Ruby master - Feature #17055

Allow suppressing uninitialized instance variable and method redefined verbose mode warnings

07/28/2020 10:03 PM - jeremyevans0 (Jeremy Evans)

Status:	Open
Priority:	Normal
Assignee:	
Target version:	

Description

These two verbose mode warnings are both fairly common and have good reasons why you would not want to warn about them in specific cases. Not initializing instance variables to nil can be much better for performance, and redefining methods without removing the method first is the only safe approach in multi-threaded code.

There are reasons that you may want to issue verbose warnings by default in these cases. For uninitialized instance variables, it helps catch typos. For method redefinition, it could alert you that a method already exists when you didn't expect it to, such as when a file is loaded multiple times when it should only be loaded once.

I propose we keep the default behavior the same, but offer the ability to opt-out of these warnings by defining methods. For uninitialized instance variables in verbose mode, I propose we call expected_uninitialized_instance_variable?(iv) on the object. If this method doesn't exist or returns false/nil, we issue the warning. If the method exists and returns true, we suppress the warning. Similarly, for redefined methods, we call expected_redefined_method?(method_name) on the class or module. If the method doesn't exist or returns false/nil, we issue the warning. If the method exists and returns true, we suppress the warning.

This approach allows high performance code (uninitialized instance variables) and safe code (redefining methods without removing) to work without verbose mode warnings.

I have implemented this support in a pull request: https://github.com/ruby/ruby/pull/3371

History

#1 - 07/29/2020 02:26 AM - shyouhei (Shyouhei Urabe)

Not against the feature itself, but JFIY you can suppress method redefinition warnings by something like:

```
def foo
end

a = 128.times.map do
  Thread.start do
    alias foo foo
    def foo
      Thread.current
    end
  end
end

a.map(&:join)

p foo
```

#2 - 07/29/2020 06:54 AM - kamipo (Ryuta Kamizono)

Rails also want a way to suppress method redefinition warnings.
For now it uses the alias hack.

https://github.com/rails/rails/blob/b2eb1d1c55a59fee1e6c4cba7030d8ceb524267c/activesupport/lib/active_support/core_ext/module/redefine_method.rb#L7-L13

```
  def silence_redefinition_of_method(method)
    if method_defined?(method) || private_method_defined?(method)
      # This suppresses the "method redefined" warning; the self-alias
      # looks odd, but means we don't need to generate a unique name
      alias_method method, method
    end
  end
```

In Perl, it have similar lexical scope pragma no warnings 'uninitialized' and no warnings 'redefine' for that.

#3 - 07/29/2020 09:16 AM - byroot (Jean Boussier)

I have no particular opinion on the instance variable part, except that it makes me think of this request of reporting on instance variable typos. It could be interesting to design the API in such a way that a DidYouMean integration would be possible.

With the currently proposed API it would look like?

```
def expected_uninitialized_instance_variable?(iv)
  DidYouMean.something(instance_variables, iv)
  false
end
```

Which doesn't seem too bad.

#4 - 07/29/2020 08:48 PM - Eregon (Benoit Daloze)

jeremyevans0 (Jeremy Evans) wrote:

> Not initializing instance variables to nil can be much better for performance

Why is that? Because just writing extra instance variables in initialize is much slower in MRI?
It's already an allocation path so it's not that fast anyway (unless escape analyzed, and then writing instance variables should be pretty much free).

It seems a very narrow use-case to me, and extremely MRI-specific.
In fact I wouldn't be surprised if on other Ruby implementations initializing led to better performance (e.g., no need to grow the ivar storage dynamically later, or change the shape/hidden class).

I would much prefer a Warning category for this.
Or probably your warning gem could be used to suppress those conveniently?

Calling more methods when doing warnings is adding more boilerplate, complexity and edge cases to these code paths.

#5 - 07/29/2020 08:55 PM - Eregon (Benoit Daloze)

Also callbacks seems a very odd way to handle this, if we really want methods to suppress warnings for specific methods/ivars, let's do it proactively like:

```
ignore_warning_method_redefinition :foo
def foo
  ...
end

ignore_warning_uninitialized_ivar :@foo
```

But again, I think there is no need for performance.
Also those warnings only happen in verbose mode, which typically shouldn't be used in production.

#6 - 07/29/2020 09:37 PM - jeremyevans0 (Jeremy Evans)

Eregon (Benoit Daloze) wrote in #note-4:

> jeremyevans0 (Jeremy Evans) wrote:
>
> > Not initializing instance variables to nil can be much better for performance
>
> Why is that? Because just writing extra instance variables in initialize is much slower in MRI?
> It's already an allocation path so it's not that fast anyway (unless escape analyzed, and then writing instance variables should be pretty much free).

Yes. When you initialize an instance variable to nil, it slows things down, and there is no benefit because the trying to access an uninitialized instance variable returns nil anyway (plus warning in verbose mode)

The difference is substantial. With 6 instance variables, it's over twice as fast to skip initializing them to nil.

```
require 'benchmark/ips'

# initialized
class A
  def initialize
    @c = @d = @e = @f = @g = @h = nil
  end
  def b
    @c || @d || @e || @f || @g || @h
```

```
  end
end

# not initialized
class B
  def initialize
    # nothing
  end
  def b
    @c || @d || @e || @f || @g || @h
  end
end

eval "def a0; #{"A.new;"*1000} end"
eval "def b0; #{"B.new;"*1000} end"

Benchmark.ips do |x|
  x.warmup = 0
  x.report('initialized'){a0}
  x.report('uninitialized'){b0}
  x.compare!
end
```

Results with Ruby 2.7.1:

```
         initialized    931.400  (_ 6.3%) i/s -      4.628k in   4.991632s
       uninitialized      2.016k (_10.3%) i/s -      9.923k in   4.987151s

Comparison:
       uninitialized:     2016.2 i/s
         initialized:      931.4 i/s - 2.16x  (_ 0.00) slower
```

> It seems a very narrow use-case to me, and extremely MRI-specific.

Results with JRuby 9.2.12 still show a significant speedup, though it is not as dramatic:

```
         initialized      5.865k (_ 9.3%) i/s -     26.736k
       uninitialized      8.712k (_ 4.1%) i/s -     41.098k

Comparison:
       uninitialized:     8712.2 i/s
         initialized:     5865.5 i/s - 1.49x slower
```

> In fact I wouldn't be surprised if on other Ruby implementations initializing led to better performance (e.g., no need to grow the ivar storage dynamically later, or change the shape/hidden class).

Do the above results on JRuby suprise you?

> I would much prefer a Warning category for this.

The issue with using a Warning category is that the change is made globally for all objects/instance variables and modules/methods. The advantage of my approach is that it allows for a granular approach, where each gem can suppress these verbose warnings as needed for their objects/modules, without turning off the advantages of these verbose warnings for the users of the gems (where the verbose warnings may be helpful in their own code).

> Or probably your warning gem could be used to suppress those conveniently?

The warning gem has always supported this. It was the primary reason I worked on adding the warning support to Ruby 2.4.

> Calling more methods when doing warnings is adding more boilerplate, complexity and edge cases to these code paths.

In the uninitialized instance variable case, I was actually able to reduce three separate code paths for issuing the warning to a single code path, plus I found a case that should produce a warning that did not and fixed that. So overall complexity could be lower for the uninitialized variable case, at least for MRI.

I considered the complexity cost of adding the feature before I proposed it, and I think the benefits of this feature outweigh the cost.

> Also callbacks seems a very odd way to handle this, if we really want methods to suppress warnings for specific methods/ivars, let's do it proactively like:

```
ignore_warning_method_redefinition :foo
def foo
  ...
end

ignore_warning_uninitialized_ivar :@foo
```

The proactive approach is substantially less flexible (e.g. can't use a regexp), and would require storing the values and a significantly more complex implementation. Considering you just complained about the complexity of my patch, I find it strange that you would propose an approach that would definitely require even greater internal complexity.

> Also those warnings only happen in verbose mode, which typically shouldn't be used in production.

The advantage of this approach is that it allows you to get the maximum possible performance in production, while suppressing unnecessary warnings in development or testing when you may run with verbose warnings. Without this, you either need to give up some production performance, or you have to require the user install a separate library to filter out the verbose warnings.

#7 - 07/30/2020 06:04 PM - headius (Charles Nutter)

Some JRuby perspective...

Personally, I have never been a big fan of the warning, but I don't have a strong opinion one way or another. I am not surprised that avoiding initialization is a faster on MRI because there's quite a few cycles spent for every instance variable assignment.

The JRuby numbers are a little misleading. The benchmark generates methods that contain 1000 new object creations, which goes well over the maximum size for JRuby to JIT compile that code. As a result, most of the overhead is still in our interpreter. Here's my numbers with the original benchmark, turning on invokedynamic to reduce the other overhead of the benchmark a bit:

```
[] ~/projects/jruby $ jruby -Xcompile.invokedynamic bench_ivar_init.rb
Warming up --------------------------------------
         initialized     1.499k i/100ms
       uninitialized     2.106k i/100ms
Calculating --------------------------------------
         initialized     15.914k (± 4.2%) i/s -     79.447k in   5.003069s
       uninitialized     20.717k (± 3.8%) i/s -    105.300k in   5.091607s

Comparison:
       uninitialized:    20716.5 i/s
         initialized:    15913.7 i/s - 1.30x  (± 0.00) slower


[] ~/projects/jruby $ rvm ruby-2.7.0 do ruby bench_ivar_init.rb
Warming up --------------------------------------
         initialized   342.000  i/100ms
       uninitialized   714.000  i/100ms
Calculating --------------------------------------
         initialized      3.410k (± 2.3%) i/s -     17.100k in   5.017475s
       uninitialized      6.967k (± 2.6%) i/s -     34.986k in   5.025282s

Comparison:
       uninitialized:     6966.7 i/s
         initialized:     3409.9 i/s - 2.04x  (± 0.00) slower
```

Modifying the script to actually JIT compile (10 allocations instead of 1000) shows the difference between initialized and uninitialized better. When JIT compiled, uninitialized variable accesses amount to two memory reads (variable slot, nil in memory) and a null check, and initializing to nil amounts to a memory move. The initialization can probably be eliminated if the allocation is eliminated, but it is harder to do otherwise.

```
[] ~/projects/jruby $ jruby -Xcompile.invokedynamic bench2.rb
Warming up --------------------------------------
         initialized     9.702k i/100ms
       uninitialized   566.506k i/100ms
Calculating --------------------------------------
         initialized      4.232M (±10.4%) i/s -     20.374M in   4.977150s
       uninitialized     17.564M (±33.4%) i/s -     66.848M in   5.016887s

Comparison:
       uninitialized: 17564016.6 i/s
         initialized:  4231794.4 i/s - 4.15x  (± 0.00) slower

[] ~/projects/jruby $ rvm ruby-2.7.0 do ruby bench2.rb
Warming up --------------------------------------
         initialized    33.617k i/100ms
       uninitialized    69.372k i/100ms
```

```
Calculating -------------------------------------
         initialized    332.282k (± 3.5%) i/s -      1.681M in   5.065437s
       uninitialized    687.154k (± 3.3%) i/s -      3.469M in   5.054528s

Comparison:
       uninitialized:   687154.4 i/s
         initialized:   332281.9 i/s - 2.07x  (± 0.00) slower
```

I do wonder if nil initialization could be optimized away by MRI, though. If we could detect that this was the first assignment of an instance variable in a new, untouched object, that assignment would be unnecessary. I know some JVMs also use read barriers to lazily initialized reference fields to null, avoiding the cost of zeroing that memory just to have it get overwritten moments later. There are options.

#8 - 07/30/2020 06:16 PM - headius (Charles Nutter)

Another note:

> In fact I wouldn't be surprised if on other Ruby implementations initializing led to better performance (e.g., no need to grow the ivar storage dynamically later, or change the shape/hidden class).

JRuby will attempt to allocate a "right-sized" object at the first allocation by analyzing all methods in a given class and looking for instance variable accesses. There's no growing of any variable table in this case; the object is allocated to have at least six real Java fields, and those fields are populated directly using movs in the resulting assembly code.

If our analysis is wrong, undetected variables will go into a separate dynamically-managed array, but this case is trivially easy.

#9 - 08/01/2020 02:36 PM - Eregon (Benoit Daloze)

jeremyevans0 (Jeremy Evans) wrote in #note-6:

> Yes. When you initialize an instance variable to nil, it slows things down, and there is no benefit because the trying to access an uninitialized instance variable returns nil anyway (plus warning in verbose mode)

And when you don't initialize, reads might become slower, because they have to check if warnings are enabled, and it might create polymorphism (see the end of my reply).
It's a two-sides blade.
I'd argue allocating without ever accessing or storing objects is an unrealistic benchmark.

Can you show a significant overhead on a realistic benchmark? (some Sequel real-world usage maybe?)

Here is the result on truffleruby 20.3.0-dev-b7a9b466 with your microbenchmark and 10 instead of 1000.
It shows the benchmark is bad mostly (i.e., it optimizes away and does nothing useful).

```
truffleruby bench_ivar_set.rb
Warming up --------------------------------------
         initialized   423.378M i/100ms
       uninitialized   191.211M i/100ms
         initialized   209.219M i/100ms
       uninitialized    60.855M i/100ms
Calculating -------------------------------------
         initialized      2.093B (± 0.2%) i/s -     10.670B in   5.098008s
       uninitialized      2.093B (± 0.4%) i/s -     10.467B in   5.000214s
         initialized      2.019B (±14.7%) i/s -      9.624B in   5.036843s
       uninitialized      2.122B (± 2.3%) i/s -     10.650B in   5.021645s

Comparison:
       uninitialized: 2122273181.8 i/s
         initialized: 2018820267.6 i/s - same-ish: difference falls within error
```

Here is the file used: bench_ivar_set.rb.

> The warning gem has always supported this. It was the primary reason I worked on adding the warning support to Ruby 2.4.

Isn't it good enough of a solution already? (can easily narrow by file & ivar name)
Adding a dependency on it doesn't seem a big deal and even deserved if you want such fine control over warnings and purposefully suppress warnings.
One can also prepend a module to Warning to handle this without an extra dependency.

> Calling more methods when doing warnings is adding more boilerplate, complexity and edge cases to these code paths.

> In the uninitialized instance variable case, I was actually able to reduce three separate code paths for issuing the warning to a single code path, plus I found a case that should produce a warning that did not and fixed that. So overall complexity could be lower for the uninitialized variable

> case, at least for MRI.

There might be other good things in the PR.
It's irrelevant to the general added complexity of a new callback.
Imagine if we wanted to add such methods for more warnings, I think that would quickly become a mess.

> The proactive approach is substantially less flexible (e.g. can't use a regexp), and would require storing the values and a significantly more complex implementation. Considering you just complained about the complexity of my patch, I find it strange that you would propose an approach that would definitely require even greater internal complexity.

I think well-designed and Ruby-like API is more important than complexity here.
But yes, both add complexity that IMHO is unneeded (I'm talking about conceptual complexity, not specifically of your patch).

> The advantage of this approach is that it allows you to get the maximum possible performance in production, while suppressing unnecessary warnings in development or testing when you may run with verbose warnings. Without this, you either need to give up some production performance, or you have to require the user install a separate library to filter out the verbose warnings.

My view is it encourages less readable code by trying to squeak a tiny bit of performance when running on current CRuby.

And it can make reads of uninitialized variables slower, if they later become initialized (simply because the inline cache needs to care about 2 cases instead of 1).
Here is a benchmark attempting to illustrate that, it's ~3x slower for uninit+init reads on TruffleRuby due to the created polymorphism, and ~1.16x slower for uninitialized reads on CRuby 2.6.6:
https://gist.github.com/eregon/561c09e0156a5530f5a100d3e2351c4b

```
ruby 2.6.6p146 (2020-03-31 revision 67876) [x86_64-linux]
  init + uninit read: 14989101.7 i/s
    initialized read: 14921107.5 i/s - same-ish: difference falls within error
  uninitialized read: 12885730.1 i/s - 1.16x  (± 0.00) slower

truffleruby 20.3.0-dev-b7a9b466, like ruby 2.6.6, GraalVM CE Native [x86_64-linux]
Comparison:
  uninitialized read: 704396153.8 i/s
    initialized read: 700673745.0 i/s - same-ish: difference falls within error
  init + uninit read: 214761238.7 i/s - 3.28x  (± 0.00) slower
```

#10 - 08/01/2020 02:54 PM - Eregon (Benoit Daloze)

In other words, lazily initializing @ivars causes reads to need some branching because they need to handle both the initialized and uninitialized cases.
So @ivars reads can no longer be straight-line code, which can impact performance as shown above.

I think it's also often less readable. If variables are initialized in initialize it's clear which state is kept in that class.
Also if the natural default is not nil but say 0 then @foo = 0 in initialize is useful information (notably it gives the type), and it can be a simple attr_reader :foo to read it instead of the convoluted:

```
def foo
  @foo || 0
end
```

#11 - 08/02/2020 03:38 PM - jeremyevans0 (Jeremy Evans)

Eregon (Benoit Daloze) wrote in #note-9:

> jeremyevans0 (Jeremy Evans) wrote in #note-6:
>
> > Yes. When you initialize an instance variable to nil, it slows things down, and there is no benefit because the trying to access an uninitialized instance variable returns nil anyway (plus warning in verbose mode)
>
> And when you don't initialize, reads might become slower, because they have to check if warnings are enabled, and it might create polymorphism (see the end of my reply).
> It's a two-sides blade.

This is correct. Whether or not to initialize instance variables depends on the object. For long lived objects (classes, constants), it definitely makes sense. For ephemeral objects, it can hurt performance.

> I'd argue allocating without ever accessing or storing objects is an unrealistic benchmark.
>
> Can you show a significant overhead on a realistic benchmark? (some Sequel real-world usage maybe?)

The last time I did testing on this in Sequel, the performance decrease from initializing instance variables to nil was around 5% for Sequel::Model instance creation depending on the plugins in use. One of the reasons it was around 5% was that many plugins had to override initialization methods just to set an instance variable and call super. 5% may not sound like a lot, but I can't justify a 5% performance decrease (or even a 1% performance decrease) just to avoid verbose warnings.

> Here is the result on truffleruby 20.3.0-dev-b7a9b466 with your microbenchmark and 10 instead of 1000.
> It shows the benchmark is bad mostly (i.e., it optimizes away and does nothing useful).

```
truffleruby bench_ivar_set.rb
Warming up --------------------------------------
         initialized   423.378M i/100ms
       uninitialized   191.211M i/100ms
         initialized   209.219M i/100ms
       uninitialized    60.855M i/100ms
Calculating -------------------------------------
         initialized      2.093B (± 0.2%) i/s -     10.670B in   5.098008s
       uninitialized      2.093B (± 0.4%) i/s -     10.467B in   5.000214s
         initialized      2.019B (±14.7%) i/s -      9.624B in   5.036843s
       uninitialized      2.122B (± 2.3%) i/s -     10.650B in   5.021645s

Comparison:
       uninitialized: 2122273181.8 i/s
         initialized: 2018820267.6 i/s - same-ish: difference falls within error
```

> Here is the file used: bench_ivar_set.rb.

Interesting. If it is not too much trouble, what are the results with 1000 instead of 10, if I may ask? Alternatively, if you modify the benchmark to access each instance variable once, what the the results of that? I'd test myself, but it appears TruffleRuby is not yet ported to the operating systems I use (OpenBSD/Windows).

> > The warning gem has always supported this. It was the primary reason I worked on adding the warning support to Ruby 2.4.
>
> Isn't it good enough of a solution already? (can easily narrow by file & ivar name)
> Adding a dependency on it doesn't seem a big deal and even deserved if you want such fine control over warnings and purposefully suppress warnings.
> One can also prepend a module to Warning to handle this without an extra dependency.

In my opinion, a ruby gem should never modify the behavior (e.g. add/override methods) of core classes, unless that is the purpose of the library. As such, modifying the Warning class is not something I would consider doing by default, and therefore people that use Sequel and want to run tests/development in verbose mode have to filter the warnings themselves. With the feature I am proposing, each library has control over their own code.

I believe verbose warning for uninitialized instance variables in code you have no control over is actively harmful to the user, because it can make it more difficult to find warnings in code you do have control over.

> > Calling more methods when doing warnings is adding more boilerplate, complexity and edge cases to these code paths.
>
> In the uninitialized instance variable case, I was actually able to reduce three separate code paths for issuing the warning to a single code path, plus I found a case that should produce a warning that did not and fixed that. So overall complexity could be lower for the uninitialized variable case, at least for MRI.
>
> There might be other good things in the PR.
> It's irrelevant to the general added complexity of a new callback.
> Imagine if we wanted to add such methods for more warnings, I think that would quickly become a mess.

Certainly this approach is not appropriate for all warnings. However, in my 15+ years of Ruby experience, I've seen that these two verbose warnings are by far the most common, and both of them have valid reasons for using the behavior that produces the verbose warnings (performance and safety).

> > The proactive approach is substantially less flexible (e.g. can't use a regexp), and would require storing the values and a significantly more complex implementation. Considering you just complained about the complexity of my patch, I find it strange that you would propose an approach that would definitely require even greater internal complexity.
>
> I think well-designed and Ruby-like API is more important than complexity here.

I fail to see how this is not a Ruby-like API. It's similar to other callbacks, such as method_added. It's also the simplest thing I can think of that would

work. Additionally, as byroot (Jean Boussier) mentioned, it may be possible to use this approach with did_you_mean for even more helpful warnings.

> The advantage of this approach is that it allows you to get the maximum possible performance in production, while suppressing unnecessary warnings in development or testing when you may run with verbose warnings. Without this, you either need to give up some production performance, or you have to require the user install a separate library to filter out the verbose warnings.

> My view is it encourages less readable code by trying to squeak a tiny bit of performance when running on current CRuby.

This is only half about performance. You haven't mentioned anything about the method redefinition warning yet. Can you provide your thoughts on that?

> In other words, lazily initializing @ivars causes reads to need some branching because they need to handle both the initialized and uninitialized cases.
> So @ivars reads can no longer be straight-line code, which can impact performance as shown above.

As I stated above, whether you want to initialize instance variables lazily or eagerly for performance depends on the object in question. Not all situations are the same. It is faster in some situations and slower in others.

> Also if the natural default is not nil but say 0 then @foo = 0 in initialize is useful information (notably it gives the type), and it can be a simple attr_reader :foo to read it instead of the convoluted:

```
def foo
  @foo || 0
end
```

I think we can all agree that there are cases where eagerly initializing the object can improve performance. This says nothing about the cases where eagerly initializing the object hurts performance.

#12 - 08/03/2020 11:22 AM - Eregon (Benoit Daloze)

jeremyevans0 (Jeremy Evans) wrote in #note-11:

> The last time I did testing on this in Sequel, the performance decrease from initializing instance variables to nil was around 5% for Sequel::Model instance creation depending on the plugins in use. One of the reasons it was around 5% was that many plugins had to override initialization methods just to set an instance variable and call super. 5% may not sound like a lot, but I can't justify a 5% performance decrease (or even a 1% performance decrease) just to avoid verbose warnings.

Could you reproduce that again?
I would like a realistic benchmark, so unless users frequently create Sequel::Model instances themselves, I guess the normal case is the data comes from somewhere (the database, some file, etc).
In such a case I would think the overhead is not measurable.

> Interesting. If it is not too much trouble, what are the results with 1000 instead of 10, if I may ask?

Duplicating code like that is IMHO a misleading way to benchmark (it makes it all too easy to misinterpret results) and obviously unrepresentative of real code.
I see it used for implementations like CRuby for which block call is a large overhead, but IMHO that should be a reminder there is a point where it's too small to measure on its own, and there is always code around for realistic cases.
Also the Ruby implementation might see that the result is unused due to this repetition (TruffleRuby did above).
I used 10 instead of 1000 since Charles did the same. But it should be just 1, really, and the benchmark should use the result value.

Anyway, this is the result for 1000, not everything is inlined since the method is so large, yet there seems to be no meaningful difference:

```
Calculating -------------------------------------
         initialized      4.257k (±41.1%) i/s -     14.136k in   5.064145s
       uninitialized      6.307k (±62.3%) i/s -     17.864k in   5.002209s
         initialized     12.782k (±23.8%) i/s -     53.010k in   5.013438s
       uninitialized     13.123k (±25.5%) i/s -     53.012k in   5.002126s

Comparison:
       uninitialized:    13123.0 i/s
         initialized:    12782.2 i/s - same-ish: difference falls within error
```

> Alternatively, if you modify the benchmark to access each instance variable once, what the the results of that?

It still optimizes away, because the allocation is never needed:
https://gist.github.com/eregon/f279901e3df450d7a1970b76b9653c71

```
Calculating -------------------------------------
         initialized      1.614B (±43.2%) i/s -      6.184B in   5.236291s
       uninitialized      1.816B (±33.1%) i/s -      7.259B in   5.078576s
         initialized      1.879B (±28.2%) i/s -      7.890B in   5.043604s
       uninitialized      1.811B (±32.4%) i/s -      7.259B in   5.007219s

Comparison:
         initialized: 1878613372.9 i/s
       uninitialized: 1811213500.0 i/s - same-ish: difference falls within error
```

We need a realistic benchmark, then we'll see if there is an observable difference or not and how much it matters in practice.

> I'd test myself, but it appears TruffleRuby is not yet ported to the operating systems I use (OpenBSD/Windows).

Feel free to open an issue about building TruffleRuby on OpenBSD. Maybe it's not so hard (but would need to build most things from source).
Alternatively I guess a more convenient way is to use Docker (but it might affect performance).

> In my opinion, a ruby gem should never modify the behavior (e.g. add/override methods) of core classes, unless that is the purpose of the library. As such, modifying the Warning class is not something I would consider doing by default, and therefore people that use Sequel and want to run tests/development in verbose mode have to filter the warnings themselves. With the feature I am proposing, each library has control over their own code.

My view of that is Warning.warn is designed explicitly for this, so that libraries can compose and filters warnings as needed.
Even better if the warning can be avoided in the first place though IMHO.

So, for this case, how about using def foo; @foo ||= nil; end? That would avoid the warning and still let you assign the @ivar lazily, no?

I have another concern: hiding warnings like this will silently cause much worse performance in $VERBOSE mode.
Admittedly this would only happen when $VERBOSE is true, but performance might still matter to some degree in such cases, e.g., when running tests (tests frameworks often enable $VERBOSE, and people still want their tests to run reasonably fast).

In such a case, on you branch it's 6.6 times slower for reads:
https://gist.github.com/eregon/32f4119b7796ec7a6243c68990949597

```
Comparison:
         initialized: 14754531.3 i/s
       uninitialized:  2227414.1 i/s - 6.62x  (± 0.00) slower
```

> This is only half about performance. You haven't mentioned anything about the method redefinition warning yet. Can you provide your thoughts on that?

Regarding method redefinition I think the alias/alias_method trick is reasonable, and has the advantage to already work on all Ruby versions.
It's not obvious though so I wonder if we could have something clearer like suppress_redefinition_warning instance_method(:my_method), but anyway method redefinition without caring of the previous definition should be very rare so alias seems a fine enough workaround.

I think in general it would be useful to have a thread-safe way to suppress warnings for a block of code, and that could be used in this case and many other cases (e.g., 194 suppress_warning in ruby/ruby).
Unfortunately, changing $VERBOSE is not thread-safe if there are multiple Threads.
That IMHO would be a valuable addition, and be useful not just for this method-redefinition-ignoring-previous case but in many other cases too.
(it wouldn't be a good solution for the uninitialized ivar case, there I think it's really best to avoid the warning in the first place)

#13 - 08/03/2020 04:30 PM - jeremyevans0 (Jeremy Evans)

- File t.rb added

Eregon (Benoit Daloze) wrote in #note-12:

> jeremyevans0 (Jeremy Evans) wrote in #note-11:
>
> > The last time I did testing on this in Sequel, the performance decrease from initializing instance variables to nil was around 5% for Sequel::Model instance creation depending on the plugins in use. One of the reasons it was around 5% was that many plugins had to override initialization methods just to set an instance variable and call super. 5% may not sound like a lot, but I can't justify a 5% performance decrease (or even a 1% performance decrease) just to avoid verbose warnings.
>
> Could you reproduce that again?
> I would like a realistic benchmark, so unless users frequently create Sequel::Model instances themselves, I guess the normal case is the data comes from somewhere (the database, some file, etc).
> In such a case I would think the overhead is not measurable.

I'm glad you asked, as the difference is actually much greater than I remember it being. With the attached benchmark, on PostgreSQL localhost, the

difference is over 100% when not using plugins:

```
$ NO_SEQUEL_PG=1 DATABASE_URL=postgres:///?user=sequel_test ruby t.rb regular noplugin
Warming up --------------------------------------
  Retrieve 1000 rows    31.000  i/100ms
Calculating -------------------------------------
  Retrieve 1000 rows    310.225  (_ 2.6%) i/s -      1.550k in   5.000031s

$ NO_SEQUEL_PG=1 DATABASE_URL=postgres:///?user=sequel_test ruby t.rb eager_initialize noplugin
Warming up --------------------------------------
  Retrieve 1000 rows    15.000  i/100ms
Calculating -------------------------------------
  Retrieve 1000 rows    151.327  (_ 2.0%) i/s -    765.000  in   5.057570s
```

and over 150% when using plugins:

```
$ NO_SEQUEL_PG=1 DATABASE_URL=postgres:///?user=sequel_test ruby t.rb regular plugin
1000
Warming up --------------------------------------
  Retrieve 1000 rows    23.000  i/100ms
Calculating -------------------------------------
  Retrieve 1000 rows    233.096  (_ 2.6%) i/s -      1.173k in   5.036040s

$ NO_SEQUEL_PG=1 DATABASE_URL=postgres:///?user=sequel_test ruby t.rb eager_initialize plugin
Warming up --------------------------------------
  Retrieve 1000 rows     8.000  i/100ms
Calculating -------------------------------------
  Retrieve 1000 rows     90.653  (_ 1.1%) i/s -    456.000  in   5.030471s
```

With an SQLite memory database and no plugins, the difference is over 35%:

```
$ ruby t.rb regular noplugin
Warming up --------------------------------------
  Retrieve 1000 rows     8.000  i/100ms
Calculating -------------------------------------
  Retrieve 1000 rows     84.154  (_ 1.2%) i/s -    424.000  in   5.039503s

$ ruby t.rb eager_initialize noplugin
Warming up --------------------------------------
  Retrieve 1000 rows     6.000  i/100ms
Calculating -------------------------------------
  Retrieve 1000 rows     61.862  (_ 1.6%) i/s -    312.000  in   5.044981s
```

and with plugins added, the difference is over 50%:

```
$ ruby t.rb regular plugin
Warming up --------------------------------------
  Retrieve 1000 rows     7.000  i/100ms
Calculating -------------------------------------
  Retrieve 1000 rows     73.384  (_ 2.7%) i/s -    371.000  in   5.059455s

$ ruby t.rb eager_initialize plugin
Warming up --------------------------------------
  Retrieve 1000 rows     4.000  i/100ms
Calculating -------------------------------------
  Retrieve 1000 rows     47.122  (_ 0.0%) i/s -    236.000  in   5.008490s
```

As you'll see by the benchmark, the reason the performance difference is much different than you would expect is that Model loads from the database in Sequel run through Sequel::Model.call (class method, not instance method), and all of the plugins that use instance variables must override the method to set the instance variables. Because it is a class method and not an instance method, instance_variable_set must be used. The overhead of all of those additional method calls (super and instance_variable_set) is what causes the dramatic difference in performance.

Feel free to play around with the above benchmark, but I hope it shows you that I'm asking for this feature for a good reason.

> So, for this case, how about using def foo; @foo ||= nil; end? That would avoid the warning and still let you assign the @ivar lazily, no?

Then you need a method call instead of an ivar access, which is also slower. It also would require more changes across the library.

> I have another concern: hiding warnings like this will silently cause much worse performance in $VERBOSE mode.
> Admittedly this would only happen when $VERBOSE is true, but performance might still matter to some degree in such cases, e.g., when running tests (tests frameworks often enable $VERBOSE, and people still want their tests to run reasonably fast).
>
> In such a case, on you branch it's 6.6 times slower for reads:
> https://gist.github.com/eregon/32f4119b7796ec7a6243c68990949597

```
Comparison:
         initialized: 14754531.3 i/s
       uninitialized:  2227414.1 i/s - 6.62x  (± 0.00) slower
```

I think most Ruby users would trade faster production performance for slower tests. Obviously, for those that wouldn't make that trade, they can initialize all instance variables and would not need to use this.

> I think in general it would be useful to have a thread-safe way to suppress warnings for a block of code, and that could be used in this case and many other cases (e.g., 194 suppress_warning in ruby/ruby).
> Unfortunately, changing $VERBOSE is not thread-safe if there are multiple Threads.
> That IMHO would be a valuable addition, and be useful not just for this method-redefinition-ignoring-previous case but in many other cases too.
> (it wouldn't be a good solution for the uninitialized ivar case, there I think it's really best to avoid the warning in the first place)

Possibly useful, but unrelated to the current proposal. Please post a new feature request if you would like that.

#14 - 08/03/2020 04:48 PM - tenderlovemaking (Aaron Patterson)

jeremyevans0 (Jeremy Evans) wrote in #note-13:

> As you'll see by the benchmark, the reason the performance difference is much different than you would expect is that Model loads from the database in Sequel run through Sequel::Model.call (class method, not instance method), and all of the plugins that use instance variables must override the method to set the instance variables. Because it is a class method and not an instance method, instance_variable_set must be used. The overhead of all of those additional method calls (super and instance_variable_set) is what causes the dramatic difference in performance.

Could the design be improved such that instance_variable_set isn't required? Using instance_variable_set means that inline caches will not be used (in MRI anyway), so I'm not sure it should be used in code that requires high performance. I suppose we could also work on improving the performance of instance_variable_set, but I'm not sure usage is that common.

#15 - 08/03/2020 05:01 PM - jeremyevans0 (Jeremy Evans)

tenderlovemaking (Aaron Patterson) wrote in #note-14:

> jeremyevans0 (Jeremy Evans) wrote in #note-13:
>
> > As you'll see by the benchmark, the reason the performance difference is much different than you would expect is that Model loads from the database in Sequel run through Sequel::Model.call (class method, not instance method), and all of the plugins that use instance variables must override the method to set the instance variables. Because it is a class method and not an instance method, instance_variable_set must be used. The overhead of all of those additional method calls (super and instance_variable_set) is what causes the dramatic difference in performance.
>
> Could the design be improved such that instance_variable_set isn't required? Using instance_variable_set means that inline caches will not be used (in MRI anyway), so I'm not sure it should be used in code that requires high performance. I suppose we could also work on improving the performance of instance_variable_set, but I'm not sure usage is that common.

It certainly could, but it would require more work, and would result in slower performance in the the no-plugin case. We could add a private instance method can call that, and the private instance method could initialize all the instance variables to nil the usual way. That would make things somewhat faster. You still have all the super calls to slow things down, though. If you want me to work on a benchmark for that, please let me know, but it's a sure bet that even that approach would result in a slowdown significant enough that I wouldn't want to switch to it.

#16 - 08/03/2020 06:31 PM - tenderlovemaking (Aaron Patterson)

jeremyevans0 (Jeremy Evans) wrote in #note-15:

> tenderlovemaking (Aaron Patterson) wrote in #note-14:
>
> > jeremyevans0 (Jeremy Evans) wrote in #note-13:
> >
> > > As you'll see by the benchmark, the reason the performance difference is much different than you would expect is that Model loads from the database in Sequel run through Sequel::Model.call (class method, not instance method), and all of the plugins that use instance variables must override the method to set the instance variables. Because it is a class method and not an instance method, instance_variable_set must be used. The overhead of all of those additional method calls (super and instance_variable_set) is what causes the dramatic difference in performance.
> >
> > Could the design be improved such that instance_variable_set isn't required? Using instance_variable_set means that inline caches will not be used (in MRI anyway), so I'm not sure it should be used in code that requires high performance. I suppose we could also work on improving the performance of instance_variable_set, but I'm not sure usage is that common.

> It certainly could, but it would require more work, and would result in slower performance in the the no-plugin case.

I guess I need to read the Sequel implementation. It seems possible to design a system that is no overhead in the no-plugin case that also uses regular instance variables. In fact it seems like defining one "ClassMethods" module and many "InstanceMethods" modules would do the trick (with no changes to Sequel).

> We could add a private instance method can call that, and the private instance method could initialize all the instance variables to nil the usual way. That would make things somewhat faster. You still have all the super calls to slow things down, though. If you want me to work on a benchmark for that, please let me know, but it's a sure bet that even that approach would result in a slowdown significant enough that I wouldn't want to switch to it.

I think we just need to measure the difference between instance_variable_set and regular instance variable setting. That should give us an idea of the potential speed increase by switching to regular instance variables.

```
require "benchmark/ips"

class Embedded
  def initialize
    @a = @b = @c = nil
  end
end

class EmbeddedIvarSet
  def initialize
    instance_variable_set :@a, nil
    instance_variable_set :@b, nil
    instance_variable_set :@c, nil
  end
end

class NotEmbedded
  def initialize
    @a = @b = @c = @d = @e = @f = nil
  end
end

class NotEmbeddedIvarSet
  def initialize
    instance_variable_set :@a, nil
    instance_variable_set :@b, nil
    instance_variable_set :@c, nil
    instance_variable_set :@d, nil
    instance_variable_set :@e, nil
    instance_variable_set :@f, nil
  end
end

eval "def embedded; #{"Embedded.new;"*1000} end"
eval "def embedded_ivar_set; #{"EmbeddedIvarSet.new;"*1000} end"
eval "def not_embedded; #{"NotEmbedded.new;"*1000} end"
eval "def not_embedded_ivar_set; #{"NotEmbeddedIvarSet.new;"*1000} end"

Benchmark.ips do |x|
  x.report("embedded") { embedded }
  x.report("embedded ivar set") { embedded_ivar_set }
  x.report("not embedded") { not_embedded }
  x.report("not embedded ivar set") { not_embedded_ivar_set }
end
```

On my machine:

```
aaron@whiteclaw ~> ruby -v ivar_speed.rb
ruby 2.7.1p83 (2020-03-31 revision a0c7c23c9c) [x86_64-linux]
Warming up --------------------------------------
            embedded   792.000  i/100ms
   embedded ivar set   516.000  i/100ms
        not embedded   545.000  i/100ms
not embedded ivar set
                       312.000  i/100ms
Calculating -------------------------------------
            embedded      7.945k (± 0.2%) i/s -     40.392k in   5.084108s
   embedded ivar set      5.108k (± 0.2%) i/s -     25.800k in   5.051157s
        not embedded      5.310k (± 0.5%) i/s -     26.705k in   5.029699s
```

```
not embedded ivar set
                          3.124k (± 0.4%) i/s -     15.912k in   5.094197s
```

It looks like instance_variable_set requires a significant tax compared to just instance variable sets. But I didn't know if that's the bottleneck, so I rewrote the benchmark you provided to use regular instance variables here.

Surprisingly it was consistently slower! Not by much, but it was consistent:

```
aaron@whiteclaw ~/thing (master)> ruby -v t.rb eager_initialize plugin
ruby 2.7.1p83 (2020-03-31 revision a0c7c23c9c) [x86_64-linux]
/home/aaron/.gem/ruby/2.7.1/gems/sequel-5.35.0/lib/sequel/plugins/json_serializer.rb:132: warning: instance va
riable @json_serializer_opts not initialized
t.rb:28: warning: method redefined; discarding old call
/home/aaron/.gem/ruby/2.7.1/gems/sequel-5.35.0/lib/sequel/model/base.rb:221: warning: previous definition of c
all was here
/home/aaron/.gem/ruby/2.7.1/gems/sequel-5.35.0/lib/sequel/model/base.rb:918: warning: instance variable @overr
idable_methods_module not initialized
/home/aaron/.gem/ruby/2.7.1/gems/sequel-5.35.0/lib/sequel/adapters/shared/sqlite.rb:287: warning: instance var
iable @transaction_mode not initialized
/home/aaron/.gem/ruby/2.7.1/gems/sequel-5.35.0/lib/sequel/adapters/shared/sqlite.rb:287: warning: instance var
iable @transaction_mode not initialized
Warming up --------------------------------------
  Retrieve 1000 rows    27.000  i/100ms
Calculating --------------------------------------
  Retrieve 1000 rows    278.107  (± 0.4%) i/s -      1.404k in   5.048542s
aaron@whiteclaw ~/thing (master)> git checkout -
Switched to branch 'regular-ivars'
aaron@whiteclaw ~/thing (regular-ivars)> ruby -v t.rb eager_initialize plugin
ruby 2.7.1p83 (2020-03-31 revision a0c7c23c9c) [x86_64-linux]
/home/aaron/.gem/ruby/2.7.1/gems/sequel-5.35.0/lib/sequel/plugins/json_serializer.rb:132: warning: instance va
riable @json_serializer_opts not initialized
t.rb:42: warning: method redefined; discarding old call
/home/aaron/.gem/ruby/2.7.1/gems/sequel-5.35.0/lib/sequel/model/base.rb:221: warning: previous definition of c
all was here
/home/aaron/.gem/ruby/2.7.1/gems/sequel-5.35.0/lib/sequel/model/base.rb:918: warning: instance variable @overr
idable_methods_module not initialized
/home/aaron/.gem/ruby/2.7.1/gems/sequel-5.35.0/lib/sequel/adapters/shared/sqlite.rb:287: warning: instance var
iable @transaction_mode not initialized
/home/aaron/.gem/ruby/2.7.1/gems/sequel-5.35.0/lib/sequel/adapters/shared/sqlite.rb:287: warning: instance var
iable @transaction_mode not initialized
Warming up --------------------------------------
  Retrieve 1000 rows    23.000  i/100ms
Calculating --------------------------------------
  Retrieve 1000 rows    239.608  (± 0.4%) i/s -      1.219k in   5.087511s
```

These benchmarks were close enough that it made me wonder if setting instance variables was even the bottleneck of the program, so I deleted all instance variables from the program but left the method calls here.

```
aaron@whiteclaw ~/thing (methods-but-no-ivars)> ruby -v t.rb eager_initialize plugin
ruby 2.7.1p83 (2020-03-31 revision a0c7c23c9c) [x86_64-linux]
/home/aaron/.gem/ruby/2.7.1/gems/sequel-5.35.0/lib/sequel/plugins/json_serializer.rb:132: warning: instance va
riable @json_serializer_opts not initialized
t.rb:33: warning: method redefined; discarding old call
/home/aaron/.gem/ruby/2.7.1/gems/sequel-5.35.0/lib/sequel/model/base.rb:221: warning: previous definition of c
all was here
/home/aaron/.gem/ruby/2.7.1/gems/sequel-5.35.0/lib/sequel/model/base.rb:918: warning: instance variable @overr
idable_methods_module not initialized
/home/aaron/.gem/ruby/2.7.1/gems/sequel-5.35.0/lib/sequel/adapters/shared/sqlite.rb:287: warning: instance var
iable @transaction_mode not initialized
/home/aaron/.gem/ruby/2.7.1/gems/sequel-5.35.0/lib/sequel/adapters/shared/sqlite.rb:287: warning: instance var
iable @transaction_mode not initialized
Warming up --------------------------------------
  Retrieve 1000 rows    26.000  i/100ms
Calculating --------------------------------------
  Retrieve 1000 rows    267.602  (± 0.4%) i/s -      1.352k in   5.052309s
```

This is still close to the same speed as the previous benchmark. It seems like the cost of calling a method dwarfs the cost of setting an instance variable.

btw, super didn't use an inline method cache in < 2.8, so the current development branch is much faster for the above case:

```
aaron@whiteclaw ~/thing (methods-but-no-ivars)> ruby -v t.rb eager_initialize plugin
ruby 2.8.0dev (2020-07-31T12:07:19Z master f80020bc50) [x86_64-linux]
/home/aaron/.gem/ruby/2.8.0/gems/activesupport-6.0.3.2/lib/active_support/core_ext/hash/except.rb:12: warning:
 method redefined; discarding old except
```

```
/home/aaron/.gem/ruby/2.8.0/gems/sequel-5.35.0/lib/sequel/plugins/json_serializer.rb:132: warning: instance va
riable @json_serializer_opts not initialized
t.rb:33: warning: method redefined; discarding old call
/home/aaron/.gem/ruby/2.8.0/gems/sequel-5.35.0/lib/sequel/model/base.rb:221: warning: previous definition of c
all was here
/home/aaron/.gem/ruby/2.8.0/gems/sequel-5.35.0/lib/sequel/model/base.rb:918: warning: instance variable @overr
idable_methods_module not initialized
/home/aaron/.gem/ruby/2.8.0/gems/sequel-5.35.0/lib/sequel/adapters/shared/sqlite.rb:287: warning: instance var
iable @transaction_mode not initialized
/home/aaron/.gem/ruby/2.8.0/gems/sequel-5.35.0/lib/sequel/adapters/shared/sqlite.rb:287: warning: instance var
iable @transaction_mode not initialized
Warming up --------------------------------------
  Retrieve 1000 rows    31.000  i/100ms
Calculating --------------------------------------
  Retrieve 1000 rows    317.013  (± 0.0%) i/s -      1.612k in   5.084974s
```

That said, it's not as fast doing no method calls all together.

I'm not sure if I did the benchmarks 100% correctly, so I would appreciate a review. The "regular ivars is slower" result makes me think I did something wrong.

#17 - 08/04/2020 02:59 AM - jeremyevans0 (Jeremy Evans)

tenderlovemaking (Aaron Patterson) wrote in #note-16:

> I guess I need to read the Sequel implementation. It seems possible to design a system that is no overhead in the no-plugin case that also uses regular instance variables. In fact it seems like defining one "ClassMethods" module and many "InstanceMethods" modules would do the trick (with no changes to Sequel).

We would have to change the default call method:

```
      # class method
      def call(values)
        o = allocate
        o.instance_variable_set(:@values, values)
        o
      end
```

to something like

```
      # class method
      def call(values)
        o = allocate
        o.send(:initialize_from_database, values)
        o
      end

      # instance method
      def initialize_from_database(values)
        @values = values
      end
```

I believe such an approach will always be slower, as send(:method_name) and setting the instance variable inside the method will slower than a call to instance_variable_set.

> These benchmarks were close enough that it made me wonder if setting instance variables was even the bottleneck of the program, so I deleted all instance variables from the program but left the method calls here.

I agree here, the method calling overhead is more than the ivar setting overhead.

> btw, super didn't use an inline method cache in < 2.8, so the current development branch is much faster for the above case:

That's great to hear. My libraries tend to rely heavily on module inclusion, and defining methods that call super, so that should provide a nice performance improvement.

> I'm not sure if I did the benchmarks 100% correctly, so I would appreciate a review. The "regular ivars is slower" result makes me think I did something wrong.

I looked at your benchmark for using instance variables directly in instance methods and it looks fine to me. If there is a bug with it, I couldn't spot it (other than init_values is public and should be private, but using send would probably decrease performance). I'm also surprised it is slower than the instance_variable_set approach. However, the two code paths are different internally, and you are running in verbose mode. I'm not sure whether

you get different results if you are not in verbose mode.

#18 - 08/15/2020 10:54 AM - Eregon (Benoit Daloze)

I ran the measurements on both CRuby master and 2.6.6, with sqlite3 for convenience.
I see smaller differences, but also my results are about 6 times faster.
It's still a larger difference than I expect so I'll try to dig deeper.

Which version did you run with? Are you sure it's a build with default optimizations?

Ruby master

```
$ ruby -v
ruby 2.8.0dev (2020-08-15T05:17:02Z master d75433ae19) [x86_64-linux]
$ gem i sequel benchmark-ips sqlite3

$ ruby bench_sequel_ivar.rb regular noplugin
Warming up --------------------------------------
  Retrieve 1000 rows    52.000  i/100ms
Calculating --------------------------------------
  Retrieve 1000 rows    523.676  (± 0.8%) i/s -      2.652k in   5.064474s

$ ruby bench_sequel_ivar.rb eager_initialize noplugin
Warming up --------------------------------------
  Retrieve 1000 rows    41.000  i/100ms
Calculating --------------------------------------
  Retrieve 1000 rows    419.425  (± 0.2%) i/s -      2.132k in   5.083185s
```

419.425 / 523.676 = 0.80, 20% slower

```
$ gem i activemodel
$ ruby bench_sequel_ivar.rb regular plugin
Warming up --------------------------------------
  Retrieve 1000 rows    43.000  i/100ms
Calculating --------------------------------------
  Retrieve 1000 rows    435.382  (± 0.5%) i/s -      2.193k in   5.037051s

$ ruby bench_sequel_ivar.rb eager_initialize plugin
Warming up --------------------------------------
  Retrieve 1000 rows    29.000  i/100ms
Calculating --------------------------------------
  Retrieve 1000 rows    292.735  (± 0.3%) i/s -      1.479k in   5.052414s
```

292.735 / 435.382 = 0.67, 33% slower

```
Ruby 2.6
$ ruby -v
ruby 2.6.6p146 (2020-03-31 revision 67876) [x86_64-linux]

$ ruby bench_sequel_ivar.rb regular noplugin
Warming up --------------------------------------
  Retrieve 1000 rows    49.000  i/100ms
Calculating --------------------------------------
  Retrieve 1000 rows    491.918  (± 0.4%) i/s -      2.499k in   5.080182s

$ ruby bench_sequel_ivar.rb eager_initialize noplugin
Warming up --------------------------------------
  Retrieve 1000 rows    40.000  i/100ms
Calculating --------------------------------------
  Retrieve 1000 rows    396.391  (± 1.3%) i/s -      2.000k in   5.046391s
```

396.391 / 491.918 = 0.81, 19% slower

```
$ gem i activemodel
$ ruby bench_sequel_ivar.rb regular plugin
Warming up --------------------------------------
  Retrieve 1000 rows    44.000  i/100ms
Calculating --------------------------------------
  Retrieve 1000 rows    443.197  (± 0.2%) i/s -      2.244k in   5.063244s

$ ruby bench_sequel_ivar.rb eager_initialize plugin
Warming up --------------------------------------
  Retrieve 1000 rows    27.000  i/100ms
Calculating --------------------------------------
  Retrieve 1000 rows    273.062  (± 0.4%) i/s -      1.377k in   5.042868s
```

273.062 / 443.197 = 0.62, 38% slower

#19 - 08/15/2020 12:23 PM - Eregon (Benoit Daloze)

I tried on TruffleRuby and there I can't see a significant difference (at least for the noplugin case):
https://gist.github.com/eregon/e552bf55d42ce128a9d89f41d57b637f

TruffleRuby uses inline caches for almost all metaprogramming operations.
And so TruffleRuby optimizes instance_variable_set :@ivar, value to be essentially no different than @ivar = value when the ivar name is always the same.

So this might mean a good way to speed this case and remove this difference is to have an inline cache for instance_variable_set on CRuby too.

Regarding method call overhead, I think this is biased by CRuby without JIT having no inlining.
The method call overhead is something I would expect any Ruby JIT to eventually remove to at least some degree (there is essentially no overhead in TruffleRuby when inlined, the same as if manually inlining the method).
I think MJIT can inline some pure-Ruby methods (not sure if in master yet or not).

If you wanted to optimize to the extreme for the CRuby interpreter and no JIT, then one could use tricks like OptCarrot --opt, for instance in this case to generate a single #init_with method with all assignments.
I don't think that's a good idea though, and I hope it shows my point we shouldn't over-optimize for one specific way of running Ruby.

jeremyevans0 (Jeremy Evans) wrote in #note-13:

> > So, for this case, how about using def foo; @foo ||= nil; end? That would avoid the warning and still let you assign the @ivar lazily, no?
>
> Then you need a method call instead of an ivar access, which is also slower. It also would require more changes across the library.

Not necessarily, you could (manually) inline it if that's a concern (but as mentioned above, I think a JIT would easily inline it for you).
It's probably worth a method though if > 1 usage and a non-trivial default value.
I actually found a few of those in Sequel, all related to this benchmark:

```
defined?(@new) ? @new : (@new = false)
@errors ||= errors_class.new
@changed_columns ||= []
@associations ||= {}
```

(these 4 are all in their own method)

Initializing those to nil is redundant in the benchmark.
Results on master in i/s: 524 no set, 419 set all, 450 do not set redundant.

I think the last 3 make a lot of sense to initialize lazily since they involve additional allocations and might never be used.

For the first one, it's too bad that @new ||= false will repetitively assign to false, but defined?() is indeed a workaround for that.
defined?() also has the advantage it's possible to not set the @ivar at all as long as it does not need to change from the default (defined?(@new) ? @new : false).

In other words, that approach is already used, works well on existing versions, and has no overhead. It's not very pretty, but this is a micro-optimization. For pretty/readable, eager initialization seems better.

#20 - 08/15/2020 03:42 PM - jeremyevans0 (Jeremy Evans)

Eregon (Benoit Daloze) wrote in #note-18:

> > I ran the measurements on both CRuby master and 2.6.6, with sqlite3 for convenience.
> > I see smaller differences, but also my results are about 6 times faster.
> > It's still a larger difference than I expect so I'll try to dig deeper.
>
> Which version did you run with? Are you sure it's a build with default optimizations?

I ran it using Ruby 2.7.1 on OpenBSD, using the OpenBSD package, which builds with -O2. This is the same build I use to run my production applications.

Eregon (Benoit Daloze) wrote in #note-19:

> I tried on TruffleRuby and there I can't see a significant difference (at least for the noplugin case):
> https://gist.github.com/eregon/e552bf55d42ce128a9d89f41d57b637f

What's the difference in the plugin case?

```
defined?(@new) ? @new : (@new = false)
@errors ||= errors_class.new
@changed_columns ||= []
@associations ||= {}
```

(these 4 are all in their own method)

Initializing those to nil is redundant in the benchmark.
Results on master in i/s: 524 no set, 419 set all, 450 do not set redundant.

There are other cases where the instance variables are accessed directly:

```
@errors = @errors.dup if @errors
@changed_columns = @changed_columns.dup if @changed_columns
@associations.clear if @associations
```

So initializing those three to nil is not redundant in the benchmark. Even if you skipped initializing the four, it's 16% faster instead of 25% faster. That's still more than enough reason to use the lazy initialization approach.

#21 - 09/01/2020 03:56 PM - jeremyevans0 (Jeremy Evans)

At the last dev meeting, matz (Yukihiro Matsumoto) said he did not like the callback API, and decided to postpone discussion on this.

#22 - 09/02/2020 02:57 PM - Eregon (Benoit Daloze)

FWIW I noticed that using attr_reader does not warn if the @ivar is not set:

```
$ ruby -w -e 'class T; def foo; @foo; end; end; t=T.new; p t.foo'
-e:1: warning: instance variable @foo not initialized
nil

$ ruby -w -e 'class T; attr_reader :foo; end; t=T.new; p t.foo'
nil
```

Which sounds like an easy way to speed up that case in Sequel.
jeremyevans0 (Jeremy Evans) Could you try that maybe? (so no initialization, but all accesses use the generated reader methods, which can be private)

OTOH this inconsistency seems weird to me.
Maybe we should not have the uninitialized @ivar warning at all?
I do like that warning for encouraging initializing @ivars in the constructor though, since that helps readability.
But the fact there will be no warning when using attr_reader :ivar seems unfortunate.
Maybe attr_reader should take a warn_if_uninitialized: keyword argument?

#23 - 09/02/2020 03:35 PM - jeremyevans0 (Jeremy Evans)

Eregon (Benoit Daloze) wrote in #note-22:

> FWIW I noticed that using attr_reader does not warn if the @ivar is not set:
>
> ```
> $ ruby -w -e 'class T; def foo; @foo; end; end; t=T.new; p t.foo'
> -e:1: warning: instance variable @foo not initialized
> nil
>
> $ ruby -w -e 'class T; attr_reader :foo; end; t=T.new; p t.foo'
> nil
> ```
>
> Which sounds like an easy way to speed up that case in Sequel.
> jeremyevans0 (Jeremy Evans) Could you try that maybe? (so no initialization, but all accesses use the generated reader methods, which can be private)

I don't want to define private methods for all of the instance variables in use just to work around verbose warnings. The attr_reader approach is definitely faster than the approach that eagerly initializes instance variables, but still slower than accessing the instance variables directly (instance variable access is ~25% faster than attr_reader method calls).

> OTOH this inconsistency seems weird to me.
> Maybe we should not have the uninitialized @ivar warning at all?

That was my first proposal to fix the issue, back in 2014 (#10396). That ticket was closed by a commit that did something different than I proposed, with no discussion of the differences.

Since then, I've softened a little and understand that the current verbose warning is helpful in some cases, especially to new programmers. That is

why allowing for custom behavior per-object is so beneficial. High performance code can turn off the warnings, while they are still used by default to help new programmers. A single global setting as we have now is a suboptimal.

> I do like that warning for encouraging initializing @ivars in the constructor though, since that helps readability.
> But the fact there will be no warning when using attr_reader :ivar seems unfortunate.
> Maybe attr_reader should take a warn_if_uninitialized: keyword argument?

There was a bug filed for the fact that attr_reader doesn't warn (#9815), which was rejected. I'm not in favor of adding keyword arguments to attr_reader.

Files

t.rb	5.59 KB	08/03/2020	jeremyevans0 (Jeremy Evans)