Ruby master - Feature #16119

Optimize Array#flatten and flatten! for already flattened arrays

08/22/2019 11:58 PM - dylants (Dylan Thacker-Smith)

Status:	Closed
Priority:	Normal
Assignee:	nobu (Nobuyoshi Nakada)
Target version:	

Description

Problem

When doing an object profile from stackprof, I noticed object allocations being made from Array#flatten! which was unlike other in-place methods like Array#uniq! and Array#compact!. In this case, I wanted to optimize for the array already being flattened and found that ary.flatten! if ary.any?(Array) was significantly faster. The code confirmed my suspicion that ary.flatten! was almost equivalent to ary.replace(ary.flatten) in implementation. The object allocations I noticed were from a temporary result array, a stack array to handle nesting and a hash table to prevent infinite recursion, all of which can be avoided in the simple case of an already flattened array.

Solution

Iterate over the array to find the first nested array. If no nested array is found, then flatten! just returns nil without creating additional objects and flatten returns a shared copy of the original array. If a nested array is found, then it creates and initializes the temporary objects to resume with the existing algorithm for flattening the array.

Benchmark

```
require 'benchmark'

N = 100000

def report(x, name)
  x.report(name) do
    N.times do
      yield
    end
  end
end

arrays = {
  small_flat_ary: 5.times.to_a,
  large_flat_ary: 100.times.to_a,
  small_pairs_ary: [[1, 2]] * 5,
  large_pairs_ary: [[1, 2]] * 100,
}

Benchmark.bmbm do |x|
  arrays.each do |name, ary|
    report(x, "#{name}.flatten!") do
      ary.flatten!
    end
    report(x, "#{name}.flatten") do
      ary.flatten
    end
  end
end
```

results on the latest master (ruby 2.7.0dev (2019-08-22T14:10:55Z master fd20b32130) [x86_64-darwin18])

```
                              user     system      total        real
small_flat_ary.flatten!    0.082001   0.000294   0.082295 (  0.082685)
small_flat_ary.flatten     0.078655   0.000211   0.078866 (  0.079176)
```

```
large_flat_ary.flatten!     0.552551   0.001643   0.554194 (  0.556166)
large_flat_ary.flatten      0.551520   0.001327   0.552847 (  0.554091)
small_pairs_ary.flatten!    0.100073   0.000302   0.100375 (  0.100687)
small_pairs_ary.flatten     0.109440   0.000232   0.109672 (  0.109850)
large_pairs_ary.flatten!    1.021200   0.001650   1.022850 (  1.024227)
large_pairs_ary.flatten     1.049046   0.002938   1.051984 (  1.055228)
```

results with the attached patch

```
                                user     system      total        real
small_flat_ary.flatten!     0.034868   0.000150   0.035018 (  0.035180)
small_flat_ary.flatten      0.040504   0.000148   0.040652 (  0.040914)
large_flat_ary.flatten!     0.458273   0.000786   0.459059 (  0.460005)
large_flat_ary.flatten      0.453846   0.000833   0.454679 (  0.455324)
small_pairs_ary.flatten!    0.055211   0.000205   0.055416 (  0.055673)
small_pairs_ary.flatten     0.060157   0.000226   0.060383 (  0.060540)
large_pairs_ary.flatten!    0.901698   0.001650   0.903348 (  0.905246)
large_pairs_ary.flatten     0.917180   0.001370   0.918550 (  0.920008)
```

Associated revisions

Revision a1fda16b - 09/27/2019 04:24 PM - dylants (Dylan Thacker-Smith)

Optimize Array#flatten and flatten! for already flattened arrays (#2495)

- Optimize Array#flatten and flatten! for already flattened arrays
- Add benchmark for Array#flatten and Array#flatten!

[Bug #16119]

History

#1 - 08/23/2019 02:02 AM - duerst (Martin Dürst)

I was afraid that this would be an optimization for flat arrays, but increase time for nested arrays. But that's not the case, because at the bottom, there will be flat arrays, and flattening these will be faster.

However, I expect this to be slower on arrays that are 'almost flat', i.e.

```
almost_flat_ary = 100.times.to_a + [1, 2]
```

Putting the nested array at the end will make sure all elements of the big array are checked, only to discover that actual flattening work is needed. The time needed for checking will not be offset by the allocation savings on the small array at the end.

Still I think that in general, this should be faster, and so it should be worth accepting this patch.

#2 - 08/23/2019 06:31 AM - shevegen (Robert A. Heiler)

I use .flatten and .flatten! quite a lot in my ruby code, often setting the
initial input from ARGV but also from other sources, such as used from other
ruby classes, w hen I need to have an array - often something like:

```
def foobar(input)
  new_value = [input].flatten.compact
```

Or something like that. If this patch indeed improves this then that would
be quite nice to have.

#3 - 08/23/2019 07:41 AM - Hanmac (Hans Mackowiak)

i see in your patch that you not only check for Array but for to_ary too, which is nice

shevegen (Robert A. Heiler) :

instead of [input] i would use Array(input) that doesn't create an extra array if input is already one

#4 - 08/23/2019 03:32 PM - dylants (Dylan Thacker-Smith)

duerst (Martin Dürst) wrote:

> However, I expect this to be slower on arrays that are 'almost flat', i.e.
>
> ```
> almost_flat_ary = 100.times.to_a + [1, 2]
> ```

> Putting the nested array at the end will make sure all elements of the big array are checked, only to discover that actual flattening work is needed. The time needed for checking will not be offset by the allocation savings on the small array at the end.

The optimization handles this case by doing a quick ary_memcpy of everything up to the first nested array to avoid redundantly rechecking if those preceding elements were an array.

I benchmarked it by replacing arrays and Benchmark.bmbm in my above bechmark script with

```
ary = 100.times.to_a.push([101, 102])

Benchmark.bmbm do |x|
  report(x, "flatten!") do
    ary.flatten!
  end
  report(x, "flatten") do
    ary.flatten
  end
end
```

and got the following results on master

```
              user     system      total        real
flatten!   0.577976   0.002275   0.580251 (  0.582700)
flatten    0.542454   0.001994   0.544448 (  0.546523)
```

and the following results with this patch

```
              user     system      total        real
flatten!   0.420922   0.001052   0.421974 (  0.422957)
flatten    0.430728   0.001173   0.431901 (  0.433274)
```

so I actually noticed improved performance for arrays with a large flat prefix.

#5 - 08/23/2019 03:38 PM - dylants (Dylan Thacker-Smith)

Hanmac (Hans Mackowiak) wrote:

> i see in your patch that you not only check for Array but for to_ary too, which is nice

Right, but that is just preserving the current behaviour. So the ary.flatten! if ary.any?(Array) workaround could actually be a breaking change if the given array had elements that responded to to_ary other than Array.

#6 - 09/19/2019 08:30 AM - mame (Yusuke Endoh)

- Assignee set to nobu (Nobuyoshi Nakada)

- Status changed from Open to Assigned

nobu (Nobuyoshi Nakada) Could you please review this?

#7 - 09/20/2019 12:19 AM - nobu (Nobuyoshi Nakada)

This patch unrolls the while-loop for the already flatten head and seems reasonable.
But I could get only insignificant result, and a worse result for an unflattened array.

```
# array_flatten.yml
prelude: |
  ary = (0...100).to_a.push([101, 102])
  ary2 = 10.times.map {|i| (i..(i+9)).to_a}

benchmark:
  flatten!: ary.flatten!
  flatten: ary.flatten
  flatten2: ary2.flatten
loop_count: 1000000
```

```
            benchmark/array_flatten.yml
Calculating -------------------------------------
                         compare-ruby  built-ruby
             flatten!     220.944k    223.612k i/s -      1.000M times in 4.526028s 4.472037s
              flatten     216.457k    210.651k i/s -      1.000M times in 4.619859s 4.747182s
             flatten2     182.447k    166.501k i/s -      1.000M times in 5.481056s 6.005954s
```

```
Comparison:
                         flatten!
          built-ruby:    223611.7 i/s
        compare-ruby:    220944.3 i/s - 1.01x  slower

                          flatten
        compare-ruby:    216456.8 i/s
          built-ruby:    210651.3 i/s - 1.03x  slower

                         flatten2
        compare-ruby:    182446.6 i/s
          built-ruby:    166501.4 i/s - 1.10x  slower
```

#8 - 09/22/2019 06:29 PM - methodmissing (Lourens Naudé)

- File 0001-Let-empty-arrays-being-flattened-not-alloc-additiona.patch added

nobu (Nobuyoshi Nakada) wrote:

> This patch unrolls the while-loop for the already flatten head and seems reasonable.
> But I could get only insignificant result, and a worse result for an unflattened array.
>
> ```
> # array_flatten.yml
> prelude: |
>   ary = (0...100).to_a.push([101, 102])
>   ary2 = 10.times.map {|i| (i..(i+9)).to_a}
>
> benchmark:
>   flatten!: ary.flatten!
>   flatten: ary.flatten
>   flatten2: ary2.flatten
> loop_count: 1000000
> ```
>
> ```
>             benchmark/array_flatten.yml
> Calculating -------------------------------------
>                    compare-ruby  built-ruby
>             flatten!     220.944k    223.612k i/s -      1.000M times in 4.526028s 4.472037s
>              flatten     216.457k    210.651k i/s -      1.000M times in 4.619859s 4.747182s
>             flatten2     182.447k    166.501k i/s -      1.000M times in 5.481056s 6.005954s
>
> Comparison:
>                          flatten!
>           built-ruby:    223611.7 i/s
>         compare-ruby:    220944.3 i/s - 1.01x  slower
>
>                           flatten
>         compare-ruby:    216456.8 i/s
>           built-ruby:    210651.3 i/s - 1.03x  slower
>
>                          flatten2
>         compare-ruby:    182446.6 i/s
>           built-ruby:    166501.4 i/s - 1.10x  slower
> ```

Attached is a patch for early return and preventing additional allocs for the empty array case (however not very sure how often that happens in practice):

```
prelude: |
  ary = []

benchmark:
  flatten!: ary.flatten!
  flatten: ary.flatten
loop_count: 1000000
```

```
lourens@CarbonX1:~/src/ruby/ruby$ /usr/local/bin/ruby --disable=gems -rrubygems -I./benchmark/lib ./benchmark/
benchmark-driver/exe/benchmark-driver             --executables="compare-ruby::~/src/ruby/trunk/ruby --disable
=gems -I.ext/common --disable-gem"             --executables="built-ruby::./miniruby -I./lib -I. -I.ext/common
  -r./prelude --disable-gem" -v --repeat-count=24 $HOME/src/array_flatten.yml
compare-ruby: ruby 2.7.0dev (2019-09-22T17:21:54Z master 142efba93e) [x86_64-linux]
built-ruby: ruby 2.7.0dev (2019-09-22T18:10:18Z opt-flatten-empty-.. ae17889e1e) [x86_64-linux]
Calculating -------------------------------------
                    compare-ruby  built-ruby
           flatten!       6.234M     31.453M i/s -      1.000M times in 0.160418s 0.031794s
            flatten       6.271M     31.324M i/s -      1.000M times in 0.159468s 0.031924s
```

```
Comparison:
                          flatten!
          built-ruby:  31452589.1 i/s
        compare-ruby:   6233726.0 i/s - 5.05x  slower

                           flatten
          built-ruby:  31324068.7 i/s
        compare-ruby:   6270832.4 i/s - 5.00x  slower
```

#9 - 09/27/2019 03:42 AM - dylants (Dylan Thacker-Smith)

It looks like I made a mistake in my benchmarking of non-flattened arrays, since flatten! from the first iteration would cause further iterations to actually test with a flat array. It looks like the performance for arrays starting with a nested array are about the same with and without my patch.

The attached patch has a merge conflict. I've opened https://github.com/ruby/ruby/pull/2495 with a rebase of my patch that resolves the merge conflict. That PR also has an updated benchmark that uses the benchmark driver yaml format and includes the results from running that benchmark.

nobu (Nobuyoshi Nakada) wrote:

> But I could get only insignificant result, and a worse result for an unflattened array.

It looks like nobu's benchmark/array_flatten.yml has the same problem with the non-flat arrays being mutated and affecting following iterations. That explains the improvement I am seeing locally with your benchmark, although I don't know why my results differ from what you posted

```
Comparison:
                          flatten!
          built-ruby:    246214.2 i/s
        compare-ruby:    193998.4 i/s - 1.27x  slower

                           flatten
          built-ruby:    214275.0 i/s
        compare-ruby:    188818.6 i/s - 1.13x  slower

                          flatten2
          built-ruby:    159840.8 i/s
        compare-ruby:    159546.4 i/s - 1.00x  slower
```

where only the last one actually benchmarks an unflattened array properly.

methodmissing (Lourens Naudé) wrote:

> Attached is a patch for early return and preventing additional allocs for the empty array case (however not very sure how often that happens in practice):

There is no need to special case the empty array case. My patch would just treat it as a flat array, so would still avoid extra allocations and return early.

#10 - 09/27/2019 04:24 PM - dylants (Dylan Thacker-Smith)

- Status changed from Assigned to Closed

Applied in changeset git|a1fda16b238f24cf55814ecc18f716cbfff8dd91.

Optimize Array#flatten and flatten! for already flattened arrays (#2495)

- Optimize Array#flatten and flatten! for already flattened arrays
- Add benchmark for Array#flatten and Array#flatten!

[Bug #16119]

Files

0001-Optimize-Array-flatten-and-flatten-for-already-fl.diff.txt	2.79 KB	08/22/2019	dylants (Dylan Thacker-Smith)
0001-Let-empty-arrays-being-flattened-not-alloc-additiona.patch	791 Bytes	09/22/2019	methodmissing (Lourens Naudé)