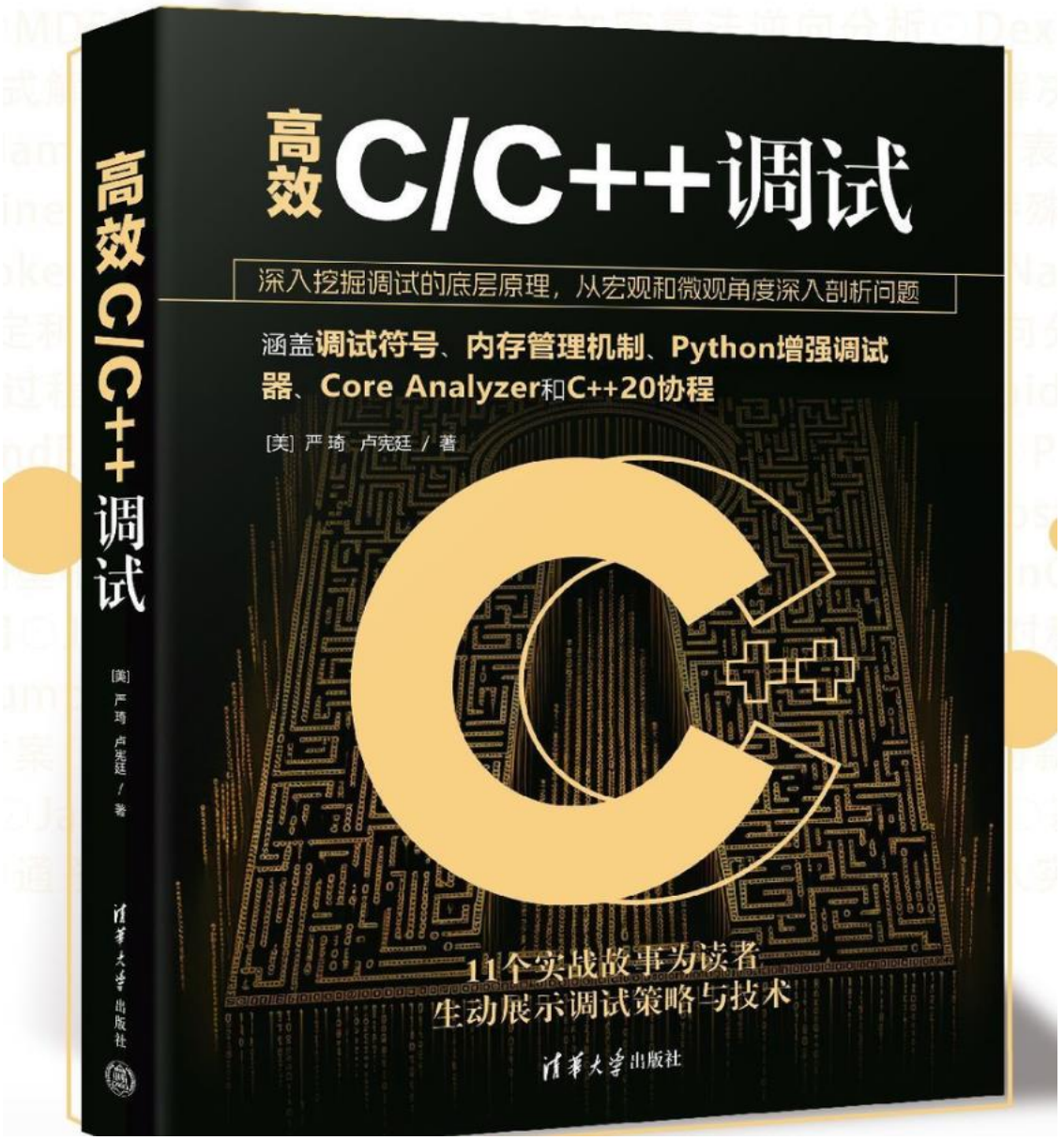
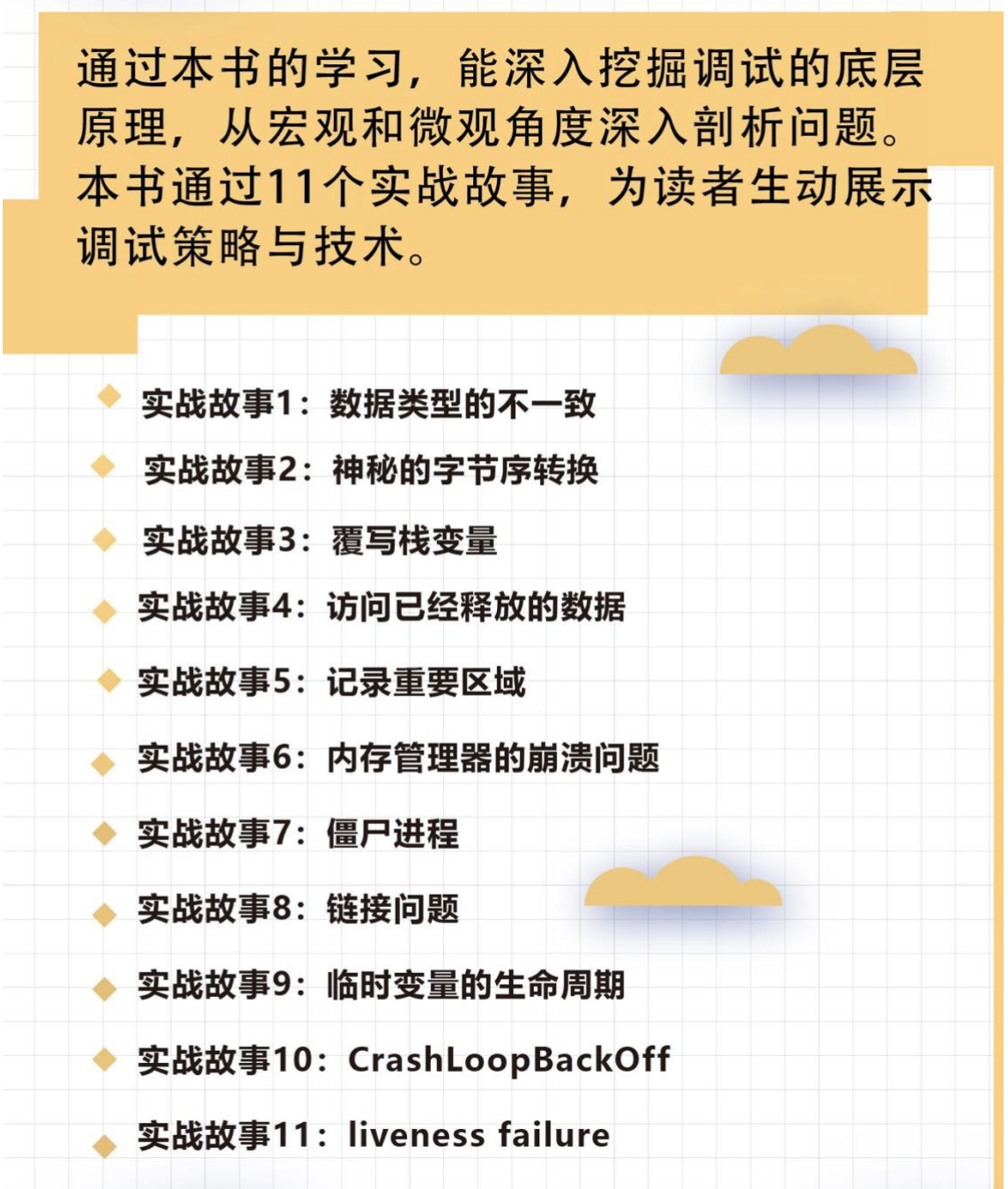

高效C/C++调试
深入挖掘调试的底层原理，从宏观和微观角度深入剖析问题
涵盖调试符号、内存管理机制、Python增强调试器、Core Analyzer和C++20协程
[美] 严琦 卢宪廷 / 著
11个实战故事为读者
生动展示调试策略与技术
清华大学出版社
高效C/C++调试
[美] 严琦 卢宪廷 / 著
清华大学出版社

配套代码：https://github.com/Celthi/effective_c_cpp
联系作者 wx: cracking_oysters



本书内容

第 1 章

调试符号和调试器

讨论程序调试时，我们首先想到的往往是调试器（Debugger），因为它在这个过程中是必不可少的一环。这种现象源于现代编程语言和操作系统的复杂性——要了解一个程序的状态，即使不是完全不可能，也会相当困难。编写代码的开发者通常对调试器有一定的了解，并能或多或少地使用它。但是，你真的对调试器有足够的了解吗？

对于这个问题，不同的人可能有不同的答案。对一些人来说，他们所需要的可能只是设置断点和检查变量的值；而其他人可能需要深入到程序的位和字节级别，以获取线索。根据我个人的经验，每个程序员都应该了解调试器如何实现其所谓的"魔法"。虽然无需深入了解所有调试器的内部细节，如调试符号的生成、组织以及调试器如何使用它们，但理解其实现的基本概念和一些具体细节，能帮助你了解调试器的优势和限制。

有了这些知识，你将能更有效地使用调试器，特别是在处理棘手问题时。例如，如果你了解在调试优化后的代码（如发布版或系统库）时能够访问哪些调试符号，你就能知道在何处设置断点以获取所需信息。你也会了解如何尽量降低调试器自身引入的干扰，如使用硬件断点，以便成功地复现问题。

本章将揭示一些调试器的内部结构，让我们能比通常更深入地了解它。你不仅会了解调试器能做什么，还会知道它是如何做到的，甚至更重要的是，你会了解为什么有时候它无法达到你的预期，以及在这种情况下你应该如何解决。在随后的章节（第 9 章）还将探讨如何通过自定义命令和插件函数来增强调试器的功能。

1.1　调试符号

调试符号由编译器生成，与相关的机器代码、全局数据对象等一同产生。然后，链接器会收集并组织这些符号，将它们写入可执行文件的调试部分（在大多数 UNIX 平台上），或存储到一个单独的文件中（如 Windows 程序的数据库或 pdb 文件）。源码级别的调试器需要从存储库中读取这些调试符号，以便理解进程的内存映像，即程序的运行实例。

在其众多特性中，调试符号可以将进程的指令与对应的源代码行数或表达式相关联，或者从源程序声明的结构化数据对象的角度，对一块内存进行描述。通过这些映射，调试器可以在源代码层面上执行用户命令来查询和操作进程。例如，特定源代码行上的断点会被转换为指令地址；一块内

存会被标记为源代码语言上下文中的变量，并按照其声明类型进行格式化。简单来说，调试符号在高级源程序和程序运行实例的原始内存内容之间架起了一座桥梁。

1.1.1　调试符号概览

源代码级别调试是对编程至关重要的一个环节，为了实现这个目标，编译器需要生成富含各类信息的调试符号。按照调试符号所描述的主题分类，我们主要有以下几类：

- 全局函数和变量：这类调试符号涵盖了跨越编译单元的可见的全局符号的类型和位置信息。全局变量的地址相对于其所属模块的基地址是固定的。在全局变量所属模块被卸载之前，比如程序结束运行或者通过链接器 API 显式地卸载，这些全局变量都是有效且可以访问的。全局变量因其可见性、固定位置和长生命周期，在任何时刻和地点都可以进行调试。这使得调试器能够在全局变量的整个生命周期内，无论程序执行的分支为何，都可以观察数据、修改数据和设置断点。
- 源文件和行信息：调试器的一项主要功能是支持源代码级别的调试，这样程序员就可以在程序的源语言上下文中跟踪（Trace）和观察被调试的程序。这一功能依赖于一种将指令序列映射到源文件和行数的调试符号。因为函数是占据进程连续内存空间的最小可执行代码单元，所以源文件和行号的调试符号会记录每个函数的开始和结束地址。然而为了优化程序性能或减少生成的机器码大小，编译器可能会对源代码进行移位，情况可能变得很复杂。由于宏和内联函数的存在，源代码的行信息可能与实际执行的指令地址并不连续或者交织在其他源代码行中。
- 类型信息：类型的调试符号描述了数据类型的组合关系和属性。这包括基本类型的数据，或是其他数据的聚合。对于复合类型，调试符号包含每个子字段的名字、大小和相对于整个结构开头的偏移量。调试器需要这些类型信息，以便以程序源语言的形式显示它。否则，数据只会是内存内容的原始形态比特位和字节。类型调试符号对于 C++这样的复杂语言特别重要，因为编译器会将隐藏的数据成员添加到数据对象中以实现语言的语义，而这些隐藏的数据成员依赖于编译器的实现。此外类型信息还包括了函数签名和它们的链接属性。
- 静态函数和局部变量：与全局符号不同，静态函数和局部变量只在特定的作用域内可见，比如一个文件，一个函数，或者一个特定的块作用域。局部变量在其作用域内存在和有效，因此是临时的。当程序的执行流程离开其作用域时，作用域内的局部变量将被销毁并在语义上变得无效。由于局部变量通常在栈上分配或与容易失效的寄存器关联，所以在程序运行到其作用域之前，它的存储位置是不确定的。因此，调试器只能在特定的作用域内对局部变量进行观察、修改和设置断点。
- 架构和编译器依赖信息：某些调试功能与特定的架构和编译器有关，比如英特尔芯片的

FPO（Frame Pointer Omission，栈指针省略），以及微软 Visual Studio 的修改和运行功能等。

调试符号的生成是一项复杂的任务，它需要编译器传递大量的调试信息给调试器。因此，即使是相对较小的程序，编译器也会生成大量的调试符号，甚至大大超过了生成的机器代码或者源代码的大小。为了节约空间，人们通常会对调试符号进行编码。

遗憾的是没有一个标准可以规定如何实现调试符号。不同的编译器厂商历来在不同的平台上采用不同的调试符号格式，如 Linux、Solaris 和 HP-UX 现在使用的是 DWARF（Debugging with attributed Record Formats），AIX 和老版本的 Solaris 使用的是 stabs（Symbol Table String），而 Windows 有多种格式，其中最常用的是程序数据库，即 pdb。这些调试符号格式的文档往往难以找到或者即使有也不完整。另一方面，随着编译器新版本的不断发布，与其紧密相关的调试符号格式也必须持续演进。

由于以上原因调试符号格式多少变成了编译器和调试器之间的“秘密”协议，通常与特定平台紧密相关。由于开源社区的努力，DWARF 在公开透明方面做得较好。因此，在下一节中，我们将以 DWARF 作为调试符号实现的示例进行深入讨论。

1.1.2　DWARF 格式

DWARF 以树形结构组织调试符号，这种方式类似于我们在大部分编程语言中看到的结构体和词法作用域。每一个树节点都是一个调试信息记录（Debug Information Entry，DIE），用于表达具体的调试符号，例如：对象、函数、源文件等。一个节点可能有任意数量的子节点或同级节点。例如，一个代表函数的 DIE 可能有许多子 DIE，这些子 DIE 代表函数中的局部变量。

本书不会详尽地阐述每一条 DWARF 格式的规定。如需了解更多，可以在 DWARF 的官网上找到关于 DWARF 的各类论文、教程和正式文档。另一种有效的学习方法是深入研究采用 DWARF 格式的开源编译器（如 GCC）和调试器（如 GDB)。从调试的角度来看，你需要知道有哪些调试符号，它们是如何组织的，以及在需要或者感兴趣的时候如何查看它们。最好的理解方法可能就是通过实例，下面让我们来看一个简单的程序：

```
foo.cpp:
1
2   int gInt = 1;
3
4   int GlobalFunc(int i)
5   {
6       return i+gInt;
7   }
```

可以使用下面的命令选项来编译这个文件：

```
$ g++ -g -S foo.cpp
```

其中-g 选项告诉 g++编译器生成调试符号，-S 选项用于生成供分析的汇编文件。编译器会生成汇编文件作为中间文件，并直接通过管道发送到汇编器。所以如果我们需要审查汇编，就需要显式地让编译器在磁盘上生成汇编文件。

我们可以使用上面的命令自行生成汇编文件，这个过程很简单。虽然这个文件可能有些长，但我还是鼓励读者从头到尾地浏览一遍，这样你将对调试符号的各个部分有一个全面的了解。下面是汇编文件的一个片段。由于这个文件是作为汇编器的输入，而并非供人阅读，所以初看可能会觉得有些困惑。但在我们学习调试符号的每一个组件的过程中，我会对它们的含义进行详细解释。

```
        .file   "foo.cpp"
        .section    .debug_abbrev,"",@progbits
    .Ldebug_abbrev0:
        .section    .debug_info,"",@progbits
    .Ldebug_info0:
        .section    .debug_line,"",@progbits
    .Ldebug_line0:
        .text
    .Ltext0:
    .globl gInt
        .data
        .align 4
        .type   gInt, @object
        .size   gInt, 4
    gInt:
        .long   1
        .text
        .align 2
    .globl _Z10GlobalFunci
        .type   _Z10GlobalFunci, @function
    _Z10GlobalFunci:
    .LFB2:
        .file 1 "foo.cpp"
        .loc 1 5 0
        pushq   %rbp
    .LCFI0:
        movq    %rsp, %rbp
    .LCFI1:
        movl    %edi, -4(%rbp)
    .LBB2:
        .loc 1 6 0
        movl    gInt(%rip), %eax
        addl    -4(%rbp), %eax
    .LBE2:
        .loc 1 7 0
        leave
```

```
    ret
.LFE2:
    .size   _Z10GlobalFunci, .-_Z10GlobalFunci
    .section    .debug_frame,"",@progbits
.Lframe0:
    .long   .LECIE0-.LSCIE0
.LSCIE0:
    .long   0xffffffff

   ...

    .section    .debug_loc,"",@progbits
.Ldebug_loc0:
.LLST0:
    .quad   .LFB2-.Ltext0
    .byte   0x0

   ...
    .section    .debug_info
    .long   0xe6
    .value   0x2
    .long   .Ldebug_abbrev0
    .byte   0x8
    .uleb128 0x1
    .long   .Ldebug_line0
    .quad   .Letext0
    .quad   .Ltext0
    .string   "GNU C++ 3.4.6 20060404 (Red Hat 3.4.6-9)"
    .byte   0x4
    .string   "foo.cpp"
    .string   "/home/myan/projects/p_debugging"
    .uleb128 0x2
    .long   0xba
    .byte   0x1
    .string   "GlobalFunc"
    .byte   0x1
    .byte   0x5
    .string   "_Z10GlobalFunci"
    .long   0xba
    .quad   .LFB2
    .quad   .LFE2
    .long   .LLST0
    .uleb128 0x3
    .string   "i"
    .byte   0x1
    .byte   0x5
```

```
        .long    0xba
        .byte    0x2
        .byte    0x91
        .sleb128 -20
        .byte    0x0
        .uleb128 0x4
        .string   "int"
        .byte    0x4
        .byte    0x5
        .uleb128 0x5
        .long    0xda
        .string   "::"
        .byte    0x2
        .byte    0x0
        .uleb128 0x6
        .string   "gInt"
        .byte    0x1
        .byte    0x2
        .long    0xba
        .byte    0x1
        .byte    0x1
        .byte    0x0
        .uleb128 0x7
        .long    0xcb
        .byte    0x9
        .byte    0x3
        .quad    gInt
        .byte    0x0
        .section   .debug_abbrev
        .uleb128 0x1

       ...

        .section   .debug_pubnames,"",@progbits
        .long    0x26

       ...

        .section   .debug_aranges,"",@progbits
        .long    0x2c

       ...
```

可以看出，汇编文件中大部分内容是为了生成调试符号的，只有一小部分是可执行指令。对于简短的程序来说，这是非常典型的情况。由于文件大小的考虑，调试符号通常会被编码到二进制文件中以减小文件的大小。我们可以使用以下工具来解码它，查看调试符号：

```
$readelf --debug-dump foo.o
```

这个命令会输出目标文件 foo.o 中的所有调试符号。这些调试符号被划分为多个节（英文对应的单词是 section，中文翻译有时对应段，请读者注意区分 section 与 segment，本文后面会讲解）。每个节代表一种类型的调试符号，并储存在 ELF 目标文件的特定区域内（在第 6 章，我们会更深入地讨论二进制文件和 ELF 的详细内容）。下面让我们逐一查看这些节。

首先我们要关注的是储存在.debug_abbrev 节的缩略表。这个表描述了一种用于减小 DWARF 文件大小的编码算法。在缩略表中的 DIE 并非真正的 DIE。相反，它们作为在其他节中具有相同类型和属性的实际 DIE 的模板。一个真实的 DIE 项只包含一个到缩略表模板 DIE 的引用和用于实例化这个模板 DIE 的数据。在我们的示例中，缩略表包含 7 项，包括编译单元、全局变量、数据类型、输入参数、局部变量等的模板。表中的第三项（显示为加粗字体）声明了一种具有五个部分调试信息的 DIE：名称、文件、行号、数据类型和位置。我们将看到真实的 DIE 是如何引用这个模板的。

```
Contents of the .debug_abbrev section:

  Number TAG
   1     DW_TAG_compile_unit   [has children]
    DW_AT_stmt_list   DW_FORM_data4
    DW_AT_high_pc     DW_FORM_addr
    DW_AT_low_pc      DW_FORM_addr
    DW_AT_producer    DW_FORM_string
    DW_AT_language    DW_FORM_data1
    DW_AT_name        DW_FORM_string
    DW_AT_comp_dir    DW_FORM_string
   2     DW_TAG_subprogram   [has children]
    DW_AT_sibling     DW_FORM_ref4
    DW_AT_external    DW_FORM_flag
    DW_AT_name        DW_FORM_string
    DW_AT_decl_file   DW_FORM_data1
    DW_AT_decl_line   DW_FORM_data1
    DW_AT_MIPS_linkage_name DW_FORM_string
    DW_AT_type        DW_FORM_ref4
    DW_AT_low_pc      DW_FORM_addr
    DW_AT_high_pc     DW_FORM_addr
    DW_AT_frame_base  DW_FORM_data4
   3     DW_TAG_formal_parameter   [no children]
    DW_AT_name        DW_FORM_string
    DW_AT_decl_file   DW_FORM_data1
    DW_AT_decl_line   DW_FORM_data1
    DW_AT_type        DW_FORM_ref4
    DW_AT_location    DW_FORM_block1
   4     DW_TAG_base_type   [no children]
    DW_AT_name        DW_FORM_string
```

```
    DW_AT_byte_size   DW_FORM_data1
    DW_AT_encoding    DW_FORM_data1
   5     DW_TAG_namespace    [has children]
    DW_AT_sibling     DW_FORM_ref4
    DW_AT_name        DW_FORM_string
    DW_AT_decl_file   DW_FORM_data1
    DW_AT_decl_line   DW_FORM_data1
   6     DW_TAG_variable    [no children]
    DW_AT_name        DW_FORM_string
    DW_AT_decl_file   DW_FORM_data1
    DW_AT_decl_line   DW_FORM_data1
    DW_AT_type        DW_FORM_ref4
    DW_AT_external    DW_FORM_flag
    DW_AT_declaration DW_FORM_flag
   7     DW_TAG_variable    [no children]
    DW_AT_specification DW_FORM_ref4
    DW_AT_location    DW_FORM_block1
```

下一节（即.debug_info 节）包含了调试符号的核心内容，包括数据类型的信息、变量、函数等。特别需要注意的是，调试信息项（DIEs）是如何编码并通过索引来引用在缩略表里的特定项。

以下是一个例子，其中我们用粗体框选的 DIE 来描述一个名为 GlobalFunc 的函数的唯一传入参数。这个 DIE 在缩略表中的索引是 3。接着，我们使用实际的信息来填充该 DIE 的五个字段：

- 参数的名称是“i”。
- 它出现在第 1 个文件中。
- 它位于该文件的第 5 行。
- 参数的类型由另一个 DIE（引用为 ba）来描述。
- 参数在内存中的位置有一个偏移量为 2。

```
The section .debug_info contains:

  Compilation Unit @ 0:
   Length:        230
   Version:       2
   Abbrev Offset: 0
   Pointer Size:  8
 <0><b>: Abbrev Number    : 1 (DW_TAG_compile_unit)
    DW_AT_stmt_list       : 0
    DW_AT_high_pc         : 0x12
    DW_AT_low_pc          : 0
    DW_AT_producer        : GNU C++ 3.4.6 20060404 (Red Hat 3.4.6-9)
    DW_AT_language        : 4    (C++)
    DW_AT_name            : foo.cpp
    DW_AT_comp_dir        : /home/myan/projects/p_debugging
```

```
<1><72>: Abbrev Number   : 2 (DW_TAG_subprogram)
   DW_AT_sibling         : <ba>
   DW_AT_external        : 1
   DW_AT_name            : GlobalFunc
   DW_AT_decl_file       : 1
   DW_AT_decl_line       : 5
   DW_AT_MIPS_linkage_name: _Z10GlobalFunci
   DW_AT_type            : <ba>
   DW_AT_low_pc          : 0
   DW_AT_high_pc         : 0x12
   DW_AT_frame_base      : 0   (location list)
<2><ad>: Abbrev Number   : 3 (DW_TAG_formal_parameter)
   DW_AT_name            : i
   DW_AT_decl_file       : 1
   DW_AT_decl_line       : 5
   DW_AT_type            : <ba>
   DW_AT_location        : 2 byte block: 91 6c   (DW_OP_fbreg: -20)
<1><ba>: Abbrev Number   : 4 (DW_TAG_base_type)
   DW_AT_name            : int
   DW_AT_byte_size       : 4
   DW_AT_encoding        : 5   (signed)
<1><c1>: Abbrev Number   : 5 (DW_TAG_namespace)
   DW_AT_sibling         : <da>
   DW_AT_name            : ::
   DW_AT_decl_file       : 2
   DW_AT_decl_line       : 0
<2><cb>: Abbrev Number   : 6 (DW_TAG_variable)
   DW_AT_name            : gInt
   DW_AT_decl_file       : 1
   DW_AT_decl_line       : 2
   DW_AT_type            : <ba>
   DW_AT_external        : 1
   DW_AT_declaration     : 1
<1><da>: Abbrev Number   : 7 (DW_TAG_variable)
   DW_AT_specification: <cb>
   DW_AT_location        : 9 byte block: 3 0 0 0 0 0 0 0 0    (DW_OP_addr: 0)
```

利用这种编码方式，我们成功地用目标文件中的仅仅 13 个字节来表示参数“i”的调试符号。接下来，我们可以使用 objdump 命令来查看.debug_info 节中的原始数据。

```
$objdump -s --section=.debug_info foo.o

foo.o:    file format elf64-x86-64

Contents of section .debug_info:
 0000 e6000000 02000000 00000801 00000000  ................
```

```
0010 00000000 00000000 00000000 00000000  ................
0020 474e5520 432b2b20 332e342e 36203230  GNU C++ 3.4.6 20
0030 30363034 30342028 52656420 48617420  060404 (Red Hat
0040 332e342e 362d3929 0004666f 6f2e6370  3.4.6-9)..foo.cp
0050 70002f68 6f6d652f 6d79616e 2f70726f  p./home/myan/pro
0060 6a656374 732f705f 64656275 6767696e  jects/p_debuggin
0070 670002ba 00000001 476c6f62 616c4675  g.......GlobalFu
0080 6e630001 055f5a31 30476c6f 62616c46  nc..._Z10GlobalF
0090 756e6369 00ba0000 00000000 00000000  unci............
00a0 00000000 00000000 00000000 00036900  ..............i.
00b0 0105ba00 00000291 6c000469 6e740004  ........l..int..
00c0 0505da00 00003a3a 00020006 67496e74  ......::....gInt
00d0 000102ba 00000001 010007cb 00000009  ................
00e0 03000000 00000000 0000              ..........
```

现在，如果我们回到汇编文件 foo.s，就可以在其中找到传入参数 i 的调试符号。这些调试符号的相关行会在以下部分高亮显示。在我们之前列出的汇编文件中，你可以轻易地找到它们。

```
        .uleb128 0x3
        .string   "i"
        .byte    0x1
        .byte    0x5
        .long    0xba
        .byte    0x2
        .byte    0x91
        .sleb128 -20
        .byte    0x0
```

上述 DIE 项在结构上类似于 C 语言的结构体。它们的编码后的字节含义如图 1-1 所示。首先，以缩略表的索引（3）开始，这指示了 DIE 的剩余数据应该如何格式化。你可以回顾前面列出的缩略表，参考第三个 DIE 模板来理解：

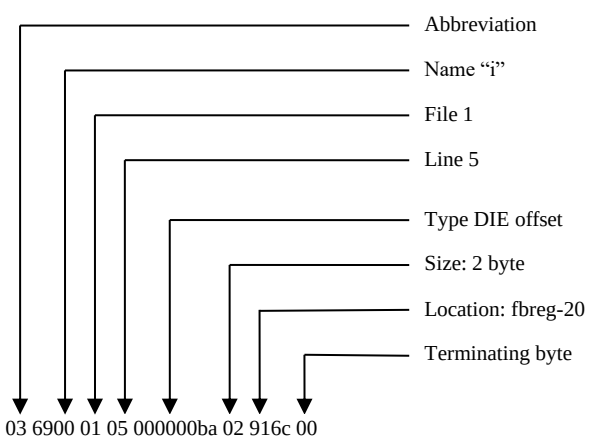


图 **Error! No text of specified style in document.**-1　DIE 的编码

接下来的两个字节代表了一个以 null 结尾的字符串，即我们的参数名称“i”。之后，我们看到的是文件编号（1）和行号（5）。参数的类型由另外一个 DIE（索引为 ba）来定义。

接着的数据是参数的大小，具体来说，就是 2 个字节。然后，参数的存储位置由接下来的两个字节指示，这对应于相对于寄存器 fbreg 的偏移量−20。

最后，这个 DIE 以一个零字节结束，标志着这一段信息的结束。

每个 DIE 都会指定其父节点、子节点和兄弟节点（如果存在的话）。图 1-2 展示了.debug_info 节中列出的 DIEs 之间的父子关系和兄弟关系。特别注意到，参数 i 的 DIE 是函数 GlobalFunc DIE 的子节点，这与源程序中的作用域设置完全一致。

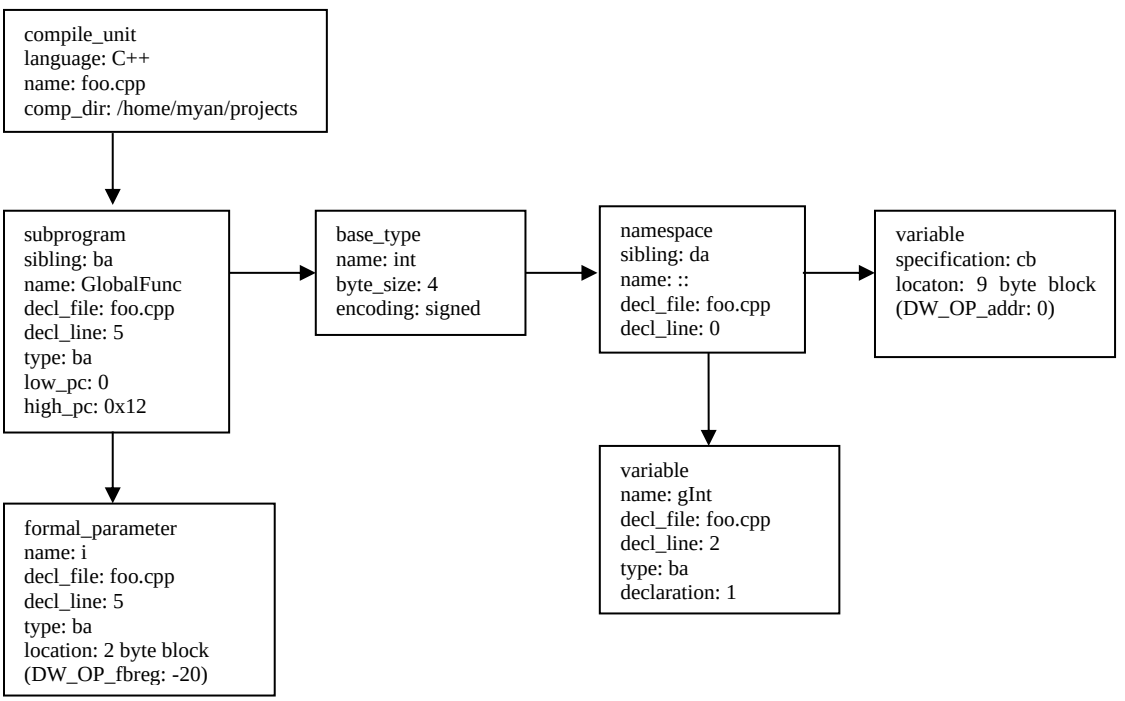


图 **Error! No text of specified style in document.**-2　树结构的 DIEs 的关系

源代码的行号调试符号被放置在.debug_line 节中，它由一系列操作码构成，调试器可以执行这些操作码以构建一个状态表。这张表是将指令地址映射到源代码行号的关键。每个状态都包括一个指令地址（以函数开头的偏移量表示）、相应的源代码行号和文件名。

你可能会问，如何从操作码创建状态表呢？这始于设置初始值的操作码，如初始指令地址。每次源代码行号发生变化时，操作码将操作地址移动一个变化量。调试器会运行操作码，并在每次状态变化时向状态表添加一行。

以下是 readelf 输出，显示了样例程序的行号调试符号。请注意高亮的行以可读的方式描述了

操作码的操作。指令地址从 0x0 开始，结束于 0x12，对应的源代码行号从 4 逐渐增加到 7。

```
Dump of debug contents of section .debug_line:

  Length:                      66
  DWARF Version:               2
  Prologue Length:             41
  Minimum Instruction Length:  1
  Initial value of 'is_stmt':  1
  Line Base:                   -5
  Line Range:                  14
  Opcode Base:                 10
  (Pointer size:               8)

 Opcodes:
  Opcode 1 has 0 args
  Opcode 2 has 1 args
  Opcode 3 has 1 args
  Opcode 4 has 1 args
  Opcode 5 has 1 args
  Opcode 6 has 0 args
  Opcode 7 has 0 args
  Opcode 8 has 0 args
  Opcode 9 has 1 args

 The Directory Table is empty.

 The File Name Table:
  Entry    Dir    Time    Size    Name
  1    0    0    0    foo.cpp
  2    0    0    0    <internal>

Line Number Statements:
 Extended opcode 2: set Address to 0x0
 Special opcode 9: advance Address by 0 to 0x0 and Line by 4 to 5
 Special opcode 104: advance Address by 7 to 0x7 and Line by 1 to 6
 Special opcode 132: advance Address by 9 to 0x10 and Line by 1 to 7
 Advance PC by 2 to 12
 Extended opcode 1: End of Sequence
```

Call Frame Information（CFI）存储在.debug_frame 节中，描述了函数的栈帧和寄存器的分配方式。调试器使用此信息来回滚（unwind）栈。例如，如果一个函数的局部变量被分配在一个寄存器中，该寄存器稍后被一个被调用的函数占用，其原始值会保存在被调用函数的栈帧中。调试器需要依赖 CFI 来确定保存寄存器的栈地址，从而观察或改变相应的局部变量。

类似于源代码行号，CFI 也被编码为一系列的操作码。调试器按照给定的顺序执行这些操作码，

以创建一个跟随指令地址前进的寄存器状态表。根据这个状态表，调试器可以确定栈帧的地址位置（通常由栈帧寄存器指向），以及当前函数的返回值和函数参数的位置。下面列出的是示例的 CFI 调试符号，它展示了简单函数 glbalFunc 的 r6 寄存器的信息。

```
The section .debug_frame contains:

00000000 00000014 ffffffff CIE
  Version:               1
  Augmentation:          ""
  Code alignment factor: 1
  Data alignment factor: -8
  Return address column: 16

  DW_CFA_def_cfa: r7 ofs 8
  DW_CFA_offset: r16 at cfa-8
  DW_CFA_nop
  DW_CFA_nop
  DW_CFA_nop
  DW_CFA_nop
  DW_CFA_nop
  DW_CFA_nop

00000018 0000001c 00000000 FDE cie=00000000 pc=00000000..00000012
  DW_CFA_advance_loc: 1 to 00000001
  DW_CFA_def_cfa_offset: 16
  DW_CFA_offset: r6 at cfa-16
  DW_CFA_advance_loc: 3 to 00000004
  DW_CFA_def_cfa_reg: r6
```

除了上述提到的部分，还有一些其他的节包含各种类型的调试信息：

- .debug_loc 节包含了宏表达式的调试符号，但是本例中并没有宏。
- .debug_pubnames 节是全局变量和函数的查找表，用于更快地访问这些调试项。在这个例子中，我们有两个项：全局变量 gInt 和全局函数 GlobalFunc。
- .debug_aranges 节包含一系列的地址长度对，说明每个编译单元的地址范围。

```
Contents of the .debug_loc section:


    Offset   Begin    End      Expression
    00000000 00000000 00000001 (DW_OP_breg7: 8)
    00000000 00000001 00000004 (DW_OP_breg7: 16)
    00000000 00000004 00000012 (DW_OP_breg6: 16)

Contents of the .debug_pubnames section:
```

```
  Length:                                38
  Version:                               2
  Offset into .debug_info section:       0
  Size of area in .debug_info section: 234

    Offset    Name
    114         GlobalFunc
    218         gInt

The section .debug_aranges contains:

  Length:                   44
  Version:                  2
  Offset into .debug_info: 0
  Pointer Size:             8
  Segment Size:             0

    Address  Length
    00000000 18
```

所有这些节为调试器提供了实现各种调试功能所需的足够信息，比如将当前的程序指令地址映射到对应的源代码行，或者计算局部变量的地址并根据其类型打印结构数值。

调试符号首先在每个编译单元中生成，就如我们在目标文件示例中看到的那样。在链接时，多个编译单元的调试符号被收集、组合，并链接到可执行文件或库文件中。

在我们继续探讨调试器的实现之前，我想先分享一个通过类型的调试符号发现的 bug 的故事。这个故事演示了看似不一致的调试符号，尤其是那些分布在不同模块中的调试符号，如何为我们揭示了代码或构建过程中的问题。

1.2　实战故事：数据类型的不一致

我们的服务器程序在测试阶段出现了随机崩溃。经过一段时间的调试，我们怀疑这可能是由于内存越界错误引起的，问题似乎出在一个特定的数据对象上。当程序尝试更新该对象的一个数据成员时，紧随其后的数据对象就会被损坏（这一问题是由我们将在第 10 章讨论的内存调试工具发现的）。

然而，这段代码看似是无辜的，因为它只是在访问自己的数据成员。令人困惑的是，这个简单操作如何会损坏另一个数据对象。通过深入调查，我们发现这个问题对象是在一个模块中创建的，然后传递到另一个模块，在那里进行数据成员的更新。

在一番探索后，我们发现两个模块对同一个数据对象的大小存在不一致的看法。调试器在第一个模块环境中显示一个尺寸，而在第二个模块中打印出另一个更大的尺寸。这令人大为惊讶，因为这个对象是在一个头文件中声明的，且这个头文件被两个项目共享。通过更深入地打印并比较每个

模块内的对象布局以及其数据成员的偏移量（对象的类型调试符号），明显看出编译器对对象的布局有所不同：一个模块中的所有数据成员都适当地对齐，而另一个模块并没有，而是将所有的数据成员打包在一起。数据对象以较小的尺寸被打包创建。

这种情况也得到了底层内存管理器分配的内存块大小的证实（在第 2 章，我们将详细讨论如何获取此类信息）。当对象从打包对齐的模块被传入未打包布局的模块时，更新数据成员的操作就会覆盖内存并损坏附近的对象。图 1-3 以更简洁的方式描述了这个 bug。一个 T 类型的结构体对象在模块 A 中被创建为打包格式，然后传到模块 B，模块 B 却认为它是未打包的格式。模块 B 中灰色的数据成员 data3 覆盖了已分配的内存块。

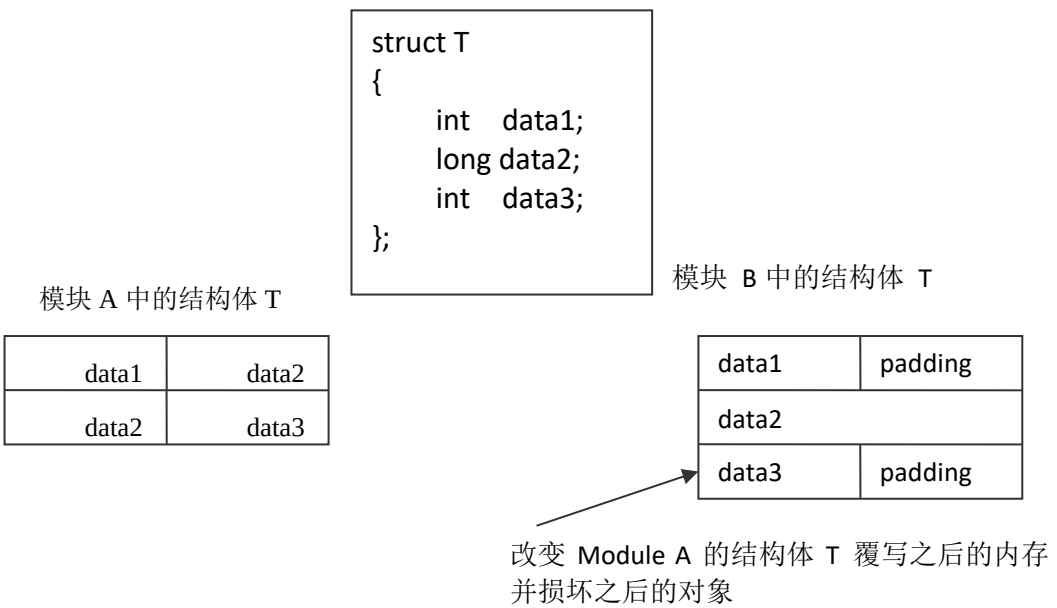


图 1-3　由于数据类型不一致导致的内存覆盖问题

你可能很想知道这种情况是如何发生的。结果表明，尽管对象在头文件中被正确地声明，但 bug 却来自另一个头文件，其中使用了以下编译指令：

```
#pragma pack(4)
...
#pragma pack()
```

开发工程师打算将编译指令中间的结构体打包为 4 字节边界。这个指令被微软的 Visual Studio 编译器正确解析。然而，当同一份代码被 AIX 的 Visual Age C++编译器编译时，问题就出现了。该编译器有一个类似但略有不同的编译指令语法来结束打包作用域：

```
#pragma pack(4)
```

```
...
#pragma pack(nopack)
```

由于这个语法差异，Visual Age C++编译器只识别了打包的开始编译指令（第一行），却忽略了结束打包的编译指令（最后一行）。在程序员试图结束数据打包的地方，编译器仍然在继续打包数据结构。在模块 A 中，我们的受害对象在引入包含上述编译指令的头文件之后声明。在模块 B 中，问题头文件没有被引入，所以对象没有被打包。这就是不一致性的产生原因。

数据类型的调试符号准确地反映了编译器如何解析数据类型，生成的机器指令也根据这个解析结果来操作数据对象。具体来说，当创建新对象时，编译器会申请与结构体大小相等的内存块；数据成员的访问地址是通过从内存块开头的偏移量来计算的。

1.3　调试器的内部结构

大多数程序员都是通过实践来学习如何使用调试器的，有些人会比其他人更熟悉调试器的各种命令的使用，然而只有少数人了解调试器的内部结构。在这一节中，我将从用户的角度来讨论调试器的一些实现细节。这不仅是为了满足你对调试器的好奇心，更重要的是，它有助于你更深入地理解调试器以便充分利用这个工具。

实际上调试器只不过是另一个应用程序。有趣的是，你可以用一个调试器进程来跟踪正在运行的另一个调试器，这实际上是理解调试器工作原理的有效方式。我曾经为了常规调试任务，编译了 GDB 调试器的调试版本。每当我对调试器本身有疑问时，我就会启动一个 GDB 程序，并将其附加到正在使用的 GDB 进程上。这样我就能看到它所有的内部数据结构。

源代码级别的调试器通常由三个模块组成：用户界面，符号管理和目标管理。

1.3.1　用户界面

用户界面是调试器的表现层，也就是前端。它与用户的交互方式与其他应用程序非常相似。调试器可能有图形用户界面（GUI）或命令行界面（CLI），或者两者都有。它的基本功能是将用户的输入转换为对后端调试引擎的 API 调用。几乎每个菜单项或按钮都直接映射到后端命令。事实上，许多具有 GUI 界面的调试器，如 ddd（Data Display Debugger）、Windbg 和 sunstudio，都有一个命令窗口，可以让用户直接向底层调试器输入命令。

1.3.2　符号管理模块

符号管理模块负责提供调试目标的调试符号。这个模块的基本功能包括读取二进制文件并解析其中的调试符号，创建调试符号的内部表示，为打印变量提供类型信息等。调试符号的可用性和内容的完整性决定了调试器的功能和限制。如果调试符号错误或不完整，那么调试器将无法正常工作。例如，不匹配的文件（可执行文件或程序数据库文件）拥有错误的调试符号；去除了调试符号的可执行文件或没有 pdb 文件的 DLLs，或者只有部分调试符号的文件，都只能提供有限的调试能

力。

在前面的章节中，我们已经看到调试符号是如何在文件中组织和存储的。首先，调试器会按照给定的调试符号路径来搜索文件，然后检查文件的大小、时间戳、校验和等信息，以验证其与被调试进程加载的映像文件的一致性。如果没有正确匹配的调试符号，调试器将无法正常工作。例如，如果没有匹配的内核符号，Windows 调试器 Windbg 会发出如下警告信息:

```
    [Frames below may be incorrect and/or missing, no symbols loaded for msvcr80.dll]
    msvcr80.dll!78138a04()
    msvcr80.dll!78138a8c()
    SHSMP.DLL!_MemFreePtr@4()  + 0x4b bytes
    SHSMP.DLL!_shi_free1()  + 0x1c bytes
    SHSMP.DLL!_shi_free()  + 0xa bytes
    M8Log2.dll!std::allocator<Base::SmartPtrI<MLog::Destination> >::deallocate(Base::S
martPtrI<MLog::Destination> * _Ptr=0x01a51638, unsigned int __formal=2)  Line 141 + 0x9
bytes C++
    M8Log2.dll!MLog::Dispatcher_Impl::LogMessage(const MLog::Logger & iLogger={...},
const char * iMessageText=0x00770010, unsigned int iMessageID=8)  Line 78 + 0x1c bytes
C++
```

请注意 msvcr80.dll 系统运行库的前两个帧。Windbg 在此时提示无法找到该 DLL 的调试符号。不仅如此，由于系统库默认开启了 FPO 编译器选项，这使得优化的代码需要 FPO 调试符号以成功回溯调用栈，否则可能会呈现出逻辑不通的调用栈。在此案例中，我们可以设置 Windbg 从微软的在线调试符号服务器下载这些符号。稍后，我们将进一步讨论 Windows 符号服务器。

如果调试符号匹配良好，调试器的符号管理将打开文件并读取其中的调试部分或者单独的数据库中的调试符号，然后解析这些调试符号，创建内部表现。为了避免在启动时消耗大量时间和空间，调试器通常不会一次性读取所有调试符号。例如，行号表和基准栈信息表会在需要时生成。调试器开始时只会扫描文件，快速定位基本信息，如源文件和当前作用域的符号。当用户命令需要某些详细调试符号时（例如，打印变量），调试器会从对应文件按需读取详细的调试符号。值得注意的是，GDB 的符号加载命令“-readnow”选项允许用户覆盖这种分阶段的符号加载策略。

1.3.3　目标管理模块

目标管理模块在系统和硬件层面处理被调试的进程。例如，控制被调试进程的运行，读写其内存，检索线程调用栈等。因为这些底层操作依赖于特定平台，在 Linux 以及许多其他 UNIX 变种中，内核提供了一个系统调用 ptrace，允许一个进程（调试器或其他工具，如系统调用追踪器 strace）查询和控制另一个进程（被调试程序）的执行。Linux 内核使用信号来同步调试器和被调试进程。ptrace 提供了以下功能：

- 追踪进程或与其分离。被追踪的进程在被追踪时，会收到一个 SIGTRAP 或者 SIGSTOP 信号。
- 读写被调试进程内存地址空间的内容，包括文本和数据段。

- 查询和修改被调试进程的用户区域信息，例如寄存器等。
- 查询和修改被调试进程的信号信息和设置。
- 设置事件触发器，例如在调用系统 API（如 fork、clone、exec 等）或被调试进程退出时停止被调试进程。
- 控制被调试进程的运行，例如使其从停止状态恢复运行，或者在下一个系统调用时停止，或者执行到下一条指令。
- 向被调试进程发送各种信号，例如发送 SIGKILL 信号结束进程。

这些内核服务为实现各种调试器特性提供了基础，稍后我们将以断点为例进行讲解。ptrace 的函数原型在头文件 sys/ptrace.h 中声明，其包括四个参数：请求类型、被调试进程的 ID、被调试进程中将被读写的内存地址以及将被读写的内存字节缓冲区。

```
/* Type of the REQUEST argument to `ptrace.'  */
enum __ptrace_request
{
  /* Indicate that the process making this request should be traced. */
  PTRACE_TRACEME = 0,

  /* Return the word in the process's text space at address addr.  */
  PTRACE_PEEKTEXT = 1,

  /* Return the word in the process's data space at address addr.  */
  PTRACE_PEEKDATA = 2,

  /* Return the word in the process's user area at offset addr.  */
  PTRACE_PEEKUSER = 3,

  /* Write the word data into the process's text space at address addr. */
  PTRACE_POKETEXT = 4,

  /* Write the word data into the process's data space at address addr. */
  PTRACE_POKEDATA = 5,

  /* Write the word data into the process's user area at offset addr.  */
  PTRACE_POKEUSER = 6,

  /* Continue the process.  */
  PTRACE_CONT = 7,

  /* Kill the process.  */
  PTRACE_KILL = 8,

  /* Single step the process.  */
```

```
  PTRACE_SINGLESTEP = 9,

  /* Get all general purpose registers used by a processes.  */
   PTRACE_GETREGS = 12,

  /* Set all general purpose registers used by a processes. */
   PTRACE_SETREGS = 13,

  ...

  /* Set ptrace filter options.  */
  PTRACE_SETOPTIONS = 0x4200,

  /* Get last ptrace message.  */
  PTRACE_GETEVENTMSG = 0x4201,
};

/* Perform process tracing functions.  REQUEST is one of the values
   above, and determines the action to be taken.  */
long ptrace (enum __ptrace_request request, pid_t pid, void *addr, void *data);
```

在下面的例子中我们用 strace 命令打印出调试器 GDB 引发的所有 ptrace 调用（更多关于 strace 的功能，我们将在第 10 章详细讨论）。这里的调试器进程正在进行一个简单的调试会话，而我们并不关心程序 a.out 做了什么，我们只关注调试器的操作。在这个例子中，GDB 在测试程序的入口函数 main 处设置了一个断点，然后运行这个程序。当程序运行结束后，GDB 也结束这个调试会话。系统调用跟踪程序打印了许多 ptrace 调用，我在这里仅列出一部分内容，目的是为了突出展示 GDB 的底层实现方式。

```
$ strace -o/home/myan/ptrace.log -eptrace gdb a.out
(gdb) break main
Breakpoint 1 at 0x400590: file foo.cpp, line 12.
(gdb) run
Starting program: /home/myan/a.out

Breakpoint 1, main () at foo.cpp:12
12            int* ip = new int;
(gdb) cont
Continuing.

Program exited normally.
(gdb) quit

$ cat /home/myan/ptrace.log
ptrace(PTRACE_GETREGS, 28361, 0, 0x7fbfffe650) = 0
ptrace(PTRACE_PEEKUSER, 28361, offsetof(struct user, u_debugreg) + 48, [0]) = 0
```

```
    ptrace(PTRACE_CONT, 28361, 0x1, SIG_0)  = 0
    --- SIGCHLD (Child exited) @ 0 (0) ---
    ptrace(PTRACE_GETREGS, 28361, 0, 0x7fbfffe650) = 0
    ptrace(PTRACE_PEEKUSER, 28361, offsetof(struct user, u_debugreg) + 48, [0]) = 0
    ptrace(PTRACE_SETOPTIONS, 28361, 0, 0x2) = 0
    ptrace(PTRACE_SETOPTIONS, 28366, 0, 0x2) = 0
    ptrace(PTRACE_SETOPTIONS, 28366, 0, 0x22) = 0
    ptrace(PTRACE_CONT, 28366, 0, SIG_0)    = 0
    --- SIGCHLD (Child exited) @ 0 (0) ---
    ptrace(PTRACE_GETEVENTMSG, 28366, 0, 0x7fbfffeb90) = 0
    ptrace(PTRACE_SETOPTIONS, 28361, 0, 0x3e) = 0
    ptrace(PTRACE_PEEKTEXT, 28361, 0x5007e0, [0x1]) = 0
    ptrace(PTRACE_PEEKTEXT, 28361, 0x5007e8, [0x1]) = 0
    ...
    ptrace(PTRACE_PEEKTEXT, 28361, 0x400590, [0xff06e800000004bf]) = 0
    ptrace(PTRACE_POKEDATA, 28361, 0x400590, 0xff06e800000004cc) = 0
    ptrace(PTRACE_PEEKTEXT, 28361, 0x36a550b830, [0x909090909090c3f3]) = 0
    ptrace(PTRACE_PEEKTEXT, 28361, 0x36a550b830, [0x909090909090c3f3]) = 0
    ptrace(PTRACE_POKEDATA, 28361, 0x36a550b830, 0x909090909090c3cc) = 0
    ptrace(PTRACE_CONT, 28361, 0x1, SIG_0)  = 0
    --- SIGCHLD (Child exited) @ 0 (0) ---
    ptrace(PTRACE_GETREGS, 28361, 0, 0x7fbfffe750) = 0
    ptrace(PTRACE_GETREGS, 28361, 0, 0x7fbfffe790) = 0
    ptrace(PTRACE_SETREGS, 28361, 0, 0x7fbfffe790) = 0
    ptrace(PTRACE_PEEKUSER, 28361, offsetof(struct user, u_debugreg) + 48, [0]) = 0
    ptrace(PTRACE_PEEKTEXT, 28361, 0x400590, [0xff06e800000004cc]) = 0
    ptrace(PTRACE_POKEDATA, 28361, 0x400590, 0xff06e800000004bf) = 0
    ptrace(PTRACE_PEEKTEXT, 28361, 0x36a550b830, [0x909090909090c3cc]) = 0
    ptrace(PTRACE_POKEDATA, 28361, 0x36a550b830, 0x909090909090c3f3) = 0
    ptrace(PTRACE_PEEKTEXT, 28361, 0x5007e0, [0x1]) = 0
    ...
    ptrace(PTRACE_PEEKTEXT, 28361, 0x400588, [0x10ec8348e5894855]) = 0
    ptrace(PTRACE_SINGLESTEP, 28361, 0x1, SIG_0) = 0
    --- SIGCHLD (Child exited) @ 0 (0) ---
    ptrace(PTRACE_GETREGS, 28361, 0, 0x7fbfffe750) = 0
    ptrace(PTRACE_PEEKUSER, 28361, offsetof(struct user, u_debugreg) + 48,
[0xffff4ff0]) = 0
    ptrace(PTRACE_PEEKTEXT, 28361, 0x400590, [0xff06e800000004bf]) = 0
    ptrace(PTRACE_POKEDATA, 28361, 0x400590, 0xff06e800000004cc) = 0
    ptrace(PTRACE_PEEKTEXT, 28361, 0x36a550b830, [0x909090909090c3f3]) = 0
    ptrace(PTRACE_PEEKTEXT, 28361, 0x36a550b830, [0x909090909090c3f3]) = 0
    ptrace(PTRACE_POKEDATA, 28361, 0x36a550b830, 0x909090909090c3cc) = 0
    ptrace(PTRACE_CONT, 28361, 0x1, SIG_0)  = 0
    ...
```

如上所示，GDB 通过 PTRACE_GETREGS 和 PTRACE_SETREGS 请求获取和修改被调试进

程的上下文，又通过 PTRACE_PEEKTEXT 和 PTRACE_POKETEXT 请求读取和写入被调试进程的内存，以及执行其他一系列操作。当有事件发生时，内核通过发送 SIGCHLD 信号来暂停调试器。

让我们通过上述例子深入了解断点是如何工作的。在 GDB 控制台中，我们能够看到，一个断点被设置在了函数 main 的起始地址 0x400590。以下是其执行过程：

（1）调试器首先读取地址 0x400590 处的代码，即{0xbf 0x04 0x00 0x00 0x00 0xe8 0x06 0xff}。需要注意的是，x86_64 架构使用小端序（详情请参见 4.1.2 节）。

（2）接着，GDB 使用 PTRACE_POKEDATA 请求替换该地址的代码。原始数据 0xff06e800000004bf 被更改为 0xff06e800000004cc，其中第一个字节从 0xbf 更改为 0xcc。这里的 0xcc 是一种特殊的陷阱指令。

（3）通过此操作，调试器在被调试进程的代码段中设定了断点，并使用 PTRACE_CONT 命令继续执行该进程。

（4）当程序执行到陷阱指令 0xcc 时，它会触发断点并被内核终止。内核检查后发现进程正在被调试，于是会向调试器发送一个信号。

（5）GDB 显示这一信息并等待用户输入。在这个例子中，我们选择继续执行程序。

（6）为确保程序的完整性，GDB 会替换回原始指令 0xbf，并使用 PTRACE_SINGLESTEP 请求执行单个指令。执行后，为使断点再次生效，调试器重新插入 0xcc。

（7）如果用户设置的是一次性断点，上述操作则不会进行。处理完断点后，调试器使用 PTRACE_CONT 让程序继续执行。

调试器在一个持续的循环中工作，时刻等待被调试进程产生的事件或用户的手动干预。当被调试进程出现事件并进入暂停状态时，内核会向调试器发出信号。此时，调试器会对该事件进行检查，并基于事件类型采取相应的行动。

1.4　技巧和注意事项

在大多数情况下，使用调试器是直观的，无需了解太多的细节。只要调试器所依赖的所有内容都处于正确和良好的状态，它就能完美地工作。然而，有时候小小的问题就能给你带来一整天的麻烦。当调试器在你需要的时候不工作，那将会是极度令人沮丧的。更糟糕的是，它可能给出“错误”的信息，导致你得出错误的结论。当你花费大量时间去追究一个错误的根源，最终发现你最基本的假设就是错误的，这将会很令人挫败。在很多情况下，我们不应该错怪调试器本身，通常是我们自己的误解导致了这些困扰。

调试器会抱怨任何它不喜欢的事情。例如，如果源代码文件的时间戳比二进制文件晚，这可能意味着源代码已经被修改了；或者如果库文件的校验和与核心转储文件中指示的库文件不一致。如果我们忽视这些抱怨，调试器就可能像出了问题一样，比如程序不会在你设定的断点处停止；或者你不能捕获变量被意外修改的时刻；或者调用栈显然是混乱的，等。

另一方面，调试器具有许多工程师不了解的强大功能。在大多数情况下，我们只使用了所有功能中的一小部分，以处理常见的调试需求。但是，如果我们能花更多的时间去学习调试器的高级功能，那么我们会得到相应的回报。这将帮助我们更有效地进行调试，并解决那些偶尔会遇到的复杂问题。

1.4.1　特殊的调试符号

在之前的章节中，我们探讨了调试符号及如何在调试过程中使用这些信息。作为功能的扩展，我们可以在需要的时候向调试器添加更多的调试符号。这在我们已知某个变量的具体类型，但却无法打印该变量的情况下非常有用。

调试器无法理解变量的原因，主要是因为缺乏对应变量的调试符号。这在系统库、第三方库或遗留的二进制文件中并不罕见，因为这些库的符号可能被部分或完全剥离，或者在某些情况下，它们在编译时就没有生成调试符号。

解决这种问题的方法有两种：一种是重新编译并生成包含我们所需调试符号的新库文件；另一种方法将在第 9 章中介绍。当调试器成功加载新库文件的符号后，我们就能更好地调试这些二进制文件。下面，让我们通过一个第三方库的数据结构例子来看看具体的操作过程。

考虑一种情况，我们想要打印第三方库管理的一系列自由的内存块，这需要使用在头文件 mm_type.h 中声明的以下数据结构：

```
typedef struct _FreeBlock
{
   PageSize sizeAndTags;
   struct _FreeBlock *next;
   struct _FreeBlock *prev;
} FreeBlock;
```

我们首先编译上述文件，生成带有所有调试符号的目标文件：

```
gcc -g -c -fPIC -o mm_symbol.o mm_symbol.c
```

然后，我们把这个目标文件加入到调试会话中，这样就可以得到这个数据结构 FreeBlock 的类型符号。GDB 命令 add-symbol-file 可以从下面显示的目标文件中读取额外的调试符号，如下所示。在这里，地址参数 0x3f68700000 并不重要。输入的文件通常是共享库，但也可以是目标文件。你可以通过这种方式加入更多你需要的符号：

```
(gdb) add-symbol-file /home/myan/bin/sh_symbols.o 0x3f68700000
(gdb) print *(FreeBlock*)0x290c098
$1 = {
  sizeAndTags = 490,
  next = 0x290d560,
  prev = 0x290ffe8
}
```

这种方法为用户在使用调试器解释数据时提供了更大的灵活性。但请注意，这只能提供额外的类型信息，无法替换在原始二进制文件生成过程中确定的其他调试符号，如行号或变量位置。

可能使用最频繁的调试器功能就是断点设置，比如函数断点或者源代码行断点。然而在许多情况下，一个简单的断点可能是不够的。例如，当怀疑变量可能被错误修改时，常见的做法是在相关的地方设置多个断点或者在频繁执行的代码处设置断点。这可能导致另外一个问题，那就是断点会多次触发，这可能非常烦琐甚至不切实际，你甚至会因此错过你期盼的关键时刻，这是因为你需要在许多合法的状态中找到有错误的那一个，而人的注意力是有限的。

一种常见的解决方案是设置条件断点，即将特定的条件表达式与断点关联起来。当达到断点时，调试器计算这个表达式。如果计算结果为真，那么程序会停下来等待用户的操作；如果计算结果为假，那么程序会继续运行。

读者应该意识到条件断点的性能损耗。即使看起来程序在达到断点时（条件表达式为假）没有停下来，实际上程序还是会在每次达到断点时停止，并在计算表达式后由调试器恢复运行。如果这种开销过大，例如在频繁调用的函数中设置断点可能导致程序明显变慢，我们必须找到一种更快的方式来检查数据。比如，通过函数拦截可以避免调试器的介入（参看第 6 章获取更多细节）。

实际上有许多新颖的断点条件表达式，这是反映开发者经验水平的很好例子。以下是一些条件断点的例子：第一条命令告诉 GDB 在它停止程序之前忽略断点 100 次；第二条命令在变量 index 为 5 的条件下在函数 foo 入口停止；最后一条命令在指令地址 0x12345678 处设置断点，并附加条件函数 GetRefCount 返回值为 0，这种情况需要调试器调用一个函数来计算表达式。

```
(gdb)ignore 1 100
(gdb)break foo if index==5
(gdb)break *0x12345678 if GetRefCount(this)==0
```

断点可以设置在代码中，也可以设置在数据对象上。后者被称为监测点，或者数据断点。程序 bug 通常与特定的数据对象相关，并通过对这个对象的访问表现出来。在代码中设置断点的目的是允许我们检查可能错误更改数据的指令的程序状态。这种方法的一个明显不足之处是，它侧重于代码而非数据。被监视的代码可能会处理很多数据对象，大部分时间这些处理都是合法且正确的，除了可能出错的那个。所以，当怀疑的是特定的数据对象时，这种方法的覆盖范围太广，以至于无法有效地进行调试。

如果在正确的数据对象上设置监测点，我们就有更大的机会找到问题所在。当有太多可能错误修改数据对象的地方时，监测点是适用的。在这些情况下，代码断点可能无法提供太多帮助，因为它会过于频繁地停止程序，而这些停止并没有提供太多有用的信息。每当被监控的数据对象被覆盖或读取时，取决于监测点的模式，监测点会停止程序。所以，当我们知道某个数据对象是程序失败的关键，但不清楚何时和如何它被修改为无效状态时，这种方式是最有效的。监测点是一个强大的功能，它通过关注数据引用来定位程序失败。

在大多数情况下，设置断点和监测点都很直观。但是，如果调试器的介入对问题的复现产生了显著影响，那么就需要仔细考虑。断点和监测点使用不同的机制实现。

如前面所诉，调试器是通过将指定位置的指令替换为短陷阱指令来设置断点，原来的指令代码被保存在缓冲区中。当程序执行到陷阱指令时，也就是达到断点时，内核会停止程序运行并通知调试器，后者从等待中醒来，显示所跟踪程序的状态然后等待用户的下一条命令。如果用户选择继续运行，调试器会使用原来的代码替换陷阱指令，恢复程序运行。

监测点不能用指令断点的方法实现，因为数据对象是不可执行的。所以它的实现是要么定期地（软件模式）查询数据的值，要么使用 CPU 支持的调试寄存器（硬件模式）。软件监测点是通过单步运行程序并在每一步检查被跟踪的变量，这种方式使程序比正常运行要慢数百倍。

由于单步运行不能保证在多线程环境下结果的一致性，因此在多线程和多处理器的环境下，这种方法可能无法捕获到数据被访问的瞬间。硬件监测点则没有这个问题，因为被跟踪的变量的计算是由硬件完成的，调试器不用介入，它根本不会减慢程序的执行速度。但是硬件监测点在数量上是非常有限的，大多数 CPU 只有少数几个可用的调试寄存器。如果监测点表达式复杂或需要设置很多监测点，数据大小可能会超过硬件的总容量。在这种情况下，调试器会隐式地回归到软件监测点，这可能会导致程序运行变得极慢。所以你应该始终注意调试器是否在软件模式下设置了监测点。如果是这样，那么你可能需要调整你的调试策略。例如，你可以将复杂的数据结构分解为更小的部分，以便尽量使用硬件断点。

监测点可以设置与断点类似的条件。例如，以下的 GDB 命令在变量 sum 改变且变量 index 大于 100 时停止程序：

```
(gdb)watch sum if index > 100
```

这条命令可以解读为“监视变量 sum，如果 index 大于 100，则停止程序”。

虽然硬件断点对性能的影响较小，但计算条件表达式会带来与前文提到的同样的性能损耗。内核必须临时停止程序并与调试器通信，然后由调试器计算条件并确定下一步的操作。

1.4.2　改变执行与副作用

调试器的主要用途之一是观察被追踪进程的状态。然而，它也能够更改被调试进程的状态，从而改变其本来预设的执行路径。这种能力为调试带来无限的创新可能性。例如，若要验证当内存耗尽时程序会发生什么错误，调试器可以简单地设置 malloc 函数的返回值为 NULL。这是一种有效且低成本的方式，用于测试一些难以模拟或模拟成本较高的极端情况。

调试器提供了多种方式来改变程序的执行路径。最直接的方式是设置变量为新的值。调试器利用调试符号确定变量的内存地址，然后通过如 ptrace 方法以及内核的帮助，覆盖目标进程的内存。例如，下面的命令将变量 gFlags 的值设置为 5。

```
(gdb)set var gFlags=5
```

改变线程的上下文也会影响程序的执行。例如，程序计数器（下一条要执行的指令）可以设置为另一指令的地址。这个功能通常用于重新执行一段已经运行过的代码，以便更仔细地查看其运行过程。如果要重新执行的代码段中包含了断点，那么这个断点将再次被触发。例如，下面的命令将

当前线程的执行点移动到 foo.c 文件的第 123 行。

```
(gdb)jump foo.c:123
```

上述命令仅更改线程的程序计数器，线程上下文的其他部分保持不变。当前函数的栈帧仍然位于线程栈的顶部。这意味着该功能有一个限制，如果你使用上述 jump 命令跳转到了另一个函数的地址，那么根据两个函数的参数和局部变量的布局，结果可能是不可预见的。除非你对函数调用的所有细节都了如指掌，否则跳转到另一个函数通常是不明智的。

当被调试的进程已经暂停时，你可以在调试器内调用任何函数。调试器会在当前线程最内层的帧为被调用的函数创建一个新的栈帧。请注意，调用 C++类方法有些特殊，因为它"隐秘地"将 this 指针视为被调用函数的第一个参数，而调用 C 函数就简单直观一些。下面的例子中，调试器调用了函数 malloc 来分配一个 8 字节的内存块，并打印出返回的内存块地址。

```
(gdb) print /x malloc(8)
     $1 = 0x501010
```

如果被调用的函数有副作用，那么它可能在不易察觉的情况下改变了程序的行为。例如，下面的条件断点将启用临时跟踪和日志记录。每次变量 sum 的值发生改变时，GDB 命令都会调用 Logme 函数。

```
(gdb)watch sum if Logme(sum) > 0
```

1.4.3　符号匹配的自动化

我希望前面的讨论已经使你相信调试符号需要准确匹配才能使调试器真正有效。缺少正确的符号，调试器要么拒绝执行用户的命令，要么更糟糕，它可能给出错误的数据从而误导你走入歧途。从理论上讲，找到包含匹配符号的文件并不复杂，但如果一个产品包含许多模块，并且有许多要支持的版本、服务包、热修复和补丁，要手动找到正确的调试符号文件可能会变得烦琐且易于出错。在这种情况下，自动化查找正确的调试符号文件就显得更为必要。

Windows 符号服务器就是一个可以实现这种自动化功能的工具。这个工具的基本原理很简单。首先，将调试符号文件上传到称为符号存储的服务器。这些文件会按照时间戳、校验和、文件大小等参数进行排序和索引。每个文件都有多个版本，并且有不同的索引，以便进行快速查找。创建符号存储后，用户可以设置调试器的符号搜索路径使其包含符号存储服务器。调试器将会自动通过符号服务器获取到正确版本的符号文件。符号存储可以通过公司内部的 LAN 或全球互联网进行访问。例如，下面的符号搜索路径指向 Windows 的所有系统 DLL 的在线符号服务器。第一个星号后的路径指向一个已下载文件的本地缓存，这将加速已下载符号文件的搜索速度。第二个星号之后的 URL 指向微软的公共下载网址。

```
SRV*D:\Public\WinSymbol*http://msdl.microsoft.com/download/symbols
```

有了符号服务器的帮助，开发人员就无需再手动寻找正确的符号文件了。在 Linux 或 UNIX 上，长期以来都没有类似的统一工具，直到最近几年，出现了如 elfutils debuginfod 这样的工具，实现了

类似的功能。推荐读者阅读官方文档[1]来更好地设置它。

如果由于系统要求无法使用 debuginfod，可以利用其基本原理编写脚本来自动化此过程。例如，当各种版本的二进制文件安装在文件服务器的某个位置时，脚本可以创建一个临时文件夹，找到具有匹配调试符号的正确二进制文件，然后在这些临时文件夹中创建对应的软链接。调试器 GDB 可以设置将原始二进制文件的搜索路径映射到新的临时文件夹，从而获取匹配的符号。

1.4.4　后期分析

调试器可以跟踪正在运行的进程，也可以对由系统例程在进程崩溃时生成的核心转储文件（Core Dump）进行调试。系统还为应用程序或工具软件提供了 API，可以在不终止目标进程的情况下生成进程的核心转储文件，这对于调查不间断运行的服务器程序的性能或者对难以访问的远程程序的线下分析非常有用。

核心转储文件基本上是进程内存映像在某一刻的快照。调试器可以将其视为一个正在运行的进程，例如，我们可以检查内存内容、列出线程的调用栈、打印变量等。然而，需要明白的是，核心转储文件只是一个静态磁盘文件，它与活动进程有本质上的区别，因为在宿主机器的内核中并没有相应的进程运行的上下文。

这样的进程不能被调度到任何 CPU 上运行，用户也无法在核心转储文件中执行任何代码。进程的状态只能被查看，无法被改变。这就意味着我们无法调用函数，也无法打印需要调用函数的表达式，例如类的运算符函数。一个常见的让人感到困惑的例子是，调试器拒绝打印像 vec[2]这样的简单表达式，其中 vec 是一个 STL 向量。这是因为调试器需要调用 std::vector 的 operator[]方法来计算表达式。出于同样的原因，我们也不能在后期分析中设置断点或单步执行代码。

核心转储文件有一个标志位表明它是为什么生成的，这也是你最先想知道的事情（我们将在第 6 章讨论更多核心转储文件的结构体细节）。一些常见原因是：

（1）段错误。内存访问越界或者数据内存保护陷入。它表示程序正在试图访问未分配给进程的地址空间或者被保护免于特定的操作（读、写或者是运行）的内存。正在运行的指令试图从这个地址读取或者向这个地址写入，因而被硬件异常捕获。比如，悬空指针指向已经释放的内存块和野指针指向随机地址；使用其中任何一个访问内存可能导致段错误。

（2）总线错误。这个错误通常由访问未对齐的数据导致。比如，从奇数地址的内存读取整数。一些体系结构允许这样的行为，但可能带来潜在的性能消耗（x86），而其他体系结构（SPARC)则会以总线错误异常让程序崩溃。

（3）非法地址。当程序下一个运行指令不属于 CPU 指令集，这个异常会被抛出。例如，一个函数指针持有一个不正确的地址，该地址落入堆段而不是文本段。

（4）未处理的异常。当 C++程序抛出异常且没有代码来捕获它时，就会发生此错误。C++运行库有一个默认的处理函数来捕获异常，它的作用是生成一个核心转储文件然后终止程序。

[1] https://sourceware.org/elfutils/Debuginfod.html

（5）浮点数异常。除以 0，太大或者太小的浮点数都可能会导致这个错误。

（6）栈溢出。当程序没有足够的栈空间来存储函数调用和本地变量时，就会发生栈溢出。这种情况通常发生在函数被递归调用次数过多，或者函数使用了过多的本地变量的情况下。

后期分析的一种常见问题是核心转储文件不完整或被截断，这种情况下我们只能看到调试对象的部分内存图像。这通常会阻止我们得出崩溃原因的结论，因为一些重要的数据对象是不可访问的。例如，调试器可能无法显示堆上的相关数据对象或线程的调用栈，因为与它们相关的内存没有保存在核心转储文件中。

完整的核心转储文件的大小与运行的进程的内存大小大致相同，其中不包含有本地磁盘备份的加载文件，比如可执行的二进制文件。核心转储文件被截断的原因有很多，例如，系统默认设置仅允许部分核心转储；系统管理员可能将最大核心转储文件大小设置为较低的值，以避免磁盘使用过度；核心转储设备上剩下的磁盘空间不足；或用户的磁盘配额超过了限制。如果任何上面的条件没有被满足，系统会选择一部分内存镜像保存起来而丢弃剩下的部分。

由于核心转储文件不包含任何二进制代码，而只记录每个可执行文件或库的名称、大小、路径、加载地址和其他信息，当我们在另外一台机器上用调试器加载和分析核心转储文件时，这些二进制文件可能缺失或者安装在不同的路径。用户需要负责设置正确的二进制文件路径并告知调试器。如果根据用户提供的搜索路径，找到了不匹配的二进制文件，调试往往打印出警告信息但是不会停止，结果可能导致误导。这跟前面章节讨论的符号匹配是一样的。

1.4.5　内存保护

在一些平台上，如 HP-UX，用户可能无法在已加载的共享库中设置断点。这是因为共享库默认被加载在仅可读的公共内存段中，如果允许被某个进程覆盖，必然会影响到与之共享的别的进程。因此调试器无法在代码段插入断点的陷入指令。有多种方法改变这个默认行为，使用户可以修改共享库加载的模式。

下面的 HP-UX 命令在输入的模块中设置标志让系统运行时将模块加载到私有的可写段中。

```
chatr +dbg enable <modules>
```

系统加载器还会读取以下环境变量，并将所有模块加载到私有的可写段中。

```
setenv _HP_DLDOPTS -text_private
```

我们也可以指定特定模块加载到私有、可写的段中。

```
setenv _HP_DLDOPTS -text_private=libfoo.sl;libbar.sl
```

1.4.6　断点不工作

如果程序没有在预设的断点处停下，你可以根据以下列表逐一排查，确保断点的正确设置。

（1）调试器读到的源代码与调试符号不匹配。常见原因是源代码文件在编译生成二进制以后

又有新的修改。调试符号包含的源代码文件路径是在二进制构建时候的，但是它并不包含源代码的实际内容。除非用户指定另外的源码搜索路径，调试器会从调试符号里面的路径来加载源代码文件。如果源码文件的时间戳比二进制创建时间戳更新，调试器会发出一个警告信息。如果忽略这个警告，那么调试器看到的源代码行将不会与调试符号中的源代码行匹配，这并不罕见，因为警告消息可能被淹没在大量的其他信息中。当调试器被要求在某个特定行设置断点时，它实际上可能会把陷入指令插入了不同的行。

（2）如果断点将要设置在一个共享库中，则在库被映射到目标进程的地址空间之前，调试器不能够插入陷入指令。如果你希望调试库的初始化代码，这是很困难的，因为当我们有机会设置断点的时候，通常有点晚了。比如，调用函数 dlopen 或者 LoadLibrary 可以动态加载库，但是当函数返回时，库的初始代码已经执行完成了。幸运的是，像 GDB 这样的调试器可以将断点设置推迟到库加载到进程中时。当库文件被加载到目标进程里但是在执行任何代码之前，内核将向调试器发送一个事件。这使调试器有机会检查其延迟断点并正确设置它们。Windows Visual Studio 支持在“项目设置”对话框的“调试”选项卡上添加其他 DLL，允许用户在要加载的 DLL 中设置断点。

（3）如果启用了优化，编译器可能会打乱源代码的执行顺序。因此，调试器可能无法完全按照用户的意愿在源代码的某行设置断点。在这种情况下，最好在函数入口或指令级别设置断点，以便可靠地触发断点（有关更多详细信息，请参见第 5 章）。

1.5　本章小结

使用调试器是程序开发人员和其他某些工程师必备的基本技能之一。调试器通常具有大量命令，取决于调试器的实现和宿主系统的能力，它的许多功能都可能显著影响被调试的进程，有些非常有用，而有些会改变程序的行为干扰调试的预期，这要求使用者对这些功能有较深入的理解。除了常用的命令外，为了更有效率地使用调试器，我们需要学习更多调试器高级的功能。当问题变得更复杂和在影响范围更大时，希望调试器具有更多能力的需求也越来越多。自定义调试器命令和插件是解决这个挑战的办法。在接下来的章节，我们将看到更多关于调试器插件的示例。

第 9 章

拓展调试器能力

调试器在设计上需要支持尽可能多的程序和场景的调试。尽管像 GDB 和 Windbg 这样的工具提供了丰富而强大的功能，但随着你对这些工具的熟练程度的提高以及所遇到问题的复杂性的增加，可能会发现自己需要更多的功能。例如，你可能希望调试器能够解析并显示某些特定于应用程序的数据，或者希望调试器能够自动化重复的任务，或者可能只是想自定义调试器以满足你的特定需求。另一方面，调试器的实现往往与特定的体系结构和平台紧密相关。我们通常使用由操作系统或编译器供应商提供的调试器，这些调试器所支持的功能集可能会有所不同。你可能会发现某个特定的调试器缺少需要的功能。

这时，调试器的自定义命令就可能派上用场。自定义命令大致上可以分为两种类型：命令脚本和调试器插件。这两者都需要调试器的支持才能进行底层操作。命令脚本是由一系列调试器命令组成的，通过流程控制关键字进行组织，可以方便地进行临时编写。而调试器插件涉及的内容更多，能力也更强大。它可以访问调试器后端引擎的更多内部结构，一旦实现了良好的插件命令，你的工作将更加轻松。在本章中，我将介绍如何通过这两种方式向 GDB 添加新功能。你将能够在提供的源代码链接中中找到许多实用功能的实例。

9.1 使用 Python 拓展 GDB

本节将重点介绍如何利用 Python 来提升你在 GDB 中的调试技能，并帮助摆脱烦琐的重复任务，享受更自由的编程环境。

首先，确保你正在使用的是 GDB 7.x 或更高版本。这是因为 Python 的支持是从 GDB 7.0（2009 年）开始提供的。

你可能会问，既然 GDB 已经支持自定义脚本辅助调试，为什么还要使用 Python 脚本呢？这是因为 GDB 的自定义脚本语法相对较老，使用 Python 编写将会更加流畅和高效。当然，如果你仍然更倾向于使用原有的自定义脚本，那么也完全可以。

美化输出，利用 Python 可以把复杂难懂的数据展示得更加清晰易读；自动化工作，利用 Python 可以把重复的任务简化为一个简单的命令；高效调试，利用 Python 可以更高效地进行 bug 调试，比如定制遇到断点的行为。

9.1.1　美化输出

以下面的代码为例：

```
#include <map>
#include <iostream>
#include <string>
using namespace  std;

int main() {
    std::map<string, string> lm;
    lm["good"] = "heart";
    // 查看 map 里面内容
    std::cout<<lm["good"];
}
```

当代码运行到 std::cout<<lm[“good”]时，你想查看 map 里面的内容，如果没有 python 和自定义的脚本，print lm 看到的内容类似如下：

```
$2 = {_M_t = {
    _M_impl = {<std::allocator<std::_Rb_tree_node<std::pair<std::__
cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > const,
std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > > > >>
=
{<__gnu_cxx::new_allocator<std::_Rb_tree_node<std::pair<std::__cxx11::basic_string<char
, std::char_traits<char>, std::allocator<char> > const,
std::__cxx11::basic_string<char, std::char_traits<char>, std::allocator<char> > > > >>
= {<No data fields>}, <No data fields>},
<std::_Rb_tree_key_compare<std::less<std::__cxx11::basic_string<char,
std::char_traits<char>, std::allocator<char> > > >> = {
        _M_key_compare = {<std::binary_function<std::__cxx11::basic_string<char,
std::char_traits<char>, std::allocator<char> >, std::__cxx11::basic_string<char,
std::char_traits<char>, std::allocator<char> >, bool>> = {<No data fields>}, <No data
fields>}}, <std::_Rb_tree_header> = {_M_header = {
          _M_color = std::_S_red, _M_parent = 0x55555556eeb0,
          _M_left = 0x55555556eeb0, _M_right = 0x55555556eeb0},
        _M_node_count = 1}, <No data fields>}}}
```

但是当你在 GDB 9.2 里面输入 print lm 的时候，看到的将是：

```
(gdb) p lm
$3 = std::map with 1 element = {["good"] = "heart"}
```

map 里面有什么一清二楚。这是因为 GDB 9.x 自带了一系列标准库的 Python Pretty Priniter。如果你使用的是 GDB 7.x，那么可以手动的导入这些 Pretty Printer 实现同样的效果。具体步骤如下：

步骤 01 下载 Pretty Printer: svn co svn://gcc.gnu.org/svn/gcc/trunk/libstdc++-v3/python[1]。

步骤 02 在 GDB 里面输入（将路径改成你下载的路径）：

```
    ```python
    python
    import sys
    sys.path.insert(0, '/home/maude/gdb_printers/python')
    from libstdcxx.v6.printers import register_libstdcxx_printers
    register_libstdcxx_printers (None)
    end
    ```
```

这样你就可以使用了，如果需要更详细得步骤请看链接[2]。

9.1.2　编写你自己的美化器

接下来，我们详细介绍一下如何编写“美观打印器”（Pretty Printer），用于显示自己的数据结构。例如，你有一个包含多个数据成员的结构体：

```
struct MyStruct {
  std::name mName;
  std::map mField1;
  std::set mField2;
  int mI;
  int mj;
};
```

然而，你在打印此结构体时，大部分时候只关注 mName 和 mI 字段。此时，你就可以定义一个针对该数据结构的“美观打印器”，这样大部分时候你看到的只是你关心的那部分字段，而无需在几十个字段中寻找。

如果不使用任何的“美观打印器”，打印一个 MyStruct 数据结构的效果可能如下所示：

```
$2 = {mName = {static npos = <optimized out>,
    _M_dataplus = {<std::allocator<char>> = {<__gnu_cxx::new_allocator<char>> =
{<No data fields>}, <No data fields>},
      _M_p = 0x618c38 "student"}}, mField1 = {_M_t = {
      _M_impl = {<std::allocator<std::_Rb_tree_node<std::pair<int const,
std::basic_string<char, std::char_traits<char>, std::allocator<char> > > > >> =
{<__gnu_cxx::new_allocator<std::_Rb_tree_node<std::pair<int const,
std::basic_string<char, std::char_traits<char>, std::allocator<char> > > > >> = {<No
data fields>}, <No data fields>}, <std::_Rb_tree_key_compare<std::less<int> >> = {
```

[1] 如果链接失效了，可以查看 GCC 的文档找到新的链接

[2] https://sourceware.org/gdb/wiki/STLSupport
https://codeyarns.com/2014/07/17/how-to-enable-pretty-printing-for-stl-in-GDB/

```
            _M_key_compare = {<std::binary_function<int, int, bool>> = {<No data
fields>}, <No data fields>}}, <std::_Rb_tree_header> = {
            _M_header = {_M_color = std::_S_red, _M_parent = 0x0, _M_left =
0x7fffffffe4e0, _M_right = 0x7fffffffe4e0},
            _M_node_count = 0}, <No data fields>}}}, mField2 = {_M_t = {
         _M_impl = {<std::allocator<std::_Rb_tree_node<std::basic_string<char,
std::char_traits<char>, std::allocator<char> > > >> =
{<__gnu_cxx::new_allocator<std::_Rb_tree_node<std::basic_string<char,
std::char_traits<char>, std::allocator<char> > > >> = {<No data fields>}, <No data
fields>}, <std::_Rb_tree_key_compare<std::less<std::basic_string<char,
std::char_traits<char>, std::allocator<char> > > >> = {
            _M_key_compare = {<std::binary_function<std::basic_string<char,
std::char_traits<char>, std::allocator<char> >, std::basic_string<char,
std::char_traits<char>, std::allocator<char> >, bool>> = {<No data fields>}, <No data
fields>}}, <std::_Rb_tree_header> = {_M_header = {
              _M_color = std::_S_red, _M_parent = 0x0, _M_left = 0x7fffffffe510,
_M_right = 0x7fffffffe510},
            _M_node_count = 0}, <No data fields>}}}, mI = 3, mj = 4}
```

看起来会让人感到困惑，因为信息过于复杂。

如果使用 GDB 自带的 STL 美观打印器，那么我们会得到如下简洁的结果：

```
(gdb) p s
$1 = {mName = "student", mField1 = std::map with 0 elements, mField2 = std::set
with 0 elements, mI = 3, mj = 4}\
```

如果自己编写美观打印器，那么就会得到如下的结果：

```
(gdb) p s
$2 = MyStruct
 name: "student"  integer: 3
```

这样，只会打印自己关心的数据。如果你希望查看原始的数据，那么可以使用 p /r s 命令。

美观打印器的实现思路如下：

（1）定义打印类，提供 to_string() 方法，该方法返回你希望打印出来的字符串。

（2）创建判断函数，判断一个值是否需要使用你定义的类来打印。

（3）将你的判断函数注册到 GDB 美观打印函数中。

编写打印类的代码如下：

```
class MyPrinter:
    def __init__(self, val):
        self.val = val
    def to_string(self):
         return "name: {}  integer: {}".format(self.val['mName'], self.val['mI']
```

从这简单明了的代码中，我们可以看到，val 存储的是对应类的值。我们可以访问对应的类成

员。

判断函数的代码如下：

```
#判断一个 value，是否需要使用自己定义的打印类
def lookup_pretty_printer(val):
    if val.type.code == gdb.TYPE_CODE_PTR:
        return None # to add
    if 'MyStruct' == val.type.tag:
        return MyPrinter(val)
    return None
```

注册到 GDB 中，则使用如下的函数：

```
gdb.printing.register_pretty_printer(
    gdb.current_objfile(),
    lookup_pretty_printer, replace=True)
```

完成上面的步骤和代码以后，可以编译如下程序，并测试是否成功：

```
struct MyStruct {
  std::string mName;
  std::map<int, std::string> mField1;
  std::set<std::string> mField2;
  int mI;
  int mj;
};

int main() {
   MyStruct s = {std::string("student"), lm, ls, 3, 4}
   return 0;
}
```

9.1.3　将重复的工作变成一个命令

例如，在调试过程中，如果你知道当前的栈指向了一个字符串，但是并不清楚它具体在哪里，想要遍历栈以找到它。此时，可以使用 Python 自定义一个名为"stackwalk"的命令。这个命令可以直接利用 Python 代码遍历栈，将目标字符串找出。

```
#####################################################
#用法：将它加载到 GDB 中运行：
# (gdb) source ..../path/to/<script_file>.py

Import gdb

class StackWalk(gdb.Command):
    def __init__(self):
        # 将我们的类注册为“StackWalk”
```

```
            super(StackWalk, self).__init__("stackwalk", gdb.COMMAND_DATA)

        def invoke(self, arg, from_tty):
            # 当我们从 GDB 调用 StackWalk 时，这个方法将要被调用
            print("Hello from StackWalk!")
            # 获取寄存器
            rbp = gdb.parse_and_eval('$rbp')
            rsp = gdb.parse_and_eval('$rsp')
            ptr = rsp
            ppwc = gdb.lookup_type('wchar_t').pointer().pointer()
            while ptr < rbp:
                try:
                    print('pointer is {}'.format(ptr))
                    print(gdb.execute('wc_print
{}'.format(ptr.cast(ppwc).dereference())))
                    print('===')
                except:
                    pass
                ptr += 8


    # 在“源”时间将我们的类注册到 GDB 运行时.
    StackWalk()
```

wc_print 是我写的另外一个简单 Python 命令，用于打印给定地址的宽字符串。

9.1.5　使用 Python 设置断点

如果我们在调试过程中遇到以下情况，应该如何设置断点呢？

- 需要在某个变量被修改时，将其修改的调用栈打印出来。
- 某个变量会被多次修改，只对某个特定时间点之后的修改感兴趣，而对之前的修改不感兴趣，那么如何设置断点？
- 如果觉得盯着屏幕等待条件满足时设置断点过于烦琐，那么有什么方法可以在需要我们关注的时刻，也就是期望的条件被满足时自动设置断点呢？

我们以 GDB 为例进行说明，其他的调试器也有类似的功能，如果对此感兴趣，你可以查阅相应调试器的使用文档。

首先，对于第一个需求，我们可以通过使用 data point 来实现。这一功能在 Visual Studio 和 GDB 中都有支持，在 GDB 中被称为 watch point。对于经常在同一个地方设置断点和操作，我会倾向于使用 Python script 来创建断点。

对于第二个需求，我们可以首先设置一个条件断点，等到条件断点满足时，再设置 data point。

两者结合起来的 Python 脚本如下。尽管代码看起来较长，但实际上非常直观，容易理解。

```
    try:
        import gdb
    except ImportError as e:
        raise ImportError("This script must be run in gdb: ", str(e))
    '''
    First define the global settings
    '''
    creation_function = 'after_this_function()'
    creation_breakpoint = None
    watch_point = None
    symbol = 'referenceCount'

    def print_stack_trace():
        gdb.execute('bt')

    class CustomWatchPoint(gdb.Breakpoint):
        '''
        gdb watchpoint expression '..' doesn't work. It will complains 'You may have
requested too many hardware breakpoints/watchpoints.'
        The workaround is set wp by address, like 'watch *(long *) 0xa0f74d8'
        '''

        def __init__(self, expr, cb):
            self.expr = expr
            self.val = gdb.parse_and_eval(self.expr)
            self.address = self.val.address
            self.ty = gdb.lookup_type('int')
            addr_expr = '*(int*)' + str(self.address)
            gdb.Breakpoint.__init__(self, addr_expr, gdb.BP_WATCHPOINT)
            self.silent = True
            self.callback = cb

        def stop(self):
            addr = int(str(self.address), 16)
            val_buf = gdb.selected_inferior().read_memory(addr, 4)
            val = gdb.Value(val_buf, self.ty)
            print('symbal value = ' + str(val))
            self.callback()
            return False

    class CustomBreakPoint(gdb.Breakpoint):
        '''
        gdb breakpoint expression
        '''
```

```
    def __init__(self, bp_expr, cb, temporary=False):
        # spec [, type ][, wp_class ][, internal ][, temporary ][, qualified ])
        gdb.Breakpoint.__init__(
            self, bp_expr, gdb.BP_BREAKPOINT, False, temporary)
        self.silent = True
        self.callback = cb

        def stop(self):
            self.callback()
        return False

class CustomFinishBreakpoint(gdb.FinishBreakpoint):
    def stop(self):
        print("normal finish")
        global watch_point
        try:
            if watch_point is None:
                watch_point = CustomWatchPoint(symbol, print_stack_trace)
            if not watch_point.is_valid():
                print("Cannot watch " + symbol)
            else:
                print("watching " + symbol)
        except RuntimeError as e:
            print(e)

        return False

def out_of_scope():
    print("abnormal finish")

class CreateWatchPointCommand(gdb.Command):
    '''
        A gdb command that traces memory usage
    '''
    _command = "watch_ref"

    def __init__(self):
        gdb.Command.__init__(self, self._command, gdb.COMMAND_STACK)

    def invoke(self, argument, from_tty):
        '''
        1. Try to create the watchpoint
        2. If fail, then create a breakpoint after the variable has been created.
        '''
        global symbol
```

```
        global watch_point
        global creation_breakpoint
        watch_point = None
        creation_breakpoint = None

def watch():
    customFinishPoint = CustomFinishBreakpoint()
    try:
        creation_breakpoint = CustomBreakPoint(creation_function, watch)
    except Exception as e:
        print(e)

gdb.execute("set pagination off")
gdb.execute("set print object on")
gdb.execute("handle SIGSEGV nostop noprint pass")
CreateWatchPointCommand()
gdb.execute('watch_ref')
gdb.execute('set logging file debug-logging.txt')
gdb.execute('set logging on')
gdb.execute("c")
```

9.1.6　通过命令行来启动程序和设置断点

对于第三个需求，可以通过将 bash 脚本、GDB 以及 Python 三者结合起来实现。GDB 支持在启动程序的时候执行一段代码，因此我们可以让 GDB 在运行的时候执行 Python 脚本以实现自动设置断点。

一个简单的命令行示例如下，它使用一个循环，在感兴趣的程序出现时，会自动 attach 并且设置相应的断点。这个实现方法的步骤如下：

```
gdb -p <pid> -q -x scripts/watch_ref.py
```

-q 表示不打印 GDB 启动时的输出信息。

-x <script>表示 GDB 启动时候执行脚本 script。

如果你喜欢还尝试其他方法，可以试试第二种方法：

```
#!/bin/bash
function get_pid()
{
       ps -ef | grep -v grep | grep multip| awk '{print $2}'
}

proc_pid=''
while [ -z "$proc_pid" ]
do
proc_pid=$(get_pid)
```

```
    done;    /usr/local/bin/gdb -p $proc_pid -q -x scripts/watch_ref.py
```