Practical Perl Tools

Do I Look Better in Profile?

DAVID N. BLANK-EDELMAN

David N. Blank-Edelman is the director of technology at the Northeastern University College of Computer and Information Science and the author of the O'Reilly book *Automating System Administration with Perl* (the second edition of the Otter book), available at purveyors of fine dead trees everywhere. He has spent the past 24+ years as a system/network administrator in large multi-platform environments, including Brandeis University, Cambridge Technology Group, and the MIT Media Laboratory. He was the program chair of the LISA '05 conference and one of the LISA '06 Invited Talks co-chairs. David is honored to have been the recipient of the 2009 SAGE Outstanding Achievement Award and to serve on the USENIX Board of Directors beginning in June of 2010.
dnb@ccs.neu.edu

There may come a time in your Perl career when you find yourself asking, "Self..." (because what else would you say?), "Is there a way to make this Perl code run faster?" Many mental health professionals would say that talking to yourself is fine as long as you don't answer yourself, so let me do it on your behalf. When this time comes, you have at least a couple of options.

The first is to buy a faster machine. Unbox it, plug it in, run your code. Done. You may laugh, but that's not a bad solution sometimes. It can actually be a better option than the second one I'll propose: figure out why your code is running slowly and fix it. This option is often more fraught with peril than the first, because in the process of trying to optimize, there are many chances to introduce new bugs into the code. I could quote Knuth at you in an attempt to further scare you away, but then I don't think you'd read the rest of this column.

So how do you find out just where your code lags and how to fix it? The answer to this question in general is one of those lifelong quests. You can strive to even be a better coder, you can become a performance geek, or, heck, you can conduct basic research on how to improve programming in general. I salute those of you who already headed down any of these paths. In this column I'm going to try to help anyone else who is a Perl programmer to take another step or two in this direction.

First, I recommend that you check out Richard Foley's documentation shipped with later versions of Perl (5.10.1+) by typing "perldoc perlperf". The documentation is a little out-of-date, but it is a good start. Some of this column will overlap with the doc, but we're going to spend much more time on the current best practices that have evolved since 2008, when the doc was first written.

Benchmarks

Before we can directly answer the question "Why does my code run slower than I'd like?" I think it is important to bring our legs into lotus position and spend a bit of time meditating on some fundamental questions such as:

- What is "slow"?
- How will I know "slow" when I see it?
- Can I prove something is slow?
- Can I make "slow"?

I realize these all sound a bit more peyote-influenced than Perl-influenced, but bear with me. We need to be able to find a way to time just how fast a piece of Perl

code runs. Once we have that, we need to be able to change things and see how that compares to the time it took to run the first version. Perhaps we want to see several different versions of the same code go head-to-head so we can start to get a better sense of what makes for a slow or fast implementation. I realize there is a bit of handwaving in the last statement, because it is certainly possible to speed up some code without having the foggiest idea just how you did it, but let's assume the best for the moment.

The easiest way to start with this stuff is to benchmark your code. The canonical way to do this is to use a module that has shipped with Perl ever since it graduated to version 5: Benchmark.pm. It includes the following routines (as described by its documentation):

timethis: run a chunk of code several times
timethese: run several chunks of code several times
cmpthese: print results of timethese as a comparison chart
timeit: run a chunk of code and see how long it goes
countit: see how many times a chunk of code runs in a given time

Most often you see people using timethis() or timethese() to see how long a piece of code or several pieces take to run many, many times. Modern machines are so fast these days it often requires a huge number of runs of a piece of code to get a good handle on just how fast that code might be (and to eliminate anomalies in the testing environment). Here's an example:

```
use Benchmark;
use Math::Random::ISAAC::XS;
use Math::Random::ISAAC::PP;

my $time = time();

our $xrng = Math::Random::ISAAC::XS->new($time);
our $prng = Math::Random::ISAAC::PP->new($time);

my $count = 10_000_000;
timethese($count, {
         'XS' => '$xrng->rand()',
         'PP' => '$prng->rand()',

       });
```

In this example, we load the two modules that implement the very cool ISAAC pseudo-random number-generator algorithm. The first is a Perl plus C code version, and the second implements it entirely in Perl. We then specify how many times we plan to run the test code ($count). The number 10 million here doesn't have any deep significance. I just started with 1,000 and added zeroes to it until Benchmark.pm stopped complaining about the code not running long enough to get a reliable test (ISAAC is fast). We then look to timethese() to run both the XS and the PP subroutines, generating 10 million random numbers each. The program runs and spits out a very nice result:

```
Benchmark: timing 10000000 iterations of PP, XS...
       PP: 32 wallclock secs (32.58 usr + 0.01 sys = 32.59 CPU)
          @ 306842.59/s (n=10000000)
       XS: 2 wallclock secs ( 2.34 usr + 0.00 sys = 2.34 CPU)
          @ 4273504.27/s (n=10000000)
```

As expected, the XS version is *much* faster. For fun, I ran this sample code with a $count of a hundred million. The difference in speed is even more pronounced:

```
Benchmark: timing 100000000 iterations of PP, XS...
        PP: 321 wallclock secs (320.46 usr + 0.11 sys = 320.57 CPU)
            @ 311944.35/s (n=100000000)
        XS: 23 wallclock secs (23.54 usr + 0.02 sys = 23.56 CPU)
            @ 4244482.17/s (n=100000000)
```

So there you have it, the very basics of how to do benchmarking in Perl. And I do mean "basics." Understanding how to really benchmark code (or anything) is a very detailed and intricate art/science. I bow in the direction of the people who do that for a living.

Profiling

Let's get back to the original question: "Why does my code run slower than I'd like?" We had to talk about benchmarking because it is an important tool for being able to interpret and act upon the results of the process we really want to look at: profiling. Profiling is the process of instrumenting a pile of code to determine just how long each part of that code took to run (and how often it was run). With that data it becomes easier to determine the parts of your code you could change to improve the performance.

The tricky thing with profiling is determining just what "how long/often" actually means. I know this is starting to sound like the contemplative moment from the first section, but semantics here really do matter. For example, do you care how busy you are keeping the machine (raw CPU time) or how long you should expect to be tapping your foot waiting for the job to complete (real time)? A script can take almost no CPU time, but take eons to finish running if it is waiting for data to come in from a slow outside source or if the machine itself is bogged down. Which of these two things you care about at any one time really depends on the circumstances.

If you think that's a trick question, let's continue splitting gigantic, important hairs and ask the following: Do you care how long each specific statement in a program takes to run or how long various parts of your code as a whole (e.g., the subroutines) take? And if you care about the latter, do you want to know the total time for each subroutine, including any calls it made, or do you care only about how long just the code in that routine took? Or maybe you care about all of this? (If you don't care about any of this, see you next column!)

Luckily all of this information is available to you using current Perl tools...well, more precisely, a single tool. Over the years there have been a number of profiling tools written for Perl, but unless you have a good reason outside of the normal use case, there's really only one that you'll want to consider using. Even though there is a profiling tool that ships with Perl (Devel::DProf, which has been deprecated in the latest Perl distributions), the tool of choice here is Devel::NYTProf. Devel::NYTProf is currently maintained by Tim Bunce, a name you might recognize because he's the author of the Perl DBI framework.

Just to set your expectations accordingly, if I were paid by the word to write this column, I wouldn't be very happy with this tool. Yes, you can tweak how it works with various flags, but I've never had to use them. This is one of those tools that just work well right out of the box. I won't have to go into a huge amount of detail on

how to get the most out of Devel::NYTProf because it tries to give its all every time you use it. Remember all of the questions above about what sort of information you might want to collect when profiling? Devel::NYTProf provides all of them by default. It's a lovely tool, really.

The first step for using Devel::NYTProf is to run the code you want to profile, but to do so in a way that Devel::NYTProf gets loaded first so it can do its magic. Perl offers a few ways to do this, including:

```
perl -d:NYTProf yourcode.pl       # run that Devel module as the debugger
                                    code
PERL5OPT=-d:NYTProf               # set this environment variable and then
                                    run the code
perl -MDevel::NYTProf yourcode.pl # load the module first
```

Let's actually run Devel::NYTProf on the previous code we used in the benchmarking session:

```
$ perl -d:NYTProf maevebenchy.pl
Benchmark: timing 10000000 iterations of PP, XS...
       PP: 127 wallclock secs (127.21 usr + 0.22 sys = 127.43 CPU)
           @ 78474.46/s (n=10000000)
       XS: 24 wallclock secs (23.12 usr + 0.03 sys = 23.15 CPU)
           @ 431965.44/s (n=10000000)
```

Doesn't look like anything happened. Yes, the run times got slower (running under Devel::NYTProf does extract a bit of a performance penalty) but nothing else was immediately visible. However, if we look at the same directory our script is in we will see a new file there:

```
$ ls -l
total 66704
-rw-r--r-- 1 dnb dnb         343    Jul 25    15:20 maevebenchy.pl
-rw-r--r-- 1 dnb dnb    34147379    Jul 25    15:35 nytprof.out

$ file nytprof.out
nytprof.out: data
```

Devel::NYTProf has created a compressed file of profiling data. To actually use this profiling data, we have to convert it to a more useful format or a report of some sort. The distribution ships with three utilities for this purpose:

nytprofcg: Convert an NYTProf profile into Callgrind format
nytprofcsv: Devel::NYTProf::Reader CSV format implementation
nytprofhtml: Generate reports from Devel::NYTProf data

The first utility lets you put it into a format that the cool KCachgegrind utility can read. This GUI utility builds on Linux and OS X (via MacPorts or Homebrew) and Windows (see http://sourceforge.net/projects/precompiledbin/ for a pre-built version for Windows). It shows you the profiling data, call chains, and other stuff in a very pretty format. The second spits it out in CSV format, which might be useful if you are inclined to further process the data with some other tool. When I use Devel::NYTProf, I almost always use the HTML reports it provides, so let's choose that option:

```
$ nytprofhtml
Reading nytprof.out
```

```
Writing sub reports to nytprof directory
   100% ...
Writing block reports to nytprof directory
   100% ...
Writing line reports to nytprof directory
   100% ...
```

If we look now at the directory, we see:

```
$ ls
maevebenchy.pl     nytprof     nytprof.out
```

The "nytprof" entry is a directory with a bunch (in my case 77) HTML and .dot files in it. The .dot files are Graphviz source files (a subject we've talked about in past columns), which you can translate into pretty pictures of call graphs and such using the utilities in that package.

If we open up the index.html file in a browser, we see the Devel::NYTProf top-level report, which includes something like Figure 1 (for space reasons, I've cropped the page so you can see just the first half of the report):

Profile of maevebenchy.pl for 125s (of 181s), executing 160483224 statements and 30044310 subroutine calls in 16 source files and 6 string evals.

Top 15 Subroutines

Calls	P	F	Exclusive Time	Inclusive Time	Subroutine
10000000	1	1	29.6s	69.4s	Math::Random::ISAAC::PP::rand
1	1	1	23.6s	93.0s	Benchmark::__ANON__[(eval 5)[Benchmark.pm:426]:11]
10000000	1	1	20.2s	39.8s	Math::Random::ISAAC::PP::irand
39063	2	1	19.6s	19.6s	Math::Random::ISAAC::PP::_isaac
1	1	1	18.9s	24.3s	Benchmark::__ANON__[(eval 7)[Benchmark.pm:426]:11]
2	1	1	8.06s	8.06s	Benchmark::__ANON__[(eval 4)[Benchmark.pm:426]:11] (merge of 2 subs)
10000000	1	1	5.45s	5.45s	Math::Random::ISAAC::XS::rand (xsub)
2495	3	1	12.3ms	14.5ms	Benchmark::new
4	2	1	6.93ms	125s	Benchmark::runloop
1	1	1	3.56ms	16.9ms	main::BEGIN@4
1	1	1	3.14ms	5.73ms	Benchmark::BEGIN@432
1	1	1	2.38ms	7.41ms	Benchmark::BEGIN@453
1	1	1	2.34ms	2.37ms	Carp::BEGIN@4
2495	1	1	2.12ms	2.12ms	Benchmark::mytime
4	2	1	1.56ms	1.64ms	Exporter::as_heavy

See all 178 subroutines

Figure 1: Part of the index page from the HTML report generated by nytprofhtml

If we click on one of the subroutines, we can drill down into it and see more detail like that found in Figure 2 (again cropped for size).

129					# spent 39.8s (20.2+19.6) within Math::Random::ISAAC::PP::irand which was called 10000000 times, avg 4µs/call: # 10000000 times (20.2s+19.6s) by Math::Random::ISAAC::PP::rand at line 118, avg 4µs/call sub irand {
130	10000000	1.28s			my ($self) = @_;
131					
132					# Reset the sequence if we run out of random stuff
133	10000000	1.25s			if (!$self->{randcnt}--)
134					{
135					# Call method like this because of our hack above
136	39062	27.9ms	39062	19.6s	_isaac($self); # spent 19.6s making 39062 calls to Math::Random::ISAAC::PP::_isaac, avg 501µs/call
137	39062	14.2ms			$self->{randcnt} = 255;
138					}
139					
140	10000000	24.1s			return sprintf('%u', $self->{randrsl}->[$self->{randcnt}]);
141					}

Figure 2: Drilling down into the Devel::NYTProf report

As you can see, Devel::NYTProf is giving us a ton of data about what is running and how long it takes. We can click on lots of links in the reports to look deeper into what it has found. Unfortunately, in some ways, this was not the best code to

profile, because it is highly artificial. We are intentionally running two specific subroutines 10 million times (via timethese() in Benchmark.pm), so it is no surprise that they dominate the profiling results.

If we were to simplify the code to just this:

```
use strict;

use Math::Random::ISAAC::PP;

my $time = time();

our $prng = Math::Random::ISAAC::PP->new($time);
print $prng->rand();
```

and run Devel::NYTProf on it, we might get a better sense of what parts of the code are taking up time. Figure 3 shows a shot of the result, cropped this time to show the second half of the index page.

Source Code Files — ordered by exclusive time then name

Stmts	Exclusive Time	Reports	Source File
29	50.9ms	line • block • sub	Carp.pm
7	32.2ms	line • block • sub	nonbench.pl
4240	27.5ms	line • block • sub	Math/Random/ISAAC/PP.pm (including 1 string eval)
40	315µs	line • block • sub	strict.pm
36	278µs	line • block • sub	warnings.pm
7	85µs	line • block • sub	Exporter.pm
5	22µs	line • block • sub	integer.pm
4364	111ms	*Total*	
623	15.9ms	*Average*	
	315µs	*Median*	
	0.00029	*Deviation*	

Figure 3: A more informative Devel::NYTProf result

I'd encourage you to run this on your own code. You'll get a much better sense of what it is doing. Once you have that understanding, you can begin to improve it. For more information on Devel::NYTProf and how to improve code using it, I highly recommend Bunce's talk on v4 of Devel::NYTProf, a screencast of which can be found here: http://blip.tv/timbunce/devel-nytprof-v4-oscon-201007-3932242.

Take care, and I'll see you next time.