Python 对 Linux perf 性能分析器的支持

发行版本 *3.13.1*

Guido van Rossum and the Python development team

一月 29, 2025

Python Software Foundation
Email: docs@python.org

Contents

1 如何启用 perf 性能分析支持 3

2 如何获取最佳结果 4

3 如何在不带帧指针的情况下使用 4

索引 6

作者
Pablo Galindo

Linux perf 性能分析器 是一个非常强大的工具，它允许你分析并获取有关你的应用程序运行性能的信息。`perf` 还拥有一个非常活跃的工具生态系统可以帮助分析它所产生的数据。

将 `perf` 性能分析器与 Python 应用程序配合使用的主要问题在于 `perf` 只能获取原生符号的信息，即以 C 编写的函数和过程的名称。这意味着在你的代码中的 Python 函数名称和文件名称将不会出现在 `perf` 输出中。

从 Python 3.12 开始，解释器可以运行于一个允许 `perf` 性能分析器的输出中显示 Python 函数的特殊模式下。当启用此模式时，解释器将在每个 Python 函数执行之前插入一小段即时编译的代码，它将使用 perf 映射文件来告知 `perf` 这段代码与相关联的 Python 函数之间的关系。

> **备注**
>
> 对 `perf` 性能分析器的支持目前仅在特定架构的 Linux 上可用。请检查 `configure` 构建步骤的输出或检查 `python -m sysconfig | grep HAVE_PERF_TRAMPOLINE` 的输出来确定你的系统是否受到支持。

例如，考虑以下脚本:

```
def foo(n):
    result = 0
    for _ in range(n):
        result += 1
    return result
```

(续下页)

(接上页)

```
def bar(n):
    foo(n)

def baz(n):
    bar(n)

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

我们可以运行 `perf` 以 9999 赫兹的频率来对 CPU 栈追踪信息进行采样:

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

然后我们可以使用 `perf report` 来分析数据:

```
$ 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%--_PyEval_
↪EvalFrameDefault
                        |          |          |                     |          |
                        |          |          |                     |          |--11.52%--_
↪PyLong_Add
                        |          |          |                     |          |          |
                        |          |          |                     |          |          |--
↪2.97%--_PyObject_Malloc
...
```

如你所见，Python 函数不会显示在输出中，只有 `_PyEval_EvalFrameDefault` (评估 Python 字节码的函数) 会显示出来。不幸的是那没有什么用处因为所有 Python 函数都使用相同的 C 函数来评估字节码所以我们无法知道哪个 Python 函数与哪个字节码评估函数相对应。

相反，如果我们在启用 `perf` 支持的情况下运行相同的实验代码我们将获得:

```
$ 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_Vectorcall
                        |          |          |                     _PyEval_Vector
                        |          |          |                     py::foo:/src/script.py
                        |          |          |                     |
                        |          |          |                     |--51.81%--_PyEval_
↪EvalFrameDefault
                        |          |          |                     |          |
                        |          |          |                     |          |--13.77%--_
↪PyLong_Add
                        |          |          |                     |          |          |
                        |          |          |                     |          |          |--
↪3.26%--_PyObject_Malloc
```

1 如何启用 perf 性能分析支持

要启动 perf 性能分析支持可以通过使用环境变量 PYTHONPERFSUPPORT 或 -X perf 选项，或者动态地使用 sys.activate_stack_trampoline() 和 sys.deactivate_stack_trampoline() 来运行。

sys 函数的优先级高于 -X 选项，-X 选项的优先级高于环境变量。

示例，使用环境变量:

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

示例，使用 -X 选项:

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

示例，在文件 `example.py` 中使用 `sys` API:

```
import sys

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

non_profiled_stuff()
```

... 然后:

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

2 如何获取最佳结果

要获取最佳结果，Python 应当使用 `CFLAGS="-fno-omit-frame-pointer -mno-omit-leaf-frame-pointer"` 来编译因为这将允许性能分析器仅使用帧指针而不是基于 DWARF 调试信息进行展开。这是因为被插入以允许 `perf` 支持的代码是动态生成的所以它没有任何 DWARF 调试信息可用。

你可以通过运行以下代码来检查你的系统是否为附带此旗标来编译的:

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

如果你没有看到任何输出则意味着你的解释器没有附带帧指针来编译因而它将无法在 `perf` 的输出中显示 Python 函数。

3 如何在不带帧指针的情况下使用

如果你使用在不带帧指针的情况下编译的 Python 解释器，你仍然可以使用 `perf` 性能分析器，但会有较高的资源开销因为 Python 需要为每个 Python 函数即时生成回撤信息。此外，`perf` 将花费更多时间来处理数据因为它需要使用 DWARF 调试信息来回撤栈而这是一个缓慢的过程。

要启用此模式，你可以使用环境变量 `PYTHON_PERF_JIT_SUPPORT` 或 `-X perf_jit` 选项，它将为 `perf` 性能分析器启用 JIT 模式。

> **备注**
>
> 由于 `perf` 工具的一个程序错误，只有 `perf` 版本号高于 v6.8 才能使用 JIT 模式。修复也向下移植到了此工具的 v6.7.2 版。
>
> 请注意在检测 `perf` 工具的版本时（这可通过运行 `perf version` 来完成）你必须将某些发行版添加包括了 `-` 字符的自定义版本号纳入考虑。这意味着 `perf 6.7-3` 不一定等于 `perf 6.7.3`。

当使用 perf JIT 模式时，在你运行 `perf report` 之前你还需要一个额外的步骤。你需要调用 `perf inject` 命令来将 JIT 信息注入 `perf.data` 文件。:

```
$ perf record -F 9999 -g --call-graph dwarf -o perf.data python -Xperf_jit my_script.py
$ perf inject -i perf.data --jit --output perf.jit.data
$ perf report -g -i perf.jit.data
```

或者使用环境变量:

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

`perf inject --jit` 命令将读取 `perf.data`，自动获取 Python 创建的 perf 转储文件（在 `/tmp/perf-$PID.dump` 中），然后创建将所有 JIT 信息合并到一起的 `perf.jit.data` 文件。它还会在当前目录下创建大量 `jitted-XXXX-N.so` 文件，它们是 Python 所创建的所有 JIT 中间数据的 ELF 映像。

> **警告**
>
> 请注意当使用 `--call-graph dwarf` 时 `perf` 工具将保存被分析的进程栈的快照并将信息保存到 `perf.data` 文件中。在默认情况下栈转储数据的大小是 8192 字节但用户可以通过在逗号后传入大小值例如 `--call-graph dwarf,4096` 来改变这个大小。栈转储数据的大小很重要因为如果这个大小过小则 `perf` 将无法回撤栈而输出将是不完整的。另一方面，如果这个大小过大，则 `perf` 将无法足够频繁地对进程执行采样而导致更高的开销。

索引

非字母

环境变量
    PYTHON_PERF_JIT_SUPPORT, 4
    PYTHONPERFSUPPORT, 3

P

PYTHON_PERF_JIT_SUPPORT, 4
PYTHONPERFSUPPORT, 3