Ruby - Feature #4917

`NilClass#to_ary`

06/22/2011 02:20 PM - y_feldblum (Jay Feldblum)

Status:	Rejected
Priority:	Normal
Assignee:	
Target version:	

Description

Kernel#Array, when passed nil, first tries to send to_ary, which ends up calling method_missing, and then tries to send to_a, which finally succeeds. When Kernel#Array is used frequently, for example in library/gem code, this can have a noticeable, if relatively small, negative impact on the overall application performance.

For performance improvement, nil should respond to to_ary. I propose:

```
NilClass.class_eval { alias to_ary to_a }
```

Using the following code,

```
require 'benchmark'
def bench(times) Benchmark.bmbm{|x| x.report{times.times(&Proc.new)}} end

# Sees how many times method_missing is called
class NilClass
  alias method_missing_without_hit method_missing
  def method_missing(name, *args, &block)
    $method_missing_hits += 1
    method_missing_without_hit(name, *args, &block)
  end
end
```

I measured the benchmark and method_missing calls.

```
NilClass.class_eval { undef to_ary }

$method_missing_hits = 0
bench(100_000) { Array(nil) }
$method_missing_hits # => 200005

NilClass.class_eval { alias to_ary to_a }

$method_missing_hits = 0
bench(100_000) { Array(nil) }
$method_missing_hits # => 0
```

It is observed that the former is very slow. The latter is instantaneous.

History

#1 - 06/23/2011 12:58 AM - shyouhei (Shyouhei Urabe)

- Tracker changed from Bug to Feature

#2 - 06/23/2011 02:55 AM - marcandre (Marc-Andre Lafortune)

- Status changed from Open to Rejected

The method to_ary is for classes that can be implicitly converted to an Array. This doesn't apply to NilClass.

I'm also highly sceptical as to the actual real life impact of such an optimization.

#3 - 06/23/2011 07:34 AM - headius (Charles Nutter)

Perhaps if an optimization is needed, it could just be adding a nil check to Kernel#Array.

#4 - 06/24/2011 01:23 AM - Eregon (Benoit Daloze)

On 22 June 2011 19:55, Marc-Andre Lafortune ruby-core@marc-andre.ca wrote:

> The method to_ary is for classes that can be implicitly converted to an Array. This doesn't apply to NilClass.
>
> I'm also highly sceptical as to the actual real life impact of such an optimization.

I agree with you, such optimization is not worth it.

But, for the sake of curiosity, I did my own benchmark.
The numbers are (optArray doing the nil check, for 1 000 000 calls):
Array(nil) 0.487444
optArray(nil) 0.234128

Array(Object.new) 0.462688
optArray(Object.new) 0.467910

It is 2 times faster for nil, and does not seem to impact the
performance for other objects.

But the real gain is 0.2s on 1 million calls, which should never
happen in real life (or then it would be insignificant compared to the
total time to run).

#5 - 06/24/2011 03:59 AM - naruse (Yui NARUSE)

(2011/06/24 1:00), Benoit Daloze wrote:

> On 22 June 2011 19:55, Marc-Andre Lafortune ruby-core@marc-andre.ca wrote:
>
> > The method to_ary is for classes that can be implicitly converted to an Array. This doesn't apply to NilClass.

nil is not an array; NilClass can't have to_ary.

> > I'm also highly sceptical as to the actual real life impact of such an optimization.
>
> I agree with you, such optimization is not worth it.
>
> But, for the sake of curiosity, I did my own benchmark.
> The numbers are (optArray doing the nil check, for 1 000 000 calls):
> Array(nil) 0.487444
> optArray(nil) 0.234128
>
> Array(Object.new) 0.462688
> optArray(Object.new) 0.467910
>
> It is 2 times faster for nil, and does not seem to impact the
> performance for other objects.
>
> But the real gain is 0.2s on 1 million calls, which should never
> happen in real life (or then it would be insignificant compared to the
> total time to run).

It needs a check whether NilClass#to_a is redefined or not.

--
NARUSE, Yui naruse@airemix.jp

#6 - 06/27/2011 01:42 AM - y_feldblum (Jay Feldblum)

This minor performance issue becomes a huge problem when NilClass#method_missing is defined, such as in ActiveSupport (https://github.com/rails/rails/blob/master/activesupport/lib/active_support/whiny_nil.rb), which is a popular choice when developing a Rails application.

Using the following line in IRB (1.9.2-p180) for measurement:

```
require 'ruby-prof'
@times = 1_000_000
@proc = proc { Array(nil) }
RubyProf::FlatPrinter.new(RubyProf.profile{@times.times(&@proc)}).print
```

If NilClass#method_missing is not defined, then performance is good:

```
 %self     total     self     wait    child    calls  name
 39.57      1.69     1.02     0.00     0.67  1000000  Kernel#Array
 34.59      2.59     0.89     0.00     1.69        1  Integer#times
 25.84      0.67     0.67     0.00     0.00  1000000  NilClass#to_a
  0.00      2.59     0.00     0.00     2.59        1  Object#irb_binding
```

If NilClass#method_missing is defined, such as with

```
class NilClass
  def method_missing(name, *args)
  end
end
```

then the time taken doubles:

```
 %self     total     self     wait    child    calls  name
 54.72      3.11     2.20     0.00     0.91  1000000  Kernel#Array
 22.63      4.02     0.91     0.00     3.11        1  Integer#times
 12.16      0.49     0.49     0.00     0.00  1000000  NilClass#to_a
 10.48      0.42     0.42     0.00     0.00  1000000  NilClass#method_missing
  0.00      4.02     0.00     0.00     4.02        1  Object#irb_binding
```

If NilClass#method_missing is defined, and just calls super:

```
class NilClass
  def method_missing(name, *args)
    super
  end
end
```

Then the time taken is 30x:

```
 %self     total     self     wait    child    calls  name
 81.17     46.09    38.52     0.00     7.57  1000000  Kernel#Array
  3.15      3.92     1.50     0.00     2.43  1000000  NoMethodError#initialize
  2.89     47.46     1.37     0.00    46.09        1  Integer#times
  2.65      1.26     1.26     0.00     0.00  1000000  Exception#initialize
  2.46      2.43     1.17     0.00     1.26  1000000  NameError#initialize
  2.33      1.10     1.10     0.00     0.00  1000000  Exception#set_backtrace
  1.19      0.57     0.57     0.00     0.00  1000000  Exception#backtrace
  1.10      0.52     0.52     0.00     0.00  1000000  <Class::BasicObject>#allocate
  1.09      0.52     0.52     0.00     0.00  1000000  NilClass#to_a
  1.00      0.48     0.48     0.00     0.00  1000000  Exception#exception
  0.97      0.46     0.46     0.00     0.00  1000000  Kernel#respond_to_missing?
  0.00     47.46     0.00     0.00    47.46        1  Object#irb_binding
```

Let alone if NilClass#method_missing is defined as in ActiveSupport to print the error.

Likewise the following progression:

```
require 'ruby-prof'
@times = 1_000_000
@proc = proc { a, b = nil }
RubyProf::FlatPrinter.new(RubyProf.profile{@times.times(&@proc)}).print
```

Without defining NilClass#method_missing:

```
 %self     total     self     wait    child    calls  name
100.00      0.80     0.80     0.00     0.00        1  Integer#times
  0.00      0.80     0.00     0.00     0.80        1  Object#irb_binding
```

With defining NilClass#method_missing to do nothing:

```
class NilClass
  def method_missing(name, *args)
  end
end
```

then the time taken triples:

```
 %self     total     self     wait    child    calls  name
 80.87      2.23     1.80     0.00     0.43        1  Integer#times
 19.13      0.43     0.43     0.00     0.00  1000000  NilClass#method_missing
```

```
  0.00      2.23     0.00     0.00     2.23        1  Object#irb_binding
```

With defining NilClass#method_missing to call super:

```
class NilClass
  def method_missing(name, *args)
    super
  end
end
```

then the time taken multiplies 50x:

```
 %self     total      self      wait     child    calls  name
 76.30     43.80    37.04      0.00      6.76  1000000  BasicObject#method_missing
  6.36     48.55     3.09      0.00     45.46        1  Integer#times
  3.03      3.94     1.47      0.00      2.47  1000000  NoMethodError#initialize
  2.64      1.28     1.28      0.00      0.00  1000000  Exception#initialize
  2.44      2.47     1.19      0.00      1.28  1000000  NameError#initialize
  2.35      1.14     1.14      0.00      0.00  1000000  Exception#set_backtrace
  2.32     44.92     1.13      0.00     43.80  1000000  NilClass#method_missing
  1.20      0.58     0.58      0.00      0.00  1000000  Exception#backtrace
  1.18      0.57     0.57      0.00      0.00  1000000  <Class::BasicObject>#allocate
  1.11      0.54     0.54      0.00      0.00  1000000  Kernel#respond_to_missing?
  1.06      0.52     0.52      0.00      0.00  1000000  Exception#exception
  0.00     48.55     0.00      0.00     48.55        1  Object#irb_binding
```

Let alone if NilClass#method_missing is defined as in ActiveSupport to print the error.

So all calls to Kernel#Array and all uses of multiple return values, when the argument is nil or the right-hand side is nil, can cause a large slowdown in Rails development mode.

I am seeing this in particular in Sass (https://github.com/nex3/sass/blob/3.1.1/lib/sass/importers/filesystem.rb#L130), where the call in that line gets called often and usually will return nil. This causes a very noticeable slowdown when developing, where NilClass#method_missing is defined to print the error.

A very quick-and-dirty solution to this problem is simply to not let Ruby try to call method_missing in these cases:

```
class NilClass
  alias to_ary to_a
end
```

This causes Ruby to call to_ary (which is an alias for to_a) rather than try to call to_ary, fallback to calling method_missing (which is very slow in Rails development), and subsequently call to_a.

When I use this hack, then performance of Array(nil) and a, b = nil returns to being very fast. That is why I showed it. I am using this hack to make Rails development faster, but it is a hack.

Ruby should check if the argument is nil and should return [], just as NilClass#to_a does, in the definition of Kernel#Array and in whatever code implements a, b = nil. Ruby should not try to call to_ary on nil because that relies, for its performance, on NilClass#method_missing being undefined.

#7 - 09/23/2020 12:43 AM - DanRathbun (Dan Rathbun)

y_feldblum (Jay Feldblum) wrote in #note-6:

> This minor performance issue becomes a huge problem when NilClass#method_missing is defined, such as in ActiveSupport (https://github.com/rails/rails/blob/master/activesupport/lib/active_support/whiny_nil.rb), which is a popular choice when developing a Rails application.

I'm curious, now that Ruby 2.x has refinements (and we're near 10 years on,) would there be acceptable speed improvements using refinements rather than direct-patching (ie, hacks) ?

#8 - 09/23/2020 11:51 AM - sawa (Tsuyoshi Sawada)

- Subject changed from NilClass#to_ary to `NilClass#to_ary`

- Description updated