深入理解 debuginfo

2015/12/5
renyl

1 关于 debuginfo 的疑惑

程序员应该都知道，为了能够使用 gdb 跟踪程序，需要在编译期使用 gcc 的-g 选项。而对于系统库或是 Linux 内核，使用 gdb 调试或使用 systemtap 探测时，还需要安装相应的 debuginfo 包。

例如 glibc 及它的 debuginfo 包为：

```
[allen@fedora t]$ rpm -qa | grep glibc
glibc-2.18-12.fc20.x86_64
glibc-debuginfo-2.18-12.fc20.x86_64
...
```

于是我不禁有如下这些疑问：

- glibc-debuginfo 中包含了什么信息？
- glibc-debuginfo 是如何创建出来的？
- gdb 或 systemtap，是如何把 glibc 与 glibc-debuginfo 关联起来的？

本文将通过一些例子，来解答这些问题。

2 debuginfo 中包含了什么信息？

让我们来看看 glibc-debuginfo 中，包含有什么内容：

```
[allen@fedora t]$ rpm -ql glibc-debuginfo-2.18-12.fc20.x86_64
/usr/lib/debug
/usr/lib/debug/.build-id
/usr/lib/debug/.build-id/00
/usr/lib/debug/.build-id/00/a32f1b9405f5fcd41a7618f3c2c895ee4aab09
/usr/lib/debug/.build-id/00/a32f1b9405f5fcd41a7618f3c2c895ee4aab09.debug
…
/usr/lib/debug/lib64/libthread_db.so.1.debug
/usr/lib/debug/lib64/libutil-2.18.so.debug
/usr/lib/debug/lib64/libc-2.18.so.debug
…
/usr/src/debug/glibc-2.18/wcsmbs/wcwidth.h
/usr/src/debug/glibc-2.18/wcsmbs/wmemcmp.c
...
```

由上可见，glibc-debuginfo 大致有三类文件：

- 存放在/usr/lib/debug/下的.build-id/nn/nnn...nnn.debug 文件，文件名是 hash key。

- 存放在/usr/lib/debug/下的其它*.debug 文件，其文件名是库文件名+.debug 后缀。
- glibc 的源代

当使用 gdb 调试时，需要在机器码与源代码之间，建立起映射关系。这就需要三个信息：

- 机器码：可执行文件、动态链接库，例如：/lib64/libc-2.18.so
- 源代码：显然就是 glibc-debuginfo 中，包含的*.c 和*.h 等源文件。
- 映射关系：你应该猜到了，它们就保存在*.debug 文件中。

3 debuginfo 是如何创建出来的？

当我们使用 gcc 的-g 选项编译程序时，机器码与源代码的映射关系，会被默认地与可执行程序、动态链接库合并在一起。

例如下面 a.out 可执行程序，已经包含了映射关系：

```
[allen@fedora t]$ nl main.c
        1   #include <stdio.h>

        2   int main()
        3   {
        4       printf("hello, world!\n");
        5       return 0;
        6   }
[allen@fedora t]$ gcc -g main.c
[allen@fedora t]$ ls -l
total 16
-rwxrwxr-x 1 allen allen   9502 Apr   9 14:55 a.out
-rw-rw-r-- 1 allen allen     76 Apr   9 14:49 main.c
```

把映射关系等调试信息，与可执行文件、动态链接库合并在一起，会带来一个显著的问题：可执行文件或库的 size 变得很大。这对于那些不关心调试信息的普通用户是不必要的。

例如，Linux 的内核，如果带上 debuginfo，会无谓的增加几百 M 的大小。如果一个 Linux 操作系统的所有库都带上各自的 debuginfo，那么光是一个干净的操作系统，就需要浪费掉几 G 甚至十几 G 的磁盘空间。正是了为解决这个问题，在 Linux 上的各种程序和库，在生成 RPM 时，就已经把 debuginfo 单独的抽取出来，因此形成了独立的 debuginfo 包。

问题是，如何让程序生成分离的 debuginfo 呢？我们可以通过 objcopy 命令的--only-keep-debug 选项来实现，下面的命令把调试信息从 a.out 中读取出来，写到 a.out.debug 文件中：

```
[allen@fedora t]$ objcopy --only-keep-debug ./a.out a.out.debug
[allen@fedora t]$ ls -l
```

```
total 24
-rwxrwxr-x 1 allen allen   9502 Apr   9 14:55 a.out
-rwxrwxr-x 1 allen allen   6022 Apr   9 15:22 a.out.debug
-rw-rw-r-- 1 allen allen     76 Apr   9 14:49 main.c
```

既然已经把调试信息保存到了 a.out.debug 文件中，就可以通过 objcopy 的--strip-debug 选项给 a.out 瘦身了（也可以使用 strip --strip-debug ./a.out，效果一样）：

```
[allen@fedora t]$ objcopy --strip-debug ./a.out
[allen@fedora t]$ ls -l
total 24
-rwxrwxr-x 1 allen allen   8388 Apr   9 15:27 a.out
-rwxrwxr-x 1 allen allen   6022 Apr   9 15:22 a.out.debug
-rw-rw-r-- 1 allen allen     76 Apr   9 14:49 main.c
```

当把调试信息从 a.out 中清除后，使用 gdb 对 a.out 进行调试，会报 no debugging symbols found：

```
[allen@fedora t]$ gdb ./a.out
GNU gdb (GDB) Fedora 7.6.50.20130731-19.fc20
...
Reading symbols from /home/allen/t/a.out...(no debugging symbols found)...done.
(gdb)
```

显然，gdb 找不到调试信息了。因此，我们需要在 a.out 中埋下一些线索，以便 gdb 借助这些线索，可以正确地查找到它对应的 debug 文件：a.out.debug。

在 Linux 下，可执行文件或库，通常是 ELF(Executable and Linkable Format)格式。这种格式含有 section headers。而调试信息的线索，正好可以通过一个约定的 section header 来保存，它叫.gnu_debuglink。可通过 objcopy 的--add-gnu-debuglink 选项，把调试信息的文件名（a.out.debug）保存到 a.out 的.gnu_debuglink 这个 header 中,然后 gdb 就可以正常调试了：

```
[allen@fedora t]$ objcopy --add-gnu-debuglink=a.out.debug ./a.out
[allen@fedora t]$ objdump -s -j .gnu_debuglink ./a.out
./a.out:         file format elf64-x86-64

Contents of section .gnu_debuglink:
0000 612e6f75 742e6465 62756700 3fe5803b    a.out.debug.?..;
[allen@fedora t]$ gdb a.out
...
Reading symbols from /home/allen/t/a.out...Reading symbols from
/home/allen/t/a.out.debug...done.
```

上面的 objcopy 命令，其实是把 a.out.debug 的文件名以及这个文件的 CRC 校验码，写到了.gnu_debuglink 这个 header 的值中，但是并没有告诉 a.out.debug 所在的路径（上面通过 objdump 命令，可以打印出.gnu_debuglink 这个 header 的内容）。

那么 gdb 是按照怎样的规则，去查找 a.out.debug 文件呢？在解答这个问题之前，我们先来看另一个 section header，叫.note.gnu.build-id：

```
[allen@fedora t]$ readelf -t ./a.out | grep build-id
[ 3] .note.gnu.build-id
[allen@fedora t]$ readelf -n ./a.out
...
Notes at offset 0x00000274 with length 0x00000024:
  Owner                                Data size            Description
  GNU                                  0x00000014           NT_GNU_BUILD_ID
(unique build ID bitstring)
Build ID: 888010ffb999590e7158422ea813169be34085a1

[allen@fedora t]$ readelf -n ./a.out.debug
...
Notes at offset 0x00000274 with length 0x00000024:
  Owner                                Data size            Description
  GNU                                  0x00000014           NT_GNU_BUILD_ID
(unique build ID bitstring)
Build ID: 888010ffb999590e7158422ea813169be34085a1
```

这个 section header 是 a.out 原生就存在的，因此也被拷贝到了 a.out.debug 中。这个 header，保存了一个 Build ID，这个 ID 是根据 a.out 文件自动计算出来的，每个执行文件或库，都有它唯一的 Build ID。

在第 2 节中，我们注意到这种文件：.build-id/nn/nnnn...nnnn.debug，前两个“nn”就是它的 Build ID 前两位，后面的 nnnn...nnnn 则是 Build ID 的剩余部分。而这个 nnnn...nnnn.debug 文件，只是改了个名字而已。

而 gdb 则是通过下面的顺序查找 a.out.debug 文件：

1) <global debug directory>/.build-id/nn/nnnn...nnnn.a.out.debug
2) <the path of a.out>/a.out.debug
3) <the path of a.out>/.debug/a.out.debug
4) <global debug directory>/<the patch of a.out>/a.out.debug

其中，<global debug directory>默认为/usr/lib/debug/。 可以在 gdb 中，通过 set/show debug-file-directory 命令来设置或查看这个值：

```
[allen@fedora t]$ gdb ./a.out
...
(gdb) show debug-file-directory
The directory where separate debug symbols are searched for is "/usr/lib/debug".
```

既然 a.out 的 Build ID 为：888010ffb999590e7158422ea813169be34085a1，可以把 a.out.debug 文件移动到/usr/lib/debug/.build-id/*目录下：

```
[allen@fedora t]$ sudo cp a.out.debug \
/usr/lib/debug/.build-id/88/8010ffb999590e7158422ea813169be34085a1.debug
```

```
[allen@fedora t]$ gdb ./a.out
...
Reading symbols from /home/allen/t/a.out...Reading symbols from
/usr/lib/debug/.build-id/88/8010ffb999590e7158422ea813169be34085a1.debug...done.
done.
```

由上可见，gdb 就会优先从/usr/lib/debug/.build-id/查找到对应的 debug 信息。

4 a.out.debug 里有什么内容？

gcc 目前会默认会采用 DWARF 4 格式来保存调试信息。可以通过 readelf -w 来查看 DWARF 的内容：

```
[allen@fedora t]$ readelf -w ./a.out.debug
...
Contents of the .debug_info section:

  Compilation Unit @ offset 0x0:
    Length:                0x8d (32-bit)
    Version:               4
    Abbrev Offset: 0x0
    Pointer Size:   8
<0><b>: Abbrev Number: 1 (DW_TAG_compile_unit)
   <c>   DW_AT_producer       : (indirect string, offset: 0x6a): GNU C 4.8.2
20131212 (Red Hat 4.8.2-7) -mtune=generic -march=x86-64 -g
   <10>     DW_AT_language    : 1               (ANSI C)
   <11>     DW_AT_name        : (indirect string, offset: 0x2f): main.c
   <15>     DW_AT_comp_dir    : (indirect string, offset: 0x5b): /home/allen/t
...
```

DWARF 内部通过 DIE(Debugging Information Entry)，形成一颗调用树，DWARF 在设计的时候，就考虑到了各种语言的支持，虽然它通常与 ELF 格式的文件一起工作，但它其实并不依赖 ELF。

由于 DWARF 比较自由的设计，使它不仅支持 C/C++，也支持 Java/Python 等等几乎所有语言的调试信息的表达。

在 DWARF 里，通常包含：源代码与机器码的映射关系的行号表、宏信息、inline 函数的信息、Call Frame 信息等。

但对于普通用户，通常不需要了解 DWARF 的太多细节，如果好奇的话，推荐阅读文献 5。

5 参考文献

1) http://sourceware.org/gdb/onlinedocs/gdb/Separate-Debug-Files.html
2) http://sourceware.org/binutils/docs-2.17/binutils/objcopy.html
3) https://blogs.oracle.com/dbx/entry/gnu_debuglink_or_debugging_system
4) https://blogs.oracle.com/dbx/entry/creating_separate_debug_info
5) http://dwarfstd.org/doc/DWARF4.pdf