Article
Fabian Haupt · Dec 22, 2016 8m read

On the importance of benchmarking your code

I was benchmarking the populate utils when I noticed something strange. Consider this simple benchmarking method:

```
ClassMethod runBench(count As %Integer = 300000)
{
        s types=["name","ssn","company","title","phone","city","street","zip","missio
n","state","color","product","string","integer","float"]
        //s types=["name","ssn"]
        s iter=types.%GetIterator()
        while iter.%GetNext(.key,.value){
            s start=$P($ZTS,",",2)
            s opt={}
            s opt.type=value
            for i=1:1:count{
                s res=##class(DataGen.Generator).genVal(opt)            }
            s end=$P($ZTS,",",2)
            s total=end-start
            w value_$C(9)_" total time:"_total_$C(9)_" rate:  "_$NORMALIZE(count/tota
l,2)_" per second",!
        }
}
```

With my genVal method being this:

```
ClassMethod genVal(value As %DynamicObject) As %String
{
        s ret="""_value.name_""""_":"
          if value.type="name" {
            q ret_""""_##class(%PopulateUtils).Name()_""""
          } elseif value.type="ssn" {
            q ret_""""_##class(%PopulateUtils).SSN()_""""
          } elseif value.type="company" {
            q ret_""""_##class(%PopulateUtils).Company()_""""
          } elseif value.type="title" {
            q ret_""""_##class(%PopulateUtils).Title()_""""
          } elseif value.type="phone" {
            q ret_""""_##class(%PopulateUtils).USPhone()_""""
          } elseif value.type="city" {
            q ret_""""_##class(%PopulateUtils).City()_""""
          } elseif value.type="street" {
            q ret_""""_##class(%PopulateUtils).Street()_""""
          } elseif value.type="zip" {
            q ret_""""_##class(%PopulateUtils).USZip()_""""
          } elseif value.type="mission" {
            q ret_""""_##class(%PopulateUtils).Mission()_""""
          } elseif value.type="state" {
```

```
            q ret_""""_##class(%PopulateUtils).USState()_""""
          } elseif value.type="color" {
            q ret_""""_##class(%PopulateUtils).Color()_""""
          } elseif value.type="product" {
            q ret_""""_##class(%PopulateUtils).Product()_""""
          } elseif value.type="string" {
            q ret_""""_##class(%PopulateUtils).String(5)_""""
          } elseif value.type="integer" {
            q ret_##class(%PopulateUtils).Integer(0,1000)
          } elseif value.type="float" {
            q ret_##class(%PopulateUtils).Float(,,3)
          }

    q $$$OK
}
```

I didn't expect anything from the results so at first glance they're fine:

```
USER>d ##class(DataGen.Benchmark).runBench()
name     total time:1.921        rate:  156168.66 per second
ssn      total time:1.297        rate:  231303.01 per second
company  total time:1.844        rate:  162689.8 per second
title    total time:1.969        rate:  152361.6 per second
phone    total time:1.968        rate:  152439.02 per second
city     total time:2.297        rate:  130605.14 per second
street   total time:2.578        rate:  116369.28 per second
zip      total time:2.563        rate:  117050.33 per second
mission  total time:3.453        rate:  86880.97 per second
state    total time:3.281        rate:  91435.54 per second
color    total time:3.391        rate:  88469.48 per second
product  total time:3.453        rate:  86880.97 per second
string   total time:3.906        rate:  76804.92 per second
integer  total time:3.844        rate:  78043.7 per second
float    total time:4.769        rate:  62906.27 per second
USER>
```

The increasing times for the different populate runs seemed a bit odd. And sure enough, when I move the float check at the top of the big elseif statement, the times look different:

```
name     total time:1.906        rate:  157397.69 per second
ssn      total time:1.469        rate:  204220.56 per second
company  total time:2.047        rate:  146555.94 per second
title    total time:2.14         rate:  140186.92 per second
phone    total time:2.156        rate:  139146.57 per second
city     total time:2.454        rate:  122249.39 per second
street   total time:2.89         rate:  103806.23 per second
zip      total time:2.906        rate:  103234.69 per second
mission  total time:3.532        rate:  84937.71 per second
state    total time:3.406        rate:  88079.86 per second
color    total time:3.562        rate:  84222.35 per second
product  total time:3.594        rate:  83472.45 per second
string   total time:4.062        rate:  73855.24 per second
integer  total time:4.141        rate:  72446.27 per second
float    total time:1.656        rate:  181159.42 per second
```

Now the question is: is the comparison slow, or the dynamic object dispatch for value.type?
And sure enough, after modifying my genVal method like this:

```
 s ret="""_value.name_"""_":"
        s type=value.type
          if type="name" {
        ....
```

I'm getting much better times (we call this point a now):

```
USER>d ##class(DataGen.Benchmark).runBench()
name     total time:1.968        rate:  152439.02 per second
ssn      total time:1.047        rate:  286532.95 per second
company  total time:1.265        rate:  237154.15 per second
title    total time:1.172        rate:  255972.7 per second
phone    total time:1.063        rate:  282220.13 per second
city     total time:1.109        rate:  270513.98 per second
street   total time:1.219        rate:  246103.36 per second
zip      total time:1    rate:  300000 per second
mission  total time:1.562        rate:  192061.46 per second
state    total time:1.203        rate:  249376.56 per second
color    total time:1.125        rate:  266666.67 per second
product  total time:1.11         rate:  270270.27 per second
string   total time:1.156        rate:  259515.57 per second
integer  total time:1.094        rate:  274223.03 per second
float    total time:1.437        rate:  208768.27 per second
```

Further playing with the order of comparisons doesn't seem to have a significant impact on the times.
However, moving the code from genVal into my benchmark method, gives us another rate increase of about 16%.
This means the dispatch time is also significant.
Of course, we sacrifice some readability in our code.

Now that we've gotten to this stage, we're going to have a look at the code if we have other hotspots we might be able to optimize a bit better. For this we're going to use MONLBL. This gives us a line-by-line overview of how much time we spent on each line. The various options and data points gathered are a bit much to discuss in the scope of this piece.

The abbreviated output from ^%SYS.MONLBL (some of the columns are hidden):

```
DataGen.Benchmark.1 55  4500000 4500000 4.066849    11.59207
DataGen.Benchmark.1 68  1200000 1200000 1.004845    3.388754
DataGen.Benchmark.1 83  900000  900000  0.648747    2.045229
DataGen.Benchmark.1 60  300000  300000  0.252473    1.663838
DataGen.Benchmark.1 72  300000  300000  0.293236    1.323061
DataGen.Benchmark.1 66  600000  600000  0.481358    1.316253
DataGen.Benchmark.1 57  4500000 0   1.30593 1.30593
DataGen.Benchmark.1 62  300000  300000  0.306045    1.154256
DataGen.Benchmark.1 64  300000  300000  0.28068 0.883587
DataGen.Benchmark.1 58  300000  300000  0.25206 0.799962
DataGen.Benchmark.1 70  300000  300000  0.261574    0.628042
DataGen.Benchmark.1 84  900000  0   0.199095    0.199095
DataGen.Benchmark.1 91  900000  0   0.158523    0.158523
DataGen.Benchmark.1 61  300000  0   0.058536    0.058536
```

Looking at the results, we see it points us to line 55 in my loop as the one taking the most time (last column):

s ret="""value.name""":"

Which, again, is the dynamic dispatch. Moving this out of the loop gets rid of that, without giving up the flexibility we get from using %DynamicObjects. This gets us to about a 68% higher rate than in point a. Our code now looks like this:

```
ClassMethod runBench(count As %Integer = 300000)
{
    s types=["name","ssn","company","title","phone","city","street","zip","mission","
state","color","product","string","integer","float"]
    //s types=["name","ssn"]
    s iter=types.%GetIterator()
    while iter.%GetNext(.key,.v){
        s start=$P($ZTS,",",2)
        s value={}
        s value.type=v
        s name=value.name
        s type=$E(value.type,1,3)
        for i=1:1:count{
            s ret="""_name_"""_":"
            if type="int" {
                s res=ret_##class(%PopulateUtils).Integer(0,1000)
            }elseif type="nam" {
                s res= ret_"""_##class(%PopulateUtils).Name()_"""
            } elseif type="flo" {
                s res= ret_##class(%PopulateUtils).Float(,,3)
            } elseif type="tit" {
                s res= ret_"""_##class(%PopulateUtils).Title()_"""
            } elseif type="pho" {
                s res= ret_"""_##class(%PopulateUtils).USPhone()_"""
            } elseif type="str" {
                s res= ret_"""_##class(%PopulateUtils).Street()_"""
            } elseif type="zip" {
                s res= ret_"""_##class(%PopulateUtils).USZip()_"""
            } elseif type="mis" {
                s res= ret_"""_##class(%PopulateUtils).Mission()_"""
            } elseif type="sta" {
                s res= ret_"""_##class(%PopulateUtils).USState()_"""
            } elseif type="ssn" {
                s res= ret_"""_##class(%PopulateUtils).SSN()_"""
            } elseif type="str" {
                s res= ret_"""_##class(%PopulateUtils).String(5)_"""
            } elseif type="com" {
                s res= ret_"""_##class(%PopulateUtils).Company()_"""
            } elseif type="col" {
                s res= ret_"""_##class(%PopulateUtils).Color()_"""
            } elseif type="cit" {
                s res= ret_"""_##class(%PopulateUtils).City()_"""
            } elseif type="pro" {
                s res= ret_"""_##class(%PopulateUtils).Product()_"""
            }
        }
        s end=$P($ZTS,",",2)
        s total=end-start
        w v_$C(9)_" total time:"_total_$C(9)_" rate:  "_$NORMALIZE(count/total,2)_" p
er second",!
```

```
    }
}
```

We still have the dynamic dispatch in there, and we wouldn't actually need it in this case, as we are calling the code directly. However, in the actualy application I'd still be using it to pass in more parameters to populate.

So spending a little time on the code being run the most in my application, we were able to improve the performance by over 83% (compared to the very first run). Either just simple timing benchmarks or more advanced methods like using ^%SYS.MONLBL are really useful in making sure your application is performing the best it can. Not only does this increase your potential maximum throughput, but it also means you need to invest less in hardware as you can handle a bigger load on the same capacity as before.

Final run:

```
USER>d ##class(DataGen.Benchmark).runBench()
name     total time:1.078        rate:  278293.14 per second
ssn      total time:.297         rate:  1010101.01 per second
company  total time:.437         rate:  686498.86 per second
title    total time:.328         rate:  914634.15 per second
phone    total time:.282         rate:  1063829.79 per second
city     total time:.328         rate:  914634.15 per second
street   total time:.344         rate:  872093.02 per second
zip      total time:.203         rate:  1477832.51 per second
mission  total time:.671         rate:  447093.89 per second
state    total time:.344         rate:  872093.02 per second
color    total time:.281         rate:  1067615.66 per second
product  total time:.282         rate:  1063829.79 per second
string   total time:.343         rate:  874635.57 per second
integer  total time:.297         rate:  1010101.01 per second
float    total time:.547         rate:  548446.07 per second
```

#Caché #Ensemble #ObjectScript #Performance #Tips & Tricks