (dynamic relocation table)

当 ELF 文件中引用了其他模块的符号时我们需要在适当时刻将这些未定义的引用进行修正，也即需要重定位。在静态链接的程序中未知的地址会在链接过程中被修正，而动态链接的程序则需要在装载时或者运行时进行修正（针对数据段，因为通过“地址独立代码”技术代码段已不存在绝对地址的引用）。动态链接的重定向表主要是段.rel.dyn 和.rel.plt，前者是对数据引用（包括全局偏移量表）的修正，后者是对函数引用（也即链接过程表）的修正。

当动态链接器需要对某个符号进行重定位时，它将在全局符号表中查找该符号的地址。当查找到后会将该地址填入重定位表通过偏移量指定的特定位置来完成重定位。

Note. 我们可以通过 readelf -r 来查看重定位表。

2.4. 指定库的依赖

链接时为一个程序指定所依赖的动态库有着多种办法。

- 我们可以直接将.so 文件与.o 文件一并作为链接器参数传入。当.so 包含路径时路径信息也一并保留。但要注意当使用相对路径时装载会以进程执行路径为准。
- 我们也可以通过-lxxx 的形式传入。当通过这种方式传入时该动态库的名字必须以 libxxx.so 的形式命名，且该库不在链接器的默认搜索路径时我们需要通过-L 参数进行路径指定。-L 参数指定的是链接器的搜索路径，与动态链接器没有关系。其并不影响共享库的查找过程。
- 我们甚至可以不指定它所依赖的动态库，而通过 libdl.so 提供的接口在运行时进行库的装载和链接。

3. 装载动态库

3.1. 可执行文件的装载

3.1.1. 转载阶段

当一个可执行文件被要求执行时，父进程生成一个子进程，子进程通过调用装载器 (loader) 删除现有的虚拟地址空间，并会创建一组新的数据段代码段等，即创建一个新的虚拟地址空间。之后检查文件头，确定魔数匹配，然后俩将可执行文件中的数据和代码从外存拷贝到内存中。这个过程事实上是从文件头取出每个段并按照每个段的不同要求将其映射到进程虚拟空间的相关位置。由于可执行文件事实上指定了虚拟地址空间的映射关系，所以可执行文件也叫做“映像文件 (image file)”。加载器可以通过 execve 族函数调用。

3.1.2. 跳转阶段

系统之后将控制权转移至可执行文件的入口点 (entry point)。入口点即为 glibc 中 start.S 文件中定义的 _start 函数，在真正执行程序时的调用顺序为 _start->__libc_start_main->LIBC_START_MAIN->main– 事实上 start.S 会被编译成 crt1.o 并作为 C runtime 的一部分。

程序的入口点是通过程序头 (Program Headers) 指定的，但内核只知道进程的起始地址，无法找到入口点。但 ELF 文件头一定会加到偏移量为 0 的地址空间上，所以 ELF 文件头中保留了程序头的起始地址。

之后程序调用.text 和.init 段调用初始化历程，并通过 atexit 函数附加一些正常终止时应进行的操作。程序结束时会运行 atexit 中注册的函数，之后通过 _exit 函数将控制返回操作系统。

Note. 我们可以通过 readelf -h 来参看 ELF 文件头。通过 readelf -r 可以看到可执行文件有着对 __libc_start_main 的重定位需求，说明程序装载时需要动态链接 libc.so。

3.2. 动态库的装载

3.2.1. 程序加载时装载

当该可执行文件使用了动态链接库时系统会将控制权先转移至动态链接器的入口地址，完成装载时链接后才会将控制权转移至可执行文件。

动态加载器首先会重定位动态库的文本段和数据段到某个段。由于此时可执行文件的符号表中会有许多未定义的外部地址，动态链接器会来完成这些符号的重地位。我们可以通过读取可执行文件中的.interp 段来获得一个字符串，这个字符串就是动态链接器的路径。Linux 系统下的动态链接器 ld-linux.so 本身就是一个动态库，所以它在完成自举后再将可执行文件中的符号表与其余动态库的符号表进行重定位。完成这些操作后系统才将控制权真正交给可执行文件的入口。

3.2.2. 显式运行时装载

当系统支持动态链接库时它往往也支持运行时链接动态库。将可执行文件与 libdl.so 链接，程序运行时便可以通过 libdl.so 提供的接口进行动态库的装载和重定位操作。事实上，libdl.so 就是 dl-linux.so 的应用程序编程接口 (application program interface)。这种运行时加载可以使得程序的模块组织变得灵活，并且可以实现按需加载插件、驱动等功能，所以可以减少程序的启动时间和内存占用。

Note. 我们可以通过查看/proc 下进程 ID 对应的文件下的 map 文件来了解都有那些库被打开，在虚拟地址空间的什么位置。

4. 共享库的查找过程

4.1. 动态链接器

动态链接器将以深度优先搜索的形式试图打开可执行文件所依赖的所有库，也即会递归打开直到所有的动态库都正确装载或者出现装载失败，也即查找不到所依赖的库。动态链接器会将新打开的库中的动态符号表导出至全局符号表。对于未定义符号的解析是双向进行的，之前加载的库也可以利用之后加载的库中的符号进行重定向。当发生重名时，将以先打开的库的符号为准。

Note. 我们可以通过 ldd 命令或者 readelf -d 来查看一个 ELF 文件所依赖的动态库。

在静态链接时如果通过“目录 + 库名”将动态库作为参数传入，这样在生成的动态库的依赖库的库名中也会包含“目录 + 库名”的结构。这种包含反斜线的库名将仅会被动态链接器尝试直接打开，如果直接打开失败将直接报错。

当不包含反斜杠时，动态链接器的查找顺序如下。当能打开动态库文件时结束查找，否则进入下一优先级进行查找直到完成或者出错。

1. 当该可执行文件的.dynamic 段中有 DT_RPATH 属性时且没有 DT_RUNPATH 属性时动态链接器将会在 DT_RPATH 属性指定的目录中进行搜索。该选项可以可以在链接时通过向链接器设置-rpath 参数来指定。
2. 当系统的环境变量中包含 LD_LIBRARY_PATH 时动态链接器将在该环境变量中指定的路径中进行查找。
3. 当可执行文件的 dynamic 段中包含 DT_RUNPATH 时在该路径中进行查找。
4. 之后动态链接器将在/etc/ld.so.cache 缓存的路径中进行查找。该缓存文件由 ldconfig 命令生成，该命令将在/etc/ld.so.conf.d/*.conf 文件中指定的路径进行查找并将结果写入缓存文件。
5. 在系统目录/lib 和/usr/lib 下查找。

4.2. 动态链接器的 C 语言接口

通过在链接时指定-ldl 我们可以使用 dlopen、dlsym 等接口在运行时加载动态库。当内存中已存在该动态库时 dlopen 返回该动态库的句柄并将引用计数加一。dlclose 将关闭动态库并将引用计数减一。当引用计数为零时操作系统将真正卸载该库。

dlopen 有多种方式来决定如何处理新打开的动态库中的符号表。与动态链接器表现相同，当动态库名包含反斜线时将试图直接打开，当不包含时则会依据动态链接器的查找顺序进行查找。

dlopen 必须指定一下两个标志符之一：

- RTLD_LAZY 将延迟绑定引用：只有当这个引用被真正使用时才会触发符号解析，也即使用动态链

接器进行解析。这需要上文中全局偏移量表和过程连接表的配合使用。

- RTLD_NOW 将立刻绑定引用：不管这个引用是否真正在后文用到，都将使用动态链接器进行解析。

dlopen 还可以选择零个或者多个标志符来实现某些特殊的功能：

- RTLD_GLOBAL 将本库中的符号全局化，可以被后来的打开的库使用
- RTLD_LOCAL 作用和上者相反，不能让本库中的符号全局化。这也是默认行为
- RTLD_NODELETE 不真正卸载这个库，因此引用计数不会变化
- RTLD_NOLOAD 不真正加载这个库，可以用来改变已打开的库的标志符
- RTLD_DEEPBIND 符号查找将先从本库开始，之后才去查找全局范围

4.2.1. 手动打开依赖树

在某些复杂的依赖情况下，动态库本身还要依赖其他的动态库。这样最好的实现方式是在链接时直接指定相关的依赖，但有些情况下如果没有指定，我们依旧可以通过动态链接器的编程接口来完成相应的依赖。

比如 bar@A.so 依赖 foo@B.so，而两者都未设定自身的依赖。这时我们可以通过 RTLD_GLOBAL 打开 B.so，然后用普通方式打开 A.so。通过 dlsym 获取 bar@A.so 的指针，之后调用它。这会出发符号解析，完成后控制权转移到 bar@A.so 中。这个函数中又调用了 foo@B.so，再次触发富豪解析：由于 RTLD_GLOBAL 的缘故，动态链接器会去 B.so 中进行查找，因此可以找到相应的符号并设置在全局偏移量表中。

但如果 A.so 制定了自身依赖 B.so，但某些情况下动态链接器找不到 B.so 时，我们依旧可以通过提前打开依赖树的方式来满足这一要求。这时我们需要保证 B.so 的 soname 和 A.so 中 ELF 文件头中 NEEDED 的库名相同。这样系统将认为被依赖的库已被打开，因此不会报错。