SystemTap 简介

李瑞彬 (cheeselee@fedoraproject.org)

2013 年 8 月 17 日

概述

- 可编程的(脚本化)系统跟踪工具
- Tap: 窃听
- Linux 专用，旨在替代并超越传统 Unix 中的 DTrace 工具
- 模仿 AWK 语言
- 内核层工具，触及全局环境，获取所有运行时信息
- 侵入式，可跟踪正在运行的程序
- 一般用 root 权限运行

概述(续)

- 通用工具，并不直接针对某种目的
- 区别于：tcpdump
- 没有智能，不能直接告诉你程序哪里出问题了
- 区别于：Valgrind/memcheck
- 使用 SystemTap 建基于对代码的熟悉

基本用途

- 跟踪：在一段时间内，某进程执行了哪些系统调用以及次数
- 计时：一个函数执行了多长时间
- 采样：进程函数调用栈与火焰图 (Flame Graph)
- 其它的用途可随意发挥

必要: 安装 SystemTap

- `yum install systemtap systemtap-runtime`
- 检查当前内核的开发包已安装
- `rpm -q kernel-devel-`uname -r``
- 没装?`yum update kernel; yum install kernel-devel; reboot`
- 注意：根据具体内核类型要把 kernel 换成 kernel-PAE 或 kernel-xen

安装调试信息

- `yum install /usr/bin/debuginfo-install`
- 安装内核调试符号 (内核态跟踪):
- `debuginfo-install kernel-`uname -r``
- 安装依赖库的调试符号 (用户态跟踪) :
- `debuginfo-install glibc gcc`

检查内核是否支持用户态跟踪

- 内核编译时配置：CONFIG_UTRACE=y 或 CONFIG_UPROBES=y
- `grep -E '(UTRACE|UPROBES)' /boot/config-`uname -r``
- Fedora 及 RHEL/CentOS 5+ 默认内核都支持

Hello world

- 例：hello.stp
- 运行：stap hello.stp

```
probe begin
{
  print("hello␣")
  exit()
}

probe end
{
  print("world!\n")
}
```

探测点(事件)

- 以 `probe` 指定探测点(事件)
- 一些简单的事件，详见 man stapprobes：
- begin：在整个探测的最开始运行
- end：在整个探测结束后运行
- syscall.read：read 系统调用
- syscall.*：所有系统调用
- vfs.read：往文件系统读取文件
- timer.profile：快速地周期执行
- `-L`：stap 命令选项，用于列表探测点

哪些进程在调用系统调用 read

执行 read 系统调用的进程：who-is-reading.stp

```
probe syscall.read
{
  printf("[%d] %s\n", pid(), execname())
}
```

哪些进程(真的)在做磁盘 IO

who-missed-kernel-cache.stp

```
probe ioblock.request
{
  printf("[%d] %s\n", pid(), execname())
}
```

运行：

```
stap -c "cat hello.stp" who-missed-kernel-cache.stp
```

获取信息

- 调用函数来取得状态信息
- `pid()`：进程 ID
- `execname()`：进程名(程序文件名)
- `probefunc()`：事件发生时所在的函数名
- `gettimeofday_s()`：当前 Unix 时间

自定义事件异名(事件集)

执行指定系统调用的进程：who-is-reading-or-writing.stp

```
probe read_or_write = syscall.read, syscall.write
{
  print("I␣am␣here\n")
}

probe read_or_write
{
  printf("[%d]␣%s␣calls␣%s\n", pid(), execname(),
         probefunc())
}
```

变量

- 动态类型
- `foo = 10`
- `foo = "a-string"`
- 强类型不能`1 + "1"`
- 声明全局变量`global foo`
- 键值对 (hash/dict)
- `bar["yes"] = 234`
- 组合 hash 键
- `bar["yes",30,"no"] = 432`
- 相当于 Python 的 `bar["yes"][30]["no"]`

通过字典变量保存信息

统计系统调用：syscall-counting.stp

```
global syscalls
probe syscall.* {
  syscalls[execname() . "/" . probefunc()]++
}
probe timer.ms(5000) {
  exit()
}
probe end {
  foreach (entry+ in syscalls) # 键的字典序升序
  {
    printf("%s : %d\n", entry, syscalls[entry])
  }
}
```

聚合变量

- `global a`
- `a <<< 10`
- `a <<< 30`
- `a <<< 40`
- `@count(a) # 3` 插值次数
- `@sum(a) # 80` 总和
- `@min(a) # 10`
- `@max(a) # 40`
- `@avg(a) # 26`

使用聚合变量计算

函数执行所用的时间片：probe_slow_sum.stp

```
global sum_count, whole_count
probe timer.profile {
  if (execname() == "slow_sum") {
    whole_count <<< 1;
    if (probefunc() == "sum") {
      sum_count <<< 1;
    }
  }
}
probe end {
  printf("进程: %d ticks\n", @count(whole_count));
  printf("sum: %d ticks\n", @count(sum_count));
}
```

执行

- `gcc -g -std=c99 -o slow_sum slow_sum.c`
  `stap --ldd -d slow_sum -c ./slow_sum probe_slow_sum.stp`
- `-c [executable]`：stap 随指定程序启动和结束
- `-d [elf]`：载入 elf 文件的调试信息
- `--ldd`：通过 elf 的链接信息添加调试符号

C 代码中加入 Probe mark

- 要安装 systemtap-sdt-devel
- `#include <sys/sdt.h>`
- 加入 probe mark:
- `STAP_PROBE(provider, probe)`
- alias `DTRACE_PROBE`
- 例如：`STAP_PROBE(slow_sum, mymark)`
- 就可以使用跟踪点：
  `process("..").provider("slow_sum").mark("mymark")`
- 如果没有冲突的话 provider 可以省略

程序变量

- $a 对应于程序中的 a 变量
- $$vars 以字符串形式打印当前作用域的所有变量
- $$parms 以字符串形式打印当前函数的参数
- print_var_slow_sum.stp

```
probe process("slow_sum").mark("mymark")
{
  printf("a = %d\n", $a)
  printf("str: %s\n", user_string($str))
}
```

牛力：程序变量作为左值

- 改变程序变量的值，改变程序行为
- 改变a 的返回值
- change_target_variable.stp

```
probe process("slow_sum").mark("mymark")
{
  printf("stap: a = %d\n", $a)
  printf("stap: change a to 223\n");
  $a = 223;
  printf("stap: a = %d\n", $a)
}
```

定义函数

一秒钟后我落在哪个进程: stap-function.stp

```
function mylog (str)
{
  printf("%s[%d]: %s\n", execname(), pid(), str)
}

probe timer.ms(1000)
{
  mylog("I got into this process")
  exit()
}
```

tapset

- tapset: 预定义的函数或事件
- 自动遍历目录: /usr/share/systemtap/tapset/
- log 函数定义于 /usr/share/systemtap/tapset/logging.stp

```
probe timer.ms(1000)
{
  log("I got into this process")
  exit()
}
```

深入 tapset 定义

- tcpdumplike.stp
- tcp.receivce
- /usr/share/systemtap/tapset/linux/tcp.stp
- probe tcp.receivce = tcp.ipv4.receive, tcp.ipv6...
- tcp.ipv4.receive = kernel.fucntion("tcp_v4_rcv")
- tcp_v4_rcv 定义于net/ipv4/tcp_ipv4.c
- 通过$skb 程序变量获取 IP 头
- struct sk_buff 定义于include/linux/skbuff.h

综合应用 FlameGraph

- `stap-emacs.svg`
- 完整用户态调用栈的采样
- 横座标是采样数
- 纵座标是栈高
- 横座标方向越长的函数在程序中运行时间越长
- 很可能是程序速度的瓶颈，可以从此入手优化

生成 FlameGraph(1)

- 下载 FlameGraph 工具
- `git clone git://github.com/brendangregg/FlameGraph.git`
- 选择在跟踪的程序 (`/usr/bin/emacs`)
- 安装该程序的调试符号 (`debuginfo-install emacs`)
- 选择采样的时间段 (程序启动阶段)
- 准备 SystemTap 脚本

```
probe timer.profile {
  if (execname() == "emacs") {
    print_ubacktrace()
    print("\n");
  }
}
```

生成 FlameGraph(2)

- 运行 SystemTap 脚本

```
sudo stap --ldd -d /usr/bin/emacs -c /usr/bin/emacs \
          > emacs-stack.stp
```

- 统计重复的栈采样

```
./extract_block_for_flamegraph.pl emacs-stack.out \
          > out.stap-stacks
```

- 折叠重叠的栈采样

```
./stackcollapse-stap.pl out.stap-stacks \
          > out.stap-folded
```

- 从折叠重叠的栈采样生成交互式 SVG 图片

```
./flamegraph.pl out.stap-folded > stap-emacs.svg
```

参考

“自己动手”，“丰衣足食”— 毛泽东

- 教程：http://sourceware.org/systemtap/tutorial/
- 新手指导：http://sourceware.org/systemtap/SystemTap_Beginners_Guide/
- man stap: SystemTap 语法与命令行参数
- man stapprobes: 常用内置的探测点
- /usr/share/doc/systemtap-client-2.3