第 15 章 代理测试与调试

本章主要讲述以下的内容：

- Net-SNMP 代理测试方法；
- Net-SNMP 代理原生调试方法
- GDB 在 Linux C/C++程序中调试方法与技巧；
- Linux 常用（网络）调试工具与方法；

测试和调试是开发人员非常关注和喜欢的内容！至少笔者是这样的。有时候是为了测试和调试不得不接触和学习，有时候则是出于好奇心，想研究代码真实的运行路线和内部细节。

调试的过程或许很漫长，甚至让自己苦不堪言，情绪低落，但往往会柳暗花明，灵光闪现，这种体验非常有意思。当然，本章所述的行为非常有趣，我们不讲述测试部门的测试人员准备测试用例，时刻重复着测试用例的行为，而是讲述调试的方法和技巧。实际上，每一位开发人员几乎从编程的那一刻起，就没有将编写代码和调试或测试代码分开来过。所以，测试和调试本身就是开发人员必须要掌握的技能。当你偶尔能发出“大招”解决令伙伴们焦头烂额的 bug 时，这种成就感比编写代码所带来的成就感来得更激烈和痛快吧！

测试或调试涉及到的内容非常多，也有很多相关的书籍和资料。在 Linux 下的调试就不得不说 GDB，其官方手册就超过 500 页，简直就是一部“葵花宝典”，令人“欲罢又不能”。

在本章中，笔者将针对 Net-SNMP 直接给出一些干货！

15.1代理测试方法与技术

测试过程的本质是模拟程序运行的过程，并观察程序是否正确的反应。SNMP 代理的测试同样遵循这样的一个过程。测试 SNMP 代理我们需要准备好管理站 NMS，并按测试方案和方法进行测试。SNMP 协议历史悠久，应用广泛，市面上早已有很多的 SNMP 管理端的软件，它们大多都是商业软件，或者个人评估版（功能有所限制）。这些软件使用方便，有助于代理的测试。本章接下来介绍了两个笔者认为不错的管理端——代理测试软件。当然，我们也可以使用或改进第 8 章中开发的 NMS 程序。这些软件或工具都可以辅助我们测试 SNMP 代理！

进行测试 Net-SNMP 代理前，我们至少需要设计如下的测试方案：

- 代理可正常进行 SNMP 通讯；
- 代理支持 SNMPv1, SNMPv2c，SNMPv3 协议操作；
- 代理支持 Get 操作；
- 代理支持 Get-Next 操作；
- 代理支持 Set 操作；
- 代理支持 Get-Bulk 操作；
- 代理支持 Walk 操作；
- 代理支持 SNMPv1 Trap，SNMPv2，v3 Notification；
- SNMPv1，SNMPv2 共同体测试、SNMPv3 认证和加密测试；
- 代理响应请求延迟测试；
- 旧模块的功能性测试；
- 新模块的功能和业务测试（新开发模块的重点测试工作）；
- 子代理组网及协议相关测试（测试子代理）；
- 动态库模块相关测试（测试以动态库模式开发的代理）；
- 其他相关的测试：性能测试、稳定性测试等。

在 SNMP 协议的测试或调试过程中，会见到如下的错误号。在此将其列出来，方便测试过程中比对。请读者根据字面名称理解其含义！

```
/*
* in SNMPv2p, SNMPv2c, SNMPv2u, SNMPv2*, and SNMPv3 PDUs
*/
#define SNMP_ERR_NOACCESS      (6)
#define SNMP_ERR_WRONGTYPE     (7)
#define SNMP_ERR_WRONGLENGTH   (8)
#define SNMP_ERR_WRONGENCODING         (9)
#define SNMP_ERR_WRONGVALUE    (10)
#define SNMP_ERR_NOCREATION    (11)
#define SNMP_ERR_INCONSISTENTVALUE    (12)
```

```
#define SNMP_ERR_RESOURCEUNAVAILABLE (13)

#define SNMP_ERR_COMMITFAILED (14)

#define SNMP_ERR_UNDOFAILED   (15)

#define SNMP_ERR_AUTHORIZATIONERROR  (16)

#define SNMP_ERR_NOTWRITABLE  (17)
```

15.1.1 使用 MG Soft MIBbrowser 测试

在 4.5.2 节中我们提到了 MG-SOFT（http://www.mg-soft.si/）套件中的 MIB Builder。由 MIB Builder 建立的 MIB 文件，经 MIB Complile 后加载到 MIB browser 就可以进行 MIB 的浏览了。实际上 MIB browser 不仅是浏览 MIB 的工具，同时也是一款测试代理的 NMS！下面我们简要的介绍该软件的使用。

- MIB 编译：因为 MG-SOFT 中使用自有的 MIB 二进制格式，所以我们需要将编写好的 MIB 文本文件使用 MIB Complile 编译并保存。操作过程：打开 MG-SOFT MIB Compiler 软件、打开 MIB 文件、编译并保存。当然，有语法错误的 MIB 是无法编译通过的！
- MIB 加载：加载的操作界面如下图所示。图中下方为系统中的所有 MIB 文件，包括我们刚才编译通过和保存的 MIB。中间向上的箭头表示加载下方选中的 MIB。

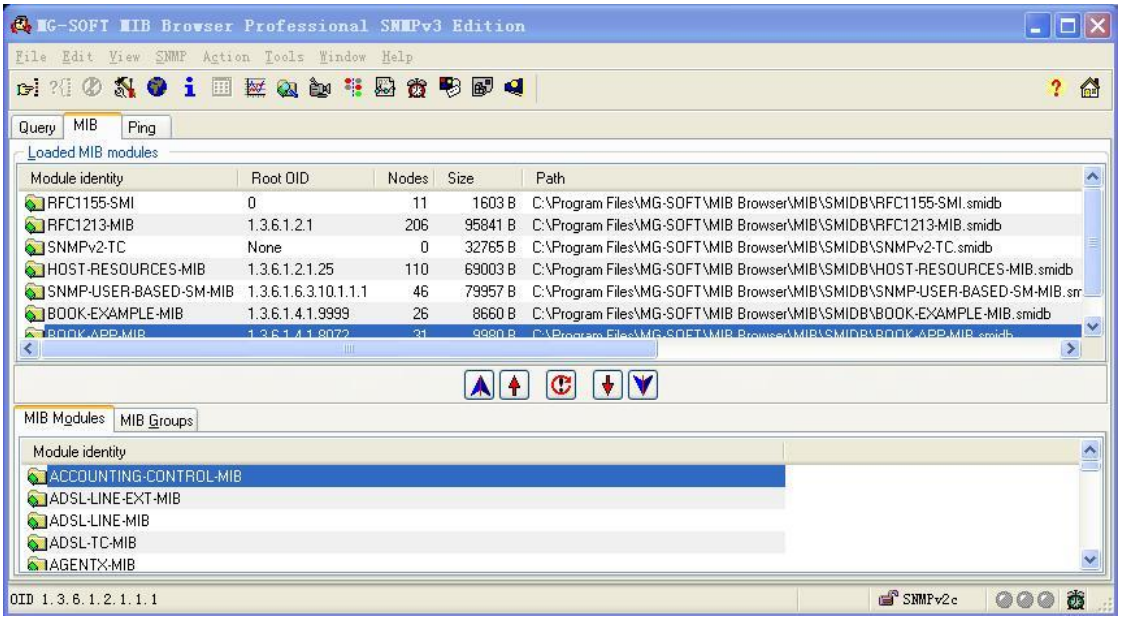

图15-1　　MIB 加载

- NMS 配置：在测试前，我们需要配置 NMS 监控代理的 IP 地址、端口、协议版本。这些配置是 NMS 监控代理的通讯的必要条件，如图 15-2 到 15-4。

图15-2　　配置代理 IP 地址

由菜单“View->SNMP Protocal Preferences…”进入到配置 SNMP 版本界面：

图15-3　　配置协议、端口、共同体、密钥等

选择 v3 版本时，需要点击按钮“Load user profile…”新建 v3 用户、编辑现有用户使用“Edit user…”进入编辑对话框，如图 15-4。

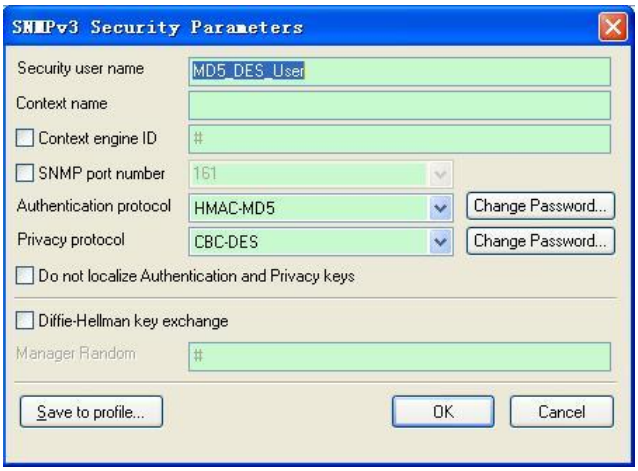

图15-4　　SNMPv3 用户配置对话框

- 操作方法：将 OID 树展开，找到目的 OID，右键操作，如图 15-5。先“Contact”上代理，再进行 SNMP 相关的操作。右侧将显示返回的结果。

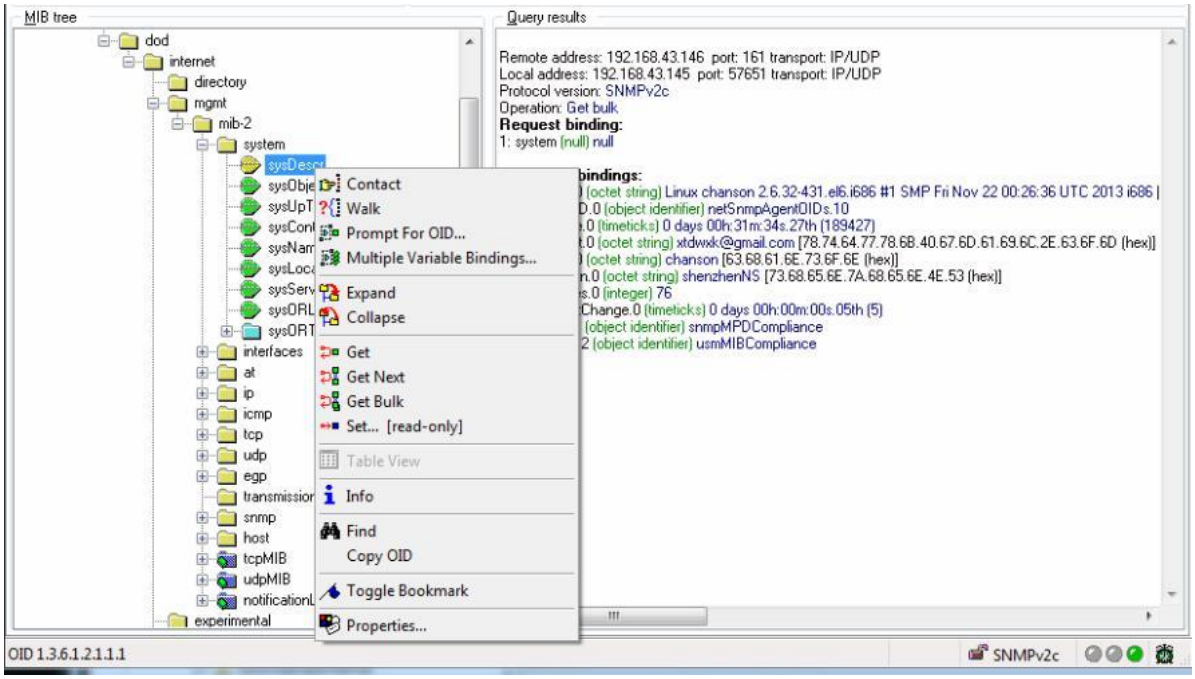

图15-5　　MG Soft MIB browser 操作

- Trap 查收：Trap 控制台在菜单选项下“Tools->Trap Ringer Console”，或者点击工具栏中的闹钟样的图标。如图 15-6。图中左侧显示的 Trap 信息包括：Trap 时间、Trap 节点、版本、Trap 类型、源地址、目的地址及端口号。右侧则为 Trap 中详细的内容（包括 Trap 中的变量绑定）。

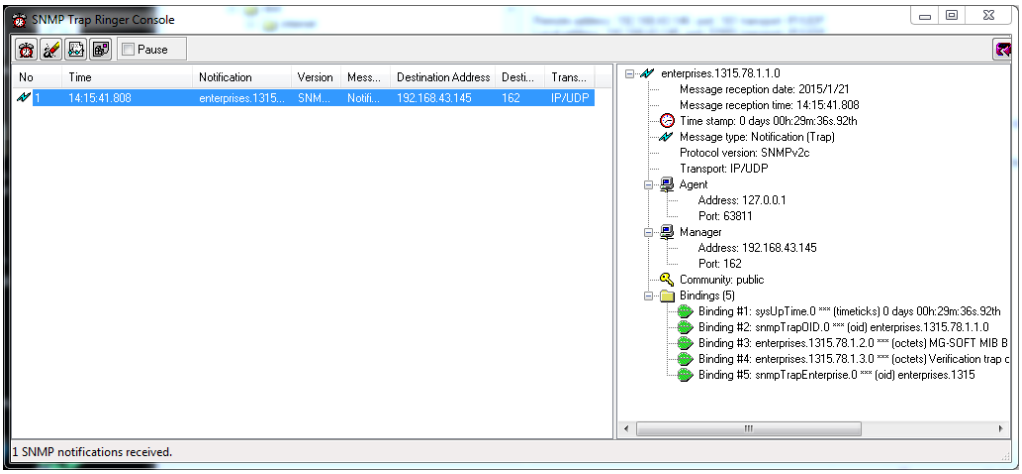

图15-6　　MG Soft Trap Console

15.1.2 使用 iReasoning 测试

从操作上来说，iReasoning（http://www.ireasoning.com/）不需要编译 MIB，直接加载文本形式的 MIB 文件即可，实为方便。其配置和操作方法与 MG Soft MIBbrowser 类似，在此不再细说。iReasoning 的左侧界面详细列出 MIB 节点的信息，非常直观！同时，它支持结果的排序和导出表格，相对 MG Soft MIBbrowser 来说是一大优势！其主界面如下：

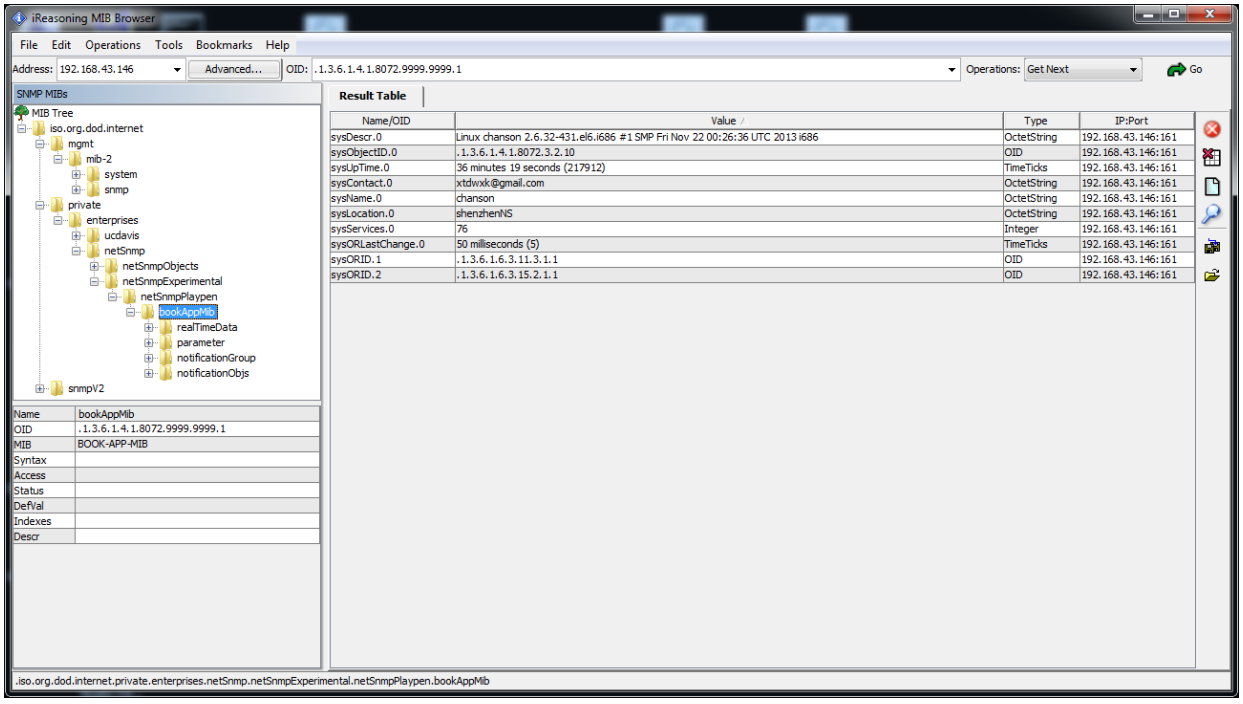

图15-7　　iReasoning 主界面

15.1.3 编写脚本测试

以上我们使用了两款软件作为模拟的 NMS 对代理进行测试，实际的测试过程是手动对节点一一进行测试和比对测试的结果！很明显，它是一项手眼并用，重复性的劳动！在大型软件测试中这种测试效率极低甚至不可接受！在测试理论和测试方法学成熟的今天，有很多测试的实践方法。其中自动化测试是互联网和各大 IT 型企业应用广泛的方法。自动化测试包含测试过程的自动化和测试结果分析的自动化。其最常见的实践方法莫过于使用自动化测试脚本，实现批量执行测试用例和测试结果的分析！

自动化实际上是模拟上述手动测试的过程，并且可以实现测试过程的复用，其优点不言而喻。当然，自动化测试脚本中核心的内容是测试用例，它是测试的关键所在！针对测试 SNMP 代理，如果采用自动化测试，最便捷的方法是使用 Net-SNMP 中命令工具集，诸如 snmpget，snmpset 等；以及使用绑定的脚本模块（Python 和 Perl 模块）开发相应的自动化测试脚本。它们不仅具有原生的操作 SNMP 协议的 API，重要的是 Python 和 Perl 语言本身的强大，适合编写具有一定的测试逻辑的结构化的测试脚本。

使用脚本测试，每次测试时只要执行一个脚本文件就能执行完所有的测试用例，完成大部分的测试工作。当然该脚本文件是主测试脚本，调用其他模块化的子脚本。这些子脚本可以是按照 MIB 节点分类，亦可以按照代理功能或业务分类。这样代理业务有所变化时，只要变更部分的脚本文件中的部分内容。如果是这样，我们的测试工作就步入了正轨！

使用 Python 和 Perl 开发自动化测试脚本，请参考第 10 章和第 11 章相关的内容。

15.2程序调试方法概述

被誉为计算机软件第一夫人的 Grace Hopper 曾说过：“From then on，when anything went wrong with a computer，we said it had bugs in it.”（Rear Admiral Grace Hopper，coined the term debugging in 1945）。调试即修改软件 bug 的一个过程，涉及到调试的原理、方法和技术与技巧。调试相关的内容非常丰富，完全可以编写出一本厚厚的书籍。本章的重点将聚焦于 Net-SNMP 中的调试方法、工具和技巧。

程序调试的方法多种多样，常见的如下：

1、使用内存转储，即 coredump，通过分析程序发生错误那一刻，导致 dump 时生成的内存转储，找到程序出错的蛛丝马迹。

2、打印：将待调试的程序中的关键位置或者所关心的位置，人为的“注入”打印代码，也就是术语“插桩”。通过注入的代码显式的记录此刻变量的值或事件发生的位置等信息，这些记录可用于分析程序执行的流程和出错的现场。这种调试方式是几乎所有程序员从接触编程的那一刻起就具有的本领，也一直是一种重要和常见的调试方法。我们习惯性的在程序中加入“printf”系列的函数，打印我们所关注的内容。如：

```
if(__DEBUG__)
    printf(….);
```

这种打印调试方法，非常简单而且有效，一般用于程序开发阶段，程序员自我测试和观察使用。不过它并不算是一种完善的调试方法。主要的原因有：一、提供的信息不完整，很难明确的指出深层次的错误、二、printf 相关的库函数调用耗费较大的资源，且一般与终端关联，影响程序运行的效率、三、在发布程序时容易忽视关闭调试打印的开关。

3、日志：由于打印调试方法的限制，出现以日志为主流的调试或运行记录的机制。日志就类比于黑匣子，记录了程序中的我们所关注的一切。通过日志，我们可以分析和诊断程序。不同的系统或编程语言出现了很多的日志框架。这些日志框架提供的功能非常丰富，如，灵活的日志开启和关闭功能；多种日志级别满足多样的日志需求；精确的信息记录，包括源代码行、模块、函数等；规范化记录的日志便于日志相关的工具分析处理！Linux 中提供了 syslog 的日志机制。syslog 底层通信使用的是 Unix 域的 socket，对程序的性能几乎没有影响，在效率和管理上都很有优势。

4、使用专业的调试工具，如 Linux 中的 GDB。

以上所述的调试方法中“打印和日志调试”的方法基本上能够满足所谓的 80%的调试需求，并且让人欣慰的是，在 Net-SNMP 中这两种调试方式实现得实现非常好！下面我们就来学习 Net-SNMP 中提供的调试方法。

15.3Net-SNMP 原生调试方法

从第 6.1.2 节我们了解到，Net-SNMP 具有较为完善的调试功能。其自带的调试功能主要包括两类。一类是程序运行期间的自定义打印功能和 dump 数据报功能，另一类是在后台运行的日志功能。这两种调试方法中，前者主要用于开发阶段，后者主要用于系统拷机或正常运行期间记录程序运行的关键信息。Net-SNMP 作为开源代码，在调试方面实际上实现得非常好，尤其是自定义打印功能的机制和实现方法非常值得我们学习。当然，有了源码，我们完全可以将这些模块剥离出来、

学习并应用到其他项目中去！

15.3.1 token 调试机制

Net-SNMP 中定义了多个与调试打印相关的宏，支持不同内容或格式打印方法。这种打印机制非常灵活，我们可以结合代码和命令行，在程序运行时决定打印哪些内容，不打印哪些内容，做到有的放矢！

首先，让我们来回顾 7.2.1 节中使用过的命令行参数：-Dread_config，使用它的效果是打印出 snmpd 在启动过程中读取了哪些配置文件，从而使我们了解代理启动时的行为。该命令行参数由选项“-D”和紧接着的字符串“read_config”组成。这里的字符串在 Net-SNMP 中称之为“token”，可理解为“记号”之意，它们在代码中作为一个常量字符串记录在 DEBUGMSG 系列宏中，并作为调试开关。下面我们看看其具体的使用方法和编程方法。

- 命令行开启 token 的方法：在 Net-SNMP 的应用程序命行中开启指定 token 的格式如下所示，多个 token 由逗号隔开（或使用多个-D）。如，“-Dread_config, init_mib”。当指定某个 token 后，程序运行期间将打印代码中所指定的内容。“ALL”表示开启所有 token 标记的内容。

```
-Dtoken[,token[,token]...]
```

Net-SNMP 源码中已经编写了很多的 token，如与注册和初始化 MIB 相关的“register_mib”和“mib_init”，与系统句柄相关的“handler::register”、“handler::inject”和“handler::calling”等回调相关的“callback”，与 Trap 相关的“trap”、调试相关的“helper:debug”、与执行 SHELL 命令相关的“run:shell”等等。我们可以通过这些 token 来了解 Net-SNMP 的运行机制！

- 配置 token 方式：和在命令行中指定 token 等效的方法是在配置文件中指定 token。我们可以通过前文所述的配置方法在配置文件中进行相关的配置。下面的示例中指明了[snmp]字段，表明应用到所有的 SNMP 应用程序。

```
# snmpd.conf
[snmp]
doDebugging 1 # 开启所有 Net-SNMP 应用程序的调试开关
debugTokens token1,token2,token3… # 开启指定 token
```

- 编程方法：Net-SNMP 的应用程序之所以可以通过 token 打印与之相关的内容，是因为代码中

记录了这些 tokens，并将其注册到了系统中。其编程格式如下：

```
// 打印信息
DEBUGMSG(( token, format, ...))
//带行号的打印(TRACE LINE)
DEBUGMSGTL(( token, format, ...))
// 打印 oid
DEBUGMSGOID((token, oid, oid_len))
```

如，源码中有下列的一行代码：

```
DEBUGMSGTL(("read_config", "read_conf: %s\n", ctmp->fileHeader));
```

当程序执行到该句时，将检查程序中是否已经注册了关键字“read_config”，如果已经注册，则打印后续的字符串，如果没有注册则将不打印任何内容。从上可以看出，宏 DEBUGMSGTL 打印内容的方式基本与 printf 一致（最终也是调用 printf），只是它多了一层括号和灵活的控制方式。

依葫芦画瓢，我们可以编写自己的 token 和打印的内容。如，第 9 章中实现代理中参数部分的我们使用了“parameter”。这样当我们在调试参数部分时就可以打开该部分的调试开关，而不打开其他无关的调试 token！代码示例如下：

```
DEBUGMSGTL(("parameter", "index = %d(%d)\n",index,vp->magic));
```

实践分享：要想详细的显示调试信息，建议使用如下的组合。如果系统支持，DEBUGMSGTL 宏将打印文件名和行号，即常见的 TRACE 功能。

```
DEBUGMSGTL(("token", "here"));
    DEBUGMSG(("token", " %d", var));
    DEBUGMSGOID(("token", oid, oid_len));
 DEBUGMSG(("token", "\n"));
```

除了上述的几个宏外，还有诸如 DEBUGPRINTINDENT、DEBUGMSGHEX 等。相关的宏定义读者可以参考源代码。

注意

1、Net-SNMP 编辑配置时不能使用：--disable-debugging；

2、DEBUGMSG 系列都使用双括号！

15.3.2 日志

除了上述打印调试信息到终端外，Net-SNMP 同样地支持以日志的方式记录警告或错误等程序运行信息，并可根据实际情况选择日志级别，记录合适的日志信息，防止过大的日志文件。实际上，日志本身就是监控软件运行的一种机制。一般来说日志记录格式中，每行开始都记录了日志记录的时间戳，Net-SNMP 中的记录格式也是如此。

和打印信息一样，Net-SNMP 中也实现了自己的日志模块，我们同样可以提取其中的源码，学习并应用到其他的项目中去！其日志模块实现原理是将不同种类(stderr，file，syslog)的句柄注册到按级别（7-0）排序的链表中。下面我们看看其具体的使用方法和编程方法。

日志的开启方法是在命令行中指定相关的参数：

- -Le：日志信息输出到标准错误；
- -Lf FILE：日志信息输出到文件；取消日志输出，则可以指定文件为/dev/null。
- -Lo：日志信息输出到标准输出；它是我们最常用的选项。
- -Ls FACILITY：按指定日志级别将日志信息输出到 syslog。日志级别从 LOG_EMERG 到 LOG_DEBUG，对应数字 0-7。关于更详细的日志级别说明，请读者参考 syslog。

日志输出 3 种方向的选项，如以大写字母表示则后接优先级：E、O、F、S、N(dnot log)，实现注册某类型（type）和某优先级(priority)的日志。优先级可以由字母或数字表示。

错误（error）级别：

- LOG_EMERG：0/!
- LOG_ALERT：1/a/A
- LOG_CRIT：2/c/C
- LOG_ERR：3/e/E

警告（warning）级别：

- LOG_WARNING：4/w/W

消息（information）级别：

- LOG_NOTICE：5/n/N
- LOG_INFO：6/i/I
- LOG_DEBUG：7/d/D

下面的以一个较为复杂的例子说明如何在 Net-SNMP 中使用日志功能：

```
-LF6 filerecore.txt -LO7 -LS3u
```

上述的命令行参数的含义是：文件中记录的是 6 级及以下日志、标准输出打印所有日志、syslog 记录的是用户信息 3 级及以下日志。关于更多选项-L 的含义读者可以在命令行中查看！

在代码中则按如下的格式和示例编程：

```
// 格式

snmp_log(level, format, value);

// 真实例子

snmp_log(LOG_INFO, "NET-SNMP version %s\n", netsnmp_get_version());
```

这样，按照 Net-SNMP 中原生的打印和日志方式，能使我们的调试更具有组织性和计划性！

15.3.3 打印原始数据报

Net-SNMP 中支持 dump 收/发的 PDU 数据报，并以 16 进制显示，这有助于分析底层信息，如 PDU 及 SNMP 相关编码方式。它有两种开启方法：

- 命令行参数：-d。
- dumpPacket yes：在配置文件中配置。

注意 由于 dumpPacket 等配置选项并非代理独有，而是针对整个 Net-SNMP 的应用程序（库中实现），所以当将这些选项配置在 snmpd.conf 中需要添加[snmp]标记！或者将其添加到全局的配置文件 snmp.conf 中。

图 15-8 示例了本机获取 sysUpTime.0 对象时抓取到的数据报。

```
Sending 43 bytes to UDP: [127.0.0.1]:161->[0.0.0.0]:0
0000: 30 29 02 01  00 04 06 70  75 62 6C 69  63 A0 1C 02    0).....public...
0016: 04 68 66 09  47 02 01 00  02 01 00 30  0E 30 0C 06    .hf.G......0.0..
0032: 08 2B 06 01  02 01 01 03  00 05 00                    .+.........


Received 46 byte packet from UDP: [127.0.0.1]:161->[0.0.0.0]:38746
0000: 30 2C 02 01  00 04 06 70  75 62 6C 69  63 A2 1F 02    0,.....public...
0016: 04 68 66 09  47 02 01 00  02 01 00 30  11 30 0F 06    .hf.G......0.0..
0032: 08 2B 06 01  02 01 01 03  00 43 03 00  AF 80          .+.......C....

DISMAN-EVENT-MIB::sysUpTimeInstance = Timeticks: (44928) 0:07:29.28
```

图15-8　dump 数据报示例

15.4 GDB 调试方法与技术

GDB 是一款开源、免费的、跨平台、源代码级别（调试具有源代码的应用程序）、功能强大的符号调试器（symbolic debugger），也称为 GNU 调试器。它支持多种硬件平台、多种编程语言，既可以本地调试也可以远程调试，同时支持多线程调试。

在 Linux 系统中，它是一个基于命令行形式的控制台程序（具有前端图形界面的 GDB 则称 DDD）。如图 15-9 所示（-quiet 安静模式启动选项，则不显示启动信息，只显示提示符）。

```
[~]# gdb
GNU gdb (GDB) Red Hat Enterprise Linux (7.2-60.el6_4.1)
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "i686-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
(gdb)
```

图15-9　gdb 启动界面

gdb 可以完全控制载入其中的应用程序或附着已经运行的程序，包括程序的起停、更改和监视变量，修改程序的执行流程、查看程序运行的详细信息和变量信息等。有了这些功能我们可以探索程序的运行细节、运行结果是否和预期的效果一致等等，从而掌握，理解程序，解决 bug！

gdb 既可以调试普通的应用程序、GUI 程序，也可以客户端/服务器程序。它提供了很多命令，功能非常强大，如何巧妙地利用它们提高调试效率会是一件令人兴奋的事！本章介绍的 gdb 调试方法和技巧都是 gdb 通用的调试方法，而不仅仅限定与 SNMP 代理的调试。更多的有关 gdb 的内容请参考其手册。

调试 snmpd 时建议加入运行参数“－f”，防止其进入后台运行（当然 gdb 也支持后台进程的调试：set follow-fork-mode [parent|child]）。下面我们就开始调试之旅吧！以下的内容主要是笔者在调试和学习过程中积攒收集的。更多的内容请读者查考 gdb 手册。

15.4.1 调试前的准备

在调试某个程序前，我们可能会理所当然的认为该程序可调试。不过事实并非如此！我们发现，在 gdb 调试过程中往往出现局部变量找不到、无法设置断点、某些代码不被执行、执行的顺序与代码行不一致等诡异情况。出现这种情况往往是 GCC 编译过程中对程序进行了优化，导致优化后的机器指令与行号不完全对应、变量被优化掉了、调试符号不存在。事实上，程序完全按照机器指令

由低到高的顺序执行，而不是按照代码行的顺序！

要使程序具有可调试性，使用 gcc 编译源程序时，建议遵循以下的几点：

1、程序必须包含调试信息，这就要求在使用 GCC 编译时使用“-g”的选项，生成系统原生的调试符号表。“符号表”指的是变量名，函数名等字符变量与内存地址的关系映射表。除此之外，在编译期间还可以指定调试信息记录的级别：-glevel（level 取值 1，2，3；数值越大生成的调试信息越丰富，如可以查看宏定义；默认为 2）和 gdb 专用的调试信息-ggdb。

2、为了实现对程序更好的调试，在编译时应该避免编译器对其进行优化。在 GCC 中的可以显式地指定其优化选项为“-O0”。

3、待调试的程序不能使用 strip 去除调试信息。否则出现“no debugging symbols found”的提示。

在代理 snmpd 的编译过程中，Net-SNMP 默认的优化选项是“-O2”，为了防止 GCC 对 snmpd 过多的优化而影响调试效果，我们需要在 configure 时显式的指定配置选项“--with-cflag=-g –O0”。当然有时觉得重新 configure 过于麻烦，耗时，可以手动更改已有的 Makefile 文件，在需要更改的目录层级下（如，调试内嵌的私有代理），找到关键字“CFLAGS”和 “-O2”将其临时更改。

另外，在编译期间还可以注意以下几点：

1、当程序没有察觉到或难以察觉的 bug 时，可以先将程序运行起来，长时间拷机，当出现严重错误时程序自动生成核心转储（core）文件，便于后续的分析和调试。生成 core 文件则需要操作系统相关的环境设置。在 Linux 系统中，可以使用“ulimit –c unlimited”解除系统对 core 文件生成大小的限制。

2、可以添加有利于调试的编译选项，如，-DDEBUG，则源码中带有“#ifdef DEBUG”的代码块就将生效。当然“DEBUG”可以是用户任意定义的标识符，不过一般遵循这样的使用方式。

3、开启日志功能记录程序运行的关键点。

当然，良好的代码设计是代码具有可调试性先天条件。关于代码编写和设计层面的可调试性，本文就不再叙述了！

15.4.2 调试过程与指令

使用 gdb 调试程序的主要流程很简单。一般的操作方法是将程序装载到 gdb 中、设置断点、运行程序（暂停程序，设置新断点）、单步、查看变量等方法。虽说过程很简单，不过这里面涉及到很多的技巧和方法会让我们更高效率的调试，使得我们有更多的精力专注于问题本身！下面我们就

按照这里所说的简化的调试流程，讲述一般的调试过程。由于一般的调试过程就是按照这里所讲述的顺序，读者实践起来会更方便！

1. 启动 gdb

使用 gdb 调试程序有多种方式，假设某二进制文件名为 snmpd，那么有以下几种方式启动 gdb 调试该程序，在当前程序的目录下执行：

- gdb snmpd：调试 snmpd；
- gdb snmpd pid：调试已经运行的程序，其中 pid 指的是程序的进程号；
- (gdb) file snmpd：在 gdb 启动后加载程序的方式；
- (gdb) attach pid 或(at pid)：链接到已经运行的程序。detach 命令则断开链接。
- gdb snmpd core.1234，调试 core 文件（假设 core 文件名为 core.1234）。

其中，使用“file”命令加载程序的优势是在于：调试程序的过程中对源文件有新的更改并重新编译。这时，我们可以将新的可执行文件通过 file 命令加载，从而在未退出 gdb 的情况下，保留原有的调试环境，如之前设置的断点等，这样便于我们继续调试并验证新的更改。

实际操作过程中 gdb 命令使用技巧：

1、支持 TAB 建命令补全；

2、直接回车，表示执行上一条命令

3、支持命令缩写

4、不区分大小写

5、多使用 help 命令

接下来，我们就开始调试工作了。在这之前，或许我们希望记录调试过程中使用到的命令操作，以便后续必要时查看和分析！如果有这样的需求可以使用命令“set logging on”开启 gdb 命令输出信息到文件中（输出到屏幕的同时再记录到文件中，默认文件名为当前路径下 gdb.txt），或者将所有的打印内容重定向到文件。

2. 断点与监视点操作

调试程序实际上是控制程序的过程。断点操作是调试过程中非常重要的一环，我们需要在适当的地方，适当的时间将程序停下来。即在运行前我们往往会在关心的函数或位置设置断点。在 gdb

中每个断点都有唯一的一个序号，用于标识该断点。我们可以使用“info break”查看断点的序号、断点使能情况、已经遇到断点次数等信息。设置断点的格式如下：

```
break [LOCATION] [thread THREADNUM] [if CONDITION]
```

很明显，break 的参数都是可选参数：

LOCATION：支持源文件行号、函数名、内存地址；

THREADNUM：在指定线程上设置断点；

CONDITION：表达式为真时设置断点，并暂停程序；（允许多次设置同一个位置的断点）

下面列举常见设置断点的例子：

```
break main

break app.c:10

break thread 2 *0x123456

break myfun if (myvar==11)
```

最后一行的代码中，使用了变量 myvar。一般来说 myvar 必须为全局变量，否则在 gdb 无法定位该变量。另外，还有其他的断点设置方法，如：

tbreak：临时断点，一次有效；

hbreak 设置硬件断点等。

注意 在实际调试过程中，在程序未运行前，可能会出现不能设置断点，提示找不到相关的文件。这往往是所需要的动态库还未加载。可以先在 main 函数处设置断点，进入程序后再设置所需要的断点。当提示如下时选择“y”：

Make breakpoint pending on future shared library load?

当然也可以先在 main 函数处设置断点，待程序运行起来后暂停在 main 函数时再设置其他的断点。另外需要注意的是：调试的动态库，同样需要有调试信息，否则无法正常进行。这就是为什么一部分的开源库有调试版本的，也有正式发布版本的。一般来说正式发布版本的动态库没有调试信息，无法顺利地进行调试！

已调试完的，过多的断点，我们可以禁用或删除之。禁用的方法是 disable，不带参数的 disable 表示禁用所有断点。它同样地支持指定断点序号，表示禁用指定的断点，多个序号使用空格隔开。启用的方式是 enable。删除断点则使用 delete 和 clear，使用方法与上相同。以上的命令都可以缩写，如 break 可缩写为 b。类似的内容将不再提示！

使用断点的目的往往是将程序停在某处，分析其运行情况和变量变化情况。不过 gdb 还有另外一个可监视变量变化的功能，那就是“监视点”。一旦指定的变量或表达式发生变化（被写时），gdb 将暂停程序，保留现场，这有种“守株待兔”的味道。比如说某个变量被莫名其妙的更改，但又不知道何种原因，此时监视点可能是非常好的选择，也让人为之兴奋。

一般来说，监视的变量只能是全局变量（局部变量在程序退出当前栈后将消失），函数中的静态变量也是无法监视的，不过笔者的经验还可以通过监视其地址达到监视的目的。使用监视点的方法如下：

```
watch [-l|-location] expr [thread threadnum] [mask maskvalue]
```

另外还有 rwatch 和 awatch，分别用于监视读操作和读或写操作。

如，监视 foo 变量：

```
watch foo
```

监视指定地址的变量：

```
watch *(int *) 0x600850
```

监视表达式，由 false 到 true 时暂停程序：

```
watch (x>11)

watch div==value
```

查看已经设置的监视点：

```
info watchpoints
```

监视点分为两类：硬监视点（hardware watchpoints）和软监视点（software watchpoints）。gdb 会尝试使用效率高的硬监视点。不过当系统架构不支持硬监视点时，只能使用软监视点。软监视点会使整个程序运行异常缓慢，往往会让你失去继续使用它的决心。这就是监视点“可能是非常好的选择”的说法原因。

最后要说明的是监视点不仅仅是监测变量，也可以监测系统级别的行为，如：

catch load——程序载入动态库时暂停；

catch fork ——程序 fork 时暂停；

catch signal：信号发生时暂停等等。

3. 运行与控制程序

运行程序，run 命令，缩写为 r。如调试 snmpd 一般会带入参数，传入参数的方式很简单和直接——像不使用 gdb 一样，直接写在后面即可。

```
(gdb) r -Lo -f
```

- run：命令重新开始运行程序，而命令 continue（c）则是继续运行被暂停的程序。
- CTRL+C：暂停程序的运行。如果程序是多线程的，该组合键将停止所有线程的运行。我们常常在程序停止时，设置新的断点、查看变量与函数的返回值等。

调试中最常用的命令估计是单步操作命令 step（s）和 next（n）：

- step/step N：执行下一行或 N 行代码，如果遇到函数则进入到函数体中；
- stepi/ stepi N：单条或 N 条指令执行；（与此相关的 x /i $pc 则查看下一条汇编指令，支持一次显示多条指令：x /Ni $pc）
- next/next N：执行下一行或 N 行代码；不进入遇到的函数体中；
- nexti/ nexti N：单条或 N 条指令执行；

在这么断断续续，来来回回调试程序和琢磨后，偶尔也会让程序员摸不着头脑“我执行到哪了？”。这时使用 where 命令查看当前的行号和所处的函数，或许顿时让你喜笑颜开吧（笔者曾经还真调试得过摸不着北啊，一问周围人都不知此用法）!

如果进入到某个子函数里，并且已经调试完所关注的内容，可以使用 finish 直接运行完该函数以节省调试时间。如果够任性，你可以直接在当前敲入 jump < address>使程序从指定的地方运行（jump 不改变当前栈和内存，如果 jump 到当前函数之外可能会导致奇怪的现象），或敲入 return（支持 return 表达式，返回指定的值）指令忽略该函数后续所有的指令，离开这是非之地！

不过有时，我们只想退出当前循环 100 次的代码块，这时 until（u）就派上了用场。不带参数的 until 表示越过当前栈中的下一条语句，如果在循环体的最后一条语句执行它（一般 for 循环的写法中最后一条语句即是 for 的那一行），效果就是退出该循环体，这正是 until 和 next 的差异之处（next 将回到循环体中）。until 也支持执行到指定的目的地，目的地的表示方法类似 break，支持行号等方式，不过越过当前栈时将停止程序。

continue 命令可以使暂停后的程序继续运行，如果带参数 N 表示忽略后续 N 个断点。

除了上面的暂停程序，也可以 kill 程序。kill 命令结束当前正在调试的程序。一般使用的场景是：已经发现程序错误，需要重新更改和编译源代码时使用 kill 立即结束当前调试，并在程序编译

后使用 file 命令加载新编译的可执行文件。

此外，我们还可以在 gdb 中直接运行程序中的函数以及当前已加载库中的函数。当想查看某个函数的返回值时，这非常有用。如下的例子说明了在 gdb 中调用函数的方法：

```
call myfun(1)
print strlen("12345")
call (int)close(fd)
```

4. 查看信息

查看程序中的信息是我们探究程序内部最直接的方式。这里所述的查看信息包括，查看变量、寄存器的信息和源代码相关的信息。根据变量在代码中的定义不同，查看的方式也有多种，如查看栈中的变量，查看堆中的变量。同样的，查看内容的表现形式也有多种。下面我们简洁的归纳为以下几类命令：

1、查看程序概览信息

- info proc：查看程序启动路径等信息。
- info functions：查看已加载的函数声明信息。

2、print（p）变量

支持当前栈的局部、全局变量、地址、表达式等方式查看变量值。可以通过“file::variable”和“function::variable”格式打印指定域的变量。如，

```
print  mylocal
print  *0x123456
print  *structptr; 打印完整的结构体变量内容
```

为了使得打印结构体更具有可读性，可以运行如下的指令。它们分别对应结构体、联合体和数组输出格式化。

```
set print pretty on
set print union
set print array on
```

除了查看变量的值，还可以使用 ptype（详细）和 whatis（简单），查看变量的数据类型。如查看结构体变量的详细定义可以使用 ptype。对于特别关心的变量，我们可以使用 display 命令。“display myvar”表示程序每次暂停时都自动显示 display 指定的变量名，而不用手动输入。这主要应用到极为关切的变量。该命令的使用方法与 break 类似，如禁止显示：disable display N，删除自动显示：undisplay N。

print 除了打印功能之外，实际上也可以改变对象的值。下面的两个命令效果一致：

```
p var=1

set var=1
```

3、查看内存

examine(x)命令以指定的解析方式查看内存地址中的数据。其格式如下：

```
x/<n/f/u> <addr>
```

- n：是一个正整数，表示需要查看的内存单元的个数。一个单元的字节数由下面的 u 指定。
- f：表示显示的格式，如 o(octal), x(hex), d(decimal), u(unsigned decimal),c(char) and s(string)；
- u：表示一个内存单元占用的字节数（默认是 4 个字节）。其可取的值有：
- b：表示单字节，h 表示双字节，w 表示四字节，g 表示八字节。
- 以上命令都可以使用 help command 查看详细的说明。下面是两则例子：
- x /2xb address：以单字节 16 机制的方式查看地址 address 的内容，共 2 个字节；
- x /3xw address：以 4 字节 16 机制的方式查看地址 address 的内容，共 12 个字节；

4、查看数组

数组分为普通数组（正常定义的数组）和堆中的数组结构（malloc 动态分配的动态数组、数组指针）。对于普通的数组，直接使用 print 即可。对于堆中的数组结构，则使用

```
p *arraName@N
```

表示输出动态数组 arraName 中前 N 个值。显示格式：p/x,c,s,f，分别表示以十六进制、字符、字符串、浮点。

5、查看寄存器

info register。显示当前寄存器中的值。

6、查看调用栈

可以查看栈中的信息有：传入到当前函数（栈）的参数、函数的局部变量、返回地址信息。backtrace（bt）命令可以查看当前调用栈（栈序号为 0），各个栈按调用顺序依次递增排列。我们可以从栈中看出哪些函数在源文件中的多少行调用了哪些下级函数，即函数的调用关系路线图。查看当前栈的函数名、参数值、函数所在文件及行号等信息使用：

```
info frame（info f）
```

查看当前栈的局部变量使用：

```
info locals
```

查看函数的参数信息：参数名，值则使用：

```
info args
```

查看非当前栈内容的方法是：使用 bt 找到对应的栈序号，然后使用“frame No”进入到指定的函数栈中，最后使用查看当前栈的的方法查看其中的信息（被优化的局部变量无法查看）。另外，还有一个查看非当前栈的小技巧，那就是使用作用域运算符“::”，如，“p function::x”表示查看函数名为 function 中的变量 x。当然，全局变量可直接查看，而不用关心当前程序执行到哪。

7、查看动态库

与调试普通程序一样，调试动态库也要动态库具有调试信息。这要求编译动态库时加入“-g”选项、不使用“strip”、不使用“-fomit-frame-pointer”（否则无法调试栈帧）。在 gdb 中查看动态库加载情况：

```
info sharedlibrary（info share）
```

手动加载动态库：

```
sharedlibrary regex
```

不带表达式 regex 时表示载入程序所依赖的所有动态库。否则加载由表达式指定的动态库。

8、查看源代码

gdb 支持在调试过程中查看当前执行的代码或其他代码块。list：查看程序的源代码（要求将源代码放在当前目录或指定的目录中）。此命令便于查看当前程序执行的源代码位置，也支持显示具体的函数或行，如：

```
list myfun

list app.c:10
```

9、GUI 模式

TUI（TextUser Interface）为 GDB 调试的文本用户界面，一般人很少知道该功能。该字符界面可以分栏显示源代码窗口、汇编窗口和寄存器窗口等。这些窗口会随着代码的执行同步更新，非常直观！调试小程序可以考虑使用 GUI 模式。进入该模式的方法有如下几种：

- gdbtui
- gdb-tui
- CTRL+X+A 组合键

各个窗口间的查看可以使用如下的命令：

```
layout <src, asm, regs, split> #显示指定的窗口
focus: <src, asm, regs, split> #固定显示某个窗口。
```

5. 多进程和多线程

gdb 中支持线程和进程的调试，不过多进程和多线程运行时具有不确定性，所以调试起来相对困难！多进程程序调试中首要的任务是清楚调试的进程对象。由于 Linux 中多进程涉及到 fork，原进程会分支出多个进程。使用下面的命令跟踪对应的进程：

```
set follow-fork-mode [parent|child]
```

parent：fork 之后继续调试父进程，子进程不受影响。

child：fork 之后调试子进程，父进程不受影响。

尽管 snmpd 是单线程的程序，不过正常运行时它会 fork 成后台进程。如果这样运行的话，调试时需要使用“set follow-fork-mode child”。当然在调试 snmpd 时，加入"-f"的选项是最好不过了！

使用如下的命令查看 fork 的进程列表，每个进程由 gdb 中的进程编号所标识：

```
info forks
```

使用如下的命令将序号 No 的进程作为当前进程：

```
fork No
```

另外，detach-fork，delete fork No 分别表示使进程 No 脱离 gdb 和 kill 进程。

gdb 对线程的支持包括新线程产生的通知、线程间的切换、查看信息、设置指定线程的断点、在指定的线程运行命令等。同一个进程中，所有的线程共用堆空间、进程 ID 和信号处理函数；每

个线程拥有自己的栈，信号掩码和优先级。我们可以使用下面的命令查看线程数量和线程 ID：

```
info threads
```

上述命令会显示当前进程的各个线程编号，我们可以使用：

```
threads No
```

切换到序号为 No 的线程作为当前线程，就可以按照之前所介绍的方法进行调试。

多线程调试时最容易出现线程锁的问题，在 POSIX 多线程调试时建议加入如下的内容避免线程常规的错误：

- 编译时加入“-D_GNU_SOURCE”；
- 代码中设置线程属性，pthread_mutexattr_settype(&attr,PTHREAD_MUTEX_ERRORCHECK)；
- 全局变量的定义考虑加入 volatile 关键字。

15.4.3 gdb 脚本调试技术

gdb 的脚本功能可能知道的人不多。不过一旦你了解后，或许就进入了调试高手的行列了！

1、配置.gdbinit

gdb 启动时会按照一定的顺序（当前目录./.gdbinit、用户目录~/.gdbinit）查找隐藏的.gdbinit 文件。如果存在该文件，gdb 就会自动执行里面的命令。这种功能一般用于配置软件的运行环境，就如同配置文件，实现个性化和自定义。在 gdb 中，我们一般把常用的命令放在这个文件里，这样就不用每次进入 gdb 后再去手动执行这些命令。如，笔者会把自定义的命令、显示输出数据的缺省基数（如设置为十六进制）、以及类似于“set print pretty on”的命令放于此文件中。

当然，我们也可以将待调试程序中的断点等等写入到脚本中，避免每次启动后再一一手动输入，从而避免影响调试的心情和效率。

2、编写 gdb 脚本——宏

上一节中查看程序的信息使用的方式是直接在 gdb 界面中输入相关的命令。不过，当需要查看某种数据结构时上述的命令输出一般难以达不到预期。比如，我们希望查看链表中的内容，每个链表节点是某个结构体，同时希望选择性的打印结构体中的信息并格式化输出。那么，直接使用 print *plist 是实现不了的。实际上，gdb 中支持宏编写的脚本，即自定义命令。这些宏语法与 shell 语法类似。定义用户命令的格式如下：

```
define newCommand
```

```
// code
end
```

下面是一个简单的脚本。该脚本实现 3 个数字的相加。脚本名：gdbsh。

```
define addr3
     if $argc==3
      print $arg0+$arg1+$arg2
     end
end
```

然后在 gdb 中执行：

```
(gdb) source gdbsh
(gdb) addr3 1 2 3
$1=6
```

一旦脚本加载到 gdb 中，则可以使用下面的命令查看当前环境下的用户自定义命令：

```
show user。
```

另外也可以，给自定义命令加入帮助信息，如：

```
document gdbsh
     gdbsh: add three variable.
     Example: gdbsh 1 2 3
end
```

当然我们也可以使用已有的命令组合新的命令，至于什么样的形式可以任由我们天马行空。下面的命令 myf，显示当前栈的详细信息。

```
define myf
     info frame
     info args
     info locals
end
document myf
```

```
          Print stack frame
      end
```

下面的例子则是打印链表的示例。相关链表结构体的定义请参考“第 9 章代理开发实战”的源码。其中用户自定义的命令 dplisttable1 支持打印链表所有行、指定一行、指定行范围的打印。这些功能由参数控制！

```
      #dump list of MIBIDSTRUCT,T_TableSimple,T_TableIndex1
      #dump node of MIBIDSTRUCT
      define dpmibnode
        set $node = (MIBIDSTRUCT )$arg0
           printf   "%d,
0x%04X,    %d\n",$node.utype,$node.t_tacheID.ipcNo,$node.t_tacheID.snmpmagic
      end


      #dump list of T_TableSimple
      define dplistrow
        set $simtable = (T_TableSimple *)$arg0
           printf   "type       No    snmpmagic\n"
        while $simtable
           set $node=$simtable->node
            dpmibnode $node
           set $simtable=$simtable->next
        #end of while
        end
      #end of define
      end


      #dump list of T_TableIndex1
      define dplisttable1
      set $table1=(T_TableIndex1 *)$arg0
```

```
    while $table1
       set $allrownum=$table1->index

       if $argc == 1
           printf "\n--row:%04d--\n", $table1->index
            dplistrow $table1->list_node
       end

       if $argc == 2
          if $table1->index == $arg1
           printf "\n--row:%04d--\n", $table1->index
            dplistrow $table1->list_node
          else
            printf "."
          end
       end

      if $argc == 3
          if $table1->index >= $arg1  && $table1->index <= $arg2
            printf "\n--row:%04d--\n", $table1->index
            dplistrow $table1->list_node
          else
            printf "."
          end
       end

      set $table1=$table1->next
    end
    printf "\n<<-- all row number: %d -->>\n", $allrownum
```

```
end
```

由于脚本编写灵活，可以实现不同的调试需求，而不仅仅是查看变量的信息。有时候我们甚至可以将内存中的信息通过脚本生成指定格式的文件等，至于这些文件有何用，那就是自己的事了！

当然，就打印信息而言，脚本只是另一种查看的方式。就上述的查看链表的脚本来说，我们完全可以在源码中编写打印链表的函数（该函数在主函数中并不被调用），然后在需要的时候使用 call 命令调用即可。因为在源码中实现这样的函数可能会更为简单！

最后，再分享一个相关的技巧：attach 一个已经运行的后台程序时，由于后台程序关闭了标准输入输出，调试信息往往不会打印到当前屏幕。此时可以使用 call 命令，重新打开或重定向到文件中。其中使用到的多个指令组合就合适放在脚本中。

3、执行 shell 命令

可能让读者不曾想过的是，GDB 中也支持 shell 命令。其使用的方法是在命令前加入“shell”关键字，格式如下：

```
shell <command string>
```

实际上 gdb 中也可以直接运行内置的“shell 命令”，如 pwd 查看 gdb 当前的工作目录，cd 切换目录等。另外，一个在调试过程中方便 gdb 调试和源码编译的命令就是 make。

```
make <make-args>
```

其功能与 shell make 是一致的。该命令主要应用于：在不脱离 gdb 的环境下，更改，编译和调试程序。使用流程可以是：kill->(shell vi/vim)->make->（file yourapp）->run。

调试结束，使用 quit（q）退出 gdb！

以上的内容，笔者认为是 GDB 调试中的方法与技巧的部分精华。掌握了它们，我们才能更顺利的开展调试工作。我想，只有当我们对事物有了了解才能心生坦荡！

15.5辅助调试工具介绍

除了 gdb 外，下面再分享一些辅助的诊断和调试工具。它们是 LinuxC/C++开发中较常使用的小工具，这些工具都能应用到 Net-SNMP 代理的调试。

15.5.1 tcpdump

tcpdump 是抓取网络数据包和分析的经典工具。它驻扎在网络接口的咽喉要道，捕获所有满足布尔表达式过滤器指定的网络数据包。这些表达式支持对网络层、协议、主机、网络或端口的过滤。由这些数据包，我们可以了解最原始和直接的网络间通信内容，以分析相关的通信问题或细节。它支持大部分的网络协议，如，ip/ip6，tcp，udp，ppp，icmp，wlan 等等，是我们调试协议相关问题必须了解的分析利器。其官网是：http://www.tcpdump.org/。当然分析每一种数据包都需要我们了解相应协议的背景知识。由于 SNMP 使用 UDP 或 TCP 协议，我们可以使用 tcpdump 查看 snmpd 通讯的内容。

tcpdump 支持很多的可选参数选项，覆盖过滤表达式、控制输出格式、文件功能等选项。其基本的输出格式是：“系统时间 协议 来源主机.端口 > 目标主机.端口 数据包”。下面我们简要的介绍该工具主要的命令行参数，更详细的内容请参考官网资料。

- -i：指定网络接口，如 eth0,lo 等（使用 ifconfig 或 tcpdump -D 查看网络接口）
- -c：抓取指定数量的包后停止；如，关注 TCP 的建链过程。
- -C file_size：将数据包写入到文件中，一旦文件大小超过 file_size（单位兆字节，1,000,000 bytes）则新建文件。
- -F file：以 file 中的内容作为过滤表达式。
- -X：十六进制显示包内容；
- -l：输出行缓冲（主要控制输出的格式化——遇到换行即输出，便于查看）
- -n：数字形式显示网络地址，避免 DNS 查询。
- -s snaplen：截断指定的长度显示；
- -t：不在每一行打印时间戳。
- -tt 在每一行中输出至 1970 年的秒数（-ttt：年月日，时分秒）。
- -X：以 16 进制和 ASCII 码形式显示每个报文（去掉链路层报头，-XX 则包含链路层报文头）
- -w file：以原始网络包写入文件而不是打印出来。该文件可以使用 wireshark 来分析或使用-r 读取。

如果直接运行 tcpdump ，将监测所有网络包，信息量非常大，不利于有的放矢的专注分析。为此 tcpdump 提供了过滤器，通过自身的表达式过滤指定的信息。这些表达式由如下的几大类关键字和逻辑运算符等组成。我们在 root 用户权限下运行下面的例子。

- 类型关键字：host，port，net；

tcpdump host sundown：截获来自和发往主机 sundown 的网络包。可直接写 ip 地址

tcpdump host helios and \(hot or ace \)：截获主机 helios 和主机 hot 或 ace 间的网络包（注意使用转义符或使用引号）。

tcpdump ip host ace and not helios：截获除 helios 外，所有与主机 ace 通信的 IP 网络包。

tcpdump net 192.168.1.0/24：截获来自和发往 192.168.1.0/24 的所有主机的网络包。

指定端口（数字端口或知名端口）：

tcpdump port 80

tcpdump port not ssh

tcpdump port snmp

指定网络：

tcpdump net ucb-ether：打印本机与 Berkeley 网络主机的网络包。

指定接口：

tcpdump -i eth0

- 传输方向的关键字：src，dst。源地址和目的地址，即来自和发往两个方向。由逻辑表达式还可构成 dst or src、dst and src；

tcpdump src host sundown：截获来自主机 sundown 的网络包；

tcpdump dst host sundown：截获发往主机 sundown 的网络包；

tcpdump -i ppp0 dst net 192.168.43.132：跟踪网卡接口 ppp0，目标主机 192.168.43.132

- 协议的关键字：ip，arp，rarp，tcp，udp，ppp 等类型；

tcpdump 'tcp port 80 and (((ip[2：2] - ((ip[0]&0xf)<<2)) - ((tcp[12]&0xf0)>>2)) != 0)'：打印端口 80 所有 IPv4 HTTP 的网络包。

tcpdump 'gateway snup and ip[2：2] > 576'：打印由网关 snup 发送的，大于 576 字节的网络包

tcpdump 'icmp[icmptype] != icmp-echo and icmp[icmptype] != icmp-echoreply'：打印所有非 ping 的 ICMP 包。

tcpdump -i lo udp：跟踪本机回环接口 udp 协议；

tcpdump dst net 192.168.43.132 and port 162 and not src 192.168.43.145 and not src 192.168.43.146：跟踪目的主机为 192.168.43.132，端口号为 162 和 1234；

tcpdump udp and port 161：跟踪 udp 和端口 161；

- 逻辑运算关键字；

  与：and、&&；

  或：or、||；

  非：not、!；

  tcpdump dst host 192.168.43.132 and (port 162 or port 161)：截获发往主机 192.168.43.132 的 161 或 162 端口的网络包；

- 其他重要的关键字：portrange、gateway、broadcast、less、greater 等；

  tcpdump -n dst portrange 161-1611：截获目标端口在 161 和 1611 之间的所有报文；

  tcpdump greater 20：截获所有报文长度大于 20 的报文；

  图 15-10 是在 146 机器上运行“snmpwalk -v 2c -c public 192.168.43.132 sysORTable”命令，132 机器上 tcpdump 的部分打印信息：

```
chanson: ~ # tcpdump -t -n port snmp
tcpdump: verbose output suppressed, use -v or -vv for full protocol
decode
listening on eth0, link-type EN10MB (Ethernet), capture size 96 bytes

IP 192.168.43.146.50656 > 192.168.43.132.161: GetNextRequest(27)
.1.3.6.1.2.1.1.9
IP 192.168.43.132.161 > 192.168.43.146.50656: GetResponse(39)
.1.3.6.1.2.1.1.9.1.2.1=.1.3.6.1.6.3.11.3.1.1
IP 192.168.43.146.50656 > 192.168.43.132.161: GetNextRequest(30)
.1.3.6.1.2.1.1.9.1.2.1
IP 192.168.43.132.161 > 192.168.43.146.50656: GetResponse(39)
```

图15-10 tcpdump snmpwalk 命令

与 tcpdump 功能类似的网络抓包工具不得不说 Wireshark。Wireshark 是 Windows 下非常简单易用的图形化的抓包工具、开源免费非常流行、查看和分析网络数据包也非常直观，如果在 windows 下进行相关的网络数据报文分析，建议使用它。

与 tcpdump 类似，Wireshark 同样使用过滤器的机制抓取指定的数据报，并提供了两层过滤机制：捕捉过滤器和显示过滤器。分别过滤要捕获的报文以及要显示的报文。过滤器中表达式的使用方法也与 tcpdump 类似。更多相关的内容请读者参考官方网站：http：//www.wireshark.org

15.5.2 nm

nm 属于 Binutil 工具套件中的一个小工具。nm 列出 ELF(Executable and Linking Format，可执行连接格式)对象文件的符号表，一般用于辅助软件的调试。对象文件包括如目标文件、库文件、可执行文件等。符号表一般指的是导出的函数、全局变量等符号的对应关系。

注意 Binutil 是 linux 下的一个辅助 GCC 的一个工具集（http://www.gnu.org/software/binutils/）。Binutil 包含一系列的工具，由一套用于编译、汇编和链接内核及应用程序的组件组成。诸如 GCC 编译代码时会用到的 ld，as，ar 以及发布代码时去除调试信息的 strip 都属于其中一部分。下面简要的例举了在 LinuxC/C++嵌入式开发（或移植）中常用的工具：

addr2line——将地址转换成文件名和行号。

ar——创建、修改、提取档案文件。

c++filt——C++符号命名改编过滤。

gprof——显示性能分析信息。

nm——列出对象文件符号.

objcopy——复制和转换对象文件。

objdump——显示对象文件信息。

ranlib——生成档案文件内容索引。

readelf——显示 ELF 格式的文件信息。

size——显示对象文件或档案文件程序段信息。

strings——显示文件中的可打印字符。

strip——去除文件中的符号信息。

在软件调试阶段，我们很可能会用到 Binutil 中的工具。当然这些工具与待查看文件要求是同样处理器架构类型。如，待查看的程序是某 arm-linux 版本，那么 binutil 也需要在 arm-linux 下构建！

nm 常用的命令行参数有：

- -A：每个符号前显示文件名；
- -D：显示动态符号；
- -l：显示符号所在的源文件及行号（需要调试信息，gcc -g）；
- -u：打印出未定义的符号；
- -n：按照地址/符号值来排序。这样有助于信息的查找。
- -S：显示符号所占用的地址空间大小。
- -g：仅显示外部符号。如，查看某可执行程序是否依赖了外部函数。
- -r：反序显示符号表。

nm 默认的输出格式为：符号（虚拟）值、符号类型、符号名字。

- 符号（虚拟）值：其具体含义因符号类型而异。如代码段中的符号，其值表示在代码段中的偏

移。

- 符号类型：符号类型有多种，A，B，C，D，N...R，S，T，U 等。参见表 15-1。

表15-1　nm 输出类型说明

符号类型	含义
A	表示符号值是绝对的，在以后的链接过程中，不会改变
B/b	表示符号在非初始化数据段(.bss)中，如未初始化的全局变量
C	表示符号是没有被初始化的公共符号，在链接过程中才进行分配
D/d	表示符号位于已初始话数据段（.data）中
G/g	表示符号位于初始化数据段中。主要用于 small object(如一个全局的整形变量)提高访问效率
N	表示符号是调试符号
p	表示符号位于栈回溯段中
R/r	表示符号位于只读数据区（.rdata）
S/s	表示符号值位于非初始化数据区，用 small object
T/t	表示符号位于代码段中(.text)，如 static 函数
U	表示符号没有定义，如引用了其他库中的 API

- 符号名字：即文本名，如函数名。

经验分享：在较大型软件开发中往往有多个头文件，常常会出现“unresolved symbol”的错误。这时我们可以使用 nm 辅助调试来定位该符号的位置或者调试其他相似的由于共享库的错误导致程序无法运行起来的 bug。我们可以查看相关的依赖库（可以使用 ldd 查看）是否导出了该函数。查看动态库中导出的符号使用-D 选项。

图 15-11 以第 9 章编写的 libsnmpipc.so 库文件为例说明 nm 查看导出符号的使用方法。这里显示的导出符号基本都是该动态库中导出的 API。

```
[/mnt/hgfs/centosShare/9/src]# nm -D ./libsnmpipc.so | grep T
00001878 T _fini
000006e4 T _init
0000171a T app_get_data
0000173c T app_set_data
0000127d T del_sem
0000115e T del_shm
000009bd T get_maptable
0000180e T init_shm_sem_master
00001822 T init_shm_sem_slave
00000dbf T shm_attach
0000175e T snmp_get_data
0000178e T snmp_set_data
```

图15-11　nm 打印动态库导出的符号表

更详细的内容请参考“man nm”。

注意 在编译期间，同样可以使用 nm 查看符号的使用或定义处，如：“nm -A ./*.o | grep 符号名”。与 nm 相似的工具——strings 则列出对象文件中的可打印字符串常量，包括字符串常量、宏定义、文件名等。

15.5.3 objdump

objdump 也是 Binutil 中的一个小工具，使得我们以可阅读的方式查看目标文件中的信息。这些信息包括目标文件的段信息、调试信息、程序运行平台架构、反汇编等较为底层的信息。如，我们可以查看可执行程序的代码段（.text）、数据段（.data）等。对这些信息的了解一方面有助于我们了解目标文件的组织结构和内容，另外一方面也有助于我们在 Linux 中调试程序的疑难杂症时多一种手段。

以上描述的信息由 objdump 的命令行参数控制输出。其使用格式为：

```
objdump <option(s)> <file(s)>
```

常用的命令行参数如下：

- -f（--file-headers）：显示文件头信息：包括文件格式、架构、起始地址等。
- h：显示每段 section 头信息。这些信息包括每段的大小（Size）、程序运行地址（VMA——Virtue Memory Address 虚拟内存地址，LMA——Load Memory Address 加载内存地址）、该段在文件中的偏移地址（File off）以及每一段的对齐字节数等信息。如：

```
[~]# objdump -h libsnmpipc.so

libsnmpipc.so:     file format elf32-i386

Sections:

Idx Name          Size      VMA       LMA       File off  Algn

  0 .note.gnu.build-id 00000024  000000d4  000000d4  000000d4  2**2

                CONTENTS, ALLOC, LOAD, READONLY, DATA

……
```

- -d（--disassemble）：反汇编目标文件中可执行的 section；
- -D（--disassemble-all）则反汇编目标文件中所有的 section；
- -S（--source）：尽可能反汇编出源代码；

- s（--full-contents）：显示非空 section 的十六进制码以及对应的 ascii 码；
- -g（--debugging）：显示目标文件中的调试信息；
- -x（--all-headers）：显示所有头部的信息，包括 f、h 选项的内容以及符号表。
- -j, --section=NAME：只显示 NAME 部分的 section；如：objdump -j.text -S libsnmpipc.so 将显示每个函数的反汇编代码；
- -T（--dynamic-syms）：显示动态符号表，如导出的函数信息；
- -t（--syms）：显示所有符号表，如导出的函数和全局变量等信息；
- -r（--reloc）：显示目标文件的重定位入口；
- -R（--dynamic-reloc）：显示目标文件的动态重定位入口。一般用于查看共享库信息。
- -l（--line-numbers）：显示反汇编对应的文件行号。如使用“objdump -l -d libsnmpipc.so”可查看源码中的每行对应的汇编代码！这对研究高级语言对应的汇编语言实现极有帮助；
- -C：对有命名改编的函数名转化为更具有可读性的函数名，如调试 C++程序；

从以上的选项中可以看出 objdump 使二进制文件原形毕露，正因为此，objdump 等是编译器相关和底层相关的程序员的必备工具。相比于 nm 来说，objdump 也可以输出符号表，不过更适用于调试信息的查看。

经验分享：当出现某个静态局部变量被异常更改时，并且使用 gdb 监视点无法凑效时，比如说监视点导致程序运行过于缓慢。我们可以使用一些看似“笨拙”的方法调试该变量何时被更改。首先，我们知道静态变量被分配在可执行程序中的 data 段，这样静态变量被更改极有可能是其他函数中的变量操作而影响（溢出）到。为此，我们可以 display 该变量，人为的观察该变量的改变，或许会有意想不到的效果。

当然我们也可以使用更为科学的方法，如使用

```
objdump –x program
```

或者

```
readelf –t program
```

查看数据区域的分配空间情况以及其相邻变量有哪些，并重点观察这些周边的变量涉及到的操作，往往会事半功倍！

15.5.4 strace

Linux 系统中有一项非常小巧的工具——strace（ trace system calls and signals），能够详细的追踪系统调用、信号执行的过程和结果以及对信息的统计，对解决底层系统调用问题、了解程序工作原理和性能大有裨益，堪称一款轻量级的调试器。

它输出的每一行是一个等式。等号左边是系统调用的函数名及其参数，右边是该调用的返回值，信息详细，稍显繁杂。不过这些信息对于调试的程序员来说，一定会有所收获。实际操作时，可以通过表达式（过滤）功能观察指定系统调用的情况，也可以通过关键词的过滤找到指定的信息。strace 的使用方法是：

```
strace program

strace -p pid
```

以上的操作后，直到程序退出，strace 会输出该程序所执行的系统调用和接收到的信号。添加选项-f：跟踪由 fork 调用所产生的子进程。如，可以追踪 snmpd 不带-f 参数启动时系统调用等情况。图 15-12 显示了 strace 常规输出（内容有省略）：

```
[~/zcq/net-snmp-5.7.2/include]# strace ls
......
write(1, "net-snmp  ucd-snmp\n", 19net-snmp  ucd-snmp
)     = 19
close(1)                                = 0
munmap(0xb7420000, 4096)                = 0
close(2)                                = 0
exit_group(0)                           = ?
```

图15-12　strace 运行示例

上图显示了 ls 最终会调用 write，将当前目录下的所有的文件名写入到标准输出中，即显示到屏幕！

下面是常用的命令行参数：主要分为追踪类型、内容与格式两大类。

- 追踪类型的参数：不加类型相关的参数表示监控所有的系统调用。

-e expr：由表达式指定的追踪的系统调用类型。表达式的格式如下：

```
[qualifier=][!]value1[, value2]...
```

qualifier 可取的值有 trace（默认），abbrev，verbose，raw，signal，read，write。

value 常见的取值有 open，close，read，write，file，process，network，signal，ipc。

我们可以根据字面意思理解它们的含义。如：

```
-e trace=open (-e open)

-e trace!=open
```

分别表示追踪 open 和 open 以外的系统调用的情况。

多个系统调用以逗号隔开，如：

```
-e trace=open, close
```

- 内容及显示格式的参数：

-c：系统调用统计：统计系统调用。统计内容包括，调用的函数，调用次数，消耗时间，错误次数等信息。

-o：重定向输出到指定的文件中，默认 STDERR；

-T：系统调用计时。每个系统调用的时间花销显示在每行最右边的尖括号里。

-t：系统调用发生时间，单位秒；

-tt：系统调用发生时间，精确到微秒；

-ttt：系统调用发生时间，精确到微秒，以 unix 时间戳表示；

-s：截断输出，指定打印字符串的最大长度（不包括文件名字符串）。当信息显示过于冗长时，可以使用该参数。

下面的命令可统计 snmpd 的网络系统调用的相关信息：以微妙计时单位统计网络系统调用。

```
strace -T -tt -e trace=network -c snmpd -f -Lo
```

snmpd 停止后，输出统计信息如图 15-13：

```
[~]# strace -T -tt -e trace=network -c snmpd -f -Lo
Hello, world! Net-SNMP
Turning on AgentX master support.
NET-SNMP version 5.7.2
^C% time     seconds  usecs/call     calls    errors syscall
------ ----------- ----------- --------- --------- ----------------
100.00    0.000119           3        42           sendmsg
  0.00    0.000000           0        54           socket
  0.00    0.000000           0         6           bind
  0.00    0.000000           0         4         2 connect
  0.00    0.000000           0         1           listen
  0.00    0.000000           0         6           getsockname
  0.00    0.000000           0         4           sendto
  0.00    0.000000           0         1           setsockopt
  0.00    0.000000           0        10           getsockopt
  0.00    0.000000           0        12           recvmsg
------ ----------- ----------- --------- --------- ----------------
100.00    0.000119                   140         2 total
```

图15-13　strace snmpd 示例

经验分享：strace 用于一般用于研究软件的运行机制、掌握其运行流程和工作原理、排查程序

的疑难杂症。当调试过程中对问题毫无头绪时，试试使用 strace 或许会有新的发现！如，我们可以通过 strace 研究程序的启动过程：初始化时读取了哪些配置文件。排查程序无法启动时，往往会排查程序是否是没有找到具体的文件，还是没有权限读写文件，抑或动态库缺失，库版本号不匹配等等。定位程序运行缓慢的原因时则可以通过 strace 工具查看时间是否主要消耗在某个系统调用上，并依此为依据，做出相关的优化。

下面的操作显示了如何查看 snmpd 打开了哪些文件，从中我们可以看到 snmpd 的启动流程：读取了哪些库文件、加载了哪些 mib 文件等等。

```
strace -o snmpdopen.log -e trace=open snmpd -Lo -f
```

以上命令将追踪 snmpd 中的 open 调用，并记录到 snmpdopen.log 中。下面使用 grep 命令查看成功的打开了哪些文件：

```
grep -v "ENOENT" snmpdopen.log
```

图 15-14（图中有省略）显示了 snmpd 打开了哪些库文件、mib 文件、系统文件，最后在结束程序时打开了持久配置文件“/var/net-snmp/snmpd.conf”。从这些输出我们基本上可以了解 snmpd 的工作流程。

```
[~]# grep -v "ENOENT" snmpdopen.log
open("/usr/local/lib/libnetsnmpagent.so.30", O_RDONLY) = 3
open("/usr/local/lib/libnetsnmpmibs.so.30", O_RDONLY) = 3
open("/usr/local/lib/libnetsnmp.so.30", O_RDONLY) = 3
......
open("/usr/local/share/snmp/snmpd.conf", O_RDONLY) = 5
open("/var/net-snmp/snmpd.conf", O_RDONLY) = 5
......
open("/usr/local/share/snmp/mibs/SNMP-NOTIFICATION-MIB.txt", O_RDONLY) = 5
open("/usr/local/share/snmp/mibs/SNMPv2-SMI.txt", O_RDONLY) = 6
open("/usr/local/share/snmp/mibs/SNMPv2-TC.txt", O_RDONLY) = 6
......
open("/etc/localtime", O_RDONLY)        = 9
open("/proc/net/dev", O_RDONLY)         = 9
......
--- SIGINT (Interrupt) @ 0 (0) ---
open("/var/net-snmp/snmpd.conf", O_WRONLY|O_CREAT|O_APPEND, 0666) = 9
open("/var/net-snmp/snmpd.conf", O_WRONLY|O_CREAT|O_APPEND, 0666) = 9
......
```

图15-14　strace snmpd open 示例

与 strace 师出同门的还有 ltrace。ltrace 是一款跟踪进程调用库函数的小工具。它们都使用 ptrace 系统调用跟踪进程。感兴趣的读者请参考相关的资料！

15.6小结

本章介绍了 Net-SNMP 代理测试和调试的方法、工具的使用、相关的指导建议，同时融入了作

者的宝贵经验。一旦 SNMP 代理测试完成，我们就可以考虑 strip 掉相关的调试信息（主要是资源相对紧张的嵌入式系统），并准备发布到生产环境中，接受真实环境的考验！

Net-SNMP 测试有多种工具和方法，这些工具应用都非常成熟，只需读者掌握其方法即可胜任相关的测试工作。另外，Net-SNMP 中提供了多种（语言）开发模式。不同的开发模式可以相互借鉴和应用。上述所讲述到的测试方法主要用于测试基于 Net-SNMP 开发的应用程序，而调试方法基本覆盖了以 C 语言模式开发 Net-SNMP 的场景。不过就调试方法和技巧来说，它适用于 Linux 平台 C/C++语言开发的所有应用程序，尤其是 GDB 一节中，笔者按照常规的调试流程讲述了 GDB 实战中非常有意思的方法和技巧。希望读者有所收获！

总之，软件的调试可以用任何工具，手段，方法和思路，而不仅限于此！