深入理解debuginfo

@Chinainvent

一、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：

```
[yunkai@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。

二、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：

```
[yunkai@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/wmemchr.c
/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的源代码

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。

三、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：

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

     2  int main()
     3  {
     4    printf("hello, world!\n");
     5    return 0;
     6  }
[yunkai@fedora t]$ gcc -g main.c
[yunkai@fedora t]$ ls -l
total 16
-rwxrwxr-x 1 yunkai yunkai 9502 Apr  9 14:55 a.out
-rw-rw-r-- 1 yunkai yunkai   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。如果是通过网络安装，还将浪费所有用户的带宽，并显著的拖慢安装的进度。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：

```
[yunkai@fedora t]$ objcopy --only-keep-debug ./a.out a.out.debug
[yunkai@fedora t]$ ls -l
total 24
-rwxrwxr-x 1 yunkai yunkai 9502 Apr  9 14:55 a.out
-rwxrwxr-x 1 yunkai yunkai 6022 Apr  9 15:22 a.out.debug
```

```
-rw-rw-r-- 1 yunkai yunkai   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）：

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

```
[yunkai@fedora t]$ gdb ./a.out
GNU gdb (GDB) Fedora 7.6.50.20130731-19.fc20
...
Reading symbols from /home/yunkai/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session headers。而调试信息的线索，正好可以通过一个约定的session 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：

```
[yunkai@fedora t]$ objcopy --add-gnu-debuglink=a.out.debug ./a.out
[yunkai@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.?..;
[yunkai@fedora t]$ gdb a.out
...
Reading symbols from /home/yunkai/t/a.out...Reading symbols from
/home/yunkai/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session header，叫.note.gnu.build-id：

```
[yunkai@fedora t]$ readelf -t ./a.out | grep build-id
  [ 3] .note.gnu.build-id
[yunkai@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

[yunkai@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session 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

而<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：

```
[yunkai@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：8888010ffb999590e7158422ea813169be34085a1，可以把a.out.debug文件，移动到/usr/lib/debug/.build-id/88/8010ffb999590e7158422ea813169be34085a1.debug：

```
[yunkai@fedora t]$ sudo cp a.out.debug \
/usr/lib/debug/.build-id/88/8010ffb999590e7158422ea813169be34085a1.debug
[yunkai@fedora t]$ gdb ./a.out
...
Reading symbols from /home/yunkai/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。

四、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：

```
[yunkai@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/yunkai/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。

五、在代码中生成Marker探针

通过gcc的-g选项，所有函数名，都会自动的生成相应的debuginfo，供systemtap进行探测，这种方法在英文上称为：Debuginfo-based instrumentation，它的局限性在于，只能收集到函数调用的初始时刻、以及函数返回的结束时刻的上下文信息。

为了解决这个问题，又提出了一种新方法：Compiled-in instrumentation，它可以让程序员，把探针安插到指定的某行代码中，从而可以收集到那行代码执行时的上下文信息，这种探针被称为Marker探针。

编写Marker探针，需要在代码中包含头文件：

```
#include <sys/sdt.h>
```

然后在目标行，插入下面的Marker宏之一：

```
DTRACE_PROBE(provider, name)
DTRACE_PROBE4(provider, name, arg1, arg2, arg3, arg4)
```

写好Marker探针并成功编译后，可以使用下面的systemtap指令来查看Marker探针是否生效：

```
stap -L 'process("/path/to/a.out").mark("*")'
```

更具体的操作方法详见文献6，值得一提的是，Marker探针是非常轻量的，它几乎对程序的性能没有影响，因为它只会在代码中生成nop汇编指令。它是通过把现场的上下文信息，保存在ELF文件的特定的section header(.stapsdt.base)来实现的，只会增加debuginfo文件的大小。

六、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
6. https://sourceware.org/systemtap/wiki/AddingUserSpaceProbingToApps