

# WolfcsTech

[🏠 首页](#)[📁 归档](#)[👤 关于](#)[📧 订阅](#)

## Simpleperf介绍

📅 Feb 21, 2017 | 📁 Android开发 | 👤 43 Hits



## 什么是simpleperf

Simpleperf是Android平台的一个本地层性能分析工具。它的命令行界面支持与linux-tools perf大致相同的选项，但是它还支持许多Android特有的改进。

Simpleperf是Android开源项目 (AOSP) 的一部分。其源代码 [位于](#)。其最新的文档 [位于](#)。Bugs 和 功能需求可以提交到 [github](#)上。

## Simpleperf是如何工作的

现代的CPU具有一个硬件组件，称为性能监控单元 (PMU)。PMU具有一些硬件计数器，计数一些诸如 经历了多少次CPU周期，执行了多少条指令，或发生了多少次缓存未命中 等的事件。

Linux内核将这些硬件计数器包装到硬件perf事件 (hardware perf events)中。此外，Linux内核还提供了独立于硬件的软件事件和跟踪点事件。Linux内核通过 `perf_event_open` 系统调用将这些都暴露给了用户空间，这正是simpleperf所使用的机制。

Simpleperf具有三个主要的功能：`stat`，`record` 和 `report`。

`Stat`命令给出了在一个时间段内被分析的进程中发生了多少事件的摘要。以下是它的工作原理：

1. 给定用户选项，simpleperf通过对linux内核进行系统调用来启用分析。
2. Linux 内核在调度到被分析进程时启用计数器。
3. 分析之后，simpleperf从内核读取计数器，并报告计数器摘要。

`Record` 命令在一段时间内记录剖析进程的样本。它的工作原理如下：

1. 给定用户选项，simpleperf通过对linux内核进行系统调用来启用分析。
2. Simpleperf 在simpleperf 和 linux 内核之间创建映射缓冲区。
3. Linux 内核在调度到被分析进程时启用计数器。
4. 每次给定数量的事件发生时，linux 内核将样本转储到映射缓冲区。
5. Simpleperf 从映射缓冲区读取样本并生成 `perf.data`。

`Report` 命令读取“`perf.data`”文件及所有被剖析进程用到的共享库，并输出一份报告，展示时间消耗在了哪里。

## 主 simpleperf 命令

Simpleperf 支持一些子命令，包括 `list`，`stat`，`record`，`report`。每个子命令支持不同的选项。这一节只描述最重要的子命令和选项。要了解所有的子命令和选项，请使用 `-help`。

```
1  # List all subcommands.
2  $simpleperf --help
3  # Print help message for record subcommand.
4  $simpleperf record --help
5
```

## simpleperf list

simpleperf list 被用于列出设备上所有可用的事件。由于应该和内核的差异，不同的设备可以支持不同的事件。

```
1 $simpleperf list
2 List of hw-cache events:
3   branch-loads
4   ...
5 List of hardware events:
6   cpu-cycles
7   instructions
8   ...
9 List of software events:
10  cpu-clock
11  task-clock
12  ...
```

## simpleperf stat

simpleperf stat 被用于获取被剖析程序或系统范围内的原始事件计数器信息。通过传入选项，我们可以选择使用哪些事件，监视哪个进程/线程，监视多长时间，以及打印的间隔。下面是一个例子。

```
1 # Stat using default events (cpu-cycles,instructions,...), and monitor
2 # process 7394 for 10 seconds.
3 $simpleperf stat -p 7394 --duration 10
4 Performance counter statistics:
5 1,320,496,145 cpu-cycles # 0.131736 GHz (100%)
6 510,426,028 instructions # 2.587047 cycles per instruction (100%)
7 4,692,338 branch-misses # 468.118 K/sec (100%)
8 886.008130(ms) task-clock # 0.088390 cpus used (100%)
9 753 context-switches # 75.121 /sec (100%)
10 870 page-faults # 86.793 /sec (100%)
11 Total test time: 10.023829 seconds.
12
13
```

## 选择事件

我们可以通过 `-e` 选项选择使用哪个事件。下面是例子：

```
1 # Stat event cpu-cycles.
2 $simpleperf stat -e cpu-cycles -p 11904 --duration 10
3 # Stat event cache-references and cache-misses.
4 $simpleperf stat -e cache-references,cache-misses -p 11904 --duration 10
5
```

当运行 `stat` 命令时，如果硬件事件的数量大于 PMU 中可用的硬件计数器的数量，则内核在事件间共享硬件计数器，因此每个事件只在总时间中的一部分内被监视。在下面的例子中，每一行的最后都有一个百分比，展示了每个事件实际被监视的时间占总时间的百分比。

```
1 # Stat using event cache-references, cache-references:u,...
2 $simpleperf stat -p 7394 -e cache-references,cache-references:u,cache-references:k,cache-misses,cache-misses:u
3 Performance counter statistics:
4 4,331,018 cache-references # 4.861 M/sec (87%)
5 3,064,089 cache-references:u # 3.439 M/sec (87%)
6 1,364,959 cache-references:k # 1.532 M/sec (87%)
7 91,721 cache-misses # 102.918 K/sec (87%)
8 45,735 cache-misses:u # 51.327 K/sec (87%)
9 38,447 cache-misses:k # 43.131 K/sec (87%)
10 9,688,515 instructions # 10.561 M/sec (89%)
11 Total test time: 1.026802 seconds.
12
13
```

在上面的例子中，每个事件被监视的时间大概占总时间的 87%。但是，不保证任何一对事件总是在相同的时间被监视。如果我们想要让一些事件在同一时间被监视，我们可以使用 `-group` 选项。下面是一个例子。

```
1 # Stat using event cache-references, cache-references:u,...
2 $simpleperf stat -p 7394 --group cache-references,cache-misses --group cache-references:u,cache-misses:u --group ca
3 Performance counter statistics:
4 3,638,900 cache-references # 4.786 M/sec (74%)
5 65,171 cache-misses # 1.790953% miss rate (74%)
```

```
6 2,390,433 cache-references:u # 3.153 M/sec (74%)
7 32,280 cache-misses:u # 1.350383% miss rate (74%)
8 879,035 cache-references:k # 1.251 M/sec (68%)
9 30,303 cache-misses:k # 3.447303% miss rate (68%)
10 8,921,161 instructions # 10.070 M/sec (86%)
11 Total test time: 1.029843 seconds.
12
13
```

## 选择监视目标

我们可以通过 `-p` 选项或 `-t` 选项选择监视哪个进程或线程。监视一个进程如同监视进程中的所有线程。Simpleperf 也可以 `fork` 一个子进程来运行新命令，然后监视子进程。下面是例子。

```
1 # Stat process 11904 and 11905.
2 $simpleperf stat -p 11904,11905 --duration 10
3 # Stat thread 11904 and 11905.
4 $simpleperf stat -t 11904,11905 --duration 10
5 # Start a child process running `ls`, and stat it.
6 $simpleperf stat ls
7
8
```

## 决定监视多长时间

当监视已有线程时，我们可以使用 `-duration` 选项决定监视多长时间。当监视执行一个新命令的子进程时，simpleperf 将一直监视子进程直至其结束。在这种情况下，我们可以使用 `Ctrl-C` 在任何时间停止监视。例子如下。

```
1 # Stat process 11904 for 10 seconds.
2 $simpleperf stat -p 11904 --duration 10
3 # Stat until the child process running `ls` finishes.
4 $simpleperf stat ls
5 # Stop monitoring using Ctrl-C.
6 $simpleperf stat -p 11904 --duration 10
7 ^C
8
9
```

## 决定打印的间隔

当监视 perf 计数器时，我们还可以使用 `-interval` 选项决定打印的间隔。例子如下。

```
1 # Print stat for process 11904 every 300ms.
2 $simpleperf stat -p 11904 --duration 10 --interval 300
3 # Print system wide stat at interval of 300ms for 10 seconds (rooted device only).
4 # system wide profiling needs root privilege
5 $su 0 simpleperf stat -a --duration 10 --interval 300
6
```

## 在 systrace 中显示计数器

simpleperf 还可以与systrace一起工作来将计数器转储进收集的trace中。下面是一个执行系统范围的 stat 的例子。

```
1 # capture instructions (kernel only) and cache misses with interval of 300 milliseconds for 15 seconds
2 $su 0 simpleperf stat -e instructions:k,cache-misses -a --interval 300 --duration 15
3 # on host launch systrace to collect trace for 10 seconds
4 (HOST)$external/chromium-trace/systrace.py --time=10 -o new.html sched gfx view
5 # open the collected new.html in browser and perf counters will be shown up
```

## simpleperf record

simpleperf record用于转储被剖析程序的记录。通过传入选项，我们可以选择使用哪个事件，监视哪个进程/线程，以什么频率转储记录，监视多长时间，以及将记录存储到哪里。

```
1 # Record on process 7394 for 10 seconds, using default event (cpu-cycles),
2 # using default sample frequency (4000 samples per second), writing records
3 # to perf.data.
4 $simpleperf record -p 7394 --duration 10
5 simpleperf I 07-11 21:44:11 17522 17522 cmd_record.cpp:316] Samples recorded: 21430. Samples lost: 0.
```

## 选择事件

在大多数情况下，cpu-cycles 事件被用于评估消耗的CPU时间。作为一个硬件事件，它精确而高效。我们还可以通过 -e 选项使用其它事件。下面是一个例子。

```
1 # Record using event instructions.
2 $simpleperf record -e instructions -p 11904 --duration 10
```

## 选择监视目标

record命令中选择目标的方式与 stat 命令中的类似。例子如下。

```
1 # Record process 11904 and 11905.
2 $simpleperf record -p 11904,11905 --duration 10
3 # Record thread 11904 and 11905.
4 $simpleperf record -t 11904,11905 --duration 10
5 # Record a child process running `ls`.
6 $simpleperf record ls
7
8
```

## 设置记录的频率

我们可以通过 -f 或 -c 选项设置转储记录的频率。比如，-f 4000 意味着当监视的线程运行时每秒转储接近 4000 个记录。如果监视的线程一秒钟运行了 0.2 s（其它时间它可能被抢占或阻塞），simpleperf 每秒转储大约  $4000 * 0.2 / 1.0 = 800$  个记录。另一种方式是使用 -c 选项。比如，-c 10000 意味着每发生 10000 次事件转储一个记录。例子如下。

```
1 # Record with sample frequency 1000: sample 1000 times every second running.
2 $simpleperf record -f 1000 -p 11904,11905 --duration 10
3 # Record with sample period 100000: sample 1 time every 100000 events.
4 $simpleperf record -c 100000 -t 11904,11905 --duration 10
5
```

## 决定监视多长时间

record 命令中决定监视多长时间的方式与 stat 命令中的类似。例子如下。

```
1 # Record process 11904 for 10 seconds.
2 $simpleperf record -p 11904 --duration 10
3 # Record until the child process running `ls` finishes.
4 $simpleperf record ls
5 # Stop monitoring using Ctrl-C.
6 $simpleperf record -p 11904 --duration 10
7 ^C
8
9
```

## 设置存储记录的路径

默认情况下，simpleperf 将记录保存至当前文件夹下的 perf.data 文件中。我们可以使用 -o 选项设置存储记录的路径。下面是一个例子。

```
1 # Write records to data/perf2.data.
2 $simpleperf record -p 11904 -o data/perf2.data --duration 10
```

## simpleperf report

simpleperf report 被用来基于 simpleperf record 命令生成的 perf.data 产生报告。Report 命令将记录分组为不同的样本项，基于每个样本项包含的事件的多少对样本项排序，并打印每个样本项。通过传入选项，我们可以选择到哪里寻找被监视的程序使用的 perf.data 和 可执行二进制文件，过滤不感兴趣的记录，并决定如何分组记录。

下面是一个例子。记录被分为 4 个样本项，每项一行。有一些列，每列展示了属于一个样本项的信息片段。第一列是 Overhead，它展示了当前样本项中的事件占总事件的百分比。由于 perf 事件是 cpu-cycles，overhead 可被视为是每个函数占用的 cpu 的百分比。

```
1 # Reports perf.data, using only records sampled in libsudo-game-jni.so,
2 # grouping records using thread name(comm), process id(pid), thread id(tid),
3 # function name(symbol), and showing sample count for each row.
4 $simpleperf report --dsos /data/app/com.example.sudogame-2/lib/arm64/libsudo-game-jni.so --sort comm,pid,tid,symbol
5 Cmdline: /data/data/com.example.sudogame/simpleperf record -p 7394 --duration 10
6 Arch: arm64
```



```
7 | Event: cpu-cycles (type 0, config 0)
8 | Samples: 28235
9 | Event count: 546356211
10 | Overhead Sample Command Pid Tid Symbol
11 | 59.25% 16680 sudogame 7394 7394 checkValid(Board const&, int, int)
12 | 20.42% 5620 sudogame 7394 7394 canFindSolution_r(Board&, int, int)
13 | 13.82% 4088 sudogame 7394 7394 randomBlock_r(Board&, int, int, int, int)
14 | 6.24% 1756 sudogame 7394 7394 @plt
15 |
```

## 设置存储记录的路径

默认情况下, simpleperf 读取当前目录下的 perf.data。我们可以使用 -i 选项选择从另一个文件读取记录。

```
1 | $simpleperf report -i data/perf2.data
```

## 设置查找可执行二进制文件的路径

如果要生成函数符号报告, simpleperf 需要读取被监视的进程使用的可执行二进制文件来获取符号表和调试信息。默认情况下, 路径是记录时被监视的进程使用的可执行二进制文件, 然而, 在生成报告时这些二进制文件可能不存在, 或不包含符号表和调试信息。因此我们可以使用 -symfs 来重定向路径。下面是一个例子。

```
1 | $simpleperf report
2 | # In this case, when simpleperf wants to read executable binary /A/b,
3 | # it reads file in /A/b.
4 | $simpleperf report --symfs /debug_dir
5 | # In this case, when simpleperf wants to read executable binary /A/b,
6 | # it prefers file in /debug_dir/A/b to file in /A/b.
7 |
```

## 过滤记录

当生成报告时, 可能不是对所有记录都感兴趣。Simpleperf 支持五钟过滤器来选择感兴趣的记录。下面是例子。

```
1 # Report records in threads having name sudogame.
2 $simpleperf report --comms sudogame
3 # Report records in process 7394 or 7395
4 $simpleperf report --pids 7394,7395
5 # Report records in thread 7394 or 7395.
6 $simpleperf report --tids 7394,7395
7 # Report records in libsudo-game-jni.so.
8 $simpleperf report --dsos /data/app/com.example.sudogame-2/lib/arm64/libsudo-game-jni.so
9 # Report records in function checkValid or canFindSolution_r.
10 $simpleperf report --symbols "checkValid(Board const&, int, int);canFindSolution_r(Board&, int, int)"
11
12
13
14
```

## 决定如何将记录分组为样本项

Simpleperf 使用 `-sort` 选项决定如何分组样本项。下面是例子。

```
1 # Group records based on their process id: records having the same process
2 # id are in the same sample entry.
3 $simpleperf report --sort pid
4 # Group records based on their thread id and thread comm: records having
5 # the same thread id and thread name are in the same sample entry.
6 $simpleperf report --sort tid,comm
7 # Group records based on their binary and function: records in the same
8 # binary and function are in the same sample entry.
9 $simpleperf report --sort dso,symbol
10 # Default option: --sort comm,pid,tid,dso,symbol. Group records in the same
11 # thread, and belong to the same function in the same binary.
12 $simpleperf report
13
14
15
```

## simpleperf 的特性

Simpleperf 的工作方式与 `linux-tools-perf` 类似，但它有如下的提升：

1. Aware of Android environment. Simpleperf 在剖析时处理一些Android特有的情形。例如，它可以剖析 apk 中嵌入的共享库，从 .gnu\_debugdata 段读取符号表和调试信息。如果可能，当出现错误时它会给出一些提示，如如何禁用perf\_harden启用分析。
2. 记录时支持展开。如果我们想使用 -g 选项来记录并报告一个程序的调用图，我们需要转储每个记录中的用户栈和寄存器集合，然后展开栈来查找调用链。Simpleperf 支持在记录时展开，因此它无需在 perf.data 中存储用户栈。因而我们可以在设备上有限的空间内剖析更长的时间。
3. 支持脚本使Android上的剖析更方便。
4. 编译进静态的二进制文件。Simpleperf 是一个静态库，因此它无需支持运行共享库。这意味着对于 simpleperf 可以运行的Android 版本没有限制，尽管有些设备不支持剖析。

## ndk中的Simpleperf工具

ndk 中的 simpleperf 工具包含三个部分：在 Android 设备上运行的 simpleperf 可执行文件，在主机上运行的 simpleperf 可执行文件，和 python 脚本。

### 设备上的 simpleperf

在设备上运行的 simpleperf 位于 bin/android 目录下。它包含不同体系架构的 Android 上运行的静态二进制文件。它们可被用于剖析运行在设备上的进程，并生成 perf.data。

### 主机上的 simpleperf

运行与主机上的 Simpleperfs 位于bin/darwin, bin/linux 和 bin/windows。它们可被用于在主机上解析 perf.data。

### 脚本

脚本被用于使得剖析和解析剖析结果更方便。app\_profiler.py 被用于剖析一个 android 应用程序。它准备剖析环境，下载 simpleperf 到设备上，在主机上生成并拉出 perf.data。它由 app\_profiler.config 配置。binary\_cache\_builder.py 被用于从设备拉出本地层二进制文件到主机上。它由 app\_profiler.py 使用。annotate.py 被用于使用 perf.data 注解源文件。它由 annotate.config 配置。report.py 在一个 GUI 窗口中报告 perf.data。simpleperf\_report\_lib.py 被用于枚举 perf.data 中的样本。在内部它使用 libsimpleperf\_report.so 来解析 perf.data。它可被用于将 perf.data 中的样本翻译为其它形式。使用 simpleperf\_report\_lib.py 的一个例子是 report\_sample.py。

# 使用 simpleperf 工具的例子

这个部分展示了如何使用 simpleperf 工具来剖析一个 Android 应用。

## 准备一个可调试(debuggable)的应用

应用程序的包名是 com.example.sudogame。它包含 java 代码和 c++ 代码。我们需要运行一份在其 AndroidManifest.xml 元素中 android:debuggable="true" 的app，因为我们不能为 non-debuggable apps 使用 run-as。应用应该已经安装在设备上了，而且我们可以通过 adb 连接设备。

## 使用命令行剖析

为了记录剖析数据，我们需要下载 simpleperf 和包含调试信息的本地库到设备上，运行 simpleperf 产生剖析数据：perf.data，并运行 simpleperf 为 perf.data 生成报告。步骤如下。

### 1. Enable profiling

```
1 | $adb shell setprop security.perf_harden 0
```

### 2. 寻找运行 app 的进程

在 app 的上下文运行 ps。在 >=0 的设备上，用 ps -e 来替代。

```
1 | $adb shell
2 | angler:/ $ run-as com.example.sudogame
3 | angler:/data/data/com.example.sudogame $ ps
4 | u0_a93      10324 570    1030480 58104 SyS_epoll_ 00f41b7528 S com.example.sudogame
5 | u0_a93      10447 10441 7716    1588 sigsuspend 753c515d34 S sh
6 | u0_a93      10453 10447 9112    1644          0 7ba07ff664 R ps
```

因此是进程 10324 运行app。

### 3. 将 simpleperf 下载到 app 的 data 目录

首先我们需要找出 app 使用的是哪个体系架构。有许多中方式，这里我们只检查进程的映射。

```
1 angler:/data/data/com.example.sudogame $cat /proc/10324/maps | grep boot.art
2 70f34000-7144e000 r--p 00000000 fd:00 1082 /system/framework/arm/boot.oat
```

路径显示它是 arm。因此我们将 simpleperf 下载到设备上的 arm 目录下。

```
1 $adb push bin/android/arm/simpleperf /data/local/tmp
2 $adb shell
3 angler:/ $ run-as com.example.sudogame
4 angler:/data/data/com.example.sudogame
5 $ cp /data/local/tmp/simpleperf .
```

### 4. 记录 perf.data

```
1 angler:/data/data/com.example.sudogame $./simpleperf record -p 10324 --duration 30
2 simpleperf I 01-01 09:26:39 10598 10598 cmd_record.cpp:341] Samples recorded: 49471. Samples lost: 0.
3 angler:/data/data/com.example.sudogame $ls -lh perf.data
4 -rw-rw-rw- 1 u0_a93 u0_a93 2.6M 2017-01-01 09:26 perf.data
```

在记录时不要忘记运行 app。否则，我们可能无法获得样本，由于进程仍在休眠。

### 5. 为 perf.data 生成报告

有不同的方式来为 perf.data 生成报告。下面展示了一些例子。

报告不同线程中的样本。

```
1 angler:/data/data/com.example.sudogame $./simpleperf report --sort pid,tid,comm
2 Cmdline: /data/data/com.example.sudogame/simpleperf record -p 10324 --duration 30
3 Arch: arm64
```

```
4 Event: cpu-cycles (type 0, config 0)
5 Samples: 49471
6 Event count: 16700769019
7 Overhead  Pid    Tid    Command
8 66.31%    10324  10324  xample.sudogame
9 30.97%    10324  10340  RenderThread
10 ...
11
```

报告主线程中不同二进制文件中的样本。

```
1 angler:/data/data/com.example.sudogame $./simpleperf report --tids 10324 --sort dso -n
2 ...
3 Overhead  Sample  Shared Object
4 37.71%    9970    /system/lib/libc.so
5 35.45%    9786    [kernel.kallsyms]
6 8.71%     3305    /system/lib/libart.so
7 6.44%     2405    /system/framework/arm/boot-framework.oat
8 5.64%     1480    /system/lib/libcutils.so
9 1.55%     426     /data/app/com.example.sudogame-1/lib/arm/libsudo-game-jni.so
10 ...
```

报告主线程中 libsudo-game-jni.so 中的不同函数的采样。

```
1 angler:/data/data/com.example.sudogame $./simpleperf report --tids 10324 --dsos /data/app/com.example.sudogame-1/
2 ...
3 Overhead  Sample  Symbol
4 8.94%     35     libsudo-game-jni.so[+1d54]
5 5.71%     25     libsudo-game-jni.so[+1dae]
6 5.70%     23     @plt
7 5.09%     22     libsudo-game-jni.so[+1d88]
8 4.54%     19     libsudo-game-jni.so[+1d82]
9 3.61%     14     libsudo-game-jni.so[+1f3c]
10 ...
```

在上面的结果中，大多数符号是 二进制文件名[+virtual\_addr] 的形式呈现。那是由于设备上使用的 libsudo-game-jni.so 已经抛离了 .symbol 段。我们可以将带有调试信息的 libsudo-game-jni.so 下载到设备上。在 android studio 工程中，它位于 app/build/intermediates/binaries/debug/arm/obj/armeabi-v7a/libsudo-game-jni.so。我们不得不下载 libsudo-game-jni.so 到与 perf.data 中记录的相同的相对路径（否则，simpleperf 无法找到它）。在这个例子中，是/data/app/com.example.sudogame-1/lib/arm/libsudo-game-jni.so。

为包含了调试信息的库使用的符号生成报告。

```

1 $adb push app/build/intermediates/binaries/debug/arm/obj/armeabi-v7a/libsudo-game-jni.so /data/local/tmp
2 $adb shell
3 angler:/ $ run-as com.example.sudogame
4 angler:/data/data/com.example.sudogame $ mkdir -p data/app/com.example.sudogame-1/lib/arm
5 angler:/data/data/com.example.sudogame $ cp /data/local/tmp/libsudo-game-jni.so data/app/com.example.sudogame-1/lib
6 angler:/data/data/com.example.sudogame $ ./simpleperf report --tids 10324 --dsos /data/app/com.example.sudogame-1/
7 ...
8 Overhead Sample Symbol
9 75.18% 317 checkValid(Board const&, int, int)
10 14.43% 60 canFindSolution_r(Board&, int, int)
11 5.70% 23 @plt
12 3.51% 20 randomBlock_r(Board&, int, int, int, int, int)
13 ...

```

报告一个函数中的采样。

```

1 angler:/data/data/com.example.sudogame $ ./simpleperf report --tids 10324 --dsos /data/app/com.example.sudogame-1/
2 ...
3 Overhead Sample VaddrInFile
4 11.89% 35 0x1d54
5 7.59% 25 0x1dae
6 6.77% 22 0x1d88
7 6.03% 19 0x1d82
8 ...

```

## 6. 记录并报告调用图

调用图是显示了函数调用关系的树。下面是一个例子。

```
1  main() {
2      FunctionOne();
3      FunctionTwo();
4  }
5  FunctionOne() {
6      FunctionTwo();
7      FunctionThree();
8  }
9  callgraph:
10     main-> FunctionOne
11         |
12         |-> FunctionTwo
13         |-> FunctionThree
14         |
15     -> FunctionTwo
```

### 基于调用图记录 dwarf

为了生成调用图，simpleperf 需要为每个记录生成调用链。Simpleperf 需要内核为每个记录转储用户栈和用户寄存器集，然后它追踪用户栈来查找函数调用链。为了解析调用链，它需要 dwarf 调用帧信息的支持，这些通常位于二进制文件的 .eh\_frame 或 .debug\_frame 段。因此我们需要使用 -symfs 指出带有调试信息的 libsudo-game-jni.so 位于哪里。

```
1  angler:/data/data/com.example.sudogame
2  $./simpleperf record -p 10324 -g --symfs . --duration 30
3  simpleperf I 01-01 09:59:42 11021 11021 cmd_record.cpp:341] Samples recorded: 60700. Samples lost: 1240.
```

注意内核无法转储  $\geq 64K$  的用户栈，因此基于调用图的 dwarf 不要包含消耗了  $\geq 64K$  栈的调用链。此外，由于我们需要转储每个记录的栈，则可能丢失一些记录。通常，失去一些记录没关系。

### 基于调用图记录栈帧



另外一种生成调用图的方式依赖内核为每个记录解析调用链。为了使它成为可能，内核需要能够识别每个函数调用的栈帧。这不总是可能的，因为编译器可能优化掉栈帧，或内核无法识别使用的栈帧风格。因此它如何工作视情况而定（它在 arm64 上工作的很好，但在 arm 上不行）。

```
1 angler:/data/data/com.example.sudogame
2 ./simpleperf record -p 10324 --call-graph fp --symfs . --duration 30
3 simpleperf I 01-01 10:03:58 11267 11267 cmd_record.cpp:341] Samples recorded: 56736. Samples lost: 0.
```

## 报告调用图

报告累积的周期。在下面的表中，第一列是“Children”，它是函数及那个函数调用的函数所占的cpu 周期百分比。第二列是“Self”，它只是一个函数的cpu 周期百分比。比如，checkValid() 自身消耗 1.28% 的 cpus，但通过运行它自身及调用其它函数，它消耗了 29.43%。

```
1 angler:/data/data/com.example.sudogame ./simpleperf report --children --symfs .
2 ...
3 Children Self Command Pid Tid Shared Object Symbol
4 31.94% 0.00% xample.sudogame 10324 10324 [kernel.kallsyms] [kern
5 31.10% 0.92% xample.sudogame 10324 10324 /system/lib/libc.so write
6 29.43% 1.28% xample.sudogame 10324 10324 /data/app/com.example.sudogame-1/lib/arm/libsudo-game-jni.so check
7 28.43% 0.34% xample.sudogame 10324 10324 /system/lib/liblog.so __and
8 28.24% 0.00% xample.sudogame 10324 10324 /system/lib/libcutils.so libcu
9 28.10% 0.27% xample.sudogame 10324 10324 /data/app/com.example.sudogame-1/lib/arm/libsudo-game-jni.so canF
10 ...
```

## 报告调用图。

```
1 angler:/data/data/com.example.sudogame ./simpleperf report -g --symfs . >report
2 angler:/data/data/com.example.sudogame $exit
3 angler:/ $cp /data/data/com.example.sudogame/report /data/local/tmp
4 angler:/ $exit
5 $adb pull /data/local/tmp/report .
6 $cat report
7 ...
8 29.43% 1.28% xample.sudogame 10324 10324 /data/app/com.example.sudogame-1/lib/arm/libsudo-game-jni.so check
9 |
10 -- checkValid(Board const&, int, int)
```

```

11 |
12 |      |--95.50%-- __android_log_print
13 |      |      |--0.68%-- [hit in function]
14 |      |
15 |      |--51.84%-- __android_log_buf_write
16 |      |      |--2.07%-- [hit in function]
17 |      |
18 |      |--30.74%-- libcutils.so[+c69d]
19 |      ...

```

以 callee 模式报告调用图。我们还可以展示一个函数是如何被其它函数调用的。

```

1  angler:/data/data/com.example.sudogame ./simpleperf report -g callee --symfs . >report
2  $adb shell run-as com.example.sudogame cat report >report
3  $cat report
4  ...
5  28.43%    0.34%  xample.sudogame  10324  10324  /system/lib/liblog.so
6  |
7  -- __android_log_print
8  |
9  |--97.82%-- checkValid(Board const&, int, int)
10 |      |--0.13%-- [hit in function]
11 |
12 |      |--94.89%-- canFindSolution_r(Board&, int, int)
13 |      |--0.01%-- [hit in function]
14 |
15 |      ...

```

## 剖析 Java 代码

Simpleperf 只支持剖析 ELF 格式二进制文件中的本地指令。如果 java 代码由解释器执行，或使用 jit 缓存，则它不能由 simpleperf 剖析。由于 Android 支持提前编译，它可以将 java 字节码编译为包含调试信息的本地层指令。在 Android 版本  $\leq M$  的设备上，我们需要 root 权限来编译包含调试信息的 java 字节码。然而，在 Android 版本  $\geq N$  的设备上，我们无需 root 权限就可以做这些。

## 在Android N上

### 1. 将 java 代码完整编译为本地层指令。

```
1 $adb shell setprop debug.generate-debug-info true
2 $adb shell cmd package compile -f -m speed com.example.sudogame
3 // restart the app to take effect
```

### 2. 记录 perf.data

```
1 angler:/data/data/com.example.sudogame
2 ./simpleperf record -p 11826 -g --symfs . --duration 30
3 simpleperf I 01-01 10:31:40 11859 11859 cmd_record.cpp:341] Samples recorded: 50576. Samples lost: 2139.
```

### 3. 为 perf.data 生成报告

```
1 angler:/data/data/com.example.sudogame ./simpleperf report -g --symfs . >report
2 angler:/data/data/com.example.sudogame $exit
3 angler:/ $cp /data/data/com.example.sudogame/report /data/local/tmp
4 angler:/ $exit
5 $adb pull /data/local/tmp/report .
6 $cat report
7 ...
8 21.14%    0.00%  xample.sudogame  11826  11826  /data/app/com.example.sudogame-1/oat/arm/base.odex      boole
9      |
10      -- boolean com.example.sudogame.MainActivity.onOptionsItemSelected(android.view.MenuItem)
11      |
12      --99.99%-- void com.example.sudogame.GameView.startNewGame()
13      |      --0.01%-- [hit in function]
14      |
15      |      --99.87%-- void com.example.sudogame.GameModel.reInit()
16      |      |      --0.01%-- [hit in function]
17      |      |
18      |      |      --89.65%-- boolean com.example.sudogame.GameModel.canFindSolution(int[][]))
19      |      |      |
20      |      |      |      --99.95%-- Java_com_example_sudogame_GameModel_canFindSolution
21      |      |      |      |
```

```
22 | | | | | --99.49%-- canFindSolution(Board&)
23 | | | | | --0.01%-- [hit in function]
24 |
25 | | | | | --99.97%-- canFindSolution_r(Board&, int, int)
26 | | | | | canFindSolution_r(Board&, int, int)
27 | ...
```

## 在 Android M上

在 M 设备上, 我们需要 root 权限来强制 Android 将 java 代码完全编译为带调试信息的 ELF 二进制文件中的本地层指令。我们还需要 root 权限来读取编译后的本地层二进制文件 (由于 `installld` 将它们写到了一个 uid/gid 是 `system:install` 的目录下)。因而剖析 java 代码只能在 root 了的设备上完成。

```
1 $adb root
2 $adb shell setprop dalvik.vm.dex2oat-flags -g
3 # Reinstall the app.
4 $adb install -r app-debug.apk
5 # Change to the app's data directory.
6 $ adb root && adb shell
7 device# cd `run-as com.example.sudogame pwd`
8 # Record as root as simpleperf needs to read the generated native binary.
9 device# ./simpleperf record -p 25636 -g --symfs . -f 1000 --duration 30
10 simpleperf I 01-02 07:18:20 27182 27182 cmd_record.cpp:323] Samples recorded: 23552. Samples lost: 39.
11
12
13
```

## 在 Android L上

在 L 设备上, 我们也需要 root 权限来编译带调试信息的 app 并访问本地层二进制文件。

```
1 $adb root
2 $adb shell setprop dalvik.vm.dex2oat-flags --include-debug-symbols
3 # Reinstall the app.
4 $adb install -r app-debug.apk
5
```

# 使用脚本剖析

尽管使用命令行很灵活，但它可能太复杂了。因而我们提供了 `python` 脚本来帮助运行命令。

## 使用 `app_profiler.py` 记录

`app_profiler.py` 用于剖析 Android 应用程序。它设置剖析环境，下载 `simpleperf` 和带有调试信息的本地层库，运行 `simpleperf` 产生 `perf.data`，并从设备上将 `perf.data` 和二进制文件拉到主机上。它由 `app_profiler.config` 配置。下面是一个例子。

`app_profiler.config`:

```
1  app_package_name = "com.example.sudogame"
2  android_studio_project_dir = "/AndroidStudioProjects/SudoGame" # absolute path of the project
3  ...
4  record_options = "-e cpu-cycles:u -f 4000 -g --dump-symbols --duration 30"
5  ...
```

运行 `app_profiler.py`:

```
1  $python app_profiler.py
2  ...
3  INFO:root:profiling is finished.
```

它将生成的 `perf.data` 拉到主机上，并从设备的 `binary_cache` 中收集二进制文件。

## 使用 `report.py` 生成报告

```
1  $python report.py -g
```

它生成一个GUI接口来报告数据。

## 使用 simpleperf\_report\_lib.py 处理样本

simpleperf\_report\_lib.py 提供了一个接口从 perf.data 读取样本。一个例子是 report\_sample.py。

### 展示流程图

```
1 $python report_sample.py >out.perf
2 $stackcollapse-perf.pl out.perf >out.folded
3 $./flamegraph.pl out.folded >a.svg
```

## 注解源代码

annotate.py 读取 perf.data 和 binary\_cache 下的二进制文件。然后它知道每个样本命中哪个 源文件:line。因此它可以注解源代码。annotate.py 由 annotate.config 配置。下面是一个例子。

annotate.config:

```
1 ...
2 source_dirs = ["/AndroidStudio/SudoGame"] # It is a directory containing source code.
3 ...
```

运行 annotate.py:

```
1 $python annotate.py
```

它生成 annotated\_files 目录。 annotated\_files/summary 文件包含每个源文件的概要信息。例子如下。

```
1 /AndroidStudioProjects/SudoGame/app/src/main/jni/sudo-game-jni.cpp: accumulated_period: 25.587937%, period: 1.25090
2 function (checkValid(Board const&, int, int)): line 99, accumulated_period: 23.564356%, period: 0.908457%
3 function (canFindSolution_r(Board&, int, int)): line 135, accumulated_period: 22.260125%, period: 0.142359%
4 function (canFindSolution(Board&)): line 166, accumulated_period: 22.233101%, period: 0.000000%
5 function (Java_com_example_sudogame_GameModel_canFindSolution): line 470, accumulated_period: 21.983184%, period:
```

```

6      function (Java_com_example_sudogame_GameModel_initRandomBoard): line 430, accumulated_period: 2.226896%, period:
7      line 27: accumulated_period: 0.011729%, period: 0.000000%
8      line 32: accumulated_period: 0.004362%, period: 0.000000%
9      line 33: accumulated_period: 0.004427%, period: 0.000000%
10     line 36: accumulated_period: 0.003303%, period: 0.000000%
11     line 39: accumulated_period: 0.010367%, period: 0.004123%
12     line 41: accumulated_period: 0.162219%, period: 0.000000%
13

```

annotated\_files/ 还包含由 annotate.py 找到的经过注解的源文件。比如，libsudo-game-jni.cpp 中的 checkValid() 函数的一部分注解后如下。

```

1      /* [func] acc_p: 23.564356%, p: 0.908457% */static bool checkValid(const Board& board, int curR, int curC) {
2      /* acc_p: 0.037933%, p: 0.037933%          */    int digit = board.digits[curR][curC];
3      /* acc_p: 0.162355%, p: 0.162355%          */    for (int r = 0; r < BOARD_ROWS; ++r) {
4      /* acc_p: 0.020880%, p: 0.020880%          */        if (r == curR) {
5      /* acc_p: 0.034691%, p: 0.034691%          */            continue;
6
7      /* acc_p: 0.176490%, p: 0.176490%          */        }
8      /* acc_p: 14.957673%, p: 0.059022%         */        if (board.digits[r][curC] == digit) {
9      /* acc_p: 0.016296%, p: 0.016296%         */            LOGI("conflict (%d, %d) (%d, %d)", curR, curC, r, curC);
10
11                                     return false;
12                                     }
13     }

```

原文地址

[Android开发](#) [翻译](#)

[Android Things介绍](#)

[Wireshark源码安装](#)

🔍 请输入关键字...

## 📁 分类

Android 图形系统 (16)

Android开发 (26)

C/C++开发 (3)

Java开发 (7)

Linux内核 (1)

live555 (13)

业界趣闻 (1)

后台开发 (3)

安全 (6)

网络协议 (39)

网络调试 (8)

随想杂谈 (3)

音视频开发 (3)

## 🏷 标签

OpenGL 网络协议 Android开发 源码分析 chromium 后台开发 Java开发 QUIC 翻译 网络调试 HTTP2 UDT 图形图像  
安全 HTTPS C/C++开发 音视频开发 Linux内核 live555 Django Go 语言

## 📁 归档



九月 2017  
八月 2017  
七月 2017  
六月 2017  
五月 2017  
四月 2017  
三月 2017  
二月 2017  
一月 2017  
十二月 2016  
十一月 2016  
十月 2016  
八月 2016  
七月 2016  
四月 2016  
九月 2015  
七月 2015  
九月 2013  
七月 2013  
五月 2013  
二月 2013

## 最新文章

Android 图形系统之gralloc

Android 图形系统之图形缓冲区分配

Android 硬件 OpenGL ES 模拟设计概述

Android 图形驱动 OpenGL ES 图形设计概述

EGL Context 创建

Android 图形驱动初始化

在 Android 中使用 OpenGL

使用 GDB 调试 Android 应用

Android 模拟器下载、编译及调试

live555 源码分析：播放启动

live555 源码分析：子会话 SETUP

### 友情链接

[Android Developers](#)

[Android Open Source Project](#)

[蚂蚁网](#)

### 最近评论

[网站地图](#) | [订阅本站](#) | 联系博主：[✉](#) [🐼](#) [🐱](#)

Copyright ©Han Pengfei    Powered by [Hexo](#).    Theme by [Charles](#).