Профилирование в Erlang

Перечень доступных инструментов:

- fprof, eprof, cover, cprof (входят в стандартную поставку)
- top-like утилиты: сторонний https://github.com/zaa/entop и http://erlang.org/doc/apps/observer/etop_ug.html
- https://github.com/virtan/eep/
- https://ferd.github.io/recon/
- https://github.com/proger/eflame и https://github.com/slfritchie/eflame (eflame2)
- https://github.com/massemanet/eper
- timer:tc/3, statistics/1
- latency_histogram_tracer.erl https://gist.github.com/slfritchie/159a8ce1f49fc03c77c6 (требует наличия `folsom_metrics`)

Примечание: здесь не приведены инструменты, которыми пользуются в редких случаях (http://erlang.org/doc/man/percept.html), или на основе которых строят более мощные штуки (http://erlang.org/doc/man/ttb.html).

Введение

Обзор всех инструментов отнял бы очень много времени. Да и, я сомневаюсь, что обычному Erlang программисту могут понадобиться все вышеперечисленные инструменты.

Гораздо важнее знать пару-тройку, и на основании результатов, полученных в ходе профилирования приложения, сделать выводы: что тормозит, насколько, будем ли оптимизировать.

fprof, cprof

Как ни крути, один или два стандартных инструмента надо знать.

Лучшее место, чтобы начать знакомиться со стандартными инструментами - http://erlang.org/doc/efficiency_guide/profiling.html

Вывод fprof можно преобразовать в формат, пригодный для kcachegrind (читай ниже) с помощью https://github.com/isacssouza/erlgrind.

```
1> fprof:start().
2> fprof:trace([start, {procs, all}]). % трейсим все процессы (существенно замедляет систему)
3> fprof:trace([stop]).
4> fprof:profile().
5> fprof:analyse([totals, {dest, "fprof.analysis"}]).
6> fprof:stop().
```

Пояснения по каждой команде можно найти в документации и в https://timanovsky.wordpress.com/2009/01/20/profiling-running-erlang-server/.

```
1> cprof:start().
2> cprof:pause().
3> cprof:analyse().
4> cprof:analyse(cprof). % анализируем вызовы отдельного модуля
5> cprof:stop().
```

eep

Последовательность необходимых шагов довольно проста и описана в readme проекта.

Мы запускаем профайлер:

```
1> eep:start_file_tracing("appname"), timer:sleep(10000), eep:stop_tracing().
```

Потом копируем файл (предварительно пожав его; у меня результирующий `.trace` доходил до 3GB) и трансформируем его в текстовый формат для kcachegrind https://kcachegrind.github.io/html/Home.html.

```
1> eep:convert_tracing("appname").
```

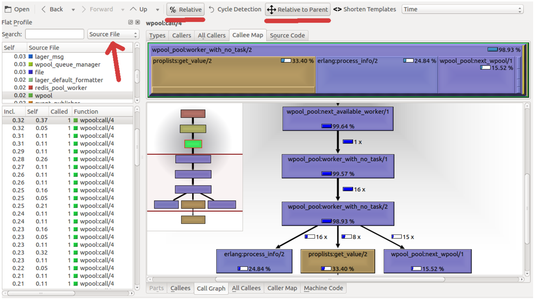
Если вы еще не установили kcachegrind, давайте сделаем это:

В Ubuntu:

```
$ sudo apt-get install kcachegrind
```

После того, как конвертация закончилась, можно начать проводить анализ в kcachegrind.

```
$ kcachegrind callgrind.out.appname
```



3 важных момента:

- группировать лучше по "Source File" (ещё может быть полезна группировка по "ELF Object" если вы нацелены на конкретный PID);
- лучше все же смотреть на проценты ("Relative");
- относительно родителя ("Relative to Parent"), иначе ни черта не понятно.

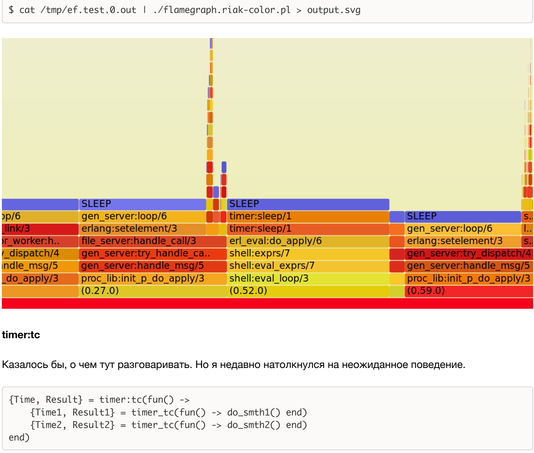
"No Grouping" или "Function Cycle" не дают картины происходящего. Например первая опция приводит к верхним строчкам из разряда gen_server:decode_msg, gen_server:loop, и свидетельствует лишь о том, что большее время приложение проводит в gen_server:" - да я это и так знаю, спасибо капитан очевидность! Вторая опция приводит к непонятным результатам.

eflame2

```
1> eflame2:write_trace(global_calls_plus_new_procs, "/tmp/ef.test.0", all, 10*1000).
2> eflame2:format_trace("/tmp/ef.test.0", "/tmp/ef.test.0.out").
```

Согласно документации, нужно запустить отдельный процесс. Но, когда я пробовал так делать, получал "Segmentation fault".

```
$ cat /tmp/ef.test.0.out | ./flamegraph.riak-color.pl > output.svg
```



timer:tc

Казалось бы, о чем тут разговаривать. Но я недавно натолкнулся на неожиданное поведение.

```
{Time, Result} = timer:tc(fun() ->
    {Time1, Result1} = timer_tc(fun() -> do_smth1() end)
    {Time2, Result2} = timer_tc(fun() -> do_smth2() end)
end)
```

Time должен быть равен Time1 + Time2, верно?

Нет, не в этой реальности. **Time зачастую намного больше Time1 + Time2** (даже если мерить N=100 раз и более). Я думаю, это происходит потому что timer:tc использует os:timestamp и в Time "попадают" всякие переключения (шедулер запускает другой поток).

Полезные ссылки:

- Actively measuring and profiling Erlang code by S. L. Fritchie http://www.snookles.com/erlang/ef2015/slf-presentation.html
- Profiling Erlang Apps using Redbug (using eper) http://roberto-aloi.com/erlang/profiling-erlang-applications-using-redbug/

Copyright 2016 @ Anton Kalyaev