

# FIT PEACHES

Výpočet směrodatné odchylky  
Zpráva z profilingu

19. dubna 2019

## Úvod

Protože jsme si zvolili pro vývoj aplikace včetně matematické knihovny, již bylo nutné ve výpočtu směrodatné odchylky využít, JavaScript, nemohli jsme použít doporučený postup pro jazyk C a podobné a museli jsme sáhnout po jiných metodách profilingu.

Pokusili jsme se použít doporučovaný nástroj Clinic.js<sup>1</sup>, nicméně při snahách profilovat výpočet směrodatné odchylky o 10, 100 a 1000 vstupech nám hlásil, že běh programu byl tak krátký, že není co analyzovat. Provedli jsme tedy jedno profilování výpočtu směrodatné odchylky o milionu vstupů a jedno o deseti milionech vstupů, více informací dále v kapitole 4.

Protože s tímto nástrojem nebylo možné splnit zadání, museli jsme najít jiné řešení. Po testování různých *profilérů* jsme se rozhodli využít **easy-profiler**<sup>2</sup>.

## 1 Způsob profilování

Knihovna a kód pro profilování je zanesen přímo v souboru **profiling.js**. Ať už bude výpočet spouštěn pomocí příkazu **node** nebo po přeložení klasickým způsobem (např. **./profiling**, **profiling** a výstup profileru je možné zapnout jednoduše přidáním argumentu **--profile** programu.

Sami jsme pro dodání čísel pro výpočet využili generátor desetinných čísel. Vždy jsme si tedy nejprve vygenerovali soubory se seznamem číselných vstupů prostřednictvím generátoru. Generovaná čísla byla v intervalu  $<-10000;10000>$  s desetinnou přesností na 4 místa.

## 2 Profilování 10, 100 a 1000 vstupů

Při deseti a sto vstupech se neukazoval žádný zásadní rozdíl v době vykonávání funkcí a cyklů ve výpočtu odchylky.

Na následujících výstupech profileru je patrné, že běh byl velice rychlý. Teoreticky je možné konstatovat, že při takto malém počtu vstupů není ani co profilovat (jak nám řekl i Clinic.js, který jsme chtěli využít pro profilování původně).

Celkový běh v prvním případě nepřesáhl čtyři milisekundy, ve druhém případě pak nepřesáhl milisekund jedenáct.

```
jansvabik-mac:src jansvabik$ ./profiling --profile < /Users/jansvabik/Desktop/profiling/10.txt
6229.118332509895
```

Profiling Summary				
name	hitCount	duration(avg)	start	end
Line reading	1	4.000000000ms	1555684452857	1555684452861
Line processing	10	0.000000000ms	na	na
Average calculation	1	0.000000000ms	1555684452862	1555684452862
Mult. average power and the number count	1	0.000000000ms	1555684452862	1555684452862
Power summarize loop	1	0.000000000ms	1555684452862	1555684452862
Calc. finishing (root, dividing, subtr.)	1	0.000000000ms	1555684452862	1555684452862

Obrázek 1: Profilování vstupu o deseti číslech

<sup>1</sup><https://www.npmjs.com/package/clinic>

<sup>2</sup><https://www.npmjs.com/package/easy-profiler>

```
jansvabik-mac:src jansvabik$ ./profiling --profile < /Users/jansvabik/Desktop/profiling/100.txt
5664.542437378926
```

Profiling Summary				
name	hitCount	duration(avg)	start	end
Line reading	1	7.000000000ms	1555684501313	1555684501320
Line processing	100	0.020000000ms	na	na
Average calculation	1	2.000000000ms	1555684501320	1555684501322
Mult. average power and the number count	1	0.000000000ms	1555684501322	1555684501322
Power summarize loop	1	0.000000000ms	1555684501322	1555684501322
Calc. finishing (root, dividing, subtr.)	1	0.000000000ms	1555684501322	1555684501322

Obrázek 2: Profiling vstupu o sto číslech

Při profilování výpočtu o tisíci vstupech začíná být patrné, že nejvíce času trvá načítání dat. Je možné, že to je způsobeno čtením ze souboru, které je logicky vždy mnohem pomalejší než práce s operační pamětí.

V našem případě samotné načítání trvalo 32 milisekund, což je, dle našeho názoru, poměrně hodně (když vezmeme v potaz, že se načítalo *jen* tisíc čísel).

Celkový běh programu pro výpočet směrodatné odchylky o tisíci vstupech pak trval 41 milisekund. Načítání čísel k výpočtu tedy trvalo 78 % času – to jsou téměř čtyři pětiny času běhu programu.

```
jansvabik-mac:src jansvabik$ ./profiling --profile < /Users/jansvabik/Desktop/profiling/1000.txt
5708.1850016533635
```

Profiling Summary				
name	hitCount	duration(avg)	start	end
Line reading	1	32.000000000ms	1555684603655	1555684603687
Line processing	1000	0.005000000ms	na	na
Average calculation	1	2.000000000ms	1555684603687	1555684603689
Mult. average power and the number count	1	0.000000000ms	1555684603689	1555684603689
Power summarize loop	1	2.000000000ms	1555684603689	1555684603691
Calc. finishing (root, dividing, subtr.)	1	0.000000000ms	1555684603691	1555684603691

Obrázek 3: Profiling vstupu o tisíci číslech

### 3 Možná řešení

Možností jak urychlit běh programu není mnoho – také proto, že jsme neshledali očividným, že některý z výpočtů (ať už v cyklech nebo mimo ně) trvá znatelně dlouhou dobu (a to ani při milionech vstupů).

Jedinou zjevnou možností se zdá být vyřešit problém s dobou načítání vstupů před zpracováním. Je však možné, že by se dalo některé výpočty vykonávat již při zpracování každého jednotlivého řádku (zejména výpočet druhé mocniny aktuálně načteného čísla). Protože je JavaScript asynchronní jazyk, měl by se tento výpočet vykonávat paralelně s načítáním vstupů a případnými dalšími výpočty mocniny. Takto by bylo možné běh programu nepatrně urychlit.

Jinou možností by bylo sáhnout po jiném programovacím jazyku – některém nižším, například C nebo C++, který bude pro tyto činnosti vhodnější než námi zvolený JavaScript.

## 4 Profilování programu s velmi vysokým počtem čísel

Protože nás zajímal i výstup jiného profileru, využili jsme zmiňovaný nástroj Clinic.js. Ten kromě zobrazování latence v rámci zpracovávání cyklů zobrazuje také využití operační paměti v čase a využití procesoru v čase, což jsou další užitečné data potřebná pro zefektivňování algoritmu.

Níže jsou vloženy screenshoty z hlášení profileru. Protože jsme spouštěli profilování několikrát, v jednom případě se nám *podařilo* spatřit i chybovou zprávu, resp. předpokládaný problém v algoritmu (viz Obrázek 7).



Obrázek 4: Profiling vstupu o milionu čísel



Obrázek 5: Profiling vstupu o deseti milionech čísel

Z výsledků profileru je vidět, že procesor je zatěžován, rozdělíme-li běh programu na stejně velké části, průměrně rovnoměrně. Využití operační paměti naproti tomu stále roste, u deseti milionu čísel na vstupu pak zatelně poskočí. Nepodařilo se nám zjistit příčinu, je však možné, že jde o časový úsek po ukončení načítání vstupů.

Z výstupu **easy-profileru** pro sto tisíc vstupů je pak zjevné, že v načítání je opravdu zásadní problém. Při profilování vstupu se sto tisíce čísla trvalo načítání přibližně 30 sekund – zatímco výpočetním operacím stačilo v součtu 29 milisekund.

Zatížení jen načítáním dat se v tomto případě pohybuje již okolo 98,62 % z celkové doby běhu programu.

```
jansvabik-mac:src jansvabik$ ./profiling --profile < /Users/jansvabik/Desktop/profiling/100000.txt
5769.879054561299
```

Profiling Summary				
name	hitCount	duration(avg)	start	end
Line reading	1	28.957sec	1555690087682	1555690116639
Line processing	100000	0.0037700000ms	na	na
Average calculation	1	6.0000000000ms	1555690116640	1555690116646
Mult. average power and the number count	1	0.0000000000ms	1555690116646	1555690116646
Power summarize loop	1	23.0000000000ms	1555690116646	1555690116669
Calc. finishing (root, dividing, subtr.)	1	0.0000000000ms	1555690116670	1555690116670

Obrázek 6: Profiling vstupu o stu tisících čísel

Na obrázku následujícím je pak vidět dříve zmiňované hlášení pravděpodobného problému z Clinic.js.



Obrázek 7: Profiling vstupu o deseti milionech čísel – s hlášením problému

Vojtěch Dvořák  
 Lukáš Gurecký  
 Jan Švábík  
 Radim Zítka