
Suporte do Python ao perfilador perf do Linux

Release 3.13.7

Guido van Rossum and the Python development team

outubro 01, 2025

Python Software Foundation
Email: docs@python.org

Sumário

1	Como habilitar o suporte a perfilação com <code>perf</code>	4
2	Como obter os melhores resultados	4
3	Como trabalhar sem ponteiros de quadro	5
	Índice	7

autor

Pablo Galindo

O perfilador `perf` do Linux é uma ferramenta muito poderosa que permite criar perfis e obter informações sobre o desempenho da sua aplicação. `perf` também possui um ecossistema muito vibrante de ferramentas que auxiliam na análise dos dados que produz.

O principal problema de usar o perfilador `perf` com aplicações Python é que `perf` apenas obtém informações sobre símbolos nativos, ou seja, os nomes de funções e procedimentos escritos em C. Isso significa que os nomes de funções Python e seus nomes de arquivos em seu código não aparecerão na saída de `perf`.

Desde o Python 3.12, o interpretador pode ser executado em um modo especial que permite que funções do Python apareçam na saída do criador de perfilador `perf`. Quando este modo está habilitado, o interpretador interporá um pequeno pedaço de código compilado instantaneamente antes da execução de cada função Python e ensinará `perf` a relação entre este pedaço de código e a função Python associada usando arquivos de mapa `perf`.

Nota

O suporte para o perfilador `perf` está atualmente disponível apenas para Linux em arquiteturas selecionadas. Verifique a saída da etapa de construção `configure` ou verifique a saída de `python -m sysconfig | grep HAVE_PERF_TRAMPOLINE` para ver se o seu sistema é compatível.

Por exemplo, considere o seguinte script:

```
def foo(n):  
    result = 0
```

(continua na próxima página)

(continuação da página anterior)

```
for _ in range(n):
    result += 1
return result

def bar(n):
    foo(n)

def baz(n):
    bar(n)

if __name__ == "__main__":
    baz(1000000)
```

Podemos executar `perf` para obter amostras de rastreamentos de pilha da CPU em 9999 hertz:

```
$ perf record -F 9999 -g -o perf.data python meu_script.py
```

Então podemos usar `perf report` para analisar os dados:

```
$ perf report --stdio -n -g
```

#	Children	Self	Samples	Command	Shared Object	Symbol	
#	
↪	
#							
	91.08%	0.00%	0	python.exe	python.exe	[.] _start	
	---	_start					
		--90.71%	--	__libc_start_main			
				Py_BytesMain			
		--56.88%	--	pymain_run_python.constprop.0			
				--56.13%	--	_PyRun_AnyFileObject	
						_PyRun_SimpleFileObject	
					--55.02%	--run_mod	
						--54.65%	--PyEval_EvalCode
						PyEval	
↪	EvalFrameDefault						
						PyObject_	
↪	Vectorcall						
						_PyEval_Vector	
						PyEval	
↪	EvalFrameDefault						
						PyObject_	
↪	Vectorcall						
						_PyEval_Vector	
						PyEval	
↪	EvalFrameDefault						
						PyObject_	
↪	Vectorcall						
						_PyEval_Vector	
						--51.67%	--

(continua na próxima página)

(continuação da página anterior)

↪ PyEval_EvalFrameDefault					
					--
↪ 11.52% -- _PyLong_Add					
↪					
↪	-- 2.97% -- _PyObject_Malloc				
...					

Como você pode ver, as funções Python não são mostradas na saída, apenas `_PyEval_EvalFrameDefault` (a função que avalia o bytecode Python) aparece. Infelizmente isso não é muito útil porque todas as funções Python usam a mesma função C para avaliar bytecode, portanto não podemos saber qual função Python corresponde a qual função de avaliação de bytecode.

Em vez disso, se executarmos o mesmo experimento com o suporte `perf` ativado, obteremos:

```
$ perf report --stdio -n -g
```

#	Children	Self	Samples	Command	Shared Object	Symbol	
#	
↩	-----						
#	90.58%	0.36%	1	python.exe	python.exe	[.] _start	
	---	_start					
		--89.86%--		__libc_start_main			
				Py_BytesMain			
				--55.43%--	pymain_run_python.constprop.0		
					--54.71%--	_PyRun_AnyFileObject	
						_PyRun_SimpleFileObject	
					--53.62%--	run_mod	
						--53.26%--	PyEval_EvalCode
						py::<module>:/	
↩src/script.py							
						PyEval	
↩EvalFrameDefault							
						PyObject_	
↩Vectorcall							
						_PyEval_Vector	
						py::baz:/src/	
↩script.py							
						PyEval	
↩EvalFrameDefault							
						PyObject_	
↩Vectorcall							
						_PyEval_Vector	
						py::bar:/src/	
↩script.py							
						PyEval	
↩EvalFrameDefault							
						PyObject_	

(continua na próxima página)

(continuação da página anterior)

```

↪Vectorcall
|
|
|
_PyEval_Vector
py::foo:/src/
↪script.py
|
|
|
|
--51.81%--_
↪PyEval_EvalFrameDefault
|
|
|
|
|
|
--
↪13.77%--_PyLong_Add
|
|
|
|
|
|
↪
|
|
|
|
|
|
--3.26%--PyObject_Malloc

```

1 Como habilitar o suporte a perfilação com perf

O suporte à perfilação com `perf` pode ser habilitado desde o início usando a variável de ambiente `PYTHONPERFSUPPORT` ou a opção `-X perf`, ou dinamicamente usando `sys.activate_stack_trampoline()` e `sys.deactivate_stack_trampoline()`.

As funções `sys` têm precedência sobre a opção `-X`, a opção `-X` tem precedência sobre a variável de ambiente.

Exemplo usando a variável de ambiente:

```
$ PYTHONPERFSUPPORT=1 perf record -F 9999 -g -o perf.data python meu_script.py
$ perf report -g -i perf.data
```

Exemplo usando a opção -X:

```
$ perf record -F 9999 -g -o perf.data python -X perf meu_script.py
$ perf report -q -i perf.data
```

Exemplo usando as APIs de `sys` em `example.py`:

```
import sys

sys.activate_stack_trampoline("perf")
do_profiled_stuff()
sys.deactivate_stack_trampoline()

non_profiled_stuff()
```

... então:

```
$ perf record -F 9999 -g -o perf.data python ./example.py
$ perf report -g -i perf.data
```

2 Como obter os melhores resultados

Para melhores resultados, Python deve ser compilado com `CFLAGS="-fno-omit-frame-pointer -mno-omit-leaf-frame-pointer"`, pois isso permite que os perfiladores façam o desenrolamento de pilha (ou *stack unwinding*) usando apenas o ponteiro de quadro e não no DWARF informações de depuração. Isso ocorre porque como o código interposto para permitir o suporte `perf` é gerado dinamicamente, ele não possui nenhuma informação de depuração DWARF disponível.

Você pode verificar se o seu sistema foi compilado com este sinalizador executando:

```
$ python -m sysconfig | grep 'no-omit-frame-pointer'
```

Se você não vir nenhuma saída, significa que seu interpretador não foi compilado com ponteiros de quadro e, portanto, pode não ser capaz de mostrar funções Python na saída de `perf`.

3 Como trabalhar sem ponteiros de quadro

Se você estiver trabalhando com um interpretador Python que foi compilado sem ponteiros de quadro, você ainda pode usar o perfilador `perf`, mas a sobrecarga será um pouco maior porque o Python precisa gerar informações de desenrolamento para cada chamada de função Python em tempo real. Além disso, `perf` levará mais tempo para processar os dados porque precisará usar as informações de depuração DWARF para desenrolar a pilha e este é um processo lento.

Para habilitar esse modo, você pode usar a variável de ambiente `PYTHON_PERF_JIT_SUPPORT` ou a opção `-X perf_jit`, que habilitará o modo JIT para o perfilador `perf`.

Nota

Devido a um bug na ferramenta `perf`, apenas versões `perf` superiores à v6.8 funcionarão com o modo JIT. A correção também foi portada para a versão v6.7.2 da ferramenta.

Note que ao verificar a versão da ferramenta `perf` (o que pode ser feito executando `perf version`) você deve levar em conta que algumas distros adicionam alguns números de versão personalizados, incluindo um caractere `-`. Isso significa que `perf 6.7-3` não é necessariamente `perf 6.7.3`.

Ao usar o modo JIT do `perf`, você precisa de uma etapa extra antes de poder executar `perf report`. Você precisa chamar o comando `perf inject` para injetar as informações JIT no arquivo `perf.data`:

```
$ perf record -F 9999 -g -k 1 --call-graph dwarf -o perf.data python -Xperf_jit_
↪meu_script.py
$ perf inject -i perf.data --jit --output perf.jit.data
$ perf report -g -i perf.jit.data
```

ou usando a variável de ambiente:

```
$ PYTHON_PERF_JIT_SUPPORT=1 perf record -F 9999 -g --call-graph dwarf -o perf.data_
↪python meu_script.py
$ perf inject -i perf.data --jit --output perf.jit.data
$ perf report -g -i perf.jit.data
```

O comando `perf inject --jit` lerá `perf.data`, pegará automaticamente o arquivo de dump `perf` que o Python cria (em `/tmp/perf-$PID.dump`) e, em seguida, criará `perf.jit.data` que mescla todas as informações JIT. Ele também deve criar muitos arquivos `jitted-XXXX-N.so` no diretório atual, que são imagens ELF para todos os trampolins JIT que foram criados pelo Python.

Aviso

Ao usar `--call-graph dwarf`, a ferramenta `perf` fará snapshots da pilha do processo que está sendo perfilado e salvará as informações no arquivo `perf.data`. Por padrão, o tamanho do dump da pilha é de 8192 bytes, mas você pode alterar o tamanho passando-o após uma vírgula, como `--call-graph dwarf,16384`.

O tamanho do dump da pilha é importante porque, se for muito pequeno, o `perf` não conseguirá desfazer o descompasso da pilha e a saída será incompleta. Por outro lado, se for muito grande, o `perf` não conseguirá amostrar o processo com a frequência desejada, pois a sobrecarga será maior.

O tamanho da pilha é particularmente importante ao criar perfis de código Python compilado com níveis de otimização baixos (como `-O0`), pois essas construções tendem a ter quadros de pilha maiores. Se você estiver

compilando Python com `-O0` e não estiver vendo funções Python na saída do perfil, tente aumentar o tamanho do despejo de pilha para 65528 bytes (o máximo):

```
$ perf record -F 9999 -g -k 1 --call-graph dwarf,65528 -o perf.data python -  
→Xperf_jit meu_script.py
```

Diferentes sinalizadores de compilação podem impactar significativamente os tamanhos de pilha:

- Construções com `-O0` geralmente têm quadros de pilha muito maiores do que aquelas com `-O1` ou superior
- Adiciona otimizações (`-O1`, `-O2`, etc.) normalmente reduz o tamanho da pilha
- Os ponteiros de quadro (`-fno-omit-frame-pointer`) geralmente fornecem um desenrolamento de pilha mais confiável

Índice

P

`PYTHON_PERF_JIT_SUPPORT`, 5

`PYTHONPERFSUPPORT`, 4

V

variável de ambiente

`PYTHON_PERF_JIT_SUPPORT`, 5

`PYTHONPERFSUPPORT`, 4