
Instrumentando o CPython com DTrace e SystemTap

Release 3.13.2

Guido van Rossum and the Python development team

fevereiro 17, 2025

Python Software Foundation
Email: docs@python.org

Sumário

| | | |
|---|-------------------------------------|---|
| 1 | Habilitando os marcadores estáticos | 2 |
| 2 | Sondas estáticas do DTrace | 3 |
| 3 | Marcadores estáticos do SystemTap | 4 |
| 4 | Marcadores estáticos disponíveis | 5 |
| 5 | Tapsets de SystemTap | 6 |
| 6 | Exemplos | 7 |

autor

David Malcolm

autor

Łukasz Langa

DTrace e SystemTap são ferramentas de monitoramento, cada uma fornecendo uma maneira de inspecionar o que os processos em um sistema de computador estão fazendo. Ambas usam linguagens específicas de domínio, permitindo que um usuário escreva scripts que:

- filtrar quais processos devem ser observados
- colem dados dos processos de interesse
- gerem relatórios sobre os dados

A partir do Python 3.6, o CPython pode ser criado com “marcadores” incorporados, também conhecidos como “sondas” (*probes*), que podem ser observados por um script DTrace ou SystemTap, facilitando o monitoramento do que os processos CPython em um sistema estão fazendo.

Os marcadores DTrace são detalhes de implementação do interpretador CPython. Não há garantias sobre a compatibilidade de sondas entre versões do CPython. Os scripts DTrace podem parar de funcionar ou funcionar incorretamente sem aviso ao alterar as versões do CPython.

1 Habilitando os marcadores estáticos

O macOS vem com suporte embutido para DTrace. No Linux, para construir o CPython com os marcadores incorporados para SystemTap, as ferramentas de desenvolvimento SystemTap devem ser instaladas.

Em uma máquina Linux, isso pode ser feito via:

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

ou:

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

O CPython deve então ser configurado com a opção `--with-dtrace`:

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

No macOS, você pode listar as sondas DTrace disponíveis executando um processo Python em segundo plano e listando todas as sondas disponibilizadas pelo provedor Python:

```
$ 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 |

No Linux, você pode verificar se os marcadores estáticos do SystemTap estão presentes no binário compilado, observando se ele contém uma seção `“.note.stapsdt”`.

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

Se você construiu o Python como uma biblioteca compartilhada (com a opção de configuração `--enable-shared`), você precisa procurar dentro da biblioteca compartilhada. Por exemplo:

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

Um `readelf` moderno o suficiente pode exibir os metadados:

```
$ 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) |

(continua na próxima página)

```

↪bitstring)
    Build ID: df924a2b08a7e89f6e11251d4602022977af2670

Displaying notes found at file offset 0x002d6c30 with length 0x00000144:
    Owner          Data size          Description
    stapsdt         0x000000031          NT_STAPSDT (SystemTap probe_
↪descriptors)
    Provider: python
    Name: gc__start
    Location: 0x00000000004371c3, Base: 0x0000000000630ce2, Semaphore:_
↪0x000000000008d6bf6
    Arguments: -4@%ebx
    stapsdt         0x000000030          NT_STAPSDT (SystemTap probe_
↪descriptors)
    Provider: python
    Name: gc__done
    Location: 0x00000000004374e1, Base: 0x0000000000630ce2, Semaphore:_
↪0x000000000008d6bf8
    Arguments: -8@%rax
    stapsdt         0x000000045          NT_STAPSDT (SystemTap probe_
↪descriptors)
    Provider: python
    Name: function__entry
    Location: 0x000000000053db6c, Base: 0x0000000000630ce2, Semaphore:_
↪0x000000000008d6be8
    Arguments: 8@%rbp 8@%r12 -4@%eax
    stapsdt         0x000000046          NT_STAPSDT (SystemTap probe_
↪descriptors)
    Provider: python
    Name: function__return
    Location: 0x000000000053dba8, Base: 0x0000000000630ce2, Semaphore:_
↪0x000000000008d6bea
    Arguments: 8@%rbp 8@%r12 -4@%eax

```

Os metadados acima contêm informações sobre o SystemTap descrevendo como ele pode corrigir instruções de código de máquina estrategicamente posicionadas para habilitar os ganchos de rastreamento usados por um script do SystemTap.

2 Sondas estáticas do DTtrace

O script DTrace de exemplo a seguir pode ser usado para mostrar a hierarquia de chamada/retorno de um script Python, rastreando apenas dentro da invocação de uma função chamada “start”. Em outras palavras, invocações de função em tempo de importação não serão listadas:

```

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);

```

(continua na próxima página)

```

        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;
}

```

Pode ser invocado assim:

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

O resultado deve ser algo assim:

```

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 Marcadores estáticos do SystemTap

A maneira de baixo nível de usar a integração do SystemTap é usar os marcadores estáticos diretamente. Isso requer que você declare explicitamente o arquivo binário que os contém.

Por exemplo, este script SystemTap pode ser usado para mostrar a hierarquia de chamada/retorno de um script Python:

```

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

```

(continua na próxima página)

```

        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);
    }

```

Pode ser invocado assim:

```

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

```

O resultado deve ser algo assim:

```

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

```

sendo as colunas:

- tempo em microssegundos desde o início do script
- nome do executável
- PID do processo

e o restante indica a hierarquia de chamada/retorno conforme o script é executado.

Para uma construção com `--enable-shared` do CPython, os marcadores estão contidos na biblioteca compartilhada `libpython`, e o caminho pontilhado da sonda precisa refletir isso. Por exemplo, esta linha do exemplo acima:

```

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

```

deve ler-se em vez disso:

```

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

```

(presumindo uma construção de depuração do CPython 3.6)

4 Marcadores estáticos disponíveis

function__entry(str filename, str funcname, int lineno)

Este marcador indica que a execução de uma função Python começou. Ele é acionado somente para funções Python puro (bytecode).

O nome do arquivo, o nome da função e o número da linha são fornecidos de volta ao script de rastreamento como argumentos posicionais, que devem ser acessados usando `$arg1`, `$arg2`, `$arg3`:

- \$arg1: nome de arquivo como (const char *), acessível usando user_string(\$arg1)
- \$arg2: nome da função como (const char *), acessível usando user_string(\$arg2)
- \$arg3: número da linha como int

function__return(str filename, str funcname, int lineno)

Este marcador é o inverso de function__entry(), e indica que a execução de uma função Python terminou (seja via return, ou via uma exceção). Ele é acionado somente para funções Python puro (bytecode).

Os argumentos são os mesmos de function__entry()

line(str filename, str funcname, int lineno)

Este marcador indica que uma linha Python está prestes a ser executada. É o equivalente ao rastreamento linha por linha com um perfilador Python. Ele não é acionado dentro de funções C.

Os argumentos são os mesmos de function__entry().

gc__start(int generation)

Dispara quando o interpretador Python inicia um ciclo de coleta de lixo. arg0 é a geração a ser percorrida, como gc.collect().

gc__done(long collected)

Dispara quando o interpretador Python termina um ciclo de coleta de lixo. arg0 é o número de objetos coletados.

import__find__load__start(str modulename)

Dispara antes de importlib tentar encontrar e carregar o módulo. arg0 é o nome do módulo.

Adicionado na versão 3.7.

import__find__load__done(str modulename, int found)

Dispara após a função find_and_load do importlib ser chamada. arg0 é o nome do módulo, arg1 indica se o módulo foi carregado com sucesso.

Adicionado na versão 3.7.

audit(str event, void *tuple)

Dispara quando sys.audit() ou PySys_Audit() é chamada. arg0 é o nome do evento como string C, arg1 é um ponteiro PyObject para um objeto tupla.

Adicionado na versão 3.8.

5 Tapsets de SystemTap

A maneira mais avançada de usar a integração do SystemTap é usar um “tapset”: o equivalente do SystemTap a uma biblioteca, que oculta alguns dos detalhes de nível inferior dos marcadores estáticos.

Aqui está um arquivo tapset, baseado em uma construção não compartilhada do CPython:

```
/*
  Fornece um envólucro de mais alto nível em volta dos marcadores
  function__entry e function__return:
  */
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")
```

(continua na próxima página)

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

Se este arquivo for instalado no diretório de tapsets do SystemTap (por exemplo, `/usr/share/systemtap/tapset`), estes pontos de sondagem adicionais ficarão disponíveis:

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

Este ponto de sondagem indica que a execução de uma função Python começou. Ele é acionado somente para funções Python puro (bytecode).

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

Este ponto de sondagem é o inverso de `python.function.entry`, e indica que a execução de uma função Python terminou (seja via `return`, ou via uma exceção). Ele é acionado somente para funções Python puro (bytecode).

6 Exemplos

Este script SystemTap usa o tapset acima para implementar de forma mais limpa o exemplo dado acima de rastreamento da hierarquia de chamada de função Python, sem precisar na diretamente

```
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);
}
```

O script a seguir usa o tapset acima para fornecer uma visão geral de todo o código CPython em execução, mostrando os 20 quadros de bytecode mais frequentemente inseridos, a cada segundo, em todo o sistema:

```
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;
}
```