SystemTap学习手册

目 录

第1章 SystemTap概要

1.1 关于SystemTap

SystemTap是一种对Linux系统进行简单的信息收集的工具，遵循GPL协议。可进行复杂的动作和机能问题的诊断。

用户可以通过编写简单的脚本程序来控制使用SystemTap,也可以直接从命令行输入脚本命令。另外，还可以通过stap命令，读取stap程序来控制使用SystemTap。

stap程序是SystemTap的主要构成部分。Stap程序首先解析脚本(script)，将其翻译成C语言代码，然后通过编译器编译此C语言代码，生成内核模块(kernel-Module)。然后，将此内核模块加载进(load)Linux内核中来实现系统的追踪和probe函数的运行——当probe函数所监视的进程的某事件(event)触发了的时候，probe函数开始运行，将事件(event)的信息采集出来。

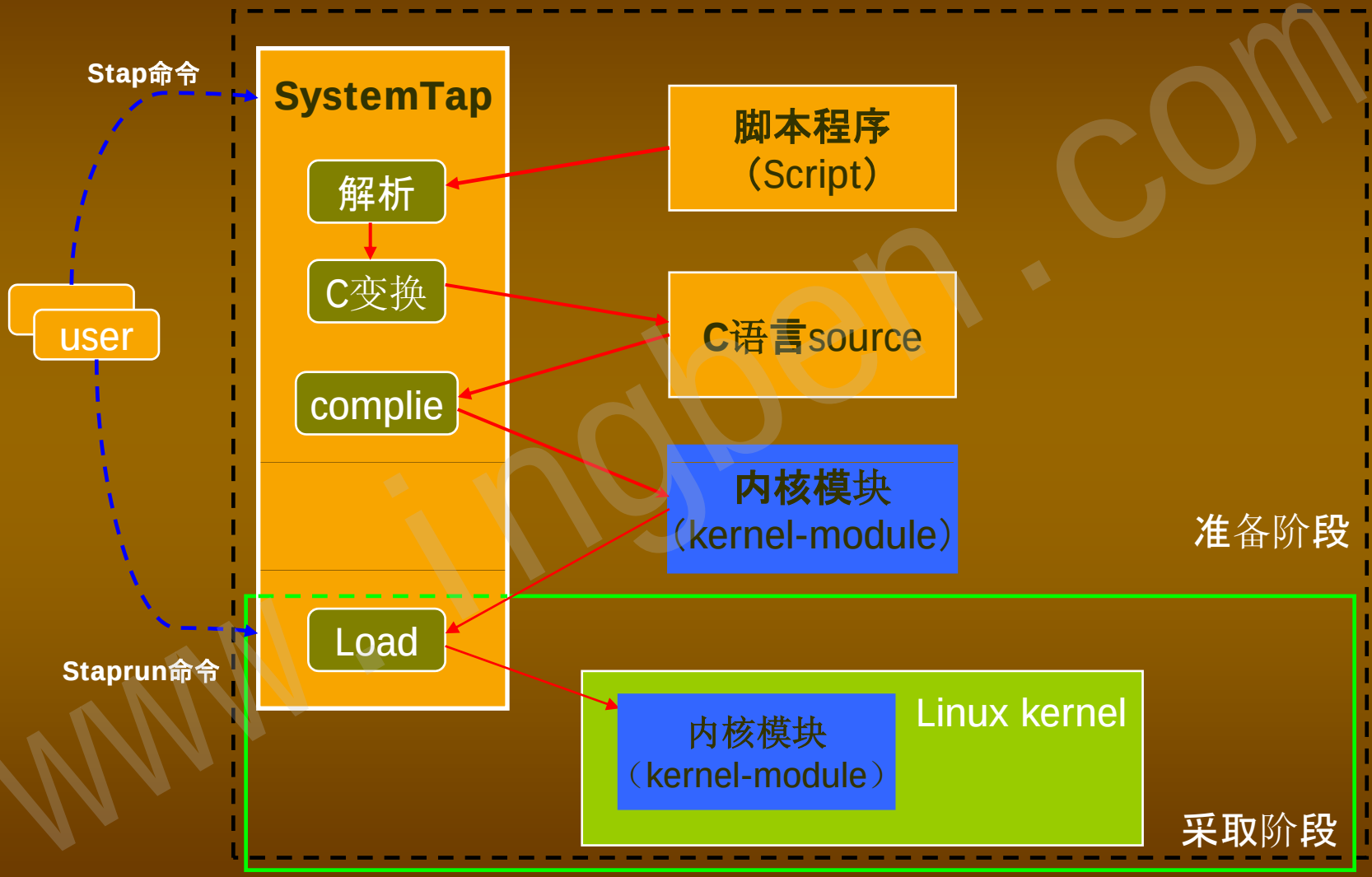
stap程序在用户发出终止请求，或者probe程序内部的exit()函数被执行，或者异常发生的时候终止。终止后，SystemTap的内核模块(kernel-Module)从Linux内核中卸载(unload)。

http://sourceware.org/systemtap/langref/

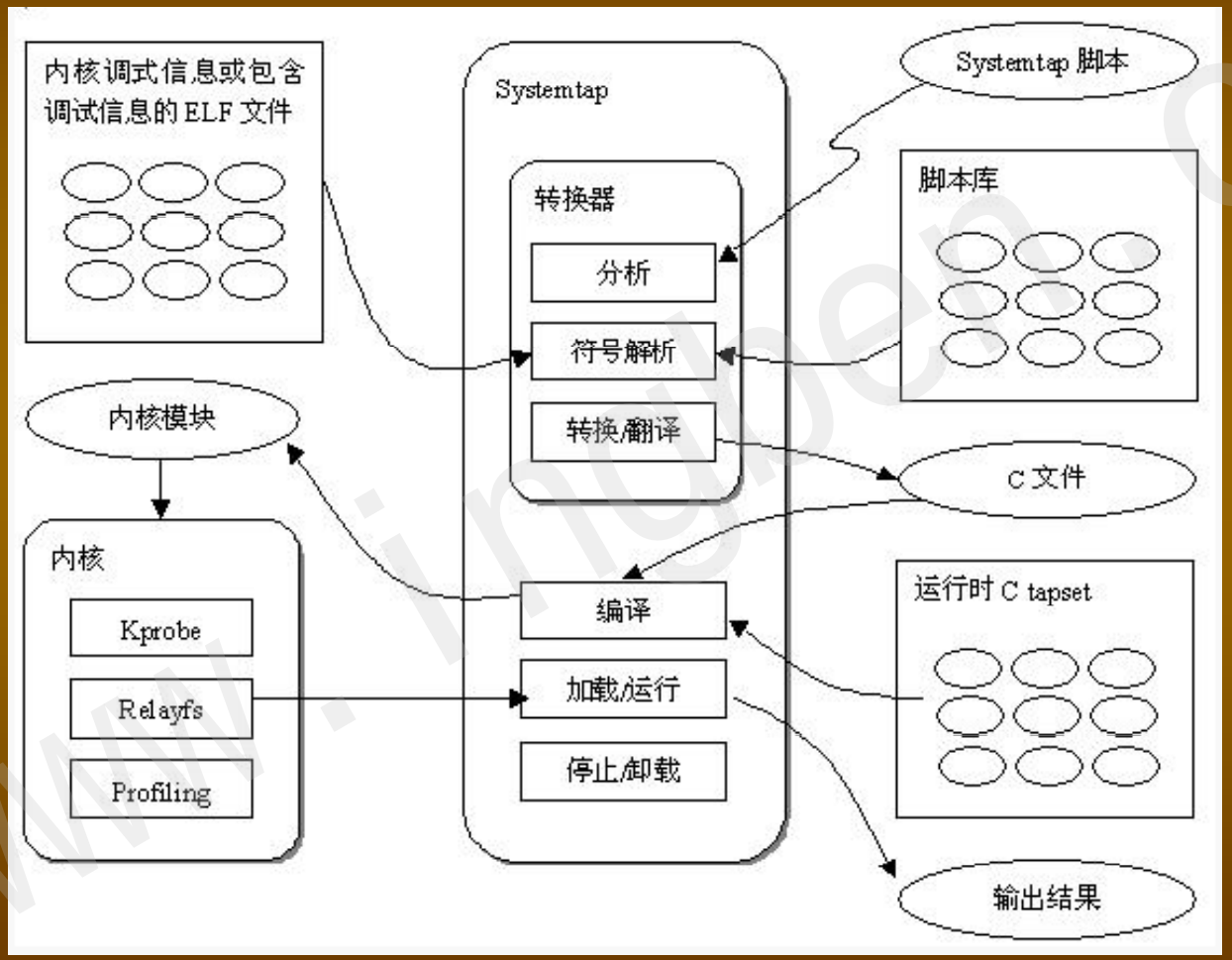
1.2 处理概要

SystemTap的处理概要，如下图所示：

n SystemTap处理流程图



以下是我在网上找的另一幅SystemTap工作原理图，请参照：



第2章 SystemTap安装

- 必需得package：

SystemTap必须要有以下环境才可运行。

Package	说明
systemtap	SystemTap的包。
elfutils	SystemTap需要elfutils软件包提供的库函数来分析调试信息 。
gcc	C语言的编译器。
kernel-devel	编译内核模块所需的内核头文件以及模块配置信息 。
kernel-debuginfo	SystemTap需要通过内核调试信息（kernel-debuginfo ）来定位内核函数和变量的位置 。

- 2.2 安装
（具体安装略）

第3章 stap和staprun命令概要

- 3.1 stap命令概要
- 3.1.1 命令简介

stap命令的形式如下所示:

- stap [OPTIONS] FILENAME [ARGUMENTS]
- stap [OPTIONS] - [ARGUMENTS]
- stap [OPTIONS] –e SCRIPT [ARGUMENTS]

其中：

OPTIONS： 属性

FILENAME： stp文件名（记录了script脚本程序）

ARGUMENTS： 参数

SCRIPT： 脚本命令（直接在命令行输入的）

- （1）stap [OPTIONS] FILENAME [ARGUMENTS]的例子：

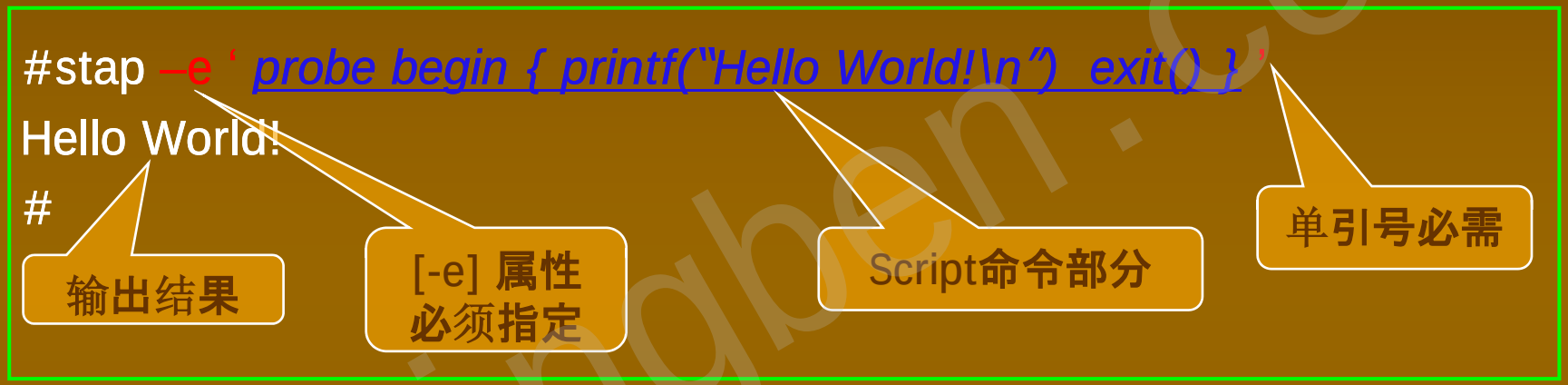
```
#cat hello-world.stp
 probe begin
 {
      printf(“Hello World!\n”)
      exit()
 }

 # stap hello-world.stp
 Hello World!
 #
```

这是个指定了stap文件的例子，本例中不需要输入任何参数。

- （2）stap [OPTIONS] –e SCRIPT [ARGUMENTS]的例子：

  此方法直接在命令行输入脚本(script)命令。此时，必须指定属性[-e]，且脚本(script)命令必须用两个单引号“‘‘”引起来。



- 3.1.2 OPTIONS: 属性 详解

  （具体内容请参照资料）

n 3.1.3 ARGUMENTS: 参数 详解

SystemTap的script中如果想要传入参数，可以通过stap命令直接指定。数值和字符串都可以指定，也可以指定多个参数。

使用以下形式传入参数：

stap [OPTIONS] FILENAME [ARGUMENTS]

如下例：

```
#cat args-test1.stp
probe begin
{
    printf("BEGIN NOW !\n")
    argstr1 = @1;
    argstr2 = @2;
    argint3 = $3;
}
probe timer.ms(1000)  #after 1000 seconds
{
    printf("argstr1 = %s, argstr2 = %s, argint3 = %d\n", argstr1, argstr2,argint3)
    exit()
}
global argstr1, argstr2, argint3

# stap args-test1.stp STR1 STR2 17
BEGIN NOW !
argstr1 = STR1, argstr2 = STR2, argint3 = 17
#
```

参数按顺序传入

stap命令中输入三个参数

- 由上例可以看出，
  数值参数用$1~$n的形式指定，
  字符串参数以@1~@n的形式指定。

- 3.2 staprun命令概要
- 3.2.1 命令简介

staprun命令的形式，如下所示：

- staprun [OPTIONS] MODULE [MODULE-OPTIONS]

其中：
OPTIONS: 属性
MODULE: 内核模块名（kernel-module名）
MODULE-OPTIONS: 内核模块的属性

- 3.2.2 MODULE：属性 详解

（具体属性略，请参照资料。）

- 本章学习小结及问题：

（1）stap命令与staprun命令的区别在于：

stap命令的操作对象是stp文件或script命令等；

staprun命令的操作对象是编译生成的内核模块。（个人观点）

（2）staprun操作的内核模块是如何保存下来的？在将脚本程序翻译成C代码编译成内核模块之后，是否可以在此时将操作中止（PASS 4阶段？？）并将生成内核模块保存下来，以便后面使用staprun进行操作？

（3）staprun命令格式中的[MODULE-OPTIONS],具体的属性有哪些？和3.2.2中的属性相同？？

问题回答：

（1）staprun命令的操作对象是编译生成的内核模块。

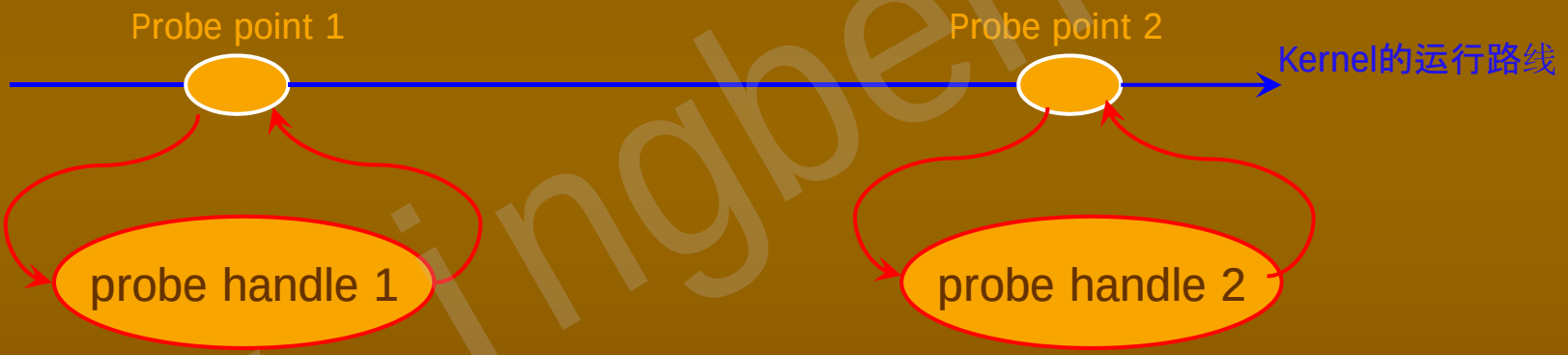
（2）staprun操作的内核模块（以.ko为后缀名）会在pass4阶段自动保存到临时路径下（次路径可由用户指定）。

第4章 脚本语言介绍

n 4.1 关于probe

“probe”是“探测”的意思。是SystemTap进行具体地收集数据的所在。

SystemTap如下图所示的方式，从运行的内核中的probe-point点开始取得信息。



“probe point” 是probe动作的时机。(也就是probe程序监视的某事件点，一旦侦测的事件触发了，则probe将从此处插入内核)

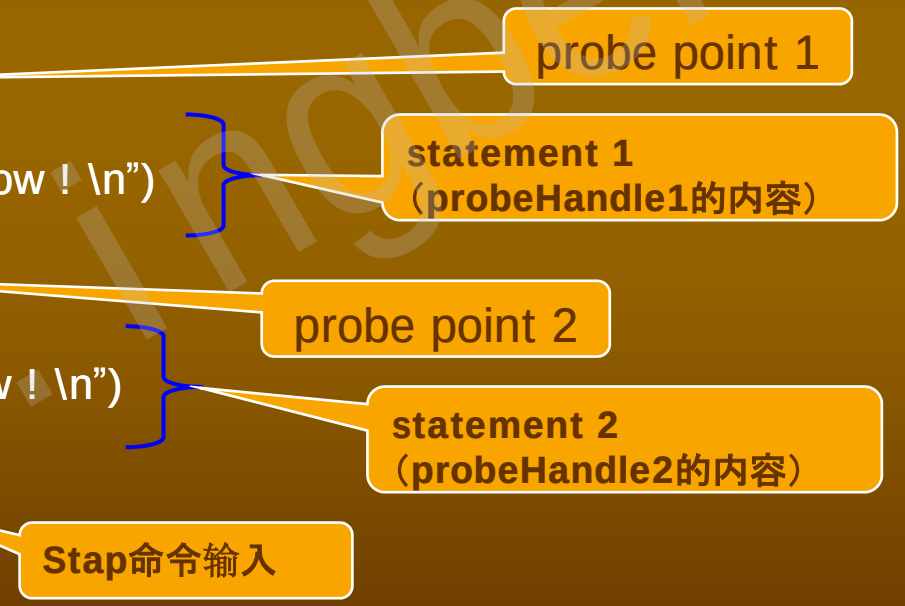
“probe handle” 是当probe插入内核后所做的具体动作。当这个动作完成以后，回归到元内核的处理处。

- probe的基本指定如下所示：
  - probe *probe-Point* { *statement* }

  其中：
  statement也就是probeHandle的处理内容，必须要用一对大括号“{ }“引起来。
  下面给出一个具体的例子：

例1：

```
#cat exp1.stp
probe begin
{
    printf(“Begin Now ! \n”)
}
probe end
{
    printf(“End Now ! \n”)
}
#stap exp1.stp
Begin Now !
End Now !
#
```



- 上例中，有begin和end两个probe-Point。begin是SystemTap启动时直接（最先）执行的probe-Point，当begin的probe-Handle做完以后，此probe程序并没有结束。按下[Ctrl+C]后，SystemTap将停止，此时会先处理end这个probe-Point，end执行完成后SystemTap才结束。

- 也可以直接用script命令实现，如下：

例2：

```
# stap –e ‘ probe begin{ print(“Begin Now ! \n”) }  probe end{print(“End Now !\n”)} ’

Begin Now !
End Now !
#
```

Probe-Point1　statement1　Probe-Point2　statement2

输出

例1是将probe-point和statement记录在exp1.stp文件中的，例2是直接在命令行输入script命令，两者功能相同，但注意例2中【-e】属性必需，且命令需要用一对单引号引起来。多个probe-point之间不需要分号分隔。

- 4.2 script程序的基本语法
- 4.2.1 脚本语言的变量

变量由基本的拉丁字母和数字组合而成，不可以包含”_“和”$”等字符。

script中不需要明确声明变量类型，脚本语言会根据函数参数等自动判断变量是什么类型的。

局部变量：在声明的probe和block（”{ }“范围内的部分）内有效。

全局变量：用”global“声明的变量，在此SystemTap的整个动作过程中都有效。全局变量的声明位置没有具体要求。通常全局变量都要进行初期化。

具体例子请参照资料。

- 4.2.2 脚本语言的正文（statement）

（a.）注释：

三种注释方式：

#..... ：Shell语言风格

//..... ：C++语言风格

/*...*/ ：C语言风格

（b.）基本书写方式：

脚本程序都要用一对花括号引用起来，有多段statement时，在各段代码之间可以用 ”;“分隔开，也可以不用。一行代码可分多行写。

（c.）if语句：

if语句与其他语言的语法相似，指出一点：if()的条件的结果可以不是bool型，条件的结果和0比较，非0的结果if判断都认为是true，0 认为是false。

（d.）while语句：

while的判断条件也可以不是bool型，非0值都认为是true，0认为是false。

（e.）for/foreach语句：

与其他语言相似，略。

（f.）break/continue语句：

break：跳出当前循环体。

continue：跳出本次循环，做下一次循环。

（g.）return语句：

用来返回函数的结果。无返回值时可不写。

（h.）next语句：

调用此语句时，立刻从调用函数中退出。不同于exit()的是，next只是退出当前的调用函数，而此SystemTap并没有终了，但exit()则会终止SystemTap。

（i.）delete语句：

语句形式：delete 配列名[index1, index2,...]

作用：从配列中删除掉index1,index2,...等数据，被删除的数据不可以再被使用。如果想要删除不存在的数据，也不会报错。（？）

delete allay语句：

语句形式：delete 配列名

作用：删除掉配列内的所有数据。

delete scalar语句：

语句形式：delete 配列名

作用:删除掉scalar的值。整数和字符串分别清空为0和空字符“”。全部重置为初始状态。

- 4.2.3 操作符

脚本语言的操作符和C语言的相同，构成、意义、优先级等也基本相似。

特别指出几点:

(1). “.”操作符: 连接两个字符串，

如str1 = “abc” str2 = “def” 则:str1.str2 为“abcdef”

(2). 比较运算符也可以使用在字符串比较时。如上面的str1 > str2 是成立的。

(3). 唯一的三目运算符: [条件] ？ [statement1] : [statement2]

条件成立时执行statement1的内容，否则执行statement2的内容。

- 4.2.4 函数

如果函数有若干个传入参数，则至少要有一个传出参数。

- 函数和函数的参数的声明可以省略，由编译器自动推断。但是必要的时候，还是要显式地声明一下。
- 4.2.5 输出

SystemTap有四种标准输出函数：

（1）print()函数

输出后不会自动换行。

（2）sprint()函数

将内容以字符串形式输出。

（3）printf()函数

输出后会自动换行。

（4）sprintf()函数

将内容以字符串形式输出，再输出一个换行符。

- 4.2.6 统计（特有的操作符和方法）

（1）操作符“<<<”

例： global g_value

g_value <<< VALUE

作用:“<<<”操作符类似于“+=”操作符，会将新赋予的值累加上原有的值。

（2）相关的函数

函 数	说 明
@count(g_value)	所有统计操作的操作次数
@sum(g_value)	所有统计操作的操作数的总和
@min(g_value)	所有统计操作中操作数的最小值
@max(g_value)	所有统计操作中操作数的最大值
@avg(g_value)	所有统计操作中操作数的平均值

```
global x
probe begin
{
    x <<< $1 ; x <<< $2;  x <<< $3
}
 probe end
 {
     printf(“ avg %d = sum %d / count %d\n”, @avg(x), @sum(x), @count(x))
 }
```

```
#stap  operation.stp 10 20 30
```

- 4.2.7 嵌入C语言代码

  脚本语言中可以嵌入C代码。要嵌入的C代码必须以“ %{ ”和“ %} ”作为函数的开始与结束。因为在Pass 3阶段将脚本翻译成C代码，所以可以允许在脚本中嵌入C，翻译时此段C代码不翻译。另，执行脚本程序时，需添加“-g”选项才可正确执行嵌C的代码。(见前面例子)

- 4.2.8 关于Tapset

  在/usr/share/systemtap/tapset/目录下提供了一些stp文件，可参阅。

- 4.3 注意点

- 4.3.1 SystemTap工具必须有管理员权限才可使用，所以安全方面要注意。

- 4.3.2

一个prob可以有多个event，每个event间用逗号“，”相隔，比如：

```
probe event1, event2, …eventN
{
    statements
}
```

表示若干个event做相同的事情。

- 4.3.3 联合数组的使用

联合数组一般在多个probe之间被使用，所以是被声明为global类型的。例如数组foo用来记录年龄，则可如下表示：

```
foo[“Marry”] = 22
foo[“Harry”] = 23
foo[“Tom”] = 24
```

最多只能指定5个元素，如

```
device[pid(),execname(),uid(),ppid(),"W"] = devname
```

联合数组必须被定义为global，不管是在一个还是多个probe间使用。

使用delete来删除或重置数组内容。

第5章 probe详解

probe	概要说明
BEGIN	SystemTap 开始时的动作。
END	SystemTap 结束时的动作。
NEVER	不做动作。
TIMERS	周期性动作。
DWARF	在source和object内的某点动作。
MARKERS	与内核或模块内写入的静态的probe-marker挂钩。

以上为probe的一些主要probe-point。下面将具体介绍：

n 5.1 BEGIN

「BEGIN」是SystemTap的执行顺序中，session的开始(startup)时执行的探测点。所有的全局变量的初始化工作，必须在此节点前进行。目标变量(target variables)不可以在此探测点内使用。(？？)

(target variables : 以“$”赋值的变量为目标变量)

例:

```
#cat begin-end-test.stp
probe begin
 {
      printf(“ Begin Now !\n ”)
 }
 # stap begin-end –test.stp
  Begin Now !
 #
```

启动时只执行一次

n 5.2 END

[END]探测点在SystemTap执行顺序中session终了时执行。例如，exit()函数调用时、或者用户强制的终止时。注意，由于异常错误导致的终止时，[END]探测点不会被执行。目标变量在此探测点内不可用。（？？）

例：

```
#cat begin-end-test.stp
probe end
{
    printf(“ End Now  !\n ”)
}
# stap begin-end-test.stp
  End Now  !
#
```

终了时只执行一次

- 补充：

begin和end探测点可以通过指定一个执行顺序数（sequence number ）来控制执行的先后顺序。如果不指定，则执行顺序数默认为0，且按照在脚本中出现的先后顺序来执行。

例如：

```
# In a tapset file:
probe begin(-1000) { ... }

# In a user script:
probe begin { ... }
```

因为tapset file中的begin的执行顺序数小于user file的顺序（user file的执行数为默认的0），所以tapset文件的begin探测点先执行。

n 5.3 NEVER

[never]探测点是被翻译器定义成的所谓的什么也不做(never)。Never探测点的内容会被进行文法和类型的正确性的分析，但是never探测点的probe-handle永远不会被执行。

n 5.4 TIMERS

我们可以通过由标准内核的[jiffies time](standard kernel jiffies timer)定义的时间间隔来异步地触发probe-handle。

[jiffies time] 是由内核定义的具有代表性的时间单位，这个时间单位的大小在1~60msec.之间。

翻译器支持两种probe-point：

```
timer.jiffies(N)
timer.jiffies(N).randomize(M)
```

上面形式中：probe-handler每N个jiffies时间便执行一次。

如果指定了随机数M，则每次执行probe-handler时，一个线性的随机数（这个随机数位于[-M ... +M]之间）将会与N相加。

N与M的范围也有合理性限制：N必须位于（1~近似100万）的范围内，M必须小于N。

目标变量（target variables）不可以在此probe内使用。

此probe可以在多处理器的机器上并发执行。

- 另外，间隔也可以由时间单位来指定。有两个probe-point与jiffies timer相似：

```
timer.ms(N)
timer.ms(N).randomize(M)
```

#stap time-s.stp

- 此处，N和M都是以毫秒为单位。

所有的选项如下图所示：

其他几种单位的选项及形式：

单位	选项	记录形式
秒	s	timer.s(N)、timer.s(N).randomize(M)
毫秒	ms	timer.ms(N)、timer.ms(N).randomize(M)
微妙	us	timer.us(N)、timer.us(N).randomize(M)
纳秒	ns	timer.ns(N)、timer.ns(N).randomize(M)
赫兹	hz	timer.hz(N)

注意：随机数randomize（M）不支持赫兹单位。

小结：第一种形式是以自定义的时间单位为时间间隔，
第二种形式是以标准的时间单位为时间间隔。

5.5 DWARF

// 该系列探测点对kernel、module、program等使用象征性的调式信息作为可以

下面列出的是目前支持的probe point：

kernel.function(PATTERN)
kernel.function(PATTERN).call
kernel.function(PATTERN).return
kernel.function(PATTERN).renturn.maxactive(VALUE)
kernel.function(PATTERN).inline

可以表示成：
kernel.inline(PATTERN)

module(MPATTERN).function(PATTERN)
module(MPATTERN).function(PATTERN).call
module(MPATTERN).function(PATTERN).return.maxactive(VALUE)
module(MPATTERN).function(PATTERN).inline

可以表示成：
Module(MPATTERN).inline(PATTERN)

kernel.statement(PATTERN)
kernel.statement(ADDRESS).absolute
module(MPATTERN).statement(PATTERN)

- .function变量指定了probe的开始位置。(probe的开始位置会放在.function指定的位置)
- .return变量将probe放置在function返回值的时候(即当function返回值时触发此probe)，此返回值是可用的，以[$return]的形式返回。传入参数也是可用的，即便可能在函数中已经改变了它的值。.maxactive表示有多少个被指定的函数的实例可以同时被执行。
- .inline变量和.function很相似，探测点是从.inline处插入。.inline是.function中只包含内联函数(inlined Function)的实例。.call变量则正好代表了剩下的子集。内联函数(inlined Function)没有可确定的返回点，所以.inline的probe不支持.return。
- .statement变量将探测点放置在指定位置，在这个位置可以收集到局部变量的值。
- 实例:

```
kernel.function("*init*"),  kernel.function("*exit*")
kernel.function("*@kernel/sched.c:240")
module("scsi_mod").inline("scsi_*")
kernel.statement(0x00800000)
kernel.statement("*@kernel/sched.c:2410")
module("*").statement(0xc0044852)
```

n 5.6 MARKERS

下面是内核文档(kernel documentation)中对Markers的一些描述:

(http://sourceware.org/systemtap/wiki/UsingMarkers)

代码中出现的Marker(标记)提供了一个钩子(hook，相当于接口？)，这个hook可以调用用户在运行时提供的函数(probe)。一个Marker可以有“on”和“off”两种状态(“开”与“关”的状态吧)，“on”的状态时，probe可以被连接上它，“off”的状态时，没有与任何probe连接。

当Marker是“off”状态时，除了用来添加一个微小的时间penalty(time penalty)和空间penalty(space penalty)，别无它用。(time penalty: 为一个分支检查某种状态或条件; space penalty: 为函数调用增加一些字节到instrumented function末尾，并且增加一个数据结构在指定的部位。)

当Marker是“on”状态时，每次这个Marker被执行时都会去调用你所提供的函数(probe)，这个调用是在调用者的执行过程中实现的。当提供的函数执行完时，返回到调用者处(从Marker的位置继续)。

我们可以在代码的重要位置设置一个Marker。Marker可以传递任意数量的参数和格式化字符串到所连接的probe函数中去。

Marker可以用来追踪和执行累加处理(accounting)。

- Marker的形式：

```
#include <linux/marker.h>
//...
int kfunc(struct inode *i, int op)
{ int rc = 0;      // return code
  trace_mark(kfunc_entry, "inode %p op %d", i, op);

  //... bulk of kfunc() here...
  trace_mark(kfunc_exit, "rc %d", rc);
  return(rc);
}
```

- 内核版本从2.6.24开始支持Marker，但仍需要打上三个补丁包，具体操作可见网站。

- 如何在SystemTap中使用Marker？

  你需要使用‘kernel.mark(“NAME”)’的形式来讲一个SystemTap的probe和kernel marker挂上钩。

- 如下所示：

```
probe kernel.mark("kfunc_entry") { printf("kfunc_entry marker hit\n") }
probe kernel.mark("kfunc_exit") { printf("kfunc_exit marker hit\n") }
```

当kfunc()被内核调用的时候，这个SystemTap的两个probe都可能被执行到然后你会看到适当的输出。

- 如何访问marker的标准字符串？

某个probe（marker-based probe）可能在marker被调用的地方读取一个标准字符串。这个标准字符串是以“$format”的形式命名的：

```
probe kernel.mark("kfunc_entry") { printf("kfunc_entry marker hit: %s, %p, %d\n",
$format, $arg1, $arg2) }
```

第6章 probe中常用函数一览

- 本章主要是对/usr/share/systemetap/tapset下的一些标准的，由tapset提供的函数进行介绍。

  函数一览如下，后续章节会对这些函数做详细说明：

函数群	章节	包含的函数
LOGGING	6.1	log(), print(), printf(), warn(), error(), exit()
CONVERSIONS	6.2	kernel_string(), user_string()
STRING	6.3	strlen(), substr(), isinstr()
TIMESTAMP	6.4	get_cycles(), gettimeofday_ns(), gettimeofday_us(), gettimeofday_ms(), gettimeofday_s()
CONTEXTINFO	6.5	cpu(), execname(), pexecname(), tid(), pid(), ppid(), uid(), euid(), gid() egid(), print_regs(), backtrace(), print_stack(), print_backtrace(), pp(), probefunc(), probemod(), target(), is_return()
ERRNO	6.6	errno_str()
QUEUE_STATS	6.7	qs_wait(), qs_run(), qs_done(), qsq_start(), qsq_print(), qsq_utilization(), qsq_blocked(), qsq_wait_queue_length(), qsq_service_time(), qsq_wait_time(), qsq_throughput()

函数群	章节	包含的函数
INDENT	6.8	thread_indent(), thread_timestamp()
SYSTEM	6.9	system()
NUMA	6.10	addr_to_node()
CTIME	6.11	ctime()
PERFMON	6.12	read_counter()

下面将分别对以上函数群做详细介绍：

- 6.1 LOGGING

（1）log：unknown（msg：string）

将指定的文字列记入共有buffer内，并在行尾追加字符串结束符 。

（2）print：unknown（…）

将指定的整数、文字列或者静态的值输出到共有buffer内。

（3）printf：unknown（fmt：string，…）

与C语言的printf相同用法，可通过%s和%d分别传入字符串和整形参数。

（4）warn:unknown(msg:string)

此函数将指定的文字列记入警告流中，并在行尾添加字符串结束符。使用staprun时，会添加前缀“WARNING:”。

（5）error:unknown(msg:string)

此函数将指定的文字列记入错误流中，并在行尾添加字符串结束符。使用staprun时，会添加前缀“ERROR:”。如果错误数超过了最大错误数（MAXERRORS），将触发一个exit()。

n 6.2 CONVERSIONS

（1）kernel_string:string(addr:long)

从内核空间的给定地址处copy出字符串。对这个地址的确认只是局部的。

（2）user_string:string(addr:long)

从用户空间的给定地址处copy出字符串。对这个地址的确认只是局部的。

6.3 STRING

（1）strlen:long(str:string)

返回str的长度。

（2）substr:string(str:string, start:long, stop:long)

从str字符串中的start索引处开始，取得stop长度的子串。

（3）isinstr:long(s1:string. s2:string)

如果s1包含s2，则返回1，否则返回0.

6.4 TIMESTAMP

（1）get_cycles:long()

此函数返回处理器循环计数器的值(the processor cycle counter value)，如果不可用时，返回0。

（2）gettimeofday_ns:long()

返回从公元1970年1月1日0时开始至今的纳秒数。

（3）gettimeofday_us:long()

返回从公元1970年1月1日0时开始至今的毫秒数。

（4）gettimeofday_ms:long()

返回从公元1970年1月1日0时开始至今的微秒数。

（5）gettimeofday_s:long()

返回从公元1970年1月1日0时开始至今的秒数。

n 6.5 CONTEXTINFO

(1) cpu:long()

返回当前的CPU数量。

（2）execname:string()

返回当前的进程名。

（3）pexecname:string()

返回当前的进程的父进程名。

（4）tid:long()

返回当前线程(Thread)的ID。

（5）pid:long()

返回当前进程(Process)的ID。

(6) ppid:long()

返回父进程的进程ID。

(7) uid:long()

返回当前任务的用户ID。

(8) euid:long()

返回当前进程的有效的用户ID。

(9) gid:long()

返回当前进程的组ID。

(10) egid:long()

返回当前进程的有效的组ID。

(11) print_regs:unknown()

此函数打印出注册表缓存(register dump???)。

(12) backtrace:string()

以16进制地址的字符串形式显示堆栈的回溯信息。此字符串将可能被截断以适应字符串的最大长度。

(13) print_stack:unknown(bt:string)

此函数将在由backtrace()返回的字符串中查找地址。每一行打印一个地址。每一行包含地址，含有地址的函数，和在此函数中的大概位置。此函数没有返回值。

（14）print_backtrace:unknown()

此函数与print_stack(backtrace())功能相同,只是此函数支持更深层的堆栈嵌套。此函数也不返回任何值。

（15）pp:string()

此函数返回与当前正在执行的probe-handler关联的probe-point，并包含别名和wild-card扩展效果。

（16）probefunc:string()

此函数返回probe-point的函数名。

（17）probemod:string()

此函数返回包含probe-point的模块名。

（18）target:long()

此函数返回目标进程的进程ID。多和“-x PID”或者“-c <COMMAND>”等选

项连用。

-x <PID>:

target()返回由-x指定的进程的进程ID。

-c <COMMAND>:

target()返回进程ID以执行由-c指定的系统命令。

（19）is_return:long()

如果此probe-point是一个返回的probe，则此函数返回1，否侧返回0.

n 6.6 ERRNO

(1) errno_str:string(e:long)

指定的与错误代码关联的记号以字符串的形式返回。例如:2代表“ENOENT”;E#3333代表一个越界值，如3333.

n 6.7 QUEUE_STATS

queue_stats tapset提供了一些函数，这些函数当接收到队列的通知（等待，执行，执行完了）时，会追踪平均（例如队列长度、服务时间、等待时间、利用等）。一下三个函数在适当的probe中必须按照顺序调用。

(1) qs_wait:unknown(qname:string)

此函数记录了：将一条新的请求插入到指定的请求队列中（使其处于等待状态）。

（2）qs_run:unknown(qname:string)

此函数记录了：将之前入队的请求，从指定的等待队列中移除，并开始被服务（从等待状态转为执行状态）。

（3）qs_done:unknown(qname:string)

此函数记录了：指定的队列中的请求已经被服务结束。

- 以“qsq_”为前缀的函数用来查询从第一条请求被执行，或者qsq_start()被执行开始的平均值进行统计。因为此统计数经常是非常微小的，所以在此结果上增加了一个刻度参数。对于结果是分数的，以百分比的形式输出。

（4）qsq_start:unknown(qname:string)

此函数为指定的队列重置统计计数器，并且在此函数被调用的同时重新启动追踪(track)。这个命令在创建一个队列时被使用。

（5）qsq_print:unknown(qname:string)

此函数为给定的队列输出一行信息，此信息包含以下内容：

- queue name
- average rate of requests per second
- average wait queue length
- average time on the wait queue
- average time to service a request
- percentage of time the wait queue was used
- percentage of time any request was being serviced

（6）qsq_utilization:long(qname:string, scale:long)

此函数返回至少一条请求被服务的平均时间（以微妙为单位）

（7）qsq_blocked:long(qname:string, scale:long)

此函数返回一条或多条请求在等待队列中所花费的时间长。（即等待队列的长度。）

（8）qsq_wait_queue_length:long(qname:string, scale:long)

此函数返回等待队列的平均长度。

（9）qsq_service_time:long(qname:string, scale:long)

此函数返回服务请求所花费的平均时间（以微秒为单位）。

（10）qsq_wait_time:long(qname:string, scale:long)

此函数返回某请求从进入等待队列到被执行完所花费的平均时间（以微秒为单位）。（qs_wait() -> qs_done()）

(11) qsq_throughput:long(qname:string, scale:long)

此函数返回单位时间内的请求的平均数。（每微秒处理的请求数的平均值）

n 6.8 INDENT

indent是缩进的意思。

（1）thread_indent:string(delta:long)

此函数返回一个被适当缩进之后的字符串。用一个正的或负的小值delta来调用此函数。返回的字符串中追加了相对的时间戳(timestamp)、进程名、线程ID、delta值对应的缩进量等。

delta的指定值与时间戳(timestamp)以及缩进(indent)之间的关系如下：

delta的指定值	timestamp	indent
delta > 0	最初值作为0	向右缩进(增加)delta的值个空格
delta = 0	0	无
delta < 0	最初值作为0	向左缩进(减少)delta的值个空格

（2）thread_timestamp:long()

此函数为使用缩进功能的函数返回一个绝对时间戳值。

6.9 SYSTEM

system(cmd:string)

此函数执行一条系统命令。当当前probe执行完成时此命令会在后台运行。

6.10 NUMA

addr_to_node(addr:long)

此函数接受一个地址，然后返回此地址在NUMA系统中所属的节点。

6.11 CTIME

ctime:string(seconds:long)

此函数将指定的秒数(此秒数由gettimeofday_s()返回得到)转换为简单的记述形式，如“Wed Jun 30 21:49:008 2006”的形式。

- 6.12 PERFMON

  read_counter:long(handle:long)

  返回处理器的性能计数器。此perfmon的主体部分应该进行设置以便记录被某事件使用了的句柄(handle)。

- 另外，可参考以下链接：
- http://sourceware.org/systemtap/langref/node11.html#SECTION00011840000000000000
- http://sourceware.org/systemtap/man/stapfuncs.5.html

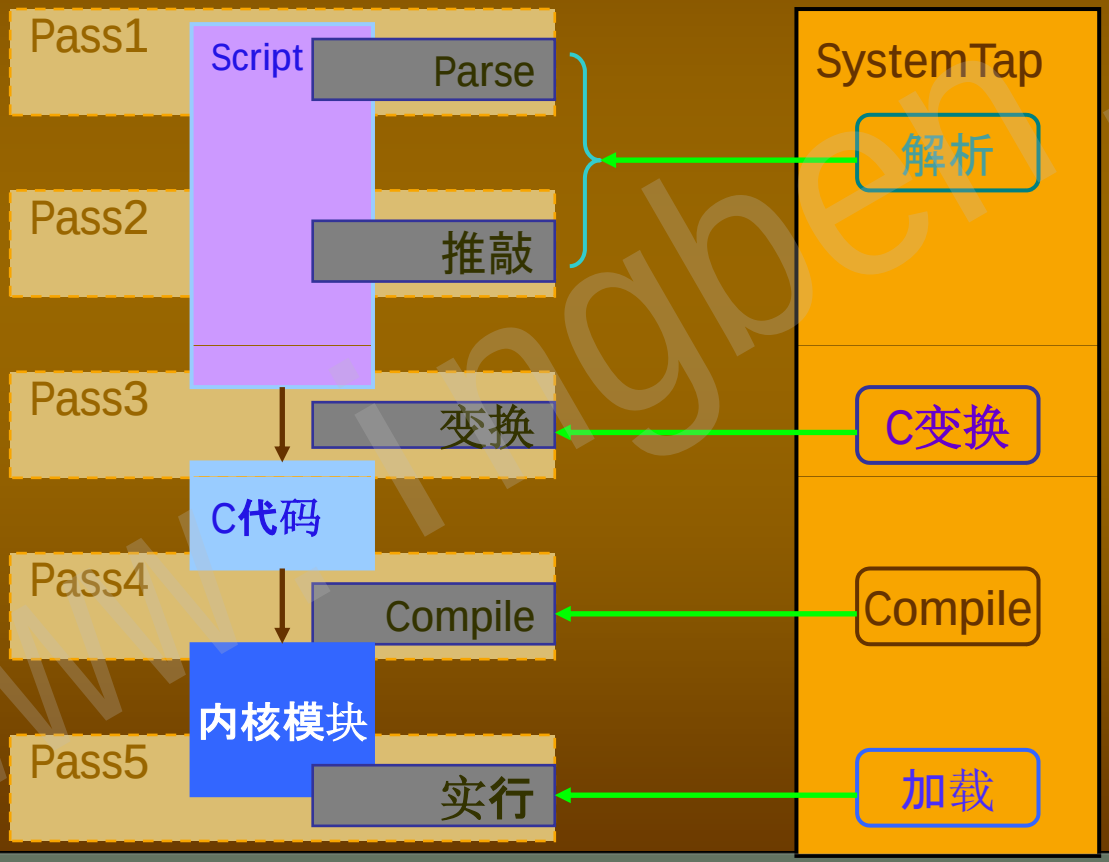
- 补充：thread_indent(delta)

  返回值为：经历的时间(ms)、进程名、进程ID以及由delta确定的缩进空格数。

  probefunc()返回被进程调用的函数名。

n 附录A　SystemTap的处理流程

SystemTap一系列的处理流程大概是：解析script文件、script文件翻译成C语言代码、编译C语言代码（生成内核模块）、加载内核模块等。如下图所示：



用户可以通过-p NUM选项来自行确定执行到第几步。

- （1）Pass 1：

  解析由命令行和文件（以.stp为后缀名）指定的script程序。如果用户指定了选项“-p 1”的话，在Pass1处理完之后便停止，打印出parse tree。

- （2）Pass 2：

  a. 解决符号和类型的问题

  为解决内部未解决的变量、函数等，将检索tapset内的script。此工作在所有的符号被解决了之后才返回。

  b. 最适化

  分析所有的probe和函数，将无效的变量和未使用的函数去除掉。

  注：最适化时，会将类型不匹配的变量和无效的目标变量隐式地作为代码错误。可以使用“-u”选项使最适化功能无效。

  c. 最适化

  根据上下文推断所有的变量、函数、参数、配列等的类型，如果有矛盾或者未解决的类型存在，则当做代码错误。

注：如果用户指定“-p 2”选项，则在Pass 2阶段执行完后便停止，然后list出所有的probe、函数、变量等。

- （3）Pass 3：

  a. 翻译成C语言代码

  Pass2阶段之后，将所有选择了的script文件和同等的动作写成C代码，生成Makefile文件，然后将这些文件放入临时目录下。

  注：如果用户指定“-p 3”选项，则在Pass3阶段执行完后便停止，然后输出翻译后的C代码的内容。

- （4）Pass 4：

  a. 内核模块做成

  调用Linux内核的编译系统（build System），做成内核模块（内核目标）。

  注：如果不想执行此内核模块的话，在Pass4阶段停止是最后的机会了。（Pass4之后，编译出来的内核将被加载）

- （5）Pass 5：

  a. 加载内核模块

  调用staprun程序，将内核目标模块加载，执行。

结束时，将模块卸载，清除（cleanup）。

- 注：可以通过staprun命令，来再利用Pass4阶段生成的内核模块。

附录B　SystemTap的sample和选项

- （1）-k 选项

```
#stap -k test3.stp
Begin now……!
Tue May 19 15:21:17 2009
End now!
Keeping temporary directory "/tmp/stapCsaeYL“
#
```

指定-k 选项后，将会多打印出“Keeping temporary directory …”这条信息，此临时目录下保存了Pass3和Pass4阶段生成的C代码和内核模块。

n （2）-b 选项

```
# stap -v test3.stp
Pass 1: parsed user script and 38 library script(s) in 570usr/10sys/582real ms.
Pass 2: analyzed script: 2 probe(s), 3 function(s), 1 embed(s), 1 global(s) in 10usr/0sys/12real ms.
Pass 3: using cached /root/.systemtap/cache/18/stap_183dfd83c271f99a1b46fd35932be177_3183.c
Pass 4: using cached /root/.systemtap/cache/18/stap_183dfd83c271f99a1b46fd35932be177_3183.ko
Pass 5: starting run.
Begin now.....!
Tue May 19 15:33:03 2009
End now!
Pass 5: run completed in 60usr/40sys/2792real ms.
```

-b选项**会解析、编译，但不执行。**

```
# stap -b -v test3.stp
Pass 1: parsed user script and 38 library script(s) in 550usr/20sys/579real ms.
Pass 2: analyzed script: 2 probe(s), 3 function(s), 1 embed(s), 1 global(s) in 10usr/0sys/11real ms.
Pass 3: using cached /root/.systemtap/cache/8e/stap_8e244751d660a4840ef0fff5fa0e5693_3183.c
Pass 4: using cached /root/.systemtap/cache/8e/stap_8e244751d660a4840ef0fff5fa0e5693_3183.ko
Pass 5: starting run.
```

- （3）-m 选项：指定生成的文件的名称：

```
# stap -m TempName -k -v  test3.stp
Warning: using '-m' disables cache support.
Pass 1: parsed user script and 38 library script(s) in 560usr/20sys/583real ms.
Pass 2: analyzed script: 2 probe(s), 3 function(s), 1 embed(s), 1 global(s) in 10usr/0sys/13real ms.
Pass 3: translated to C into "/tmp/stapZZzjTY/TempName.c" in 10usr/0sys/2real ms.
Pass 4: compiled C into "TempName.ko" in 6280usr/750sys/7050real ms.
Pass 5: starting run.
Begin now......!
Tue May 19 15:48:14 2009
End now!
Pass 5: run completed in 60usr/60sys/1959real ms.
Keeping temporary directory "/tmp/stapZZzjTY"
# ls -l /tmp/stapZZzjTY/
total 580
-rw-r--r-- 1 root root   1407 May 19 23:48 Makefile
-rw-r--r-- 1 root root      0 May 19 23:48 Module.symvers
-rw-r--r-- 1 root root  20573 May 19 23:48 TempName.c
-rw-r--r-- 1 root root 271942 May 19 23:48 TempName.ko
-rw-r--r-- 1 root root   3600 May 19 23:48 TempName.mod.c
-rw-r--r-- 1 root root  47620 May 19 23:48 TempName.mod.o
-rw-r--r-- 1 root root 225580 May 19 23:48 TempName.o
#
```

通过-m可以指定生成的文件名

- （4）-x选项：将target()值设置为PID

```
#cat test4.stp
Probe begin
{
    printf(“Begin….\n”)
}
probe syscall.open
{
    if(pid() == target())
    {
        printf(“Mach!! Pid = %d\n”, pid())
        exit()
    }
}
probe end
{
   printf(“End !\n”)
}
#stap –x 4865 test4.stp
Begin….
Mach!! Pid = 4865
End !
```

-x选项会将参数作为target()的值传给pid()