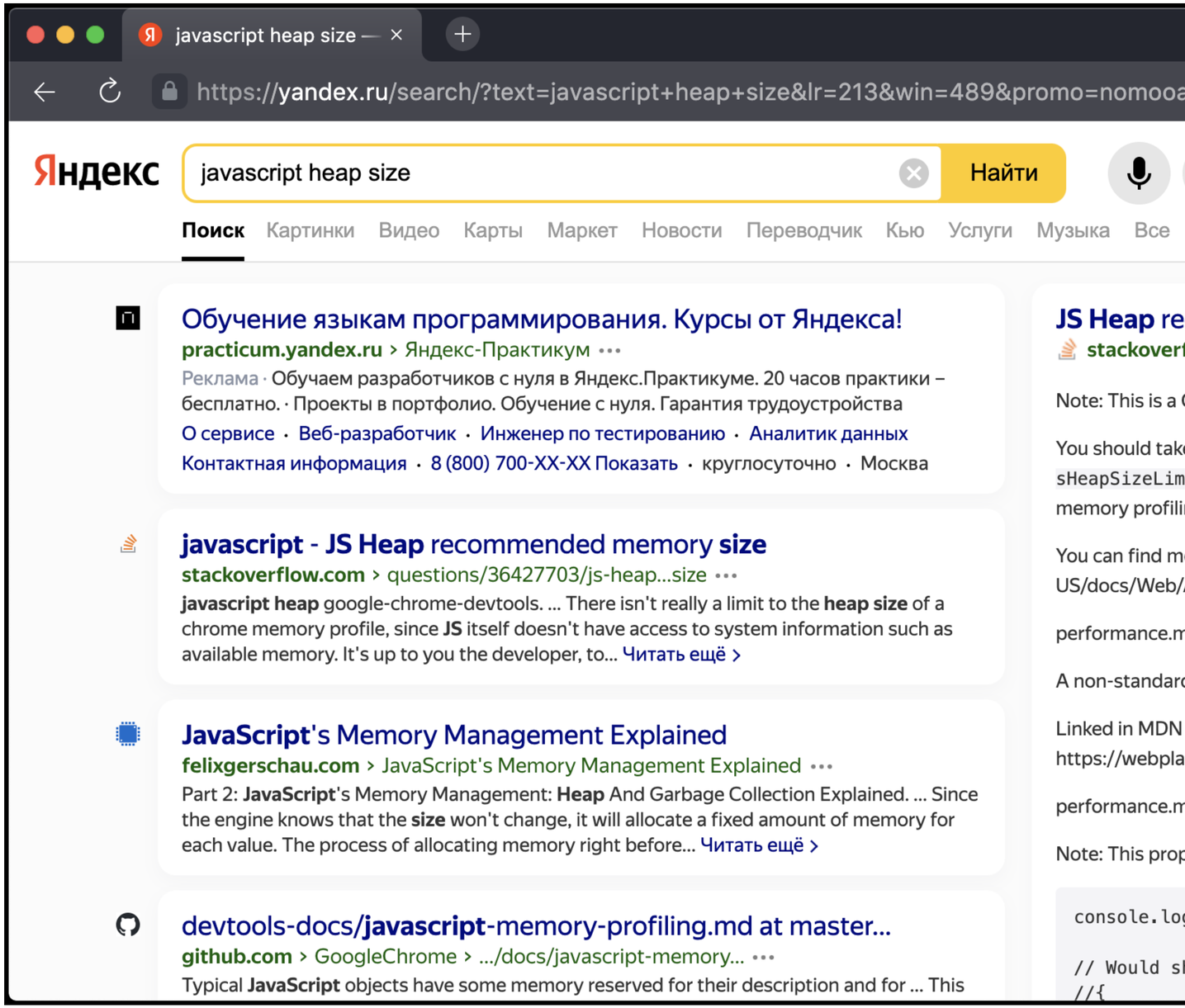
Ускорение приложений на Node: когда стандартного профайлера недостаточно

Виктор Хомяков

О себе

- 4 года в разработке страницы результатов поиска Яндекса
- команда Скорости



Инструментирование и семплирование

Инструментирование vs семплирование

Похоже на code coverage

`console.time … console.timeEnd`

`start=Date.now() … end=Date.now()`

Было в IE, в JProfiler (Java)

- ✅ Точное кол-во вызовов
- ✅ Видна сложность алгоритма
- ⛔ Время на инструментирование (как в code coverage)
- ⛔ Overhead на мелких функциях (решено в JProfiler)

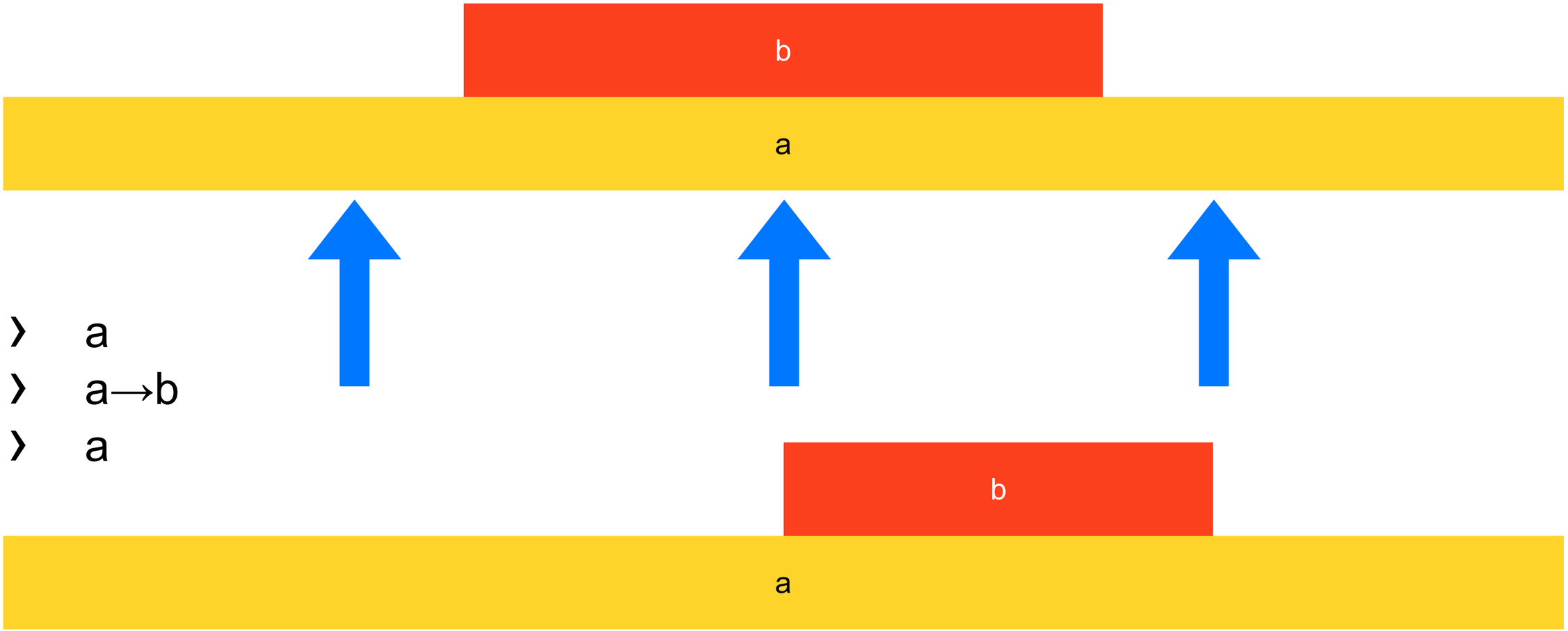
Все существующие профайлеры JS

Собираем стеки с заданной частотой

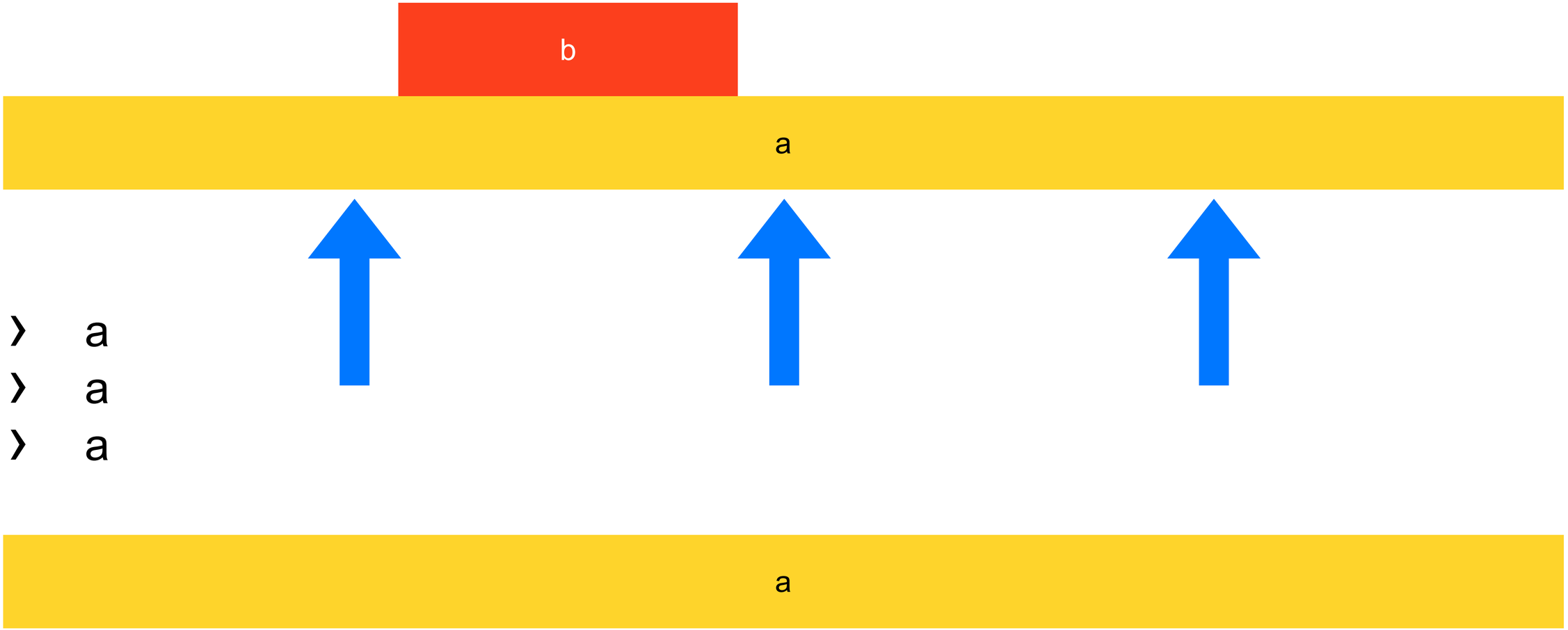
- ⛔ Приблизительное кол-во вызовов
- ⛔ Больше семплов — ближе к правде
- ✅ Не нужно инструментирование кода

Семплирование

- a
- a→b
- a

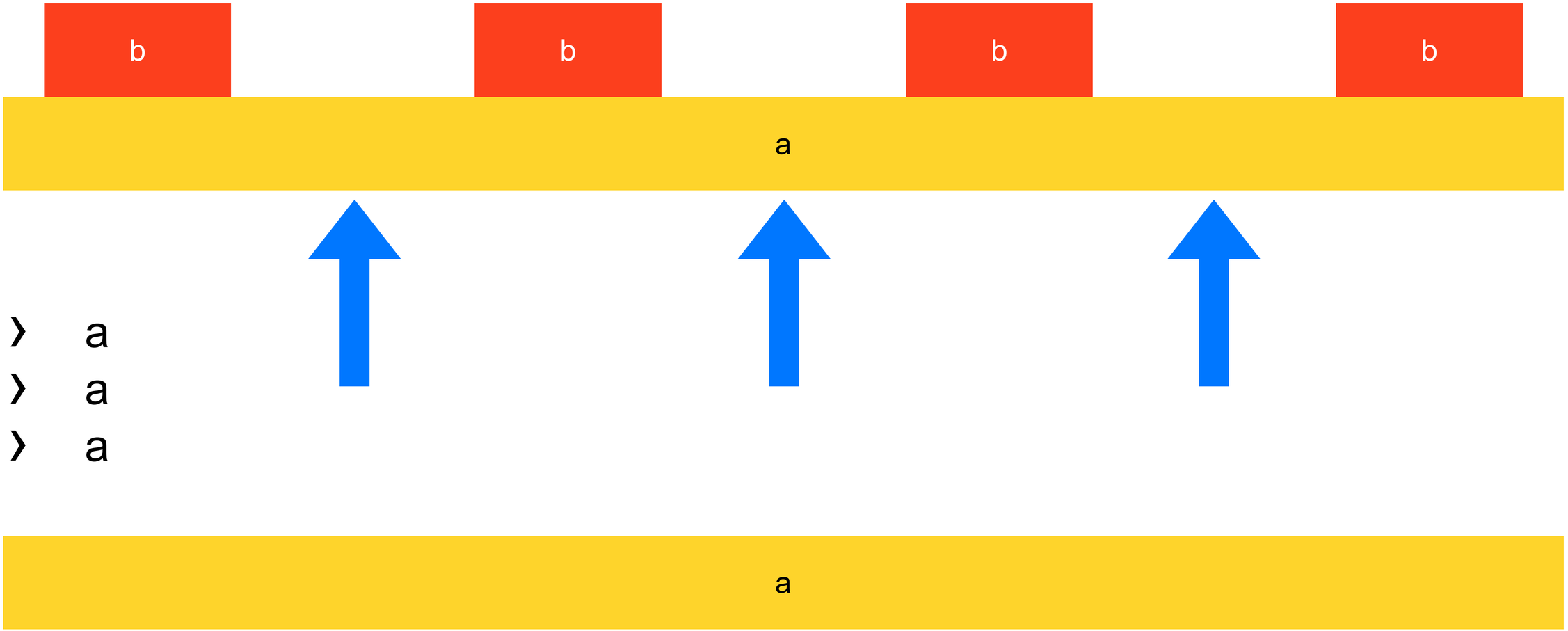


Быстрая функция



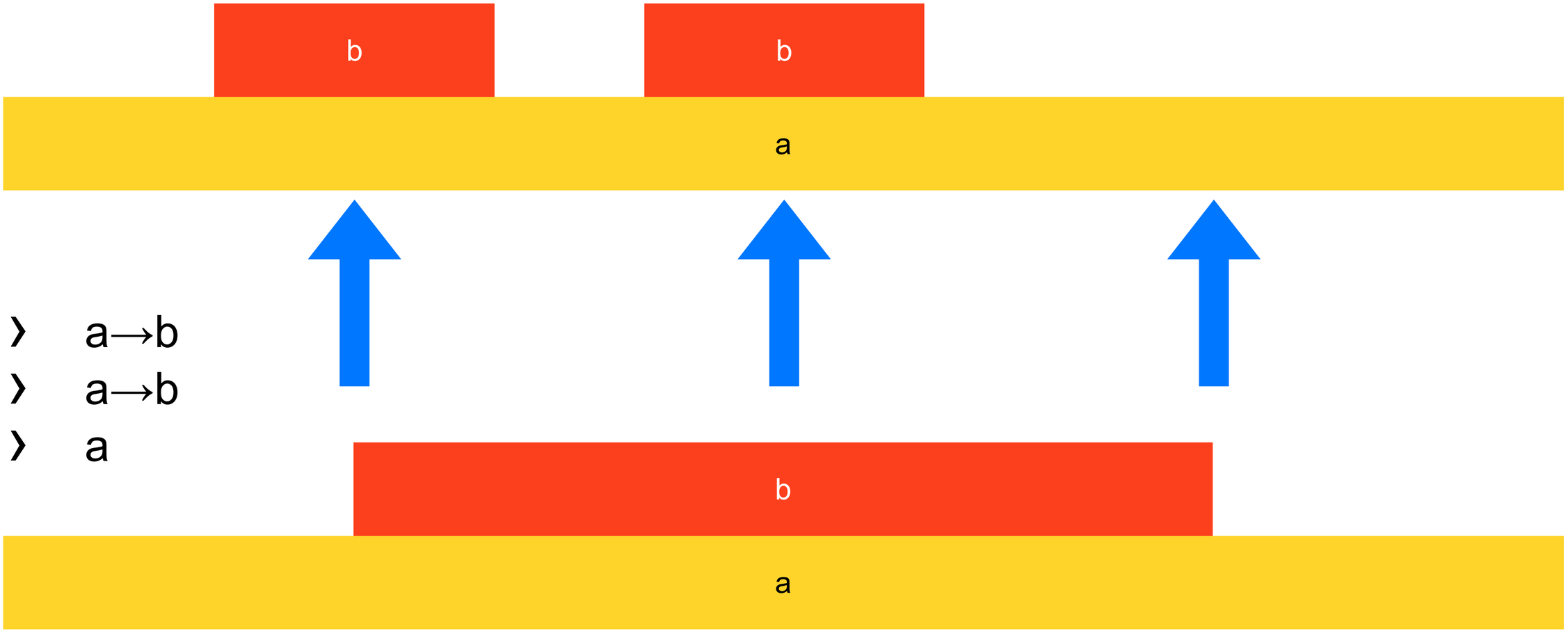
- a
- a
- a

Периодическая функция



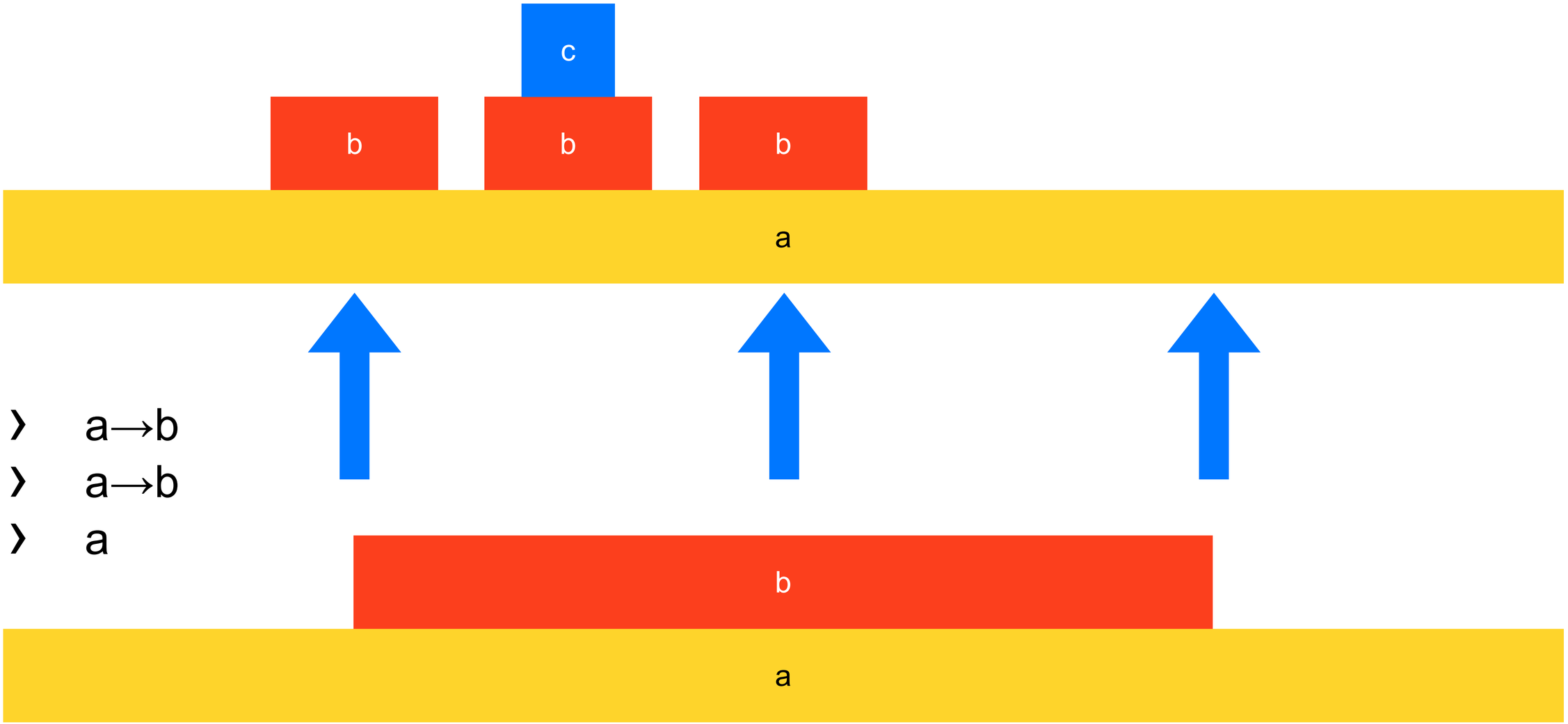
- a
- a
- a

Объединение семплов



- a→b
- a→b
- a

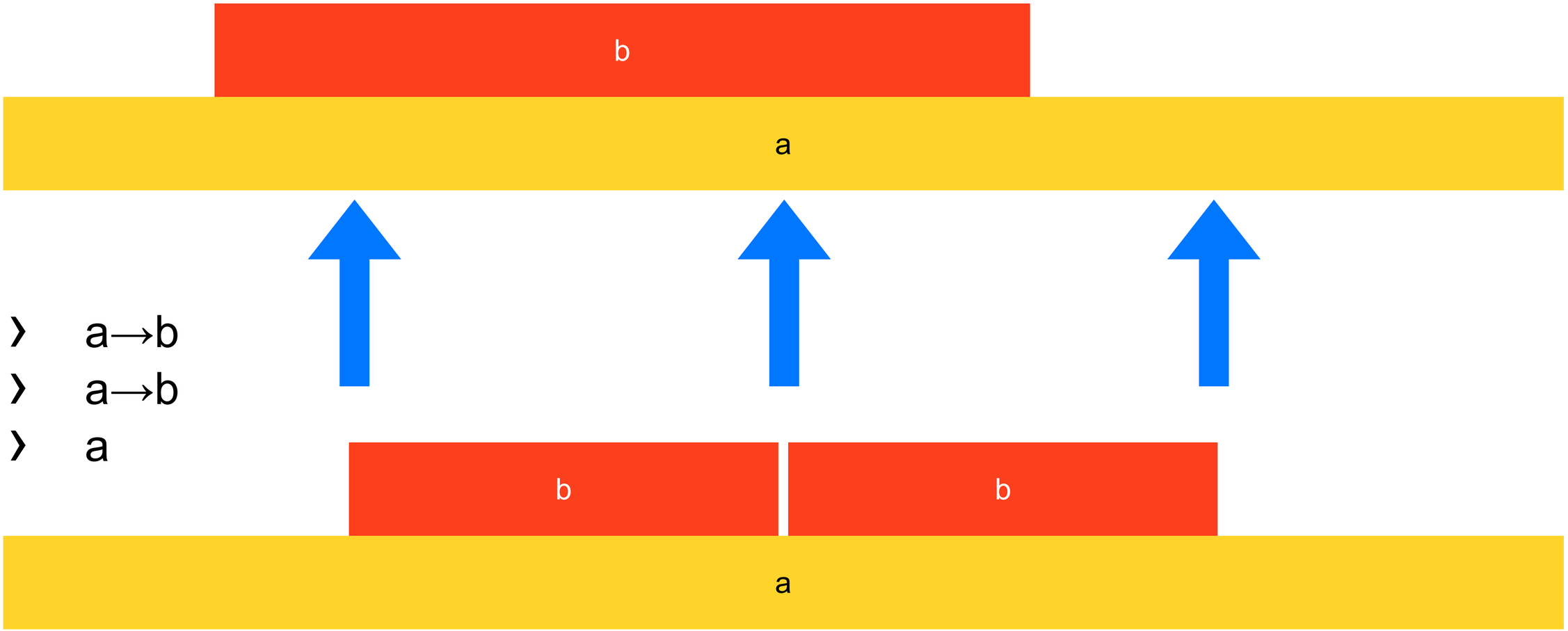
Объединение семплов + пропуск



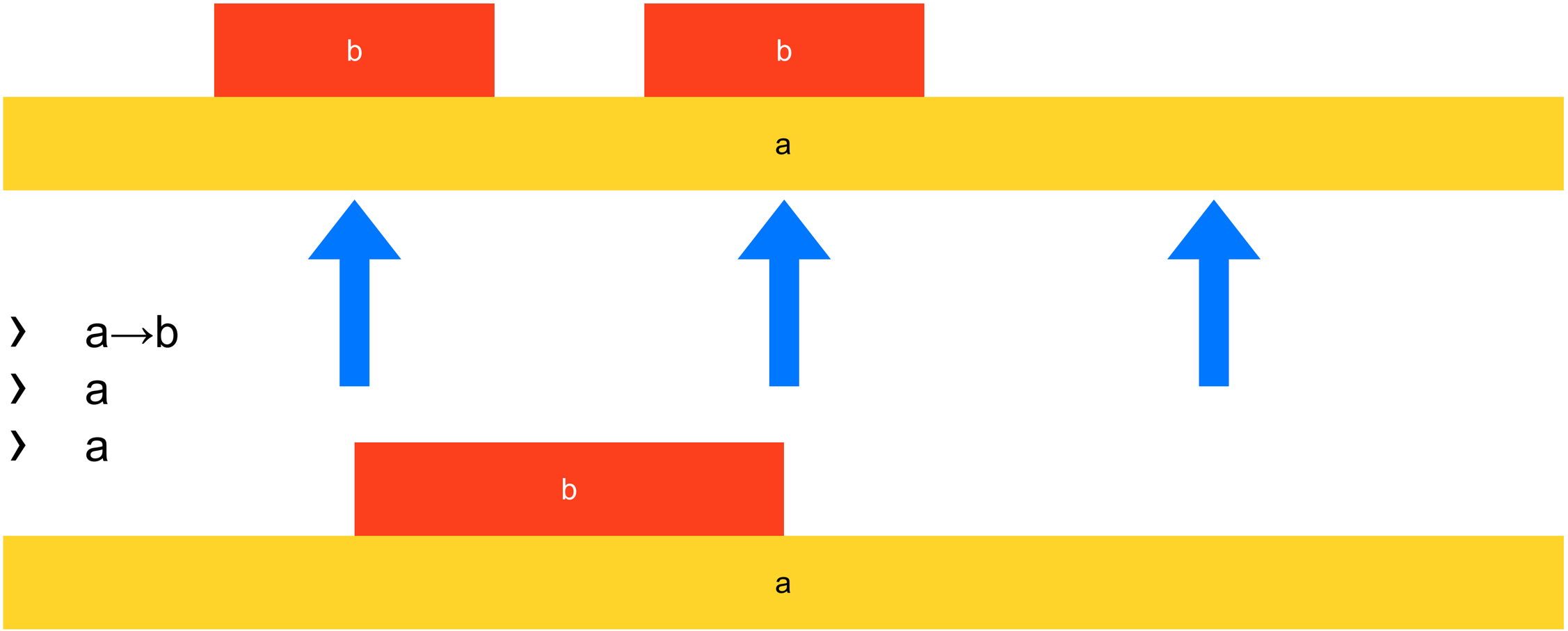
- a→b
- a→b
- a

Разделение семплов

- a→b
- a→b
- a



Инлайнинг



- a→b
- a
- a

Как мы профилируем Node.js

Профилирование Node:

Два встроенных профайлера:

- --prof
- --cpu-prof

Профилирование Node: --prof

- node --prof index.js
- получаем isolate-*-*-v8.log
- node --prof-process isolate-12345-v8.log > isolate-12345-v8.txt

Профилирование Node: --cpu-prof

- node --cpu-prof index.js
- получаем CPU.*.cpuprofile
- открываем в DevTools
- node --cpu-prof-interval=<интервал в микросекундах>
- 1000мкс (1мс) по умолчанию
- API node:inspector

Встроенный профайлер

- ⛔ Профиль после завершения node (если профилируем не из API)
- ⛔ Не в проде сложно воспроизвести продовые условия и нагрузку
- ⛔ В проде overhead, node под нагрузкой таймаутит или дохнет
- ⛔ В профиле только JS, не видно вызовы стандартных API

Perf: общая информация

Профилировщик perf

- встроен в ядро Linux
- код находится в репозитории ядра Linux
- доступен сразу после установки дистрибутива
- или ставится из пакета linux-tools-common или perf
- семплирующий профилировщик

1. Запись семплов

```
perf record -F 999 -p <PID> -e cycles -g -- sleep 100
perf record -F 999 -e cycles -g -- node index.js
```

- -F — частота семплирования, Гц
- -p — PID процесса
- -e — какие события считать
- -g — надо записывать стектрейсы
- perf с заданной частотой получает стектрейсы потоков процесса
- стектрейс — массив адресов в памяти
- пишет в файл perf.data

Примеры

```
perf record -F 999 -p <PID> -e cycles -g -- sleep 100
perf record -F 999 -e cycles -g -- node index.js
sudo perf record -F 999 -ag -p `pgrep -n node`
perf record -F 999 -g -p $(pgrep -x -n node) -- sleep 100
sudo perf record -e cycles:u -g -- npm run watch
```

-F — частота семплирования

```
perf … -F 999 …
perf … -F max …
```

```
cat /proc/sys/kernel/perf_event_max_sample_rate
```

max у меня 6000 (6кГц)

увеличение максимальной частоты:

```
sudo sysctl kernel.perf_cpu_time_max_percent=70
sudo sysctl kernel.perf_event_max_sample_rate=300000
```

https://youtu.be/Cn4bw7cE68w?t=662

-e — события

```
perf list

List of pre-defined events (to be used in -e):
  cache-misses                                       [Hardware event]
  cache-references                                   [Hardware event]
  cpu-cycles OR cycles                               [Hardware event]
  instructions                                       [Hardware event]
  context-switches OR cs                             [Software event]
  cpu-migrations OR migrations                       [Software event]
  major-faults                                       [Software event]
  minor-faults                                       [Software event]
  page-faults OR faults                              [Software event]
```

https://perfwiki.github.io/main/tutorial/#events

-e — события

Самое частое: `-e cycles` или `-e cycles:u`

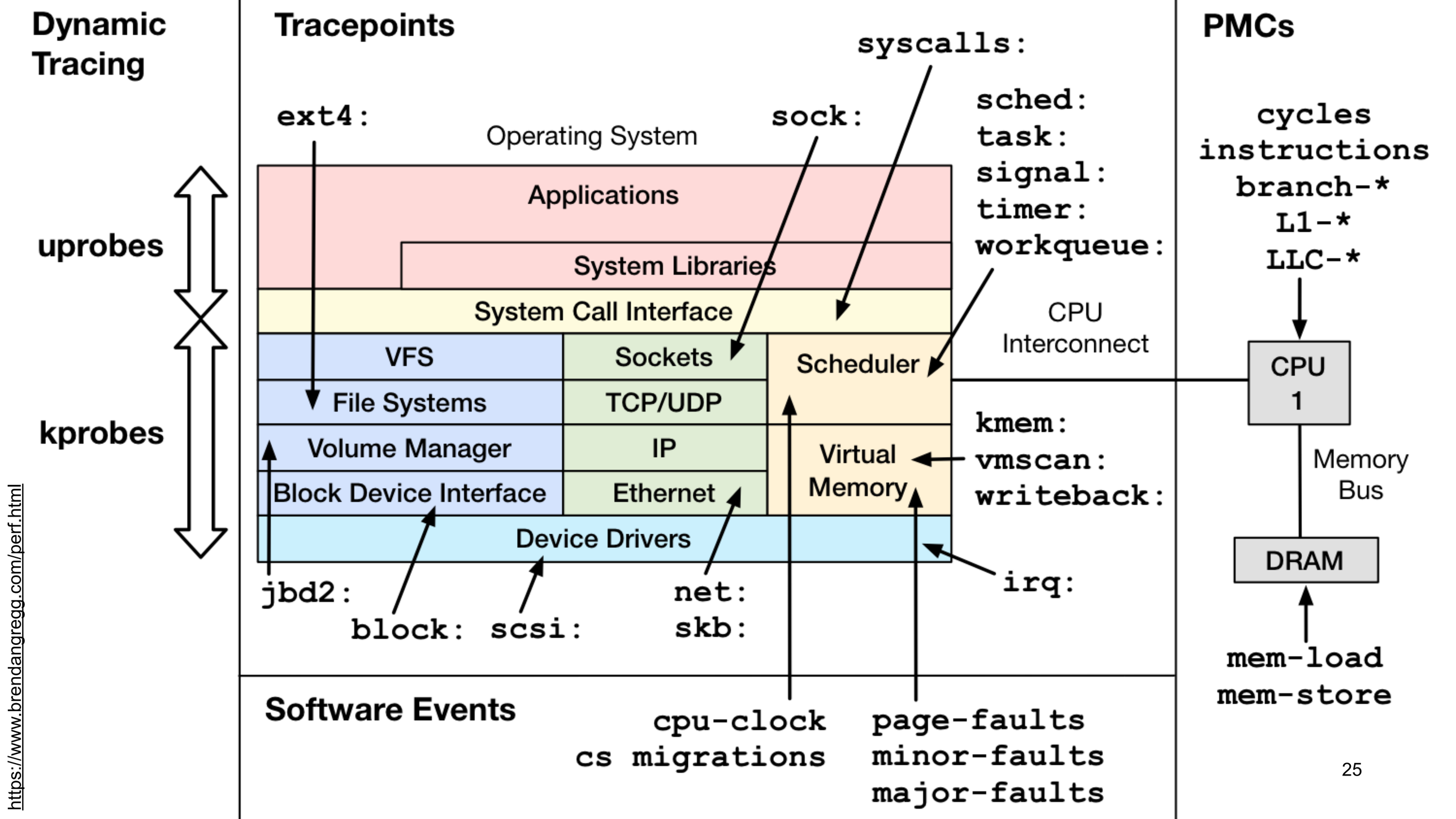
- cycles:u — мониторить такты (cpu cycles) на user level
- cycles:k — kernel level
- cycles — kernel + user

https://perfwiki.github.io/main/tutorial/#events

-e — события

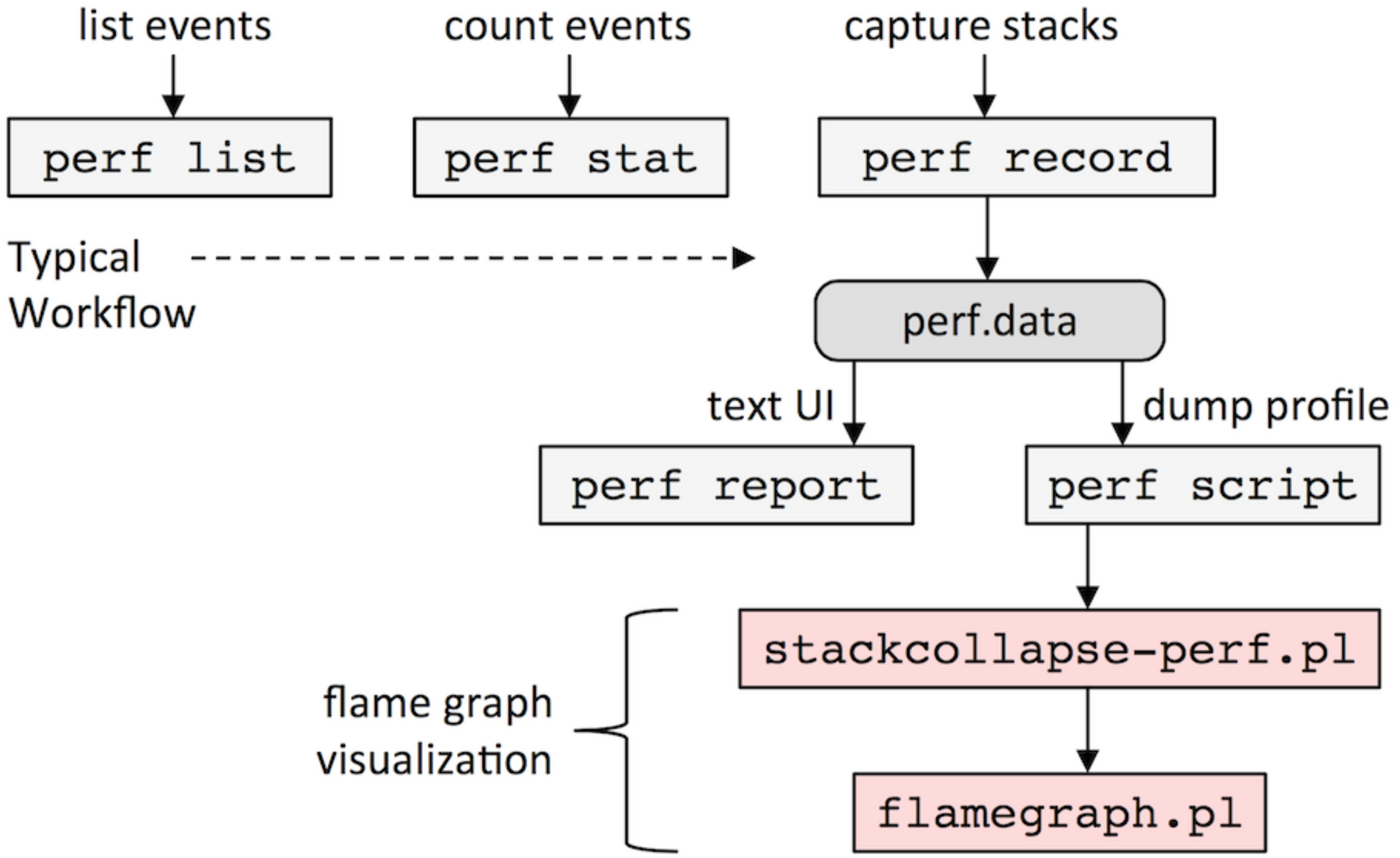
```
sudo perf list
```

```
…
ext4:ext4_error                                    [Tracepoint event]
…
filemap:file_check_and_advance_wb_err              [Tracepoint event]
filemap:filemap_set_wb_err                         [Tracepoint event]
filemap:mm_filemap_add_to_page_cache               [Tracepoint event]
filemap:mm_filemap_delete_from_page_cache          [Tracepoint event]
…
net:napi_gro_frags_entry                           [Tracepoint event]
net:napi_gro_frags_exit                            [Tracepoint event]
net:napi_gro_receive_entry                         [Tracepoint event]
net:napi_gro_receive_exit                          [Tracepoint event]
net:net_dev_queue                                  [Tracepoint event]
net:net_dev_start_xmit                             [Tracepoint event]
net:net_dev_xmit                                   [Tracepoint event]
net:net_dev_xmit_timeout                           [Tracepoint event]
net:netif_receive_skb                              [Tracepoint event]
net:netif_receive_skb_entry                        [Tracepoint event]
net:netif_receive_skb_exit                         [Tracepoint event]
net:netif_receive_skb_list_entry                   [Tracepoint event]
net:netif_receive_skb_list_exit                    [Tracepoint event]
net:netif_rx                                       [Tracepoint event]
net:netif_rx_entry                                 [Tracepoint event]
net:netif_rx_exit                                  [Tracepoint event]
net:netif_rx_ni_entry                              [Tracepoint event]
net:netif_rx_ni_exit                               [Tracepoint event]
…
sock:inet_sock_set_state                           [Tracepoint event]
sock:sock_exceed_buf_limit                         [Tracepoint event]
sock:sock_rcvqueue_full                            [Tracepoint event]
…
tcp:tcp_destroy_sock                               [Tracepoint event]
tcp:tcp_probe                                      [Tracepoint event]
tcp:tcp_rcv_space_adjust                           [Tracepoint event]
tcp:tcp_receive_reset                              [Tracepoint event]
tcp:tcp_retransmit_skb                             [Tracepoint event]
tcp:tcp_retransmit_synack                          [Tracepoint event]
tcp:tcp_send_reset                                 [Tracepoint event]
…
udp:udp_fail_queue_rcv_skb                         [Tracepoint event]
…
```



https://www.brendangregg.com/perf.html

perf Actions: Workflow



https://www.brendangregg.com/perf.html

-g — запись стека вызовов

Для стектрейсов и флеймчарта
`perf record -g`

В документации совсем непонятно: g enables call-graph recording

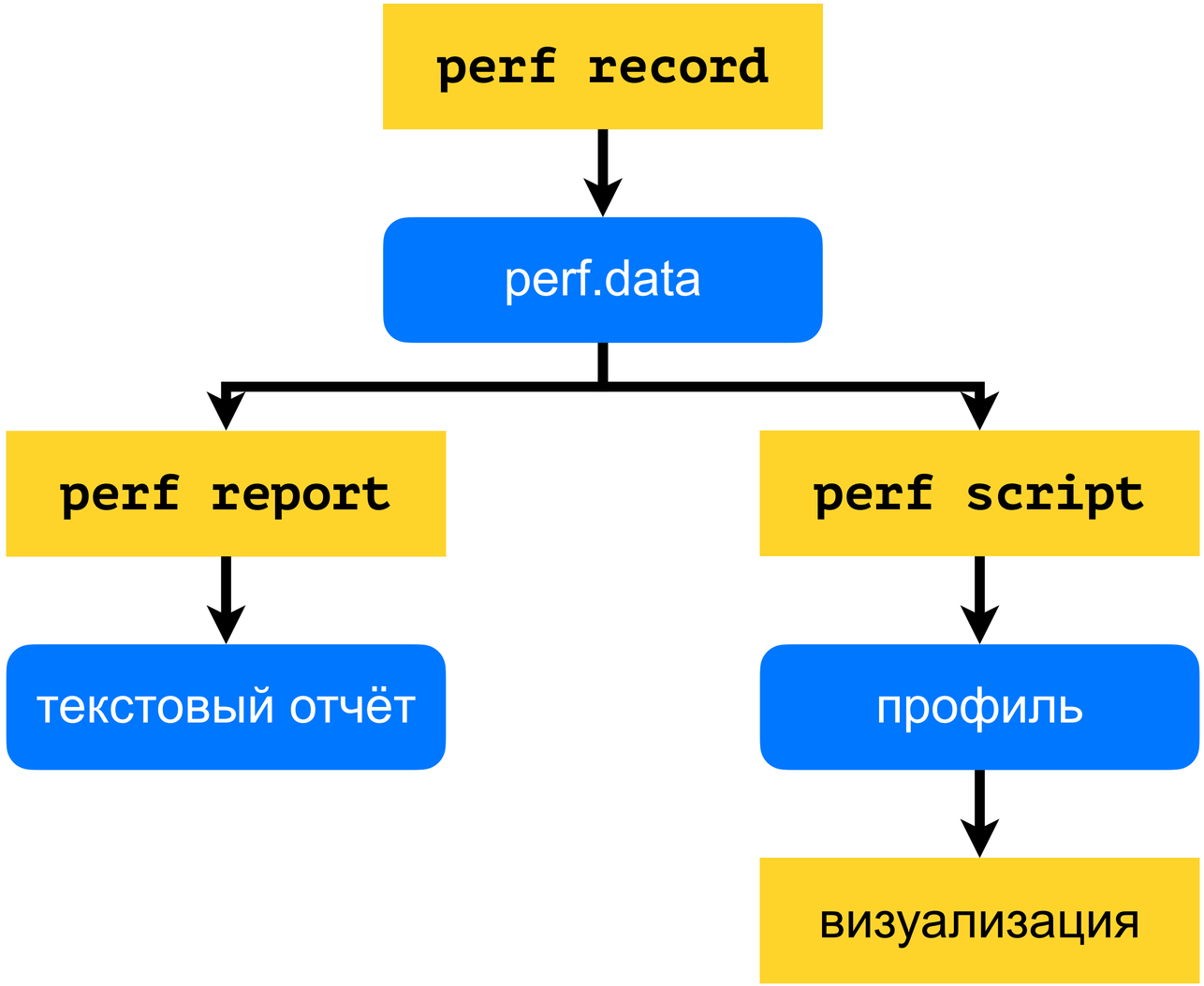
-a — профилирование всех процессов

- Удобно при запуске дочерних процессов
- Не все приложения умеют с таким работать

2. Расшифровка perf.data, получение perf.txt

```
perf report
perf script > perf.txt
```

- снова запускаем perf
- perf обрабатывает семплы из perf.data
- группирует одинаковые, агрегирует etc.
- из доступных источников (маппинг) даёт адресам название
- маппинг можем генерировать сами



https://perfwiki.github.io/main/tutorial/#sampling-with-perf-record

Perf: профилирование unix-way

- Профилирование для любого языка
- В движке языка не надо реализовывать профилирование
- Достаточно сделать маппинг «адрес — имя функции»

Кто умеет работать с perf

Linux, C, C++	всегда
Node.js 0.11	~2014
Java 8	~2015
Python 3.12	2023

Perf + Node

Опция для Node.js

```
--perf-basic-prof-only-functions
--perf-basic-prof
```

- Node.js создаёт для каждого процесса маппинг `/tmp/perf-PID.map`
- В маппинге адреса в памяти процесса + имена функций JS

Как выглядит маппинг

```
19277c0 84 Builtin:GlobalIsNaN
1927880 c Builtin:JsonParse
19278c0 c Builtin:JsonStringify
7efc9814e6c0 b40 JS:^emit node:events:466:44
7efc98150740 a14 JS:^Readable.on node:internal/streams/readable:1125:33
7efc98151200 22c JS:^Socket.read node:net:768:33
7efc98151440 46c JS:^Timeout node:internal/timers:170:14
7efc98151cc0 14c JS:^parseurl node_modules/parseurl/index.js:35:19
7efc98151e40 1ac JS:^TimersList node:internal/timers:246:14
7efc981522c0 194 JS:^handle node_modules/express/lib/router/layer.js:86:49
7efc98154000 378 JS:~ app.js:45:21
1ba1e837bce e JS:~ app.js:45:21
7efc981543c0 378 JS:~exit node:internal/process/per_thread:175:16
1ba1e837d6e a2 JS:~exit node:internal/process/per_thread:175:16
7efc98154780 378 JS:~get node:internal/bootstrap/node:113:8
```

Ещё опция для интерпретируемого кода

`--interpreted-frames-native-stack`

- Интерпретатор в памяти обычно один
- V8 создаёт отдельную копию для каждой функции
- Since the InterpreterEntryTrampoline has to be copied
  this comes at slight performance and memory regression
- Интерпретируемый код и так медленный, почти незаметно

Собираем всё вместе

```
node \
  --perf-basic-prof \
  --interpreted-frames-native-stack \
  index.js
```

Perf + Node:
что может пойти не так

Ключи запуска Perf

```
perf record -g
```

Ключи запуска Node

Запуск бинарника node завернут во много слоёв абстракций

Про возможность донести туда свои ключи часто забывают

Неправильный путь к маппингу

Если в профиле имена функций остались не расшифрованы
На одном сервере

- node пишет маппинг в /tmp/
- perf script ищет маппинг в /var/tmp/

Неправильные имена из маппинга

Одна версия perf script брала неправильные имена, если map не отсортирован

```
156f100 a fn1
105a1566f8f6 1c fn6
156f640 a fn2
156f2000 a fn4
```

Маппинг до и после сортировки

```
156f100 a fn1
105a1566f8f6 1c fn6
156f640 a fn2
156f2000 a fn4
```

```
156f100 a fn1
156f640 a fn2
156f2000 a fn4
105a1566f8f6 1c fn6
```

https://gist.github.com/victor-homyakov/ac65fb7c5fd21c0a05c5a78486bc8549

Маппинги могут сожрать всё место

Один процесс node = один файл маппинга

Воркеры/пул/кластер + автоматические рестарты = новые маппинги

Кодогенерация = разрастание маппинга?

Лайфхак: длинные пути в профиле

- Путь к node
- Путь к приложению
- Путь к node_modules (особенно pnpm)

Лайфхак: длинные пути в профиле

- Меньше размер файла (e.g. perf.txt 20MB → 15MB)
- Удобно читать флеймчарт
- Можно объединять профили разных серверов
- Можно сравнивать профили разных версий

Получили профиль.
И что с ним делать?

Визуализация, генерация флейм-графа

Флейм-граф, флейм-чарт, flame graph, flame chart

- stackvis из документации Node
- speedscope
- github.com/brendangregg/FlameGraph
- perf script

stackvis

```
perf record …
perf script >perf.txt
npx stackvis perf <perf.txt >flamegraph-stackvis.html
```

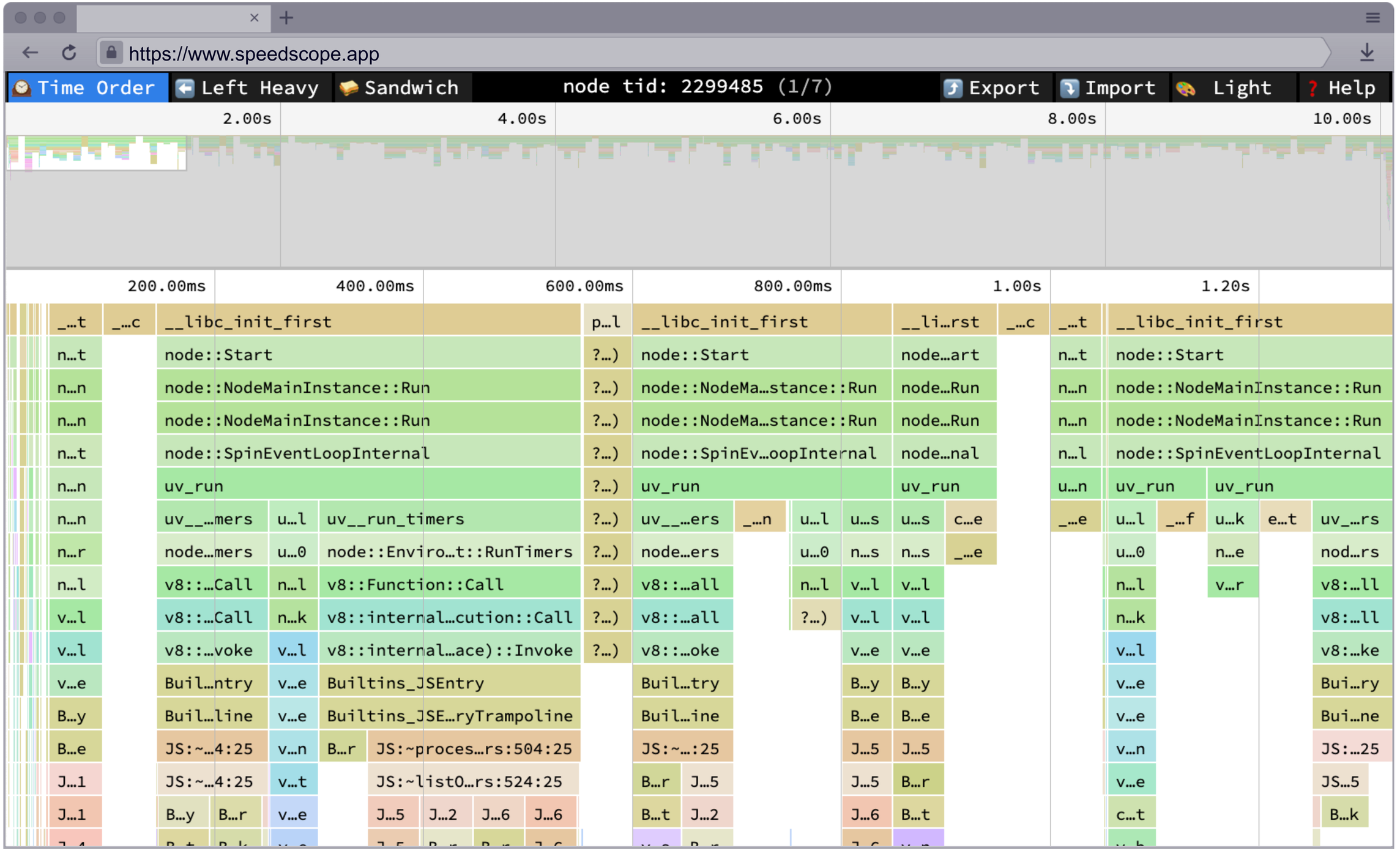
- Советуют в документации Node.js
- Результат — svg в html-страничке
- Криво рисуется, тормозит даже для простого профиля

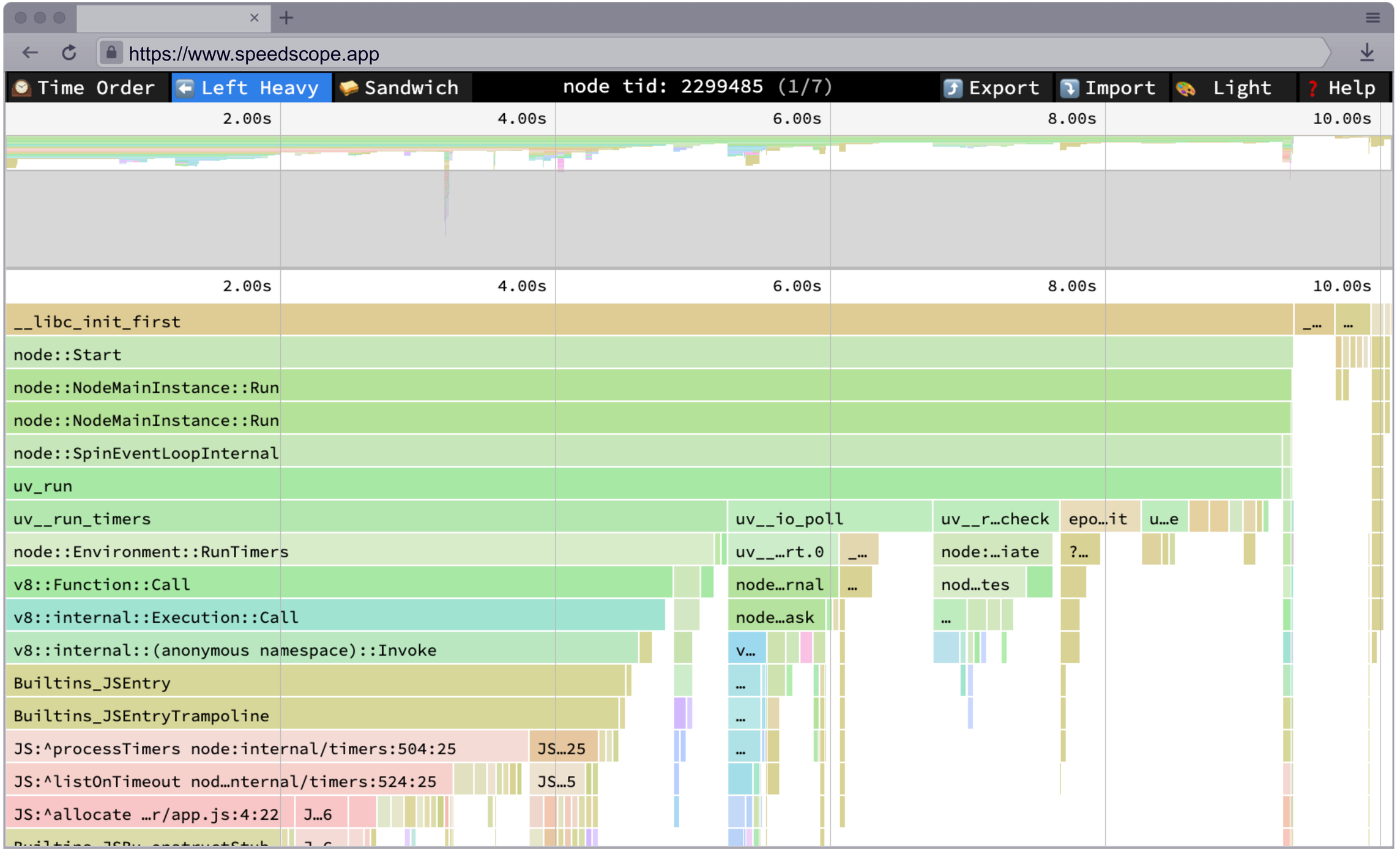
https://nodejs.org/en/learn/diagnostics/flame-graphs#create-a-flame-graph-with-system-perf-tools

speedscope

- Интерактивный, управление мышкой/WASD-+/тачпадом
- Поиск
- Таймлайн/флеймграф/сэндвич

https://www.speedscope.app/


https://www.speedscope.app
Time Order
Left Heavy
Sandwich
node tid: 2299485 (1/7)
Export
Import
Light
Help
2.00s
4.00s
6.00s
8.00s
10.00s
200.00ms
400.00ms
600.00ms
800.00ms
1.00s
1.20s
__libc_init_first
node::Start
node::NodeMainInstance::Run
node::SpinEventLoopInternal
uv_run
uv__run_timers
node::Environment::RunTimers
v8::Function::Call
v8::internal::Builtins_JSEntry
v8::internal::Builtins_JSEntryTrampoline


https://www.speedscope.app
Time Order
Left Heavy
Sandwich
node tid: 2299485 (1/7)
Export
Import
Light
Help
2.00s
4.00s
6.00s
8.00s
10.00s
__libc_init_first
node::Start
node::NodeMainInstance::Run
node::NodeMainInstance::Run
node::SpinEventLoopInternal
uv_run
uv__run_timers
uv__io_poll
uv__r…check
epo…it
u…e
node::Environment::RunTimers
uv__…rt.0
node:…iate
v8::Function::Call
node…rnal
nod…tes
v8::internal::Execution::Call
node…ask
v8::internal::(anonymous namespace)::Invoke
Builtins_JSEntry
Builtins_JSEntryTrampoline
JS:^processTimers node:internal/timers:504:25
JS…25
JS:^listOnTimeout nod…nternal/timers:524:25
JS…5
JS:^allocate …r/app.js:4:22
J…6

FlameGraph от Брендана Грегга

```
git clone --depth 1 \
  https://github.com/brendangregg/FlameGraph
cd FlameGraph

# получаем/копируем perf.txt

./stackcollapse-perf.pl perf.txt | \
  ./flamegraph.pl --colors js > flamegraph.svg
```

https://github.com/brendangregg/FlameGraph


file:///flamegraph.svg
I.. InterpretedFunction: ..
I.. InterpretedFunction: ..
I.. InterpretedFunction: ..
I.. InterpretedFunction: ..
I.. InterpretedFunction: ..
I.. InterpretedFunction: ..
I.. InterpretedFunction: ..
I.. InterpretedFunction: ..
I.. InterpretedFunction: ..
I.. InterpretedFunction: ..
I.. InterpretedFunction: ..
I.. InterpretedFunction: ..
I.. InterpretedFunction: ..
I.. InterpretedFunction: ..
B.. Builtins_ArgumentsAda..
I.. InterpretedFunction: ..
L.. [perf-3729.map]
Builtins_ArgumentsAdaptor..
InterpretedFunction: /hom..
InterpretedFunction: /hom..
Builtins_JSEntryTrampoline
Builtins_JSEntry
v8::internal::(anonymous ..
v8::internal::Execution::C..
v8::Function::Call
napi_call_function
[binding-x64-linux.node]
v..
[unknown]
v8..
[unknown]
na..
napi
I..
I..
La..
I..
I..
In..
[binding-x64-linux.node]

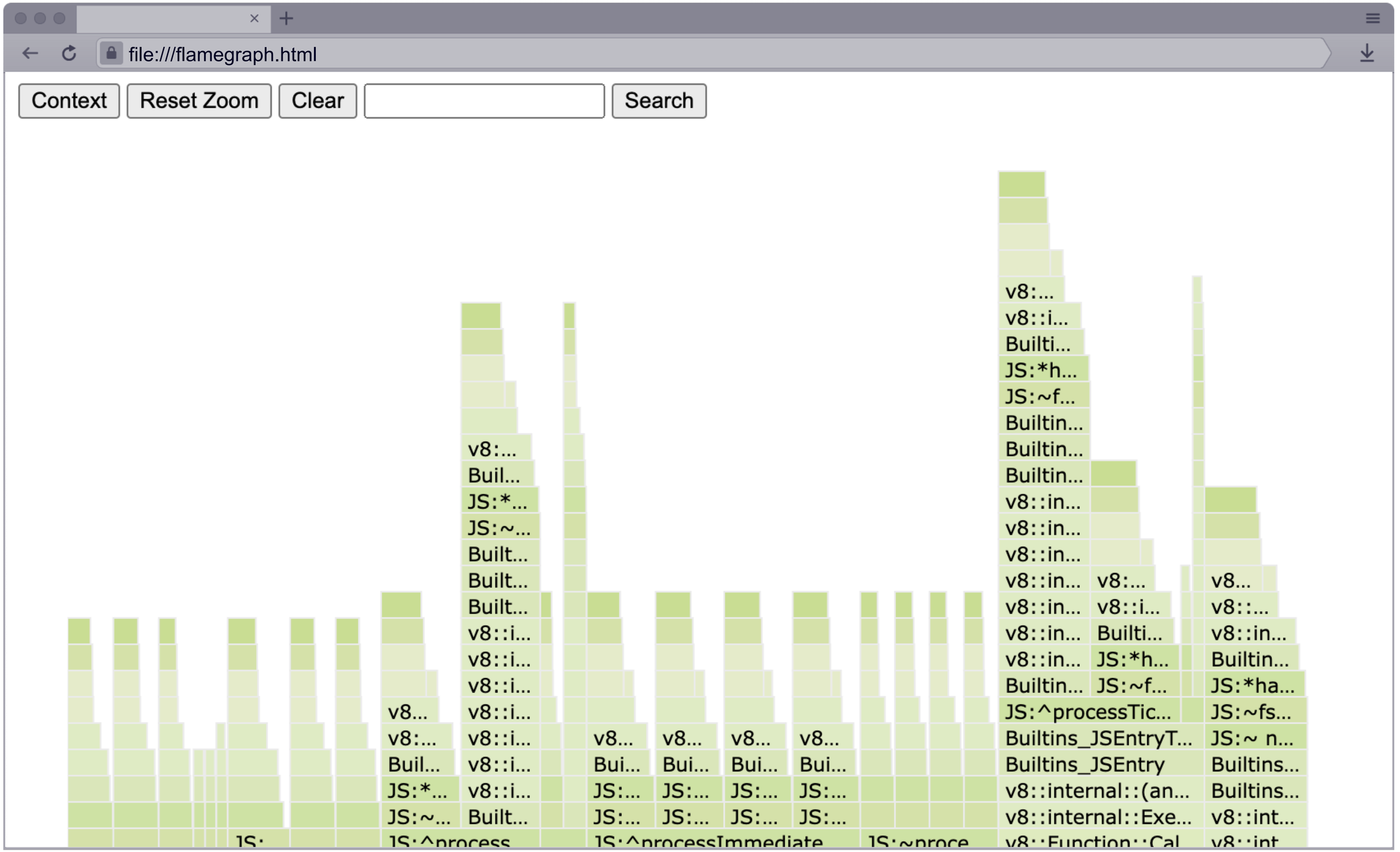
Встроенные скрипты perf

```
perf script -l

# ставим и собираем свой perf, затем
PERF_EXEC_PATH=~/perf-6.10/tools/perf/ \
  ~/perf-6.10/tools/perf/perf script -l
```

https://askubuntu.com/questions/1397419/how-can-i-get-pre-canned-scripts-for-perf-script

perf script flamegraph

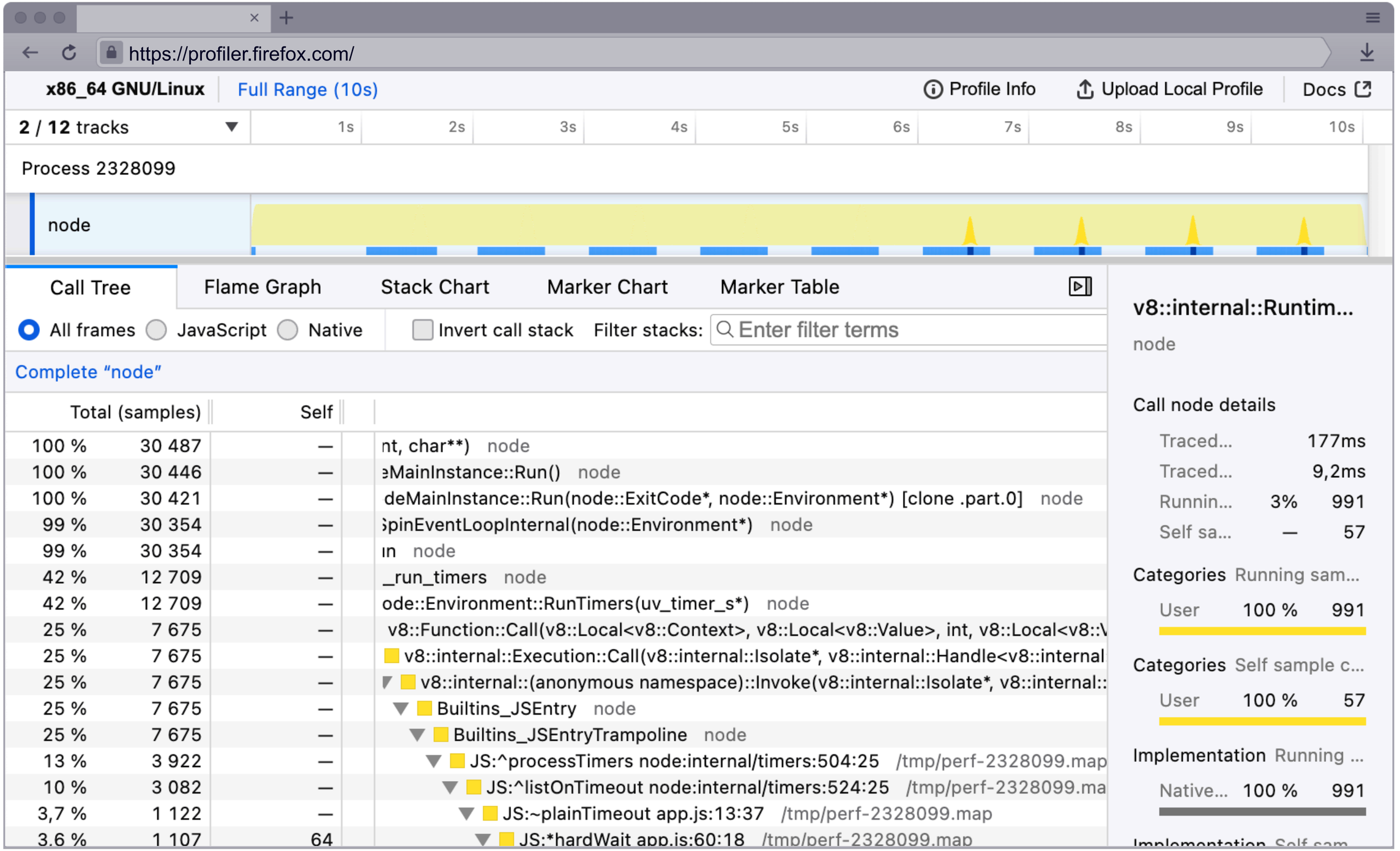
```
# создаст perf.data
perf record -a -g -F 999 sleep 60
# из perf.data создаст flamegraph.html
perf script report flamegraph

# одной командой:
perf script flamegraph -a -F 999 sleep 60
```


file:///flamegraph.html
Context
Reset Zoom
Clear
Search
v8:...
v8::i...
Builti...
JS:*h...
JS:~f...
Builtin...
Builtin...
Builtin...
v8::in...
v8::in...
v8::in...
v8::in...
v8::in...
v8::in...
Builtin...
JS:^processTic...
Builtins_JSEntryT...
Builtins_JSEntry
v8::internal::(an...
v8::internal::Exe...
v8::Function::Cal...
v8:...
v8::i...
Builti...
JS:*h...
JS:~f...
v8...
v8::...
v8::in...
Builtin...
JS:*ha...
JS:~fs...
JS:~ n...
Builtins...
Builtins...
v8::int...
v8::int
v8:...
Buil...
JS:*...
JS:~...
Built...
Built...
Built...
v8::i...
v8::i...
v8::i...
v8::i...
v8::i...
v8...
v8:...
Buil...
JS:*...
JS:~...
Built...
v8...
v8...
v8...
v8...
Bui...
Bui...
Bui...
Bui...
JS:...
JS:...
JS:...
JS:...
JS:...
JS:...
JS:...
JS:...
JS:^process
JS:^processImmediate
JS:~proce

perf script gecko

```
# создаст perf.data
perf record -a -g -F 999 sleep 60
# из perf.data создаст gecko_profile.json
perf script report gecko

# одной командой:
perf script gecko -a sleep 60
```

https://profiler.firefox.com/


https://profiler.firefox.com/
x86_64 GNU/Linux
Full Range (10s)
Profile Info
Upload Local Profile
Docs
2 / 12 tracks
1s
2s
3s
4s
5s
6s
7s
8s
9s
10s
Process 2328099
node
Call Tree
Flame Graph
Stack Chart
Marker Chart
Marker Table
All frames
JavaScript
Native
Invert call stack
Filter stacks:
Enter filter terms
Complete “node”
Total (samples)
Self
100 % 30 487 — nt, char**) node
100 % 30 446 — eMainInstance::Run() node
100 % 30 421 — deMainInstance::Run(node::ExitCode*, node::Environment*) [clone .part.0] node
99 % 30 354 — SpinEventLoopInternal(node::Environment*) node
99 % 30 354 — in node
42 % 12 709 — _run_timers node
42 % 12 709 — ode::Environment::RunTimers(uv_timer_s*) node
25 % 7 675 — v8::Function::Call(v8::Local<v8::Context>, v8::Local<v8::Value>, int, v8::Local<v8::V
25 % 7 675 — v8::internal::Execution::Call(v8::internal::Isolate*, v8::internal::Handle<v8::internal
25 % 7 675 — v8::internal::(anonymous namespace)::Invoke(v8::internal::Isolate*, v8::internal::
25 % 7 675 — Builtins_JSEntry node
25 % 7 675 — Builtins_JSEntryTrampoline node
13 % 3 922 — JS:^processTimers node:internal/timers:504:25 /tmp/perf-2328099.map
10 % 3 082 — JS:^listOnTimeout node:internal/timers:524:25 /tmp/perf-2328099.ma
3,7 % 1 122 — JS:~plainTimeout app.js:13:37 /tmp/perf-2328099.map
3,6 % 1 107 64 JS:*hardWait app.js:60:18 /tmp/perf-2328099.map
v8::internal::Runtim...
node
Call node details
Traced... 177ms
Traced... 9,2ms
Runnin... 3% 991
Self sa... — 57
Categories Running sam...
User 100 % 991
Categories Self sample c...
User 100 % 57
Implementation Running ...
Native... 100 % 991

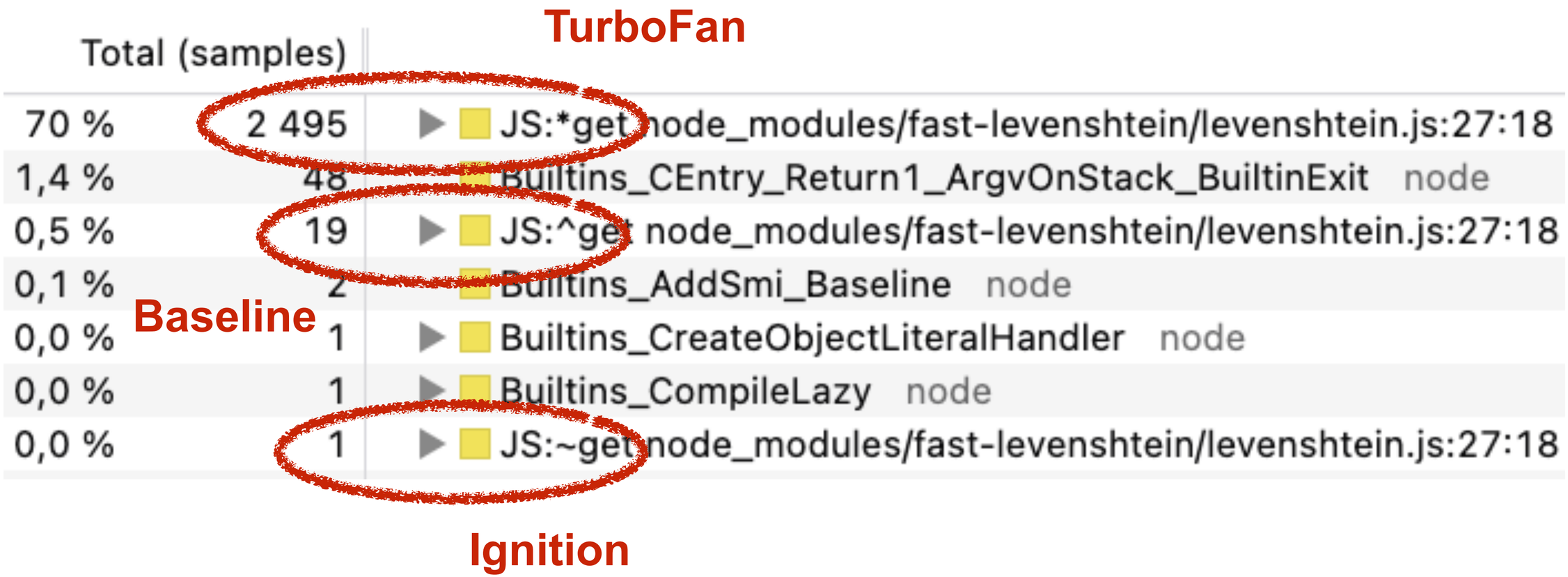
Что в профиле значат ~^+*

JS → AST → Bytecode

~	Ignition Interpreter	
^	Baseline compiler	machine code
+	Maglev fast optimising compiler	slightly optimised machine code
*	TurboFan slow optimising compiler	fully optimised machine code

https://github.com/v8/v8/blob/f794a1777c8624e625dd992527b133fd0cd6f1d6/src/objects/code-kind.cc#L21

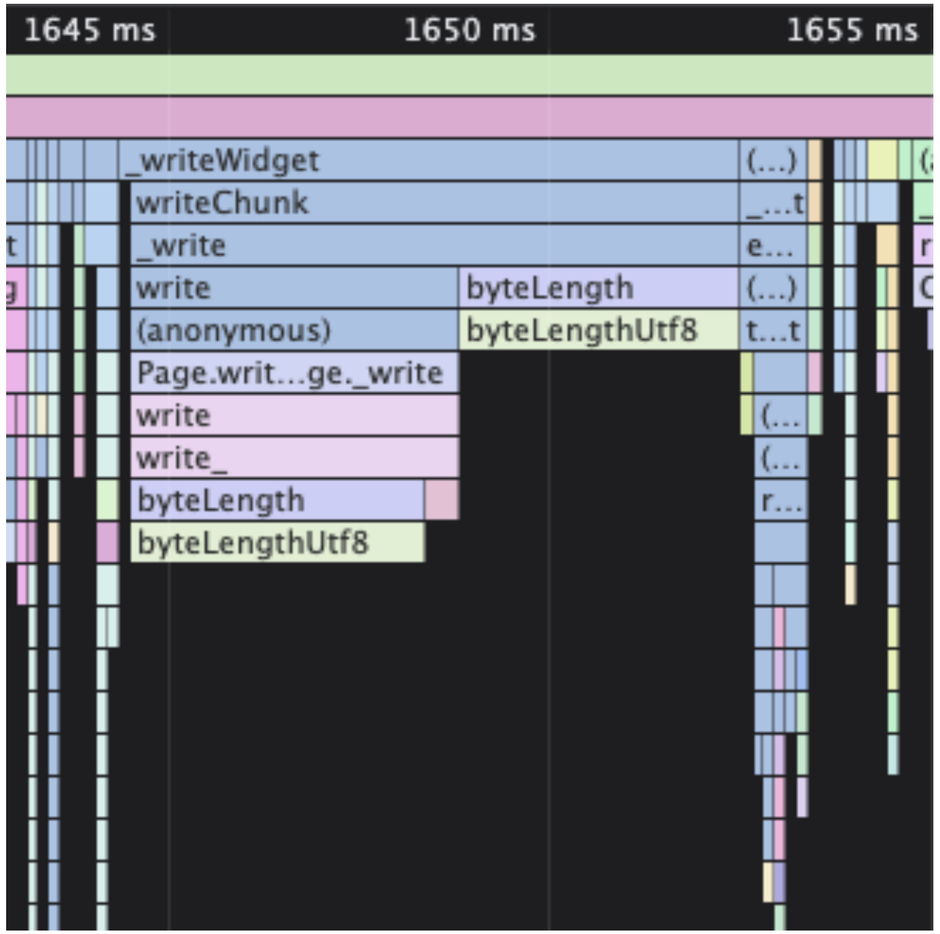
Что в профиле значат ~^+*



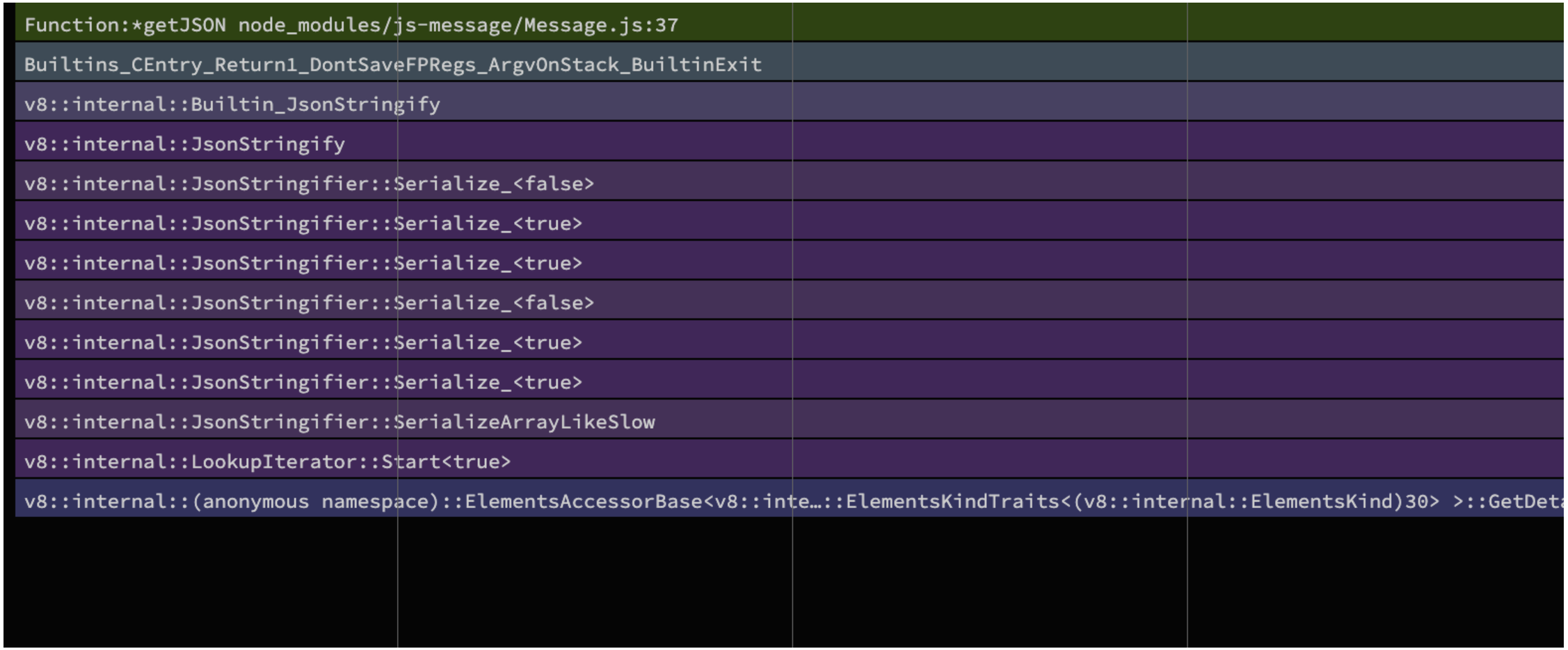
Анализ: что можно увидеть в профиле

- Кишки event loop
- Долгую обработку запросов/стримов
  не в той фазе event loop (в фазе io poll)
- Аллокации памяти на ArrayBuffer
- JSON.stringify, JSON.parse
  (обычно видим только вызывающую функцию)
- Работа с диском и сетью

Пример: логирование длины буфера



Пример: JSON.stringify array-like



Итог

Рабочее решение

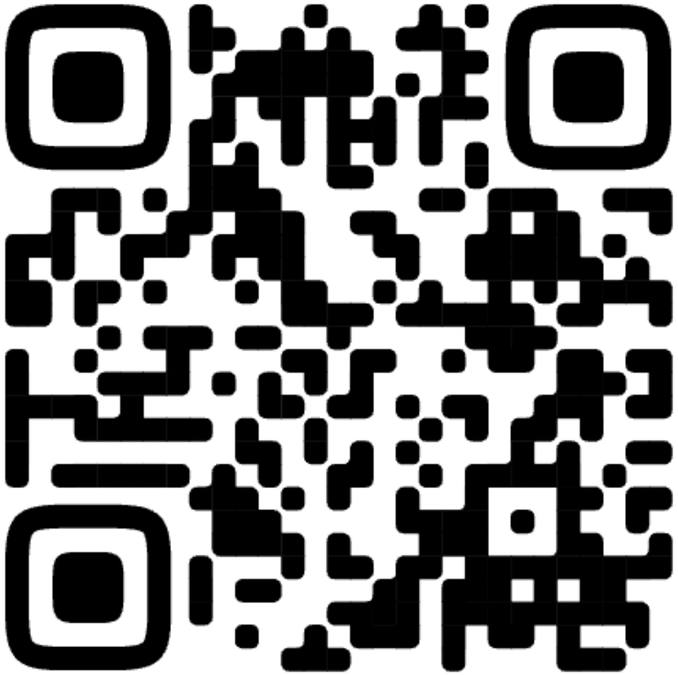
- Node с правильными ключами
- Perf с правильными ключами
- Проверяю/копирую map в нужное для perf место
- Сортирую map по возрастанию адресов
- Сокращаю длинные пути

Достоинства и недостатки perf

- ✅ Low overhead
- ✅ Профиль нескольких процессов/потоков
- ✅ Потоки без активности не семплирует
- ✅ Частота семплирования > 1kHz
- ✅ Видим внутренности Node
- ✅ Бонус: процессы Python/Java

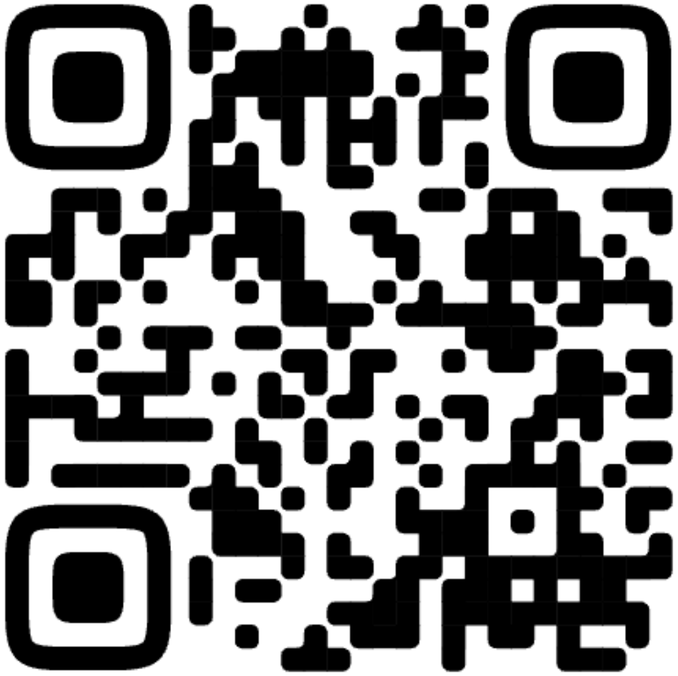
- ⛔ Только Linux
- ⛔ Запуск Node с параметрами
- ⛔ Маппинги могут забить диск (воркеры+рестарты)

Ссылки



https://clck.ru/3EJYQf

Спасибо! Вопросы?



https://clck.ru/3EJYQf