yuanya的专栏

Mac OS X应用程序格式详解

分类： Mac OS 2014-09-13 21:53 200人阅读 评论(0) 收藏 举报

mac os x

OS X应用程序格式详解

OS X 如何执行应用程序
译者：51test2003 译自http://0xfe.blogspot.com/2006/03 ... s-applications.html
作为长期的 UNIX 用户, 我通常有一些排除系统故障的工具. 最近, 我正在开发软件并新增了Apple's OS X 系统支持; 但是和其他传统UNIX 变种不同, OS X 不支持许多与加载，链接和执行程序相关的工具.

例如, 当共享库重定位出错时, 我所做的首要事情就是对可执行文件运行ldd. ldd工具列出了可执行文件所依赖的共享库（包括所在路径）。但是在OS X , 试图运行ldd将报错.
evil:~ mohit$ ldd /bin/ls
-bash: ldd: command not found

没找到? 但在所有的UNIX上基本上都有的啊. 我想知道objdump是否可用.
$ objdump -x /bin/ls
-bash: objdump: command not found

命令未找到. 怎么回事?

问题在于与Linux, Solaris, HP-UX, 和其他许多UNIX 变种不同, OS X 不使用 ELF二进制文件. 另外, OS X 不属于GNU项目的一部分。该项目包含想ldd和objdump这样的工具.

为了在OS X获得可执行文件所依赖的共享库列表，需要使用 otool 工具.
evil:~ mohit$ otool /bin/ls
otool: one of -fahlLtdoOrTMRIHScis must be specified
Usage: otool [-fahlLDtdorSTMRIHvVcXm] object_file ...
-f print the fat headers
-a print the archive header
-h print the mach header
-l print the load commands
-L print shared libraries used
-D print shared library id name
-t print the text section (disassemble with -v)
-p start dissassemble from routine name
-s print contents of section
-d print the data section
-o print the Objective-C segment
-r print the relocation entries

-S print the table of contents of a library
-T print the table of contents of a dynamic shared library
-M print the module table of a dynamic shared library
-R print the reference table of a dynamic shared library
-I print the indirect symbol table
-H print the two-level hints table
-v print verbosely (symbolicly) when possible
-V print disassembled operands symbolicly
-c print argument strings of a core file
-X print no leading addresses or headers
-m don't use archive(member) syntax

evil:~ mohit$ otool -L /bin/ls
/bin/ls:
/usr/lib/libncurses.5.4.dylib (compatibility version 5.4.0, current version 5.4.0)
/usr/lib/libSystem.B.dylib (compatibility version 1.0.0, current version 88.0.0)

好多了. 我们可以看见/bin/ls引用了两个动态库. 尽管, 文件扩展名我们根本不熟悉.

我相信许多UNIX / Linux 用户使用OS X系统时有类似的经历，所以我决定写一点目前我所知道的关于 OS X 可执行文件的知识.

OS X 运行时架构运行时环境是OS X上代码扩展的一个框架。它一组定义代码如何被加载，被管理，被执行的集合组成。一旦应用程序运行, 合适的运行时环境就加载程序到内存, 解决外部库的引用, 并为执行准备代码.

OS X 支持三种运行时环境:
dyld 运行时环境:基于 dyld库管理器的推荐环境.
CFM 运行时环境: OS 9遗留环境. 实际用来设计需要使用 OS X新特色, 但还没完全移植到dyld的应用程序.
The Classic环境: OS 9 (9.1 or 9.2) 程序无需修改直接在OS X运行.
本文主要关注于Dyld 运行时环境.

Mach-O 可执行文件格式在 OS X, 几乎所有的包含可执行代码的文件，如：应用程序、框架、库、内核扩展……, 都是以Mach-O文件实现. Mach-O 是一种文件格式，也是一种描述可执行文件如何被内核加载并运行的ABI (应用程序二进制接口). 专业一点讲, 它告诉系统:
使用哪个动态库加载器
加载哪个共享库.
如何组织进程地址空间.
函数入口点地址，等.
Mach-O 不是新事物. 最初由开放软件基金会 (OSF) 用于设计基于 Mach 微内核OSF/1 操作系统. 后来移植到 x86 系统OpenStep.

为了支持Dyld 运行时环境, 所有文件应该编译成Mach-O 可执行文件格式.

Mach-O 文件的组织

Mach-O 文件分为三个区域: 头部、载入命令区Section和原始段数据. 头部和载入命令区描述文件功能、布局和其他特性；原始段数据包含由载入命令引用的字节序列。为了研究和检查 Mach-O 文件的各部分, OS X 自带了一个很有用的程序otool，其位于/usr/bin目录下.

接下来, 将使用 otool来了解 Mach-O 文件如何组织的.

头部查看文件的 Mach-O头部, 使用otool 命令的 -h参数
evil:~ mohit$ otool -h /bin/ls
/bin/ls:
Mach header

```
      magic cputype cpusubtype   filetype ncmds sizeofcmds      flags
0xfeedface      18          0          2    11       1608 0x00000085
```

头部首先指定的是魔数（magic number）. 魔数标明文件是32位还是64位的Mach-O 文件. 也标明 CPU字节顺序. 魔数的解释,参看/usr/include/mach-o/loader.h.

头部也指定文件的目标架构. 这样就允许内核确保该代码不会在不是为此处理器编写的CPU上运行。例如, 在上面的输出, cputype 设成18, 它代表CPU_TYPE_POWERPC, 在 /usr/include/mach/machine.h中定义.

从上两项信息，我们推断出此二进制文件用于32-位基于PowerPC 的系统.

有时二进制文件可能包含不止一个体系的代码。通常称为Universal Binaries, 通常以 fat_header这额外的头部开始。检查 fat_header内容, 使用otool命令的 -f开关参数.

cpusubtype 属性制定了CPU确切模型, 通常设成CPU_SUBTYPE_POWERPC_ALL 或 CPU_SUBTYPE_I386_ALL.

filetype 指出文件如何对齐如何使用。实际上它告诉你文件是库、静态可执行文件、core file等。上面的 filetype等于MH_EXECUTE, 指出demand paged executable file. 下面是从/usr/include/mach-o/loader.h截取的片段，列出了不同的文件类型。

```
#define MH_OBJECT 0x1   /* relocatable object file */
#define MH_EXECUTE  0x2   /* demand paged executable file */
#define MH_FVMLIB 0x3   /* fixed VM shared library file */
#define MH_CORE   0x4   /* core file */
#define MH_PRELOAD  0x5   /* preloaded executable file */
#define MH_DYLIB  0x6   /* dynamically bound shared library */
#define MH_DYLINKER 0x7   /* dynamic link editor */
#define MH_BUNDLE 0x8   /* dynamically bound bundle file */
#define MH_DYLIB_STUB 0x9   /* shared library stub for static */
         /*  linking only, no section contents */
```

接下来的两个属性涉及到载入命令区段, 指定了命令的数目和大小.

最后, 获得了状态信息, 这些可能在装载和执行时被内核使用。
载入命令载入命令区段包含一个告知内核如何载入文件中的各个原始段的命令列表。典型的描述如何对齐，保护每个段及各段在内存中的布局.

查看文件中的载入命令列表, 使用otool 命令的 -l开关参数.

```
evil:~/Temp mohit$ otool -l /bin/ls
/bin/ls:
Load command 0
      cmd LC_SEGMENT
  cmdsize 56
  segname __PAGEZERO
   vmaddr 0x00000000
   vmsize 0x00001000
  fileoff 0
filesize 0
  maxprot 0x00000000
initprot 0x00000000
   nsects 0
    flags 0x4
Load command 1
      cmd LC_SEGMENT
  cmdsize 600
```

```
  segname __TEXT
   vmaddr 0x00001000
   vmsize 0x00006000
  fileoff 0
filesize 24576
  maxprot 0x00000007
initprot 0x00000005
   nsects 8
    flags 0x0
Section
  sectname __text
   segname __TEXT
      addr 0x00001ac4
      size 0x000046e8
    offset 2756
     align 2^2 (4)
    reloff 0
    nreloc 0
     flags 0x80000400
reserved1 0
reserved2 0

[ ___SNIPPED FOR BREVITY___ ]

Load command 4
          cmd LC_LOAD_DYLINKER
      cmdsize 28
         name /usr/lib/dyld (offset 12)
Load command 5
          cmd LC_LOAD_DYLIB
      cmdsize 56
         name /usr/lib/libncurses.5.4.dylib (offset 24)
   time stamp 1111407638 Mon Mar 21 07:20:38 2005
      current version 5.4.0
compatibility version 5.4.0
Load command 6
          cmd LC_LOAD_DYLIB
      cmdsize 52
         name /usr/lib/libSystem.B.dylib (offset 24)
   time stamp 1111407267 Mon Mar 21 07:14:27 2005
      current version 88.0.0
compatibility version 1.0.0
Load command 7
     cmd LC_SYMTAB
cmdsize 24
  symoff 28672
   nsyms 101
  stroff 31020
strsize 1440
Load command 8
            cmd LC_DYSYMTAB
        cmdsize 80
      ilocalsym 0
      nlocalsym 0
```

```
     iextdefsym 0
     nextdefsym 18
      iundefsym 18
      nundefsym 83
         tocoff 0
           ntoc 0
      modtaboff 0
        nmodtab 0
   extrefsymoff 0
    nextrefsyms 0
indirectsymoff 30216
  nindirectsyms 201
      extreloff 0
        nextrel 0
      locreloff 0
        nlocrel 0
Load command 9
     cmd LC_TWOLEVEL_HINTS
cmdsize 16
  offset 29884
  nhints 83
Load command 10
        cmd LC_UNIXTHREAD
    cmdsize 176      flavor PPC_THREAD_STATE
      count PPC_THREAD_STATE_COUNT
    r0  0x00000000 r1  0x00000000 r2  0x00000000 r3   0x00000000 r4   0x00000000
    r5  0x00000000 r6  0x00000000 r7  0x00000000 r8   0x00000000 r9   0x00000000
    r10 0x00000000 r11 0x00000000 r12 0x00000000 r13  0x00000000 r14  0x00000000
    r15 0x00000000 r16 0x00000000 r17 0x00000000 r18  0x00000000 r19  0x00000000
    r20 0x00000000 r21 0x00000000 r22 0x00000000 r23  0x00000000 r24  0x00000000
    r25 0x00000000 r26 0x00000000 r27 0x00000000 r28  0x00000000 r29  0x00000000
    r30 0x00000000 r31 0x00000000 cr  0x00000000 xer  0x00000000 lr   0x00000000
    ctr 0x00000000 mq  0x00000000 vrsave 0x00000000 srr0 0x00001ac4 srr1 0x00000000
```

上面的文件在头部下有11 加载命令直接定位, 从 0 到 10.

前四个命令(LC_SEGMENT), 从 0 到 3, 定义了文件中的段如何映射到内存中去。段定义了Mach-O binary 二进制文件中的字节序列, 可以包含零个或更多的 sections. 稍候我们谈谈段。

Load command 4 (LC_LOAD_DYLINKER) 指定使用哪个动态链接器. 几乎总是设成OS X默认动态链接器 /usr/lib/dyld。

Commands 5 and 6 (LC_LOAD_DYLIB) 指定文件需要链接的共享库。它们由command 4规定的动态链接器载入。

Commands 7 and 8 (LC_SYMTAB, LC_DYNSYMTAB) 指定由文件和动态链接器分别使用的符号表. Command 9 (LC_TWOLEVEL_HINTS) 包含两级名称空间的hint table。最后, command 10 (LC_UNIXTHREAD), 定义进程主线程的初始状态. 该命令仅仅包含在可执行文件里。

Segments and Sections

上面涉及到的大多数加载命令都引用了文件中的段. 段是Mach-O文件直接被内核和动态链接器映射到虚拟内存中的一系列字符序列. 头部和加载命令区域认为是文件的首段。一个典型的 OS X 可执行文件通常由下列五段：：
__PAGEZERO：定位于虚拟地址0，无任何保护权利。此段在文件中不占用空间，访问NULL导致立即崩溃.
__TEXT：包含只读数据和可执行代码.

__DATA：包含可写数据. 这些 section通常由内核标志为copy-on-write .
__OBJC：包含Objective C 语言运行时环境使用的数据。
__LINKEDIT :包含动态链接器用的原始数据.
__TEXT和 __DATA段可能包含0或更多的section. 每个section由指定类型的数据, 如, 可执行代码, 常量, C 字符串等组成.

查看某section内容, 使用otool命令 -s选项.

```
evil:~/Temp mohit$ otool -sv __TEXT __cstring /bin/ls
/bin/ls:
Contents of (__TEXT,__cstring) section
00006320 00000000 5f5f6479 6c645f6d 6f645f74
00006330 65726d5f 66756e63 73000000 5f5f6479
00006340 6c645f6d 616b655f 64656c61 7965645f
00006350 6d6f6475 6c655f69 6e697469 616c697a
__SNIP__
```

反汇编__text section, 使用 the -tv 开关参数.

```
evil:~/Temp mohit$ otool -tv /bin/ls
/bin/ls:
(__TEXT,__text) section
00001ac4        or      r26,r1,r1
00001ac8        addi    r1,r1,0xfffc
00001acc        rlwinm  r1,r1,0,0,26
00001ad0        li      r0,0x0
00001ad4        stw     r0,0x0(r1)
00001ad8        stwu    r1,0xffc0(r1)
00001adc        lwz     r3,0x0(r26)
00001ae0        addi    r4,r26,0x4
__SNIP__
```

在 __TEXT段里, 存在四个主要的 section:
__text：编译后的机器码。
__const：通用常量数据.
__cstring：字面量字符串常量.
__picsymbol_stub：动态链接器使用的位置无关码stub 路由.
这样保持了可执行的和不可执行的代码在段里的明显隔离.

运行应用程序既然知道了Mach-O 文件的格式, 接下来看看OS X 如何载入并运行应用程序的。运行应用程序时, shell首先调用fork()系统调用. fork 创建调用进程(shell) 逻辑拷贝并准备好执行. 子进程然后调用execve()系统调用，当然需要提供要执行的程序路径.

内核载入指定的文件, 检查其头部验证是否是合法的Mach-O 文件. 然后开始解释载入命令，将子进程地址空间替换成文件中的各段。同时,内核也执行有二进制文件指定的动态链接器, 着手加载、链接所有依赖库。在绑定了运行所必备的各个符号后，调用entry-point 函数.

在build应用程序时entry-point 函数通常从/usr/lib/crt1.o静态链接（标准函数）. 此函数初始化内核环境，调用可执行文件的main()函数.

应用程序现在运行了.

动态链接器

OS X 动态链接器/usr/lib/dyld, 负责加载依赖的共享库, 导入变量符号和函数，与当前进程的绑定。进程首次运行时, 链接器所做的就是把共享库导入到进程地址空间。取决于程序的build方式, 实际绑定也足执行不同的方式。

载入后立即绑定—— load-time绑定.
当符号引用时—— just-in-time绑定.
预绑定
如未指定绑定类型, 使用 just-in-time绑定.

应用程序仅仅当所有需要的符号和段从不同的目标文件解决是才能继续运行。为了寻找库和框架, 标准动态链接器/usr/bin/dyld, 将搜索预定义的目录集合. 要修改目录, 或提供回滚路径, 可以设置DYLD_LIBRARY_PATH或DYLD_FALLBACK_LIBRARY_PATH环境变量

上一篇 Linking and Install Names
下一篇 使用Sparkle为OS X App添加自动更新功能

暂无评论

您还没有登录,请[登录]或[注册]

* 以上用户言论只代表其个人观点，不代表CSDN网站的观点或立场