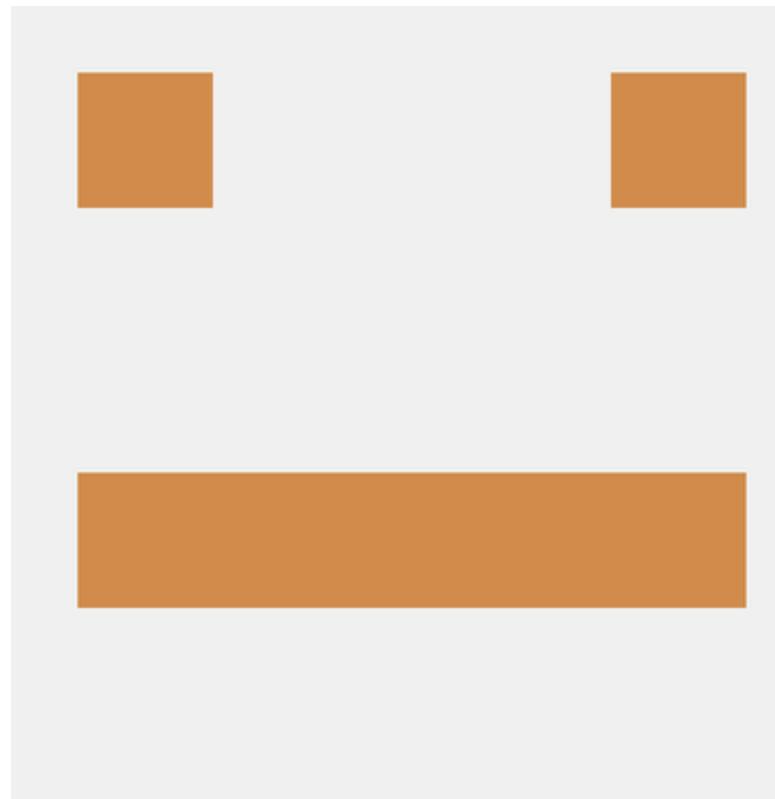


PHP 8 と v8 (JavaScript) で 速さを見比べてみよう！

五十嵐 進士 / sji / sj-i / @sji_ch

自己紹介



- @sji_ch
- SNS 上でのアイコンは GitHub が自動生成した奴

生まれも育ちも仙台



PHP カンファレンス仙台とかやった



—昨年娘ができた

- かわいい
- 絵本好き

Agenda

- PHP8とJITの概要
- V8エンジンの概要
- ベンチマーク対決しつつグダグダ話す
- まとめ

免責事項

役に立つ話はしません

想定する聴講者

- PHP スクリプトの性能や測定方法が気になる人
- 今後の PHP の性能の伸びしろに思いを馳せたい人
- しょうもないマイクロベンチマークが好きな人
- 世俗的な利益ばかり追求しないスローライフ志向の人？

PHP8 と JIT のおさらい

PHP8 で JIT 入ったよ

JIT コンパイラって何

- Just In Time の略
- 実行時に非機械語 → 機械語にコンパイル

インタプリタと中間言語

- ・多くのインタプリタがまずソースコード → 中間言語 にコンパイル
- ・インタプリタは中間言語を逐次実行する VM を持つ
- ・逐次実行せず JIT コンパイルすれば高速実行できる

PHP コードの実行と opcache

- PHP もコードを実行前に VM 命令列へコンパイル
- コンパイルは毎回のリクエストでやるには重い
- opcache でメモリ上にコンパイル結果をキャッシュ

PHP8 での JIT

- opcacheの追加機能
- キャッシュしたVM命令を更に機械語命令へ

JIT の使いどころ

- 典型的な Web アプリケーションは I/O or DB バウンド
- PHP 側が速くなっても全体として速くならないことが多い
- 静的解析ツールとか、CPU をぶん回すようなアプリケーションには向く
- 皆で面白い使い道を考えようね

JavaScript と v8

V8 とは

- Chrome や Node の JavaScript 実行エンジン
- ブラウザベンダの熾烈な競争の中で磨かれてきた

V8 の構成と JIT

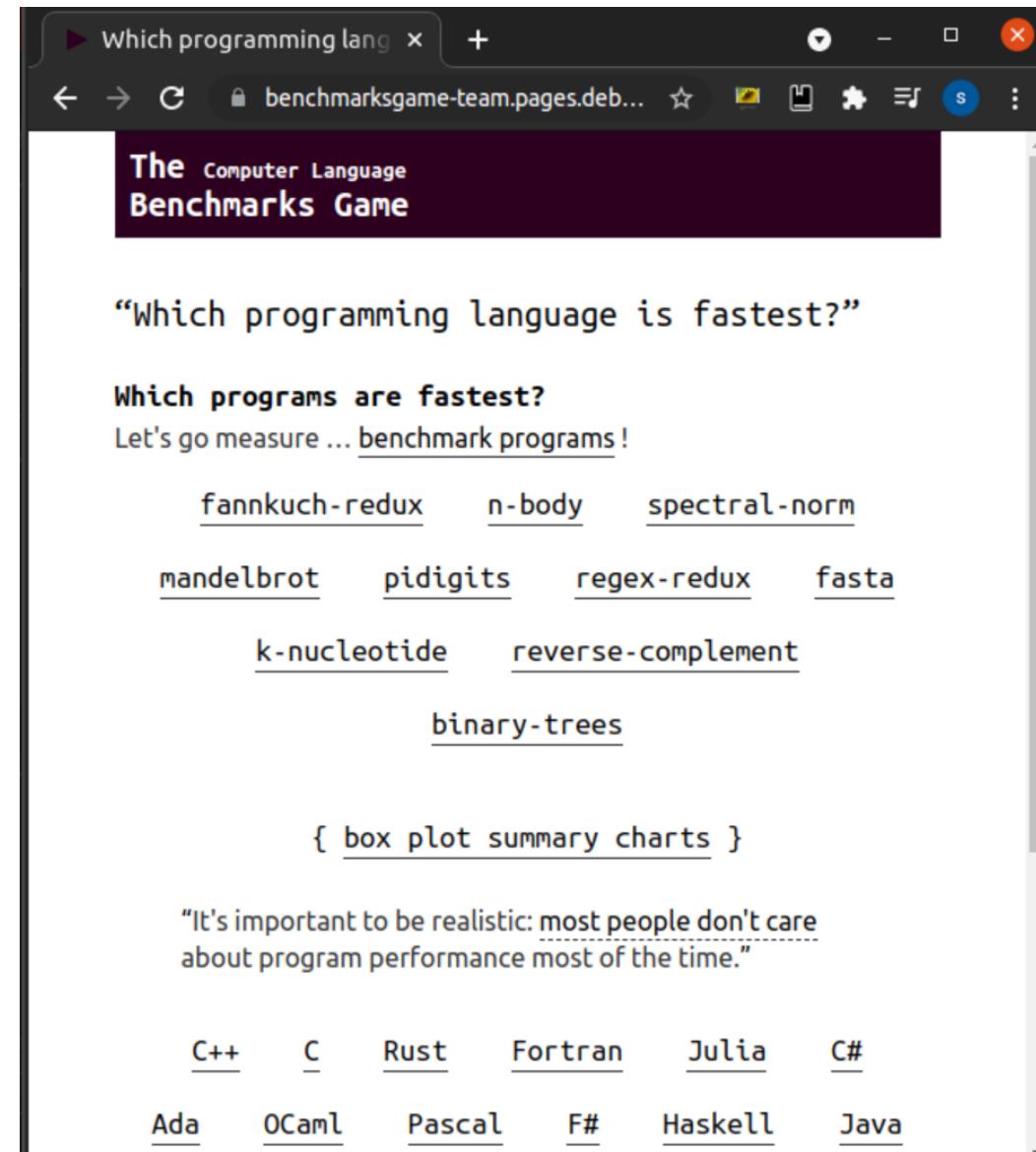
- Ignition: ソース→中間言語のコンパイラとインタプリタ
- Turbofan: 中間言語→機械語のコンパイラ、実行時情報を元に最適化
- Sparkplug: 中間言語→機械語のコンパイラ、最適化抑えめで軽量

PHP8 vs V8、 真の 8 はどっちだ！！！

ベンチマークで決めよう！

- いいのかそれで！

The Computer Language Benchmarks Game



- <https://benchmarksgame-team.pages.debian.net/benchmarksgame/>
- プログラム言語ベンチマークサイト
- 複数言語で同じアルゴリズム的な問題を解いて性能比較
- 煽りに反し「一番速いのはどの言語か？」を決めるサイトではない

"It's important to be realistic: most people don't care about program performance most of the time."

https://tratt.net/laurie/blog/entries/what_challenges_and_trade_offs_do_optimising_compilers_face.html

- なお今回使ったベンチマークスクリプトは GitHub にて公開
 - <https://github.com/sj-i/phpcon2021/tree/master/benchmarks>

regex-redux (1)

- <https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/regexredux.html>
- FASTA 形式の入力をとる(1)
 - 塩基配列やアミノ酸配列用のデータ形式
- 改行やコメント部分を除去(2)
- 既定の複数の正規表現で整形し、各正規表現ごとのマッチ数を出力
- 入力に対して別の既定の複数の正規表現を指定順で適用(3)
- (1)(2)(3)の長さを出力

regex-redux (2)

regex-redux

source	secs	mem	gz	busy
<u>PHP</u>	1.73	162,348	816	3.58
<u>Node.js</u>	4.82	1,150,424	668	5.90

```
if($key == 0 || $key == 2 || $key == 4 || $key == 6) {  
    $pid = pcntl_fork();  
    if($pid) $workers[] = $pid;  
}  
...  
...
```

- サイトに掲載のデータだとPHPが圧倒的に速い
- コードを見比べると、PHPコードの方が fork で効率的に並列化している
- 適切な比較にならない

regex-redux (3)

- Node 側コードのうち愚直に直列実行するコードを PHP へ翻訳

```
const endLen = data
  .replace(/tHa[Nt]/g, '<4>')
  .replace(/aND|caN|Ha[DS]|WaS/g, '<3>')
  .replace(/a[NSt]|BY/g, '<2>')
  .replace(/<[^>]*>/g, '|')
  .replace(/\|[^\|][^\|]*\|/g, '-')
  .length;

console.log(`\n${initialLen}\n${cleanedLen}\n${endLen}`)
```

```
$tmp = preg_replace('/tHa[Nt]/S', '<4>', $data);
$tmp = preg_replace('/aND|caN|Ha[DS]|WaS/S', '<3>', $tmp);
$tmp = preg_replace('/a[NSt]|BY/S', '<2>', $tmp);
$tmp = preg_replace('/<[^>]*>/S', '|', $tmp);
$tmp = preg_replace('/\|[^\|][^\|]*\|/S', '-', $tmp);
$endLen = strlen($tmp);

echo PHP_EOL,
$initialLen, PHP_EOL,
$cleanedLen, PHP_EOL,
```

regex-redux (4)

```
$ time node regex-redux.js <input5000000
real    0m3.729s
user    0m3.450s
sys     0m0.377s
```

```
$ time php -dopcache.jit_buffer_size=16M -dmemory_limit=1024M
regex-redux.php <input5000000
real    0m2.349s
user    0m2.196s
sys     0m0.135s
```

- PHP の方が速い !

regex-redux (5)

JIT 無効

```
$ time php -dopcache.jit_buffer_size=0M -dmemory_limit=1G  
regex-redux.php <input5000000  
  
real    0m2.347s  
user    0m2.196s  
sys     0m0.153s
```

JIT 有効

```
$ time php -dopcache.jit_buffer_size=16M -dmemory_limit=1G  
regex-redux.php <input5000000  
  
real    0m2.349s  
user    0m2.196s  
sys     0m0.135s
```

- PHP で JIT 有効でも速度が変わらない？
- プロファイルをとって結果を比較してみる

ところでプロファイルとは

- プログラムの部品単位での実行性能を調べること
- 大体ごく一部の処理が実行時間の大部分
 - 遅いごく一部=ボトルネック
- 性能改善の際はまずボトルネックを見つける

node のプロファイル

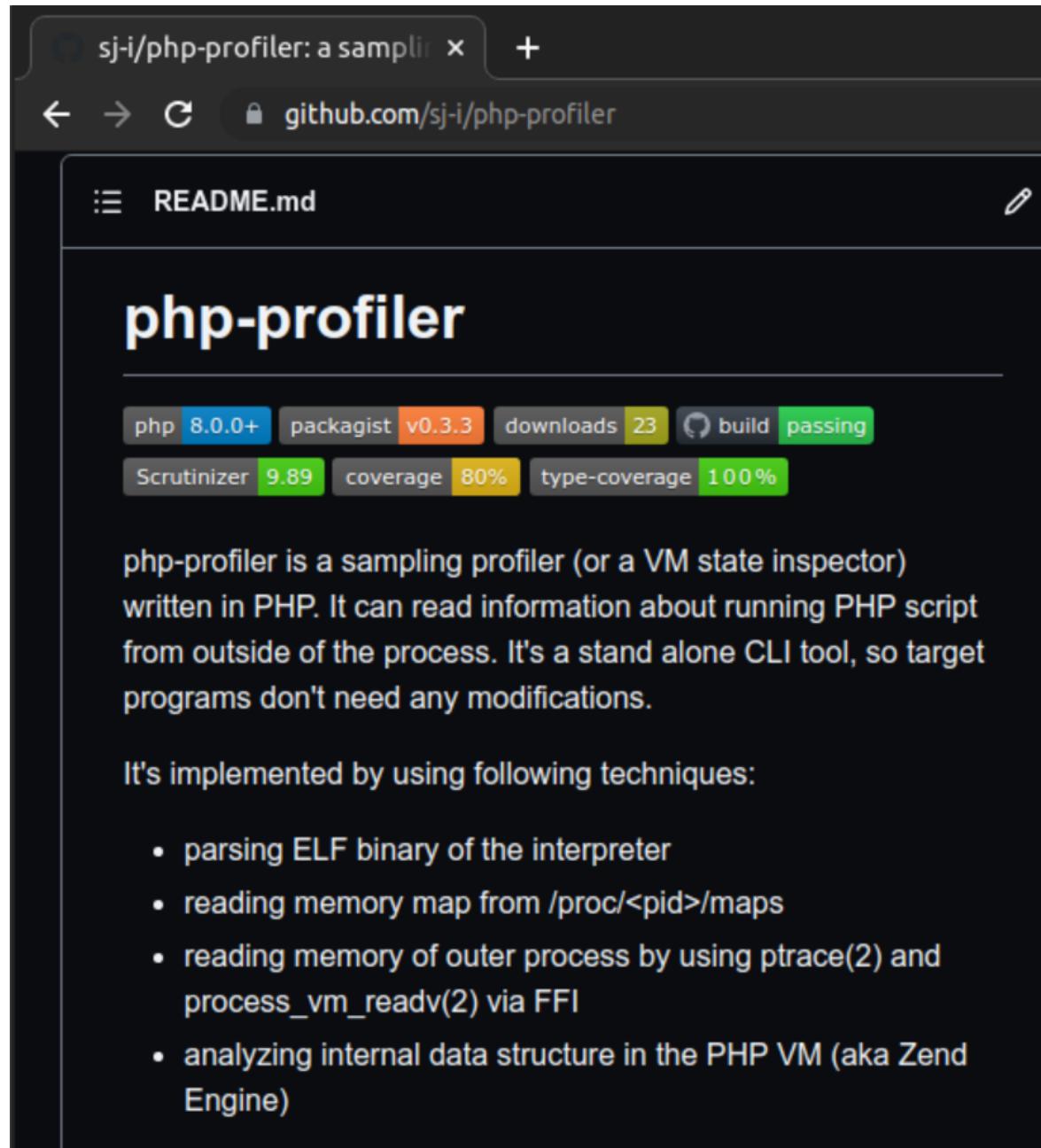
- 標準でプロファイルが付属
- --prof で結果がファイル出力される
- --inspect-brk を使うとchromeからつなげることもできる

regex-redux: V8 プロファイル結果

Self Time	Total Time	Function
2090.8 ms 54.13 %	2090.8 ms 54.13 %	(garbage collector)
1802.0 ms	1802.0 ms	(idle)
541.9 ms 14.03 %	1756.6 ms 45.48 %	▶ mainThread
245.7 ms 6.36 %	245.7 ms 6.36 %	▶ RegExp: a[NSt]lBY
154.6 ms 4.00 %	154.6 ms 4.00 %	▶ RegExp: aND aN Ha[DS] WaS
78.6 ms 2.03 %	78.6 ms 2.03 %	▶ RegExp: a[act]ggtaaa tttacc[agt]t
75.3 ms 1.95 %	75.3 ms 1.95 %	▶ RegExp: <[^>]*>
74.0 ms 1.91 %	74.0 ms 1.91 %	▶ RegExp: ag[act]gtaaa tttac[agt]ct
67.8 ms 1.76 %	67.8 ms 1.76 %	▶ RegExp: [cgt]gggtaaa tttaccc[acg]
66.9 ms 1.73 %	66.9 ms 1.73 %	▶ RegExp: aggg[acg]aaa ttt[cgt]ccct
65.7 ms 1.70 %	65.7 ms 1.70 %	▶ RegExp: agg[act]taaa ttta[agt]cct
64.4 ms 1.67 %	64.4 ms 1.67 %	▶ RegExp: agggta[cgt]a t[acg]taccct
61.8 ms 1.60 %	61.8 ms 1.60 %	▶ RegExp: aaaaat[cat]aaltt[acal]accct

- 全体の半分以上が GC
- GC 分をのぞくと正規表現自体の実行が大きい

PHP の（自作）プロファイラ



- <https://github.com/sj-i/php-profiler>
- phpspy リスペクトの PHP プロファイラ
- FFI 経由でプロセス外から処理系のメモリを読む
- ソースコードの行番号 / VM 命令レベルで計測がとれる

regex-redux: PHP8 プロファイル結果 (1)

```
php-profiler inspector:trace -S -- php regex-redux.php <input5000000 >profiled
cat profiled | sed '/^$/d' | sed 's/^*[0-9]*//' | sort | uniq -c | sort -nr
```

regex-redux: PHP8 プロファイル結果 (2)

```
187 <main> /home/sji/work/talk/phpcon2021/benchmark/regex-redux/regex-redux.php:48:ZEND_DO_UCALL
125 mainThread /home/sji/work/talk/phpcon2021/benchmark/regex-redux/regex-redux.php:32:ZEND_DO_ICALL
118 preg_match_all_ <internal>:-1:
62 preg_replace_ <internal>:-1:
22 mainThread /home/sji/work/talk/phpcon2021/benchmark/regex-redux/regex-redux.php:37:ZEND_DO_ICALL
15 mainThread /home/sji/work/talk/phpcon2021/benchmark/regex-redux/regex-redux.php:38:ZEND_DO_ICALL
12 mainThread /home/sji/work/talk/phpcon2021/benchmark/regex-redux/regex-redux.php:36:ZEND_DO_ICALL
```

- 実行時間のほとんどは `preg_replace()` の処理
- PHP 側スクリプトの動きはほとんどサンプリングされてこない

regex-redux (考察)

- 実質 C vs C++
- V8 側は GC に足を引っ張られてもいる
- PHP 側がほぼ C の処理を実行しているだけだと JIT の影響は小さい

regex-redux (ひどいまとめ)

- PHP の正規表現は速い
- というより C は速い
- JIT の有効性は CPU バウンドかどうかだけでは推測できない

pidigits、はスキップ

- <https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/pidigits.html>
- gmp 拡張のベンチマークになる
- C が速いのはもう分かった！

k-nucleotide (1)

- <https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/knucleotide.html>
- FASTA 形式の入力をとる(1)
- (1)から DNA 配列 THREE を取得する(2)
- ハッシュテーブルを用意(3)
- (2) の読み込み区間に応する内容へ(3)を更新する関数を用意(4)
 - ヌクレオチド種別がキー、数が値
- (3)と(4)を使って処理を行う
 - 全 1-nucleotide と 2-nucleotide の配列の数と割合を出し、ソートして出力
 - 全 [346]12|18-nucleotide の配列を数え、そのうち特定の一部の配列を出力

k-nucleotide (2)

サイト掲載の結果

k-nucleotide						
source	secs	mem	gz	busy	cpu load	
PHP	20.15	254,792	1079	69.22	91%	96% 80% 76%
Node.js	15.82	396,716	1812	44.48	63%	47% 81% 90%

手元での計測結果

	JIT無効	JIT有効
PHP	15.716s	13.878s
Node	74.128s (--jitless)	16.837s

- ・ サイトの方はPHPが負けてるが
- ・ 手元で実行するとPHPの方が勝つ
- ・ やったか？

悲しいお知らせ

k-nucleotide(3)

- PHP コードは pcntl_fork で 7 並列になっている
- JS 側は 4 並列
- 俺のマシンは 8 コア
- サイトの結果は 4 コアでの奴
- やってること全然違ってた！
- どっちのコードに実装あわせるのも少し大変なので諦め

悲しみは続く

- このサイトの多くの JS コードが Worker Threads でマルチスレッド
- どうにか条件あわせやすいやつで比較するしかない

並列実行環境の違い (1)

- Node 側は Worker Threads
- スレッド + メッセージング
- V8 isolate でスレッドごと VM 状態を持つ
- 基本はデータ非共有だが、メッセージング + メモリ共有が使える

並列実行環境の違い (2)

- PHP 側は **krakjoe/parallel**
- ZTS にもとづきスレッドごとに VM 状態、データ非共有
- Channel でメッセージング
- 偶然にも Worker Threads と似た構成

並列実行環境の違い (3)

- PHP 側に SharedArrayBuffer 相当がない
 - データ共有が足を引っ張る
- PHP 側がイベントドリブンではない
 - I/O と他スレッドのメッセージを同時に待てない

binary-trees (1)

- <https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/binarytrees.html>
- メモリアロケーションに重点を置いたベンチマーク
- ツリーノードを扱うデータ構造と処理を定義
- 「二分木の生成 + 内容検査 + 解放」 *たくさん
- 他より大きなサイズの二分木を 1つ生成・検査・解放
- 他の木の生成解放の間ずっと生きる二分木を 1つ生成・検査・解放

binary-trees (2)

binary-trees

source	secs	mem	gz	busy	cpu load
PHP	18.64	1,588,704	760	67.29	89% 90% 96% 86%
Node js	7.20	1,282,116	744	20.39	83% 70% 68% 63%

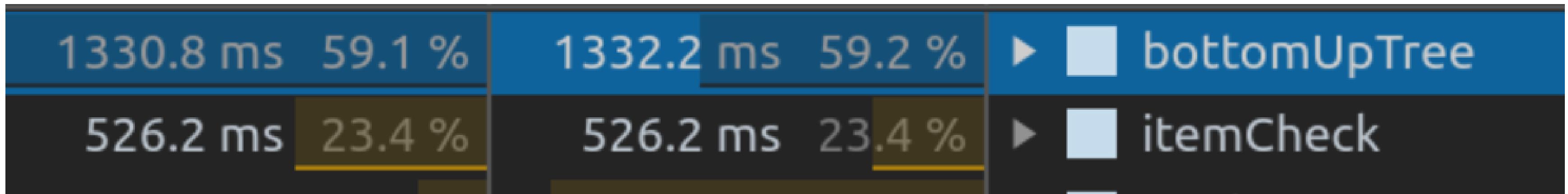
- サイト結果では PHP 側で倍以上の時間
- PHP 側既存実装は `pcntl_fork` で並列化
- データ共有負荷の影響は小さそう
- node 側と大体等価なマルチスレッドコードを用意
- ただし PHP 側コードは各スレッド冒頭で `gc_disable()` している
 - 循環参照が起きないと分かってるため

binary-trees (3)

	JIT無効	JIT有効
pcntl_fork (マルチプロセス)	18.074s	16.993s
parallel (マルチスレッド)	17.930s	15.964s
Node (マルチスレッド) (--jitless)	24.944s	6.636s

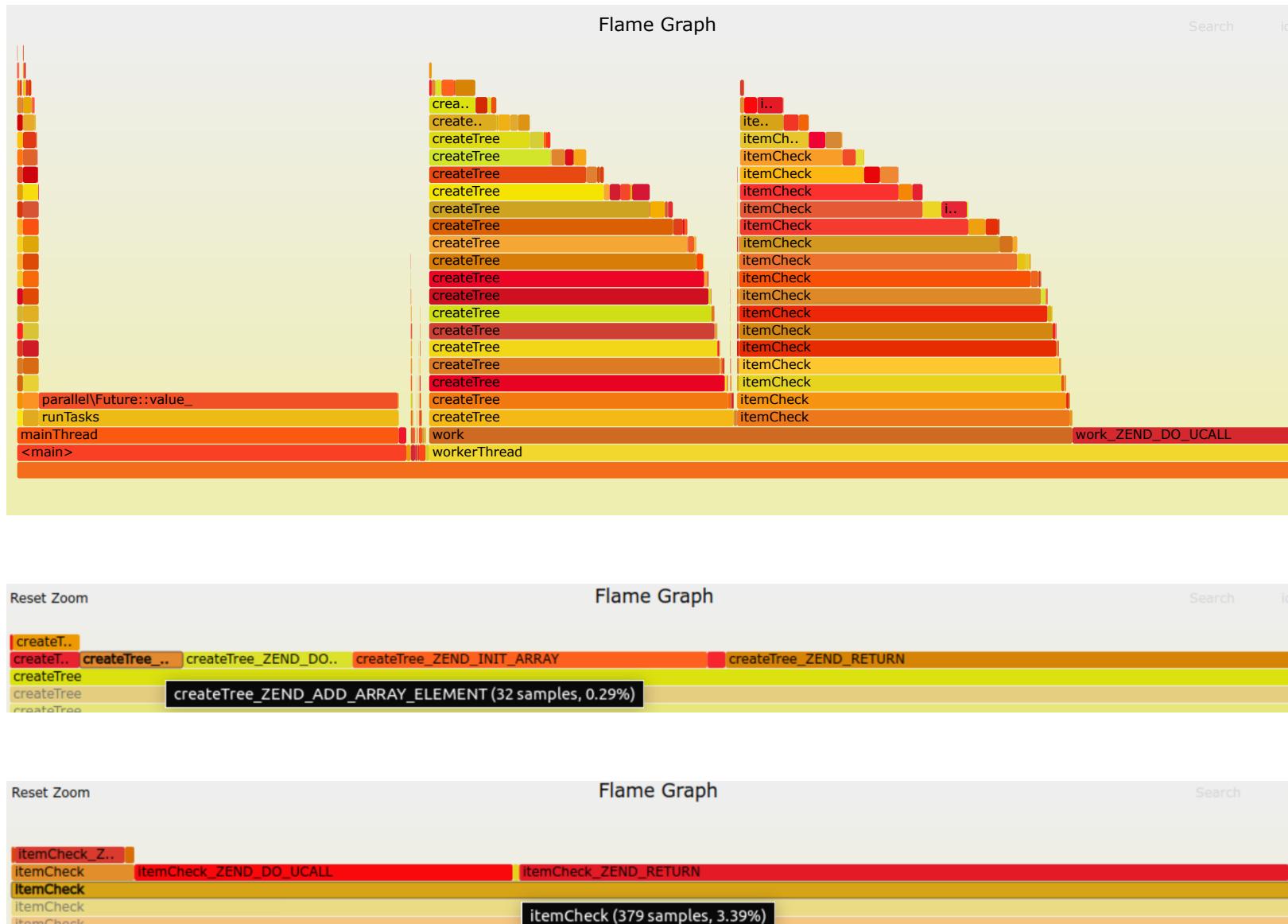
- 手元のマシンで元の fork 版 / スレッド版をそれぞれ計測
- スレッド版で pcntl_fork 版と実行性能はあまり変わらず
- Node 版との比較はしやすくなった

binary-trees: V8 プロファイル結果



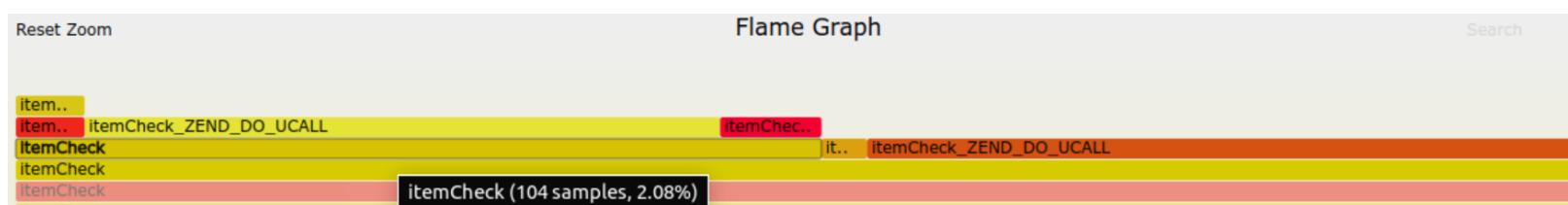
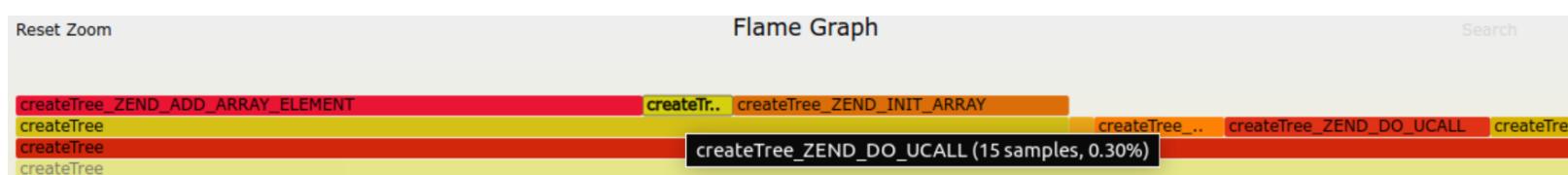
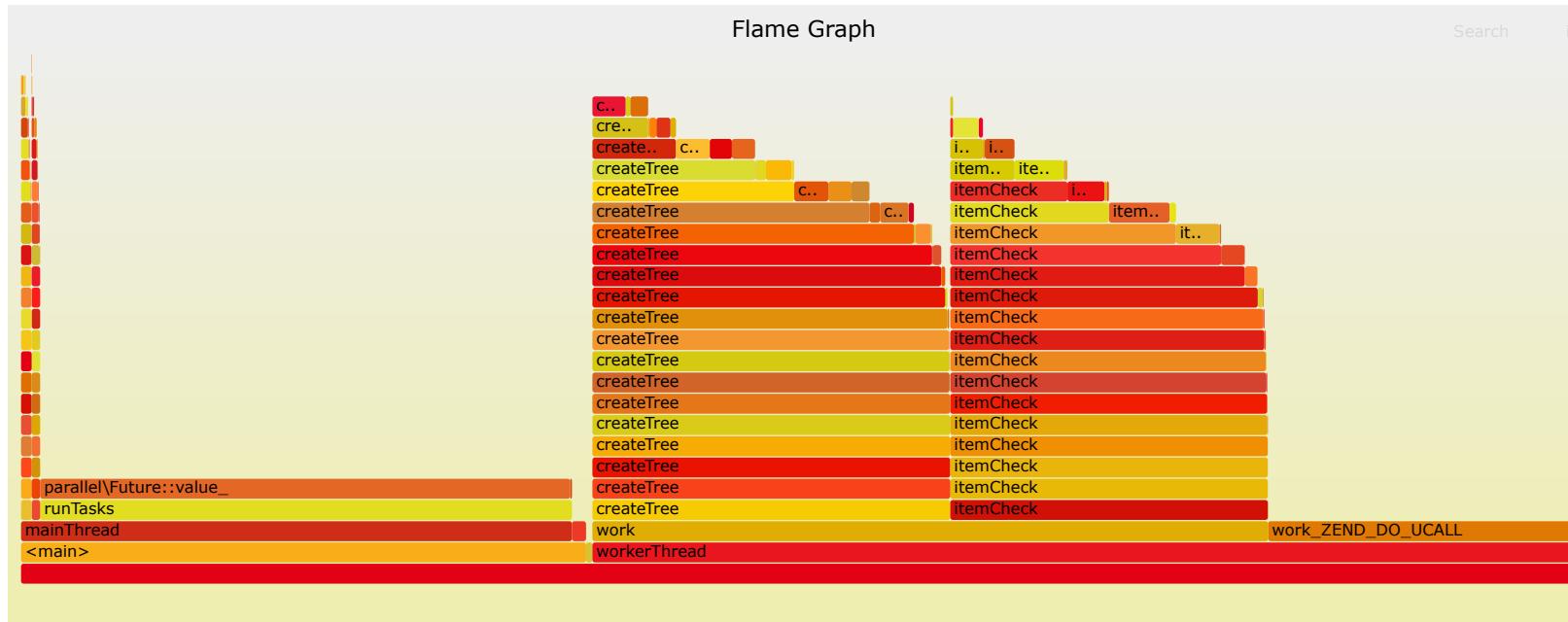
- ツリー構築の処理（`bottomUpTree`）が遅い
- 次に内容検査（`itemCheck`）が遅い

binary-trees: PHP8 プロファイル結果



- 全スレッドのサンプルを集約してフレームグラフに
- コールスタックの末端のみVM命令も添える
- 木の生成と検査コストが同等
- ZEND_DO_UCALL と ZEND_RETURN が目立つ
- ZEND_INIT_ARRAY と ZEND_ADD_ARRAY_ELEMENT も

binary-trees: PHP8 + JIT プロファイル結果



- JIT 有効化で `ZEND_RETURN` は見えなくなる
- `ZEND_INIT_ARRAY` と `ZEND_ADD_ARRAY_ELEMENT` と `ZEND_DO_UCALL` は残る
- それぞれの出現割合もあまり変わらず

binary-trees (考察1)

- 関数呼び出し自体のコストが一定ある
- 再帰をループに展開すれば速くなる可能性
- ツリー構築側はベンチマークのルール上の縛りでいじりづらい
 - アロケーションを減らしてはならない
- 検査部分の再帰はループに展開できる、JIT 有効時に数秒速くなる
 - 約 16 秒 → 約 12 秒に

binary-trees (考察2)

- それでも Node とは 2 倍差
- PHP8 の 関数呼び出し vs V8 の関数呼び出し
- PHP8 の 配列生成 vs V8 のオブジェクト生成
 - なお PHP 側を連想配列やオブジェクトにするともっと差がつく

binary-trees (※phpcon2021 後の追試)

- 追試の結果、一番速いのはオブジェクトのケースだった
- 最初に計測した際にかミスがあったっぽい

	JIT無効	JIT有効
配列	17.930s	15.964s
オブジェクト	14.402s	11.734s
連想配列	20.416s	18.159s

binary-trees (考察3)

これを

```
function createTree(int $depth): array
{
    if (!$depth) {
        return [null, null];
    }
    $depth--;
    return [createTree($depth), createTree($depth)];
}
```

JIT 有効時 15.964s、無効時 17.930s

こうしても挙動は保てる

```
function createTree(int $depth): array
{
    $tree = [null, null];
    while ($depth--) {
        $tree = [$tree, $tree];
    }
    return $tree;
}
```

JIT 有効時 1.142s、無効時 2.693s

- PHP の配列は値型で Copy on Write
- 葉の即時のアロケーションをしなくてもプログラムの意味を保てる
- キャッシュヒット率が影響してか無修正の検査部分も速くなる

binary-trees (考察4)

- このツリー構築 자체は実は PHP で非常に手軽に高速にできる
 - ベンチマークのレギュレーション違反にはなる
- 要件と言語特性を 1つ1つ把握して良い形を探るのが大事

まとめ (1)

- PHP の C 部分は速い
- C 部分は JIT で速くならない
- 負荷性質の見極めが大事

まとめ (2)

- JIT よりまず並列化
 - CPU 的な速度を求めるなら今後も模索が必要
- スレッドも使う手はある
 - メモリの共有方法はない
 - I/O とスレッドを同時に待つイベントループもない

まとめ (3)

- CPU 的な負荷に限っても JIT は銀の弾丸ではない
 - 配列の確保や要素追加にはあまり効かない
 - 関数呼び出しコストは一部減らせつつ、減らない部分も大きい
- VM命令レベルでの計測で最適化のヒントを得られる場合はある
 - 何をどう速くできるか、マイクロオプティマイザーは知見をためていくべし

それで真の 8 は？

- ~~そもそも真の 8 って何だ~~
- 少なくとも PHP にまだまだ伸びしろがあるのは分かった
- 実際 PHP 8 の JIT はいまだ現在進行系で性能向上が続いている

宣伝

- 10/24 発売の WEB+DB PRESS vol.125 に phpspy の使い方とか
- php-profiler の元ネタツールに興味があれば是非

おしまい