Ruby master - Misc #16157

What is the correct and *portable* way to do generic delegation?

09/09/2019 02:10 PM - Dan0042 (Daniel DeLorme)

Status:	Open	
Priority:	Normal	
Assignee:		

Description

With the keyword argument changes in 2.7 we must now specify keyword arguments explicitly when doing generic delegation. But this change is not compatible with 2.6, where it adds an empty hash to the argument list of methods that do not need/accept keyword arguments.

To illustrate the problem:

```
class ProxyWithoutKW < BasicObject
  def initialize(target)
    @target = target
  end
  def method_missing(*a, &b)
    @target.send(*a, &b)
  end
end

class ProxyWithKW < BasicObject
  def initialize(target)
    @target = target
  end
  def method_missing(*a, **o, &b)
    @target.send(*a, **o, &b)
  end
end

class Test
  def args(*a)    a  end
  def arg(a)      a  end
  def opts(**o)   o  end
end
                                          # 2.6         2.7               3.0
ProxyWithoutKW.new(Test.new).args(42)     # [42]        [42]              [42]          ok
ProxyWithoutKW.new(Test.new).arg(42)      # 42          42                42            ok
ProxyWithoutKW.new(Test.new).opts(k: 42)
# {:k=>42}    {:k=>42} +warn    [{:k=>42}]  incompatible with >= 2.7
ProxyWithKW.new(Test.new).args(42)
# [42, {}]    [42]              [42]        incompatible with <= 2.6
ProxyWithKW.new(Test.new).arg(42)
# error       42                42          incompatible with <= 2.6
ProxyWithKW.new(Test.new).opts(k: 42)
# {:k=>42}    {:k=>42} +warn    {:k=>42}    must ignore warning? cannot use pass_positional_hash in
2.6
```

I don't know how to solve this, so I'm asking for the **official** correct way to write portable delegation code. And by **portable** I mean code that can be used in gems that target ruby 2.6 and above.

Related issues:

Related to Ruby master - Feature #14183: "Real" keyword argument	**Closed**
Related to Ruby master - Misc #16188: What are the performance implications o...	**Open**
Related to Ruby master - Feature #16253: Shorthand "forward everything" syntax	**Closed**
Related to Ruby master - Feature #16897: General purpose memoizer in Ruby 3 w...	**Open**

History

#1 - 09/09/2019 02:22 PM - jeremyevans0 (Jeremy Evans)

I haven't tested this, but I think it should work the same between 2.0 - 3.0 (assuming #16154 is accepted):

```
class ProxyWithKW < BasicObject
  def initialize(target)
    @target = target
  end
  def method_missing(*a, **o, &b)
    if o.empty?
      @target.send(*a, &b)
    else
      @target.send(*a, **o, &b)
    end
  end
  pass_positional_hash(:method_missing) if respond_to?(:pass_positional_hash, false)
end
```

#2 - 09/09/2019 05:09 PM - Dan0042 (Daniel DeLorme)

I've compiled your delegate-keyword-argument-separation branch to test this. BTW in your example it should be respond_to?(:pass_positional_hash, true)

The portability result is... Almost there, but not quite. In 2.6 and 2.7, o.empty? cannot distinguish between foo() and foo({}). This results in the discrepancies found below via this test script.

```
class ProxyOld < BasicObject
  def initialize(target)
    @target = target
  end
  def method_missing(*a, &b)
    @target.send(*a, &b) rescue $!
  end
end

class ProxyNew < BasicObject
  def initialize(target)
    @target = target
  end
  def method_missing(*a, **o, &b)
    if o.empty?
      @target.send(*a, &b)
    else
      @target.send(*a, **o, &b)
    end rescue $!
  end
  pass_positional_hash(:method_missing) if respond_to?(:pass_positional_hash, true)
end

class Test
  def args(*a)   a  end
  def arg(a)     a  end
  def opts(**o)  o  end
  def arg0o(a=nil, **o) [a,o] end
  def arg1o(a, **o)     [a,o] end
end

e = Object.new
def e.write(*args) $stdout.print("warn! ") end
$stderr = e

opt = {}
hash = {k:42}
proxy = (ARGV.first=="old" ? ProxyOld : ProxyNew).new(Test.new)
p proxy.args(42)
p proxy.arg(42)
p proxy.arg({k:42})
p proxy.arg({})
p proxy.opts(k: 42)
p proxy.arg0o(hash)
p proxy.arg0o(hash, **{})
p proxy.arg0o(hash, **opt)
```

I've highlighted the two cases where we have the correct behavior in 2.6 ProxyOld and 3.0 ProxyNew, but not the two others.

testcase	2.6 ProxyOld	2.6 ProxyNew	2.7 ProxyNew	3.0 ProxyNew
proxy.args(42)	[42]	[42]	[42]	[42]
proxy.arg(42)	42	42	42	42

testcase	2.6 ProxyOld	2.6 ProxyNew	2.7 ProxyNew	3.0 ProxyNew
proxy.arg({k:42})	{:k=>42}	{:k=>42}	{:k=>42}	{:k=>42}
proxy.arg({})	{}	**error**	**error**	{}
proxy.opts(k: 42)	{:k=>42}	{:k=>42}	{:k=>42}	{:k=>42}
proxy.arg0o(hash)	[nil, {:k=>42}]	[nil, {:k=>42}]	warn! warn! [nil, {:k=>42}]	[{:k=>42}, {}]
proxy.arg0o(hash, **{})	[nil, {:k=>42}]	[nil, {:k=>42}]	warn! warn! [nil, {:k=>42}]	[{:k=>42}, {}]
proxy.arg0o(hash, **opt)	[{:k=>42}, {}]	**[nil, {:k=>42}]**	**warn! warn! [nil, {:k=>42}]**	[{:k=>42}, {}]

#3 - 09/09/2019 07:36 PM - jeremyevans0 (Jeremy Evans)

Dan0042 (Daniel DeLorme) wrote:

> I've compiled your delegate-keyword-argument-separation branch to test this. BTW in your example it should be respond_to?(:pass_positional_hash, true)
>
> The portability result is... Almost there, but not quite. In 2.6 and 2.7, o.empty? cannot distinguish between foo() and foo({}). This results in the discrepancies found below via this test script.

Thank you for working on this and providing the detailed example. I think your analysis is correct. We could handle this in 2.7 by introducing a method to check for whether positional->keyword hash conversion was done, but as we cannot change the behavior in 2.0-2.6, it looks like we cannot use the same method definition to delegate to all possible target method definitions with all possible arguments and be backwards compatible with <2.7. For best compatibility, you will need separate method definitions for <2.7 and 2.7+:

```
class Proxy < BasicObject
  def initialize(target)
    @target = target
  end
  if ::RUBY_VERSION < '2.7'
    def method_missing(*a, &b)
      @target.send(*a, &b) rescue $!
    end
  else
    def method_missing(*a, **o, &b)
      @target.send(*a, **o, &b) rescue $!
    end
    pass_positional_hash(:method_missing) if respond_to?(:pass_positional_hash, true)
  end
end

class Test
  def args(*a)    a  end
  def arg(a)      a  end
  def opts(**o)   o  end
  def arg0o(a=nil, **o) [a,o] end
  def arg1o(a, **o)     [a,o] end
end

e = Object.new
def e.write(*args) $stdout.print("warn! ") end
$stderr = e

opt = {}
hash = {k:42}
proxy =  Proxy.new(Test.new)
p proxy.args(42)
p proxy.arg(42)
p proxy.arg({k:42})
p proxy.arg({})
p proxy.opts(k: 42)
p proxy.arg0o(hash)
p proxy.arg0o(hash, **{})
p proxy.arg0o(hash, **opt)
```

This results in the following behavior for 2.6.4 and 2.7.0 with the delegate-keyword-argument-separation branch:

```
ruby 2.6.4p104     | ruby 2.7.0-delegate-keyword-argument-separation
[42]               | [42]
42                 | 42
{:k=>42}           | {:k=>42}
{}                 | {}
```

```
{:k=>42}           | {:k=>42}
[nil, {:k=>42}]    | warn! warn! [nil, {:k=>42}]
[nil, {:k=>42}]    | [{:k=>42}, {}]
[{:k=>42}, {}]     | [{:k=>42}, {}]
```

The only behavior difference is for **{}, which was ignored by the parser in 2.6, and is not likely to appear in production code. The only warning in 2.7 is for proxy.arg0o(hash), which correctly warns in 2.7 as the behavior will change in 3.0 to be the same as proxy.arg0o(hash, **opt) in 2.6.

Note that could use metaprogramming to reduce the duplication:

```
class Proxy < BasicObject
  def initialize(target)
    @target = target
  end

  kw = ", **kw" if ::RUBY_VERSION >= '2.7'
  class_eval(<<-END, __FILE__, __LINE__+1)
    def method_missing(*a#{kw}, &b)
      @target.send(*a#{kw}, &b) rescue $!
    end
  END
  pass_positional_hash(:method_missing) if respond_to?(:pass_positional_hash, true)
end
```

#4 - 09/10/2019 02:41 PM - Dan0042 (Daniel DeLorme)

Hmm, ok, that's what I was afraid of. I mean, it's not exactly a pretty solution. And it's not limited to method_missing; any method that accepts *args and forwards it to another method may have to be changed. Even if it doesn't have to be changed, it has to be checked, which is possibly even more work than a simple find-and-replace.

One example that comes to mind is in the stdlib; all the files in json/add/*.rb have something like this which may need to be updated:

```
  def to_json(*args)
    as_json.to_json(*args)
  end
```

Of course since that's the stdlib we don't need the RUBY_VERSION check, but for rubygems.... the issue is really how much is there to fix? If it's a small amount, even an ugly-ish solution can be good enough.

Actually, this could be checked lexically to some extent...

[workworkwork]

I've compiled a list of the most popular gems by number of downloads. https://pastebin.com/MurhpP2j
And then installed the top 500. Including pre-installed gems and dependencies, this results in 679 gems (including a few multiple versions). https://pastebin.com/KajXEt7A
Then I search through all .rb files for methods that should be checked and/or updated to use the RUBY_VERSION check if they want to stay compatible with 2.6. https://pastebin.com/cmmjMDW8

Result:
in gems/*/lib: 1949 matches in 1095 files of 225 gems: https://pastebin.com/HNU1cZcD
in gems/*/others: 256 matches in 167 files of 63 gems: https://pastebin.com/eTciQAc0

That's... a **lot** more than I expected. And only with the top 500 gems. If you can check my methodology...

But if I'm correct, I think that the migration required for syntax-based keyword separation goes beyond just "challenging" and into the realm of "unrealistic". Of course that's a decision for Matz to take...

#5 - 09/10/2019 04:37 PM - jeremyevans0 (Jeremy Evans)

Note that the RUBY_VERSION check is only needed in a subset of the cases. In cases where the target method does not accept keyword arguments, no changes are needed (no need to introduce keyword arguments at all). In cases where the last positional argument is not a hash (and even in many of those cases), the simpler code that doesn't require RUBY_VERSION will work. The only time you really need the RUBY_VERSION check is for complete argument delegation to arbitrary methods with arbitrary arguments.

I took a brief look at some things caught by your methodology. I decided to look at ActiveSupport since that is one of the more popular gems. Here are a few examples of delegation and a discussion of whether changes are needed.

```
# lib/active_support/values/time_zone.rb
    def local(*args)
      time = Time.utc(*args)
      ActiveSupport::TimeWithZone.new(nil, self, time)
    end

    def at(*args)
```

```
      Time.at(*args).utc.in_time_zone(self)
    end
```

Most methods written in C do not care if they are called with keyword arguments or a positional hash argument and will work with either. So these cases probably do not need to be updated.

```
# lib/active_support/time_with_zone.rb
    def method_missing(sym, *args, &block)
      wrap_with_time_zone time.__send__(sym, *args, &block)
    rescue NoMethodError => e
      raise e, e.message.sub(time.inspect, inspect), e.backtrace
    end
```

This is pure delegation code to arbitrary methods. However, I think time is an instance of Time, this will not require changes. Now, users can define their own methods on Time in Ruby that accept keyword arguments, and if you want to handle that, you may want to update the code to handle keyword arguments.

```
#lib/active_support/testing/setup_and_teardown.rb
        # Add a callback, which runs before <tt>TestCase#setup</tt>.
        def setup(*args, &block)
          set_callback(:setup, :before, *args, &block)
        end

        # Add a callback, which runs after <tt>TestCase#teardown</tt>.
        def teardown(*args, &block)
          set_callback(:teardown, :after, *args, &block)
        end
```

set_callback doesn't accept keyword arguments, it treats the positional arguments as an array. So no changes are needed here.

```
#lib/active_support/tagged_logging.rb
      def tagged(*tags)
        new_tags = push_tags(*tags)
        yield self
      ensure
        pop_tags(new_tags.size)
      end

    def tagged(*tags)
      formatter.tagged(*tags) { yield self }
    end
```

Here tagged at the bottom delegates all arguments to tagged at the top, which delegates all arguments to push_tags. However, as push_tags does not accept keyword arguments, no changes are needed.

I tried your scanner using Sequel (in the 500 top gems). 143 matches in 63 files, none of which actually required changes. There were a few changes needed in Sequel which the scanner didn't catch, involving passing option hashes to CSV using ** instead of as a positional argument.

In general, I would not recommend using a lexical scanner to try to find cases to fix. Run the tests for the program/library, and fix any warnings. Even for decent sized programs/libraries, it does not take very long to fix the related issues. For many cases where you would have to fix issues, there are already other existing issues due to the implicit conversion between keyword and positional arguments.

#6 - 09/10/2019 07:14 PM - Dan0042 (Daniel DeLorme)

> Note that the RUBY_VERSION check is only needed in a subset of the cases. In cases where the target method does not accept keyword arguments, no changes are needed

Yes, I know, that's exactly what I was saying.

But my point was that "Even if it doesn't have to be *changed*, it has to be *checked*, which is possibly even more work". Although now I realize that checking is easy if the tests are comprehensive enough. And by that I mean the tests have to cover not only foo(42) and foo(42, flag:true) but also (less obviously) wrapfoo(42) and wrapfoo(42, flag:true). But even without tests you can get the warnings from production logs. The lexical analysis was just intended to get a rough idea but I think I got a bit too caught up.

> The only time you really need the RUBY_VERSION check is for complete argument delegation to arbitrary methods with arbitrary arguments.

Ah yes, I made a mistake there. So in my example if to_json(*args) outputs a warning, it's ok to change it to to_json(*args,**kw) even in 2.6 since it's very unlikely you'd be delegating to two different to_json methods, one with kwargs and one without.

So the migration procedure looks like this I think?

```
if you get a warning
   if you are delegating to a specific method
      use (*args, **kw)
   else
      check RUBY_VERSION to delegate via (*args) or (*args, **kw)
else
   don't change anything, otherwise it will break on 2.6
```

> Most methods written in C do not care if they are called with keyword arguments or a positional hash argument and will work with either.

Wow, really? This is a bit off-topic but can you explain why C methods have no trouble with the hash/keyword ambiguity? I would have assumed it was the same as with ruby methods.

Well, I hope everyone has comprehensive test suites.

I hope everyone will understand that just adding **kw can result in bugs on 2.6.

I hope this migration will go as smoothly as you think it will.

Disclaimer: I may be a worrywart.

#7 - 09/10/2019 08:09 PM - jeremyevans0 (Jeremy Evans)

Dan0042 (Daniel DeLorme) wrote:

> > The only time you really need the RUBY_VERSION check is for complete argument delegation to arbitrary methods with arbitrary arguments.
>
> Ah yes, I made a mistake there. So in my example if to_json(*args) outputs a warning, it's ok to change it to to_json(*args,**kw) even in 2.6 since it's very unlikely you'd be delegating to two different to_json methods, one with kwargs and one without.

Let's say the target method is foo(*args, **kw), and the delegating method is bar(*args) foo(*args) end. If you get a keyword argument separation warning when calling foo from bar in 2.7, then changing to bar(*args, **kw) foo(*args, **kw) end should fix the warning, be forward compatible with 3.0, and be backwards compatible back to 2.0. You'll still need to fix callers of bar if they are passing a positional hash as the final positional argument and are not passing keyword arguments, but there will be separate warnings for that.

Note that a method that does not accept keyword arguments will never issue a warning. For methods that do not accept keyword arguments, the keyword arguments are converted to a positional hash argument. This is for backwards compatibility at least back to Ruby 1.6. So you will only get a warning if the called method accepts keywords.

> So the migration procedure looks like this I think?
>
> ```
> if you get a warning
>    if you are delegating to a specific method
>       use (*args, **kw)
>    else
>       check RUBY_VERSION to delegate via (*args) or (*args, **kw)
> else
>    don't change anything, otherwise it will break on 2.6
> ```

That seems reasonable. If you are only delegating to a specific method, then you only get the warning if the target method accepts keywords, in which case accepting and passing the keyword argument should work.

> > Most methods written in C do not care if they are called with keyword arguments or a positional hash argument and will work with either.
>
> Wow, really? This is a bit off-topic but can you explain why C methods have no trouble with the hash/keyword ambiguity? I would have assumed it was the same as with ruby methods.

C functions for Ruby methods that accept keywords must accept the following arguments (since keywords are optional):

```
VALUE c_function_name(int argc, VALUE *argv, VALUE self)
```

self is the method receiver in Ruby, argc is the number of arguments passed in Ruby (with keywords counted as a positional hash), and argv is a pointer such that argv[0]...argv[argc-1] are the arguments passed in Ruby.

Prior to the keyword argument separation branch merge, it was not even possible for a C method to tell if it was called with keywords or called with a positional hash. That is now possible via rb_keyword_given_p, as it is necessary for C methods if they are delegating arguments to another method (e.g. Class#new, Method#call, Object#to_enum).

> Well, I hope everyone has comprehensive test suites.

Agreed.

> I hope everyone will understand that just adding **kw can result in bugs on 2.6.

Me too. The migration would have been much easier if **kw did not send an empty positional argument to methods that do not accept keyword arguments in older Ruby versions. Alas, you cannot change the past.

> I hope this migration will go as smoothly as you think it will.

I didn't mean to imply I think the migration will go smoothly. This is definitely going to require work, especially for large projects, and it may be error prone if the projects lack comprehensive testing. However, the changes are necessary to fix the problems with keyword arguments.

Based on my experience so far (sample size=1), the hardest part of this migration will be convincing gem maintainers that it is worthwhile to have a version that will work correctly in 2.6, 2.7, and 3.0. The actual work to update the code will be less difficult in comparison.

#8 - 09/11/2019 03:36 PM - Dan0042 (Daniel DeLorme)

> Prior to the keyword argument separation branch merge, it was not even possible for a C method to tell if it was called with keywords or called with a positional hash.

That's the same situation as ruby methods used to be in. So *theoretically* C methods are susceptible to the same problem. But very very few C methods have a signature that would allow to trigger the bug. However, feast your eyes on the behavior of warn() (on ruby master)

```
>> warn({x:1})
ArgumentError (unknown keyword: :x)

>> warn({x:1}, uplevel:0)
(irb):4: warning: {:x=>1}

>> warn({x:1}, **{})

>> warn({x:1}, {})

>> warn({x:1}, {}, {})
{:x=>1}

>> warn({x:1}, {}, {}, {})
{:x=>1}
{}

>> warn({x:1}, {y:2}, {})
ArgumentError (unknown keyword: :y)

>> warn({x:1}, {y:2}, {}, {})
{:x=>1}
{:y=>2}
```

Of course this is all *extremely* unconventional usage and doesn't really deserve a fix. But I thought it was weird/interesting.

> The only time you really need the RUBY_VERSION check is for complete argument delegation to arbitrary methods with arbitrary arguments.

For reference, searching for only arbitrary delegation (send/__send__ with *args) gives 179 matches in 158 files of 86 gems (not all of which will require kwargs) https://pastebin.com/HX3w5ngK

> The only behavior difference is for **{}, which was ignored by the parser in 2.6, and is not likely to appear in production code.

Confirmed, not a single appearance in the 679 gems I installed.

> However, the changes are necessary to fix the problems with keyword arguments.

I still disagree with that, but that's another argument, another story ;-)

#9 - 09/11/2019 09:25 PM - jeremyevans0 (Jeremy Evans)

After discussion with some other committers, I have an alternative approach in https://github.com/ruby/ruby/pull/2449. This allows you to do the following:

```
class Foo
  def foo(*args, &block)
    bar(*args, &block)
  end
  pass_keywords :foo if respond_to?(:pass_keywords, true)
end
```

Then if you call foo with keywords, keywords will be passed to bar. This should allow for more backwards compatible behavior with older versions of Ruby.

Using a modified version of your example:

```
class Proxy < BasicObject
  def initialize(target)
    @target = target
  end
  def method_missing(*a, &b)
    @target.send(*a, &b) rescue $!
  end
  pass_keywords(:method_missing) if respond_to?(:pass_keywords, true)
end

class Test
  def noarg() end
  def noarg_o(**o) o end
  def arg(a) a  end
  def arg_o(a, **o) [a,o] end
  def opt_arg(a=nil) a end
  def opt_arg_o(a=nil, **o) [a,o] end
  def args(*a) a end
  def args_o(*a, **o) [a, o] end
end

e = Object.new
def e.write(*args) $stdout.print("warn! ") end
$stderr = e

opt = {}
hash = {k:42}
proxy = Proxy.new(Test.new)
%i'noarg arg opt_arg args'.each do |m|
  [m, :"#{m}_o"].each do |meth|
    p ["#{meth}(opt)", proxy.send(meth, opt)]
    p ["#{meth}(hash)", proxy.send(meth, hash)]
    p ["#{meth}(**hash)", proxy.send(meth, **opt)]
    p ["#{meth}(**opt)", proxy.send(meth, **opt)]
    p ["#{meth}(hash, **opt)", proxy.send(meth, hash, **opt)]
    p ["#{meth}(hash, **hash)", proxy.send(meth, hash, **hash)]
  end
end
```

Here's a table describing the differences and warnings (differences in ArgumentError messages are ignored). All differences are due to empty keyword splats not being passed as positional arguments in 2.7, and I would consider them at least undesired behavior in 2.6, if not an outright bug.

```
    Code                          2.6              2.7-pass_keywords
-------------------------------------------------------------------------------
  | noarg(opt)              | #<ArgE:(1:0)>    | #<ArgE:(1:0)>
  | noarg(hash)             | #<ArgE:(1:0)>    | #<ArgE:(1:0)>
 d| noarg(**opt)            | #<ArgE:(1:0)>    | nil
  | noarg(**hash)           | #<ArgE:(1:0)>    | #<ArgE:unknown keyword: :k>
  | noarg(hash, **opt)      | #<ArgE:(2:0)>    | #<ArgE:(1:0)>
  | noarg(hash, **hash)     | #<ArgE:(2:0)>    | #<ArgE:(1:0)>
w | noarg_o(opt)            | opt              | opt
w | noarg_o(hash)           | hash             | hash
  | noarg_o(**opt)          | opt              | opt
  | noarg_o(**hash)         | hash             | hash
  | noarg_o(hash, **opt)    | #<ArgE:(1:0)>    | #<ArgE:(1:0)>
  | noarg_o(hash, **hash)   | #<ArgE:(1:0)>    | #<ArgE:(1:0)>
  | arg(opt)                | opt              | opt
  | arg(hash)               | hash             | hash
w | arg(**opt)              | opt              | opt
  | arg(**hash)             | opt              | hash
```

```
 d| arg(hash, **opt)         | #<ArgE:(2:1)>    | hash
  | arg(hash, **hash)        | #<ArgE:(2:1)>    | #<ArgE:unknown keyword: :k>
  | arg_o(opt)               | [opt, opt]       | [opt, opt]
  | arg_o(hash)              | [hash, opt]      | [hash, opt]
w | arg_o(**opt)             | [opt, opt]       | [opt, opt]
w | arg_o(**hash)            | [hash, opt]      | [hash, opt]
  | arg_o(hash, **opt)       | [hash, opt]      | [hash, opt]
  | arg_o(hash, **hash)      | [hash, hash]     | [hash, hash]
  | opt_arg(opt)             | opt              | opt
  | opt_arg(hash)            | hash             | hash
 d| opt_arg(**opt)           | opt              | nil
  | opt_arg(**hash)          | hash             | hash
 d| opt_arg(hash, **opt)     | #<ArgE:(2:0..1)>| hash
  | opt_arg(hash, **hash)    | #<ArgE:(2:0..1)>| #<ArgE:unknown keyword: :k>
w | opt_arg_o(opt)           | [nil, opt]       | [nil, opt]
w | opt_arg_o(hash)          | [nil, hash]      | [nil, hash]
  | opt_arg_o(**opt)         | [nil, opt]       | [nil, opt]
  | opt_arg_o(**hash)        | [nil, hash]      | [nil, hash]
  | opt_arg_o(hash, **opt)   | [hash, opt]      | [hash, opt]
  | opt_arg_o(hash, **hash)  | [hash, hash]     | [hash, hash]
  | args(opt)                | [opt]            | [opt]
  | args(hash)               | [hash]           | [hash]
 d| args(**opt)              | [opt]            | []
  | args(**hash)             | [hash]           | [hash]
 d| args(hash, **opt)        | [hash, opt]      | [hash]
  | args(hash, **hash)       | [hash, hash]     | [hash, hash]
w | args_o(opt)              | [[], opt]        | [[], opt]
w | args_o(hash)             | [[], hash]       | [[], hash]
  | args_o(**hash)           | [[], opt]        | [[], opt]
  | args_o(**opt)            | [[], opt]        | [[], opt]
  | args_o(hash, **opt)      | [[hash], opt]    | [[hash], opt]
  | args_o(hash, **hash)     | [[hash], hash]   | [[hash], hash]
```

Every time a warning is issued, behavior is the same in 2.7 as in 2.6. For each warning, here will be the behavior in 3.0:

```
    Code                 2.6 & 2.7           3.0
-------------------------------------------------------
w | noarg_o(opt)     | opt              | #<ArgE:(1:0)>
w | noarg_o(hash)    | hash             | #<ArgE:(1:0)>
w | arg(**opt)       | opt              | #<ArgE:(0:1)>
w | arg_o(**opt)     | [opt, opt]       | #<ArgE:(0:1)>
w | arg_o(**hash)    | [hash, opt]      | #<ArgE:(0:1)>
w | opt_arg_o(opt)   | [nil, opt]       | [opt, opt]
w | opt_arg_o(hash)  | [nil, hash]      | [hash, opt]
w | args_o(opt)      | [[], opt]        | [[opt], opt]
w | args_o(hash)     | [[], hash]       | [[hash], opt]
```

#10 - 09/11/2019 11:20 PM - jeremyevans0 (Jeremy Evans)

Dan0042 (Daniel DeLorme) wrote:

> Of course this is all *extremely* unconventional usage and doesn't really deserve a fix. But I thought it was weird/interesting.

Very interesting actually. However, while you think this method is written in C, it's actually not, at least not the version you are calling:

```
$ ruby --disable-gems -e 'warn({x: 1})'
$ ruby -e 'warn({x: 1})'
-e:1: warning: The last argument is used as the keyword parameter
/home/jeremy/tmp/ruby/lib/rubygems/core_ext/kernel_warn.rb:15: warning: for method defined here
Traceback (most recent call last):
        1: from -e:1:in `<main>'
/home/jeremy/tmp/ruby/lib/rubygems/core_ext/kernel_warn.rb:15:in `block in <module:Kernel>': unknown keyword:
:x (ArgumentError)
```

Rubygems overwrites Kernel#warn! This is part of the implementation:

```
    module_function define_method(:warn) {|*messages, uplevel: nil|
      unless uplevel
        return original_warn.call(*messages)
      end
      # filter callers
      original_warn.call(*messages, uplevel: uplevel)
    end
```

So the C version is loose with the keywords, in the sense that invalid keywords will be ignored. The Rubygems version is strict with the keywords, in that passing keywords other than :uplevel will result in an error. The Rubygems version will then call the original version without keywords arguments. Combined, this causes the very weird behavior you see. Let's go through each example:

```
>> warn({x:1})
ArgumentError (unknown keyword: :x)
```

Happens when the Rubygems version is called, because :x is not a valid keyword.

```
>> warn({x:1}, uplevel:0)
(irb):4: warning: {:x=>1}
```

Rubygems version recognizes uplevel keyword and passes it to C version

```
>> warn({x:1}, **{})
```

Passes no keywords to the Rubygems version, so Rubygems version passes {x: 1} as the sole argument to the C version, which is treated as keyword arguments and ignored.

```
>> warn({x:1}, {})
```

This emits a keyword argument separation warning in the master branch, as the hash is treated as keywords to the Rubygems version. The Rubygems version then passes {x: 1} as the sole argument to the C version, which is treated as keyword arguments and ignored.

```
>> warn({x:1}, {}, {})
{:x=>1}
```

Last hash passed to Rubygems version as keyword arguments with keyword argument separation warning emitted. First two arguments passed to C version. Second argument treated as keyword argument by C version, and first argument treated as warning message.

```
>> warn({x:1}, {}, {}, {})
{:x=>1}
{}
```

Same as previous, except there is an extra argument emitted as a warning.

```
>> warn({x:1}, {y:2}, {})
ArgumentError (unknown keyword: :y)
```

Last hash passed to Rubygems version as keyword arguments with keyword argument separation warning emitted. First two arguments passed to C version. Second argument treated as keyword argument by C version, and ArgumentError raised as C version doesn't support the :y keyword.

```
>> warn({x:1}, {y:2}, {}, {})
{:x=>1}
{:y=>2}
```

Same as the warn({x:1}, {}, {}, {}) case other than the elements of the 2nd argument.

I sent a pull request to Rubygems to fix this behavior (https://github.com/rubygems/rubygems/pull/2911). Personally, I think it would be better for them to drop the override of Kernel#warn.

#11 - 09/12/2019 02:05 AM - Dan0042 (Daniel DeLorme)

> Then if you call foo with keywords, keywords will be passed to bar. This should allow for more backwards compatible behavior with older versions of Ruby.

Ah, great! So we no longer need the RUBY_VERSION check!

I have a feeling the pass_keywords behavior would be appropriate for 99% of methods with an argument splat but no keywords. In that case maybe it would make things simpler to turn on the behavior by default and turn it off via do_not_pass_keywords or something? Just a thought.

> All differences are due to empty keyword splats not being passed as positional arguments in 2.7, and I would consider them at least undesired behavior in 2.6, if not an outright bug.

Maybe often undesired, but not always. Without that behavior, opt_arg_o(hash, **opt) would pass hash as kwarg in 2.6.

#12 - 09/12/2019 12:53 PM - Dan0042 (Daniel DeLorme)

After some more thinking I believe the pass_keywords behavior would be appropriate for 100% of methods with an argument splat but no keywords.

- If forwarding to a method with no keyword arguments
  - The kwarg will be converted to positional hash anyway
- If forwarding to a method with keyword arguments
  - If the intention is to pass a kwarg
    - Will work as expected
  - If the intention is to pass a positional hash
    - This behavior is currently impossible in 2.6, so there would be no compatibility problem. In order to activate the new behavior of passing a positional hash, use pass_positional_hash or whatever the name would be.

#13 - 09/12/2019 03:40 PM - jeremyevans0 (Jeremy Evans)

Dan0042 (Daniel DeLorme) wrote:

> After some more thinking I believe the pass_keywords behavior would be appropriate for 100% of methods with an argument splat but no keywords.

We do not want this as the default behavior. Consider:

```
def foo(*args)
  bar(*quux)
  baz(*args)
end
```

Here in the call to bar, the purpose is not to pass arguments through, but to call the method quux and splat the arguments received from that method as positional arguments. If bar accepts a positional argument splat and keyword arguments, and quux returns an array where the last element is a hash, then the last element will be treated as keywords to bar, even though that is not what we want (these are the types of bugs keyword argument separation is designed to fix).

pass_keywords is designed only for arbitrary delegation methods, and only where backwards compatibility with Ruby <2.7 is desired. It should eventually be removed after Ruby <2.7 is EOL. As the implementation is based on VM frame flags, it requires extra internal work in the VM for every method call.

#14 - 09/12/2019 07:57 PM - Dan0042 (Daniel DeLorme)

Ok, I misunderstood what pass_keywords was doing; I thought it would only apply to baz(*args) in this case.

#15 - 09/13/2019 02:55 PM - Dan0042 (Daniel DeLorme)

There's something I'd like to clarify.

With 2.7, people will be asked to migrate their generic delegations to use pass_keywords. Then at some point in the future when 2.6 is no longer supported, pass_keywords will be deprecated and they'll be asked to migrate their generic delegations a second time, to use (*args, **kw)

Is that the idea?

#16 - 09/13/2019 03:15 PM - jeremyevans0 (Jeremy Evans)

Dan0042 (Daniel DeLorme) wrote:

> With 2.7, people will be asked to migrate their generic delegations to use pass_keywords. Then at some point in the future when 2.6 is no longer supported, pass_keywords will be deprecated and they'll be asked to migrate their generic delegations a second time, to use (*args, **kw)
>
> Is that the idea?

pass_keywords is proposed only to make it easier to continue to support older versions of Ruby with the same method definition. I don't think we will be pushing people to use pass_keywords, merely making it available as an option.

You are correct that pass_keywords is currently envisioned as a temporary solution, not a permanent one. However, there is nothing technically preventing it from being a permanent solution.

For 2.7+, delegation using (*args, **kw) should work the same way (*args) works in Ruby <2.7. If you only want to migrate once and still want to support backwards compatibility, you can certainly switch to having separate definitions, using the current one for <2.7 and adding one for 2.7+. Then you could drop the <2.7 definition when you drop support for Ruby <2.7.

#17 - 09/13/2019 05:00 PM - Dan0042 (Daniel DeLorme)

In that case I'd like to make one last suggestion.

For a method with *args, if the method is called with keyword arguments, flag args in some way (instance variable?) so that, in that method only, a call with *args would result in the last argument being passed as a keyword argument. Yes, it's a hack, but it's a better hack than pass_keywords imho.

This would make the migration less tedious, less confusing, less error-prone, and postpone this whole delegation mess to a date when it's easier to deal with. I really think this migration should be split into "should be done now" and "easier to do once 2.6 is no longer supported". My 2¢

#18 - 09/13/2019 05:54 PM - jeremyevans0 (Jeremy Evans)

Dan0042 (Daniel DeLorme) wrote:

> In that case I'd like to make one last suggestion.
>
> For a method with *args, if the method is called with keyword arguments, flag args in some way (instance variable?) so that, in that method only, a call with *args would result in the last argument being passed as a keyword argument. Yes, it's a hack, but it's a better hack than pass_keywords imho.

You can't make this an instance-specific flag and easily contain the scope to that method only. The user could pass args (not *args) as a argument to another method and splat args in that method.

Also, using an instance-specific flag doesn't handle this type of delegation:

```
def foo(*args)
  bar(*args.map{|arg| something(arg)})
end
```

> This would make the migration less tedious, less confusing, less error-prone, and postpone this whole delegation mess to a date when it's easier to deal with. I really think this migration should be split into "should be done now" and "easier to do once 2.6 is no longer supported". My 2¢

The approach you are proposing handles the simplest case fine and opens another Pandora's box of issues for other cases. And if you want to split the migration into "should be done now" vs. "easier to do once 2.6 is no longer supported", that's exactly what pass_keywords allows. You can use pass_keywords now, and once 2.6 is no longer supported, you can switch all delegation to (*args, **kw).

#19 - 10/14/2019 05:55 PM - Eregon (Benoit Daloze)

- Related to Feature #14183: "Real" keyword argument added

#20 - 10/14/2019 05:56 PM - Eregon (Benoit Daloze)

- Related to Misc #16188: What are the performance implications of the new keyword arguments in 2.7 and 3.0? added

#21 - 10/14/2019 06:10 PM - jeremyevans0 (Jeremy Evans)

- Status changed from Open to Closed

pass_keywords was renamed to ruby2_keywords. The correct and portable way to handle generic delegation is now:

```
class Proxy < BasicObject
  def initialize(target)
    @target = target
  end
  def method_missing(*a, &b)
    @target.send(*a, &b)
  end
  ruby2_keywords :method_missing if respond_to?(:ruby2_keywords, true)
end
```

Basically, keep your existing method definition, and after it add ruby2_keywords :method_name if respond_to?(:ruby2_keywords, true).

That will work in all ruby versions until ruby2_keywords is removed. That is currently planned for after EOL of Ruby 2.6, but potentially it could stay longer.

I fixed ruby2_keywords so it works with define_method. Note that generic delegation through procs is not currently implemented, but it is not difficult to add Proc#ruby2_keywords if it is needed. Please request that as a feature if you want it.

#22 - 10/15/2019 05:37 AM - Eregon (Benoit Daloze)

jeremyevans0 (Jeremy Evans) That is not future proof and will break as soon as ruby2_keywords is removed.

I don't think we want to keep using ruby2_keywords forever, i.e., does it make sense to use ruby2_keywords in Ruby 4?
From the name and my understanding of it, ruby2_keywords should only be used in 2.7 for transition, 3.0 later should explicitly pass kwargs.

I think using ruby2_keywords is also really not pretty, so I hope once 2.x support is dropped libraries will not use the ruby2_keywords workaround for delegation anymore.

So I believe this is the currently complete and future-proof way to do delegation that works on Ruby 2.x - 3.0+:

```
class Proxy < BasicObject
  def initialize(target)
    @target = target
  end

  if RUBY_VERSION >= "3.0"
    def method_missing(*args, **kwargs, &block)
      @target.send(*args, **kwargs, &block)
    end
  else
    def method_missing(*args, &block)
      @target.send(*args, &block)
    end
    ruby2_keywords :method_missing if respond_to?(:ruby2_keywords, true) # For 2.7
  end
end
```

We need to discuss when to remove ruby2_keywords before closing this issue, as it affects how we do the version check, so I reopen this.

#23 - 10/15/2019 05:37 AM - Eregon (Benoit Daloze)

- Status changed from Closed to Open

#24 - 10/15/2019 07:23 AM - Eregon (Benoit Daloze)

At the end of my comment in https://bugs.ruby-lang.org/issues/16188#note-5 I discussed why I believe ruby2_keywords is a non-trivial cost on performance, and so should be IMHO only in Ruby 2.7.
I also propose there an explicit way to mark call sites converting the Hash flagged by ruby2_keywords to keywords again with send_keyword_hash.
That I think would make it easier to understand by being less magic and limit the performance cost to only where it's needed.

```
BasicObject.alias_method :send_keyword_hash, :__send__ if RUBY_VERSION < "2.7"

class Proxy < BasicObject
  def initialize(target)
    @target = target
  end

  if RUBY_VERSION >= "3.0"
    def method_missing(*args, **kwargs, &block)
      @target.send(*args, **kwargs, &block)
    end
  else
    def method_missing(*args, &block)
      @target.send_keyword_hash(*args, &block)
    end
    ruby2_keywords :method_missing if respond_to?(:ruby2_keywords, true) # For 2.7
  end
end
```

#25 - 10/15/2019 03:16 PM - jeremyevans0 (Jeremy Evans)

Eregon (Benoit Daloze) wrote:

> jeremyevans0 (Jeremy Evans) That is not future proof and will break as soon as ruby2_keywords is removed.

I did say: That will work in all ruby versions until ruby2_keywords is removed.

> I don't think we want to keep using ruby2_keywords forever, i.e., does it make sense to use ruby2_keywords in Ruby 4?

It certainly could. I believe the idea of removing it after Ruby 2.6 is EOL is that gems that work on supported Ruby versions could continue to have a single method definition. That method definition may need to change if ruby2_keywords is removed, though.

> From the name and my understanding of it, ruby2_keywords should only be used in 2.7 for transition, 3.0 later should explicitly pass kwargs.

The idea behind ruby2_keywords is to be able to use a single method definition for delegation methods, that will work for older ruby versions (where ruby2_keywords is not defined and won't be called), and also newer versions (where ruby2_keywords is defined and will be called). It is designed to require the fewest possible changes to existing code in order to get it to work in Ruby 2.7 without warnings and in Ruby 3.0 without errors.

> I think using ruby2_keywords is also really not pretty, so I hope once 2.x support is dropped libraries will not use the ruby2_keywords workaround for delegation anymore.
>
> So I believe this is the currently complete and future-proof way to do delegation that works on Ruby 2.x - 3.0+:

```
class Proxy < BasicObject
  def initialize(target)
    @target = target
  end

  if RUBY_VERSION >= "3.0"
    def method_missing(*args, **kwargs, &block)
      @target.send(*args, **kwargs, &block)
    end
  else
    def method_missing(*args, &block)
      @target.send(*args, &block)
    end
    ruby2_keywords :method_missing if respond_to?(:ruby2_keywords, true) # For 2.7
  end
end
```

If you have an existing method:

```
def method_missing(*args, &block)
  @target.send(*args, &block)
end
```

The idea with ruby2_keywords is that can currently just add a line so it works in 2.7+:

```
def method_missing(*args, &block)
  @target.send(*args, &block)
end
ruby2_keywords :method_missing if respond_to?(:ruby2_keywords, true)
```

If at some point we do remove ruby2_keywords, you could then switch it to:

```
def method_missing(*args, **kwargs, &block)
  @target.send(*args, **kwargs, &block)
end
```

The assumption is by that point, the code will no longer need to support Ruby <2.7. So at no point does the code need two method definitions to work with all supported Ruby versions.

> At the end of my comment in https://bugs.ruby-lang.org/issues/16188#note-5 I discussed why I believe ruby2_keywords is a non-trivial cost on performance, and so should be IMHO only in Ruby 2.7.

I'll try to respond to that today, but it is quite long.

> I also propose there an explicit way to mark call sites converting the Hash flagged by ruby2_keywords to keywords again with send_keyword_hash.

CRuby's C-API supports this. It's not difficult to implement such a method. However, this explicit approach requires more changes to existing code, which is sort of the opposite goal of ruby2_keywords.

> That I think would make it easier to understand by being less magic and limit the performance cost to only where it's needed.

It's more explicit at the call site where the conversion of hash to keywords takes place. I'm not sure it is any easier to understand for the typical Ruby programmer.

It also requires modifying the method definition. The point of ruby2_keywords is you do not have to modify an existing method definition, you can just add ruby2_keywords :method_name if respond_to?(:ruby2_keywords, true) after the method definition, and things will continue to work as they did before.

#26 - 10/15/2019 03:49 PM - Eregon (Benoit Daloze)

jeremyevans0 (Jeremy Evans) wrote:

> It certainly could. I believe the idea of removing it after Ruby 2.6 is EOL is that gems that work on supported Ruby versions could continue to have a single method definition. That method definition may need to change if ruby2_keywords is removed, though.

As we've seen countless times on bug trackers of many gems, many gems need to support Ruby versions which are EOL'd, because many applications exist and are not updated to the latest Ruby on every release.
In other words, I believe MRI version EOL is not a good deadline for this.
After all, there are other distributions than MRI which might EOL at different times or explicitly support older Ruby versions.

I believe removing ruby2_keywords is going to break those gems if we recommend them to entirely rely on ruby2_keywords.
And since currently nobody knows when ruby2_keywords would be removed, it's a time bomb that would need every such gem to change to fix it.
I doubt anyone wants that.
I'd bet most people would prefer a bit more verbose code that will work in the future than changing code for 2.7, knowing it will break again in the future.

Unless we never remove ruby2_keywords, but that doesn't make sense to me:

- It's a workaround for compatibility, we should not leave it in forever.
- It is modifying the behavior of things like *args which is inconsistent with other methods using *args but not using ruby2_keywords, creating inconsistency.
- We should encourage new code for Ruby 3.0+ to not use such a workaround, by simply not be able to use it.
- It's slowing down every single call with a *rest argument and without explicit keywords.

I think it's not a good idea to propose a migration path that we know will need changes again in the future.

To the best of our abilities, I believe it is our responsibility to provide a clear replacement that will work in all future Ruby versions.
So Rubyists don't need to modify their code a second time and they don't have to drop compatibility with Ruby < 3 at the same time as changing the code for the second time.

#27 - 10/15/2019 03:51 PM - Eregon (Benoit Daloze)

- Related to Feature #16253: Shorthand "forward everything" syntax added

#28 - 10/15/2019 04:28 PM - jeremyevans0 (Jeremy Evans)

Eregon (Benoit Daloze) wrote:

> jeremyevans0 (Jeremy Evans) wrote:
>
> > It certainly could. I believe the idea of removing it after Ruby 2.6 is EOL is that gems that work on supported Ruby versions could continue to have a single method definition. That method definition may need to change if ruby2_keywords is removed, though.
>
> As we've seen countless times on bug trackers of many gems, many gems need to support Ruby versions which are EOL'd, because many applications exist and are not updated to the latest Ruby on every release.
> In other words, I believe MRI version EOL is not a good deadline for this.
> After all, there are other distributions than MRI which might EOL at different times or explicitly support older Ruby versions.

That's a possible argument for keeping ruby2_keywords longer or indefinitely. :)

> I believe removing ruby2_keywords is going to break those gems if we recommend them to entirely rely on ruby2_keywords.
> And since currently nobody knows when ruby2_keywords would be removed, it's a time bomb that would need every such gem to change to fix it.
> I doubt anyone wants that.
> I'd bet most people would prefer a bit more verbose code that will work in the future than changing code for 2.7, knowing it will break again in the future.

Certainly people that want that can already write separate method definitions depending on Ruby version.

As a counterpoint, most popular gems are well maintained and many of those only aim for compatibility with supported Ruby versions. So using ruby2_keywords when it is available and switching away from it if it is removed may be easier for them.

Gems that aren't maintained may have issues in Ruby 3 anyway, due to keyword argument separation. Do you expect there will be a high percentage of gems that are maintained enough to add ruby2_keywords but will not be maintained enough to switch if it is removed?

> Unless we never remove ruby2_keywords, but that doesn't make sense to me:
>
> - It's a workaround for compatibility, we should not leave it in forever.

This depends on how much you value compatibility with older versions. If you want compatibility with older versions, and don't want to define separate methods per version, you would want to keep ruby2_keywords defined.

- It is modifying the behavior of things like *args which is inconsistent with other methods using *args but not using ruby2_keywords, creating inconsistency.

It is true that it results in inconsistency, and can result in behavior changes far away from the method using ruby2_keywords. This is one reason it should be explicit and not the default behavior as some other people have proposed.

- We should encourage new code for Ruby 3.0+ to not use such a workaround, by simply not be able to use it.

Then you are forcing separate method definitions per version if you want to support Ruby 2.6 and 3.0. ruby2_keywords is designed to avoid that.

- It's slowing down every single call with a *rest argument and without explicit keywords.

I believe the slow down is insignificant compared to all the other processing going on (in CRuby). If you have benchmark results that prove otherwise, I would like to review them.

> I think it's not a good idea to propose a migration path that we know will need to change again in the future.
>
> To the best of our abilities, I believe it is our responsibility to provide a clear replacement that will work in all future Ruby versions.
> So Rubyists don't need to modify their code a second time and they don't have to drop compatibility with Ruby < 3 at the same time as changing the code for the second time.

I think that is fair. However, the best solution if we don't want to change again in the future is to support ruby2_keywords indefinitely. Any other approach requires separate methods definitions depending on the version.

I should also point out that the ruby2_keywords approach is even better at backwards compatibility then your proposed separate method definitions, since it works probably back to at least 1.6, whereas with the separate method definitions, it only works as far back as 2.0 (since keyword arguments are a syntax error in Ruby <2).

#29 - 06/05/2020 03:52 PM - Eregon (Benoit Daloze)

- Related to Feature #16897: General purpose memoizer in Ruby 3 with Ruby 2 performance added