DTrace と SystemTap で CPython を測定する

リリース *3.12.3*

Guido van Rossum and the Python development team

5 月 03, 2024

目次

1 静的マーカーの有効化 2

2 静的 DTrace プローブ 4

3 静的 SystemTap マーカー 5

4 利用可能な静的マーカー 6

5 SystemTap Tapset 7

6 使用例 8

author
: David Malcolm

author
: Łukasz Langa

Dtrace と SystemTap はモニタリングツールで、コンピュータシステムのプロセスが何をしているかを調べる方法を提供します。どちらもドメイン固有言語 (domain-specific language) を使用して、次のことができるスクリプトをユーザが書けます:

- 観測対象のプロセスを絞り込む
- 関心のあるプロセスからデータを収集する
- 収集したデータからレポートを生成する

Python 3.6 では、CPython は ” プローブ” としても知られる ” マーカー” を埋め込んだビルドが行えます。マーカーは DTrace や SystemTap のスクリプトから観測でき、システムの CPython プロセスが何をしてい

るかを観察するのが簡単になります。

CPython 実装の詳細: DTrace マーカーは CPython インタプリタの実装詳細です。CPython のバージョン間でプローブの互換性があるという保証はありません。CPython のバージョンを変えると、DTrace スクリプトは警告無しに動作しなくなったり、おかしな動作をする可能性があります。

1 静的マーカーの有効化

macOS には組み込みの DTrace サポートが備わっています。Linux では SystemTap 用のマーカーを埋め込んで CPython をビルドするためには、SystemTap 開発ツールをインストールしなければなりません。

Linux マシンでは、SystemTap 開発ツールのインストールは次のように行えます:

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

もしくは:

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

CPython must then be `configured with the --with-dtrace option`:

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

On macOS, you can list available DTrace probes by running a Python process in the background and listing all probes made available by the Python provider:

```
$ 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
```

On Linux, you can verify if the SystemTap static markers are present in the built binary by seeing if it contains a ”.note.stapsdt” section.

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

If you’ve built Python as a shared library (with the `--enable-shared` configure option), you need to look instead within the shared library. For example:

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

十分に新しい readelf ではメタデータを出力できます:

```
$ 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

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
```

The above metadata contains information for SystemTap describing how it can patch strategically placed machine code instructions to enable the tracing hooks used by a SystemTap script.

2 静的 DTrace プローブ

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++;
}

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;
}
```

この例は次のように実行できます:

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

出力はこのようになります:

```
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
```

(次のページに続く)

(前のページからの続き)

```
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 静的 SystemTap マーカー

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.

例えば、この SystemTap スクリプトは Python の呼び出し/返却 (call/return) 階層を表示するのに使えます:

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

この例は次のように実行できます:

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

出力はこのようになります:

```
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
```

(次のページに続く)

(前のページからの続き)

```
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
```

それぞれの列の内容は次の通りです:

- スクリプトを起動してからのマイクロ秒単位の時間
- 実行可能ファイルの名前
- プロセスの PID

そして、残りの部分はスクリプトが実行していた呼び出し/返却階層を表示しています。

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 利用可能な静的マーカー

function__entry(str filename, str funcname, int lineno)

このマーカーは Python の関数の実行が開始されたことを示しています。このマーカーは、ピュア Python (バイトコード) の関数でしか起動されません。

トレーススクリプトには位置引数として、ファイル名、関数名、行番号が渡され、必ず `$arg1`, `$arg2`, `$arg3` で渡されます:

- `$arg1` : `(const char *)` ファイル名、`user_string($arg1)` でアクセスできます
- `$arg2` : `(const char *)` 関数名、`user_string($arg2)` でアクセスできます
- `$arg3` : `int` 行番号

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)

このマーカーは、これから実行される Python の行を示しています。これは Python プロファイラによる行ごとのトレースと同等です。このマーカーは C 関数の内部では起動されません。

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

gc__start(int generation)

Python インタプリタによる循環参照のガベージコレクションが開始されたときに発火します。`gc.collect()` と同じように `arg0` は走査する対象の世代です。

gc__done(long collected)

Python インタプリタによる循環参照のガベージコレクションが完了したときに発火します。`arg0` は回収したオブジェクトの数です。

import__find__load__start(str modulename)

Fires before `importlib` attempts to find and load the module. `arg0` is the module name.

Added in version 3.7.

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.

Added in version 3.7.

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.

Added in version 3.8.

5 SystemTap Tapset

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
```

(次のページに続く)

(前のページからの続き)

```
}
probe python.function.return = process("python").mark("function__return")
{
    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 使用例

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;
}
```

(次のページに続く)

(前のページからの続き)

```
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;
}
```