**Linux 自检和 SystemTap**

**用于动态内核分析的接口和语言**

现代的操作系统内核提供*自检* 功能，即动态地检查内核以理解其行为的能力。这些行为可以反映内核问题和性能瓶颈。拥有这些信息时候，您就可以调优或修改内核以避免出现故障。本文探索一个名为 SystemTap 的开放源码基础设施，它为 Linux® 内核提供这种动态的自检。

2https://dw1.s81c.com/developerworks/i/v17/dw-cmts-arrow.png [评论](https://www.ibm.com/developerworks/cn/linux/l-systemtap/#icomments)

[M. Tim Jones](https://www.ibm.com/developerworks/cn/linux/l-systemtap/#authorN10024), 自由作家

2009 年 12 月 03 日

* [expand](https://www.ibm.com/developerworks/cn/linux/l-systemtap/#toggle)**内容**



在 IBM Bluemix 云平台上开发并部署您的下一个应用。

[**开始您的试用**](https://developer.ibm.com/sso/bmregistration?lang=zh_CN&ca=dwchina-_-bluemix-_-l-systemtap-_-sidebar)

SystemTap 是监控和跟踪运行中的 Linux 内核的操作的动态方法。这句话的关键词是*动态*，因为 SystemTap 没有使用工具构建一个特殊的内核，而是允许您在运行时动态地安装该工具。它通过一个名为*Kprobes* 的应用编程接口（API）来实现该目的，本文将探索这个 API。我们首先了解以前的一些内核跟踪方法，然后在深入探讨 SystemTap 的架构及其使用。

**内核跟踪**

SystemTap 与一种名为 *DTrace* 的老技术相似，该技术源于 Sun Solaris 操作系统。在 DTrace 中，开发人员可以用 D 编程语言（C 语言的子集，但修改为支持跟踪行为）编写脚本。DTrace 脚本包含许多探针和相关联的操作，这些操作在探针 “触发” 时发生。例如，探针可以表示简单的系统调用，也可以表示更加复杂的交互，比如执行特定的代码行。清单 1 显示了 DTrace 脚本的一个简单例子，它计算每个进程发出的系统调用的数量（注意，使用字典将计数和进程关联起来）。该脚本的格式包含探针（在发出系统调用时触发）和操作（对应的操作脚本）。

**清单 1. 计算每个进程的系统调用的简单 DTrace 脚本**

syscall:::entry

{

@num[pid,execname] = count();

}

DTrace 是 Solaris 最引人注目的部分，所以在其他操作系统中开发它并不奇怪。DTrace 是在 Common Development and Distribution License (CDDL) 之下发行的，并且被移植到 FreeBSD 操作系统中。

另一个非常有用的内核跟踪工具是 *ProbeVue*，它是 IBM 为 IBM® AIX® 操作系统 6.1 开发的。您可以使用 ProbeVue 探查系统的行为和性能，以及提供特定进程的详细信息。这个工具使用一个标准的内核以动态的方式进行跟踪。清单 2 显示了 ProbeVue 脚本的一个例子，它指出发出 sync 系统调用的特定进程。

**清单 2. 指出哪个进程调用 sync 的简单 ProbeVue 脚本**

@@syscall:\*:sync:entry

{

printf( "sync() syscall invoked by process ID %d\n", \_\_pid );

exit();

}

考虑到 DTrace 和 ProbeVue 在各自的操作系统中的巨大作用，为 Linux 操作系统策划一个实现该功能的开源项目是势不可挡的。SystemTap 从 2005 年开始开发，它提供与 DTrace 和 ProbeVue 类似的功能。许多社区还进一步完善了它，包括 Red Hat、Intel、Hitachi 和 IBM 等。

这些解决方案在功能上都是类似的，在触发探针时使用探针和相关联的操作脚本。现在，我们看一下 SystemTap 的安装，然后探索它的架构和使用。

[**回页首**](https://www.ibm.com/developerworks/cn/linux/l-systemtap/#ibm-pcon)

**安装 SystemTap**

您可能仅需一个 SystemTap 安装就可以支持 SystemTap，具体情况取决于您的分发版和内核。对于其他情况，需要使用一个调试内核映像。这个小节介绍在 Ubuntu version 8.10 (Intrepid Ibex) 上安装 SystemTap 的步骤，但这并不是一个具有代表性的 SystemTap 安装。在 [参考资料](https://www.ibm.com/developerworks/cn/linux/l-systemtap/#resources) 部分中，您可以找到在其他分发版和版本上安装 SystemTap 的更多信息。

对大部分用户而言，安装 SystemTap 都非常简单。对于 Ubuntu，使用 apt-get：

$ **sudo apt-get install systemtap**

在安装完成之后，您可以测试内核看它是否支持 SystemTap。为此，使用以下简单的命令行脚本：

$ **sudo stap -ve 'probe begin { log("hello world") exit() }'**

如果该脚本能够正常运行，您将在标准输出 [stdout] 中看到 “hello world”。如果没有看到这两个单词，则还需要其他工作。对于 Ubuntu 8.10，需要使用一个调试内核映像。应该使用 apt-get 获取包 linux-image-debug-generic 就可以获得它的。但这里不能直接使用 apt-get，因此您可以下载该包并使用 dpkg 安装它。您可以下载通用的调用映像包并按照以下的方式安装它：

$ **wget http://ddebs.ubuntu.com/pool/main/l/linux/**

**linux-image-debug-2.6.27-14-generic\_2.6.27-14.39\_i386.ddeb**

$ **sudo dpkg -i linux-image-debug-2.6.27-14-generic\_2.6.27-14.39\_i386.ddeb**

现在，已经安装了通用的调试映像。对于 Ubuntu 8.10，还需要一个步骤：SystemTap 分发版有一个问题，但可以通过修改 SystemTap 源代码轻松解决。查看 [参考资料](https://www.ibm.com/developerworks/cn/linux/l-systemtap/#resources) 获得如何更新运行时 time.c 文件的信息。

如果您使用定制的内核，则需要确保启用内核选项 CONFIG\_RELAY、CONFIG\_DEBUG\_FS、CONFIG\_DEBUG\_INFO 和 CONFIG\_KPROBES。

[**回页首**](https://www.ibm.com/developerworks/cn/linux/l-systemtap/#ibm-pcon)

**SystemTap 的架构**

让我们深入探索 SystemTap 的某些细节，理解它如何在运行的内核中提供动态探针。您还将看到 SystemTap 是如何工作的，从构建进程脚本到在运行的内核中激活脚本。

**动态地检查内核**

SystemTap 用于检查运行的内核的两种方法是 *Kprobes* 和 *返回探针*。但是理解任何内核的最关键要素是内核的映射，它提供符号信息（比如函数、变量以及它们的地址）。有了内核映射之后，就可以解决任何符号的地址，以及更改探针的行为。

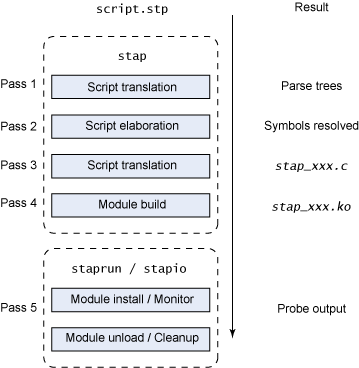
Kprobes 从 2.6.9 版本开始就添加到主流的 Linux 内核中，并且为探测内核提供一般性服务。它提供一些不同的服务，但最重要的两种服务是 Kprobe 和 Kretprobe。Kprobe 特定于架构，它在需要检查的指令的第一个字节中插入一个断点指令。当调用该指令时，将执行针对探针的特定处理函数。执行完成之后，接着执行原始的指令（从断点开始）。

Kretprobes 有所不同，它操作调用函数的返回结果。注意，因为一个函数可能有多个返回点，所以听起来事情有些复杂。不过，它实际使用一种称为 *trampoline* 的简单技术。您将向函数条目添加一小段代码，而不是检查函数中的每个返回点。这段代码使用 trampoline 地址替换堆栈上的返回地址 —— Kretprobe 地址。当该函数存在时，它没有返回到调用方，而是调用 Kretprobe（执行它的功能），然后从 Kretprobe 返回到实际的调用方。

**SystemTap 的流程**

图 1 展示了 SystemTap 的基本流程，涉及到 3 个交互实用程序和 5 个阶段。该流程首先从 SystemTap 脚本开始。您使用 stap 实用程序将 stap 脚本转换成提供探针行为的内核模块。stap 流程从将脚本转换成解析树开始 (pass 1)。然后使用细化（elaboration）步骤 (pass 2) 中关于当前运行的内核的符号信息解析符号。接下来，转换流程将解析树转换成 C 源代码 (pass 3) 并使用解析后的信息和 *tapset 脚本*（SystemTap 定义的库，包含有用的功能）。stap 的最后步骤是构造使用本地内核模块构建进程的内核模块 (pass 4)。

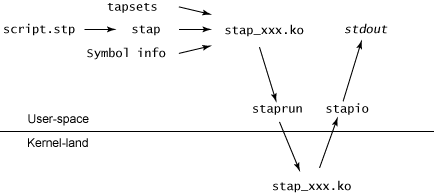
**图 1. SystemTap 流程**



有了可用的内核模块之后，stap 完成了自己的任务，并将控制权交给其他两个实用程序 SystemTap：staprun 和 stapio。这两个实用程序协调工作，负责将模块安装到内核中并将输出发送到 stdout (pass 5)。如果在 shell 中按组合键 Ctrl-C 或脚本退出，将执行清除进程，这将导致卸载模块并退出所有相关的实用程序。

SystemTap 的一个有趣特性是缓存脚本转换的能力。如果安装后的脚本没有更改，您可以使用现有的模块，而不是重新构建模块。图 2 显示了 user-space 和 kernel-space 元素以及基于 stap 的转换流程。

**图 2. 从 kernel/user-space 角度了解 SystemTap 流程**



[**回页首**](https://www.ibm.com/developerworks/cn/linux/l-systemtap/#ibm-pcon)

**SystemTap 脚本编写**

在 SystemTap 中编写脚本非常简单，但也很灵活，有许多您需要使用的选项。[参考资料](https://www.ibm.com/developerworks/cn/linux/l-systemtap/#resources) 提供一个详述语言和可行性的手册的链接，但这个小节仅讨论一些例子，让您初步了解 SystemTap 脚本。

**探针**

SystemTap 脚本由探针和在触发探针时需要执行的代码块组成。探针有许多预定义模式，表 1 列出了其中的一部分。这个表列举了几种探针类型，包括调用内核函数和从内核函数返回。

**表 1. 探针模式例子**

| **探针类型** | **说明** |
| --- | --- |
| begin | 在脚本开始时触发 |
| end | 在脚本结束时触发 |
| kernel.function("sys\_sync") | 调用 sys\_sync 时触发 |
| kernel.function("sys\_sync").call | 同上 |
| kernel.function("sys\_sync").return | 返回 sys\_sync 时触发 |
| kernel.syscall.\* | 进行任何系统调用时触发 |
| kernel.function("\*@kernel/fork.c:934") | 到达 fork.c 的第 934 行时触发 |
| module("ext3").function("ext3\_file\_write") | 调用 ext3 write 函数时触发 |
| timer.jiffies(1000) | 每隔 1000 个内核 jiffy 触发一次 |
| timer.ms(200).randomize(50) | 每隔 200 毫秒触发一次，带有线性分布的随机附加时间（-50 到 +50） |

我们通过一个简单的例子来理解如何构造探针，并将代码与该探针相关联。清单 3 显示了一个样例探针，它在调用内核系统调用 sys\_sync 时触发。当该探针触发时，您希望计算调用的次数，并发送这个计数以及表示调用进程 ID（PID）的信息。首先，声明一个任何探针都可以使用的全局值（全局名称空间对所有探针都是通用的），然后将它初始化为 0。其次，定义您的探针，它是一个探测内核函数 sys\_sync 的条目。与探针相关联的脚本将递增 count 变量，然后发出一条消息，该消息定义调用的次数和当前调用的 PID。注意，这个例子与 C 语言中的探针非常相似（探针定义语法除外），如果具有 C 语言背景将非常有帮助。

**清单 3. 一个简单的探针和脚本**

global count=0

probe kernel.function("sys\_sync") {

count++

printf( "sys\_sync called %d times, currently by pid %d\n", count, pid );

}

您还可以声明探针可以调用的函数，尤其是希望供多个探针调用的通用函数。这个工具还支持递归到给定深度。

**变量和类型**

SystemTap 允许定义多种类型的变量，但类型是从上下文推断得出的，因此不需要使用类型声明。在 SystemTap 中，您可以找到数字（64 位签名的整数）、整数（64 位）、字符串和字面量（字符串或整数）。您还可以使用关联数组和统计数据（我们稍后讨论）。

**表达式**

SystemTap 提供 C 语言中常用的所有必要操作符，并且用法也是一样的。您还可以找到算术操作符、二进制操作符、赋值操作符和指针废弃。您还看到从 C 语言带来的简化，其中包括字符串连接、关联数组元素和合并操作符。

**语言元素**

在探针内部，SystemTap 提供一组类似于 C 一样易于使用的语句。注意，尽管该语言允许您开发复杂的脚本，但每个探针只能执行 1000 条语句（这个数量是可配置的）。表 2 列出了一小部分语句作为例子。注意，在这里的许多元素和 C 中的一样，尽管有一些附加的东西是特定于 SystemTap 的。

**表 2. SystemTap 的语言元素**

| **语句** | **说明** |
| --- | --- |
| if (exp) {} else {} | 标准的 if-then-else 语句 |
| for (exp1 ; exp2 ; exp3 ) {} | 一个 for 循环 |
| while (exp) {} | 标准的 while 循环 |
| do {} while (exp) | 一个 do-while 循环 |
| break | 退出迭代 |
| continue | 继续迭代 |
| next | 从探针返回 |
| return | 从函数返回一个表达式 |
| foreach (VAR in ARRAY) {} | 迭代一个数组，将当前的键分配给 VAR |

本文在样例脚本中探索了统计数据和聚合功能，因为这是 C 语言中不存在的。

最后，SystemTap 提供许多内部函数，这些函数提供关于当前上下文的额外信息。例如，您可以使用 caller() 识别当前的调用函数，使用 cpu() 识别当前的处理器号码，以及使用 pid() 返回 PID。SystemTap 还提供许多其他函数，提供对调用堆栈和当前注册表的访问。

[**回页首**](https://www.ibm.com/developerworks/cn/linux/l-systemtap/#ibm-pcon)

**SystemTap 例子**

在简单介绍了 SystemTap 的要点之后，我们接下来通过一些简单的例子来了解 SystemTap 的工作原理。本文还展示了该脚本语言的一些有趣方面，比如聚合。

**系统调用监控**

前一个小节探索了一个监控 sync 系统调用的简单脚本。现在，我们查看一个更加具有代表性的脚本，它可以监控所有系统调用并收集与它们相关的额外信息。

清单 4 显示的简单脚本包含一个全局变量定义和 3 个独立的探针。在首次加载脚本时调用第一个探针（begin 探针）。在这个探针中，您可以发出一条表示脚本在内核中运行的文本消息。接下来是一个 syscall 探针。注意这里使用的通配符 (\*)，它告诉 SystemTap 监控所有匹配的系统调用。当该探针触发时，将为特定的 PID 和进程名增加一个关联数组元素。最后一个探针是 timer 探针。这个探针在 10,000 毫秒（10 秒）之后触发。与这个探针相关联的脚本将发送收集到的数据（遍历每个关联数组成员）。当遍历了所有成员之后，将调用 exit 调用，这导致卸载模块和退出所有相关的 SystemTap 进程。

**清单 4. 监控所有系统调用 (profile.stp)**

global syscalllist

probe begin {

printf("System Call Monitoring Started (10 seconds)...\n")

}

probe syscall.\*

{

syscalllist[pid(), execname()]++

}

probe timer.ms(10000) {

foreach ( [pid, procname] in syscalllist ) {

printf("%s[%d] = %d\n", procname, pid, syscalllist[pid, procname] )

}

exit()

}

清单 4 中的脚本的输出如清单 5 所示。从这个脚本中您可以看到运行在用户空间中的每个进程，以及在 10 秒钟内发出的系统调用的数量。

**清单 5. profile.stp 脚本的输出**

$ **sudo stap profile.stp**

System Call Monitoring Started (10 seconds)...

stapio[16208] = 104

gnome-terminal[6416] = 196

Xorg[5525] = 90

vmware-guestd[5307] = 764

hald-addon-stor[4969] = 30

hald-addon-stor[4988] = 15

update-notifier[6204] = 10

munin-node[5925] = 5

gnome-panel[6190] = 33

ntpd[5830] = 20

pulseaudio[6152] = 25

miniserv.pl[5859] = 10

syslogd[4513] = 5

gnome-power-man[6215] = 4

gconfd-2[6157] = 5

hald[4877] = 3

$

**特定的进程的系统调用监控**

在这个例子中，您稍微修改了上一个脚本，让它收集一个进程的系统调用数据。此外，除了仅捕捉计数之外，还捕捉针对目标进程的特定系统调用。清单 6 显示了该脚本。

这个例子根据特定的进程进行了测试（在本例中为 syslog 守护进程），然后更改关联数组以将系统调用名映射到计数数据。

**清单 6. 新系统调用监控脚本 (syslog\_profile.stp)**

global syscalllist

probe begin {

printf("Syslog Monitoring Started (10 seconds)...\n")

}

probe syscall.\*

{

if (execname() == "syslogd") {

syscalllist[name]++

}

}

probe timer.ms(10000) {

foreach ( name in syscalllist ) {

printf("%s = %d\n", name, syscalllist[name] )

}

exit()

}

清单 7 提供了该脚本的输出。

**清单 7. 新脚本的 SystemTap 输出 (syslog\_profile.stp)**

$ **sudo stap syslog\_profile.stp**

Syslog Monitoring Started (10 seconds)...

writev = 3

rt\_sigprocmask = 1

select = 1

$

**使用聚合步骤数字数据**

聚合实例时捕捉数字值的统计数据的出色方法。当您捕捉大量数据时，这个方法非常高效有用。在这个例子中，您收集关于网络包接收和发送的数据。清单 8 定义两个新的探针来捕捉网络 I/O。每个探针捕捉特定网络设备名、PID 和进程名的包长度。在用户按 Ctrl-C 调用的 end 探针提供发送捕获的数据的方式。在本例中，您将遍历 recv 聚合的内容、为每个元组（设备名、PID 和进程名）相加包的长度，然后发出该数据。注意，这里使用提取器来相加元组：@count 提取器获取捕获到的长度（包计数）。您还可以使用 @sum 提取器来执行相加操作，分别使用 @min或 @max 来收集最短或最长的程度，以及使用 @avg 来计算平均值。

**清单 8. 收集网络包长度数据 (net.stp)**

global recv, xmit

probe begin {

printf("Starting network capture (Ctl-C to end)\n")

}

probe netdev.receive {

recv[dev\_name, pid(), execname()] <<< length

}

probe netdev.transmit {

xmit[dev\_name, pid(), execname()] <<< length

}

probe end {

printf("\nEnd Capture\n\n")

printf("Iface Process........ PID.. RcvPktCnt XmtPktCnt\n")

foreach ([dev, pid, name] in recv) {

recvcount = @count(recv[dev, pid, name])

xmitcount = @count(xmit[dev, pid, name])

printf( "%5s %-15s %-5d %9d %9d\n", dev, name, pid, recvcount, xmitcount )

}

delete recv

delete xmit

}

清单 9 提供了清单 8 中的脚本的输出。注意，当用户按 Ctrl-C 时退出脚本，然后发送捕获的数据。

**清单 9. net.stp 的输出**

$ **sudo stap net.stp**

Starting network capture (Ctl-C to end)

^C

End Capture

Iface Process........ PID.. RcvPktCnt XmtPktCnt

eth0 swapper 0 122 85

eth0 metacity 6171 4 2

eth0 gconfd-2 6157 5 1

eth0 firefox 21424 48 98

eth0 Xorg 5525 36 21

eth0 bash 22860 1 0

eth0 vmware-guestd 5307 1 1

eth0 gnome-screensav 6244 6 3

Pass 5: run completed in 0usr/50sys/37694real ms.

$

**捕获柱状图数据**

最后一个例子展示 SystemTap 用其他形式呈现数据有多么简单 —— 在本例中以柱状图的形式显示数据。返回到是一个例子中，将数据捕获到一个名为 *histogram* 的聚合中（见清单 10）。然后，使用 netdev 接收和发送探针以捕捉包长度数据。当探针结束时，您将使用 @hist\_log 提取器以柱状图的形式呈现数据。

**清单 10. 步骤和呈现柱状图数据 (nethist.stp)**

global histogram

probe begin {

printf("Capturing...\n")

}

probe netdev.receive {

histogram <<< length

}

probe netdev.transmit {

histogram <<< length

}

probe end {

printf( "\n" )

print( @hist\_log(histogram) )

}

清单 11 显示了清单 10 的脚本的输出。在这个例子中，使用了一个浏览器会话、一个 FTP 会话和 ping 来生成网络流量。@hist\_log 提取器是一个以 2 为底数的对数柱状图（如下所示）。还可以步骤其他柱状图，从而使您能够定义 bucket 的大小。

**清单 11. nethist.stp 的柱状图输出**

$ sudo stap nethist.stp

Capturing...

^C

value |-------------------------------------------------- count

8 | 0

16 | 0

32 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 1601

64 |@ 52

128 |@ 46

256 |@@@@ 164

512 |@@@ 140

1024 |@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@ 2033

2048 | 0

4096 | 0

$

[**回页首**](https://www.ibm.com/developerworks/cn/linux/l-systemtap/#ibm-pcon)

**结束语**

本文仅探索了 SystemTap 的最简单的功能。在 [参考资料](https://www.ibm.com/developerworks/cn/linux/l-systemtap/#resources) 部分中，您可以找到许多教程、例子和语言参考的链接，这些资源提供了解 SystemTap 所需的所有详细信息。SystemTap 使用几个现有的方法并借鉴了以前的内核跟踪实现。尽管该工具还在紧张开发当中，但它现在已经可以使用。请期待未来出现的新特性。