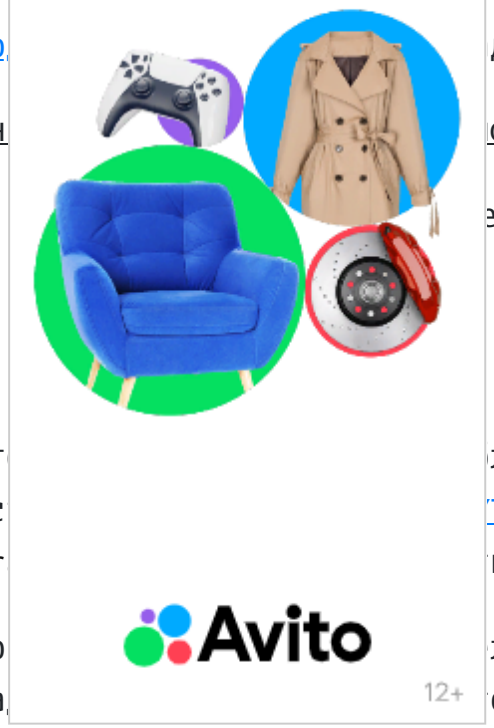
Отслеживания блоков памяти, выделенных Python

Ст / Отслеживания блоков памяти, выделенных Python



Мо дочный инструмент для отслеживания блоков памяти, выделенных программе на Python.

Он формацию:

е был выделен объект.
блокам памяти по имени файла и по номеру строки: общий размер, количество и средний
памяти.
двумя моментальными снимками для обнаружения утечек памяти.

Чт локов памяти, выделенных Python, модуль должен быть запущен как можно раньше,
ус THONTRACEMALLOC в 1 или используя параметр командной строки - X tracemalloc. Функция
tr ь вызвана во время выполнения, чтобы начать трассировку выделения памяти Python.

По ленного блока памяти хранит только самый последний кадр (1 кадр). Чтобы сохранить 25
ка е для переменной среды PYTHONTRACEMALLOC значение 25 или используйте параметр
командной строки - X tracemalloc=25.

Примеры использования модуля tracemalloc:

- 10 файлов, потребляющих наибольшую память;
- Вычисление различии между снимками памяти;
- Получим трассировку блока памяти;
- Красивый вывод трассировки;
- Запись всех отслеживаемых блоков памяти.

Отображение 10 файлов, потребляющих наибольшую память:

```
import tracemalloc

tracemalloc.start()

# ... запуск приложения ...

snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('lineno')

print("[ Top 10 ]")
for stat in top_stats[:10]:
    print(stat)

# Пример вывода статистики потребления памяти Python:
# [ Top 10 ]
# <frozen importlib._bootstrap>:716: size=4855 KiB, count=39328, average=126 B
# <frozen importlib._bootstrap>:284: size=521 KiB, count=3199, average=167 B
# /usr/lib/python3.8/collections/__init__.py:368: size=244 KiB, count=2315, average=108 B
# /usr/lib/python3.8/unittest/case.py:381: size=185 KiB, count=779, average=243 B
# /usr/lib/python3.8/unittest/case.py:402: size=154 KiB, count=378, average=416 B
# b/python3.8/abc.py:133: size=88.7 KiB, count=347, average=262 B
# importlib._bootstrap>:1446: size=70.4 KiB, count=911, average=79 B
# <frozen importlib._bootstrap>:1454: size=52.0 KiB, count=25, average=2131 B
```

```
# <string>:5: size=49.7 KiB, count=148, average=344 B
# /usr/lib/python3.8/sysconfig.py:411: size=48.0 KiB, count=1, average=48.0 KiB
```

Видно, что Python загрузил 4855 КиБ данных (байт-код и константы) из модулей и что модуль collections]m-collections выделил 244 КиБ для создания collections.namedtuple кортежей.

Вычисление различии между снимками памяти.

Сделаем два снимка и покажем различия:

```
import tracemalloc
tracemalloc.start()
# ... запуск приложения ...

snapshot1 = tracemalloc.take_snapshot()
# ... вызов функции с утечкой памяти ...
snapshot2 = tracemalloc.take_snapshot()

top_stats = snapshot2.compare_to(snapshot1, 'lineno')

print("[ Top 10 differences ]")
for stat in top_stats[:10]:
    print(stat)

# Пример вывода до/после запуска некоторых тестов Python:
# [ Top 10 differences ]
# <frozen importlib._bootstrap>:716: size=8173 KiB (+4428 KiB), count=71332 (+39369), average=117 B
# /usr/lib/python3.4/linecache.py:127: size=940 KiB (+940 KiB), count=8106 (+8106), average=119 B
# /usr/lib/python3.4/unittest/case.py:571: size=298 KiB (+298 KiB), count=589 (+589), average=519 B
# <frozen importlib._bootstrap>:284: size=1005 KiB (+166 KiB), count=7423 (+1526), average=139 B
# /usr/lib/python3.4/mimetypes.py:217: size=112 KiB (+112 KiB), count=1334 (+1334), average=86 B
# /usr/lib/python3.4/http/server.py:848: size=96.0 KiB (+96.0 KiB), count=1 (+1), average=96.0 KiB
# /usr/lib/python3.4/inspect.py:1465: size=83.5 KiB (+83.5 KiB), count=109 (+109), average=784 B
# /usr/lib/python3.4/unittest/mock.py:491: size=77.7 KiB (+77.7 KiB), count=143 (+143), average=557 B
# /usr/lib/python3.4/urllib/parse.py:476: size=71.8 KiB (+71.8 KiB), count=969 (+969), average=76 B
# /usr/lib/python3.4/contextlib.py:38: size=67.2 KiB (+67.2 KiB), count=126 (+126), average=546 B
```

Видно, что Python загрузил 8173 КиБ данных модуля (байт-код и константы), и что это на 4428 КиБ больше, чем было загружено до тестов, когда был сделан предыдущий снимок. Точно так же модуль linecache кэшировал 940 КиБ исходного кода Python для форматирования трассировок, причем все это с момента предыдущего снимка.

Если в системе мало свободной памяти, снимки можно записывать на диск с помощью метода Snapshot.dump() модуля tracemalloc для анализа снимка в автономном режиме. Затем можно использовать метод Snapshot.load(), чтобы перезагрузить снимок.

Получим трассировку блока памяти.

Код для отображения трассировки самого большого блока памяти:

```
import tracemalloc

# Store 25 frames
tracemalloc.start(25)

# ... запуск приложения ...

snapshot = tracemalloc.take_snapshot()
top_stats = snapshot.statistics('traceback')

# выберем самый большой блок памяти
stat = top_stats[0]
print("%s memory blocks: %.1f KiB" % (stat.count, stat.size / 1024))
for line in stat.traceback.format():
    print(line)

# Вверх вывода набора тестов Python (traceback ограничен 25 кадрами):
# 903 memory blocks: 870.1 KiB
```

```
#   File "<frozen importlib._bootstrap>", line 716
#   File "<frozen importlib._bootstrap>", line 1036
#   File "<frozen importlib._bootstrap>", line 934
#   File "<frozen importlib._bootstrap>", line 1068
#   File "<frozen importlib._bootstrap>", line 619
#   File "<frozen importlib._bootstrap>", line 1581
#   File "<frozen importlib._bootstrap>", line 1614
#   File "/usr/lib/python3.4/doctest.py", line 101
#     import pdb
#   File "<frozen importlib._bootstrap>", line 284
#   File "<frozen importlib._bootstrap>", line 938
#   File "<frozen importlib._bootstrap>", line 1068
#   File "<frozen importlib._bootstrap>", line 619
#   File "<frozen importlib._bootstrap>", line 1581
#   File "<frozen importlib._bootstrap>", line 1614
#   File "/usr/lib/python3.4/test/support/__init__.py", line 1728
#     import doctest
#   File "/usr/lib/python3.4/test/test_pickletools.py", line 21
#     support.run_doctest(pickletools)
#   File "/usr/lib/python3.4/test/regrtest.py", line 1276
#     test_runner()
#   File "/usr/lib/python3.4/test/regrtest.py", line 976
#     display_failure=not verbose)
#   File "/usr/lib/python3.4/test/regrtest.py", line 761
#     match_tests=ns.match_tests)
#   File "/usr/lib/python3.4/test/regrtest.py", line 1563
#     main()
#   File "/usr/lib/python3.4/test/__main__.py", line 3
#     regrtest.main_in_temp_cwd()
#   File "/usr/lib/python3.4/runpy.py", line 73
#     exec(code, run_globals)
#   File "/usr/lib/python3.4/runpy.py", line 160
#     "__main__", fname, loader, pkg_name)
```

Видно, что больше всего памяти было выделено в модуле importlib для загрузки данных (байт-кода и констант) из модулей: 870,1 КБ. Обратная трассировка - это то место, где importlib загрузил данные совсем недавно: в строке import pdb модуля doctest. Обратная связь может измениться, если будет загружен новый модуль.

Отображение 10 строк, выделяющих большую часть памяти с красивым выводом.

Код для отображения 10 строк, выделяющих большую часть памяти с красивым выводом, игнорируя файлы <frozen importlib._bootstrap> и <unknown> :

```
import linecache
import os
import tracemalloc

def display_top(snapshot, key_type='lineno', limit=10):
    snapshot = snapshot.filter_traces((
        tracemalloc.Filter(False, "<frozen importlib._bootstrap>"),
        tracemalloc.Filter(False, "<unknown>"),
    ))
    top_stats = snapshot.statistics(key_type)

    print("Top %s lines" % limit)
    for index, stat in enumerate(top_stats[:limit], 1):
        frame = stat.traceback[0]
        print("#%s: %s:%s: %.1f KiB"
              % (index, frame.filename, frame.lineno, stat.size / 1024))
        line = linecache.getline(frame.filename, frame.lineno).strip()
        if line:
            print('    %s' % line)

    other = top_stats[limit:]
    if other:
        size = sum(stat.size for stat in other)
        print("%s other: %.1f KiB" % (len(other), size / 1024))
    total = sum(stat.size for stat in top_stats)
```

```
    print("Total allocated size: %.1f KiB" % (total / 1024))

tracemalloc.start()

# запуск приложения ...

snapshot = tracemalloc.take_snapshot()
display_top(snapshot)


# Пример вывода набора тестов Python:
# Top 10 lines
# 1: Lib/base64.py:414: 419.8 KiB
#     _b85chars2 = [(a + b) for a in _b85chars for b in _b85chars]
# 2: Lib/base64.py:306: 419.8 KiB
#     _a85chars2 = [(a + b) for a in _a85chars for b in _a85chars]
# 3: collections/__init__.py:368: 293.6 KiB
#     exec(class_definition, namespace)
# 4: Lib/abc.py:133: 115.2 KiB
#     cls = super().__new__(mcls, name, bases, namespace)
# 5: unittest/case.py:574: 103.1 KiB
#     testMethod()
# 6: Lib/linecache.py:127: 95.4 KiB
#     lines = fp.readlines()
# 7: urllib/parse.py:476: 71.8 KiB
#     for a in _hexdig for b in _hexdig}
# 8: <string>:5: 62.0 KiB
# 9: Lib/_weakrefset.py:37: 60.0 KiB
#     self.data = set()
# 10: Lib/base64.py:142: 59.8 KiB
#     _b32tab2 = [a + b for a in _b32tab for b in _b32tab]
# 6220 other: 3602.8 KiB
# Total allocated size: 5303.1 KiB
```

Запись текущего и максимального размера всех отслеживаемых блоков памяти.

Следующий код неэффективно вычисляет две суммы типа 0 + 1 + 2 + ..., создавая список этих чисел. Этот список временно занимает много памяти.

Можно использовать tracemalloc.get_traced_memory() и tracemalloc.reset_peak(), чтобы наблюдать небольшое использование памяти после вычисления суммы, а также пиковое использование памяти во время вычислений:

```
import tracemalloc

tracemalloc.start()

# Пример кода: вычислите сумму с большим временным списком
large_sum = sum(list(range(100000)))

first_size, first_peak = tracemalloc.get_traced_memory()

tracemalloc.reset_peak()

# Пример кода: вычислите сумму с небольшим временным списком
small_sum = sum(list(range(1000)))

second_size, second_peak = tracemalloc.get_traced_memory()

print(f"{first_size=}, {first_peak=}")
print(f"{second_size=}, {second_peak=}")

# Вывод
# first_size=664, first_peak=3592984
# second_size=804, second_peak=29704
```

Вверх

Использование tracemalloc.reset_peak() гарантирует, что можно точно записать пик во время вычисления small_sum, даже если он намного меньше, чем общий максимальный размер блоков памяти с момента вызова tracemalloc.start(). Без вызова tracemalloc.reset_peak(), переменная second_peak все равно была бы пиком вычисления large_sum (то есть равным first_peak). В этом случае оба пика намного выше, чем окончательное использование памяти, и это предполагает, что мы можем произвести оптимизацию кода, удалив ненужный вызов list и написав sum(range(...))).