使用 DTrace 和 SystemTap 检测 CPython

发布 *3.8.2rc2*

Guido van Rossum
and the Python development team

二月 25, 2020

Python Software Foundation
Email: docs@python.org

Contents

作者 David Malcolm

作者 Łukasz Langa

DTrace 和 SystemTap 是监视工具，每个工具都提供了一种检查计算机系统上的进程正在执行的操作的方法。它们都使用特定于域的语言，允许用户编写以下脚本：

- 进程监视的过滤器
- 从感兴趣的进程中收集数据
- 生成有关数据的报告

从 Python 3.6 开始，CPython 可以使用嵌入式 “标记” 构建，也称为 “探测器”，可以通过 DTrace 或 SystemTap 脚本观察，从而更容易监视系统上的 CPython 进程正在做什么。

CPython implementation detail: DTrace 标记是 CPython 解释器的实现细节。不保证 CPython 版本之间的探针兼容性。更改 CPython 版本时，DTrace 脚本可能会停止工作或无法正常工作而不会发出警告。

1 启用静态标记

macOS 内置了对 DTrace 的支持。在 Linux 上，为了使用 SystemTap 的嵌入式标记构建 CPython，必须安装 SystemTap 开发工具。

在 Linux 机器上，这可以通过：

```
$ yum install systemtap-sdt-devel
```

或者：

```
$ sudo apt-get install systemtap-sdt-dev
```

然后必须将 CPython 配置为 "--with-dtrace"：

```
checking for --with-dtrace... yes
```

在 macOS 上，您可以通过在后台运行 Python 进程列出可用的 DTrace 探测器，并列出 Python 程序提供的所有探测器：

```
$ python3.6 -q &
$ sudo dtrace -l -P python$!  # or: dtrace -l -m python3.6

   ID   PROVIDER            MODULE                          FUNCTION NAME
29564 python18035        python3.6          _PyEval_EvalFrameDefault function-entry
29565 python18035        python3.6             dtrace_function_entry function-entry
29566 python18035        python3.6          _PyEval_EvalFrameDefault function-return
29567 python18035        python3.6            dtrace_function_return function-return
29568 python18035        python3.6                           collect gc-done
29569 python18035        python3.6                           collect gc-start
29570 python18035        python3.6          _PyEval_EvalFrameDefault line
29571 python18035        python3.6                 maybe_dtrace_line line
```

在 Linux 上，您可以通过查看是否包含 ".note.stapsdt" 部分来验证构建的二进制文件中是否存在 SystemTap 静态标记。

```
$ readelf -S ./python | grep .note.stapsdt
[30] .note.stapsdt        NOTE         0000000000000000 00308d78
```

如果您已将 Python 构建为共享库（使用--enable-shared），则需要在共享库中查找。例如：

```
$ readelf -S libpython3.3dm.so.1.0 | grep .note.stapsdt
[29] .note.stapsdt        NOTE         0000000000000000 00365b68
```

足够现代的 readelf 命令可以打印元数据：

```
$ readelf -n ./python

Displaying notes found at file offset 0x00000254 with length 0x00000020:
    Owner                 Data size          Description
    GNU                  0x00000010          NT_GNU_ABI_TAG (ABI version tag)
        OS: Linux, ABI: 2.6.32

Displaying notes found at file offset 0x00000274 with length 0x00000024:
    Owner                 Data size          Description
    GNU                  0x00000014          NT_GNU_BUILD_ID (unique build ID␣
↪bitstring)
```

(下页继续)

(续上页)

```
        Build ID: df924a2b08a7e89f6e11251d4602022977af2670

Displaying notes found at file offset 0x002d6c30 with length 0x00000144:
    Owner                 Data size          Description
    stapsdt               0x00000031         NT_STAPSDT (SystemTap probe descriptors)
        Provider: python
        Name: gc__start
        Location: 0x00000000004371c3, Base: 0x0000000000630ce2, Semaphore:␣
↪0x00000000008d6bf6
        Arguments: -4@%ebx
    stapsdt               0x00000030         NT_STAPSDT (SystemTap probe descriptors)
        Provider: python
        Name: gc__done
        Location: 0x00000000004374e1, Base: 0x0000000000630ce2, Semaphore:␣
↪0x00000000008d6bf8
        Arguments: -8@%rax
    stapsdt               0x00000045         NT_STAPSDT (SystemTap probe descriptors)
        Provider: python
        Name: function__entry
        Location: 0x000000000053db6c, Base: 0x0000000000630ce2, Semaphore:␣
↪0x00000000008d6be8
        Arguments: 8@%rbp 8@%r12 -4@%eax
    stapsdt               0x00000046         NT_STAPSDT (SystemTap probe descriptors)
        Provider: python
        Name: function__return
        Location: 0x000000000053dba8, Base: 0x0000000000630ce2, Semaphore:␣
↪0x00000000008d6bea
        Arguments: 8@%rbp 8@%r12 -4@%eax
```

上述元数据包含 SystemTap 的信息，描述如何修补策略性放置的机器代码指令以启用 SystemTap 脚本使用的跟踪钩子。

2 静态 DTrace 探针

The following example DTrace script can be used to show the call/return hierarchy of a Python script, only tracing within the invocation of a function called ”start”. In other words, import-time function invocations are not going to be listed:

```
self int indent;

python$target:::function-entry
/copyinstr(arg1) == "start"/
{
        self->trace = 1;
}

python$target:::function-entry
/self->trace/
{
        printf("%d\t%*s:", timestamp, 15, probename);
        printf("%*s", self->indent, "");
        printf("%s:%s:%d\n", basename(copyinstr(arg0)), copyinstr(arg1), arg2);
        self->indent++;
}
```

(下页继续)

(续上页)

```
python$target:::function-return
/self->trace/
{
        self->indent--;
        printf("%d\t%*s:", timestamp, 15, probename);
        printf("%*s", self->indent, "");
        printf("%s:%s:%d\n", basename(copyinstr(arg0)), copyinstr(arg1), arg2);
}

python$target:::function-return
/copyinstr(arg1) == "start"/
{
        self->trace = 0;
}
```

It can be invoked like this:

```
$ sudo dtrace -q -s call_stack.d -c "python3.6 script.py"
```

输出结果会像这样:

```
156641360502280  function-entry:call_stack.py:start:23
156641360518804  function-entry: call_stack.py:function_1:1
156641360532797  function-entry:  call_stack.py:function_3:9
156641360546807 function-return:  call_stack.py:function_3:10
156641360563367 function-return: call_stack.py:function_1:2
156641360578365  function-entry: call_stack.py:function_2:5
156641360591757  function-entry:  call_stack.py:function_1:1
156641360605556  function-entry:   call_stack.py:function_3:9
156641360617482 function-return:   call_stack.py:function_3:10
156641360629814 function-return:  call_stack.py:function_1:2
156641360642285 function-return: call_stack.py:function_2:6
156641360656770  function-entry: call_stack.py:function_3:9
156641360669707 function-return: call_stack.py:function_3:10
156641360687853  function-entry: call_stack.py:function_4:13
156641360700719 function-return: call_stack.py:function_4:14
156641360719640  function-entry: call_stack.py:function_5:18
156641360732567 function-return: call_stack.py:function_5:21
156641360747370 function-return:call_stack.py:start:28
```

3 Static SystemTap markers

The low-level way to use the SystemTap integration is to use the static markers directly. This requires you to explicitly state the binary file containing them.

For example, this SystemTap script can be used to show the call/return hierarchy of a Python script:

```
probe process("python").mark("function__entry") {
     filename = user_string($arg1);
     funcname = user_string($arg2);
     lineno = $arg3;

     printf("%s => %s in %s:%d\\n",
            thread_indent(1), funcname, filename, lineno);
```

(下页继续)

(续上页)

```
}

probe process("python").mark("function__return") {
    filename = user_string($arg1);
    funcname = user_string($arg2);
    lineno = $arg3;

    printf("%s <= %s in %s:%d\\n",
           thread_indent(-1), funcname, filename, lineno);
}
```

It can be invoked like this:

```
$ stap \
  show-call-hierarchy.stp \
  -c "./python test.py"
```

输出结果会像这样:

```
11408 python(8274):         => __contains__ in Lib/_abcoll.py:362
11414 python(8274):          => __getitem__ in Lib/os.py:425
11418 python(8274):           => encode in Lib/os.py:490
11424 python(8274):           <= encode in Lib/os.py:493
11428 python(8274):          <= __getitem__ in Lib/os.py:426
11433 python(8274):         <= __contains__ in Lib/_abcoll.py:366
```

where the columns are:

- time in microseconds since start of script
- name of executable
- PID of process

and the remainder indicates the call/return hierarchy as the script executes.

For a *--enable-shared* build of CPython, the markers are contained within the libpython shared library, and the probe's dotted path needs to reflect this. For example, this line from the above example:

```
probe process("python").mark("function__entry") {
```

should instead read:

```
probe process("python").library("libpython3.6dm.so.1.0").mark("function__entry") {
```

(assuming a debug build of CPython 3.6)

4 Available static markers

function__entry (str *filename*, str *funcname*, int *lineno*)

This marker indicates that execution of a Python function has begun. It is only triggered for pure-Python (bytecode) functions.

The filename, function name, and line number are provided back to the tracing script as positional arguments, which must be accessed using `$arg1`, `$arg2`, `$arg3`:

- `$arg1` : `(const char *)` filename, accessible using `user_string($arg1)`

- `$arg2 : (const char *)` function name, accessible using `user_string($arg2)`
- `$arg3 : int` line number

function__return (str *filename*, str *funcname*, int *lineno*)
This marker is the converse of *function__entry()*, and indicates that execution of a Python function has ended (either via `return`, or via an exception). It is only triggered for pure-Python (bytecode) functions.

The arguments are the same as for *function__entry()*

line (str *filename*, str *funcname*, int *lineno*)
This marker indicates a Python line is about to be executed. It is the equivalent of line-by-line tracing with a Python profiler. It is not triggered within C functions.

The arguments are the same as for *function__entry()*.

gc__start (int *generation*)
Fires when the Python interpreter starts a garbage collection cycle. `arg0` is the generation to scan, like `gc.collect()`.

gc__done (long *collected*)
Fires when the Python interpreter finishes a garbage collection cycle. `arg0` is the number of collected objects.

import__find__load__start (str *modulename*)
Fires before `importlib` attempts to find and load the module. `arg0` is the module name.

3.7 新版功能.

import__find__load__done (str *modulename*, int *found*)
Fires after `importlib`'s find_and_load function is called. `arg0` is the module name, `arg1` indicates if module was successfully loaded.

3.7 新版功能.

audit (str *event*, void **tuple*)
Fires when `sys.audit()` or `PySys_Audit()` is called. `arg0` is the event name as C string, `arg1` is a `PyObject` pointer to a tuple object.

3.8 新版功能.

5 SystemTap Tapsets

The higher-level way to use the SystemTap integration is to use a "tapset": SystemTap's equivalent of a library, which hides some of the lower-level details of the static markers.

Here is a tapset file, based on a non-shared build of CPython:

```
/*
   Provide a higher-level wrapping around the function__entry and
   function__return markers:
 \*/
probe python.function.entry = process("python").mark("function__entry")
{
    filename = user_string($arg1);
    funcname = user_string($arg2);
    lineno = $arg3;
    frameptr = $arg4
}
probe python.function.return = process("python").mark("function__return")
{
```

(下页继续)

(续上页)

```
    filename = user_string($arg1);
    funcname = user_string($arg2);
    lineno = $arg3;
    frameptr = $arg4
}
```

If this file is installed in SystemTap's tapset directory (e.g. `/usr/share/systemtap/tapset`), then these additional probepoints become available:

python.function.entry (str *filename*, str *funcname*, int *lineno*, frameptr)
This probe point indicates that execution of a Python function has begun. It is only triggered for pure-Python (bytecode) functions.

python.function.return (str *filename*, str *funcname*, int *lineno*, frameptr)
This probe point is the converse of *python.function.return()*, and indicates that execution of a Python function has ended (either via `return`, or via an exception). It is only triggered for pure-Python (bytecode) functions.

6 示例

This SystemTap script uses the tapset above to more cleanly implement the example given above of tracing the Python function-call hierarchy, without needing to directly name the static markers:

```
probe python.function.entry
{
  printf("%s => %s in %s:%d\n",
         thread_indent(1), funcname, filename, lineno);
}

probe python.function.return
{
  printf("%s <= %s in %s:%d\n",
         thread_indent(-1), funcname, filename, lineno);
}
```

The following script uses the tapset above to provide a top-like view of all running CPython code, showing the top 20 most frequently-entered bytecode frames, each second, across the whole system:

```
global fn_calls;

probe python.function.entry
{
    fn_calls[pid(), filename, funcname, lineno] += 1;
}

probe timer.ms(1000) {
    printf("\033[2J\033[1;1H") /* clear screen \*/
    printf("%6s %80s %6s %30s %6s\n",
           "PID", "FILENAME", "LINE", "FUNCTION", "CALLS")
    foreach ([pid, filename, funcname, lineno] in fn_calls- limit 20) {
        printf("%6d %80s %6d %30s %6d\n",
            pid, filename, lineno, funcname,
            fn_calls[pid, filename, funcname, lineno]);
    }
```

(下页继续)

(续上页)

```
    delete fn_calls;
}
```

索引