
DTrace와 SystemTap으로 CPython 계측하기

출시 버전 3.10.16

Guido van Rossum
and the Python development team

12월 07, 2024

Contents

1	정적 마커 활성화하기	2
2	정적 DTrace 프로브	3
3	정적 SystemTap 마커	4
4	사용 가능한 정적 마커	5
5	SystemTap 탭셋	6
6	예제	7

저자 David Malcolm

저자 Łukasz Langa

DTrace와 SystemTap은 컴퓨터 시스템의 프로세스가 하는 일을 검사할 수 있는 모니터링 도구입니다. 둘 다 도메인 특정 언어를 사용하여 다음과 같은 작업을 하는 스크립트를 작성할 수 있도록 합니다:

- 관찰할 프로세스를 걸러내기
- 관심 있는 프로세스에서 자료를 수집하기
- 데이터에 대한 보고서를 생성하기

파이썬 3.6부터, CPython은 DTrace나 SystemTap 스크립트에서 볼 수 있는 “마커(markers)” (“프로브(probes)”라고도 합니다)를 내장하도록 빌드할 수 있어서, 시스템에서 CPython 프로세스가 수행하고 있는 작업을 쉽게 관찰할 수 있습니다.

CPython 구현 상세: DTrace 마커는 CPython 인터프리터의 구현 세부 사항입니다. CPython 버전 간의 프로브 호환성에 대한 보장은 없습니다. CPython 버전을 변경할 때 경고 없이 DTrace 스크립트가 작동하지 않거나 올바르게 작동하지 않을 수 있습니다.

1 정적 마커 활성화하기

macOS는 DTrace를 기본적으로 지원합니다. 리눅스에서는, SystemTap을 위한 마커를 내장하도록 CPython을 빌드하려면, SystemTap 개발 도구를 설치해야 합니다.

리눅스 기계에서, 이렇게 하면 됩니다:

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

또는:

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

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

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

macOS에서, 배경에서 파이썬 프로세스를 실행하고 파이썬 공급자가 제공 한 모든 프로브를 나열하여 사용 가능한 DTrace 프로브를 나열할 수 있습니다:

```
$ 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

리눅스에서, “.note.stapsdt” 섹션이 있는지 확인하여 빌드 된 바이너리에 SystemTap 정적 마커가 있는지 확인할 수 있습니다.

```
$ 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
```

충분히 최신의 readelf는 메타 데이터를 인쇄할 수 있습니다:

```
$ 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		

(다음 페이지에 계속)

```

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 정적 DTrace 프로브

다음 예제 DTrace 스크립트는 파이썬 스크립트의 호출/반환 계층 구조를 표시하는 데 사용할 수 있습니다. “start” 라는 함수의 호출 내부에서만 추적합니다. 즉, 임포트 시점의 함수 호출은 나열되지 않습니다:

```

self int indent;

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

python$target:::function-entry
/self->trace/
{
    printf("%d\tt%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\tt%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;
}

```

다음과 같은 식으로 호출할 수 있습니다:

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

출력은 이런 식입니다:

```

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 정적 SystemTap 마커

SystemTap 통합을 사용하는 저수준의 방법은 정적 마커를 직접 사용하는 것입니다. 이를 포함하는 바이너리 파일을 명시적으로 지정해야 합니다.

예를 들어, 이 SystemTap 스크립트는 파이썬 스크립트의 호출/반환 계층 구조를 표시하는 데 사용할 수 있습니다:

```

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);
}

```

(다음 페이지에 계속)

```

lineno = $arg3;

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

```

다음과 같은 식으로 호출할 수 있습니다:

```

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

```

출력은 이런 식입니다:

```

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

```

이때 열은 다음과 같습니다:

- 스크립트 시작으로부터 마이크로초 단위의 시간
- 실행 파일의 이름
- 프로세스의 PID

나머지는 스크립트가 실행될 때 호출/반환 계층 구조를 나타냅니다.

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") {

```

대신 이렇게 되어야 합니다:

```

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

```

(assuming a debug build of CPython 3.6)

4 사용 가능한 정적 마커

function__entry(str filename, str funcname, int lineno)

이 마커는 파이썬 함수의 실행이 시작되었음을 나타냅니다. 순수 파이썬 (바이트 코드) 함수에서만 트리거 됩니다.

파일명, 함수 이름 및 줄 번호가 위치 인자로 추적 스크립트에 제공됩니다. `$arg1`, `$arg2`, `$arg3`를 사용하여 액세스해야 합니다:

- `$arg1`: (const char *) 파일명, `user_string($arg1)` 를 사용하여 액세스할 수 있습니다
- `$arg2`: (const char *) 함수 이름, `user_string($arg2)` 를 사용하여 액세스할 수 있습니다
- `$arg3`: int 줄 번호

function__return(str filename, str funcname, int lineno)

이 마커는 `function__entry()` 의 반대이며, 파이썬 함수의 실행이 종료되었음을 나타냅니다 (return를 통해서나 예외를 통해). 순수 파이썬 (바이트 코드) 함수에서만 트리거 됩니다.

인자는 `function__entry()` 와 같습니다.

line(str filename, str funcname, int lineno)

이 마커는 파이썬 줄이 실행되려고 함을 나타냅니다. 파이썬 프로파일러를 사용하는 줄 단위 추적과 동등합니다. C 함수 내에서는 트리거 되지 않습니다.

인자는 `function__entry()` 와 같습니다.

gc__start(int generation)

파이썬 인터프리터가 가비지 수집 사이클을 시작할 때 발생합니다. `arg0`은 `gc.collect()` 처럼 스캔할 세대(`generation`)입니다.

gc__done(long collected)

파이썬 인터프리터가 가비지 수집 사이클을 끝낼 때 발생합니다. `arg0`은 수집된 객체 수입니다.

import__find__load__start(str modulename)

`importlib`가 모듈을 찾고 로드하기 전에 발생합니다. `arg0`은 모듈 이름입니다.

버전 3.7에 추가.

import__find__load__done(str modulename, int found)

`importlib`의 모듈을 찾고 로드하는 함수가 호출된 후에 발생합니다. `arg0`은 모듈 이름이고, `arg1`은 모듈이 성공적으로 로드되었는지를 나타냅니다.

버전 3.7에 추가.

audit(str event, void *tuple)

`sys.audit()` 나 `PySys_Audit()` 가 호출될 때 발생합니다. `arg0`은 C 문자열로 된 이벤트 이름이고, `arg1`은 튜플 객체를 가리키는 `PyObject` 포인터입니다.

버전 3.8에 추가.

5 SystemTap 탭셋

SystemTap 통합을 사용하는 고수준의 방법은 “탭셋(tapset)”을 사용하는 것입니다: SystemTap의 라이브러리에 해당하는 것입니다, 정적 마커의 저수준 세부 정보를 숨깁니다.

다음은 CPython의 비공유 빌드에 기반한 탭셋 파일입니다:

```
/*
   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
}
```

이 파일을 SystemTap의 tapset 디렉터리(예를 들어, `/usr/share/systemtap/tapset`)에 설치하면, 다음과 같은 추가 프로브 포인트를 사용할 수 있습니다:

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

이 프로브 포인트는 파이썬 함수의 실행이 시작되었음을 나타냅니다. 순수 파이썬 (바이트 코드) 함수에서만 트리거 됩니다.

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

이 프로브 포인트는 `python.function.return`의 반대이며, 파이썬 함수의 실행이 종료되었음을 나타냅니다 (return를 통해서나 예외를 통해). 순수 파이썬 (바이트 코드) 함수에서만 트리거됩니다.

6 예제

이 SystemTap 스크립트는 위의 탭셋을 사용하여, 정적 마커의 이름을 직접 지정하지 않고도, 파이썬 함수 호출 계층 구조를 추적하는 위의 예제를 보다 명확하게 구현합니다.:

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