ftrace 的作用是幫助開發人員瞭解Linux 內核的運行時行為，以便進行故障調試或性能分析。

最早ftrace 是一個function tracer，僅能夠記錄內核的函數調用流程。如今ftrace 已經成為一個framework，採用plugin 的方式支持開發人員添加更多種類的trace 功能。

Ftrace 由RedHat 的Steve Rostedt 負責維護。到2.6.30 為止，已經支持的tracer 包括：

Function tracer和Function graph tracer :跟蹤函數調用。

Schedule switch tracer :跟蹤進程調度情況。

Wakeup tracer：跟蹤進程的調度延遲，即高優先級進程從進入ready狀態到獲得CPU的延遲時間。該tracer只針對實時進程。

Irqsoff tracer：當中斷被禁止時，系統無法相應外部事件，比如鍵盤和鼠標，時鐘也無法產生tick中斷。這意味著系統響應延遲，irqsoff這個tracer能夠跟蹤並記錄內核中哪些函數禁止了中斷，對於其中中斷禁止時間最長的，irqsoff將在log文件的第一行標示出來，從而使開發人員可以迅速定位造成響應延遲的罪魁禍首。

Preemptoff tracer：和前一個tracer類似，preemptoff tracer跟蹤並記錄禁止內核搶佔的函數，並清晰地顯示出禁止搶佔時間最長的內核函數。

Preemptirqsoff tracer :同上，跟蹤和記錄禁止中斷或者禁止搶佔的內核函數，以及禁止時間最長的函數。

Branch tracer :跟蹤內核程序中的likely/unlikely分支預測命中率情況。Branch tracer能夠記錄這些分支語句有多少次預測成功。從而為優化程序提供線索。

Hardware branch tracer：利用處理器的分支跟蹤能力，實現硬件級別的指令跳轉記錄。在x86上，主要利用了BTS這個特性。

Initcall tracer：記錄系統在boot階段所調用的init call 。

Mmiotrace tracer：記錄memory map IO的相關信息。

Power tracer：記錄系統電源管理相關的信息。

Sysprof tracer：缺省情況下，sysprof tracer每隔1 msec對內核進行一次採樣，記錄函數調用和堆棧信息。

Kernel memory tracer :內存tracer主要用來跟蹤slab allocator的分配情況。包括kfree，kmem_cache_alloc等API的調用情況，用戶程序可以根據tracer收集到的信息分析內部碎片情況，找出內存分配最頻繁的代碼片斷，等等。

Workqueue statistical tracer：這是一個statistic tracer，統計系統中所有的workqueue的工作情況，比如有多少個work被插入workqueue，多少個已經被執行等。開發人員可以以此來決定具體的workqueue實現，比如是使用single threaded workqueue還是per cpu workqueue.

Event tracer :跟蹤系統事件，比如timer，系統調用，中斷等。

這裡還沒有列出所有的tracer，ftrace 是目前非常活躍的開發領域，新的tracer 將不斷被加入內核。

回頁首

ftrace 與其他trace 工具的關係和比較

Ftrace 最初是在2.6.27 中出現的，那個時候，systemTap 已經開始嶄露頭角，其他的trace 工具包括LTTng 等也已經發展多年。那為什麼人們還要再開發一個trace 工具呢？

SystemTap 項目是Linux 社區對SUN Dtrace 的反應，目標是達到甚至超越Dtrace 。因此SystemTap 設計比較複雜，Dtrace 作為SUN 公司的一個項目開發了多年才最終穩定發佈，況且得到了Solaris 內核中每個子系統開發人員的大力支持。SystemTap 想要趕超Dtrace，困難不僅是一樣，而且更大，因此她始終處在不斷完善自身的狀態下，在真正的產品環境，人們依然無法放心的使用她。不當的使用和SystemTap 自身的不完善都有可能導致系統崩潰。

Ftrace 的設計目標簡單，本質上是一種靜態代碼插裝技術，不需要支持某種編程接口讓用戶自定義trace 行為。靜態代碼插裝技術更加可靠，不會因為用戶的不當使用而導致內核崩潰。ftrace 代碼量很小，穩定可靠。實際上，即使是Dtrace，大多數用戶也只使用其靜態trace 功能。因此ftrace 的設計非常務實。

從2.6.30開始，ftrace支持event tracer，其實現和功能與LTTng非常類似，或許將來ftrace會同LTTng進一步融合，各自取長補短。ftrace有定義良好的ASCII接口，可以直接閱讀，這對於內核開發人員非常具有吸引力，因為只需內核代碼加上cat命令就可以工作了，相當方便； LTTng則採用binary接口，更利於專門工具分析使用。此外他們內部ring buffer的實現不相同，ftrace對所有tracer都採用同一個ring buffer，而LTTng則使用各自不同的ring buffer 。

目前，或許將來LTTng 都只能是內核主分支之外的工具。她主要受到嵌入式工程師的歡迎，而內核開發人員則更喜歡ftrace 。

Ftrace 的實現依賴於其他很多內核特性，比如tracepoint[3]，debugfs[2]，kprobe[4]，IRQ-Flags[5] 等。限於篇幅，關於這些技術的介紹請讀者自行查閱相關的參考資料。

回頁首

ftrace 的使用

ftrace 在內核態工作，用戶通過debugfs 接口來控制和使用ftrace 。從2.6.30 開始，ftrace 支持兩大類tracer：傳統tracer 和Non-Tracer Tracer 。下面將分別介紹他們的使用。

傳統Tracer 的使用

使用傳統的ftrace 需要如下幾個步驟：

- 選擇一種tracer
- 使能ftrace
- 執行需要trace 的應用程序，比如需要跟蹤ls，就執行ls
- 關閉ftrace
- 查看trace 文件

用戶通過讀寫debugfs 文件系統中的控制文件完成上述步驟。使用debugfs，首先要掛載她。命令如下：

```
# mkdir /debug
 # mount -t debugfs nodev /debug
```

此時您將在/debug 目錄下看到tracing 目錄。Ftrace 的控制接口就是該目錄下的文件。

選擇tracer 的控制文件叫作current_tracer 。選擇tracer 就是將tracer 的名字寫入這個文件，比如，用戶打算使用function tracer，可輸入如下命令：

```
#echo ftrace > /debug/tracing/current_tracer
```

文件tracing_enabled 控制ftrace 的開始和結束。

```
#echo 1 >/debug/tracing/tracing_enable
```

上面的命令使能ftrace 。同樣，將0 寫入tracing_enable 文件便可以停止ftrace 。

ftrace 的輸出信息主要保存在3 個文件中。

- Trace，該文件保存ftrace 的輸出信息，其內容可以直接閱讀。
- latency_trace，保存與trace 相同的信息，不過組織方式略有不同。主要為了用戶能方便地分析系統中有關延遲的信息。
- trace_pipe 是一個管道文件，主要為了方便應用程序讀取trace 內容。算是擴展接口吧。

下面詳細解析各種tracer 的輸出信息。

Function tracer 的輸出

Function tracer 跟蹤函數調用流程，其trace 文件格式如下：

```
# tracer: function
 #
 # TASK-PID CPU# TIMESTAMP FUNCTION
 # | | | | |
  bash-4251 [01] 10152.583854: path_put <-path_walk
  bash-4251 [01] 10152.583855: dput <-path_put
  bash-4251 [01] 10152.583855: _atomic_dec_and_lock <-dput
```

可以看到，tracer 文件類似一張報表，前4 行是表頭。第一行顯示當前tracer 的類型。第三行是header 。

對於function tracer，該表將顯示4 列信息。首先是進程信息，包括進程名和PID ；第二列是CPU，在SMP 體系下，該列顯示內核函數具體在哪一個CPU 上執行；第三列是時間戳；第四列是函數信息，缺省情況下，這裡將顯示內核函數名以及它的上一層調用函數。

通過對這張報表的解讀，用戶便可以獲得完整的內核運行時流程。這對於理解內核代碼也有很大的幫助。有志於精讀內核代碼的讀者，或許可以考慮考慮ftrace 。

如上例所示，path_walk() 調用了path_put 。此後path_put 又調用了dput，進而dput 再調用 _atomic_dec_and_lock 。

Schedule switch tracer 的輸出

Schedule switch tracer 記錄系統中的進程切換信息。在其輸出文件trace 中, 輸出行的格式有兩種：

第一種表示進程切換信息：

```
Context switches:
       Previous task Next Task
  <pid>:<prio>:<state> ==> <pid>:<prio>:<state>
```

第二種表示進程wakeup 的信息：

```
        Wake ups:
       Current task Task waking up
  <pid>:<prio>:<state> + <pid>:<prio>:<state>
```

這裡舉一個實例：

```
# tracer: sched_switch
 #
 # TASK_PID CPU# TIMESTAMP FUNCTION
 # | | | |
   fon-6263 [000] 4154504638.932214: 6263:120:R + 2717:120:S
   fon-6263 [000] 4154504638.932214: 6263:120:? ==> 2717:120:R
   bash-2717 [000] 4154504638.932214: 2717:120:S + 2714:120:S
```

第一行表示進程fon進程wakeup了bash進程。其中fon進程的pid為6263，優先級為120，進程狀態為Ready。她將進程ID為2717的bash進程喚醒。

第二行表示進程切換發生，從fon切換到bash 。

irqsoff tracer 輸出

有四個tracer記錄內核在某種狀態下最長的時延，irqsoff記錄系統在哪裡關中斷的時間最長；preemptoff/preemptirqsoff以及wakeup分別記錄禁止搶佔時間最長的函數，或者係統在哪裡調度延遲最長(wakeup) 。這些tracer信息對於實時應用具有很高的參考價值。

為了更好的表示延遲，ftrace 提供了和trace 類似的latency_trace 文件。以irqsoff 為例演示如何解讀該文件的內容。

```
# tracer: irqsoff
 irqsoff latency trace v1.1.5 on 2.6.26
 -------------------------------------------------- ------------------
 latency: 12 us, #3/3, CPU#1 | (M:preempt VP:0, KP:0, SP:0 HP:0 #P:2)
    -----------------
    | task: bash-3730 (uid:0 nice:0 policy:0 rt_prio:0)
    -----------------
 => started at: sys_setpgid
 => ended at: sys_setpgid
 # _------=> CPU#
 # / _-----=> irqs-off
 # | / _----=> need-resched
```

```
# || / _---=> hardirq/ softirq
# ||| / _--=> preempt-depth
# |||| /
# ||||| delay
# cmd pid ||||| time | caller
# \ / ||||| \ | /
   bash-3730    1d...      0us : _write_lock_irq (sys_setpgid)
   bash-3730 1d..1 1us+: _write_unlock_irq (sys_setpgid)
   bash-3730 1d..2 14us : trace_hardirqs_on (sys_setpgid)
```

在文件的頭部，irqsoff tracer 記錄了中斷禁止時間最長的函數。在本例中，函數trace_hardirqs_on 將中斷禁止了12us 。

文件中的每一行代表一次函數調用。Cmd 代表進程名，pid 是進程ID 。中間有5 個字符，分別代表了CPU#，irqs-off 等信息，具體含義如下：

CPU# 表示CPU ID ；

irqs-off這個字符的含義如下：' d '表示中斷被disabled 。' . '表示中斷沒有關閉；

need-resched字符的含義：' N '表示need_resched被設置，' . '表示need-reched沒有被設置，中斷返回不會進行進程切換；

hardirq/softirq 字符的含義: 'H' 在softirq 中發生了硬件中斷， 'h' – 硬件中斷，' s '表示softirq，' . '不在中斷上下文中，普通狀態。

preempt-depth: 當搶佔中斷使能後，該域代表preempt_disabled的級別。

在每一行的中間，還有兩個域：time和delay 。time:表示從trace開始到當前的相對時間。Delay突出顯示那些有高延遲的地方以便引起用戶注意。當其顯示為!時，表示需要引起注意。

function graph tracer 輸出

Function graph tracer 和function tracer 類似，但輸出為函數調用圖，更加容易閱讀：

```
# tracer: function_graph
 #
 # CPU OVERHEAD/DURATION FUNCTION CALLS
 # | | | | | | |
 0) | sys_open() {
```

```
0) | do_sys_open() {
0) | getname() {
0) | kmem_cache_alloc() {
0) 1.382 us | __might_sleep();
0) 2.478 us | }
0) | strncpy_from_user() {
0) | might_fault() {
0) 1.389 us | __might_sleep();
0) 2.553 us | }
0) 3.807 us | }
0) 7.876 us | }
0) | alloc_fd() {
0) 0.668 us | _spin_lock();
0) 0.570 us | expand_files();
0) 0.586 us | _spin_unlock();
```

OVERHEAD 為! 時提醒用戶注意，該函數的性能比較差。上面的例子中可以看到sys_open 調用了do_sys_open，依次又調用了getname()，依此類推。

Sysprof tracer 的輸出

Sysprof tracer 定時對內核進行採樣，她的輸出文件中記錄了每次採樣時內核正在執行哪些內核函數，以及當時的內核堆棧情況。

每一行前半部分的格式和3.1.1 中介紹的function tracer 一樣，只是，最後一部分FUNCTION 有所不同。

Sysprof tracer 中，FUNCTION 列格式如下：

Identifier address frame_pointer/pid

當identifier 為0 時，代表一次採樣的開始，此時第三個數字代表當前進程的PID ；

Identifier 為1 代表內核態的堆棧信息；當identifier 為2 時，代表用戶態堆棧信息；顯示堆棧信息時，第三列顯示的是frame_pointer，用戶可能需要打開system map 文件查找具體的符號，這是ftrace有待改進的一個地方吧。

當identifier 為3 時，代表一次採樣結束。

Non-Tracer Tracer 的使用

從2.6.30 開始，ftrace 還支持幾種Non-tracer tracer，所謂Non-tracer tracer 主要包括以下幾種：

- Max Stack Tracer
- Profiling (branches / unlikely / likely / Functions)
- Event tracing

和傳統的tracer 不同，Non-Tracer Tracer 並不對每個內核函數進行跟蹤，而是一種類似邏輯分析儀的模式，即對系統進行採樣，但似乎也不完全如此。無論怎樣，這些tracer 的使用方法和前面所介紹的tracer 的使用稍有不同。下面我將試圖描述這些tracer 的使用方法。

Max Stack Tracer 的使用

這個tracer 記錄內核函數的堆棧使用情況，用戶可以使用如下命令打開該tracer：

```
# echo 1 > /proc/sys/kernel/stack_tracer_enabled
```

從此，ftrace 便留心記錄內核函數的堆棧使用。Max Stack Tracer 的輸出在stack_trace 文件中：

```
# cat /debug/tracing/stack_trace
 Depth Size Location (44 entries)
 ----- ---- --------
 0) 3088 64 update_curr+0x64/0x136
 1) 3024 64 enqueue_task_fair+0x59/0x2a1
 2) 2960 32 enqueue_task+0x60/0x6b
 3) 2928 32 activate_task+0x27/0x30
 4) 2896 80 try_to_wake_up+0x186/0x27f
...
 42) 80 80 sysenter_do_call+0x12/0x32
```

從上例中可以看到內核堆棧最滿的情況如下，有43層函數調用，堆棧使用大小為3088字節。此外還可以在Location這列中看到整個的calling stack情況。這在某些情況 下，可以提供額外的debug信息，幫助開發人員定位問題。

Branch tracer

Branch tracer 比較特殊，她有兩種模式，即是傳統tracer，又實現了profiling tracer 模式。

作為傳統tracer 。其輸出文件為trace，格式如下：

```
# tracer: branch
 #
 # TASK-PID CPU# TIMESTAMP FUNCTION
 # | | | | |
  Xorg-2491 [000] 688.561593: [ ok ] fput_light:file.h:29
  Xorg-2491 [000] 688.561594: [ ok ] fput_light:file_table.c:330
```

在FUNCTION 列中，顯示了4 類信息：

函數名，文件和行號，用中括號引起來的部分，顯示了分支的信息，假如該字符串為ok，表明likely/unlikely返回為真，否則字符串為MISS 。舉例來說，在文件file.h 的第29 行，函數fput_light 中，有一個likely 分支在運行時解析為真。我們看看file.h 的第29 行：

```
static inline void fput_light(struct file *file, int fput_needed)
 { LINE29： if (unlikely(fput_needed))
                    fput(file);
 }
```

Trace結果告訴我們，在688秒的時候，第29行代碼被執行，且預測結果為ok，即unlikely成功。

Branch tracer 作為profiling tracer 時，其輸出文件為profile_annotated_branch，其中記錄了likely/unlikely語句完整的統計結果。

```
#cat trace_stat/branch_ annotated
 correct incorrect % function file line
 ------- ---------- ---- ------------------ ----------- --- -----
 0 46 100 pre_schedule_rt sched_rt.c 1449
```

下面是文件sched_rt.c 的第1449 行的代碼：

```
        if (unlikely(rt_task(prev)) && rq->rt.highest_prio.curr > prev->prio)
    pull_rt_task(rq);
```

記錄表明，unlikely 在這裡有46 次為假，命中率為100% 。假如為真的次數更多，則說明這裡應該改成

likely 。

Workqueue profiling

假如您在內核編譯時選中該tracer，ftrace 便會統計workqueue 使用情況。您只需使用下面的命令查看結果即可：

#cat /debug/tracing/trace_stat/workqueue

典型輸出如下：

```
# CPU INSERTED EXECUTED NAME
 # | | | |
   0 38044 38044 events/0
    0 426 426 khelper
   0 9853 9853 kblockd/0
   0 0 0 kacpid
...
```

可以看到workqueue events 在CPU 0 上有38044 個worker 被插入並執行。

Event tracer

Event tracer 不間斷地記錄內核中的重要事件。用戶可以用下面的命令查看ftrace 支持的事件。

#cat /debug/tracing/available_event

下面以跟蹤進程切換為例講述event tracer 的使用。首先打開event tracer，並記錄進程切換：

```
# echo sched:sched_switch >> /debug/tracing/set_event
 # echo sched_switch >> /debug/tracing/set_event
 # echo 1 > /debug/tracing/events/sched/sched_switch/enable
```

上面三個命令的作用是一樣的，您可以任選一種。

此時可以查看ftrace 的輸出文件trace:

>head trace

```
# tracer: nop
#
# TASK-PID CPU# TIMESTAMP FUNCTION
# | | | | |
<idle>-0 [000] 12093.091053: sched_switch: task swapper:0 [140] ==>
 /user/bin/sealer:2612 [120]
```

我想您會發現該文件很容易解讀。如上例，表示一個進程切換event，從idle進程切換到sealer進程。

回頁首

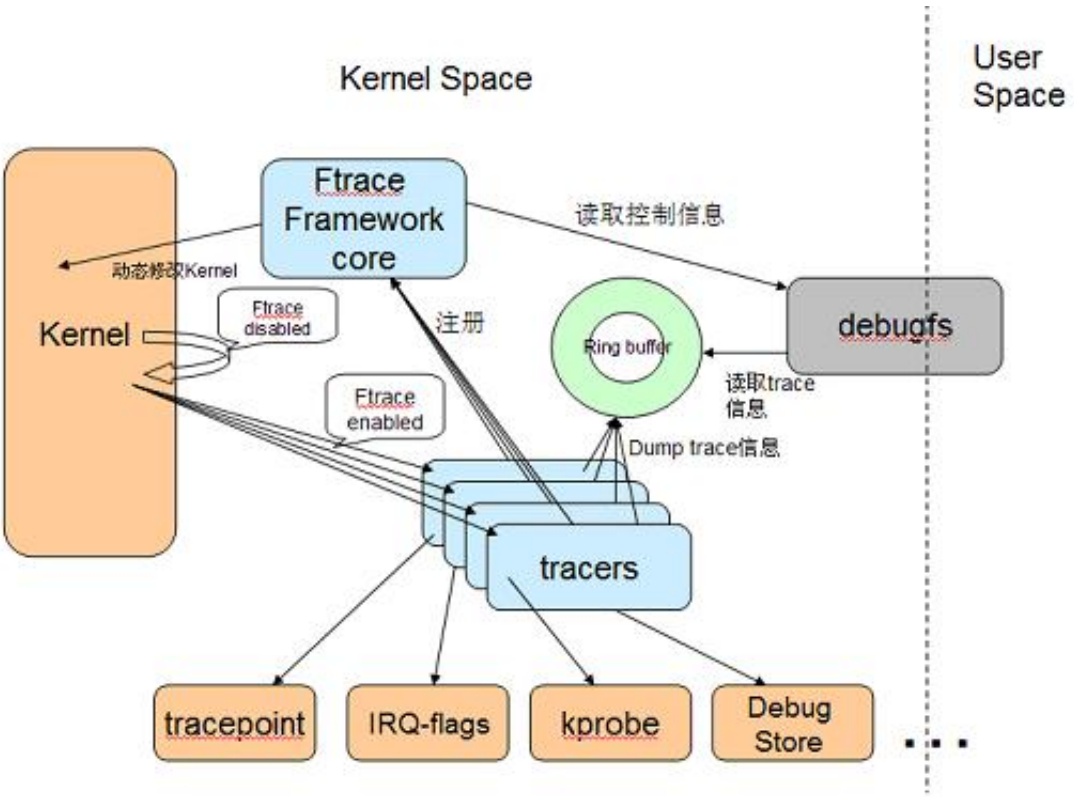
ftrace 的實現

研究tracer 的實現是非常有樂趣的。理解ftrace 的實現能夠啟發我們在自己的系統中設計更好的trace 功能。

ftrace 的整體構架

Ftrace 的整體構架：

圖1. ftrace 組成



Ftrace 有兩大組成部分，一是framework，另外就是一系列的tracer 。每個tracer 完成不同的功能，它們統一由framework 管理。ftrace 的trace 信息保存在ring buffer 中，由framework 負責管理。Framework 利用debugfs 系統在/debugfs 下建立tracing 目錄，並提供了一系列的控制文件。

本文並不打算系統介紹tracer 和ftrace framework 之間的接口，只是打算從純粹理論的角度，簡單剖析幾種具體tracer 的實現原理。假如讀者需要開發新的tracer，可以參考某個tracer 的源代碼。

Function tracer 的實現

Ftrace 採用GCC 的profile 特性在所有內核函數的開始部分加入一段stub 代碼，ftrace 重載這段代碼來實現trace 功能。

gcc的-pg選項將在每個函數入口處加入對mcount的調用代碼。比如下面的C代碼。

清單1. test.c

```
//test.c
 void foo(void)
 {
   printf( 「 foo 」 );
 }
```

用gcc 編譯：

```
gcc - S test.c
```

反彙編如下：

清單2. test.c 不加入pg 選項的彙編代碼

```
        _foo:
        pushl %ebp
        movl %esp, %ebp
        subl $8, %esp
        movl $LC0, (%esp)
        call _printf
        leave
        ret
```

再加入-gp 選項編譯：

```
gcc - pg - S test.c
```

得到的彙編如下：

清單3. test.c 加入pg 選項後的彙編代碼

```
_foo:
        pushl %ebp
        movl %esp, %ebp
        subl $8, %esp
 LP3:
        movl $LP3,%edx
        call _mcount
        movl $LC0, (%esp)
        call _printf
        leave
        ret
```

增加pg選項後，gcc在函數foo的入口處加入了對mcount的調用： call _mcount 。原本mcount由libc實現，但您知道內核不會連接libc庫，因此ftrace編寫了自己的mcount stub函數，並藉此實現trace功能。

在每個內核函數入口加入trace 代碼，必然會影響內核的性能，為了減小對內核性能的影響，ftrace 支持動態trace 功能。

當CONFIG_DYNAMIC_FTRACE 被選中後，內核編譯時會調用一個perl 腳本：recordmcount.pl 將每個函數的地址寫入一個特殊的段：__mcount_loc

在內核初始化的初期，ftrace 查詢__mcount_loc 段，得到每個函數的入口地址，並將mcount 替換為nop 指令。這樣在默認情況下，ftrace 不會對內核性能產生影響。

當用戶打開ftrace功能時，ftrace將這些nop指令動態替換為ftrace_caller，該函數將調用用戶註冊的trace函數。其具體的實現在相應arch的彙編代碼中，以x86為例，在entry_32.s中：

清單4. entry_32.s

```
        ENTRY(ftrace_caller)
        cmpl $0, function_trace_stop
        jne ftrace_stub
        pushl %eax
        pushl %ecx
        pushl %edx
        movl 0xc(%esp), %eax
        movl 0x4(%ebp), %edx
        subl $MCOUNT_INSN_SIZE, %eax
.globl ftrace_call
ftrace_call:
        call ftrace_stub line 10 popl %edx
        popl %ecx
        popl %eax

.globl ftrace_stub
ftrace_stub:
        ret
END(ftrace_caller)
```

Function tracer 將line10 這行代碼替換為function_trace_call() 。這樣每個內核函數都將調用function_trace_call() 。

在function_trace_call() 函數內，ftrace 記錄函數調用堆棧信息，並將結果寫入ring buffer，稍後，用戶可以通過debugfs 的trace 文件讀取該ring buffer 中的內容。

Irqsoff tracer 的實現

Irqsoff tracer的實現依賴於IRQ-Flags 。 IRQ-Flags是Ingo Molnar維護的一個內核特性。使得用戶能夠在中斷關閉和打開時得到通知， ftrace重載了其通知函數，從而能夠記錄中斷禁止時間。即，中斷被關閉時，記錄下當時的時間戳。此後，中斷被打開時，再計算時間差，由此便可得到中斷禁止時間。

IRQ-Flags 封裝開關中斷的宏定義：

清單5. IRQ-Flags 中斷代碼

```
#define local_irq_enable() \
```

```
do { trace_hardirqs_on (); raw_local_irq_enable(); } while (0)
```

ftrace 在文件ftrace_irqsoff.c 中重載了trace_hardirqs_on 。具體代碼不再羅列，主要是使用了sched_clock ( ) 函數來獲得時間戳。

hw-branch 的實現

Hw-branch 只在IA 處理器上實現，依賴於x86 的BTS 功能。BTS 將CPU 實際執行到的分支指令的相關信息保存下來，即每個分支指令的源地址和目標地址。

軟件可以指定一塊buffer，處理器將每個分支指令的執行情況寫入這塊buffer，之後，軟件便可以分析這塊buffer 中的功能。

Linux 內核的DS 模塊封裝了x86 的BTS 功能。Debug Support 模塊封裝了和底層硬件的接口，主要支持兩種功能：Branch trace store(BTS) 和precise-event based sampling (PEBS) 。ftrace 主要使用了BTS 功能。

branch tracer 的實現

內核代碼中常使用likely和unlikely提高編譯器生成的代碼質量。 Gcc可以通過合理安排彙編代碼最大限度的利用處理器的流水線。合理的預測是likely能夠提高性能的關鍵，ftrace為此定義了branch tracer，跟蹤程序中likely預測的正確率。

為了實現branch tracer，重新定義了likely 和unlikely 。具體的代碼在compiler.h 中。

清單6. likely/unlikely 的trace 實現

```
# ifndef likely
 # define likely(x) (__builtin_constant_p(x) ? !!(x) : __branch_check__(x, 1))
 # endif
 # ifndef unlikely
 # define unlikely(x) (__builtin_constant_p(x) ? !!(x) : __branch_check__(x, 0))
 # endif
```

其中__branch_check 的實現如下：

清單7. _branch_check_ 的實現

```
#define __branch_check__(x, expect) ({\
```

```
    int ______r; \
    static struct ftrace_branch_data \
    __attribute__((__aligned__(4))) \
    __attribute__((section("_ftrace_annotated_branch"))) \
                    ______f = { \
                      .func = __func__, \
                      .file = __FILE__, \
                      .line = __LINE__, \
                    }; \
            ______r = likely_notrace(x);\
            ftrace_likely_update(&______f, ______r, expect); \
            ______r; \
 })
```

ftrace_likely_update() 將記錄likely 判斷的正確性，並將結果保存在ring buffer 中，之後用戶可以通過ftrace 的debugfs 接口讀取分支預測的相關信息。從而調整程序代碼，優化性能。

回頁首

總結

本文講解了ftrace 的基本使用。在實踐中，ftrace 是一個非常有效的性能調優和debug 分析工具，每個人使用她的方法和角度都不相同，一定有很多best practice，這非本文所能涉及。但希望通過本文的講解，能讓讀者對ftrace 形成一個基本的瞭解，進而在具體工作中使用她。

參考資料

- [1]參看LWN上的文章a look at ftrace，該文介紹了ftrace的簡單概念。
- [2]參看文章lwn上關於debugfs的介紹文章瞭解debugFS。
- [3]參看文章內核文檔tracepoint.txt瞭解tracepoint。
- [4]參看IBM developerworks文章使用Kprobes調試內核瞭解Kprobes。

- [5]參看文章內核文檔IRQ-Flags瞭解IRQ-Flags。

- [6]參看文章內核文檔ftrace.txt瞭解ftrace更詳細的信息。

- 在developerWorks Linux專區尋找為Linux開發人員（包括 Linux新手入門）準備的更多參考資料，查閱我們 最受歡迎的文章和教程。