Python profiler

Oleg Shipulin

14 февраля 2026 г.

Содержание

1 Введение **3**
1.1 Мотивация . . . 3
1.2 Объект и предмет исследования . . . 3
1.3 Цель исследования . . . 3

2 Профилирование **4**
2.1 Виды профилирования . . . 4
2.2 Основные методы профилирования . . . 4
2.3 Инструментирование (tracing) . . . 5
2.4 Статистическое (sampling) . . . 5
2.5 Сравнение методов . . . 6

3 Профилирование на уровне интерпретатора Python **7**
3.1 Инструментирующие профилировщики . . . 7
3.1.1 Profile (Legacy) . . . 7
3.1.2 cProfile . . . 8
3.1.3 line_profiler . . . 11
3.2 Статистические профилировщики . . . 13
3.2.1 py-spy . . . 13
3.2.2 pprofile . . . 15
3.3 Специализированные решения . . . 18
3.3.1 yappi . . . 18

4 Профилирование нативных расширений (C extensions) **21**
4.1 Особенности профилирование кода на C/C++ внутри Python . . . 21
4.2 Инструменты для анализа C/C++: gprof, Valgrind (Callgrind), Google PerfTools 21
4.3 Получение смешанных стеков вызовов (Python + C) с помощью perf и eBPF . . 21

5 Системное профилирование Python-приложений **21**
5.1 Использование утилиты perf в Linux для анализа на уровне ядра . . . 21
5.2 DTrace и SystemTap: динамическая трассировка . . . 21
5.3 Профилирование операций ввода-вывода и работы с сетью . . . 21

6 Сравнительный анализ и практические рекомендации **21**

1 Введение

1.1 Мотивация

Так как Python сам по себе не самый быстрый язык программирования, то по мере роста проекта, усложнения его архитектуры и увеличения объемов обрабатывамых данных, даже незначительные куски неэффективного кода могут накапливать задержки, превращая программу в невероятно медленную. Пример такого кода - применение не самой оптимизированный структуры данных для конкретного сценария.

Найти bottleneck (узкое горлышко) и не самого оптимизированного кода помогает профилирование, о котором и пойдёт речь в последующей статье.

1.2 Объект и предмет исследования

Объектом исследования являются программные средства динамического анализа производительности кода на языке Python, включая как встроенные и сторонние профилировщики, так и системные утилиты, позволяющие исследовать выполнение Python-программ на различных уровнях абстракции: от интерпретируемого кода до нативных расширений и системных вызовов.

Предметом исследования выступают методы и техники профилирования, реализованные в этих средствах: инструментирование (tracing), статистическое профилирование (sampling), построчный анализ, профилирование памяти, а также подходы к анализу производительности на уровне операционной системы (perf, Valgrind) и их применимость к Python-приложениям.

1.3 Цель исследования

Цель работы — провести комплексный сравнительный анализ инструментов профилирования Python, охватывающий все уровни исполнения программы: от исходного кода на Python до нативных расширений и взаимодействия с ядром ОС. На основе этого анализа выявить сильные и слабые стороны каждого класса инструментов, определить области их эффективного применения и предложить рекомендации по выбору средств для решения конкретных задач оптимизации производительности.

Для достижения поставленной цели необходимо решить следующие **задачи**:

1. Рассмотреть теоретические основы профилирования: определить понятие профилирования, классифицировать существующие методы (инструментирование и статистическое профилирование), описать их достоинства и недостатки.

2. Выполнить обзор профилировщиков, входящих в стандартную библиотеку Python (profile, cProfile), и охарактеризовать особенности их реализации, достоинства и ограничения.

3. Изучить специализированные профилировщики, ориентированные на построчный анализ (line_profiler), анализ памяти (memory_profiler), работу с многопоточными и асинхронными приложениями (yappi).

4. Рассмотреть инструменты статистического профилирования (py-spy, pprofile), позволяющие выполнять анализ без остановки программы и с минимальным вмешательством в её работу.

5. Проанализировать возможности профилирования нативных расширений Python (C extensions) с использованием инструментов для отладки и профилирования на уровне языков C/C++ (gprof, Valgrind, perf) и их интеграцию с Python-процессами.

6. Исследовать системные профилировщики (perf, SystemTap, DTrace) и методы их применения для анализа Python-приложений, включая получение смешанных стеков вызовов (Python + C).

7. Сравнить перечисленные средства по ключевым критериям: скорость работы, точность измерений, область применения, накладываемые ограничения и требования к окружению.

8. На основе проведённого анализа сформулировать практические рекомендации по выбору инструментов профилирования для различных сценариев разработки и эксплуатации Python-программ.

2 Профилирование

2.1 Виды профилирования

Профилирование. Определение профилирования везде разное, но в общем смысле *профилирование* - процесс динамического анализа работы кода, направленный на сбор метрик её выполнения. К этим метрикам можно отнести:

- Время выполнения функций, методов, строк;
- Использование оперативной памяти, а именно выделение, освобождение, утечки;
- Частота и количество вызовов функций;
- Загрузка cpu;
- Сколько занимает ввод-вывод.

Полученные данные позволяют выявить узкие места (*bottlenecks*) и обоснованно провести оптимизацию.

2.2 Основные методы профилирования

По способу сбора информации профилировщики делятся на два принципиально разных класса:

1. **Инструментирование** (*tracing* или *instrumentation*).
2. **Статистическое профилирование** (*sampling*).

2.3 Инструментирование (tracing)

Инструментирование — метод динамического анализа, при котором в исходный или бинарный код программы автоматически встраиваются специальные счетчики (зонды) для точного измерения производительности: времени выполнения функций, количества вызовов и потребления памяти.

Достоинства:

- Высокая точность — фиксируется каждый вызов и точное время его выполнения
- Собирается исчерпывающая статистика (количество вызовов, длительность, иерархия).

Недостатки:

- Значительное замедление работы программы (от 2–3 до 10–100 раз в зависимости от языка и реализации)
- Сам факт внедрения зондов может искажать реальную производительность, особенно для короткоживущих функций

2.4 Статистическое (sampling)

Статистическое профилирование - метод анализа производительности ПО, при котором профилировщик через заданные интервалы времени делает снимки состояния программы, определяя выполняемые функции, что позволяет с минимальным влиянием на скорость выявить узкие места. На основе этих снимков строится вероятностная картина распределения времени.
Пример - программа прерывается по таймеру, после чего записываются текущий адрес инструкции или стек вызовов.

Вкратце: при периодическом снимке экрана нет точной, как при tracing, но зато почти не замедляется прогон программы. На особо мелких функциях может очень плохо отрабатывать.

Достоинства:

- Минимальное влияние на скорость работы программы
- Возможность профилировать уже запущенные процессы без их остановки и без доступа к исходному коду.

Недостатки:

- Меньшая точность - кратковременные функции могут быть грубо говоря не замечены.
- Результаты носят статистический характер и требуют репрезентативной выборки.

2.5 Сравнение методов

Выбор между инструментированием и статистическим профилированием зависит от конкретной задачи:

- Если требуется абсолютно точная информация о каждом вызове, то предпочтительно инструментирование.
- Если приложение работает в промышленной среде, замедление недопустимо, или нужно диагностировать проблему на лету, то применяют статистический подход.

В следующих разделах будут рассмотрены конкретные реализации этих методов в экосистеме Python, их особенности и примеры использования.

3 Профилирование на уровне интерпретатора Python

3.1 Инструментирующие профилировщики

Экосистема Python предлагает широкий выбор профилировщиков как в стандартной библиотеке, так и в сторонних пакетах. Среди них:

- profile (стандартная библиотека, Legacy)
- cProfile (стандартная библиотека)
- line_profiler (построчное профилирование)
- memory_profiler (профилирование памяти)
- yappi (многопоточное и асинхронное профилирование)

3.1.1 Profile (Legacy)

Profile - Python-профилировщик, входящий в стандартную библиотеку. Он реализует метод инструментирования (*tracing*) и собирает детальную статистику о вызовах функций, времени их выполнения и количестве обращений.

Реализация

profile написан целиком на Python (pure Python). Это обеспечивает его переносимость, но приводит к значительным накладным расходов. Из-за своей реализации он считается **устаревшим**, так как в стандартную библиотеку добавил cProfile, который написан на C extensions (то есть на языке Си, который дергается из кода Python).

Принцип работы

Профилировщик перехватывает события вызова и возврата из каждой функции с помощью механизма `sys.setprofile()`. На каждый вызов засекается время, ведётся подсчёт количества обращений и строится иерархия вызовов.

```
import profile

def heavy_func():
  total = 0
  for i in range(1000):
  total += i ** 2
  return total

profile.run('heavy_func()', 'profile_stats.prof')
profile.runctx('heavy_func()', globals(), locals(), 'profile_ctx.prof')
```

Для анализа сохранённой статистики используется модуль ***pstats***

```
import pstats
stats = pstats.Stats('profile_stats.prof')
stats.sort_stats('cumtime').print_stats(10)
```

Недостатки profile

1. Так как реализация написана на Python, то средство крайне медленное по сравнению с cProfile и может замедлять прогон кода в десятки раз.

2. Из-за высоких накладных расходов искажается реальная производительность.

3. Не поддерживает профилирование потоков.

4. Поддерживает ручную обработку результатов, что неудобно.

3.1.2 cProfile

cProfile - современный, рекомендуемый tracing профилировщик, входящий в стандартную библиотеку Python начиная с версии 2.5. Он выполняет инструментирование аналогично **profile**, но написан на языке C в виде расширения (C extensions), что обеспечивает на порядок более высокую скорость работы.

Принцип работы

cProfile, аналогично profile, использует перехват событий вызова/возврата через API профилирования виртуальной машины Python, реализованное на C. Благодаря низкоуровневой интеграции накладные расходы существенно снижено

Использование

Запустить профилирование можно тремя способами:

1. **Из командной строки:**

```
python -m cProfile [-o output_file] script.py
```

2. **Внутри скрипта:**

```
import cProfile
cProfile.run('heavy_func()', 'cprofile_stats.prof')
```

3. **Контекстный менеджер (Python 3.8+):**

```
with cProfile.Profile() as profiler:
heavy_func()
profiler.dump_stats('cprofile_context.prof')
```

Анализ результатов

Как и для `profile`, статистика сохраняется в бинарном формате и обрабатывается модулем **pstats**:

```
import pstats
stats = pstats.Stats('cprofile_stats.prof')
#
stats.strip_dirs() # remove absolute paths
stats.sort_stats('tottime') # sorting by your own time
stats.print_stats(10) # bring out the 10 most "heavy" functions
```

Возможно также получение статистики по вызывающим/вызываемым функциям:

```
stats.print_callers(5)
stats.print_callees(5)
```

Визуализация

Бинарные файлы, созданные **cProfile**, могут быть преобразованы в графические отчёты с помощью сторонних утилит:

- **snakeviz** — веб-интерфейс для интерактивного исследования (команда **snakeviz cprofile_stats.pr**
- **gprof2dot** + Graphviz — создание диаграмм в формате DOT (**python -m gprof2dot -f pstats cprofile_stats.prof | dot -Tpng -o output.png**);
- **pyprof2calltree** — конвертация для просмотра в KCachegrind.

Сравнение с profile

- **Скорость: cProfile** работает в 5–20 раз быстрее **profile** (накладные расходы приблизительно 10–30% против 200–1000%).
- **Точность:** за счёт меньшего искажения времени выполнения результаты `cProfile` ближе к реальности.
- **Функциональность:** оба инструмента предоставляют одинаковый набор данных и API, но `cProfile` дополнительно поддерживает профилирование многопоточных программ.
- **Статус:** `cProfile` активно поддерживается и рекомендуется официальной документацией; `profile` сохраняется для обратной совместимости.

Недостатки cProfile

1. Всё ещё ощутимое замедление (от 10% до 50% в зависимости от программы), что ограничивает применение в промышленной эксплуатации.
2. Невозможность профилирования асинхронного кода (`async/await`) без дополнительных ухищрений (для этого существуют специализированные инструменты, например `yappi`).
3. Отсутствие встроенных средств для анализа использования памяти.

Пример использования cProfile: от запуска до визуализации

Рассмотрим полный цикл работы с `cProfile` на примере функции, выполняющей вычисление чисел Фибоначчи рекурсивным и итеративным способами. Это позволит наглядно продемонстрировать сбор статистики, её анализ и построение графического отчёта.

```
def fib_recursive(n):
    if n <= 1:
        return n
    return fib_recursive(n-1) + fib_recursive(n-2)

def fib_iterative(n):
    a, b = 0, 1
    for _ in range(n):
        a, b = b, a + b
    return a

def main():
    fib_recursive(35)
    fib_iterative(100000)

if __name__ == "__main__":
    main()
```

1. Запуск профилирования Выполним профилирование из командной строки, сохранив результаты в файл `fib.prof`:

```
python -m cProfile -o fib.prof fib.py
```

2. Базовый анализ в консоли с помощью pstats Загрузим статистику и выведем 5 самых «тяжёлых» функций по собственному времени (`tottime`):

```
import pstats
stats = pstats.Stats('fib.prof')
stats.strip_dirs()
stats.sort_stats('tottime')
stats.print_stats(5)
```

Результат будет содержать примерно следующую информацию (время и количество вызовов зависят от системы):

```
   ncalls  tottime  percall  cumtime  percall filename:lineno(function)
        1    1.873    1.873    2.569    2.569 fib.py:1(fib_recursive)
331160281   1.695    0.000   1.695    0.000 fib.py:2(fib_recursive)
        1    0.696    0.696    2.569    2.569 fib.py:1(<module>)
        1    0.000    0.000    0.000    0.000 fib.py:6(fib_iterative)
        1    0.000    0.000    0.000    0.000 {method 'disable' ...}
```

Здесь видно, что рекурсивная функция вызвана множество раз и занимает почти всё время, тогда как итеративная выполняется мгновенно. Уже на этом этапе очевидно «узкое место».

3. Интерактивная визуализация с SnakeViz SnakeViz предоставляет браузерный интерфейс для изучения результатов. Установка и запуск:

```
pip install snakeviz
snakeviz fib.prof
```

Откроется диаграмма в виде «солнечных лучей» (sunburst) или иерархического списка, где размер каждого сектора пропорционален времени выполнения. На ней сразу заметен огромный вклад `fib_recursive` и её рекурсивных вызовов.

4. Генерация графа вызовов с gprof2dot Для создания диаграммы в формате PNG, отображающей связи между функциями, используем `gprof2dot` и Graphviz:

```
pip install gprof2dot
python -m gprof2dot -f pstats fib.prof | dot -Tpng -o fib_callgraph.png
```

Полученное изображение наглядно показывает, что `fib_recursive` вызывает саму себя многократно, тогда как `fib_iterative` изолирована и не имеет дочерних вызовов.

5. Интеграция с юнит-тестами Профилировщик можно встроить в тесты для контроля регрессий. Например, используя `unittest`:

```
import cProfile
import unittest

class TestFibonacci(unittest.TestCase):
    def test_performance(self):
        profiler = cProfile.Profile()
        profiler.enable()
        fib_iterative(100000)
        profiler.disable()
        stats = pstats.Stats(profiler)
        stats.sort_stats('cumtime')
        # Checking that the execution time does not exceed the threshold
        self.assertLess(stats.total_tt, 0.01)
```

3.1.3 line_profiler

line_profiler — сторонний инструмент для построчного профилирования кода Python. В отличие от `cProfile`, который показывает время выполнения функций целиком, `line_profiler` измеряет время, затрачиваемое на каждую отдельную строку внутри функции. Это позволяет с высокой точностью локализовать «узкие места» на уровне отдельных операций.

Установка

Устанавливается через `pip`:

```
pip install line_profiler
```

После установки становится доступен исполняемый модуль `kernprof` и сам профилировщик.

Принцип работы

`line_profiler` использует инструментирование исходного кода: он модифицирует байткод функции, вставляя измерительные зонды перед каждой строкой. При выполнении функции замеряется время, прошедшее между зондами. Накладные расходы выше, чем у `cProfile`, но несравнимо более детальная информация оправдывает это при тонкой оптимизации.

Использование

Основной способ применения — декорирование целевой функции `@profile`. Важно: декоратор автоматически добавляется при запуске через `kernprof`, вручную импортировать его не нужно.

```
@profile
def slow_function():
total = 0
for i in range(100000):
total += i ** 2
return total

if name == 'main':
slow_function()
```

Запуск профилирования:

```
kernprof -l -v example.py
```

Ключ `-l` включает построчный режим (line_profiler), `-v` выводит результат сразу в консоль. По умолчанию создаётся файл `example.py.lprof`, который можно просмотреть позже утилитой `python -m line_profiler example.py.lprof`.

Пример вывода

Результат представляет собой таблицу со следующими столбцами:

- `Line` — номер строки;
- `Hits` — количество выполнений данной строки;
- `Time` — общее время, затраченное на строку (в микросекундах);
- `Per Hit` — среднее время одного выполнения;
- `% Time` — доля от общего времени функции;
- `Line Contents` — исходный код строки.

Пример вывода для приведённой функции:

```
Total time: 0.023456 s
File: example.py
Function: slow_function at line 4

Line # Hits Time Per Hit % Time Line Contents
============================================================
4 @profile
5 def slow_function():
6 1 4.0 4.0 0.0 total = 0
7 10001 10234.0 1.0 43.6 for i in range(100000):
8 10000 13218.0 1.3 56.4 total += i ** 2
9 1 0.0 0.0 0.0 return total
```

Видно, что основное время тратится на вычисление квадрата и суммирование внутри цикла.

Достоинства

- Высочайшая детализация — информация по каждой строке.
- Простота использования — достаточно добавить декоратор.
- Интеграция с `IPython` (магическая команда `%lprun`).

Недостатки

- Замедление работы функции в разы (иногда на порядок), что ограничивает применение на реальных данных.
- Необходимость модифицировать исходный код (добавлять декоратор).
- Не работает с асинхронными функциями напрямую.
- Отсутствие встроенной визуализации (но результаты можно экспортировать).

`line_profiler` незаменим, когда нужно понять, какая именно операция внутри большой функции тормозит выполнение, и все другие методы уже не дают достаточной информации.

3.2 Статистические профилировщики

- py_spy - семплирование без остановки процесса
- pprofile - гибридный подход (детерминированный и статистический режимы)

3.2.1 py-spy

py-spy — семплирующий (статистический) профилировщик для Python-программ, написанный на языке Rust. Его ключевая особенность — полная независимость от целевого процесса: `py-spy` запускается вне интерпретатора Python и читает память профилируемой программы напрямую через системные вызовы (`process_vm_readv` в Linux, `vm_read` в macOS, `ReadProcessMemory` в Windows). Это позволяет:

- Профилировать уже запущенные процессы без их остановки;
- Не вносить никаких изменений в исходный код;
- Работать с «боевыми» (production) сервисами без заметного влияния на производительность (оверхед менее 1–2%);
- Собирать данные даже в тех случаях, когда целевой интерпретатор собран без отладочных символов (используется сканирование секции BSS).

Установка

```
pip install py-spy
```

Также доступна установка через пакетные менеджеры: `brew install py-spy` (macOS), `yay -S py-spy` (Arch Linux), а для Rust-экосистемы — `cargo install py-spy`.

Принцип работы

`py-spy` читает память процесса и обходит структуры CPython: находит глобальный `PyInterpreterState`, из него — все активные потоки, а затем и цепочки объектов `PyFrameObject`, формирующих стек вызовов. Поскольку `py-spy` не выполняется внутри интерпретатора, он не перехватывает события вызова/возврата и не замедляет работу кода. Семплирование происходит с заданной частотой (по умолчанию 100 Гц).

Базовые команды

`py-spy` предоставляет три основных режима работы:

1. `record` — запись профиля в файл с последующей визуализацией:

```
# Attach to a running process
py-spy record -o profile.svg --pid 12345

# Start a new process under profiler
py-spy record -o profile.svg -- python myprogram.py
```

   Поддерживаемые форматы: SVG (интерактивный флеймграф), `speedscope`, сырые данные.

2. `top` — интерактивный просмотр «горячих» функций в реальном времени, аналогичный утилите `top`:

```
py-spy top --pid 12345
```

3. `dump` — мгновенный снимок стека вызовов всех потоков (полезно при зависаниях):

```
py-spy dump --pid 12345
py-spy dump --pid 12345 --locals  # also show local variables
```

Ключевые возможности

- **Профилирование нативных расширений** (`-native`) — отображение имён функций из C/C++/Cython (доступно на Linux x86_64 и Windows) .
- **Отслеживание дочерних процессов** (`-subprocesses`) — автоматическое прикрепление к порождённым процессам (например, при использовании `multiprocessing` или `gunicorn`).
- **Фильтрация по GIL** (`-gil`) — учёт только потоков , удерживающих глобальную блокировку интерпретатора.

- **Исключение «спящих» потоков** — `py-spy` опрашивает состояние потоков через `/proc/PID/stat` (Linux), `thread_basic_info` (macOS) или анализ системных вызовов (Windows) и не включает в отчёт стеки, заведомо находящиеся в ожидании. Поведение можно изменить флагом `-idle`.

Особые требования

При профилировании существующего процесса `py-spy` требует прав на чтение чужой памяти. В macOS это всегда требует `sudo`, в Linux — если процесс запущен не самим профилировщиком . В контейнерах Docker и Kubernetes необходимо добавить capability `SYS_PTRACE`, иначе вызовы `process_vm_readv` будут запрещены.

Достоинства

- Нулевое вмешательство в код программы;
- Крайне низкие накладные расходы (применимо в продакшене);
- Работа с уже зависшими процессами (команда `dump`);
- Наглядная визуализация (флеймграфы, `speedscope`);
- Кроссплатформенность (Linux, macOS, Windows, FreeBSD).

Недостатки

- Статистическая природа — кратковременные функции могут быть «пропущены»;
- Требует повышенных привилегий в ряде сценариев;
- Не даёт точного числа вызовов, только долевое распределение времени.

`py-spy` — оптимальный выбор для диагностики замедлений в промышленной эксплуатации, когда невозможно (или нежелательно) перезапускать приложение и вносить изменения в код.

3.2.2 pprofile

pprofile — легковесный, полностью написанный на Python профилировщик , реализующий **оба** метода сбора информации: детерминированное (trace) и статистическое (sample) профилирование. Его главная отличительная черта — *строчная гранулярность* (line-granularity) в сочетании с автоматическим отслеживанием всех потоков и возможностью работы без модификации исходного кода.

Установка

```
pip install pprofile
```

После установки становится доступен одноимённый исполняемый модуль.

Два режима работы

1. Детерминированное профилирование (по умолчанию) В этом режиме `pprofile` перехватывает каждое событие выполнения строки (line events), используя механизм `sys.settrace`. Это даёт абсолютно полную картину: сколько раз выполнилась каждая строка, сколько времени заняла, какова иерархия вызовов. Расплата — огромные накладные расходы (замедление в 10–100 раз), что ограничивает применение только короткими тестовыми сценариями.

```
# Running the script under a deterministic profiler
pprofile myscript.py

# Ignore system paths (so as not to clutter the output)
pprofile --exclude-syspath myscript.py

# Launching the module
pprofile -m mymodule -- --arg-for-module
```

Вывод представляет собой аннотированный исходный код: перед каждой строкой указывается количество попаданий (Hits), общее время, время на один вызов и доля в процентах.

```
File: demo/threads.py
File duration: 1.00168s (99.60%)
Line #|      Hits|         Time|Time per hit|      %|Source code
------+----------+-------------+------------+-------+-----------
     4|         2|  3.21865e-05| 1.60933e-05|  0.00%|    def func():
     5|         1|      1.00111|     1.00111| 99.54%|    time.sleep(1)
```

2. Статистическое профилирование (`-statistic`) В этом режиме `pprofile` периодически опрашивает стеки всех потоков (или только текущего) с заданным интервалом. Накладные расходы снижаются до приемлемых 1–5%, что позволяет применять `pprofile` к длительно работающим приложениям.

```
# Sampling every 0.01 seconds
pprofile --statistic 0.01 myscript.py

# Current stream only (single), 1 ms period
pprofile --statistic 0.001 --single myscript.py
```

Недостаток статистического режима — потеря точности: вместо реального времени выполнения строки показывают только количество «попаданий» в сэмплы, что даёт лишь приблизительную картину.

Поддержка многопоточности

В отличие от `cProfile`, `pprofile` «из коробки» корректно отслеживает порождаемые потоки. В детерминированном режиме для этого необходимо установить флаг `-threads 1` (или `threads=True` в API), в статистическом — все потоки процесса видны автоматически.

Визуализация (Callgrind / KCachegrind)

Одна из сильнейших сторон `pprofile` — экспорт в формат `callgrind`, совместимый с KCachegrind и QCachegrind. Это даёт возможность исследовать профиль в графическом интерфейсе с древовидными диаграммами, поиском «горячих» путей и фильтрацией.

```
pprofile --format callgrind --out cachegrind.out.myprofile myscript.py
# Or short entry (file name starts with cachegrind.out)
pprofile --out cachegrind.out.myprofile myscript.py

# Additionally - package the source code in a zip to display the paths
    correctly
pprofile --out cachegrind.out.myprofile --zipfile src.zip myscript.py
```

Программный API

`pprofile` можно встраивать непосредственно в код, что полезно для профилирования отдельных участков:

```
import pprofile

# Deterministic site profile
prof = pprofile.Profile()
with prof():
    # ... the code for profiling ...
prof.print_stats()

# Statistical profile (sampling the current stream)
stat_prof = pprofile.StatisticalProfile()
with stat_prof(period=0.001):
# ... the code for profiling ...
stat_prof.print_stats()
```

Сравнение с line_profiler

Оба инструмента дают построчную информацию, но принципиально различаются:

- `line_profiler` требует декоратора `@profile` и запуска через `kernprof`, фокусируясь только на отмеченных функциях;
- `pprofile` анализирует *весь* код приложения без каких-либо изменений, что удобно для первичного поиска узких мест, но порождает огромные объёмы вывода.

Достоинства

- Чистый Python — работает там, где нельзя собрать C-расширение;
- Два режима профилирования в одном инструменте;
- Полная поддержка потоков;
- Экспорт в Callgrind для профессионального анализа;
- Не требует модификации исходного кода (в отличие от `line_profiler`).

Недостатки

- Детерминированный режим неприменим к реальным задачам из-за колоссального замедления;
- Статистический режим даёт лишь приблизительные результаты;
- Проект давно не обновлялся (последний релиз — 2016 г.) ;
- Меньше возможностей визуализации по сравнению с `py-spy` (флеймграфы не строятся напрямую).

`pprofile` занимает нишу «тяжёлой артиллерии» для всестороннего исследовательского профилирования в окружениях, где недопустима компиляция или недоступны современные альтернативы. Однако в новых проектах приоритет следует отдавать `py-spy` (продакшен) либо связке `cProfile` + `snakeviz` (разработка).

3.3 Специализированные решения

3.3.1 yappi

yappi (Yet Another Python Profiler) — профилировщик, специализирующийся на многопоточных и асинхронных приложениях. Он поддерживает как инструментирование (`tracing`), так и статистический режим (`wall time` и `cpu time`), и отличается низкими накладными расходами при работе с большим количеством потоков.

Установка

```
pip install yappi
```

Ключевые особенности

- **Поддержка потоков** — корректно учитывает время выполнения каждого потока отдельно.
- **Поддержка asyncio** — профилирование корутин и задач (с версии 1.3.0).
- **Два режима таймера**: `wall-time` (реальное время) и `cpu-time` (процессорное время).
- **Статистическое профилирование** — возможность переключения в режим `sampling` для минимизации накладных расходов.
- **Богатый API** — запуск, остановка, сброс статистики, получение отчётов в различных форматах.

Принцип работы

В режиме `tracing` `yappi` аналогично `cProfile` перехватывает вызовы функций, но реализация оптимизирована для работы в многопоточной среде (использует локальные для потока структуры данных). В режиме `sampling` профилировщик периодически опрашивает стеки всех потоков, что даёт минимальную задержку.

Использование

Рассмотрим пример профилирования многопоточной программы.

```
import yappi
import threading
import time

def worker(name):
for _ in range(5):
time.sleep(0.1)
print(f"{name} working")

threads = []
for i in range(3):
t = threading.Thread(target=worker, args=(f"Thread-{i}",))
threads.append(t)

yappi.set_clock_type("wall") # "wall" or "cpu"
yappi.start()
for t in threads:
t.start()
for t in threads:
t.join()
yappi.stop()

func_stats = yappi.get_func_stats()
func_stats.sort("ttot", type="cum").print_all()

thread_stats = yappi.get_thread_stats()
thread_stats.print_all()

func_stats.save('yappi_stats.prof', type='pstats')
```

Пример с asyncio

Начиная с версии 1.3.0, yappi может профилировать асинхронный код.

```
import yappi
import asyncio

async def fetch_data():
await asyncio.sleep(0.1)
return "data"

async def main():
results = await asyncio.gather(fetch_data(), fetch_data(), fetch_data())

yappi.set_clock_type("wall")
yappi.start()
asyncio.run(main())
```

```
14 yappi.stop()
15
16 stats = yappi.get_func_stats()
17 stats.print_all()
```

Сравнение с cProfile

- **Потоки:** `cProfile` не разделяет статистику по потокам — все вызовы смешиваются; `yappi` предоставляет отдельную статистику для каждого потока.
- **Asyncio:** `cProfile` видит только вызовы функций, но не корутины; `yappi` корректно отображает время выполнения асинхронных задач.
- **Скорость:** В режиме `tracing` `yappi` немного медленнее `cProfile` из-за дополнительных блокировок; в режиме `sampling` — значительно быстрее.
- **Гибкость:** `yappi` позволяет динамически переключать режимы, очищать статистику и профилировать только интересующие потоки.

Достоинства

- Полноценная поддержка многопоточности и asyncio.
- Два режима профилирования (инструментирование и семплирование).
- Низкие накладные расходы в режиме `sampling` (подходит для продакшена).
- Богатый API и возможность экспорта в формат `pstats`.
- Поддержка профилирования времени ЦП и реального времени.

Недостатки

- Менее распространён, чем `cProfile`, меньше интеграций с визуализаторами (хотя файлы `pstats` совместимы).
- В режиме `tracing` может быть медленнее `cProfile` для однопоточных приложений.
- Документация не всегда подробна, хотя проект активно развивается.

`yappi` становится незаменимым инструментом при разработке высоконагруженных сетевых сервисов, веб-приложений на asyncio, а также при необходимости профилирования в условиях, когда нельзя останавливать сервер (режим семплирования).

4 Профилирование нативных расширений (C extensions)

4.1 Особенности профилирование кода на C/C++ внутри Python

4.2 Инструменты для анализа C/C++: gprof, Valgrind (Callgrind), Google PerfTools

4.3 Получение смешанных стеков вызовов (Python + C) с помощью perf и eBPF

5 Системное профилирование Python-приложений

5.1 Использование утилиты perf в Linux для анализа на уровне ядра

5.2 DTrace и SystemTap: динамическая трассировка

5.3 Профилирование операций ввода-вывода и работы с сетью

6 Сравнительный анализ и практические рекомендации