http://geek.csdn.net/news/detail/195824
如果你手头上有一个你自己或者别人开发的程序，但它有一些 bug。或者你只是想知道这个程序是如何工作的。怎么办呢？你需要一个调试工具。

现在很少有人会直接对着汇编指令进行调试，通常情况下，大家都希望能对照着源代码进行调试。但是，你调试使用的主机，一般来说并不是构建程序的那台，因此你会看到如下这个令人沮丧的消息：

```
$ gdb -q python3.7

Reading symbols from python3.7...done.

(gdb) l

6   ./Programs/python.c: No such file or directory.
```

我经常会看到这些报错信息，并且对于调试程序来说，这也非常重要。所以，我认为我们需要详细了解一下 GDB 是如何在调试会话中显示源代码的。

调试信息

首先，我们从调试信息开始。调试信息是由编译器生成的存在于二进制文件中的特殊段，供调试器和其他相关的工具使用。

在 GCC 中，有一个著名的`-g` 标志用于生成调试信息。大多数使用某种构建系统的项目都会在构建时默认包含或者通过一些标志来添加调试信息。

例如，在 CPython 中，你需要执行以下命令：

```
$ ./configure --with-pydebug

$ make -j
```

`-with-pydebug` 会在调用 GCC 时添加`-g` 选项。

这个`-g` 选项会生成二进制的调试段，并插入到程序的二进制文件中。调试段通常采用 DWARF 格式。对于 ELF 二进制文件来说，调试段的名称一般都是像`.debug_ *`这样的，例如 `.debug_info` 或者`.debug_loc`。这些调试段使得调试程序成为可能，可以这么说，它是汇编级别的指令与源代码之间的映射。

要查看程序是否包含调试符号，你可以使用 `objdump` 命令列出二进制文件的所有段：

```
$ objdump -h ./python


python:     file format elf64-x86-64
```

```
Sections:
Idx Name          Size      VMA               LMA               File off  Algn
  0 .interp       0000001c  0000000000400238  0000000000400238  00000238  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .note.ABI-tag 00000020  0000000000400254  0000000000400254  00000254  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
...
 25 .bss          00031f70  00000000008d9e00  00000000008d9e00  002d9dfe  2**5
                  ALLOC
 26 .comment      00000058  0000000000000000  0000000000000000  002d9dfe  2**0
                  CONTENTS, READONLY
 27 .debug_aranges 000017f0  0000000000000000  0000000000000000  002d9e56  2**0
                  CONTENTS, READONLY, DEBUGGING
 28 .debug_info   00377bac  0000000000000000  0000000000000000  002db646  2**0
                  CONTENTS, READONLY, DEBUGGING
 29 .debug_abbrev 0001fcd7  0000000000000000  0000000000000000  006531f2  2**0
                  CONTENTS, READONLY, DEBUGGING
 30 .debug_line   0008b441  0000000000000000  0000000000000000  00672ec9  2**0
                  CONTENTS, READONLY, DEBUGGING
 31 .debug_str    00031f18  0000000000000000  0000000000000000  006fe30a  2**0
                  CONTENTS, READONLY, DEBUGGING
 32 .debug_loc    0034190c  0000000000000000  0000000000000000  00730222  2**0
```

```
                CONTENTS, READONLY, DEBUGGING
 33 .debug_ranges 00062e10  0000000000000000  0000000000000000  00a71b2e  2**0
                CONTENTS, READONLY, DEBUGGING
```

或者使用 `readelf` 命令:

```
$ readelf -S ./python
There are 38 section headers, starting at offset 0xb41840:


Section Headers:
  [Nr] Name              Type            Address          Offset
       Size              EntSize         Flags  Link  Info  Align
  [ 0]                   NULL            0000000000000000  00000000  0000000000000000 0000000000000000           0     0     0
  [ 1] .interp           PROGBITS        0000000000400238  00000238  000000000000001c  0000000000000000   A       0     0     1


...


  [26] .bss              NOBITS          00000000008d9e00  002d9dfe
       0000000000031f70  0000000000000000  WA       0     0     32
  [27] .comment          PROGBITS        0000000000000000  002d9dfe
       0000000000000058  0000000000000001  MS       0     0     1
  [28] .debug_aranges    PROGBITS        0000000000000000  002d9e56
       00000000000017f0  0000000000000000           0     0     1
```

```
  [29] .debug_info       PROGBITS        0000000000000000  0
02db646
       0000000000377bac  0000000000000000           0     0
  1
  [30] .debug_abbrev     PROGBITS        0000000000000000
006531f2
       000000000001fcd7  0000000000000000           0     0
  1
  [31] .debug_line       PROGBITS        0000000000000000  0
0672ec9
       000000000008b441  0000000000000000           0     0
  1
  [32] .debug_str        PROGBITS        0000000000000000  0
06fe30a
       0000000000031f18  0000000000000001  MS       0     0
  1
  [33] .debug_loc        PROGBITS        0000000000000000  0
0730222
       000000000034190c  0000000000000000           0     0
  1
  [34] .debug_ranges     PROGBITS        0000000000000000
00a71b2e
       0000000000062e10  0000000000000000           0     0
  1
  [35] .shstrtab         STRTAB          0000000000000000  0
0b416d5
       0000000000000165  0000000000000000           0     0
  1
  [36] .symtab           SYMTAB          0000000000000000  00
ad4940
       000000000003f978  0000000000000018          37   8762
    8
  [37] .strtab           STRTAB          0000000000000000  00
b142b8
       000000000002d41d  0000000000000000           0     0
  1
```

```
Key to Flags:
  W (write), A (alloc), X (execute), M (merge), S (strings),
 l (large)
  I (info), L (link order), G (group), T (TLS), E (exclude),
 x (unknown)
  O (extra OS processing required) o (OS specific), p (proc
essor specific)
```

在我们刚刚编译的 Python 程序中，我们可以看到.debug_ *段，因此它是包含调试信息的。

调试信息是 DIE（调试信息条目）的一个集合。每个 DIE 都有一个标签，用来表示 DIE 的类型以及它的属性，就像变量的名称和行号一样。

GDB 如何寻找源代码

为了寻找源代码，GDB 会解析.debug_info 段并查找所有带有 DW_TAG_compile_unit 标签的 DIE。具有此标签的 DIE 有两个主要属性 DW_AT_comp_dir（编译目录）和 DW_AT_name（名称），这就是源代码的路径。把这两个属性结合起来就是某个特定编译单元（对象文件）对应的源文件的完整路径。

要解析调试信息，你仍然可以使用 objdump 命令:

```
$ objdump -g ./python | vim -
```

你可以看到这些解析出来的调试信息：

```
Contents of the .debug_info section:


  Compilation Unit @ offset 0x0:
   Length:        0x222d (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: 0xb6
b): GNU C99 6.3.1 20161221 (Red Hat 6.3.1-1) -mtune=generic
 -march=x86-64 -g -Og -std=c99
```

```
   <10>   DW_AT_language    : 12   (ANSI C99)
   <11>   DW_AT_name        : (indirect string, offset: 0x10ec): ./Programs/python.c
   <15>   DW_AT_comp_dir    : (indirect string, offset: 0x7a): /home/avd/dev/cpython
   <19>   DW_AT_low_pc      : 0x41d2f6
   <21>   DW_AT_high_pc     : 0x1b3
   <29>   DW_AT_stmt_list   : 0x0
```

GDB 是这样读取的：地址从 `DW_AT_low_pc = 0×41d2f6` 到 `DW_AT_low_pc + DW_AT_high_pc = 0×41d2f6 + 0×1b3 = 0×41d4a9` 对应的源代码文件是位于 `/home/avd/dev/cpython` 路径下的`./Programs/python.c` 文件，相当简单吧。

这是 GDB 向你显示源代码的整个过程：

- 解析 `.debug_info` 查找当前对象文件的 `DW_AT_name` 属性的 `DW_AT_comp_dir` 属性
- 按照路径 `DW_AT_comp_dir/DW_AT_name` 打开文件
- 显示文件的内容

如何告诉 GDB 源代码的位置

所以，要解决`./Programs/python.c: No such file or directory.`这个问题，我们必须在目标主机上存放源代码（复制或 `git clone` 过来），并执行以下任意一个操作：

1. 重建源代码路径

   你可以在目标主机上重建源代码路径，这样，GDB 就能找到对应的源代码了。这是个愚蠢的办法，但是还是很有用的。

   在我这个例子中，我执行了这个命令 `git clone https://github.com/python/cpython.git /home/avd/dev/cpython` 来检出所需的版本。

2. 修改 GDB 源代码路径

   你可以在调试会话中使用 `directory <dir>`命令让 GDB 关联正确的源代码路径：

   ```
   (gdb) list
   6   ./Programs/python.c: No such file or directory.
   ```

```
(gdb) directory /usr/src/python
Source directories searched: /usr/src/python:$cdir:$cwd
(gdb) list
6   #ifdef __FreeBSD__
7   #include <fenv.h>
8   #endif
9
10  #ifdef MS_WINDOWS
11  int
12  wmain(int argc, wchar_t **argv)
13  {
14      return Py_Main(argc, argv);
15  }
```

3. 设置 GDB 路径替换规则

   如果目录结构层次比较复杂，有时候添加源代码路径是不够的。在这种情况下，你可以使用 `set substitute-path` 命令来添加源路径的替换规则。

```
(gdb) list
6   ./Programs/python.c: No such file or directory.
(gdb) set substitute-path /home/avd/dev/cpython /usr/src/python
(gdb) list
6   #ifdef __FreeBSD__
7   #include <fenv.h>
8   #endif
9
10  #ifdef MS_WINDOWS
11  int
12  wmain(int argc, wchar_t **argv)
13  {
```

```
14      return Py_Main(argc, argv);
15  }
```

4. 把二进制文件移到源代码目录

   你可以通过将二进制文件移动到源代码目录来改变 GDB 源代码路径。

   ```
   mv python /home/user/sources/cpython
   ```

   因为 GDB 会试着在当前目录（`$cwd`）下寻找源代码，所以这个做法也是可以的。

5. 编译时增加`-fdebug-prefix-map`选项

   你可以使用`-fdebug-prefix-map = old_path = new_path` 编译选项来替代构建阶段的源路径。下面是在 CPython 项目中执行此操作的例子：

   ```
   $ make distclean    # start clean
   $ ./configure CFLAGS="-fdebug-prefix-map=$(pwd)=/usr/src/python" --with-pydebug
   $ make -j
   ```

   这样，我们就有了新的源代码路径：

   ```
   $ objdump -g ./python
   ...
    <0><b>: Abbrev Number: 1 (DW_TAG_compile_unit)
       <c>   DW_AT_producer    : (indirect string, offset: 0xb65): GNU C99 6.3.1 20161221 (Red Hat 6.3.1-1) -mtune=generic -march=x86-64 -g -Og -std=c99
       <10>   DW_AT_language    : 12      (ANSI C99)
       <11>   DW_AT_name        : (indirect string, offset: 0x10ff): ./Programs/python.c
       <15>   DW_AT_comp_dir    : (indirect string, offset: 0x558): /usr/src/python
       <19>   DW_AT_low_pc      : 0x41d336
       <21>   DW_AT_high_pc     : 0x1b3
       <29>   DW_AT_stmt_list   : 0x0
   ...
   ```

这个办法是最粗暴了，因为你可以将其设置为类似于`/usr/src/project-name'这样的路径，把源代码包安装到这个路径下，然后就可以任性地调试了。

结论

GDB 通过以 DWARF 格式存储的调试信息来查找源代码信息。DWARF 是一种非常简单的格式，实际上，它是一棵 DIE（调试信息条目）树，它描述了程序的对象文件以及变量和函数。

有很多种方法可以让 GDB 找到源代码，其中最简单的方法是使用 `directory` 和 `set substitute-path` 命令，而`-fdebug-prefix-map` 是最最强大的。