实现程序加载和动态链接的主要技术有:

- ***程序头部***(Program Header): 描述与程序执行直接相关的目标文件结构信息。 用来在文件中定位各个段的映像。同时包含其他一些用来为程序创 建进程映像所必需的信息。
- ***程序加载***: 给定一个目标文件，系统加载该文件到内存中，启动程序执行。
- ***动态链接***: 系统加载了程序以后，必须通过解析构成进程的目标文件之间的符号引用，以便完整地构造进程映像。

1. 段

1.1 段简介

链接器在链接可执行文件或动态库的过程中，它会把来自不同可重定位对象文件中的相同名称的 section 合并起来构成同名的 section。接着，它又会把带有相同属性(比方都是只读并可加载的)的 section 都合并成所谓 segments(段)。segments 作为链接器的输出，常被称为输出section。我们开发者可以控制哪些不同.o文件的sections来最后合并构成不同名称的 segments。如何控制呢，就是通过 linker script 来指定。关于链接器脚本，我们这里不予讨论。

一个单独的 segment 通常会包含几个不同的 sections，比方一个可被加载的、只读的segment 通常就会包括可执行代码section .text、只读的数据 section .rodata以及给动态链接器使用的符号section .dymsym等等。section 是被链接器使用的，但是 segments 是被加载器所使用的。加载器会将所需要的 segment 加载到内存空间中运行。和用 sections header table 来指定一个可重定位文件中到底有哪些 sections 一样。在一个可执行文件或者动态库中，

也需要有一种信息结构来指出包含有哪些 segments。这种信息结构就是 program header table，如ELF对象文件格式中右边的 execute view 所示的那样。

对于cronet库，我们可以用 readelf -l 来查看可执行文件的程序头表，如下所示：

```
yananhdeMacBook-Pro:armeabi-v7a yananh$ readelf -l libcronet.72.0.3626.0.so

Elf file type is DYN (Shared object file)
Entry point 0x16d000
There are 9 program headers, starting at offset 52

Program Headers:
  Type           Offset   VirtAddr   PhysAddr   FileSiz MemSiz  Flg Align
  PHDR           0x000034 0x00000034 0x00000034 0x00120 0x00120 R   0x4
  LOAD           0x000000 0x00000000 0x00000000 0x16cde0 0x16cde0 R   0x1000
  LOAD           0x16d000 0x0016d000 0x0016d000 0x37abd0 0x37abd0 R E 0x1000
  LOAD           0x4e8000 0x004e8000 0x004e8000 0x1f5bc 0x26d24 RW  0x1000
  DYNAMIC        0x5062c0 0x005062c0 0x005062c0 0x000f0 0x000f0 RW  0x4
  GNU_RELRO      0x4ea000 0x004ea000 0x004ea000 0x1d5bc 0x1e000 R   0x1
  GNU_STACK      0x000000 0x00000000 0x00000000 0x00000 0x00000 RW  0
  NOTE           0x06898c 0x0006898c 0x0006898c 0x00098 0x00098 R   0x4
  EXIDX          0x000154 0x00000154 0x00000154 0x390d0 0x390d0 R   0x4

 Section to Segment mapping:
  Segment Sections...
   00
   01     .ARM.exidx .dynsym .gnu.version .gnu.version_r .gnu.hash .hash .dynstr .rel.dyn .rel.plt .note.android.ident .ARM.extab .rod
ata
   02     .text .plt
   03     .data .data.rel.ro .init_array .fini_array .dynamic .got .got.plt .bss
   04     .dynamic
   05     .data.rel.ro .init_array .fini_array .dynamic .got .got.plt
   06
   07     .note.android.ident
   08     .ARM.exidx
```

结果显示，在可执行文件 libcronet.72.0.3626.0.so 中，总共有8个 segments。同时，该结果也很明白显示出了哪些 section 映射到哪一个 segment 当中去。比方在索引为2的那个segment 中，总共有2个 sections 映射进来，其中包括我们前面提到过的 .plt section。注意这个segment 有两个标志: R 和 E。这个表示该segment是可读的，也可执行的。如果你看到标志中有 W，那表示该segment是可写的。

上面类型为PHDR的segment，用来包含程序头表本身。

在接下来的数个 segments 中，最重要的是三个 segment：**代码段，数据段和堆栈段**。代码段和堆栈段的 VirtAddr 列的值分别为 0x16d000 和 0x4e8000。这是什么意思呢？这是说对应的段要加载在进程虚拟地址空间中的起始地址。虽然在可执行文件中规定了 text segment和 data segment 的起始地址，但是最终，在内存中的这些段的真正起始地址，却可能不是这样的，因为在动态链接器加载这些段的时候，需要考虑到页面对齐的因素。为什么？因为像x86这样的架构，它给内存单元分配读写权限的最小单位是页(page)而不是字节。也就是说，它能规定从某个页开始、连续多少页是只读的。却不能规定从某个页内的哪一个字节开始，连续多少个字节是只读的。因为x86架构中，一个page大小是4k，所以，动态链接器在加载 segment 到虚拟内存中的时候，其真实的起始地址的低12位都是零，也即以 0x1000 对齐。

我们先来看看一个真实的进程中的内存空间信息，拿我们的 test 程序作为

例子。在 Linux 系统中，有一个特殊的由内核实现的虚拟文件系统 /proc。内核实现这个文件系统，并将它作为整个Linux系统面向外部世界的一个接口。我们可以通过 /proc 观察到一个正在运行着的Linux系统的内核数据信息以及各进程相关的信息。所以我们如果要查看某一个进程的内存空间情况，也可以通过它来进行。通过命令**cat /proc/xxxx/maps**获取真实的地址，使用cronet示例，运行再Android上的map如下：

a425d000-a43ca000 r--p 00000000 103:01 294442
/data/app/org.eap.cronetapplication-1/lib/arm/libcronet.72.0.3626.0.so
a43ca000-a4745000 r-xp 0016d000 103:01 294442
/data/app/org.eap.cronetapplication-1/lib/arm/libcronet.72.0.3626.0.so
a4745000-a4747000 rw-p 004e8000 103:01 294442
/data/app/org.eap.cronetapplication-1/lib/arm/libcronet.72.0.3626.0.so
a4747000-a4765000 r--p 004ea000 103:01 294442
/data/app/org.eap.cronetapplication-1/lib/arm/libcronet.72.0.3626.0.so
a4765000-a476c000 rw-p 00000000 00:00 0
a476c000-a476d000 ---p 00000000 00:00 0
a476d000-a476e000 ---p 00000000 00:00 0
a476e000-a4870000 rw-p 00000000 00:00 0 [stack:11789]
a4870000-a4876000 rw-p 00000000 00:04 113398
/dev/ashmem/dalvik-large object space allocation (deleted)
a4876000-a4881000 rw-p 00000000 00:04 113397
/dev/ashmem/dalvik-large object space allocation (deleted)
a4881000-a4bff000 r--p 00000000 103:01 98167 /data/dalvik-cache/arm/data@app@org.eap.cronetapplication-1@base.apk@classes.dex
a4bff000-a4ecf000 r-xp 0037e000 103:01 98167 /data/dalvik-cache/arm/data@app@org.eap.cronetapplication-1@base.apk@classes.dex
a4ecf000-a4ed0000 rw-p 0064e000 103:01 98167 /data/dalvik-cache/arm/data@app@org.eap.cronetapplication-1@base.apk@classes.dex
a4ed0000-a4f0d000 r--s 0000c000 103:01 294416
/data/app/org.eap.cronetapplication-1/base.apk

从这个结果上可以看出，所有的段，其起始地址和结束地址(前面两列)都是0x1000对齐的。结果中也列出了对应的段是从哪里引过来的，比方动态链接器/system/lib/libstdc++.so等。

从程序头表中我们可以看到一个类型为 GNU_STACK 的segment，这是

stack segment。程序头表中的这一项，除了 Flg/Align 两列不为空外， 其他列都为0。这是因为堆栈段在虚拟内存空间中，从哪里开始、占多少字节是由内核说了算的，而不决定于可执行程序。实际上，内核决定把堆栈段放在整个进程地址空间的用户空间的最上面，所以堆栈段的末尾地址就是 0xc0000000。别忘记在 x86 中，堆栈是从高向低生长的。

1.2 程序头部（Program Header）

程序头部（Program Header）描述与程序执行直接相关的目标文件结构信息。用来保存“Segment”的信息, 描述了ELF文件该如何被操作系统映射到虚拟空间。因为ELF目标文件不需要被装载，所以它没有程序头表，而ELF的可执行文件和共享库文件都有。

可执行文件或者共享目标文件的程序头部是一个结构数组，每个结构描述了一个段或者系统准备程序执行所必须的其他信息。**目标文件的“段”包含一个或者多个“节区”**，也就是“段内容（Segment Contents）”。程序头部仅对可执行文件和共享目标文件有意义。

可执行目标文件在 ELF 头部的 e_phentsize 和 e_phnum 成员中给出其自身程序头部 的大小。

程序头部的数据结构如下：

```
typedef struct {
  Elf32_Word p_type;      // 段的类型，或者如何解释此数组元素的信息。我们基本上只关心Load类型的段。Load类型的常量为1
  Elf32_Off  p_offset;    // 段在文件中的偏移
  Elf32_Addr p_vaddr;     // 段的第一个字节将在进程的虚拟地址的起始位置，整个程序头中，所有"Load"类型的元素按照p_vaddr从小到大排序
  Elf32_Addr p_paddr;     // 段的物理装载地址。此成员仅用于与物理地址相关的系统中。System V忽略所有应用程序的物理地址信息。此字段对与可执行文件和共享目标 文件而言具体内容是未指定的
  Elf32_Word p_filesz;    // 段在ELF文件中所占长度，可以为0，因为可能在ELF中并不存在内容。
  Elf32_Word p_memsz;     // 段在虚拟地址空间所占的长度。可以为0。
  Elf32_Word p_flags;     // 权限属性，比如可读R，可写W，可执行X
  Elf32_Word p_align;     // 对齐属性。实际对齐字节是2的p_align次方。段在文件中和内存中如何对齐。可加载的进程段的 p_vaddr 和 p_offset 取值必须合适，相对于对页面大小的取模而言。此成员给出段在文件中和内存中如何
```

对齐。数值 0 和 1 表示不需要对齐。否则 p_align 应该是个 正整数，并且是 2 的幂次数，p_vaddr 和 p_offset 对 p_align 取模后应该相等。

} Elf32_phdr;

1.2.1 段类型

名字	取值	说明
PT_NULL	0	此数组元素未用。结构中其他成员都是未定义的
PT_LOAD	1	此数组元素给出一个可加载的段，段的大小由 p_ 件中的字节被映射到内存段开始处。如果 p_mer 节要清零。p_filesz 不能大于 p_memsz。可加载 p_vaddr 成员按升序排列。
PT_DYNAMIC	2	数组元素给出动态链接信息。
PT_INTERP	3	数组元素给出一个 NULL 结尾的字符串的位置和 器调用。这种段类型仅对与可执行文件有意义(尽 发生)。在一个文件中不能出现一次以上。如果存 有可加载段项目的前面。
PT_NOTE	4	此数组元素给出附加信息的位置和大小。
PT_SHLIB	5	此段类型被保留，不过语义未指定。包含这种类
PT_PHDR	6	此类型的数组元素如果存在，则给出了程序头部 在文件中也包括在内存中的信息。此类型的段在 且只有程序头部表是程序的内存映像的一部分时 段，则必须在所有可加载段项目的前面。
PT_LOPROC	0x70000000	此范围的类型保留给处理器专用语义。
PT_HIPROC	0x7fffffff	

1.2.2 基地址

基地址用来对程序的内存映像进行重定位。可执行文件或者共享目标文件的基地址是在执行过程中从三个数值计算的:

- 内存加载地址
- 最大页面大小
- 程序的可加载段的最低虚地址。

程序头部中的虚拟地址可能不能代表程序内存映像的实际虚地址。要计算基地址， 首先要确定与 PT_LOAD 段的最低 p_vaddr 值相关的内存地址。通过对内存地址向最接 近的最大页面大小截断，就可以得到基地址。根据要加载到内存中的文件的类型，内存地址可能与 p_vaddr 相同也可能不同。

如前所述，“.bss”节区的类型为 SHT_NOBITS。尽管它在文件中不占据空间，却会占据段的内存映像的空间。通常，这些未初始化的数据位于段的末尾，所以 p_memsz 会比 p_filesz 大。

1.3 注释节区(Note Section)：SHT_NOTE、PT_NOTE

类型为 SHT_NOTE 的节区和类型为 PT_NOTE 的节区可以用作特殊信息的存放。节区 和程序头部中的注释信息可以有任意多个条目，每个条目都是一个按目标处理器格式给 出的 4 字节字的数组。

例如：

namesz
descsz
type
name
...
desc
...

其中：

- **namesz 和 name** 注释信息的 name 部分前 namesz 字节包含一个 NULL 结尾的字符串，表示该项的属主或者发起者。没有正式的机制来避免名字冲突。如果没有名字，namesz 中包含 0。如果需要的话，可以用补零来确保描述符以 4 字节对齐。这类补齐并不包含在 namesz 中。
- **descsz 和 desc** desc 中的前 descsz 字节包含注释信息的描述。ABI 对此没有作出约束。如果需要，可以用补零来确保 4 字节边界对齐。补零的字节数不计算在 descsz 中。
- **type** 此 word 给出描述符的解释。每个发起者都要负责对自己的类型进行控制；同一类型值可以包含多种不同解释。因此程序必须能够识别名称和类型，才能理解描述符。类型取值必须非负。ABI 并不定义描述符的含义[3]。

1.4 .interp节区

在动态链接的ELF可执行文件中，有一个专门的段叫做”.interp”段。里面保存的是一个字符串，记录所需动态链接器的路径。

从下图可以看出，Android用的动态链接器是linker

```
/Developer/android-ndk-r10b/toolchains/arm-linux-a
ndroideabi-4.8/prebuilt/darwin-x86_64/bin/arm-linux-androideabi-readelf -l libwebviewchromium.so| g
rep interpreter
      [Requesting program interpreter: /system/bin/linker]
```

类型为INTERP的segment只包含一个 section，那就是 .interp。在这个section中，包含了动态链接过程中所使用的解释器路径和名称。在Linux里面，这个解释器实际上就是 /lib/ ，这可以通过下面的 hexdump 看出来：

```
[yihect@juliantec test_2]$ hexdump -s 0x114 -n 32 -C  ./test
00000114  2f 6c 69 62 2f 6c 64 2d  6c 69 6e 75 78 2e 73 6f  |/lib/ld-linux.so|
```

```
00000124  2e 32 00 00 04 00 00 00  10 00 00 00 01 00 00 00  |.2..............|
```

为什么会有这样的一个 segment？这是因为我们写的应用程序通常都需要使用动态链接库.so，就像 test 程序中所使用的 libsub.so 一样。我们还是先大致说说程序在linux里面是怎么样运行起来的吧。当你在 shell 中敲入一个命令要执行时，内核会帮我们创建一个新的进程，它在往这个新进程的进程空间里面加载进可执行程序的代码段和数据段后，也会加载进动态连接器(在Linux里面通常就是/lib/ld-linux.so 符号链接所指向的那个程序，它本省就是一个动态库)的代码段和数据。在这之后，内核将控制传递给动态链接库里面的代码。动态连接器接下来负责加载该命令应用程序所需要使用的各种动态库。加载完毕，动态连接器才将控制传递给应用程序的main函数。如此，你的应用程序才得以运行。

这里说的只是大致的应用程序启动运行过程，更详细的，我们会在后续的文章中继续讨论。我们说link editor链接的应用程序只是部分链接过的应用程序。经常的，在应用程序中，会使用很多定义在动态库中的函数。最最基础的比方C函数库(其本身就是一个动态库)中定义的函数，每个应用程序总要使用到，就像我们test程序中使用到的 printf 函数。为了使得应用程序能够正确使用动态库，动态连接器在加载动态库后，它还会做更进一步的链接，这就是所谓的动态链接。为了让动态连接器能成功的完成动态链接过程，在前面运行的link editor需要在应用程序可执行文件中生成数个特殊的 sections，比方 .dynamic、.dynsym、.got和.plt等等。

1.5 dynamic段

这个段里保存了动态链接器所需要的基本信息，比如依赖哪些共享对象、动态链接符号表的位置、动态链接重定位表的位置、共享对象初始化代码的地址等。

.dynamic段里保存的信息有点像ELF文件头。

.dynamic段的结构是由Elf32_Dyn组成的数组。

Elf32_Dyn结构由一个类型值加上一个附加的数值或指针，对于不同的类型，后面附加的数值或者指针有着不同的含义

d_tag 类型	d_un 的含义
DT_SYMTAB	动态链接符号表的地址，d_ptr 表示“.dynsym”的地址
DT_STRTAB	动态链接字符串表地址，d_ptr 表示“.dynstr”的地址
DT_STRSZ	动态链接字符串表大小，d_val 表示大小
DT_HASH	动态链接哈希表地址，d_ptr 表示“.hash”的地址
DT_SONAME	本共享对象的“SO-NAME”，我们在后面会介绍“SO-NAME”
DT_RPATH	动态链接共享对象搜索路径
DT_INIT	初始化代码地址
DT_FINIT	结束代码地址
DT_NEED	依赖的共享对象文件，d_ptr 表示所依赖的共享对象文件名
DT_REL DT_RELA	动态链接重定位表地址
DT_RELENT DT_RELAENT	动态重读位表入口数量

详见：7. dynamic segment.note

2 程序加载和进程

进程除非在执行过程中引用到相应的逻辑页面，否则不会请求真正的物理页面。进程通常会包含很多未引用的页面，因此，延迟物理读操作有效提高系统性能。要想实际获得这种效率，可执行文件和共享目标文件必须具有这样的段: **其文件偏移和虚拟地址对页面大小取模后余数相同**。

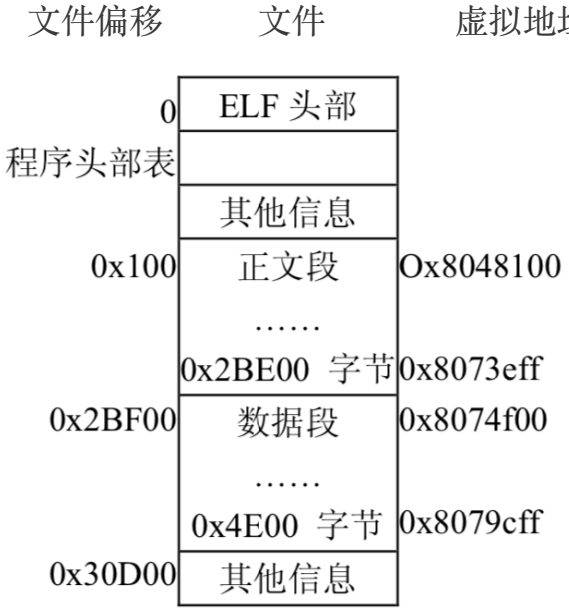


图 9 可执行文件布局示例

成员	正文段	数据段
p_type	PT_LOAD	PT_LOAD
p_offset	0X100	0x2bf00
p_vaddr	0x8048100	0x8074f00
p_paddr	未指定	未指定
p_filesz	0x2BE00	0x4e00
p_memsz	0x2BE00	0x5e24
p_flags	PF_R + PF_X	PF_R + PF_W + PF_X
p_align	0x1000	0x1000

在这个例子中，至多四个文件页面包含非纯粹的正文或者数据。

(1) 第一个页面中包含 ELF 头部、程序头部表、以及其它信息

(2) 最后一个页面包含数据开始部分的一个副本

(3) 第一数据页面包含正文段的末尾部分

(4) 最后一个数据页面可能包含与运行进程无关的文件信息

不过系统对这些页面一般会做两次映射，以保证每个段的内存访问许可是相同的。 数据段的末尾需要对未初始化数据进行特殊处理，系统应该将这些初始化为 0。

可执行文件与共享目标文件之间的段加载之间有一点不同。可执行文件的段通常包含绝对代码，为了能够让进程正确执行，所使用的段必须是构造可执行文件时所使用的 虚拟地址。因此系统会使用 p_vaddr 作为虚拟地址。另外，共享目标文件的段通常包含与位置无关的代码。这使得段的虚拟地址在不同的进程中不同，但不影响执行行为。尽管系统为每个进程选择独立的虚拟地址，仍能维持段的相对位置。因为位置独立的代码在段与段之间使用相对寻址，内存虚地址之间的差异必须与文件中虚拟地址之间的差异相匹配。

下表给出共享目标文件的针对不同进程的一种虚拟地址指定方案，说明了这种重定 位问题。

如下图为 虚拟地址指定方案示例：

源	正文	数据	基地址
文件	0x200	0x2a400	0x0
进程 1	0x80000200	0x8002a400	0x80000000
进程 2	0x80081200	0x800ab400	0x80081000
进程 3	0x900c0200	0x900ea400	0x900c0000
进程 4	0x900c6200	0x900f0400	0x900c6000

程序执行时所需要的指令和数据必需在内存中才能够正常运行。

页映射将内存和所有磁盘中的数据和指令按照“页（Page）”为单位划分成若干个页，以后所有的装载和操作的单位就是页。

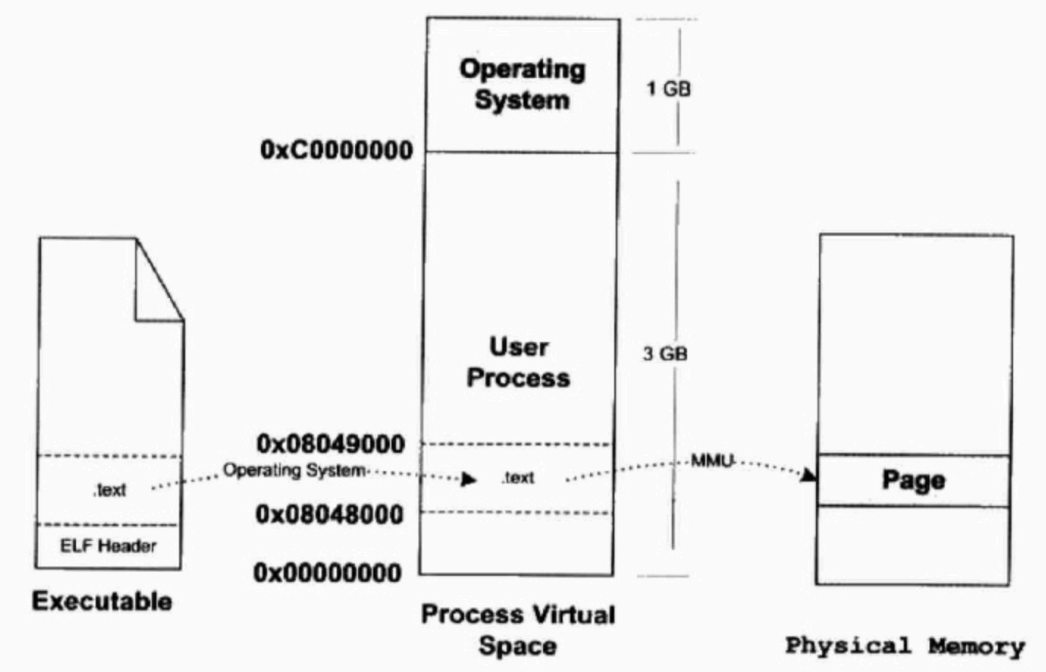
进程的建立需要做下面三件事情：

- 创建一个独立的虚拟地址空间
- 读取可执行文件头，并且建立虚拟空间与可执行文件的映射关系。
- 将CPU的指令寄存器设置成可执行文件的入口地址，启动运行。

对于第2步，当操作系统捕获到缺页错误时，它应该知道程序当前所需的页在可执行文件中的哪一个位置。

这种映射关系是保存在操作系统内部的一个数据结构VMA。

例如下图中，操作系统创建进程后，会在进程相应的数据结构中设置有一个.text段的VMA：它在虚拟空间中的地址为0x08048000~0x08049000，它对应ELF文件中偏移为0的.text，它的属性为只读，还有一些其他的属性。



2.1 页错误

在上面的例子中，程序的入口地址为0x08048000，当CPU开始打算执行这个地址的指令时，发现页面0x08048000~0x08049000（虚拟地址）是个空页面，于是它就认为这是一个页错误。CPU将控制权交给操作系统，操作系统将查询虚拟空间与可执行文件的映射关系表，找到空页面所在的VMA，计算相应的页面在可执行文件中的偏移，然后在物理内存中分配一个物理页面，将进程中该虚拟页与分配的物理页之间建立映射关系，然后把控制权再还给进程，进程从刚才页错误的位置重新开始执行。

2.2 堆和栈

对于cronet demo中，通过命令**cat /proc/xxxx/maps**获取真实的地址，使用cronet示例，运行再Android上的map如下：

```
a425d000-a43ca000  r--p 00000000 103:01 294442
/data/app/org.eap.cronetapplication-1/lib/arm/libcronet.72.0.3626.0.so
a43ca000-a4745000 r-xp 0016d000 103:01 294442
/data/app/org.eap.cronetapplication-1/lib/arm/libcronet.72.0.3626.0.so
a4745000-a4747000 rw-p 004e8000 103:01 294442
/data/app/org.eap.cronetapplication-1/lib/arm/libcronet.72.0.3626.0.so
a4747000-a4765000 r--p 004ea000 103:01 294442
/data/app/org.eap.cronetapplication-1/lib/arm/libcronet.72.0.3626.0.so
a4765000-a476c000 rw-p 00000000 00:00 0
```

a476c000-a476d000 ---p 00000000 00:00 0
a476d000-a476e000 ---p 00000000 00:00 0
a476e000-a4870000 rw-p 00000000 00:00 0 [stack:11789]
a4870000-a4876000 rw-p 00000000 00:04 113398 /dev/ashmem/dalvik-large object space allocation (deleted)
a4876000-a4881000 rw-p 00000000 00:04 113397 /dev/ashmem/dalvik-large object space allocation (deleted)
a4881000-a4bff000 r--p 00000000 103:01 98167 /data/dalvik-cache/arm/data@app@org.eap.cronetapplication-1@base.apk@classes.dex
a4bff000-a4ecf000 r-xp 0037e000 103:01 98167 /data/dalvik-cache/arm/data@app@org.eap.cronetapplication-1@base.apk@classes.dex
a4ecf000-a4ed0000 rw-p 0064e000 103:01 98167 /data/dalvik-cache/arm/data@app@org.eap.cronetapplication-1@base.apk@classes.dex
a4ed0000-a4f0d000 r--s 0000c000 103:01 294416 /data/app/org.eap.cronetapplication-1/base.apk

第一列是VMA的地址范围；

第二列是VMA的权限，“r”表示可读，“w”表示可写，“x”表示可执行，“p”表示私有（COW，Copy on Write），“s”表示共享；

第三列是偏移，表示VMA对应得段在映像中的偏移；

第四列表示映像文件所在设备的主设备号和次设备号；

第五列表示映像文件的节点号；

最后一列是映像文件的路径；

我们可以看到进程中有5个VMA, 只有蓝色是映射到可执行文件中的两个Segment，红色段的文件所在设备主设备号及文件节点号都是0，则表示他们没有映射到文件中，这种VMA叫做匿名虚拟内存区域。另外有一个很特殊的VMA叫“vdso”，它的地址已经位于内核空间了（即大于0xC0000000的地址），事实上它是一个内核的模块，进程可以通过访问这个VMA来跟内核进行一些通信。

操作系统通过给进程空间划分出一个个VMA来管理进程的虚拟空间；基本原则是将相同权限属性的、有相同映像文件的映射成一个VMA。

3 动态链接

3.1 程序解释器

可执行文件可以包含 PT_INTERP 程序头部元素。在 exec() 期间，系统从

PT_INTERP 段中检索路径名，并从解释器文件的段创建初始的进程映像。也就是说， 系统并不使用原来可执行文件的段映像，而是为解释器构造一个内存映像。接下来是解 释器从系统接收控制，为应用程序提供执行环境。

解释器可以有两种方式接受控制。

- 接受一个文件描述符，读取可执行文件并将其映射到内存中
- 根据可执行文件的格式，系统可能把可执行文件加载到内存中，而不是为解释器提供一个已经打开的文件描述符。

解释器可以是一个可执行文件，也可以是一个共享目标文件。

共享目标文件被加载到内存中时，其地址可能在各个进程中呈现不同的取值。系统在 mmap 以及相关服务所使用的动态段区域创建共享目标文件的段。因此，共享目标解释器通常不会与原来的可执行文件的原始段地址发生冲突。

可执行文件被加载到内存中固定地址，系统使用来自其程序头部表的虚拟地址创建各个段。因此，可执行文件解释器的虚拟地址可能会与原来的可执行文件的虚拟地址发生冲突。解释器要负责解决这种冲突。

3.2 动态加载程序

在构造使用动态链接技术的可执行文件时，连接编辑器向可执行文件中添加一个类型为 PT_INTERP 的程序头部元素，告诉系统要把动态链接器激活，作为程序解释器。系统所提供的动态链接器的位置是和处理器相关的。

Exec() 和动态链接器合作，为程序创建进程映像，其中包括以下动作:

(1). 将可执行文件的内存段添加到进程映像中;

(2). 把共享目标内存段添加到进程映像中;

(3). 为可执行文件和它的共享目标执行重定位操作;

(4). 关闭用来读入可执行文件的文件描述符，如果动态链接程序收到过这样的文件描述符的话;

(5). 将控制转交给程序，使得程序好像从 exec 直接得到控制。

链接编辑器也会构造很多数据来协助动态链接器处理可执行文件和共享目标文件。

这些数据包含在可加载段中，在执行过程中可用。如:

- **类型为 SHT_DYNAMIC 的 .dynamic 节区包含很多数据。位于节区头部的结构保存 了其他动态链接信息的地址。linker在加载时通过此段获取所有的section**
- **类型为 SHT_HASH 的 .hash 节区包含符号哈希表。**
- **类型为 SHT_PROGBITS 的 .got 和 .plt 节区包含两个不同的表: 全局偏移**

表和过程链接表。

因为任何符合 ABI 规范的程序都要从共享目标库中导入基本的系统服务，动态链接器会参与每个符合 ABI 规范的程序的执行。

3.3 动态节区

如果一个目标文件参与动态链接，它的程序头部表将包含类型为 PT_DYNAMIC 的元素。此“段”包含.dynamic 节区。该节区采用一个特殊符号 _DYNAMIC 来标记，其中包 含如下结构的数组。

```
typedef struct {
  Elf32_Sword d_tag;
  union {
    Elf32_Word  d_val;
    Elf32_Addr  d_ptr;
  } d_un;
} Elf32_Dyn;
extern Elf32_Dyn  _DYNAMIC[];
```

对每个这种类型的对象，d_tag 控制 d_un 的解释含义:

- d_val 此 Elf32_Word 对象表示一个整数值，可以有多种解释。
- d_ptr 此 Elf32_Addr 对象代表程序的虚拟地址。如前所述，文件的 虚拟地址可能与执行过程中的内存虚地址不匹配。在解释包含于动态结构中的地址时，动态链接程序基于原来文件值和内存基地址计算实际地址。为了保持一致性，文件中不包含用来“纠正”动态结构中重定位项地址的重定位项目。

下面的表格总结了可执行文件和共享目标文件对标志的要求。如果标志被标记为 “必需”，那么符合 ABI 规范的文件的动态链接数组必须包含一个该类型表项。“可选” 意味着该标志可以出现，但不是必需的。

名称	数值	d_un	可执行	共享目标	说明
DT_NULL	0	忽略	必需	必需	标记为 DT_NULL 的项
DT_NEEDED	1	d_val	可选	可选	此元素包含一个 NULL 的名称。所使用的字符是指在该表中的下标。顺序很重要，尽管他们
DT_PLTRELSZ	2	d_val	可选	可选	此元素给出了与过程链 DT_JMPREL 类型的条
DT_PLTGOT	3	d_ptr	可选	可选	此元素给出一个与过程

DT_HASH	4	d_ptr	必需	必需	此元素包含符号哈希表表。
DT_STRTAB	5	d_ptr	必需	必需	此元素包含字符串表的
DT_SYMTAB	6	d_ptr	必需	必需	此元素包含符号表的地Elf32_Sym 类型。
DT_RELA	7	d_ptr	必需	必需	此元素包含重定位表的Elf32_Rela。目 标文件构造重定位表时，连接件 中这些节区保持相互可执行文件创建进程 映位表，并执行相关的动DT_RELAENT 元素。如或者 DT_REL 都 可能存
DT_RELASZ	8	d_val	必需	可选	此元素包含 DT_RELA 重
DT_RELAENT	9	d_val	必需	可选	此元素包含 DT_RELA 重
DT_STRSZ	10	d_val	必需	必需	此元素给出字符串表的
DT_SYMENT	11	d_val	必需	必需	此元素给出符号表项的
DT_INIT	12	d_ptr	可选	可选	此元素包含初始化函数
DT_FINI	13	d_ptr	可选	可选	此元素包含结束函数(T
DT_SONAME	14	d_val	忽略	可选	此元素给出一个 NULL 称。该偏移实际 上是 D
DT_RPATH	15	d_val	可选	忽略	此元素包含 NULL 结尾径。该偏移实际上 是 D
DT_SYMBOLIC	16	忽略	忽略	可选	此元素出现于某个共享符号解析算法。动 态链开始搜索。如果共享目可执 行文件和其他共享
DT_REL	17	d_ptr	必需	可选	此元素与 DT_RELA 类似Elf32_Rel。如 果文件中DT_RELENT 元素。
DT_RELSZ	18	d_val	必需	可选	此元素包含 DT_REL 重
DT_RELENT	19	d_val	必需	可选	此元素包含 DT_REL 重
DT_PLTREL	20	d_val	可选	可选	此成员给出过程链接表DT_REL 或者 DT_RELA式。
DT_DEBUG	21	d_ptr	可选	忽略	此成员用于调试。ABI
DT_TEXTREL	22	忽略	可选	可选	如果文件中不包含此成改，正如程序头部表 中求对不可写段进行修改
DT_JMPREL	23	d_ptr	可选	可选	如果存在这种成员，则重定位项仅与过 程链接时忽略它们，当然后期DT_PLTREL 必须也存在
DT_LOPROC	0x70000000	未指定	未指定	未指定	这个范围的表项，包括

DT_HIPROC	0x7fffffff	未指定	未指定	未指定	的。

注:

- 没有出现在此表中的标记值是保留的。
- 除了数组末尾的 DT_NULL 元素以及 DT_NEEDED 元素的相对顺序约束以外，其他项目可以以任意顺序出现。

3.4 共享目标的依赖关系

在动态链接器为某个目标文件创建内存段时，依赖关系(记录于动态结构的 DT_NEEDED 表项中)能够提供需要哪些目标来提供程序服务的信息。通过不断地将被引用的共享目标与他们的依赖之间建立连接，动态链接器构造出完整的进程映像。

在解析符号引用时，动态链接程序使用宽度优先算法检查符号表。就是说首先检查可执行程序自身的符号表，然后检查 DT_NEEDED 条目(按顺序) 的符号表，接着在第二级 DT_NEEDED 条目上搜索。共享目标文件必须对进程而言可读，不需要其他权限。

即使某个共享目标在依赖表中出现多次，动态链接器也仅会对其连接一次。

存在于依赖表中的名称或者是 DT_SONAME 字符串的副本，或者是用来构造目标文件的共享目标的路径名称。例如，如果连接编辑器在构造某个可执行文件时使用的一个共享目标中包含 lib1 的 DT_SONAME 项，并且使用了路径名为 /usr/llib/lib2 的共享目标库，那么可执行文件将在其依赖表中包含 lib1 和 /usr/lib/lib2。

如果共享目标的名称中包含一个或者多个斜线(/)，那么动态链接器将使用该字符串作为路径名称。如果名称中没有斜线，则对共享目标的路径搜索按如下顺序进行:

- 动态数组标记 DT_RPATH 中可能包含若干字符串，这些字符串用“:”分隔， 用来通知动态链接器从哪里开始搜索。默认情况下最后搜索当前目录。
- 进程环境中可能包含一个名为 LD_LIBRARY_PATH 的变量，其中也包含若干 用“:”分隔的路径名。变量可以以“;”结尾。所有 LD_LIBRARY_PATH 都在 DT_RPATH 之后被搜索。尽管某些程序对分号前后的列表的处理有所不同， 动态链接程序并不这样，它能够接受分号，语义如上。

- 最后，如果上面两组目录搜索都失败，未能找到所需要的库，则对/usr/lib 进行 搜索。

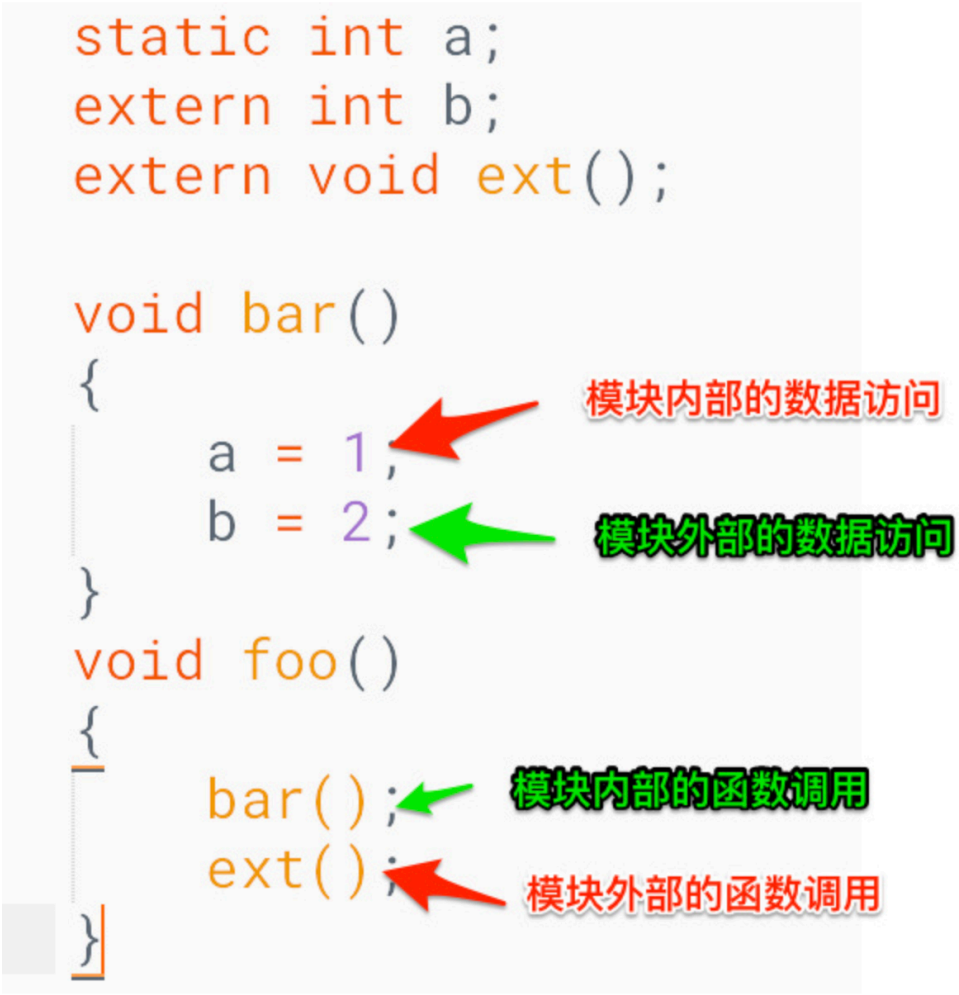
注意:

出于安全性考虑，动态链接器会针对 SUID 或者 SGID 的程序忽略环境搜索规范(如 LD_LIBRARY_PATH)。不过仍然会搜索 DT_RPATH 和/usr/lib 路径。

4 地址无关代码(PIC)

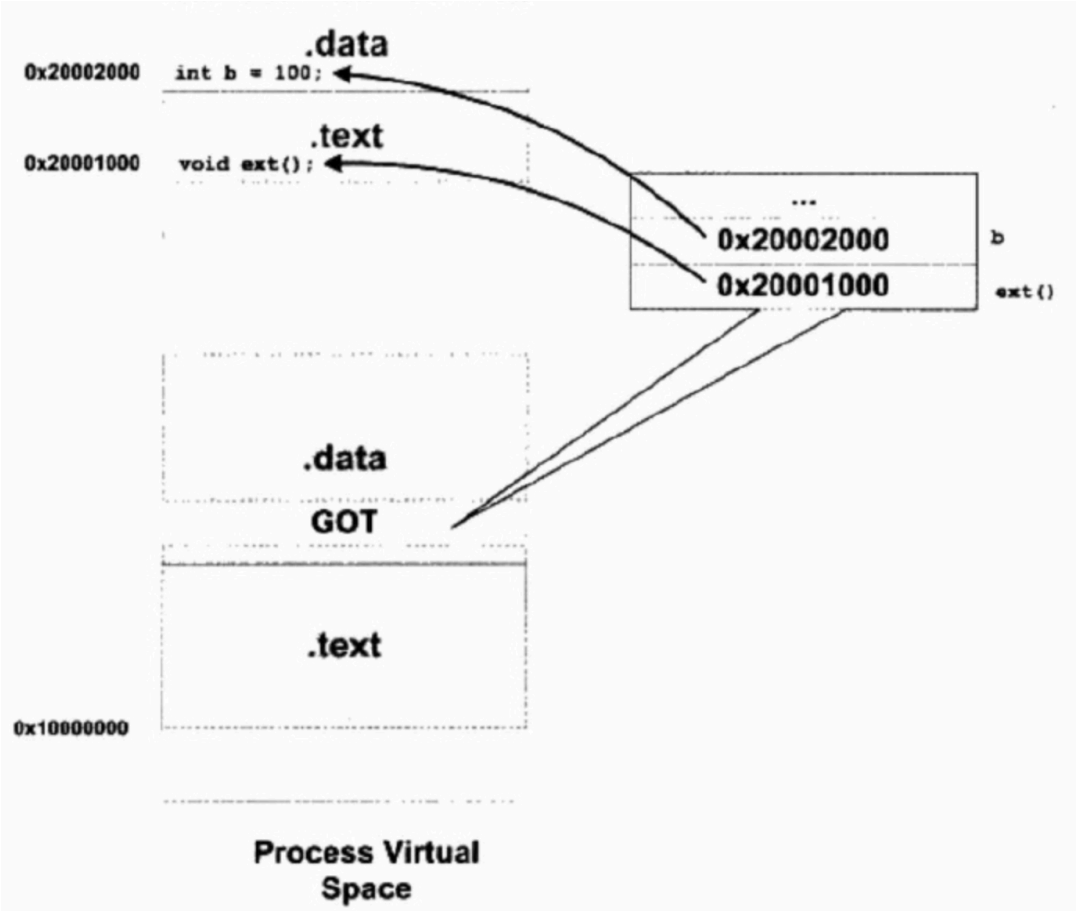
装载时重定位是解决动态模块中有绝对地址引用的方法之一，但是它有一个很大的缺点是指令部分无法在多个进程之间共享，这样就失去了动态链接节省内存的一大优势。我们还需要有一种更好的方法解决共享对象指令中对绝对地址的重定位问题。其实我们的目的很简单，希望程序模块中共享的指令部分在装载时不需要因为装载地址的改变而改变，所以实现的基本思想就是把指令中那些需要被修改的部分分离出来，跟数据部分放在一起，这样指令部分就可以保持不变，而数据部分可以在每个进程中拥有一个副本。

模块中各种类型的地址引用方式如下图:



5 全局偏移表(GOT)

用于模块间数据访问，在数据段里建立一个指向外部模块变量的指针数组。当代码需要引用该全局变量时，可以通过GOT中相对用的项间接引用，它的基本机制如下图。



当指令中需要访问变量b时，程序会先找到GOT，然后根据GOT中变量所对应的项找到变量的目标地址。每个变量都对应一个4字节的地址，链接器在装载模块的时候会查找每个变量所在的地址，然后填充GOT中的各个项，以确保每个指针所指向的地址正确。由于GOT本身是放在数据段的，所以它可以在模块装载时被修改，并且每个进程都可以有独立的副本，相互不受影响。

如果程序需要直接访问某个符号的绝对地址，那么该符号就会具有一个全局偏移表项。由于可执行文件和共享目标具有独立的全局偏移表，一个符号的地址可能出现在多个表中。动态链接器在将控制交给进程映像中任何代码之前，要处理所有的全局偏移表重定位，因而确保了执行过程中绝对地址信息可用。

表项 0 是保留的，用来存放动态结构的地址，可以用符号_DYNAMIC 引用之。这样， 类似动态链接器这种程序能够在尚未处理其重定位项的时候先找到自己的动态结构。对于动态链接器而言这点很重要，因为它必须能够在不依赖其他程序来对其内存映像进行重定位的前提下，初始化自己。在 32 位 Intel 体系结构下，全局偏移表中的表项 1 和 2 也是保留的。

系统可能在不同的程序中为相同的共享目标选择不同的内存段地址，甚至为统一程序的两次执行选择不同的库地址。尽管如此，一旦进程映像被建立起来，

内存段不会改 变其地址。只有进程存在，其内存段都位于固定的虚地址。

全局偏移表的格式和解释都是和处理器相关的。对于 32 位 Intel 体系结构而言， 符号 _GLOBAL_OFFSET_TABLE_ 可以用来访问该表。

```
extern Elf32_Addr _GLOBAL_OFFSET_TABLE[];
```

符号 _GLOBAL_OFFSET_TABLE_ 可能存在于 .got 节区的中间，允许使用负的/ 非负的下标来访问地址数组。

6 过程链接表(PLT)

动态链接下对于全局和静态的数据访问都要进行复杂的GOT定位，然后间接寻址；对于模块间的调用也要先定位GOT，然后再进行间接跳转。程序开始执行时，动态链接器都要进行一次链接工作，会寻找并装载所需的共享对象，然后进行符号查找地址重定位等工作，如此一来，程序的运行速度必定会减慢。

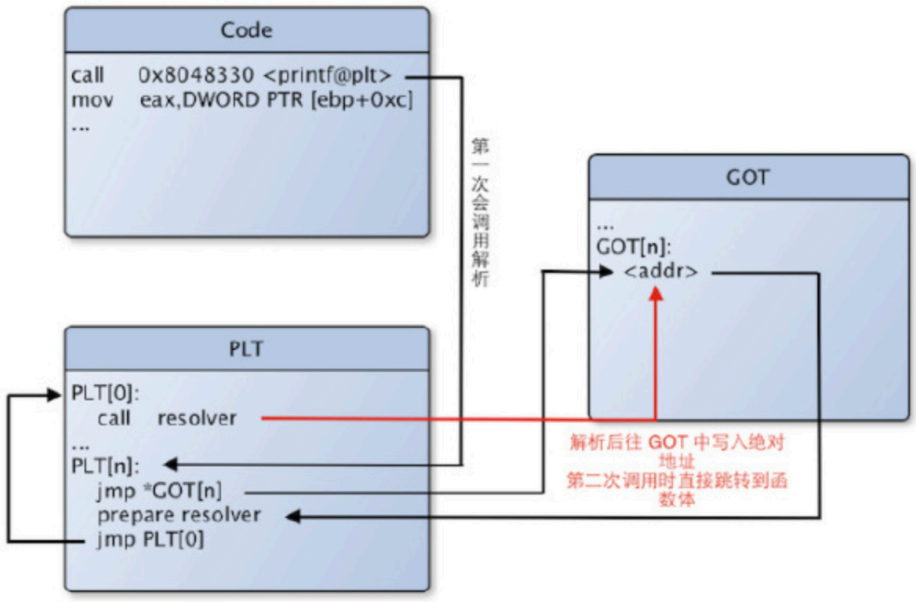
全局、静态、模块间的数据访问都要进行GOT定位。

延迟绑定的实现：函数第一次被用到时才进行绑定（符号查找、重定位等），如果没有用到则不进行绑定。

GOT 位于 .got.plt section 中，而 PLT 位于 .plt section中。

GOT 保存了程序中所要调用的函数的地址，运行一开时其表项为空，会在运行时实时的更新表项。一个符号调用在第一次时会解析出绝对地址更新到 GOT 中，第二次调用时就直接找到 GOT 表项所存储的函数地址直接调用了。

printf（）函数的调用过程如下图



全局偏移表(GOT)用来将位置隔离的地址计算重定向到绝对位置，与此相

似， 过程链接表(PLT)能够把位置隔离的函数调用重定向到绝对位置。

链接编辑器物能解析从一个可执行文件/共享目标到另一个可执行文件/共享目标控制转移(例如函数调用)。因此，链接编辑器让程序把控制转移给过程链接表中的表项。在 System V 中，过程链接表位于共享正文中，不过它们使用位于私有的全局偏移表中的地址。

动态链接器能够确定目标处的绝对地址，并据此修改全局偏移表的内存映像。动态链接器因此能够对表项进行重定位，并且不会影响程序代码的位置独立性和可共享性。 可执行文件和共享目标文件拥有各自独立的过程链接表。

如下图为绝对过程链接表：

```
.PLT0: pushl got_plus_4
       jmp   *got_plus_8
       nop; nop
       nop; nop
.PLT1:  jmp *name1_in_GOT
       pushl $offset@PC
.PLT2:  jmp *name2_in_GOT
       pushl $offset
       jmp   .PLT0@PC
       ...
```

如下图为位置独立的过程链接表：

```
.PLT0: pushl 4(%ebx)
       jmp   *8(%ebx)
       nop;   nop
       nop;   nop
.PLT1:  jmp *name1@GOT(%ebx)
       pushl $offset
       jmp .PLT0@PC
.PLT2:  jmp *name2@GOT(%ebx)
       pushl $offset
       jmp   .PLT0@PC
       ...
```

如图所示，过程链接表命令针对绝对代码和位置独立的代码使用不同的操作数寻址模式。尽管如此，它们对动态链接器的接口还是相同的。

动态链接器和程序“合作”，通过过程链接表和全局偏移表解析符号引用：

1. 在第一次创建程序的内存映像时，动态链接器为全局偏移表的第二和第三项设置特殊值。

2. 如果过程链接表是位置独立的，全局偏移表必须位于 %bx 中，进程映像中的每 个共享目标文件都有自己的过程链接表，控制向过程链接表项的传递仅发生在同 一个目标文件中。因此，调用函数用负责在调用过程链接表项之前设置全局偏移 表的基址寄存器。

3. 出于说明的目的，假定程序调用了 name1，name1 将控制传输给标号 .PLT1。

4. 第一条指令跳转到 name1 的全局偏移表项的地址。最初，全局偏移表中包含后 面的 pushl 指令的地址，而不是 name1 的真实地址。

5. 接下来，程序将重定位偏移(offset)压栈。重定位偏移是一个 32 位非负数， 是在重定位表中的字节偏移量。指定的重定位表项的类型为 R_386_JMP_SLOT， 其偏移将给出在前面的 jmp 指令中使用的 GOT 表项。重定位项也包含一个符号 表索引，借以告诉动态链接器被引用的符号是什么，在这里是 name1。

6. 在将重定位偏移压栈后，程序会跳转到 .PLT0，也就是过程链接表的第一项。 pushl 指令把第二个全局偏移表项(got_plus_4 或者 4(%ebx))压入堆栈，因 而为动态链接器提供了识别信息的机会。程序然后跳转到第三个 GOT 表项内保 存的地址(got_plus_8 或者 8(%ebx))，后者将控制传递给动态链接器。

7. 当动态链接器得到控制后，它恢复堆栈，查看指定的重定位项，寻找符号的值， 将 name1 的“真实”地址存储于全局偏移表项中，并将控制传递给期望的目的地。

8. 过程链接表项的后续执行将把控制直接传递给 name1，不会再次调用动态链接 器。就是说 .PLT1 处的 jmp 将控制传递给 name1，而不会执行后面的 pushl 指令。

环境变量 LD_BIND_NOW 可以更改动态链接行为。如果其取值非空，动态链接器会在 控制传递给程序之前，对过程链接表项进行计算。就是说动态链接器会在进程初始化的 过程中处理类型为 R_386_JMP_SLOT 的重定位项。否则，动态链接器会对过程链接表实 行懒惰计算，延迟符号解析和重定位，直到某个表项的第一次执行。

懒惰绑定通常会提供整体的应用性能，因为未使用的符号不会引入额外的动态链接开销。尽管如此，有些应用情形会使得懒惰绑定不太合适。首先，对共享目标函数的第 一次引用花的时间会超出后续调用，因为动态链接器要截获调用

以便解析符号。一些应用不能容忍这种不可预测性。第二，如果发生了错误，动态链接器无法解析某个符号， 动态链接器会终止程序。在懒惰绑定下，这类事情可能会发生任意多次。某些应用也可能无法容忍这种不可预测性。通过关闭懒惰绑定，动态链接器会迫使所有错误都发生在进程初始化期间，而不是应用程序接收控制以后。

7 哈希表(Hash Table)

由于动态链接下，需要在程序运行时查找符号，为了加快符号的查找过程，往往还有辅助的符号好戏表。

用readelf查看elf文件的动态符号表及它的哈希表。

```
[yananhdeMacBook-Pro:armeabi-v7a yananh$ readelf -sD libtest.so

Symbol table for image:
  Num Buc:    Value  Size   Type   Bind Vis      Ndx Name
   13   0: 00003004     0 NOTYPE  GLOBAL DEFAULT ABS _edata
   10   0: 00000000     0 NOTYPE  WEAK   DEFAULT UND __cxa_begin_cleanup
    9   0: 00000000     0 FUNC    GLOBAL DEFAULT UND memcpy
    5   0: 000005ed    60 FUNC    GLOBAL DEFAULT  12 say_hello
   15   1: 00003004     0 NOTYPE  GLOBAL DEFAULT ABS _end
   12   1: 00000000     0 NOTYPE  WEAK   DEFAULT UND __cxa_call_unexpected
   11   1: 00000000     0 NOTYPE  WEAK   DEFAULT UND __cxa_type_match
    2   1: 00000000     0 FUNC    GLOBAL DEFAULT UND __cxa_atexit
   14   2: 00003004     0 NOTYPE  GLOBAL DEFAULT ABS __bss_start
    8   2: 00000000     0 FUNC    GLOBAL DEFAULT UND abort
    7   2: 00000000     0 FUNC    WEAK   DEFAULT UND __gnu_Unwind_Find_exidx
    6   2: 00000000     0 FUNC    GLOBAL DEFAULT UND snprintf
    4   2: 00000000     0 FUNC    GLOBAL DEFAULT UND printf
    3   2: 00000000     0 FUNC    GLOBAL DEFAULT UND malloc
    1   2: 00000000     0 FUNC    GLOBAL DEFAULT UND __cxa_finalize
```

7.1 哈希函数：elf_hash

ELF 实现中常用的哈希函数如下，有时候会作一些优化(比如 Linux)。

```
unsigned long elf_hash (const unsigned char *name) {
    unsigned long h = 0, g;
    while (*name) {
        h=(h<<4)+*name++;
        if (g = h & 0xf0000000)
            h^=g>>24; h&=~g;
    }
    return h;
}
```

8 初始化和终止函数

在动态链接器构造了进程映像，并执行了重定位以后，每个共享的目标都获

得执行某些初始化代码的机会。这些初始化函数的被调用顺序是不一定的，不过所有共享目标初始化都会在可执行文件得到控制之前发生。

类似地，共享目标也包含终止函数，这些函数在进程完成终止动作序列时，通过 atexit() 机制执行。动态链接器对终止函数的调用顺序是不确定的。

共享目标通过动态结构中的 DT_INIT 和 DT_FINI 条目指定初始化/终止函数。通常这些代码放在.init 和.fini 节区中。

注意:

尽管 atexit() 终止处理通常会被执行，在进程消亡时并不能保证被执行。特别地，如果进程调用了 _exit 或者进程因为收到某个它既 未捕捉又未忽略的信号而终止时，不会执行终止处理