Memory usage and load time of Moose, Moo, and Class::Accessor

The other day I wrote about checking the memory usage of a Perl script. In that article we saw the memory footprint of some simple data, but have not looked at modules. Let's see how can we check the memory footprint and load time of modules.

Especially interesting are Moo and Moose, the two main object systems of Perl, and Class::Accessor that used to be a big hit before Moose came on the scene.

The comparison won't be very scientific and will only check the loading of the main modules of each system.

```
1.  use strict;
2.  use warnings;
3.  use 5.010;
4.
5.  use Memory::Usage;
6.
7.  my $mu = Memory::Usage->new();
8.  $mu->record('starting work');
9.
10. require Moose;
11.
12. $mu->record('after creating variable');
13.
14. $mu->dump();
```

In the article about memory usage you can see how the Memory::Usage module works. The only difference is that in this case, instead of some code, we load a module between the two calls of record. We do that using the require statement instead of the use statement as the former loads the module only at run-time.

We save the script as memory.pl and then run it as time memory.pl

Moose

The script prints:

```
  time    vsz (  diff)    rss (  diff) shared (  diff)   code (  diff)
data (  diff)

     0  18688 ( 18688)   2384 (  2384)   1756 (  1756)   1500 (  1500)
916 (   916) starting work
```

```
     0  54572 ( 35884)  17480 ( 15096)   2252 (   496)   1500 (     0)
15664 ( 14748) after creating variable
```

and the time command prints:

```
real    0m0.266s

user    0m0.248s

sys     0m0.016s
```

Moo

The same after replacing Moose by Moo:

```
  time    vsz (  diff)    rss (  diff) shared (  diff)   code (  diff)
data (  diff)

     0  18688 ( 18688)   2384 (  2384)   1756 (  1756)   1500 (  1500)
916 (   916) starting work

     0  29008 ( 10320)   4444 (  2060)   2048 (   292)   1500 (     0)
2900 (  1984) after creating variable

real    0m0.035s

user    0m0.030s

sys     0m0.004s
```

Class::Accessor

Class::Accessor is used by many people who want a basic constructor and default accessors to attributes without all the fancy stuff Moo and Moose provide.

```
  time    vsz (  diff)    rss (  diff) shared (  diff)   code (  diff)
data (  diff)

     0  18688 ( 18688)   2384 (  2384)   1756 (  1756)   1500 (  1500)
916 (   916) starting work

     0  21684 (  2996)   3456 (  1072)   1916 (   160)   1500 (     0)
1852 (   936) after creating variable

real    0m0.020s

user    0m0.016s
```

```
sys     0m0.004s
```

Baseline

Finally, here is the baseline script that does not load any of these modules:

```
  time    vsz (  diff)    rss (  diff) shared (  diff)   code (  diff)
data (  diff)

     0  18688 ( 18688)   2384 (  2384)   1752 (  1752)   1500 (  1500)
916 (   916) starting work

     0  18688 (     0)   2384 (     0)   1752 (     0)   1500 (     0)
916 (     0) after creating variable

real    0m0.007s

user    0m0.007s

sys     0m0.000s
```

Let's compare the results:

```
                  Time     vsz    rss

Baseline          0.007      0      0

Class::Accessor   0.029   2996   1072

Moo               0.035  10320   2060

Moose             0.266  35884  15096
```

Of course Moose provides much more than Moo does, which in turn provides more than what Class::Accessor provides so it is not surprising Moose "cost" the most.

Late loading

What this little exercise can also show you is, that you can save on start-up time if you can delay loading some of the optional modules. Let's say you need 2 modules in your script A and B, but in most cases you will only need one of them not both. If you write

```
1.  use A;
2.  use B;
```

they will be both loaded at compile time on **every** run.

On the other hand if you write something like this:

```
1.
2.   if (a_is_needed) {
3.       require A;
4.   }
5.   if (b_is_needed) {
6.       require B;
7.   }
```

then at compile none of the modules will be loaded and during run-time only the module that is really needed will be loaded.

It can save both time and memory.