

7.3.1 Linux 运行时动态库文件的定位规则

动态库运行时搜索算法由以下一组规则约束，按照优先级从高到低列出。

1. 预加载库

毫无疑问，预加载库应该拥有最高的搜索优先级，因为装载器会首先加载这些库，然后才开始搜索其他库。有两种方式可以指定预加载库：

- 通过设置 LD_PRELOAD 环境变量。

```
export LD_PRELOAD=/home/milan/project/libs/libmilan.so:$LD_PRELOAD
```

- 通过 /etc/ld.so.preload 文件。

该文件中包含的 ELF 共享库文件会在程序启动前加载，文件列表使用空格分隔。
指定预加载库并不符合标准的设计规范。相反，该方案仅用于特殊情况，比如设计压力测试、诊断以及对原始代码的紧急补丁等。

在使用该方法进行诊断时，你可以快速创建自定义版本的标准函数，并在其中附加用于调试的输出信息。然后构建一个共享库，利用预加载机制将原有函数的动态库替换掉。

在预加载阶段完成库文件的加载操作之后，装载器开始根据依赖搜索其他罗列出的库。装载器遵循一组复杂的规则执行搜索，后面几节将针对完整的规则列表（根据优先级从高到低安排）进行详细的讨论。

rpath

从很早开始，ELF 格式就使用 DT_RPATH 字段来存储与二进制文件相关的搜索路径细节，该字段使用 ASCII 字符串表示。比如，如果可执行文件 XYZ 运行时依赖于动态库 ABC，XYZ 可能在其 DT_RPATH 字段中存储字符串，指定 ABC 运行时可能出现的路径。

这项功能为开发人员提供了一种清晰且细致入微的控制部署过程的方法，最为显著的是很大程度避免了期望的库版本的库与可用的库版本不匹配的问题。

可执行文件 XYZ 中 DT_RPATH 字段所存储的信息最终会在运行时由装载器读取出来。需要记住的一个重要细节是，从哪个路径开始解析对装载器解析 DT_RPATH 信息会产生影响。在 DT_RPATH 存储了相对路径的情况下，装载器并不是将其解释成相对于库 XYZ 的相对路径，而是相对于装载器（即应用程序）启动路径的相对路径。虽然 rpath 已经满足我们的需求，不过其概念还是经过了一系列的改进。

网上的资料显示，在 1999 年左右，第 6 版 C 运行时库正式替代第 5 版成为主流版本，与此同时也注意到了 rpath 的一些缺陷，并使用 ELF 二进制文件格式中的一个相似字段 runpath (DT_RUNPATH) 取代了 rpath。

如今 rpath 和 runpath 都是可供我们使用的，但是 runpath 在运行时搜索优先级列表中被赋予了更高的优先级。只有在 runpath (DT_RUNPATH 字段) 缺失的情况下，rpath (DT_RPATH 字段) 才是 Linux 装载器剩余的搜索路径信息中具有最高优先级的。但如果 ELF 二进

P要非空,且不以tail,再找 rpath.

制文件的 runpath (DT_RUNPATH) 字段是非空的,那么 rpath 也会被忽略。

我们通常使用 -R 或者 -rpath 选项向链接器传递 rpath, 随后我们介绍的 runpath 路径的赋值方法也是一样的。此外, 根据惯例, 凡是间接调用链接器时(也就是直接调用 gcc 或者 g++), 我们需要在链接器参数前追加 “-Wl,” 前缀(也就是“减号 Wl 逗号”)。

```
$ gcc -Wl,-R/home/milan/projects/-lmilanlibrary
```

```
^ ^ ^
| | |
| actual rpath value
| |
| run path linker flag
|
-Wl, prefix required when invoking linker
indirectly, through gcc instead of
directly invoking ld
```

或者, 也可以使用 LD_RUN_PATH 环境变量来指定 rpath: ←链接时

```
$ export LD_RUN_PATH=/home/milan/projects:$LD_RUN_PATH
```

最后要提一点是, 在二进制文件生成之后, 通过 chrpath 工具, 就可以对 rpath 进行修改。但 chrpath 存在一个问题是其无法修改超出原有的 rpath 字段的长度。更准确地说, chrpath 可以改变、删除或清空 DT_RPATH 字段, 但是无法插入该字段, 或是将该字段扩展成更长的字符串。

查看二进制文件 DT_RPATH 值的方法是查看二进制文件的 ELF 头(比如执行 readelf -d 或者 objdump -f 命令)。

2. LD_LIBRARY_PATH 环境变量

从库搜索概念发展初期开始, 开发人员就希望可以使用一种临时应急的有效机制来验证他们的设计。通过特定的环境变量 (LD_LIBRARY_PATH) 就能解决我们遇到的问题。

当没有设置 rpath (DT_RPATH) 值时, 该路径就是路径搜索信息中优先级最高的。



在优先级策略中, 有关嵌入二进制文件中的值与环境变量之间的优先级问题一直以来争论不断。如果优先级策略保持不变, 一旦二进制文件中存在 rpath 字段, 由于 rpath 有更高的优先级, 因此无法在第三方软件产品中解决临时验证设计的问题。幸运的是, 新的优先级策略认为使用 rpath 这种方法并不友好, 因此使用一种临时覆盖该设置的方法来解决这个问题。而 rpath 的改进版 runpath 的优先级则比 rpath 高因此会先于 rpath 对程序产生作用, 在这种情况下, LD_LIBRARY_PATH 也可以临时地获取最

高优先级。

怎么做?

如果去掉 rpath, LD_LIBRARY_PATH 仍不起作用。

用于设置 LD_LIBRARY_PATH 的语法和设置其他环境变量的语法是一样的。我们可以按照如下所示方法, 在 shell 中输入命令来设置 LD_LIBRARY_PATH。

设置 Runpath, LD_LIBRARY_PATH 有更高优先级。

在运行时

```
$ export LD_LIBRARY_PATH=/home/milan/projects:$LD_LIBRARY_PATH
```

再强调一次，该机制只应用于实验目的。软件产品的产品版本不应该依赖于这种机制。

runpath

runpath 与 rpath 遵循相同的设计原则。在构建时，使用 ELF 二进制格式的 DT_RUNPATH 字段来指出寻找动态库的路径。与 rpath 不同的是，runpath 被设计用于支持 LD_LIBRARY_PATH 这种紧急情况下使用的需求。

设置 runpath 的方法和设置 rpath 的方法非常相似。为了传递 -R 或者 -rpath 链接器选项，需要使用额外的 `--enable-new-dtags` 链接器选项。就像我们在介绍 rpath 时提到的，但凡我们通过 gcc (或者 g++) 间接调用链接器，而不是直接调用 ld 时，根据惯例都需要在链接器选项前加上 -Wl, 前缀：

```
$ gcc -Wl,-R/home/milan/projects/ -Wl,--enable-new-dtags -lmilanlibrary
```

^ ^ ^
| | |
| | | 实际的 rpath 值将 rpath 和 runpath 设置成同一个字符串值
| | |
runpath 链接器选项

在通过 gcc 间接调用链接器，而不是直接调用 ld 时，需要加上“-Wl,” 前缀

一般来说，只要指定了 runpath，链接器均会将 rpath 和 runpath 设置成同一个值。查看二进制文件 DT_RUNPATH 值的方法是查看二进制文件的 ELF 头（比如执行 readelf -d 或者 objdump -f 命令）。

从优先级角度来看，只要 DT_RUNPATH 包含非空字符串，装载器就会忽略 DT_RPATH。这样就可以减少 rpath 带来的问题，并在需要时使用 LD_LIBRARY_PATH。

我们可以使用实用程序 `patchelf` 来改变二进制文件的 `runpath` 字段。当前无法在官方仓库中找到该工具，但是其源代码和手册页可以在 <http://nixos.org/patchelf.html> 上找到。编译成二进制文件的过程非常简单。下面的例子展示了 `patchelf` 的用法：

```
$ patchelf --set-rpath <one or more paths> <executable>
```

可以定义多个路径，
使用冒号（:）分隔

LD_LIBRARY_PATH
→ RUNPATH > RPATH

 提示 虽然在 patchelf 文档中对 rpath 有所提及，但实际上 patchelf 操作的是 runpath 字段。

3. Idconfig 缓存

一种标准的代码部署过程是基于运行 Linux 的 ldconfig 工具 (<http://linux.die.net/man/8/>)

if no rampathy:
R-PATH ① LIBRARY-PATH ②

行 Linux 的 ldconfig 工具 (<http://>)
else : LD-LIBRARY-PATH RUNPATH (②)



在文件 /etc/ld.so.conf 中引用的一些库文件可能是存储在“默认库文件路径”(trusted library paths) 中的。如果在构建可执行文件时使用了 -z nodeflib 链接器选项,那么在搜索库时操作系统默认库文件路径中的库就会被忽略。

4. 默认库文件路径 (/lib 和 /usr/lib)

路径 /lib 和 /usr/lib 是 Linux 操作系统保存动态库的两个默认路径。对于那些为超级用户权限或者所有用户设计的程序,通常需要将其动态库部署到这两个位置之一。请注意路径 /usr/local/lib 并不属于 Linux 操作系统保留路径。当然,你完全可以通过以上描述的几种机制来将该路径加入优先级列表中。



如果链接可执行文件时使用了 -z nodeflib 链接器选项,那么在搜索库时操作系统默认库文件路径中的所有库都会被忽略。

5. 优先级方案小节

总的来说,优先级方案可以归类为以下两种版本。

如果指定了 RUNPATH 字段(即 DT_RUNPATH 字段非空):

- 1) LD_LIBRARY_PATH。
- 2) runpath(DT_RUNPATH 字段)。
- 3) ld.so.cache。
- 4) 默认库路径(/lib 和 /usr/lib)。

如果没有指定 RUNPATH 字段(即 DT_RUNPATH 字段为空字符串):

1) 被加载库的 RPATH,然后是二进制文件的 RPATH,直到可执行文件或者动态库将这些库全部加载完毕为止。

- 2) LD_LIBRARY_PATH。
- 3) ld.so.cache。
- 4) 默认库路径(/lib 和 /usr/lib)。

欲了解更多有关该主题的细节,可以浏览 Linux 装载器手册页(<http://linux.die.net/man/1/ld>)。

7.3.2 Windows 运行时动态库文件的定位规则

我们可以将最为简单、常用且广泛使用的部署运行时所需 DLL 的方式归类为以下两种:

- 与应用程序二进制文件在同一目录下。
 - 系统动态链接库目录之一(比如 C:\Windows\System 或者 C:\Windows\System32)。
- 但这还不够,其实 Windows 运行时动态库搜索优先级方案要复杂得多,下面列出的一些