
使用 DTrace 和 SystemTap 检测 CPython

发布 3.8.14

Guido van Rossum
and the Python development team

九月 12, 2022

Python Software Foundation
Email: docs@python.org

Contents

| | | |
|---|-------------------|---|
| 1 | 启用静态标记 | 2 |
| 2 | 静态 DTrace 探针 | 3 |
| 3 | 静态 SystemTap 标记 | 4 |
| 4 | 可用的静态标记 | 5 |
| 5 | SystemTap Tapsets | 6 |
| 6 | 例子 | 7 |

作者 David Malcolm

作者 Łukasz Langa

DTrace 和 SystemTap 是监控工具，它们都提供了一种检查计算机系统上的进程的方法。它们都使用特定领域的语言，允许用户编写脚本，其中：

- 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) | | |
| Build ID: df924a2b08a7e89f6e11251d4602022977af2670 | | |

(下页继续)

(续上页)

```
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:_
↪0x000000000008d6bf6
    Arguments: -4@%ebx
  stapsdt         0x00000030      NT_STAPSDT (SystemTap probe_
↪descriptors)
    Provider: python
    Name: gc__done
    Location: 0x00000000004374e1, Base: 0x0000000000630ce2, Semaphore:_
↪0x000000000008d6bf8
    Arguments: -8@%rax
  stapsdt         0x00000045      NT_STAPSDT (SystemTap probe_
↪descriptors)
    Provider: python
    Name: function__entry
    Location: 0x000000000053db6c, Base: 0x0000000000630ce2, Semaphore:_
↪0x000000000008d6be8
    Arguments: 8@%rbp 8@%r12 -4@%eax
  stapsdt         0x00000046      NT_STAPSDT (SystemTap probe_
↪descriptors)
    Provider: python
    Name: function__return
    Location: 0x000000000053dba8, Base: 0x0000000000630ce2, Semaphore:_
↪0x000000000008d6bea
    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

下面的 DTrace 脚本示例可以用来显示一个 Python 脚本的调用/返回层次结构，只在调用名为“start”的函数内进行跟踪。换句话说，导入时的函数调用不会被列出。

```
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++;
}

python$target:::function-return
/self->trace/
```

(下页继续)

(续上页)

```
{
    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;
}
```

它可以这样调用:

```
$ 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 静态 SystemTap 标记

使用 SystemTap 集成的底层方法是直接使用静态标记。这需要你显式地说明包含它们的二进制文件。

例如, 这个 SystemTap 脚本可以用来显示 Python 脚本的调用/返回层次结构:

```
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);
}

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

    printf("%s <= %s in %s:%d\\n",
```

(下页继续)

(续上页)

```
        thread_indent(-1), funcname, filename, lineno);  
}
```

它可以这样调用:

```
$ 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
```

其中的列是:

- 脚本开始后经过的微秒数
- 可执行文件的名字
- 进程的 PID

其余部分则表示脚本执行时的调用/返回层次结构。

对于 `--enable-shared` 构建的 CPython 来说, 这些标记是包含在 `libpython` 共享库中的, 探针的点状路径需要反映这一点。比如上面例子中的这一行:

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

应改为:

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

(假设是 CPython 3.6 的调试构建)

4 可用的静态标记

function__entry(str filename, str funcname, int lineno)

这个标记表示一个 Python 函数的执行已经开始。它只对纯 Python (字节码) 函数触发。

文件名、函数名和行号作为位置参数提供给跟踪脚本, 必须使用 `$arg1`, `$arg2`, `$arg3` 访问:

- `$arg1: (const char *)` 文件名, 使用 `user_string($arg1)` 访问
- `$arg2: (const char *)` 函数名, 使用 `user_string($arg2)` 访问
- `$arg3: int` 行号

function__return(str filename, str funcname, int lineno)

这个标记与 `function__entry()` 相反, 表示 Python 函数的执行已经结束 (通过 `return` 或者异常)。它只对纯 Python (字节码) 函数触发。

参数和 `function__entry()` 相同

line(str filename, str funcname, int lineno)

这个标记表示一个 Python 行即将被执行。它相当于用 Python 分析器逐行追踪。它不会在 C 函数中触发。

参数和 `function__entry()` 相同

gc__start(int generation)

当 Python 解释器启动一个垃圾回收循环时被触发。arg0 是要扫描的生成器，如 `gc.collect()`。

gc__done(long collected)

当 Python 解释器完成一个垃圾回收循环时被触发。arg0 是收集到的对象的数量。

import__find__load__start(str modulename)

在 `importlib` 试图查找并加载模块之前被触发。arg0 是模块名称。

3.7 新版功能。

import__find__load__done(str modulename, int found)

在 `importlib` 的 `find_and_load` 函数被调用后被触发。arg0 是模块名称，arg1 表示模块是否成功加载。

3.7 新版功能。

audit(str event, void *tuple)

当 `sys.audit()` 或 `PySys_Audit()` 被调用时启动。arg0 是事件名称的 C 字符串，arg1 是一个指向元组对象的 `PyObject` 指针。

3.8 新版功能。

5 SystemTap Tapsets

使用 SystemTap 集成的更高层次的方法是使用“tapset”。SystemTap 的等效库，它隐藏了静态标记的一些底层细节。

这里是一个基于 CPython 的非共享构建的 tapset 文件。

```
/*
   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 = $arg4
}
probe python.function.return = process("python").mark("function__return")
{
    filename = user_string($arg1);
    funcname = user_string($arg2);
    lineno = $arg3;
    frameptr = $arg4
}
```

如果这个文件安装在 SystemTap 的 tapset 目录下（例如“`/usr/share/systemtap/tapset`”），那么这些额外的探测点就会变得可用。

python.function.entry(str filename, str funcname, int lineno, frameptr)

这个探针点表示一个 Python 函数的执行已经开始。它只对纯 Python（字节码）函数触发。

python.function.return(str filename, str funcname, int lineno, frameptr)

这个探针点是 `python.function.return` 的反义操作，表示一个 Python 函数的执行已经结束（或是通过 `return`，或是通过异常）。它只会针对纯 Python（字节码）函数触发。

6 例子

这个 SystemTap 脚本使用上面的 tapset 来更清晰地实现上面给出的跟踪 Python 函数调用层次结构的例子，而不需要直接命名静态标记。

```
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);
}
```

下面的脚本使用上面的 tapset 提供了所有运行中的 CPython 代码的顶部视图，显示了整个系统中每一秒钟最频繁输入的前 20 个字节码帧。

```
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]);
    }
    delete fn_calls;
}
```