Ruby master - Feature #14183

"Real" keyword argument

12/14/2017 06:59 AM - mame (Yusuke Endoh)

Status:	Closed
Priority:	Normal
Assignee:	
Target version:	3.0

Description

In RubyWorld Conference 2017 and RubyConf 2017, Matz officially said that Ruby 3.0 will have "real" keyword arguments. AFAIK there is no ticket about it, so I'm creating this (based on my understanding).

In Ruby 2, the keyword argument is a normal argument that is a Hash object (whose keys are all symbols) and is passed as the last argument. This design is chosen because of compatibility, but it is fairly complex, and has been a source of many corner cases where the behavior is not intuitive. (Some related tickets: #8040, #8316, #9898, #10856, #11236, #11967, #12104, #12717, #12821, #13336, #13647, #14130)

In Ruby 3, a keyword argument will be completely separated from normal arguments. (Like a block parameter that is also completely separated from normal arguments.)
This change will break compatibility; if you want to pass or accept keyword argument, you always need to use bare sym: val or double-splat ** syntax:

```
# The following calls pass keyword arguments
foo(..., key: val)
foo(..., **hsh)
foo(..., key: val, **hsh)

# The following calls pass **normal** arguments
foo(..., {key: val})
foo(..., hsh)
foo(..., {key: val, **hsh})

# The following method definitions accept keyword argument
def foo(..., key: val)
end
def foo(..., **hsh)
end

# The following method definitions accept **normal** argument
def foo(..., hsh)
end
```

In other words, the following programs WILL NOT work:

```
# This will cause an ArgumentError because the method foo does not accept keyword argument
def foo(a, b, c, hsh)
  p hsh[:key]
end
foo(1, 2, 3, key: 42)

# The following will work; you need to use keyword rest operator explicitly
def foo(a, b, c, **hsh)
  p hsh[:key]
end
foo(1, 2, 3, key: 42)

# This will cause an ArgumentError because the method call does not pass keyword argument
def foo(a, b, c, key: 1)
end
h = {key: 42}
foo(1, 2, 3, h)

# The following will work; you need to use keyword rest operator explicitly
```

```
def foo(a, b, c, key: 1)
end
h = {key: 42}
foo(1, 2, 3, **h)
```

I think here is a transition path:

- Ruby 2.6 (or 2.7?) will output a warning when a normal argument is interpreted as keyword argument, or vice versa.
- Ruby 3.0 will use the new semantics.

Related issues:

Related to Backport200 - Backport #8040: Unexpect behavior when using keyword...	**Closed**	**03/08/2013**
Related to Ruby master - Bug #8316: Can't pass hash to first positional argum...	**Closed**	
Related to Ruby master - Bug #9898: Keyword argument oddities	**Closed**	**06/03/2014**
Related to Ruby master - Bug #10856: Splat with empty keyword args gives unex...	**Closed**	
Related to Ruby master - Bug #11236: inconsistent behavior using ** vs hash a...	**Closed**	
Related to Ruby master - Bug #11967: Mixing kwargs with optional parameters c...	**Rejected**	
Related to Ruby master - Bug #12717: Optional argument treated as kwarg	**Closed**	
Related to Ruby master - Bug #12821: Object converted to Hash unexpectedly un...	**Closed**	
Related to Ruby master - Bug #13336: Default Parameters don't work	**Closed**	
Related to Ruby master - Bug #13647: Some weird behaviour with keyword arguments	**Closed**	
Related to Ruby master - Bug #14130: Keyword arguments are ripped from the mi...	**Closed**	
Related to Ruby master - Bug #15078: Hash splat of empty hash should not crea...	**Closed**	
Related to Ruby master - Bug #14415: Empty keyword hashes get assigned to ord...	**Closed**	
Related to Ruby master - Bug #12022: Inconsistent behavior with splatted name...	**Closed**	
Related to Ruby master - Bug #11860: Double splat does not work on empty hash...	**Closed**	
Related to Ruby master - Bug #10708: In a function call, double splat of an e...	**Closed**	
Related to Ruby master - Bug #11068: unable to ommit an optional keyarg if th...	**Closed**	
Related to Ruby master - Bug #11039: method_missing □ *args □□□ symbol □□□□□□...	**Closed**	
Related to Ruby master - Bug #10994: Inconsistent behavior when mixing option...	**Closed**	
Related to Ruby master - Bug #10293: splatting an empty hash in a method invo...	**Closed**	
Related to Ruby master - Bug #15753: unknown keyword when passing an hash to ...	**Closed**	
Related to Ruby master - Misc #16188: What are the performance implications o...	**Open**	
Related to Ruby master - Misc #16157: What is the correct and *portable* way ...	**Open**	

Associated revisions

Revision a1e588d1 - 08/30/2019 10:03 PM - mame (Yusuke Endoh)

NEWS: Hash-to-keywords automatic conversion is now warned

A follow up for 16c6984bb9..b5b3afadfa. [Feature #14183]

History

#1 - 12/14/2017 07:23 AM - hsbt (Hiroshi SHIBATA)

- Related to Backport #8040: Unexpect behavior when using keyword arguments added

#2 - 12/14/2017 07:23 AM - hsbt (Hiroshi SHIBATA)

- Related to Bug #8316: Can't pass hash to first positional argument; hash interpreted as keyword arguments added

#3 - 12/14/2017 07:23 AM - hsbt (Hiroshi SHIBATA)

- Related to Bug #9898: Keyword argument oddities added

#4 - 12/14/2017 07:24 AM - hsbt (Hiroshi SHIBATA)

- Related to Bug #10856: Splat with empty keyword args gives unexpected results added

#5 - 12/14/2017 07:24 AM - hsbt (Hiroshi SHIBATA)

*- Related to Bug #11236: inconsistent behavior using ** vs hash as method parameter added*

#6 - 12/14/2017 07:24 AM - hsbt (Hiroshi SHIBATA)

- Related to Bug #11967: Mixing kwargs with optional parameters changes way method parameters are parsed added

#7 - 12/14/2017 07:24 AM - hsbt (Hiroshi SHIBATA)

- Related to Bug #12104: Procs keyword arguments affect value of previous argument added

#8 - 12/14/2017 07:24 AM - hsbt (Hiroshi SHIBATA)

- Related to Bug #12717: Optional argument treated as kwarg added

#9 - 12/14/2017 07:24 AM - hsbt (Hiroshi SHIBATA)

- Related to Bug #12821: Object converted to Hash unexpectedly under certain method call added

#10 - 12/14/2017 07:25 AM - hsbt (Hiroshi SHIBATA)

- Related to Bug #13336: Default Parameters don't work added

#11 - 12/14/2017 07:25 AM - hsbt (Hiroshi SHIBATA)

- Related to Bug #13647: Some weird behaviour with keyword arguments added

#12 - 12/14/2017 07:25 AM - hsbt (Hiroshi SHIBATA)

- Related to Bug #14130: Keyword arguments are ripped from the middle of hash if argument have default value added

#13 - 12/14/2017 08:27 AM - jeremyevans0 (Jeremy Evans)

For a method definition like:

```
def foo(hsh={})
end
```

Will either of the following continue to work?:

```
foo(key: val)
foo(:key => val)
```

One performance issue with keyword arguments is that keyword splats allocate a hash per splat, even if no keywords are used.

In performance sensitive code, allocations can be avoided using a shared frozen hash as the default argument:

```
OPTS = {}.freeze
def foo(hsh=OPTS)
  bar(1, hsh)
end
def bar(val, hsh=OPTS)
end
```

By doing this, calling foo without keyword arguments does not allocate any hashes even if the hash is passed to other methods. If you use keyword arguments, you have to do:

```
def foo(**hsh)
  bar(1, **hsh)
end
def bar(val, **hsh)
end
```

Which I believe allocates a multiple new hashes per method call, one in the caller and one in the callee. Example:

```
require 'objspace'
GC.start
GC.disable
OPTS = {}

def hashes
  start = ObjectSpace.count_objects[:T_HASH]
  yield
  ObjectSpace.count_objects[:T_HASH] - start - 1
end

def foo(opts=OPTS)
  bar(opts)
end
```

```
def bar(opts=OPTS)
  baz(opts)
end
def baz(opts=OPTS)
end

def koo(**opts)
  kar(**opts)
end
def kar(**opts)
  kaz(**opts)
end
def kaz(**opts)
end

p hashes{foo}
p hashes{foo(OPTS)}
p hashes{koo}
p hashes{koo(**OPTS)}

# Output
0
0
5
6
```

I humbly request that unless keyword splats can be made to avoid allocation, then at least make:

```
def foo(hsh)
end
foo(:key => val)
```

still function as it has since ruby 1.8, since that can be considered a hash and not a keyword argument.

#14 - 12/18/2017 01:42 PM - mame (Yusuke Endoh)

- Backport deleted (2.3: UNKNOWN, 2.4: UNKNOWN)

- Tracker changed from Bug to Feature

#15 - 01/16/2018 05:33 PM - sos4nt (Stefan Schüßler)

I've filed a bug report some time ago, maybe you could add it as a related issue: https://bugs.ruby-lang.org/issues/11993

#16 - 01/19/2018 03:10 AM - dsferreira (Daniel Ferreira)

It's not clear for me all the implications of this change.
Would it be possible to exemplify the before and after behaviours in the description?

It feels to me that with this implementation it would be possible to consider both symbols and strings as keys for the keywords hash.

Would it be a possibility?

The dynamic generation of keywords hashes would be positively impacted with that move.

#17 - 07/23/2018 01:06 AM - mame (Yusuke Endoh)

- Description updated

#18 - 07/23/2018 01:06 AM - mame (Yusuke Endoh)

jeremyevans0 (Jeremy Evans) wrote:

> For a method definition like:
>
> ```
> def foo(hsh={})
> end
> ```
>
> Will either of the following continue to work?:
>
> ```
> foo(key: val)
> foo(:key => val)
> ```

No, it will not work. You need to rewrite the definition to def foo(**hsh).

> One performance issue with keyword arguments is that keyword splats allocate a hash per splat, even if no keywords are used.

If the issue really matters, it can be fixed by lazy Hash allocation, like block parameters (#14045).

dsferreira (Daniel Ferreira) wrote:

> It's not clear for me all the implications of this change.
> Would it be possible to exemplify the before and after behaviours in the description?

Added.

> It feels to me that with this implementation it would be possible to consider both symbols and strings as keys for the keywords hash.

It is a completely different topic, and I'm strongly negative against allowing strings as a key.

#19 - 07/23/2018 01:20 AM - mame (Yusuke Endoh)

Sorry, it seems my original description was unclear. I think it can be rephased very simply:

- keyword argument MUST be always received as a keyword parameter
- non-keyword argument MUST be always received as a non-keyword parameter

The following behavior will be abandoned:

- keyword argument is passed to a last normal parameter
- last normal hash argument is passed to keyword parameters

Here is an experimental patch to warn a deprecated behavior of keyword arguments, and it shows some OK/NG samples.

NG: a keyword argument is passed to a normal parameter

```
$ ./miniruby -w -e '
def foo(h)
end
foo(k: 1)
'
-e:4: warning: The keyword argument for `foo' is used as the last parameter
```

OK: receving it as a keyword rest argument

```
$ ./miniruby -w -e '
def foo(**h)
end
foo(k: 1)
'
```

NG: a normal hash argument is passed to a keyword argument

```
$ ./miniruby -w -e '
def foo(k: 1)
end
h = {k: 42}
foo(h)
'
-e:5: warning: The last argument for `foo' is used as the keyword parameter
```

OK: the hash as keyword argument by using **

```
$ ./miniruby -w -e '
def foo(k: 1)
end
h = {k: 42}
foo(**h)
'
```

It still needs more work. It does not support yet methods written in C because C methods always handles keyword arguments as normal arguments.

#20 - 07/23/2018 01:49 AM - jeremyevans0 (Jeremy Evans)

mame (Yusuke Endoh) wrote:

> jeremyevans0 (Jeremy Evans) wrote:
>
> > For a method definition like:
> >
> > ```
> > def foo(hsh={})
> > end
> > ```
> >
> > Will either of the following continue to work?:
> >
> > ```
> > foo(key: val)
> > foo(:key => val)
> > ```
>
> No, it will not work. You need to rewrite the definition to def foo(**hsh).

If that was the only change, it wouldn't be a big deal. However, in addition to foo(:key => val) calls, there are also foo(hsh) calls. So all callers that pass hashes would need to change from foo(hsh) to foo(**hsh). And that also breaks if there are any non-symbol keys in the hash.

In the libraries I maintain, this will be a bigger breaking change than 1.8 -> 1.9. If the decision has already been made and there is no turning back, there should probably be deprecation warnings added for it in 2.6, anytime keywords are passed to a method that accepts a default argument, or anytime a hash is passed when keyword arguments should be used.

> > One performance issue with keyword arguments is that keyword splats allocate a hash per splat, even if no keywords are used.
>
> If the issue really matters, it can be fixed by lazy Hash allocation, like block parameters (#14045).

This does really matter, excessive hash allocation has a significant negative effect on performance. In addition to all of the code churn in libraries required to support this change, users of the libraries will also have to accept a significant performance hit until there is an allocation-less way to pass keyword arguments from one methods to another.

> The following behavior will be abandoned:
>
> - keyword argument is passed to a last normal parameter
> - last normal hash argument is passed to keyword parameters

Is it possible to abandon one of these without the other? Abandoning "last normal hash argument is passed to keyword parameters" only breaks code that uses keyword arguments. Abandoning "keyword argument is passed to a last normal parameter" (supported at least back to Ruby 1.8) breaks tons of ruby code that never used keyword arguments, just to supposedly fix problems that were caused by keyword arguments.

If keyword arguments are not part of the method definition, then what is the issue with converting keyword arguments to a hash argument?

> It still needs more work. It does not support yet methods written in C because C methods always handles keyword arguments as normal arguments.

What will happen to external C extension gems that use rb_get_kwargs and rb_extract_keywords, both of which accept a hash?

#21 - 07/23/2018 03:10 AM - mame (Yusuke Endoh)

Jeremy, thank you for discussing this issue seriously.

jeremyevans0 (Jeremy Evans) wrote:

> If that was the only change, it wouldn't be a big deal. However, in addition to foo(:key => val) calls, there are also foo(hsh) calls. So all callers that pass hashes would need to change from foo(hsh) to foo(**hsh). And that also breaks if there are any non-symbol keys in the hash.

Yes, in the current proposal, you need to rewrite all callers that passes a hash object. Note that you can already write foo(**hsh) in caller side since 2.0 (when callee-side keyword argument was introduced). Also, I believe it is a good style because the explicit operator clarifies the intent.

I have no strong opinion whether foo(:kw => 1) should pass a normal hash argument or be interpreted as keyword argument. I think the latter is better in terms of compatibility, but I'm not sure.

> If keyword arguments are not part of the method definition, then what is the issue with converting keyword arguments to a hash argument?

I have never thought of this. I want to reject the following program,

```
def foo(*ary)
end
foo(kw: 1)
```

but it might be a good idea as a measure for compatibility.

> It still needs more work. It does not support yet methods written in C because C methods always handles keyword arguments as normal arguments.

> What will happen to external C extension gems that use rb_get_kwargs and rb_extract_keywords, both of which accept a hash?

Yes, we need to prepare C API. Ko1 has had a big plan about this since last year (or older).

#22 - 07/24/2018 11:04 PM - jeremyevans0 (Jeremy Evans)

Here's an alternative proposal, with the basic idea that behavior for historical ruby 1.6+ code that doesn't use keyword arguments remains the same.

OK: Historical ruby 1.6+ (maybe before) usage (hash argument with omitted braces)

```
def foo(h)
  # h # => {:k => 1}
end
foo(:k => 1)
foo(k: 1) # ruby 1.9+ syntax
```

OK: Ruby 2.0 keyword usage that will keep working

```
def foo(k: 1) # or foo(**h)
end
foo(:k => 1)
foo(k: 1)
foo(**{k: 1})
```

NG: Using ** splat as hash argument

```
def foo(h)
end
foo(**{k: 1})
```

NG: Using hash argument instead of keyword arguments

```
def foo(k: 1) # or foo(**h)
end
foo({k: 1})
```

My reasoning for this is that historical behavior for methods that do not use keyword arguments should not be broken to fix problems caused by keyword arguments. I reviewed all issues mentioned in this ticket:

#8040: method keyword arguments
#8316: method keyword arguments
#9898: method regular argument, caller uses **
#10856: method regular argument, caller uses ** on empty array
#11236: method keyword arguments
#11967: method keyword arguments
#12104: proc usage, unrelated to keyword argument vs regular argument
#12717: method keyword arguments
#12821: method keyword arguments
#13336: method keyword arguments
#13467: method keyword arguments
#14130: method keyword arguments

As you can see, all of the problems are with using keyword arguments in the method definition or with ** at the call site when a method regular argument is used. There are no issues when the method takes a regular argument and ** is not used at the call site, with the historical behavior and syntax of specifying a hash argument with omitted braces. I see no reason to break the ruby 1.6+ historical behavior when keyword arguments are not involved.

Regarding the following program mentioned by mame:

```
def foo(*ary)
end
foo(kw: 1)
```

there is a lot of historical ruby code that does:

```
def foo(*ary)
```

```
  options = ary.pop if ary.last.is_a?(Hash)
  # ...
end
```

For that reason I think it would be best if foo(kw: 1) continued to work in such cases, since there are no problems in terms of the keyword arguments being used (no keyword arguments in method definition implies argument syntax is a hash with omitted braces).

#23 - 07/26/2018 10:38 AM - shevegen (Robert A. Heiler)

I don't want to write too much, so just one comment - I would also prefer foo(kw: 1)
to retain being a Hash rather than to be assumed to be a keyword argument. I think
that it may surprise people when it would become a keyword suddenly.

#24 - 07/26/2018 03:32 PM - matz (Yukihiro Matsumoto)

shevegen (Robert A. Heiler) Of course, we will take plenty of time to migrate before making it a keyword.
If we made the decision, we will make it warn you first for a year or two before the actual change.

Matz.

#25 - 08/30/2018 11:10 PM - jeremyevans0 (Jeremy Evans)

To give an example of how much code this would break, let's use Redmine as an example, since it runs this bug tracker. For simplicity, let's limit our analysis to the use of a single method, ActiveRecord's where method. ActiveRecord's where method uses the following API (note, no keyword arguments):

```
def where(opts = :chain, *rest)
  # ...
end
```

where is used at least 597 times in 180 files in the application, and most of these cases appear to be calls to the ActiveRecord where method. In at least 399 cases, it appears to use an inline hash argument without braces (there are additional cases where ruby 1.9 hash syntax is used), and in 11 cases it uses an inline hash argument with braces:

```
$ fgrep -r .where\( !(public|doc|extra) |wc -l
     597
$ fgrep -lr .where\( !(public|doc|extra) |wc -l
     180
$ fgrep -r .where\( !(public|doc|extra) | fgrep '=>' | fgrep 'where(:' |wc -l
     399
$ fgrep -r .where\( !(public|doc|extra) | fgrep 'where({' |wc -l
      11
```

Examples of where usage:

```
# Inline hash without braces
@time_entries = TimeEntry.where(:id => params[:ids]).

# Inline hash with braces
Enumeration.where({:type => type}).update_all({:is_default => false})

# Noninline hash
condition_hash = self.class.positioned_options[:scope].inject({}) do |h, column|
  h[column] = yield(column)
  h
end
self.class.where(condition_hash)
```

Hopefully this serves an example of how much code this would break. Remember, this is only looking at a single method. Note that omitting the braces for hashes is almost 40x more common than including the braces.

Dropping support for braceless hashes would probably break the majority of ruby applications and libraries. Consider this another plea to limit behavior changes to methods that accept keyword arguments.

#26 - 08/31/2018 01:56 AM - mame (Yusuke Endoh)

Jeremy, thank you for investigating the examples. I'd like to discuss this issue at the next developers' meeting.

This is my personal current opinion: this change indeed requires users' action, however, I believe that the problem is not so significant, and that its advantage is significant.

This change seems to remind you the breaking change of character encoding in 1.9/2.0, but it was much worse than this change because the previous one was not trivial "where to fix". The site where an error occurred was often different to the site where a wrong encoding string was created.
On the other hand, this change requires very trivial fixes. By running a test suite on Ruby 2.6 or 2.7, the interpreter will "pinpoint" all usages like you

showed, and warn "this method call in line XX will not work in Ruby 3.x!". Users can easily fix the issue by checking the warnings and changing either the method calls or method definition.

I agree that compatibility is important, but the current wrong design has continuously caused troubles. This fact also looks important to me. This change will fix the issue, will make the language simpler, and will make users' code more explicit and less error-prone, which will pay users' action.

#27 - 08/31/2018 05:20 AM - jeremyevans0 (Jeremy Evans)

mame (Yusuke Endoh) wrote:

> Jeremy, thank you for investigating the examples. I'd like to discuss this issue at the next developers' meeting.
>
> This is my personal current opinion: this change indeed requires users' action, however, I believe that the problem is not so significant, and that its advantage is significant.
>
> This change seems to remind you the breaking change of character encoding in 1.9/2.0, but it was much worse than this change because the previous one was not trivial "where to fix". The site where an error occurred was often different to the site where a wrong encoding string was created.

I disagree. I migrated many applications and libraries from Ruby 1.8 to Ruby 1.9 (and later to Ruby 2.6). The changes for Ruby 1.8 -> 1.9 were minimal in comparison with the impact of this change, in terms of the amount of code that needed to be modified.

> On the other hand, this change requires very trivial fixes. By running a test suite on Ruby 2.6 or 2.7, the interpreter will "pinpoint" all usages like you showed, and warn "this method call in line XX will not work in Ruby 3.x!". Users can easily fix the issue by checking the warnings and changing either the method calls or method definition.
>
> I agree that compatibility is important, but the current wrong design has continuously caused troubles. This fact also looks important to me. This change will fix the issue, will make the language simpler, and will make users' code more explicit and less error-prone, which will pay users' action.

As I've already shown earlier in this issue, all problems in issues referenced in your initial post boil down to two basic cases:

1) Where the method being called accepts keyword arguments
2) Where double splat (**) is used by the caller and the method does not accept keyword arguments

No problems have been posted where the method does not accept keyword arguments and braces are just omitted when calling the method with an inline hash. That code has not continuously caused problems, it has worked fine since at least Ruby 1.6 with basically no changes.

The keyword argument problems started occurring in Ruby 2.0 when keyword arguments were introduced, and only affected people who chose to use define methods that accepted keyword arguments or use the double splat. If you never used double splats and never defined methods that accepted keyword arguments, either to avoid the usability and performance problems with keyword arguments or to retain compatibility with ruby <2.0, then you never ran into any of these problems.

This change makes sense for methods that accept keyword arguments, and for double splat usage on hashes when the method does not accept keyword arguments. I agree that those cases are problematic and we should fix those cases in Ruby 3. I'm just requesting that the changes be limited to those cases, and not break cases where keyword arguments and double splats were never used, since those cases have never been problematic.

Ruby is a beautiful language designed for programmer happiness. Having to change all calls from where(:id=>1) to where({:id=>1}) makes the code uglier and is going to make most Ruby programmers less happy. Does this argument for explicitness lead to requiring parentheses for all method calls?

#28 - 08/31/2018 05:45 AM - mame (Yusuke Endoh)

jeremyevans0 (Jeremy Evans) wrote:

> Having to change all calls from where(:id=>1) to where({:id=>1}) makes the code uglier and is going to make most Ruby programmers less happy. Does this argument for explicitness lead to requiring parentheses for all method calls?

In this specific case, it looks better to change the callee side instead of the caller side: the method definition of where should receive a keyword rest argument. Of course, it still requires us change some calls of where(opt_hash) to where(**opt_hash), but I think it is better and clearer.

#29 - 08/31/2018 08:05 AM - mame (Yusuke Endoh)

Here is a scenario where allowing "hash argument with omitted braces" causes a problem. Assume that we write a method "debug" which is equal to "Kernel#p".

```
def debug(*args)
  args.each {|arg| puts arg.inspect }
end
```

Passing a hash argument with omitted braces, unfortunately, works.

```
debug(key: 42) #=> {:key=>42}
```

Then, consider we improve the method to accept the output IO as a keyword parameter "output":

```
def debug(*args, output: $stdout)
  args.each {|arg| output.puts arg.inspect }
end
```

However, this change breaks the existing call.

```
debug(key: 42) #=> ArgumentError (unknown keyword: key)
```

This is too easy to break. So, what is bad? I believe that passing a hash argument as a normal last parameter is bad.

I'd like to make it safe to extend an existing method definition with a keyword parameter.

#30 - 08/31/2018 02:42 PM - jeremyevans0 (Jeremy Evans)

mame (Yusuke Endoh) wrote:

> jeremyevans0 (Jeremy Evans) wrote:
>
> > Having to change all calls from where(:id=>1) to where({:id=>1}) makes the code uglier and is going to make most Ruby programmers less happy. Does this argument for explicitness lead to requiring parentheses for all method calls?
>
> In this specific case, it looks better to change the callee side instead of the caller side: the method definition of where should receive a keyword rest argument. Of course, it still requires us change some calls of where(opt_hash) to where(**opt_hash), but I think it is better and clearer.

Changing the callee side will not fix all cases. The where method supports more than just symbols keys in hashes. where('table.id'=>1) is supported, for example. Accepting a keyword args splat and then appending it to the array of arguments just decreases performance for no benefit.

It is important to realize that keyword arguments are not a substitute for hash arguments, as keyword arguments only handle a subset of what a hash argument can handle.

mame (Yusuke Endoh) wrote:

> Here is a scenario where allowing "hash argument with omitted braces" causes a problem. Assume that we write a method "debug" which is equal to "Kernel#p".
>
> ```
> def debug(*args)
>   args.each {|arg| puts arg.inspect }
> end
> ```
>
> Passing a hash argument with omitted braces, unfortunately, works.
>
> ```
> debug(key: 42) #=> {:key=>42}
> ```
>
> Then, consider we improve the method to accept the output IO as a keyword parameter "output":
>
> ```
> def debug(*args, output: $stdout)
>   args.each {|arg| output.puts arg.inspect }
> end
> ```
>
> However, this change breaks the existing call.

Note how this problem does not occur until you add keyword arguments to the method. If you never add keyword arguments, you never run into this problem, and there are ways to add the support you want without using keyword arguments.

Are you assuming that all methods that use hash arguments will end up wanting to use keyword arguments at some point? I think that is unlikely. If keyword arguments are never added to the method in the future, then you have broken backwards compatibility now for no benefit.

You are implying it is better to certainly break tons of existing code now, to allow for a decreased possibility of breaking code later if and only if you decide to add keyword arguments.

> This is too easy to break. So, what is bad? I believe that passing a hash argument as a normal last parameter is bad.

That is an opinion I do not share. I believe passing a hash argument as a normal last parameter is fine and one of the nice features that makes Ruby a beautiful language to write in. I think omitting braces for hash arguments has a natural similarity to the ability to omit parentheses for method calls, which is another Ruby feature that makes it enjoyable to write in.

> I'd like to make it safe to extend an existing method definition with a keyword parameter.

Attempting to avoid backwards compatibility problems is a noble goal that I think we share. Part of that is avoiding future backwards compatibility problems. Another part of that is avoiding current backwards compatibility problems. A change that causes more current backwards compatibility problems than the future backwards compatibility problems it is designed to avoid is a step in the wrong direction, in my opinion.

#31 - 09/01/2018 01:32 AM - mame (Yusuke Endoh)

jeremyevans0 (Jeremy Evans) wrote:

> Changing the callee side will not fix all cases. The where method supports more than just symbols keys in hashes. where('table.id'=>1) is supported, for example. Accepting a keyword args splat and then appending it to the array of arguments just decreases performance for no benefit.

As an experiment, I'm now trying to check Ruby's existing APIs, and noticed that some methods had the issue: Kernel#spawn, JSON::GenericObject.from_hash, etc. It might be good to provide a variant of define_method for this case as a migration path:

```
define_last_hash_method(:foo) do |opt|
  p opt
end

foo(k: 1)      #=> {:k=>1}
foo("k"=>1) #=> {"k"=>1}
```

> Are you assuming that all methods that use hash arguments will end up wanting to use keyword arguments at some point? I think that is unlikely. If keyword arguments are never added to the method in the future, then you have broken backwards compatibility now for no benefit.

I don't think that all methods will have keyword arguemnts eventually. However, I assume that we can never predict which methods will have.

> > I'd like to make it safe to extend an existing method definition with a keyword parameter.
>
> Attempting to avoid backwards compatibility problems is a noble goal that I think we share. Part of that is avoiding future backwards compatibility problems. Another part of that is avoiding current backwards compatibility problems. A change that causes more current backwards compatibility problems than the future backwards compatibility problems it is designed to avoid is a step in the wrong direction, in my opinion.

In general, I agree. For this specific topic, however, the current spec and implementation are really a mess; the current backward compatibility problem is relatively easy to fix; the future backwards compatibility problem is hard to avoid and will become painful more and more. We should now pay the debt for the future.

But this is just my opinion. I really appreciate and respect your opinion. I'd like to tell matz your opinion as fairly as I can.

#32 - 09/03/2018 02:35 AM - duerst (Martin Dürst)

jeremyevans0 (Jeremy Evans) wrote:

> mame (Yusuke Endoh) wrote:
>
> > This change seems to remind you the breaking change of character encoding in 1.9/2.0, but it was much worse than this change because the previous one was not trivial "where to fix". The site where an error occurred was often different to the site where a wrong encoding string was created.
>
> I disagree. I migrated many applications and libraries from Ruby 1.8 to Ruby 1.9 (and later to Ruby 2.6). The changes for Ruby 1.8 -> 1.9 were minimal in comparison with the impact of this change, in terms of the amount of code that needed to be modified.

I think the amount of changes from Ruby 1.8 to Ruby 1.9 depended a lot on what kind of processing your application did, and what kind of data was involved. If you mostly just worked with US-ASCII data, the changes needed were minimal. For other data, in particular also for Japanese data, some kinds of processing may have been heavily affected.

#33 - 09/04/2018 10:46 PM - Eregon (Benoit Daloze)

I agree with Jeremy here, the current idea seems too incompatible and will require too many changes (no matter the gain).
And those changes cannot easily be automated either, they need careful considerations.

I think we need to compromise here, to avoid too many incompatible changes, especially on methods which have no keyword arguments and where the intention is clear.
I would think the number of methods like debug() is a tiny fraction of the number of places we'd need to change if hash-without-braces is no longer supported.

IMHO such a method with rest + kwargs seems a bad design in the first place as the arguments are too complex. That debug method could only

accept one argument for instance.

Also, how should foo(1, "foo" => "bar") behave?
Should it be like foo(1, {"foo" => "bar"})? In this case the syntax is inconsistent with foo(1, foo: "bar") where having or leaving out the braces matter.
Or does the => imply the braces?
I believe all Rubyists are used to foo(1, :foo => "bar") and foo(1, foo: "bar") being identical.

BTW, p foo: 1 will no longer work then, and p({foo: 1}) would be required, which feels very *unlike* Ruby, and is just impractical when debugging.

#34 - 09/05/2018 03:01 AM - mame (Yusuke Endoh)

Eregon (Benoit Daloze) wrote:

> I would think the number of methods like debug() is a tiny fraction of the number of places we'd need to change if hash-without-braces is no longer supported.
>
> IMHO such a method with rest + kwargs seems a bad design in the first place as the arguments are too complex. That debug method could only accept one argument for instance.

I think you are too familiar with the current weird keyword arguments. The original and primary purpose of keyword arguments is an extension of existing methods. It looks rather "too complex" for a mere addition of keyword parameters to disturb other parameters and to break existing calls.

That being said, I agree that the "cancer" of this issue is a combination of rest/optional agruments and keyword ones. Another, more modest idea that I have is, to prohibit (or just warn) a method definition that has both rest/optional + keyword parameters. I don't like this because this spoils the purpose of keyword arguments, though.

> Also, how should foo(1, "foo" => "bar") behave?
> Should it be like foo(1, {"foo" => "bar"})?

I think so.

> In this case the syntax is inconsistent with foo(1, foo: "bar") where having or leaving out the braces matter.

Braced hash and bare one are inconsistent, even in the current spec.

```
def foo(v=:default)
  p v
end

h={}
foo( **h )     #=> {}
foo({**h})     #=> {}
foo(1,  **h ) #=> 1
foo(1, {**h}) #=> wrong number of arguments (given 2, expected 0..1)
```

Note that **{} does not simply mean "no argument". If it was "no argument", the above foo(**h) would print :default instead of {}.

> Or does the => imply the braces?
> I believe all Rubyists are used to foo(1, :foo => "bar") and foo(1, foo: "bar") being identical.

They will be still identical because it is determined not only syntactically but also dynamically: a key-value pair whose key is a Symbol, is handled as keyword argument. This behavior is not new. Ruby 2.5 even does it:

```
def foo(h1=nil, **h2)
  p [h1, h2]
end
foo("foo" => 1, :bar => 2, baz: 3) #=> [{"foo"=>1}, {:bar=>2, :baz=>3}]
```

(This behavior has been changed in trunk, but I'm unsure if it is determined or not.)

> BTW, p foo: 1 will no longer work then, and p({foo: 1}) would be required, which feels very *unlike* Ruby, and is just impractical when debugging.

I completely agree with this. I showed the method debug as an example, but I don't think that Kernel#p itself should change. Some existing APIs that people expect to accept both foo(k:1) and foo({k:1}), e.g, ERB#result_with_hash, Sequel's where, should be kept as well.

#35 - 09/05/2018 06:47 AM - mame (Yusuke Endoh)

mame (Yusuke Endoh) wrote:

> > BTW, p foo: 1 will no longer work then, and p({foo: 1}) would be required, which feels very *unlike* Ruby, and is just impractical when

> debugging.

I completely agree with this. I showed the method debug as an example, but I don't think that Kernel#p itself should change. Some existing APIs that people expect to accept both foo(k:1) and foo({k:1}), e.g, ERB#result_with_hash, Sequel's where, should be kept as well.

Half-joking: I'm not fully satisfied with p foo: 1 #=> {:foo=>1}. If a keyword argument is separated from other ones, it can emit a much better output:

```
def p(*args, **kw)
  args.each {|arg| puts arg.inspect }
  kw.each {|label, arg| puts "#{ label }: #{ arg.inspect }" }
end

p foo: 1, bar: {"A"=>"B"}, baz: {qux: 1}
#=> foo: 1
#   bar: {"A"=>"B"}
#   baz: {:qux=>1}
```

#36 - 09/05/2018 03:33 PM - marcandre (Marc-Andre Lafortune)

- Related to deleted (Bug #12104: Procs keyword arguments affect value of previous argument)

#37 - 09/05/2018 04:59 PM - marcandre (Marc-Andre Lafortune)

mame (Yusuke Endoh) wrote:

> Braced hash and bare one are inconsistent, even in the current spec.
>
> ```
> def foo(v=:default)
>   p v
> end
>
> h={}
> foo( **h )     #=> {}
> foo({**h})     #=> {}
> foo(1,  **h ) #=> 1
> foo(1, {**h}) #=> wrong number of arguments (given 2, expected 0..1)
> ```

The fact that foo(**h) #=> {} is a bug. Note that foo(**{}) # => :default, as I believe it should. Both should have same result. See #15078.

```
def foo(h1=nil, **h2)
  p [h1, h2]
end
foo("foo" => 1, :bar => 2, baz: 3) #=> [{"foo"=>1}, {:bar=>2, :baz=>3}]
```

I believe this is currently a bug (#14130) and I hope this is not accepted in the future either. Is there a good use case for this anyways? I fear it only creates hard to find errors.

#38 - 09/05/2018 05:05 PM - marcandre (Marc-Andre Lafortune)

Let me add my voice to that of Benoit and Jeremy: the incompatibility is absolutely not worth it.

I believe that if we fix the few remaining corner cases, improve the error messages and explicitly document how Ruby handles keyword parameters vs optional positional parameters, we'll have a really solid solution.

#39 - 09/05/2018 11:03 PM - mame (Yusuke Endoh)

- Related to Bug #15078: Hash splat of empty hash should not create a positional argument. added

#40 - 09/09/2018 03:33 AM - marcandre (Marc-Andre Lafortune)

After working a lot on **{}, I still strongly believe that we must maintain conversion of keyword arguments to positional argument, e.g.:

```
def foo(*ary)
end
foo(kw: 1) # => must remain ok
```

OTOH, it may be possible to disallow promotion of last positional argument to keyword arguments without causing as huge incompatiblities. Using **hash could be required, if given enough time (say warnings in Ruby 2.6 & 2.7)

```
def foo(**options); end
foo(hash) # => Could be disallowed, only foo(**hash) would work
```

A major consequence of disallowing promotion to keyword arguments is that the naive forwarding calls (only with *args) will no longer be always valid. This means that all forwarding calls, including those of delegate library, will have to become capture arguments with *args, **options. This means that the meaning of **{} will become more important than it currently is.

As I argue in #15078, it will be important that **{} doesn't create a positional argument so that full forwarding works even for normal methods.

My recommendation:

Ruby 2.6: Fix **{} to not create positional argument (#15078). Improve wording of ArgumentErrors

If we want to have stricter keyword arguments (I'm not sure it's worth it), then:

Ruby 2.6: In verbose mode, warn about promotion of positional argument to keyword arguments, recommending using hash splat.
Ruby 2.7: Same, even if not-verbose.
Ruby 3.0: Stop promoting normal argument to keyword argument.

.... after I'm long dead

Ruby 42.0: Stop demoting keyword argument to normal argument

#41 - 09/17/2018 05:22 AM - akr (Akira Tanaka)

I have an idea to separate positional arguments and keyword arguments without incompatibility.

Basic idea is introducing an flag, keyword_given,
which means the last argument is a Hash object which represent keyword argument.
(The name, keyword_given, is inspired from block_given? method.)

The flag will be true if method call uses k => v, k: v, "k": v or **h and
all keys of the Hash object constructed from them are symbol.
(I think hash separation is not good idea.)

The flag is referenced by a new Ruby method (keyword_given?) and
C level function (rb_keyword_given_p).

This doesn't break anything because it just add new method (and new C function).

This makes the confusion of positional/keyword arguments solvable.
But I don't say the confusion is solvable easily (or by-default).
Programmers must use the flag carefully.

If we want to solve the confusion by default, we need to change
method invocation behavior incompatible way.
However positional/keyword separation by the flag makes possible to
change behavior incrementally.

If a method is changed to use keyword_given?,
only the method is changed.
We can discuss the situation about the actual method.

If Ruby-level method definition/invocation behavior is changed
(def m(h) end cannot receive m(:k=>0) for example),
it affects many applications.
However method definition/invocation behavior contains
several points which can refer the flag.
We can discuss how big/small incompatibility and
how big/small benefits for each one.

#42 - 09/17/2018 02:56 PM - jeremyevans0 (Jeremy Evans)

akr (Akira Tanaka) wrote:

> I have an idea to separate positional arguments and keyword arguments without incompatibility.

I like this idea of allowing per-method handling of arguments. Just to confirm my understanding of the proposal:

```
def m(*a)
  [keyword_given?, a]
end

m # => [false, []]
m(1) # => [false, [1]]
m({:a=>1}) # => [false, [{:a=>1}]]
```

```
m(:a=>1) # => [true, [{:a=>1}]]
m(a: 1) # => [true, [{:a=>1}]]
m("a": 1) # => [true, [{:a=>1}]]
m(**{a: 1}) # => [true, [{:a=>1}]] or ArgumentError ?
m(**{}) # => [true, [{}]], [true, []], or ArgumentError ?
m('a'=>1, :a=>1) # => [false, [{'a'=>1, :a=>1}]]
a = :a
m(a=>1) # => [true, [{:a=>1}]]

def m2(*a, **kw)
  [keyword_given?, a, kw]
end

m2 # => [false, [], {}]
m2(1) # => [false, [1], {}]
m2({:a=>1}) # => [false, [{:a=>1}], {}]
m2(:a=>1) # => [true, [], {:a=>1}]
m2(a: 1) # => [true, [], {:a=>1}]
m2("a": 1) # => [true, [], {:a=>1}]
m2(**{a: 1}) # => [true, [], {:a=>1}]
m2(**{}) # => [true, [], {}]
m2('a'=>1, :a=>1) # => [false, [{'a'=>1, :a=>1}], {}]
a = :a
m2(a=>1) # => [true, [], {:a=>1}]

def m3(a)
  [keyword_given?, a]
end

m3 # => ArgumentError
m3(1) # => [false, 1]
m3({:a=>1}) # => [false, {:a=>1}]
m3(:a=>1) # => [true, {:a=>1}]
m3(a: 1) # => [true, {:a=>1}]
m3("a": 1) # => [true, {:a=>1}]
m3(**{a: 1}) # => [true, {:a=>1}] or ArgumentError ?
m3(**{}) # => [true, {}] or ArgumentError ?
m3('a'=>1, :a=>1) # => [false, {'a'=>1, :a=>1}]
a = :a
m3(a=>1) # => [true, {:a=>1}]

def m4(**kw)
  [keyword_given?, kw]
end

m4 # => [true, {}] or [false, {}] ?
m4(1) # => ArgumentError
m4({:a=>1}) # => ArgumentError
m4(:a=>1) # => [true, {:a=>1}]
m4(a: 1) # => [true, {:a=>1}]
m4("a": 1) # => [true, {:a=>1}]
m4(**{a: 1}) # => [true, {:a=>1}]
m4(**{}) # => [true, {}] or [false, {}] ?
m4('a'=>1, :a=>1) # => ArgumentError
a = :a
m4(a=>1) # => [true, {:a=>1}]
```

If the method does not explicitly declare any keyword arguments, and the caller uses **hash is used to explicitly pass a keyword argument, does that raise an ArgumentError or does it pass the hash as a positional argument and have keyword_given? return true?

If the method does not explicitly declare any keyword arguments, does it pass the splatted empty hash as a positional argument, does it ignore it, or does it raise an ArgumentError?

If the method explicitly declares keyword arguments (either required keyword, optional keyword, or keyword splat), and is called without keyword arguments, does keyword_given? return true or false?

If the method explicitly declares keyword arguments and an empty hash is splatted, does keyword_given? return true or false?

#43 - 09/18/2018 12:00 AM - akr (Akira Tanaka)

keyword_given? provides information about the caller side.
The information is not related to callee side.
There is no chance to call keyword_given? if ArgumentError is raised, though.

For simplicity, I assume keys for keyword argument is Symbol, here.

(If non-Symbol key is provided, keyword_given? returns false.)

jeremyevans0 (Jeremy Evans) wrote:

> If the method does not explicitly declare any keyword arguments, and the caller uses **hash is used to explicitly pass a keyword argument, does that raise an ArgumentError or does it pass the hash as a positional argument and have keyword_given? return true?

keyword_given? return true.
We can discuss ArgumentError or not.

> If the method does not explicitly declare any keyword arguments, does it pass the splatted empty hash as a positional argument, does it ignore it, or does it raise an ArgumentError?

If splatted empty hash means **{} in caller side, keyword_given? return true.
We can discuss ArgumentError or not.

> If the method explicitly declares keyword arguments (either required keyword, optional keyword, or keyword splat), and is called without keyword arguments, does keyword_given? return true or false?

keyword_given? return false.
I think there is no chance for keyword argument related ArgumentError if no required keyword.

> If the method explicitly declares keyword arguments and an empty hash is splatted, does keyword_given? return true or false?

keyword_given? return true.

I assume **{} in caller add {} in arguments and keyword_given? return true.
However, another behavior is possible: **{} doesn't add {} in arguments and keyword_given? return false.
Their difference is visible until we have a way to obtain positional arguments and keyword argument in single array.
I choose former because I'm considering to distinguish them using **nil and `**{}'.
See details with https://bugs.ruby-lang.org/issues/15078#note-13

#44 - 09/18/2018 03:44 AM - marcandre (Marc-Andre Lafortune)

Very interesting.

akr (Akira Tanaka) wrote:

> The flag will be true if method call uses k => v, k: v, "k": v or **h and
> all keys of the Hash object constructed from them are symbol.
> (I think hash separation is not good idea.)

I agree that hash separation is not a good idea. I'm wondering if (k = :a) => v should be accepted. It is the only case that is not syntactical.

> This makes the confusion of positional/keyword arguments solvable.
> But I don't say the confusion is solvable easily (or by-default).
> Programmers must use the flag carefully.

IIUC, the only 100% correct way to forward a method call (including the keyword_given? flag) would be:

```
def forward(*args, &block)
  if keyword_given?
    options = args.pop
    target(*args, **options, &block)
  else
    target(*args, &block)
  end
end
```

That is assuming the current **{} creating a positional argument.

Assuming Ruby 2.x compatiblity, there's no way of a general forward with **capture though. We'd need a method lash_argument_converted_to_keyword?...

```
def forward(*args, **options, &block)
  if keyword_given?
    target(*args, **options, &block)
  else
    args << options if last_argument_converted_to_keyword?
```

```
    target(*args, &block)
  end
end
```

Maybe the API could be combined in a single method keyword_style, returning one of [nil, :keyword, :hash_to_keyword, :keyword_to_hash]:

```
def foo(*); keyword_style; end
def bar(**); keyword_style; end
h = {}

foo      # => nil
foo(**h) # => :keyword_to_hash
bar(**h) # => :keyword
bar(h)   # => :hash_to_keyword
```

Or keyword_given? could return nil in case a hash was converted to a keyword argument like in bar(h)

If we distinguish **nil and **{} as in #15078, then there's no need to even call keyword_given? and normal forwarding works I imagine... But that might be quite incompatible.

#45 - 12/03/2018 02:31 AM - mame (Yusuke Endoh)

I talked with matz about this proposal at Keep Ruby Weird conference (more precisely, Franklin BBQ at Austin). As far as I understand, Matz currently likes syntactical separation of keyword and normal arguments. (Note that it is not decided yet.)

Short summary:

```
def foo(**kw); p kw; end
def bar(kw = {}); p kw; end
h = {:k => 1}

# base (non-braced) hash arguments passed as keywords
foo(k: 1)     #=> {:k=>1} in 2.X and 3.0
foo(:k => 1) #=> {:k=>1} in 2.X and 3.0
foo(**h)      #=> {:k=>1} in 2.X and 3.0
bar(k: 1)     #=> {:k=>1} in 2.X, ArgumentError in 3.0
bar(:k => 1) #=> {:k=>1} in 2.X, ArgumentError in 3.0
bar(**h)      #=> {:k=>1} in 2.X, ArgumentError in 3.0

# braced hash arguments are passed as a last argument
foo({ k: 1 })     #=> {:k=>1} in 2.X, ArgumentError in 3.0
foo({ :k => 1 }) #=> {:k=>1} in 2.X, ArgumentError in 3.0
foo(h)            #=> {:k=>1} in 2.X, ArgumentError in 3.0
bar({ k: 1 })     #=> {:k=>1} in 2.X and 3.0
bar({ :k => 1 }) #=> {:k=>1} in 2.X and 3.0
bar(h)            #=> {:k=>1} in 2.X and 3.0
```

Unfortunately, this change will break many existing programs. But, it would be still easy to fix. We can pick up keywords or normal hash explicitly for each callsite that an error occurred. In many cases, keywords would be preferable: just change from def foo(h = {}) to def foo(**h), and from foo(h) to foo(**h).

And, this is a new topic. There is an non-Symbol-key call, like where("table.id" => 1) (which was shown by Jeremy Evans). This is difficult to change to keyword argument. To allow this, matz came up with an idea: non-Symbol key is also allowed as a keyword.

```
def foo(**kw)
  p kw
end

foo("str" => 42) #=> {"str"=>42}
```

Note that, if you need to write a library that works on both 2.X and 3.X, you must write a shim:

```
def foo(kw1 = {}, **kw2)
  kw = kw1.merge(kw2)
  kw
end
```

However, after EOL of all Ruby 2.X series, you can remove the shim and just write a simple code. This is better than my original proposal.

What do you think?

#46 - 12/03/2018 03:40 AM - jeremyevans0 (Jeremy Evans)

mame (Yusuke Endoh) wrote:

And, this is a new topic. There is an non-Symbol-key call, like where("table.id" => 1) (which was shown by Jeremy Evans). This is difficult to change to keyword argument. To allow this, matz came up with an idea: non-Symbol key is also allowed as a keyword.

```
def foo(**kw)
  p kw
end

foo("str" => 42) #=> {"str"=>42}
```

Note that, if you need to write a library that works on both 2.X and 3.X, you must write a shim:

```
def foo(kw1 = {}, **kw2)
  kw = kw1.merge(kw2)
  kw
end
```

However, after EOL of all Ruby 2.X series, you can remove the shim and just write a simple code. This is better than my original proposal.

What do you think?

I agree with the proposed changes to foo-like (keyword arguments) methods, as those changes actually solve real problems with keyword arguments.

I disagree with the proposed changes to bar-like (positional hash arguments) methods. Those changes do not solve existing problems, they just break existing code for the potential future ability to introduce keyword arguments without behavior changes.

Let's consider if this proposed changes to bar-like (positional hash arguments) methods is accepted. The main argument for acceptance is the ability to introduce keyword arguments without behavior changes. However, in many if not most cases where bar-like methods are used, keyword arguments would be used to replace the option hashes (something that works OK in 2.X except for corner cases with optional position arguments and argument splats), not as an addition to option hashes. With the changes discussed to foo-like methods, you would no longer be able to replace an option hash argument with keyword arguments in a backwards compatible manner. So the fact that the proposed changes to bar-like methods allow keyword arguments to be introduced in a backwards compatible manner will not help, since replacing the option hashes with keywords will still be a backwards incompatible change.

There are many cases where you have a method that accepts a hash where you have cases where you want to pass an existing hash and other cases where you want pass a new hash, and having to add braces to all call-sites where you currently can omit them would be annoying, add no value, and make the code slightly harder to read.

The performance disadvantages to keyword splats that I discussed earlier still have not be addressed, and it is still impossible to create a method that accepts arbitrary keyword arguments and delegates the call to another method that accepts arbitrary keyword arguments without at least 3 hash allocations per-call (and you can have 0 hash allocations per call with a option-hash based approach).

In conclusion, the ability to add keyword arguments in a backwards compatible manner to methods that accept option hashes adds very little benefit. I think there are huge costs in breaking existing compatibility (potentially leading to a Python 2/3-like situation in libraries), and other costs in making code using bar-like methods harder to read (by requiring braces), as well as hurting performance by encouraging unoptimized keyword splats as a replacement for option hashes.

The ability for **kw to accept non-Symbol keys would make it a slightly easier to convert option hash methods to keyword arguments methods, but the keyword argument approach would still perform worse due to the additional hash allocations, and converting option hashes to keyword splats would still not be backwards compatible, and I think in most cases using an options hash would still be the preferable approach. So even if **kw handled non-Symbol keys, I would still be strongly against changing the behavior for bar-like methods.

#47 - 12/22/2018 11:41 AM - decuplet (Nikita Shilnikov)

I'm not sure if this was discussed but one more thing to consider is hash destructuring using keywords. As far as I understand there's a use case I rely on which is going to be broken by the proposed changes. Specifically, things like

```
xs = [a: 1, b: 2, c: 3]
xs.map { |a:, b:, c:| ... }
```

Perhaps it makes sense to make hash destructuring a separate feature with the following syntax

```
xs.map { |{a, b, c}| ... }
xs.map { |{a, **rest}| ... }
```

But it seems to be a separate feature and I don't see how this can help with other compatibility issues mentioned here.

#48 - 12/28/2018 12:01 AM - ioquatix (Samuel Williams)

I agree we should fix this issue. It is very much unexpected behaviour and even context sensitive behaviour. Double splat operator should be required in all cases to turn hash into keyword arguments.

#49 - 03/18/2019 09:21 AM - mame (Yusuke Endoh)

Sorry for leaving this ticket. Matz, akr and I talked about this issue several times since the last year, and we have never reached a perfect solution.

But I try to re-summarize the problem, current proposal, and migration path.

Problem

The current spec of keyword arguments is broken in several senses.

1. Keyword extension is not always safe

We call "keyword extension" to add a keyword parameter to an existing method.
Unfortunately, keyword extension is not safe when the existing method accepts rest arguments.

```
def foo(*args)
  p args
end
foo(key: 42) #=> [{:key=>42}]
```

If we add a new mode to the method, the existing call will break.

```
def foo(*args, output: $stdout)
  output.puts args.inspect
end
foo(key: 42) #=> unknown keyword: key
```

Safe keyword extension is a fundamental expectation for keyword arguments, so that is a pity.

2. Explicit Delegation of keywords backfires

You are writing a delegation, and you think of keywords, so you wrote:

```
def foo(*args, **kw, &blk)
  bar(*args, **kw, &blk)
end
```

However, this does not work correctly.

```
def bar(*args)
  p args
end

foo() #=> excepted:[], actual:[{}]
```

3. There are many unintuitive corner cases

There are many bug reports about keyword arguments. One of the most weird cases:

```
def foo(opt=42, **kw)
  p [opt, kw]
end

foo({}, **{})  #=> expected:[{}, {}], actual:[42, {}]
```

All of these issues are caused by the fundamental design flaw of the current keyword arguments which handles a keyword as a last positional argument that is a Hash object. Matz, akr and I have considered these issues seriously. Actually, matz came up with multiple ideas that would be compatible (or mildly incompatible) and solve the issues. However, all of them were proved to be incompatible, complex, and/or not to solve some of the above issues.

Proposal for 3.X semantics

The current proposal consists of two parts:

A) Separate keyword arguments from positional arguments completely
B) Allow non-Symbol keys as a keyword

(A) is the original proposal of this ticket.

- A keyword argument is passed only by foo(k: 1) or foo(**opt), and accepted only by def foo(k: 1) or def foo(**opt).
- A positional Hash argument is passed only by foo({ k: 1 }) or foo(opt), and accepted only by def foo(opt) or def foo(opt={}) or def foo(*args)

See the next section in detail.

(B) allows some DSL usages of brace omission:

```
def where(**kw)
  p kw
end

where("table.id" => 42) #=> {"table.id"=>42}
```

Actually, this behavior is not new. Ruby 2.0.0-p0 allowed non-Symbol keys.

Typical rewrite cases

This change brings incompatibility, so you need to rewrite existing code. Typical rewrite cases are three (plus one):

1. Accept keywords by **opt, not by opt={}

```
# NG in 3.X
def foo(opt={})
end

# OK in 3.X
def foo(**opt)
end
```

2. Pass keywords without braces, or with explicit **

```
def foo(**opt)
end

# NG in 3.X
foo({ k: 1 })
h = { k: 1 }
foo(h)

# OK in 3.X
foo(k: 1)
foo(**h)
```

3. Delegate keyword argument explicitly

```
# NG in 3.X
def foo(*args, &blk)
  bar(*args, &blk)
end

# OK in 3.X
def foo(*args, **kw, &blk)
  bar(*args, **kw, &blk)
end
```

Plus one. Manually merge the last argument with a keyword argument

If you want to allow both calling styles, you can do it manually.

```
# NG in 3.X
def foo(opt={})
  p opt
end
foo({ k: 1 }) #=> {:k=>1}
foo(k: 1)     #=> expected:{:k=>1}, actual:error

# OK in 3.X
def foo(opt={}, **kw)
  opt = opt.merge(kw)
  p opt
end
foo({ k: 1 }) #=> {:k=>1}
foo(k: 1)     #=> {:k=>1}
```

Migration path: 2.7 semantics

Basic approach:

- If a code is valid (no exception raised) in 3.X, Ruby 2.7 should run it in the same way as 3.X
- If a code is invalid (an exception raised) in 3.X, Ruby 2.7 should run it in the same way as 2.6, but a warning is printed

Typical examples:

```
def foo(opt)
end
foo(k: 1) #=> test.rb:3: warning: The keyword argument for `foo' (defined at test.rb:1) is used as the last pa
rameter

def foo(**opt)
end
foo({ k: 1 }) #=> test.rb:3: warning: The last argument for `foo' (defined at test.rb:1) is used as the keywor
d parameter
```

These warnings tell users how to fix the source code.

(A naive implementation of this approach is not enough. Very subtle hack is required for delegation. This is explained in the last appendix section.)

Experiment

I have implemented 2.7's candidate semantics:

https://github.com/ruby/ruby/compare/trunk...mame:keyword-argument-separation

And I actually modified the standard libraries and tests to support the keyword argument separation. Many of the changes are one of the three (plus one) typical rewrite cases. There are a few tricky modifications, but in my opinion, almost all of them were trivial.

In addition, I tested an internal Rails app in my company (about 10k lines) with my prototype. Honestly speaking, when running rake spec, it produces about 120k (!) warnings, but there are many duplicated warnings. By removing the duplications, we got about 1k warnings. And, I found that almost all warnings were produced in gems. If we focus on only the application itself, we found only five method definitions to be modified. All fixes were the first typical rewrite case: def foo(opt={}) -> def foo(**opt)). We will need to rewrite some more calls to add an explicit ** if some libraries decided that their APIs only accept keywords.

Appendix: Special frozen Hash object for delegation

Unfortunately, the naive implementation of the migration path is incomplete with regard to delegation.
Consider the following code.

```
# in 2.7
def f1(k: 1)
  p k
end

def f2(*args)
  p args
end

def dispatch(target, *args, &blk)
  if target == :f1
    f1(*args, &blk)
  else
    f2(*args, &blk)
  end
end

dispatch(:f1, k: 1) #=> 1
#=> t.rb:17: warning: The keyword argument for `dispatch' (defined at t.rb:9) is used as the last parameter
#   t.rb:11: warning: The last argument for `f1' (defined at t.rb:1) is used as the keyword parameter
#   1

dispatch(:f2, 1, 2, 3) #=> [1, 2, 3]
```

You see a warning, so you rewrite it by explicit keyword delegation:

```
# in 2.7
def f1(k: 1)
  p k
end

def f2(*args)
  p args
end
```

```
def dispatch(target, *args, **kw, &blk)
  if target == :f1
    f1(*args, **kw, &blk)
  else
    f2(*args, **kw, &blk)
  end
end

dispatch(:f1, k: 1)     #=> 1
dispatch(:f2, 1, 2, 3) #=> [1, 2, 3, {}]
#=> t.rb:18: warning: The keyword argument for `f2` (defined at t.rb:4) is used as the last parameter
```

dispatch(:f1, k: 1) works perfectly with no warnings. However, the result of dispatch(:f2, 1, 2, 3) changed and a new warning is emitted. This is because **kw was automatically converted to a positional argument (due to 2.6 compatibility layer).

To fix this issue, we introduce a Hash flag to distinguish between "no keyword given" and "empty keyword given".

```
def foo(**kw)
  p kw
end

foo(**{}) #=> {}
foo()     #=> {(NO KEYWORD)}
```

{} is a normal empty hash object, and {(NO KEYWORD)} is the special empty hash object that represents "no keyword given".

If we pass the flagged empty hash to another method with ** operator, it is omitted.

```
def bar(*args)
  p args
end

def foo(**kw)
  # kw is {(NO KEYWORD)}
  bar(**kw) # **{(NO KEYWORD)} is equal to nothing: bar()
end

foo({}) #=> [{}]
foo()   #=> []
```

This is akr's idea that was explained at https://bugs.ruby-lang.org/issues/14183#note-41.

This hack of special empty hash flag is temporal just during the migration. After 3.X completes the separation of keyword arguments, this dirty hack can be removed.

#50 - 03/18/2019 07:22 PM - jeremyevans0 (Jeremy Evans)

mame,

Thanks for your continued work on this.

I still agree that for methods that accept keyword arguments, we should make changes to avoid the problems that currently exist for keyword arguments.

I still believe that breaking all backwards compatibility for methods that do not currently accept keyword arguments, just to allow keyword arguments to be added safely in the future, is not a worthy tradeoff, for the following reasons:

- Many if not most of the methods may never be converted to keyword arguments, in which case backwards compatibility is broken for no benefit.
- This encourages the use of keyword arguments, while the use of keyword arguments hurts performance in all cases where keyword splats are used (either at the caller side or the callee side). The option hash approach can be made faster and allocation-less, while all keyword splats are currently slower as they require allocations. I'm not sure that the keyword argument splat performance issues could be fixed without breaking backwards compatibility for all keyword argument splats.
- For methods that currently use option hashes, requiring braces around the option hash can make it more difficult to convert to keyword arguments, not less. A method such as def foo(opts={}) end that is usually called using foo(bar: 1), will still work if you switch to keyword arguments: def foo(**opts) end. It is true that the foo(hash) calling style would require modifications with the switch to keyword arguments, though.

I think the biggest problem with keeping backwards compatibility for methods that do not accept keyword arguments is handling delegation.

```
def foo(*a, **kw, &block)
  bar(*a, **kw, &block)
```

```
end
```

I believe with your proposal, this is expected to work regardless of whether bar accepts keyword arguments. If bar doesn't accept keyword arguments, then calling foo with a keyword argument will raise an exception when foo calls bar. I think one possible way to get that simple delegation to work would be to allow double-splat when calling methods that do not accept keyword arguments (keep backwards compatibility). For example, allow this:

```
def bar(hash={})
  hash[:a]
end

bar(**{a: 1})
# => 1

foo(**{a: 1})
# => 1
```

This keeps backwards compatibility back to Ruby 2.0. It will also make it easier to transition such code to keyword arguments later without breaking backwards compatibility, since changing the definition of bar to def bar(**hash) hash[:a] end would still work in that case.

The main problematic case would be if bar accepted a positional splat but did not accept keyword arguments, where an empty hash would be provided if no keyword arguments were used:

```
def bar(*a)
  a
end

bar
# => []

foo
# => [{}]
```

One possible way around that would be that if a method accepts a positional splat and does not accept keyword arguments, then calling the method with an empty keyword argument splat would not pass a positional argument. Proposed behavior:

```
def bar(*a)
  a
end

bar
# => []

foo
# => []

bar(**{})
# => []

foo(**{})
# => []

bar(1, a: 1)
# => [1, {a: 1}]

foo(1, a: 1)
# => [1, {a: 1}]
```

My Proposed Alternative

To sum up, here is my proposed alternative approach:

- For methods that accept keyword arguments, the same as your proposal
- For methods that do not accept keyword arguments:
  - Allow braceless hashes as positional arguments (keep backwards compatibility)
  - Allow **keyword splats
    - If keyword is empty hash, do not add the empty hash positional argument (new behavior)
    - Otherwise, add keyword as positional hash argument (keep backwards compatibility)

I think this alternative proposal handles "2. Explicit Delegation of keywords backfires" and "3. There are many unintuitive corner cases". It does not handle "1. Keyword extension is not always safe". However, I believe you could keep safe keyword extension if using keyword splat, using an approach that works and is backwards compatible to Ruby 2.0. From your example:

```
# Before
```

```
def foo(*args)
  p args
end
foo(key: 42)
# => [{:key=>42}]

# Add keyword arguments
def foo(*args, output: $stdout, **kw)
  args << kw
  output.puts args.inspect
end
foo(key: 42)
# => [{:key=>42}]
```

Issues with keyword-argument-separation branch

In terms of the specific implementation in your keyword-argument-separation branch:

The rb_no_keyword_hash approach breaks modification of the hash, which I believe is unexpected:

```
def foo(**opts)
  opts
end

foo
#  => {(NO KEYWORD)}

def foo(**opts)
  opts[:a] = 1
  opts
end

foo
# FrozenError (can't modify frozen Hash)
```

It may be possible to work around that by setting a flag on the hash instead of using a shared frozen hash, assuming there is a spare flag we can use for that purpose. If a flag isn't available, we probably could use an instance variable that doesn't start with @ (making it only visible to C).

The warning seems inconsistent. For positional splats, you get warned if the braceless hash is the first argument, but not if it is a subsequent argument:

```
def bar(*a)
  a
end

bar
=> []

bar(a: 1)
# warning: The keyword argument for `bar' (defined at XXX) is used as the last parameter
# => [{:a=>1}]

bar(1, a: 1)
# => [1, {:a=>1}]
```

This situation also occurs for methods without splats where both arguments are optional (and maybe other cases):

```
def baz(a=1, b={})
  [a, b]
end

baz
# => [1, {}]

baz(a: 2)
# warning: The keyword argument for `baz' (defined at XXX) is used as the last parameter
[{:a=>2}, {}]

baz(1, a: 2)
# => [1, {:a=>2}]
```

Is that behavior in regards to warnings expected?

Behavior is different for methods defined in C, as C methods are always passed a hash, so the brace, braceless, and splat forms all work:

```
String.new(capacity: 1000)
# => ""
String.new({capacity: 1000})
# => ""
String.new(**{capacity: 1000})
# => ""
```

This results in inconsistent behavior depending how how the method is defined. This will lead to backwards compatibility problems if you move a method definition from C to ruby, or if you have a method defined in both C and ruby, with the pure ruby version used as a fallback if the C version cannot be used.

I look forward to discussing this issue in person at the developer meeting next month.

#51 - 03/25/2019 10:50 PM - jeremyevans0 (Jeremy Evans)

- File vm_args.diff added

Since I think it is best to back proposed behavior changes with a proposed implementation, attached is a patch based on mame's keyword-argument-separation branch that implements my proposal:

- Same behavior as mame's for methods that accept keyword arguments
- For methods that do not accept keyword arguments
  - Allow use of braceless hash without warning (keep backwards compatibility)
  - Allow **keyword splats
    - If keyword splat is empty, do not add positional argument (new behavior)
    - Otherwise, add hash as positional argument (keep backwards compatibility)

mame, if you have the time, could you try this patch with your internal Rails app, using the same checkout that resulted in about 120k warnings, and see how many warnings it causes and whether not adding a positional argument for an empty keyword splat breaks any code?

I'm not sure the patch is the best approach possible. I have limited knowledge of and experience with the VM internals. This patch is the minimum change necessary, it doesn't remove the rb_no_keyword_hash variable, even though I don't think the variable is needed if we do not pass positional arguments for empty keyword splats.

#52 - 03/29/2019 10:31 AM - mame (Yusuke Endoh)

- File vm_args_v2.diff added

Jeremy,
I really appreciate you to use time for this issue. And sorry for my late response.

I have misunderstood some points of your proposal, and now I feel that it is fairly good. But please let me consider for a while... This topic is really hard to exhaust corner cases.

My Proposed Alternative

Just confirm. I think your following snippet lacks unless kw.empty?, right?

```
# Add keyword arguments
def foo(*args, output: $stdout, **kw)
  args << kw unless kw.empty? # This "unless" modifiler is needed, I think.
  output.puts args.inspect
end
foo(key: 42)
# => [{:key=>42}]
```

And, foo({}) will assign args = [{}], right? If so, your proposal looks good enough to me.
Of course, if there was a call foo(output: 42) before adding keywords, the call will break.
That is unfortnate, but this may be a good compromise.

Issues with keyword-argument-separation branch

Thank you for checking my prototype deeply!

> The rb_no_keyword_hash approach breaks modification of the hash, which I believe is unexpected:

Yes. Akr and I knew that this would bring some incompatibility. We expected that the incompatibility should be small, but I noticed that it doesn't, unfortunately. It should be fixed by something like special instance variable, as you said.

> The warning seems inconsistent. For positional splats, you get warned if the braceless hash is the first argument, but not if it is a subsequent argument:

Good catch, I didn't intend it. I fixed my branch. And this brings more warnings ;-) so I re-examined our internal Rails app (explained later). And the modification of my branch made your patch inapplicable, so I'm attaching a modified version of your patch.

> Behavior is different for methods defined in C, as C methods are always passed a hash, so the brace, braceless, and splat forms all work:

We will keep the compatibility of C API because it would be more difficult to fix. Regardless of whether the brace is used, C method consisitently receives a hash.
By the way, ko1 is now working on replacement of built-in methods from C to Ruby. (He will talk about this plan in RubyKaigi: Write a Ruby interpreter in Ruby for Ruby 3.)
His main motivation is performance, but this will also reduce the problem of C methods that receives keyword arguments.

And, thank you for your alternative patch. I tried it with our internal Rails app again. It emitted about 8k warnings (much less than 120k!). Unfortunately I have no enough time to analyze the result, but it looks that no modification is required in our code base. Great.

#53 - 03/29/2019 02:57 PM - jeremyevans0 (Jeremy Evans)

mame (Yusuke Endoh) wrote:

> # My Proposed Alternative
>
> Just confirm. I think your following snippet lacks unless kw.empty?, right?

Correct. Sorry about that.

```
# Add keyword arguments
def foo(*args, output: $stdout, **kw)
  args << kw unless kw.empty? # This "unless" modifiler is needed, I think.
  output.puts args.inspect
end
foo(key: 42)
# => [{:key=>42}]
```

> And, foo({}) will assign args = [{}], right?

Correct, in Ruby 3 assuming the behavior changes for keyword arguments are in effect. With your branch+my patch:

```
foo({})
# warning: The last argument for `foo' (defined at (irb):1) is used as the keyword parameter
# output: []
```

With your branch+my patch, you can work around the warning by passing an empty keyword splat:

```
foo({}, **{})
# output: [{}]
```

> If so, your proposal looks good enough to me.

Great!

> # Issues with keyword-argument-separation branch
>
> Thank you for checking my prototype deeply!
>
> > The rb_no_keyword_hash approach breaks modification of the hash, which I believe is unexpected:
>
> Yes. Akr and I knew that this would bring some incompatibility. We expected that the incompatibility should be small, but I noticed that it doesn't, unfortunately. It should be fixed by something like special instance variable, as you said.

One issue with the special instance variable approach is that if you add an entry to the keyword hash, you probably do want to pass the keyword arguments even if the instance variable is present. So you would want to also check that the hash is still empty. Example:

```
def foo(*a, **kw)
  kw[:b] = 1 if a.length == 1
  bar(*a, **kw)
end

def bar(*a)
  a
end

foo
# => []

foo(1)
# => [1, {:b=>1}]
```

In my patch, we skip passing all empty keyword argument splats as hashes, so it should already handle this case (once the keyword splat hash is no longer frozen).

> The warning seems inconsistent. For positional splats, you get warned if the braceless hash is the first argument, but not if it is a subsequent argument:

> Good catch, I didn't intend it. I fixed my branch. And this brings more warnings ;-) so I re-examined our internal Rails app (explained later). And the modification of my branch made your patch inapplicable, so I'm attaching a modified version of your patch.

Thank you, I will try to do some more testing with your revised branch and the modified patch next week.

> Behavior is different for methods defined in C, as C methods are always passed a hash, so the brace, braceless, and splat forms all work:

> We will keep the compatibility of C API because it would be more difficult to fix. Regardless of whether the brace is used, C method consisitently receives a hash.

I figured that would be difficult to change. I think my patch would make C-methods perform the same as Ruby methods without keywords, which is probably best for compatibility other Ruby implementations that do not implement the C-API and use alternatives written in Ruby.

> By the way, ko1 is now working on replacement of built-in methods from C to Ruby. (He will talk about this plan in RubyKaigi: Write a Ruby interpreter in Ruby for Ruby 3.)
> His main motivation is performance, but this will also reduce the problem of C methods that receives keyword arguments.

That is very interesting. I will make sure to attend ko1's presentation.

> And, thank you for your alternative patch. I tried it with our internal Rails app again. It emitted about 8k warnings (much less than 120k!). Unfortunately I have no enough time to analyze the result, but it looks that no modification is required in our code base. Great.

It is great to hear that it required no changes in your app's code, only requiring changes in gems that are passing hashes to methods that expect keywords.

#54 - 04/01/2019 02:52 PM - matz (Yukihiro Matsumoto)

jeremyevans0 (Jeremy Evans) I **will** investigate your proposal. I was not fully satisfied with the complete separation model proposed for 3.0, but I didn't think of any other model which is intuitive and clean, especially considering static type analysis. Your proposal could be a better alternative.

Matz.

#55 - 04/01/2019 11:09 PM - jeremyevans0 (Jeremy Evans)

jeremyevans0 (Jeremy Evans) wrote:

> mame (Yusuke Endoh) wrote:
>
> > Good catch, I didn't intend it. I fixed my branch. And this brings more warnings ;-) so I re-examined our internal Rails app (explained later). And the modification of my branch made your patch inapplicable, so I'm attaching a modified version of your patch.
>
> Thank you, I will try to do some more testing with your revised branch and the modified patch next week.

mame,

With your revised branch, it looks like the the keyword argument separation for positional splats has already happened, and there is no warning. Both

your revised branch and your previous branch also already implement keyword argument separation for optional positional arguments without a warning. There is still a warning for the case where all positional arguments are required, though.

Example code:

```
def foo(a, *b, **c)
  [a, b, c]
end

def bar(a, b=1, **c)
  [a, b, c]
end

def baz(a, **c)
  [a, c]
end
```

Your revised branch (commit 73a9633114ef00bf793d7ca39e49f24448499487)

```
foo(1, {a: 1})
# => [1, [{:a=>1}], {(NO KEYWORD)}]

bar(1, {a: 1})
# => [1, {:a=>1}, {(NO KEYWORD)}]

baz(1, {a: 1})
# warning: The last argument for `baz' (defined at (irb):9) is used as the keyword parameter
# => [1, {:a=>1}]
```

Your previous branch (commit 3903e75678eca4874e3122a42bd073b018f9458e):

```
foo(1, {a: 1})
# warning: The last argument for `foo' (defined at (irb):15) is used as the keyword parameter
# => [1, [], {:a=>1}]

bar(1, {a: 1})
# => [1, {:a=>1}, {(NO KEYWORD)}]

baz(1, {a: 1})
# warning: The last argument for `baz' (defined at (irb):9) is used as the keyword parameter
# => [1, {:a=>1}]
```

Ruby 2.6:

```
foo(1, {a: 1})
# => [1, [], {:a=>1}]

bar(1, {a: 1})
# => [1, 1, {:a=>1}]

baz(1, {a: 1})
# => [1, {:a=>1}]
```

I believe the expected behavior in Ruby 2.7 is to warn but return the same results as Ruby 2.6 in all three cases, is that correct?

I applied your vm_args_v2.diff on top of your revised branch, and also removed the rb_no_keyword_hash variable and related handling. No compilation issues, and some basic tests work, but many stdlib tests fail due to the keyword argument separation already being applied for methods that use positional splats (mostly tmpdir and csv).

To make testing easier, I uploaded my branch GitHub: https://github.com/jeremyevans/ruby/commits/keyword-argument-separation

After the issues with positional splats and optional positional arguments are fixed, I'll rebase my patch on top of that. Note that my branch does not include your changes to the standard library and tests to avoid warnings. I believe the changes required to the standard library and tests should be much less extensive with my proposal, and I would like to only make the minimum changes necessary. I want to make sure the branch does not cause any failures before attempting to remove warnings.

#56 - 04/08/2019 02:29 AM - jeremyevans0 (Jeremy Evans)

I have updated my GitHub branch (https://github.com/jeremyevans/ruby/commits/keyword-argument-separation) to fix the issues in mame's branch that I identified in my previous comment.

Now, my branch keeps compatibility with Ruby 2.6 in regards to treating a hash argument as keywords, issuing warnings as expected for all three cases where behavior will change in Ruby 3:

```
def foo(a, *b, **c)
```

```
  [a, b, c]
end

def bar(a, b=1, **c)
  [a, b, c]
end

def baz(a, **c)
  [a, c]
end

foo(1, {a: 1})
# warning: The last argument for `foo' (defined at (irb):1) is used as the keyword parameter
# => [1, [], {:a=>1}]

bar(1, {a: 1})
# warning: The last argument for `bar' (defined at (irb):5) is used as the keyword parameter
# => [1, 1, {:a=>1}]

baz(1, {a: 1})
# warning: The last argument for `baz' (defined at (irb):9) is used as the keyword parameter
# => [1, {:a=>1}]
```

I have also found another behavior change with mame's branch that I think is undesirable, and that is how positional hash arguments with non-Symbol keys are converted to keywords. That breaks backwards compatibility with Ruby 2.6, and there is no reason to change behavior and emit a warning in Ruby 2.7 when the Ruby 3 behavior will be same as Ruby 2.6 in this case.

Ruby 2.6 and my branch:

```
def a(x=1, **h)
  [x, h]
end

a({:a=>1})
# => [{}, {:a=>1}]

a({"a"=>1})
# [{"a"=>1}, {}]
```

There is still backwards compatibility breakage in cases where the last positional hash has both Symbol keys and non-Symbol keys. In Ruby 2.6, those hashes would be split, but I'm not sure if we want to keep backwards compatibility and warn about that case in Ruby 2.7. Doing so would probably increase complexity significantly. My branch changes the behavior so that the positional hash is always treated as a positional argument if it contains a non-Symbol key:

Ruby 2.6:

```
a({"a"=>1, :a=>1})
# => [{"a"=>1}, {:a=>1}]
```

My branch:

```
a({"a"=>1, :a=>1})
# => [{"a"=>1, :a=>1}, {}]
```

I have fixed all issues in lib that caused warnings, and no changes are required in ext. Most changes were in the csv library, with minor changes in net, rdoc, rubygems, tempfile, and tmpdir. Here is a stat for issues in lib:

```
 lib/csv.rb                              | 36 ++++++++++++++++++------------------------
 lib/csv/core_ext/array.rb               |  2 +-
 lib/csv/core_ext/string.rb              |  2 +-
 lib/csv/row.rb                          |  2 +-
 lib/csv/table.rb                        |  4 ++--
 lib/net/ftp.rb                          |  2 +-
 lib/net/protocol.rb                     |  2 +-
 lib/rdoc/generator/darkfish.rb          | 12 ++++++------
 lib/rubygems.rb                         |  2 +-
 lib/rubygems/commands/setup_command.rb  |  2 +-
 lib/rubygems/package.rb                 |  2 +-
 lib/tempfile.rb                         |  4 ++--
 lib/tmpdir.rb                           |  4 ++--
```

Here's a comparison with the changes mame's branch requires in lib and ext:

```
 ext/etc/extconf.rb                                      |   2 +-
 ext/json/lib/json/common.rb                             |  28 ++++++++++++++--------------
```

```
 ext/json/lib/json/generic_object.rb                    |   4 ++--
 ext/openssl/lib/openssl/ssl.rb                         |   4 ++--
 ext/psych/lib/psych.rb                                 |   2 +-
 ext/psych/lib/psych/core_ext.rb                        |   4 ++--
 lib/bundler/dsl.rb                                     |   6 +++---
 lib/bundler/runtime.rb                                 |   2 +-
 lib/cgi/core.rb                                        |   7 ++++---
 lib/cgi/html.rb                                        |  11 ++++++-----
 lib/csv.rb                                             |  24 ++++++++++++------------
 lib/csv/core_ext/array.rb                              |   2 +-
 lib/csv/core_ext/string.rb                             |   2 +-
 lib/csv/row.rb                                         |   2 +-
 lib/csv/table.rb                                       |   4 ++--
 lib/erb.rb                                             |   3 ++-
 lib/mkmf.rb                                            |   4 ++--
 lib/net/ftp.rb                                         |  10 +++++-----
 lib/net/http.rb                                        |   5 ++---
 lib/net/http/generic_request.rb                        |   2 +-
 lib/net/imap.rb                                        |   2 +-
 lib/net/protocol.rb                                    |   2 +-
 lib/open-uri.rb                                        |  23 ++++++++++++-----------
 lib/open3.rb                                           |  96 +++++++++++++++++++++++++-----------------------
------------------------------------------------
 lib/rdoc/generator/darkfish.rb                         |  18 +++++++++---------
 lib/resolv.rb                                          |  18 +++++++-----------
 lib/rexml/document.rb                                  |   6 ++----
 lib/rexml/xpath.rb                                     |   2 +-
 lib/rss/parser.rb                                      |   7 +------
 lib/rss/rss.rb                                         |   9 ++-------
 lib/rubygems.rb                                        |   6 +++---
 lib/rubygems/command.rb                                |   4 ++--
 lib/rubygems/commands/install_command.rb               |   4 ++--
 lib/rubygems/commands/pristine_command.rb              |   4 ++--
 lib/rubygems/commands/setup_command.rb                 |   2 +-
 lib/rubygems/dependency_installer.rb                   |   4 ++--
 lib/rubygems/installer.rb                              |   8 ++++----
 lib/rubygems/package.rb                                |   2 +-
 lib/rubygems/request_set.rb                            |  10 +++++-----
 lib/rubygems/request_set/gem_dependency_api.rb         |  11 +++++------
 lib/rubygems/resolver/git_specification.rb             |   4 ++--
 lib/rubygems/resolver/lock_specification.rb            |   2 +-
 lib/rubygems/resolver/specification.rb                 |   8 ++++----
 lib/rubygems/test_case.rb                              |  18 +++++++++---------
 lib/rubygems/test_utilities.rb                         |  14 +++++++-------
 lib/tempfile.rb                                        |   8 ++++----
 lib/uri/file.rb                                        |   4 ++--
 lib/uri/ftp.rb                                         |   2 +-
 lib/uri/generic.rb                                     |  22 +++++++++++-----------
 lib/uri/http.rb                                        |   6 +++---
 lib/uri/mailto.rb                                      |   2 +-
 lib/yaml/store.rb                                      |   4 ++--
```

I have fixed all issues in test that caused warnings. These were also much less extensive than in mame's branch.

I have fixed a few broken tests that expected a specific format for warning messages for invalid keywords, as well as a test that assumed TypeError for a non-Symbol keyword (an ArgumentError is now used).

#57 - 04/09/2019 03:20 AM - jeremyevans0 (Jeremy Evans)

I have updated my branch (https://github.com/jeremyevans/ruby/commits/keyword-argument-separation) to restore backwards compatibility for methods using keyword arguments when calling with a final positional hash with mixed Symbol and non-Symbol keys. These calls are now handled like Ruby 2.6, splitting the hash into a positional hash for the non-Symbol keys and using the Symbol keys as keywords. A warning is added so that developers know they need to update their code, as in Ruby 3 this will always be treated as a positional argument without being split.

Example (same behavior as Ruby 2.6 except for warnings):

```
def a(x=1, **h)
  [x, h]
end

a({:a=>1})
# (irb):5: warning: The last argument for `a' (defined at (irb):1) is used as the keyword parameter
# => [1, {:a=>1}]

a({"a"=>1})
```

```
# => [{"a"=>1}, {}]

a({"a"=>1, :a=>1})
# (irb):7: warning: The last argument for `a' (defined at (irb):1) is split into positional and keyword parame
ters
# => [{"a"=>1}, {:a=>1}]
```

I did not change the behavior for bare-keywords with Symbol and non-Symbol keys, only for positional hashes. Now that keywords can support non-Symbol keys (in this branch), I do not think it makes sense to restore backwards compatibility with Ruby 2.6 for those. I think that would only make sense if we are going to delay support for non-Symbol keys until Ruby 3.

My Branch:

```
a(:a=>1)
# =>[1, {:a=>1}]

a("a"=>1)
# => [1, {"a"=>1}]

a("a"=>1, :a=>1)
# => [1, {"a"=>1, :a=>1}]
```

Ruby 2.6:

```
a(:a=>1)
# => [1, {:a=>1}]

a("a"=>1)
# => [{"a"=>1}, {}]

a("a"=>1, :a=>1)
# => [{"a"=>1}, {:a=>1}]
```

#58 - 04/12/2019 08:26 AM - mame (Yusuke Endoh)

Jeremy, thank you for working on this issue.

> I believe the expected behavior in Ruby 2.7 is to warn but return the same results as Ruby 2.6 in all three cases, is that correct?

I had intended the incompatibility, as I said in "Migration path: 2.7 semantics" section of note-45:

> Basic approach:
>
> - If a code is valid (no exception raised) in 3.X, Ruby 2.7 should run it in the same way as 3.X
> - If a code is invalid (an exception raised) in 3.X, Ruby 2.7 should run it in the same way as 2.6, but a warning is printed

Akr also objects this approach, so I withdraw this proposal.

However, akr and I think that "2.7 is completely compatible with 2.6 except warnings" approach is not good enough. We need to provide a migration path that allows users to rewrite their code for 3.0 gradually. So, Ruby 2.7 must run (reasonably almost) all programs that are valid as either 2.6 or 3.0. (I expected the above proposal to be good enough, but turned out somewhat too breaking.)

In other words, if a warning is printed, there must be a reasonable change that is not warned (i.e., will work in 3.0) and that causes no behavior change (in, at least, 2.7 semantics).

Consider delegation. Currently we write:

```
# we call this "old-style delegation"
def foo(*args, &blk)
  bar(*args, &blk)
end
```

but this is warned when keywords are passed. So we'd like to rewrite it as:

```
# we call this "new-style delegation"
def foo(*args, **kw, &blk)
  bar(*args, **kw, &blk)
end
```

However, this rewrite changes the behavior unfortunately in 2.6 semantics because of the problem "2. Explicit Delegation of keywords backfires"

```
def bar(*args)
  p args
```

```
end

def foo0(*args, &blk)
  bar(*args, &blk)
end

def foo1(*args, **kw, &blk)
  bar(*args, **kw, &blk)
end

bar()  #=> []
foo0() #=> []
foo1() #=> [{}] # broken
```

rb_no_keyword_hash trick works gracefully in this case. The trick allows the new-style delegation in (almost) 2.6 semantics. So the hack is needed, we think.

#59 - 04/12/2019 01:52 PM - mame (Yusuke Endoh)

jeremyevans0 (Jeremy Evans),

You registered this ticket for pre-RubyKaigi [Misc#15459]. Do you have an idea how to discuss the issue?

ko1 (Koichi Sasada) is now creating an agenda, and maybe 30 minutes will be allotted to this issue. The agenda is not decided yet, though.

To make it easy to discuss the issue, I'm creating a slide deck.

https://docs.google.com/presentation/d/16rReiCVzUog3s5vV702LzcIFM2LNcX53AX8m5K8uCZw

I hope that this would be helpful and fair, but could you check the content? If you want to edit it yourself, emaiil me (mame@ruby-lang.org) your google acount.

If you have already prepared something, you can ignore my slide. Anyway, I'm happy if you let me know. Thanks.

#60 - 04/12/2019 03:34 PM - jeremyevans0 (Jeremy Evans)

mame (Yusuke Endoh) wrote:

> However, akr and I think that "2.7 is completely compatible with 2.6 except warnings" approach is not good enough. We need to provide a migration path that allows users to rewrite their code for 3.0 gradually. So, Ruby 2.7 must run (reasonably almost) all programs that are valid as either 2.6 or 3.0. (I expected the above proposal to be good enough, but turned out somewhat too breaking.)
>
> In other words, if a warning is printed, there must be a reasonable change that is not warned (i.e., will work in 3.0) and that causes no behavior change (in, at least, 2.7 semantics).

I agree we should aim for this. I think it is true with my proposal, but there may be cases I have not considered.

> Consider delegation. Currently we write:
>
> ```
> # we call this "old-style delegation"
> def foo(*args, &blk)
>   bar(*args, &blk)
> end
> ```
>
> but this is warned when keywords are passed. So we'd like to rewrite it as:
>
> ```
> # we call this "new-style delegation"
> def foo(*args, **kw, &blk)
>   bar(*args, **kw, &blk)
> end
> ```
>
> However, this rewrite changes the behavior unfortunately in 2.6 semantics because of the problem "2. Explicit Delegation of keywords backfires"
>
> ```
> def bar(*args)
>   p args
> end
>
> def foo0(*args, &blk)
>   bar(*args, &blk)
> end
>
> def foo1(*args, **kw, &blk)
>   bar(*args, **kw, &blk)
> end
> ```

```
bar()  #=> []
foo0() #=> []
foo1() #=> [{}] # broken
```

rb_no_keyword_hash trick works gracefully in this case. The trick allows the new-style delegation in (almost) 2.6 semantics. So the hack is needed, we think.

The rb_no_keyword_hash hack is not needed in my branch, as my branch returns [] for bar, foo0, and foo1, since double-splatting empty hashes to a method that does not accept keyword arguments does not pass a positional hash in my branch.

I think using the rb_no_keyword_hash hack will cause problems, because it treats some empty hashes different from other empty hashes. Consider the following case, where you want to limit which keyword arguments are passed when delegating:

```
ALLOWED_KEYWORDS = [:baz, :quux]
def foo2(*args, **kw, &blk)
  kw = kw.select{|k| ALLOWED_KEYWORDS.include?(k)}
  bar(*args, **kw, &blk)
end
```

This could be fixed by switching to a mutating method (select!), though.

> You registered this ticket for pre-RubyKaigi [Misc#15459]. Do you have an idea how to discuss the issue?
>
> ko1 (Koichi Sasada) is now creating an agenda, and maybe 30 minutes will be allotted to this issue. The agenda is not decided yet, though.
>
> To make it easy to discuss the issue, I'm creating a slide deck.
>
> https://docs.google.com/presentation/d/16rReiCVzUog3s5vV702LzcIFM2LNcX53AX8m5K8uCZw
>
> I hope that this would be helpful and fair, but could you check the content? If you want to edit it yourself, emaiil me (mame@ruby-lang.org) your google acount.

I briefly reviewed your slide deck and I think it does a good job explaining the various aspects of this issue, and I think we should present it during the developer meeting. I will do a more thorough review later today and email you if I have any suggested changes to the slides.

#61 - 04/17/2019 05:05 AM - Eregon (Benoit Daloze)

With Jeremy's proposal, I think there is no need to support non-Symbol keywords.
I think in the case of ActiveRecord or Sequel's #where, the method doesn't need to accept keywords, it can just accept a Hash.
This Hash might contain numbers or strings as keys, and as such doesn't feel like "keywords arguments" to me.
For instance, method definitions do not have a way to specify non-Symbol keywords.

Re current behavior of splitting a Hash passed as keyword argument, mixing Symbols and non-Symbols keys, I would just raise an ArgumentError or TypeError if the Hash contains a non-Symbol key. It's somewhat similar to the "missing keyword" ArgumentError.

#62 - 04/19/2019 01:32 PM - jeremyevans0 (Jeremy Evans)

During the developer meeting on Wednesday, Matz mentioned that with my approach, it would be useful to have a way to indicate that a method should be treated as a keyword argument method even if does not accept any keyword arguments. Doing so would make it so you could add keyword arguments to the method later in a backwards compatible manner without a workaround (e.g. safe keyword extension, the advantage of mame's approach). Matz recommended the currently invalid **nil syntax for this feature. I have implemented this feature in my branch at https://github.com/jeremyevans/ruby/tree/keyword-argument-separation. Example:

```
def a(a=1, **nil, &b)
  [a, b, local_variables]
end

a(a:1)
# ArgumentError (no keywords accepted)

a({a:1})
# => [{:a=>1}, nil, [:a, :b]]

a(**{a:1})
# ArgumentError (no keywords accepted)

a(**{})
# => [1, nil, [:a, :b]]
```

You can contrast this behavior with the behavior for a method that does not use the **nil syntax:

```
def b(a=1, &b)
```

```
  [a, b, local_variables]
end

b(a:1)
# => [{:a=>1}, nil, [:a, :b]]

b({a:1})
# => [{:a=>1}, nil, [:a, :b]]

b(**{a:1})
# => [{:a=>1}, nil, [:a, :b]]

b(**{})
# => [1, nil, [:a, :b]]
```

More work should be done if this is accepted. Specifically, we need to decide:

- What should Method#parameters return for a method that uses **nil?
- How should ripper handle for the **nil syntax?
- How should RubyVM::AbstractSyntaxTree handle the **nil syntax?

#63 - 04/24/2019 08:03 PM - jeremyevans0 (Jeremy Evans)

jeremyevans0 (Jeremy Evans) wrote:

> More work should be done if this is accepted. Specifically, we need to decide:
>
> - What should Method#parameters return for a method that uses **nil?
> - How should ripper handle for the **nil syntax?
> - How should RubyVM::AbstractSyntaxTree handle the **nil syntax?

I've updated my branch (https://github.com/jeremyevans/ruby/tree/keyword-argument-separation) to add support for the **nil syntax in Method/Proc#parameters, ripper, and RubyVM::AbstractSyntaxTree

Method/Proc#parameters uses a :nokey entry if the **nil syntax is used:

```
proc{||}.parameters
# => []

proc{|**a|}.parameters
# => [[:keyrest, :a]]

proc{|**nil|}.parameters
# => [[:nokey]]
```

Ripper uses :nil for the keyword rest argument if the **nil syntax is used:

```
Ripper.sexp('def a() end')
# => [:program, [[:def, [:@ident, "a", [1, 4]], [:paren, [:params, nil, nil, nil, nil, nil, nil, nil]], [:body
stmt, [[:void_stmt]], nil, nil, nil]]]]

 Ripper.sexp('def a(**b) end')
# => [:program, [[:def, [:@ident, "a", [1, 4]], [:paren, [:params, nil, nil, nil, nil, nil, [:kwrest_param, [:
@ident, "b", [1, 8]]], nil]], [:bodystmt, [[:void_stmt]], nil, nil, nil]]]]

Ripper.sexp('def a(**nil) end')
# => [:program, [[:def, [:@ident, "a", [1, 4]], [:paren, [:params, nil, nil, nil, nil, nil, :nil, nil]], [:bod
ystmt, [[:void_stmt]], nil, nil, nil]]]]
```

RubyVM::AbstractSyntaxTree uses false instead of nil for the keyword and keyword rest arguments if the **nil syntax is used:

```
node = RubyVM::AbstractSyntaxTree.parse("def a() end")
node.children.last.children.last.children[1].children
# => [0, nil, nil, nil, 0, nil, nil, nil, nil, nil]

node = RubyVM::AbstractSyntaxTree.parse("def a(**a) end")
node.children.last.children.last.children[1].children
# => [0, nil, nil, nil, 0, nil, nil, nil, #<RubyVM::AbstractSyntaxTree::Node:DVAR@1:6-1:9>, nil]

node = RubyVM::AbstractSyntaxTree.parse("def a(**nil) end")
node.children.last.children.last.children[1].children
# => [0, nil, nil, nil, 0, nil, nil, false, false, nil]
```

Hopefully these are all of the introspection methods that we need to modify to support **nil. I'm not sure that these are the best ways of handling

each case, but I'm open to better ideas.

#64 - 06/20/2019 08:28 PM - jeremyevans0 (Jeremy Evans)

- Related to Bug #14415: Empty keyword hashes get assigned to ordinal args. added

#65 - 06/27/2019 10:12 PM - jeremyevans0 (Jeremy Evans)

- Related to Bug #12022: Inconsistent behavior with splatted named arguments added

#66 - 06/27/2019 10:12 PM - jeremyevans0 (Jeremy Evans)

- Related to Bug #11860: Double splat does not work on empty hash assigned via variable added

#67 - 06/27/2019 10:12 PM - jeremyevans0 (Jeremy Evans)

- Related to Bug #10708: In a function call, double splat of an empty hash still calls the function with an argument added

#68 - 07/04/2019 11:05 AM - sawa (Tsuyoshi Sawada)

I would like to ask for clarification.

I understand that this feature removes the rule that complements argument-final brace-less key-value pairs with braces. That is, the rule that interprets:

```
foo("bar", a: 1, b:2)
```

as

```
foo("bar", {a: 1, b: 2})
```

will be removed.

There is another case where a similar complementation rule exists. That is, complementing the final key-values pairs in an array literal:

```
[e1, e2, k1: v1, k2: v2]
```

with braces to make them a hash as:

```
[e1, e2, {k1: v1, k2: v2}]
```

What would happen to this rule? Will it be removed as well (in which case the first example above would become ungrammatical)?

#69 - 07/06/2019 12:00 AM - jeremyevans0 (Jeremy Evans)

- Related to Bug #11068: unable to ommit an optional keyarg if the previous arg is an optional hash added

#70 - 07/07/2019 05:36 AM - jeremyevans0 (Jeremy Evans)

*- Related to Bug #11039: method_missing □ *args □□□ symbol□□□□□□ hash□□□□□□ □□□□□□ added*

#71 - 07/07/2019 04:23 PM - jeremyevans0 (Jeremy Evans)

- Related to Bug #10994: Inconsistent behavior when mixing optional argument and keyword splat added

#72 - 07/08/2019 01:58 AM - sawa (Tsuyoshi Sawada)

Can someone answer my question in https://bugs.ruby-lang.org/issues/14183#note-68? Perhaps mame (Yusuke Endoh) knows? If it is not decisive yet, that is fine. I just want to know what the developers have in mind at this point.

#73 - 07/08/2019 02:35 AM - jeremyevans0 (Jeremy Evans)

sawa (Tsuyoshi Sawada) wrote:

> Can someone answer my question in https://bugs.ruby-lang.org/issues/14183#note-68? Perhaps mame (Yusuke Endoh) knows? If it is not decisive yet, that is fine. I just want to know what the developers have in mind at this point.

It is a literal and not a method call, so it should continue to work. It does in my branch and I believe mame's branch as well.

#74 - 08/16/2019 06:52 AM - mame (Yusuke Endoh)

Sorry for leaving this ticket untouched.

Eregon (Benoit Daloze) wrote:

> With Jeremy's proposal, I think there is no need to support non-Symbol keywords.
> I think in the case of ActiveRecord or Sequel's #where, the method doesn't need to accept keywords, it can just accept a Hash.

Yes indeed, but in my opinion, the behavior is just for compatibility. If there is no non-Symbol-key support, we need to continue to rely on the compatibility behavior to write even a new method that wants to accept both Symbol-key and non-Symbol key.

> This Hash might contain numbers or strings as keys, and as such doesn't feel like "keywords arguments" to me.

I agree somewhat. But, foo("key" => 42) actually looks a keyword argument, doesn't it?

But honestly I'm not so sure. jeremyevans0 (Jeremy Evans) what do you think?

#75 - 08/16/2019 06:55 AM - mame (Yusuke Endoh)

Let me reboot the discussion. The problems and some proposals are described in the slides:

https://docs.google.com/presentation/d/16rReiCVzUog3s5vV702LzcIFM2LNcX53AX8m5K8uCZw/edit?usp=sharing

My understanding is that Jeremy's proposal is the most prospect. It is:

- In principle, keyword arguments and non-keyword ones are separated.
  - You need to pass keyword arguments by foo(k: 42) or foo(**hash)
  - You need to accept keyword argument by def foo(k: 42) or def foo(**hash)
  - All other arguments are considered as non-keyword arguments; foo(hash) is not a keyword argument.
- However, it is allowed to pass keyword arguments to a method that does not accept keyword arguments. (Jeremy's compatibility layer)
- Non-symbol keys are allowed: def foo(**kw); end; foo("k" => 42)

```
# Jeremy's compatibility layer
def foo(opt = {})
  p opt
end
foo(k: 42) # {:k=>42}
```

Assuming that Ruby 3.0 will pick up Jeremy's proposal, I'd like to discuss the semantics of Ruby 2.7. In principle, it should:

1) Warns the behavior that won't work after 3.0.
2) Reasonably run a code that is valid in 3.0 for gradual migration.

However, it is very tough to discuss it on paper. (Actually matz, akr, and I spent a few month to discuss this issue.)

I think that Jeremy's branch is a great start. So, jeremyevans0 (Jeremy Evans) , how about merging it experimentally? And then, if we face some actual issues, we can discuss each of them. If you are not against, I'd like to propose the merge at the next dev-meeting.

#76 - 08/16/2019 02:36 PM - jeremyevans0 (Jeremy Evans)

mame (Yusuke Endoh) wrote:

> > This Hash might contain numbers or strings as keys, and as such doesn't feel like "keywords arguments" to me.
>
> I agree somewhat. But, foo("key" => 42) actually looks a keyword argument, doesn't it?
>
> But honestly I'm not so sure. jeremyevans0 (Jeremy Evans) what do you think?

I don't have a strong opinion about allowing non-Symbols for keyword hash keys.

With your original proposal, there is a definite need, since in order to get nice calling syntax, you want to allow the conversion of all def a(opts={}) to def a(**opts). With my proposal, you can have a nice calling syntax with both def a(opts={}) and def a(**opts), so I don't think there is need. If you want to support non-Symbol keys, you would just continue to use def a(opts={}).

That doesn't imply it is bad to allow non-Symbol keys for def a(**opts). It does change the original purpose of keyword arguments, but it adds flexibility, and would allow for easier switching between def a(opts={}) and def a(**opts).

mame (Yusuke Endoh) wrote:

> I think that Jeremy's branch is a great start. So, jeremyevans0 (Jeremy Evans) , how about merging it experimentally? And then, if we face some actual issues, we can discuss each of them. If you are not against, I'd like to propose the merge at the next dev-meeting.

I am in favor of the merge and eager to see matz's decision. I'll rebase my branch against master today to make it easier for people to comparison test.

#77 - 08/16/2019 11:05 PM - jeremyevans0 (Jeremy Evans)

jeremyevans0 (Jeremy Evans) wrote:

> I am in favor of the merge and eager to see matz's decision. I'll rebase my branch against master today to make it easier for people to comparison test.

I've rebased my branch against master: https://github.com/jeremyevans/ruby/tree/keyword-argument-separation

After rebasing against master, I ran the specs on my branch. I guess I didn't run the specs previously (only the tests), because the specs found a backwards compatibility issue on my branch:

```
def m(a=1, b:) [a, b] end

m("a" => 1, b: 2)
# 2.6: [{"a"=>1}, 2]
# my branch: ArgumentError (unknown keyword: "a")
```

While my branch has the behavior we want in 3.0, my branch currently does not have a good transition path for this case. Since this worked in 2.6, we probably want a warning and the same behavior, correct? I think that means adding a keyword argument to positional hash split. My branch already implements the positional hash to keyword argument split, but has not implemented the reverse split yet. I will try to implement that before the next developer meeting.

Note that there is still an expected behavior change when using keyword splats:

```
def b(a=1, **b) [a, b] end
p b('a'=>1, :b=>3)
# 2.6: [{"a"=>1}, {:b=>3}]
# my branch: [1, {"a"=>1, :b=>3}]
```

I still believe this change is reasonable, because it doesn't make sense to have a warning in 2.7 for something that will be valid in 3.0.

#78 - 08/17/2019 12:33 AM - mame (Yusuke Endoh)

jeremyevans0 (Jeremy Evans) wrote:

> jeremyevans0 (Jeremy Evans) wrote:
>
> > I am in favor of the merge and eager to see matz's decision. I'll rebase my branch against master today to make it easier for people to comparison test.
>
> I've rebased my branch against master: https://github.com/jeremyevans/ruby/tree/keyword-argument-separation

Okay, I'll talk with matz. And so quick rebase, thanks!

> Since this worked in 2.6, we probably want a warning and the same behavior, correct?

That's the toughest decision. Ruby 2.7 should definitely run all "reasonable" programs that were valid in 2.6. But I don't think that Ruby 2.7 has to be 100% compatible with 2.6. Rather, Ruby 2.7 should run all "reasonable" programs that will be valid in 3.0. Otherwise, people cannot make their programs ready for 3.0.

So, Ruby 2.7 need to allow both 2.6-valid code and 3.0-valid code as far as possible. (This is the reason why I introduced the dirty rb_no_keyword_hash that was a trick to somehow run both 2.6-valid code and 3.0-ready code.) If we cannot find a great solution to allow both 2.6 and 3.0 programs simultaneously, we need to discuss each incompatibility case.

The particular case you showed (def m(a=1, b:) [a, b] end; m("a" => 1, b: 2)) is acceptable, I believe. Luckily, we have an evidence. Ruby 2.6.0 once prohibited mixing non-Symbol-key and Symbol-key:

```
$ ./local/bin/ruby -ve '
def m(a=1, b:) [a, b] end
m("a" => 1, b: 2)
'
ruby 2.6.0p0 (2018-12-25 revision 66547) [x86_64-linux]
Traceback (most recent call last):
-e:3:in `<main>': non-symbol key in keyword arguments: "a" (ArgumentError)
```

The behavior was reverted at Ruby 2.6.1 because it turned out to do more harm than good: it prevents Ruby 3.0 from allowing non-symbol key. Anyway, I didn't see any complaint about the 2.6.0 behavior change, so I guess that very few people mix non-Symbol-key and Symbol-key.

#79 - 08/17/2019 01:44 AM - jeremyevans0 (Jeremy Evans)

I think the changes to support the keyword to last positional hash are minimal. I'm testing a patch now that provides the following behavior:

```
def a(a=1, b:2) [a, b] end;

a('a'=>1, :b=>3)
# 2.6: => [{"a"=>1}, 3]
# my branch: warning: The last argument for `a' (defined at t/t2.rb:1) is split into positional and keyword pa
rameters
#            =>[{"a"=>1}, 3]

a('a'=>1, 'b'=>3)
# 2.6: => [{"a"=>1, "b"=>3}, 2]
# my branch: warning: The keyword argument for `a' (defined at t/t2.rb:1) is passed as the last hash parameter
#            => [{"a"=>1, "b"=>3}, 2]
```

The logic is: if the last argument is a keyword hash that does not contain all symbols, and the method accepts keyword arguments but not a keyword splat, split the keyword hash. If the hash is split, emit the split warning. If there are no elements in the keyword hash (all were transferred to the positional hash), then emit the keyword to last hash warning.

I need some more time to test this and make sure it doesn't emit false positive warnings, but I should be able to push it to my branch next week.

One thing I learned during this work was that the VM currently treats the following exactly the same:

```
a('a'=>1, :b=>1)
a(**{'a'=>1, :b=>1})
```

Basically, if the keyword parameter contains a non-Symbol key, the entire keyword parameter is treated as a splatted hash (VM_CALL_KW_SPLAT), instead of being treated as a normal keyword call (VM_CALL_KWARG).

#80 - 08/20/2019 07:40 PM - jeremyevans0 (Jeremy Evans)

I've completed work on my keyword-argument-separation branch, and now make check passes without any emitted warnings in my environment. To get some early CI testing of this, I have submitted it as a GitHub pull request: https://github.com/ruby/ruby/pull/2395

#81 - 08/20/2019 08:21 PM - jeremyevans0 (Jeremy Evans)

- Related to Bug #10293: splatting an empty hash in a method invocation sends an argument to the method (should send nothing) added

#82 - 08/29/2019 03:26 AM - Dan0042 (Daniel DeLorme)

I tried testing to see what behavior changes are introduced via this. This is what I found so far.

```
def foo(*c, **f); p [c,f]; end; foo(9=>9, f:12)
before: [[{9=>9}], {:f=>12}]
after:  [[], {9=>9, :f=>12}]
```

Good bugfix I think

```
def foo(**f); p [f]; end
foo({})
warning: The last argument for `foo' (defined at kwtest.rb:4110) is used as the keyword parameter
[{}]
#but if it was not used as keyword parameter we'd get "wrong number of arguments" error

def foo(b=5, e:15); p [b,e]; end
foo(9=>9)
warning: The keyword argument for `foo' (defined at kwtest.rb:73776) is passed as the last hash parameter
[{9=>9}, 15]
#but if it was not passed as the last hash parameter we'd get "unknown keyword" error

def foo(a, b=5, d:); p [a,b,d]; end
foo(0, 9=>9, d:10)
warning: The last argument for `foo' (defined at kwtest.rb:213108) is split into positional and keyword parame
ters
[0, {9=>9}, 10]
#but if it wasn't split we'd get "unknown keyword" error

def foo(*a, **o); p [a,o]; end
a = [1,2,3,{x:1}]
foo(*a)
(irb):6: warning: The last argument for `foo' (defined at (irb):1) is used as the keyword parameter
[[1, 2, 3], {:x=>1}]
```

Am I correct in assuming these warnings will be errors in ruby 3?

But it seems to me there's no need to raise an error since none of those have any ambiguity. Wouldn't it be better here for ruby to just "do the right thing"?

The last one in particular seems problematic. Using *args to pass arguments to another method is a common pattern. All proxy objects do that. If the receiver happens to have keyword arguments and it breaks, that means we'd be required to always specify *args, **opts for all proxying. That's going to break a lot of code.

#83 - 08/29/2019 03:53 AM - jeremyevans0 (Jeremy Evans)

Dan0042 (Daniel DeLorme) wrote:

```
def foo(**f); p [f]; end
foo({})
warning: The last argument for `foo' (defined at kwtest.rb:4110) is used as the keyword parameter
[{}]
#but if it was not used as keyword parameter we'd get "wrong number of arguments" error
```

Correct. That is expected, as the method takes no arguments, only keyword parameters.

```
def foo(b=5, e:15); p [b,e]; end
foo(9=>9)
warning: The keyword argument for `foo' (defined at kwtest.rb:73776) is passed as the last hash parameter
[{9=>9}, 15]
#but if it was not passed as the last hash parameter we'd get "unknown keyword" error
```

Correct. This is expected, as the method does not support a keyword parameter for 9.

```
def foo(a, b=5, d:); p [a,b,d]; end
foo(0, 9=>9, d:10)
warning: The last argument for `foo' (defined at kwtest.rb:213108) is split into positional and keyword pa
rameters
[0, {9=>9}, 10]
#but if it wasn't split we'd get "unknown keyword" error
```

Correct. This is expected, as the method does not support a keyword parameter for 9.

```
def foo(*a, **o); p [a,o]; end
a = [1,2,3,{x:1}]
foo(*a)
(irb):6: warning: The last argument for `foo' (defined at (irb):1) is used as the keyword parameter
[[1, 2, 3], {:x=>1}]
```

This warning is expected, because behavior will change in Ruby 3 to return [[1, 2, 3, {:x=>1}], {}]

> Am I correct in assuming these warnings will be errors in ruby 3?

Not the last one, since it a behavior change but would not cause an error in the example given.

> But it seems to me there's no need to raise an error since none of those have any ambiguity. Wouldn't it be better here for ruby to just "do the right thing"?

That's what Ruby has been trying to do for a long time. You probably want to see all of the referenced issues to see why it isn't a good idea, or at least understand what problems it causes.

> The last one in particular seems problematic. Using *args to pass arguments to another method is a common pattern. All proxy objects do that. If the receiver happens to have keyword arguments and it breaks, that means we'd be required to always specify *args, **opts for all proxying. That's going to break a lot of code.

This is correct. Generic method forwarding will require *args, **opts, &block in Ruby 3. We are aware that this change will break forwarding to methods that accept keyword arguments if you just use *args, &block. However, it should not break forwarding to methods that do not accept keyword arguments.

#84 - 08/30/2019 09:11 PM - jeremyevans0 (Jeremy Evans)

- Status changed from Open to Closed

Matz approved this feature at the developer meeting a couple days ago, and it was merged earlier today (https://github.com/ruby/ruby/compare/b0a291f6f6a5834fd84807eb48be906ade429871...b5b3afadfab4072f55320075ccac6afe333a140c)

#85 - 08/31/2019 03:06 AM - Dan0042 (Daniel DeLorme)

I was trying to think of a way to have better backward compatibility for generic forwarding, and I think I managed to stumble on a relatively clean way of doing it.

Basically the idea is to use a subclass of Hash for **kw

```
class KwHash < Hash
  def none?
    #similar to akr's keyword_given?
  end
  def dup
    Hash.new(self)
  end
end
def foo(a=nil, **kw); kw.none?; end
foo()      #=> true
foo(**{})  #=> false
foo(k:1)   #=> false
foo({k:1}) #=> true (at least in ruby 3)
```

So when the kwarg is demoted to LPH (last positional hash), we still know it was originally a kwarg. And so if forwarded to another method, the LPH is actually a KwHash so we know it's safe to promote to kwarg.

```
def foo(*args, **kw)
  #if args.last is KwHash, two possibilities:
  #1. kw=args.pop if kw.none? (definitely safe)
  #2. merge with kw (I *think* it is conceptually sound)
  [args, kw]
end
def bar(*args)
  foo(*args)
end
foo(x:1) #=> [[], {:x=>1}]
bar(x:1) #=> [[], {:x=>1}] even in ruby 3
```

So far the only edge case I can think of is related to keyword extension:

```
def my_puts(*args, out: $stdout, **kw)
  args << kw unless kw.empty?
  out.puts(*args) #args.last is KwHash, so it will be promoted to kwarg if out.puts supports keyword args
end
#behavior above may be acceptable, or:
def my_puts(*args, out: $stdout, **kw)
  args << kw.dup unless kw.empty? #dup to convert KwHash to Hash
  out.puts(*args)
end
```

I think in addition to the advantages for forwarding, it feels nicely object-oriented to have the kwarg be a different class from Hash; it mirrors the separation of position and keyword arguments.

Kindly awaiting your thoughts.

#86 - 09/02/2019 04:35 AM - jeremyevans0 (Jeremy Evans)

- Related to Bug #15753: unknown keyword when passing an hash to a method that accepts a default argument and a named argument added

#87 - 09/02/2019 09:43 PM - jeremyevans0 (Jeremy Evans)

Dan0042 (Daniel DeLorme) wrote:

> So far the only edge case I can think of is related to keyword extension:
>
> ```
> def my_puts(*args, out: $stdout, **kw)
>   args << kw unless kw.empty?
>   out.puts(*args) #args.last is KwHash, so it will be promoted to kwarg if out.puts supports keyword args
> end
> #behavior above may be acceptable, or:
> def my_puts(*args, out: $stdout, **kw)
>   args << kw.dup unless kw.empty? #dup to convert KwHash to Hash
>   out.puts(*args)
> end
> ```

There are probably many edge cases with this approach. Here's an example of one such edge case:

```
def foo(*args, **kw)
```

```
  [args, bar(kw)]
end

def bar(hash={}, skip: false)
  hash unless skip
end

# Ruby 2.6 behavior:
foo # [[], {}]
foo(:a=>1) # ArgumentError: unknown keyword: :a
```

Here, the intention is to pass the keyword arguments from one method as a positional hash to another method. This is one of the cases that currently breaks in 2.6, that will warn and break in 2.7, and that will be fixed in Ruby 3. With your approach, it will remain broken, since kw in foo will be implicitly converted to keyword arguments to bar.

This leads to the same types of problems that keyword argument separation is designed to prevent. Any form of automatic conversion of positional hashes to keyword arguments and vice versa is going to have corner cases like these.

I understand that keyword argument separation is going to require updating some code. It is not going to be fully backwards compatible for methods that accept keyword arguments. The good news is that if you don't use keyword arguments in your methods, the behavior will remain backwards compatible. Additionally, any cases that will break will be warned about in Ruby 2.7 so you can update the related code.

#88 - 09/04/2019 05:20 PM - Dan0042 (Daniel DeLorme)

jeremyevans0 (Jeremy Evans) First, thank you very much for taking your time to engage with me like this.

> Here, the intention is to pass the keyword arguments from one method as a positional hash to another method. This is one of the cases that currently breaks in 2.6, that will warn and break in 2.7, and that will be fixed in Ruby 3. With your approach, it will remain broken, since kw in foo will be implicitly converted to keyword arguments to bar.

I think in this case the 2.6 behavior is better. Because :a=>1 is specified without braces, it should ideally remain a keyword all the way down to bar. It should not become a Hash in foo without explicit conversion.

To my eyes, the intention signaled via bar(kw) in your example would be to pass-through the keyword arguments. Because kw is not a Hash but KwHash (provisional name). In order to pass it as positional argument you would need to first convert it to Hash via bar({**kw}) or such. The idea is that a KwHash can only be passed as a kwarg. No automatic conversion. Passing a KwHash kw to a method is strictly equivalent to **kw. After all the entire point of "Real keyword arguments" is to keep them distinct from the rest right? But syntax is not the only way to do that; this KwHash class would also be a way to achieve the same result. Even though bar(kw) may look like a positional argument, it's really a keyword argument, properly and fully differentiated from positional arguments via its class. I realize that's a fairly different interpretation than the current one but I believe it makes sense. Matz may like syntactical separation but I think he would remain open to other possibilities.

```
args = [1, 2, hash]
foo(*args) #=> args.last is Hash -> positional; warning in 2.7
args = [1, 2, **hash]
foo(*args) #=> args.last is KwHash -> keyword
```

I would even go as far as saying that with this KwHash, bar(kw1, 2, kw3, 4) must either raise an error or be equivalent to bar(2, 4, **kw1, **kw3), otherwise the separation of keyword and positional arguments doesn't hold. I realize this is not backward compatible, but it's the kind of incompatibility I'm ok with because it fixes incorrect semantics (if it was originally a kwarg it shouldn't suddenly be a Hash).

And on the receiver side, even if the kwarg is converted to positional argument because of your compatibility mode, it would still be a KwHash and behave as such unless *explicitly* converted to Hash. The positional/keyword separation is maintained even despite the compatibility mode.

I know that code speaks loudest so I would like to write a branch for this idea, but I'm too unfamiliar with the VM code. I wouldn't be able to write something in time to make it for review before the November code freeze. :-(

> I understand that keyword argument separation is going to require updating some code. It is not going to be fully backwards compatible for methods that accept keyword arguments. The good news is that if you don't use keyword arguments in your methods, the behavior will remain backwards compatible. Additionally, any cases that will break will be warned about in Ruby 2.7 so you can update the related code.

Updating some code in itself is not a problem at all. What makes me uncomfortable is that updating code in order to fix 2.7 warnings can result in code that is no longer compatible with 2.6. This now seems to be the only way to write correct forwarding code?

```
if RUBY_VERSION.to_f <= 2.6
  def method_missing(*a, &b)
    @x.send(*a, &b)
  end
else
  def method_missing(*a, **o, &b)
    @x.send(*a, **o, &b)
  end
end
```

If it was unavoidable then I'd just say that's the cost of progress. But I'm convinced it's avoidable.

Please understand that I'm not clinging to old behavior just as a knee-jerk reaction to change. I've taken your earlier words to heart and spent several hours reading this entire thread carefully as well as related tickets, digesting and pondering the information. So I think I've reached a pretty decent understanding. The current changes are obviously great and fix a lot of problems. It's just that adding keyword separation via class in addition to syntax allows to keep better backward compatibility with **stricter** keyword/positional separation, while still fixing all the issues related to the previous implementation. I think that's worth serious consideration.

And as a bonus, it even becomes easier to optimize the KwHash implementation specifically for keyword arguments.

Thank you for your patience and forgive the verbosity; I find it hard to convey the nuance of my argument. This is my last post about the KwHash idea (unless you have questions :-). If I still can't gain the interest of the ruby maintainers with this... I guess we'll just have to go down the backward-incompatible route.

#89 - 09/04/2019 11:51 PM - jeremyevans0 (Jeremy Evans)

Dan0042 (Daniel DeLorme) wrote:

> jeremyevans0 (Jeremy Evans) First, thank you very much for taking your time to engage with me like this.
>
> > Here, the intention is to pass the keyword arguments from one method as a positional hash to another method. This is one of the cases that currently breaks in 2.6, that will warn and break in 2.7, and that will be fixed in Ruby 3. With your approach, it will remain broken, since kw in foo will be implicitly converted to keyword arguments to bar.
>
> I think in this case the 2.6 behavior is better. Because :a=>1 is specified without braces, it should ideally remain a keyword all the way down to bar. It should not become a Hash in foo without explicit conversion.
>
> To my eyes, the intention signaled via bar(kw) in your example would be to pass-through the keyword arguments. Because kw is not a Hash but KwHash (provisional name). In order to pass it as positional argument you would need to first convert it to Hash via bar({**kw}) or such. The idea is that a KwHash can only be passed as a kwarg. No automatic conversion. Passing a KwHash kw to a method is strictly equivalent to **kw. After all the entire point of "Real keyword arguments" is to keep them distinct from the rest right? But syntax is not the only way to do that; this KwHash class would also be a way to achieve the same result. Even though bar(kw) may look like a positional argument, it's really a keyword argument, properly and fully differentiated from positional arguments via its class. I realize that's a fairly different interpretation than the current one but I believe it makes sense. Matz may like syntactical separation but I think he would remain open to other possibilities.

I disagree that taking a keyword argument hash in one method and passing them as a positional argument to another method should force the argument to become keyword arguments in the other method. Conceptually, once a method has been entered, there is no longer a distinction between positional arguments and keyword arguments, they are all just local variables at that point. Your proposal attempts to introduce a distinction that does not and should not exist.

Here's a simple example showing undesired behavior with your approach:

```
class A
  # Same as Kernel#p, except do nothing if :skip keyword is present
  def p(*args, skip: false)
    super(*args) unless skip
  end

  def foo(*args, **kw)
    # In debug mode, prints method name, positional arguments, and keyword arguments
    p(:foo, *args, kw) if $DEBUG

    # do something
  end
end

# No problems
A.new.foo(1, 2, a: 1)

$DEBUG = true

# ArgumentError! (unknown keyword: :a)
# Even though the only purpose was to print out the arguments for debugging
A.new.foo(1, 2, a: 1)
```

```
    args = [1, 2, hash]
    foo(*args) #=> args.last is Hash -> positional; warning in 2.7
    args = [1, 2, **hash]
    foo(*args) #=> args.last is KwHash -> keyword
```

> I would even go as far as saying that with this KwHash, bar(kw1, 2, kw3, 4) must either raise an error or be equivalent to bar(2, 4, **kw1, **kw3), otherwise the separation of keyword and positional arguments doesn't hold. I realize this is not backward compatible, but it's the kind of incompatibility I'm ok with because it fixes incorrect semantics (if it was originally a kwarg it shouldn't suddenly be a Hash).

**hash in arrays is not done for keyword argument purposes (after all, there are no arguments). It is used to merge multiple hashes and literal keywords:

```
[1, a: 2, **{b: 3}, **{c: 4}]
# => [1, {:a=>2, :b=>3, :c=>4}]
```

So you are trying to introduce an idea ** in arrays as being for keywords arguments, when it has not been used for that in the past. The introduction of such behavior would result in additional backward incompatibility.

> And on the receiver side, even if the kwarg is converted to positional argument because of your compatibility mode, it would still be a KwHash and behave as such unless *explicitly* converted to Hash. The positional/keyword separation is maintained even despite the compatibility mode.
>
> I know that code speaks loudest so I would like to write a branch for this idea, but I'm too unfamiliar with the VM code. I wouldn't be able to write something in time to make it for review before the November code freeze. :-(

The keyword argument branch in this ticket was my first time working significantly in the VM code. mame posted his initial patch on March 18. I posted my initial patch based off his patch on March 25. It's the start of September, there is still time to work on an actual proposal with code if you are passionate about this change. The only person you need to convince is matz :) .

> I understand that keyword argument separation is going to require updating some code. It is not going to be fully backwards compatible for methods that accept keyword arguments. The good news is that if you don't use keyword arguments in your methods, the behavior will remain backwards compatible. Additionally, any cases that will break will be warned about in Ruby 2.7 so you can update the related code.

> Updating some code in itself is not a problem at all. What makes me uncomfortable is that updating code in order to fix 2.7 warnings can result in code that is no longer compatible with 2.6. This now seems to be the only way to write correct forwarding code?
>
> ```
> if RUBY_VERSION.to_f <= 2.6
>   def method_missing(*a, &b)
>     @x.send(*a, &b)
>   end
> else
>   def method_missing(*a, **o, &b)
>     @x.send(*a, **o, &b)
>   end
> end
> ```

You do not need to have two separate definitions of method_missing, unless you want to be backwards compatible with 1.9 (which doesn't support ** for keyword parameters). You should always use *a, **o, &b when forwarding. Example:

```
class B
  def initialize(x) @x = x end
  def method_missing(*a, **o, &b)
    @x.send(*a, **o, &b)
  end
end

class C
  def initialize(x) @x = x end
  def method_missing(*a, &b)
    @x.send(*a, &b)
  end
end

class D
  def method_missing(*a, **o, &b)
    [*a, o, b]
  end
end

b = B.new(D.new)
c = C.new(D.new)

b.a == c.a                 # true
b.a(1) == c.a(1)           # true
b.a(a: 1) == c.a(a: 1)     # true, c.a warns in 2.7
b.a({a: 1}) == c.a({a: 1}) # true, both b.a and c.a warn in 2.7
b.a({a: 1}, **(;{})) == c.a({a: 1}, **(;{})) # true, c.a warns in 2.7
```

For the case where b.a warns, you'll need to make changes for Ruby 3 to get the same behavior. However, you probably wouldn't want to make changes in this example, as it is pretty obvious you intend to pass a positional hash and not keywords.

> If it was unavoidable then I'd just say that's the cost of progress. But I'm convinced it's avoidable.
>
> Please understand that I'm not clinging to old behavior just as a knee-jerk reaction to change. I've taken your earlier words to heart and spent several hours reading this entire thread carefully as well as related tickets, digesting and pondering the information. So I think I've reached a pretty decent understanding. The current changes are obviously great and fix a lot of problems. It's just that adding keyword separation via class in addition to syntax allows to keep better backward compatibility with **stricter** keyword/positional separation, while still fixing all the issues related to the previous implementation. I think that's worth serious consideration.

I understand what you want and why you want it. You want def m(*a, &b) n(*a, &b) end to implicitly forward keyword arguments as keyword arguments, so you don't have to modify the related code. Unfortunately, that's not possible when separating keyword arguments, and trying to work around it with separate classes causes more problems then it solves. You will need to switch the code to: def m(*a, **o, &b) n(*a, **o, &b) end.

Your proposal does not necessarily keep better backwards compatibility. By attempting to implicitly convert positional arguments to keyword parameters, it introduces new backwards compatibility issues. Whether the backwards compatibility issues it introduces are better or worse than the behavior currently planned for Ruby 3 is subjective, but I think your proposal would make hurt backwards compatibility more than it helps.

Your proposal does not result in stricter keyword/positional separation. It makes the separation less strict by using implicit conversion of positional argument to keyword argument.

Your proposal does not fix all of the issues with the previous implementation. It enables you to not have to modify some code, at the expense of opening a pandora's box of possible issues, such as the example given above.

#90 - 09/05/2019 01:54 AM - Dan0042 (Daniel DeLorme)

Obviously we have different ideas of "what conceptually should be" and must agree to disagree. I'd really like to know what matz thinks of all this though, if his reaction is "hmm?" or "yuck!" ... Oh well.

> Your proposal does not result in stricter keyword/positional separation. It makes the separation less strict by using implicit conversion of positional argument to keyword argument.

I'm afraid I utterly failed to communicate my point. :-(

Thank you for at least taking me seriously. I may yet try to challenge the VM!

#91 - 09/05/2019 01:57 AM - Dan0042 (Daniel DeLorme)

jeremyevans0 (Jeremy Evans) wrote:

> You do not need to have two separate definitions of method_missing, unless you want to be backwards compatible with 1.9 (which doesn't support ** for keyword parameters). You should always use *a, **o, &b when forwarding. Example:

Counter-example:

```
class B
  def initialize(x) @x = x end
  def method_missing(*a, **o, &b)
    @x.send(*a, **o, &b)
  end
end

class C
  def initialize(x) @x = x end
  def method_missing(*a, &b)
    @x.send(*a, &b)
  end
end

class D
  def method_missing(*a, &b)
    [*a, b]
  end
end

b = B.new(D.new)
c = C.new(D.new)

b.a == c.a                                  # false in 2.6 / true in 2.7
b.a(1) == c.a(1)                            # false in 2.6 / true in 2.7
b.a(a: 1) == c.a(a: 1)                      # true / b.a warns in 2.7
b.a({a: 1}) == c.a({a: 1})                  # true / b.a warns in 2.7
b.a({a: 1}, **(;{})) == c.a({a: 1}, **(;{})) # true
```

I don't know how to have this work in both 2.6 and 2.7 without checking RUBY_VERSION

#92 - 09/29/2019 06:27 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

#93 - 10/14/2019 05:55 PM - Eregon (Benoit Daloze)

*- Related to Misc #16157: What is the correct and *portable* way to do generic delegation? added*

#94 - 10/23/2019 05:59 PM - bughit (bug hit)

The original intent seems to have been to separate named args from hashes and make them a distinct language feature with more consistent, easier to understand syntax and semantics. However, how does the following fit with that goal:

```
def foo(a: nil, **args)
  args
end

foo(a: 1, 'a' => 2, 1 => 3, nil => 4, true => 5, Object.new => 6)
```

This invocation of foo should not be legal. 'a', 1, nil, true, Object.new, are valid keys of a hash but they are not names of arguments, with which they are conflated.

#95 - 11/08/2019 07:01 AM - sam.saffron (Sam Saffron)

Matz said:

> If we made the decision, we will make it warn you first for a year or two before the actual change.

I have just been testing Discourse with 2.7, it boots and it appears faster than 2.6 which is great.

But.... this deprecation as implemented is brutal, in fact I would say a breaking change in many ways is less painful cause at least if forces you to fix stuff up right away.

As it stands the deprecation warning writes to STDERR every time it hits a bad usage. Running our spec suite causes about 3 million duplicate lines of sorts to be written to STDERR.

Clearly tracking call sites in a hash may be a bit on the expensive side, but I think we should invest in only warning on first or maybe up to 100th occurrence, it gives people more time to fix. Warning every time is basically breaking Ruby.

I get that the "sweeping under carpet" means we need more accounting but on the upside it means less string generation just to output on STDERR.

Thoughts?

#96 - 11/08/2019 10:14 AM - mame (Yusuke Endoh)

Hi sam.saffron (Sam Saffron) , the issue you are talking about is being discussed in #16289.

It might be a good idea to limit up to 100th occurrence per call.

#97 - 11/08/2019 01:50 PM - Dan0042 (Daniel DeLorme)

> As it stands the deprecation warning writes to STDERR every time it hits a bad usage. Running our spec suite causes about 3 million duplicate lines of sorts to be written to STDERR.

Did you try counting how many unique warnings were generated? I'm interested in exactly how much backward incompatibility this is all causing. ruby tests.rb 2> >(sort -u|wc -l)

#98 - 11/08/2019 02:25 PM - mame (Yusuke Endoh)

Dan0042 (Daniel DeLorme) wrote:

> Did you try counting how many unique warnings were generated? I'm interested in exactly how much backward incompatibility this is all causing. ruby tests.rb 2> >(sort -u|wc -l)

Nitpicking, but note that the estimation is rough. One warning has two lines. And sometimes, fixing one code removes multiple different warnings. The result of wc -l does not exactly mean how many code fragment you need to tweak.

#99 - 12/03/2019 10:02 AM - mame (Yusuke Endoh)

I've written a draft of a note about this change in https://github.com/ruby/www.ruby-lang.org/pull/2293.

#100 - 12/03/2019 11:45 PM - jeremyevans0 (Jeremy Evans)

Shortly after the release of 2.7.0, I would like to merge the branch at https://github.com/jeremyevans/ruby/tree/r3, which fully separates positional and keyword arguments. I've been maintaining this branch for a couple months now and using it for testing. I rebased this branch against today's master, and will try to rebase it weekly for the next few weeks.

#101 - 12/06/2019 09:13 PM - koic (Koichi ITO)

I have a question because my understanding is not enough. A kwarg warning is displayed with the following code:

```
# example.rb
str = 'Hello, %<foo>s, %<bar>s'
ary = [foo: 'foo', bar: 'bar']

ary.each do |foo:, bar:|
  format(str, foo: foo, bar: bar)
end
```

```
% ruby -v
ruby 2.7.0dev (2019-12-06T16:28:20Z master dcf89b20d7) [x86_64-darwin17]
% ruby example.rb
example.rb:5: warning: The last argument is used as the keyword parameter
example.rb:5: warning: for method defined here; maybe ** should be added to the call?
```

As far as I know, assigning to a hash will solve it.

```
# example.rb
str = 'Hello, %<foo>s, %<bar>s'
ary = [foo: 'foo', bar: 'bar']

ary.each do |h|
  foo = h[:foo]
  bar = h[:bar]
  format(str, foo: foo, bar: bar)
end
```

Is there any other way to suppress this warning using block arguments?

#102 - 12/06/2019 09:26 PM - jeremyevans0 (Jeremy Evans)

koic (Koichi ITO) wrote:

> I have a question because my understanding is not enough. A kwarg warning is displayed with the following code:
>
> ```
> # example.rb
> str = 'Hello, %<foo>s, %<bar>s'
> ary = [foo: 'foo', bar: 'bar']
>
> ary.each do |foo:, bar:|
>   format(str, foo: foo, bar: bar)
> end
> ```
>
> ```
> % ruby -v
> ruby 2.7.0dev (2019-12-06T16:28:20Z master dcf89b20d7) [x86_64-darwin17]
> % ruby example.rb
> example.rb:5: warning: The last argument is used as the keyword parameter
> example.rb:5: warning: for method defined here; maybe ** should be added to the call?
> ```
>
> As far as I know, assigning to a hash will solve it.
>
> ```
> # example.rb
> str = 'Hello, %<foo>s, %<bar>s'
> ary = [foo: 'foo', bar: 'bar']
>
> ary.each do |h|
>   foo = h[:foo]
>   bar = h[:bar]
>   format(str, foo: foo, bar: bar)
> end
> ```
>
> Is there any other way to suppress this warning using block arguments?

No. This is expected, as ary = [foo: 'foo', bar: 'bar'] is short for ary = [{foo: 'foo', bar: 'bar'}]. So the Array#each block is yielded a Hash, not keywords, which triggers the warning if the block accepts keywords.

The maybe ** should be added to the call does not apply in this case as the call happens internally, it's not something the user has control over.

#103 - 12/06/2019 10:44 PM - zverok (Victor Shepelev)

So, there is no way to use block's keyword arguments to unpack a hash with symbol keys when passing it to the block?.. That's very disruptive change, this technique is super-useful when working with complex structured data :(

#104 - 12/06/2019 10:59 PM - jeremyevans0 (Jeremy Evans)

zverok (Victor Shepelev) wrote:

> So, there is no way to use block's keyword arguments to unpack a hash with symbol keys when passing it to the block?.. That's very disruptive change, this technique is super-useful when working with complex structured data :(

Block calls are treated just like any other method call, and you would generally fix the keyword argument separation issue on the caller side:

```
def foo(h)
  yield **h
end

foo({foo: 1, bar: 2}) do |foo:, bar:|
  # foo = 1, bar = 2
end
```

The issue in the case shown by koic (Koichi ITO) is that he does not control the caller. Switching from keyword arguments to Hash#fetch calls is probably easiest in his case. He could also override the each method for the array object, but that is probably not a good idea unless there were many cases where you need this and you could use a separate class for it.

I think everyone agrees that keyword argument separation is disruptive, but it is no more disruptive to blocks than it is to method calls in general.

#105 - 12/06/2019 11:14 PM - zverok (Victor Shepelev)

> I think everyone agrees that keyword argument separation is disruptive, but it is no more disruptive to blocks than it is to method calls in general.

(inb4, I am obviously not saying "the change is bad, undo! undo!" Just trying to think out loud/provide some additional things to consider.)

The problem with blocks is indeed Array/Enumerable (and similar code). I am not sure how widespread this approach is, but in our production code (lots of small hashes, too many and to short-living to wrap in objects), we frequently do this:

```
words
  .select { |paragraph_id:, text:, **| paragraph_id > first_paragraph_id && !text.include?('foo') }
  .map { |text:, timestamp:, **| {at: timestamp.to_i, content: "#{text} [#{timestamp}]"} }
  .group_by { |at:, **| at % 1000 }

# Or even:
large_config_hash.then { |this_value:, that_value: DEFAULT, **| do_something(this_value, that_value) }
```

This allows constructing readable chains of expressive transformations, which Hash#fetch/Hash#[] would make arguably much less clear.

It seems keyword argument separation just clearly prohibits this technique without any alternative (of the same expressiveness).

#106 - 12/07/2019 12:19 AM - jeremyevans0 (Jeremy Evans)

zverok (Victor Shepelev) wrote:

> The problem with blocks is indeed Array/Enumerable (and similar code). I am not sure how widespread this approach is, but in our production code (lots of small hashes, too many and to short-living to wrap in objects), we frequently do this:
>
> ```
> words
>   .select { |paragraph_id:, text:, **| paragraph_id > first_paragraph_id && !text.include?('foo') }
>   .map { |text:, timestamp:, **| {at: timestamp.to_i, content: "#{text} [#{timestamp}]"} }
>   .group_by { |at:, **| at % 1000 }
>
> # Or even:
> large_config_hash.then { |this_value:, that_value: DEFAULT, **| do_something(this_value, that_value) }
> ```
>
> This allows constructing readable chains of expressive transformations, which Hash#fetch/Hash#[] would make arguably much less clear.

> It seems keyword argument separation just clearly prohibits this technique without any alternative (of the same expressiveness).

You are correct that this style will no longer be supported without modification or overriding of Array/Enumerable methods. I think switching to a Hash#fetch/Hash#[] approach would not make the code less clear or less expressive, but that is subjective. A Hash#fetch/Hash#[] approach is certainly going to be faster, saving at least one hash allocation per block call.

In most cases a Hash#[] approach will be more concise since you don't have to repeat the keys. For example:

```
.select { |h| h[:paragraph_id] > 0 && !h[:text].include?('foo') }
.map { |h| {at: h[:timestamp].to_i, content: "#{h[:text]} [#{h[:timestamp]}]"} }
.group_by { |h| h[:at] % 1000 }
```

The Hash#[] approach reduces the character count in each line:

- select: 80 -> 65
- map: 88 -> 80
- group_by : 33 -> 31

A Hash#fetch approach will probably be longer, and for mandatory keywords, that would be a more accurate translation.

Keyword arguments were introduced to Ruby to make API design more flexible, as mentioned in the Ruby 2.0.0 release announcement. They were not intended as a hack to extract data from hashes. We actually have a new hack for extracting data from hashes:

```
.select { |h| h in {paragraph_id: paragraph_id, text: text}; paragraph_id > 0 && !text.include?('foo') }
.map { |h| h in {text: text, timestamp: timestamp}; {at: timestamp.to_i, content: "#{text} [#{timestamp}]"} }
.group_by { |h| h in {at: at}; at % 1000 }
```

It is still experimental, though. :)

#107 - 12/08/2019 11:22 AM - zverok (Victor Shepelev)

> In most cases a Hash#[] approach will be more concise since you don't have to repeat the keys. For example:

```
.select { |h| h[:paragraph_id] > 0 && !h[:text].include?('foo') }
.map { |h| {at: h[:timestamp].to_i, content: "#{h[:text]} [#{h[:timestamp]}]"} }
.group_by { |h| h[:at] % 1000 }
```

Doesn't it look a lot like "pre-keyword args" method for you? For me, it does :)

The thing with "deconstruction" is not character economy (it is almost never the thing, unless you are playing code golf competition), but concepts economy. With "my" version, you are specifying what the block expects in block definition. Add to the code cases like "parameter used several times", "parameter is required", "parameter has non-nil default value", and "just a hash" version becomes total mess while keyword args one stays clear and readable. And, as for me, it is not a "hack" or "side-effect of keyword args mess", but one of the primary features.

This reasoning is totally in line with "why we need keyword arguments (for methods)". The thing is, yes, with blocks most of the time you don't have control over the caller, and also, lot of the time block is passed to Enumerable or other core methods (tap and then, File.open etc.).

I'd say (just a noise in the air, I don't expect something would be changed 3 weeks before final release, nor I am optimistic about my voice being heard for upcoming 3.0) that the same way non-lambda proc has "implicit (arrays) unpacking" as a part of its definition, probably "hash unpacking into keyword args" (probably with a strict boundary limitation: like, only if proc accepts ONLY keyword args, and value passed is ONLY a hash) should be part of it.

#108 - 12/08/2019 10:26 PM - decuplet (Nikita Shilnikov)

With pattern matching added I would expect the next step to be "advanced" argument destructuring, i.e. applying PM for method arguments. For me, this seems to be a fair replacement for keyword unpacking.

```
.select { |{paragraph_id:, text:}| ... }
```

I know it's a story for another feature request but let's first wait for 2.7.

#109 - 12/09/2019 03:47 AM - Dan0042 (Daniel DeLorme)

koic (Koichi ITO), would this not be good enough for your case?

```
str = 'Hello, %<foo>s, %<bar>s'
ary = [foo: 'foo', bar: 'bar']

ary.each do |h|
  format(str, **h)
end
```

But I have to say this would not even have been an issue if using class-based keyword arguments like I had suggested. Then ary[0] would have been a KwHash and the keyword semantics would naturally have flowed through to the block. I'm sorry to beat on a dead horse but it's so frustrating that I can't help myself :-(

> I think everyone agrees that keyword argument separation is disruptive, but it is no more disruptive to blocks than it is to method calls in general.

Yeah, it's pretty disruptive. To the extent that imho the cost is 10x the benefit. But oh well, dead horses, ships that sailed...

But more to the point, I think this example shows keyword separation is actually more disruptive to blocks than to method calls in general. Because the method that yields to the block is often/usually out of our control. I find this is quite similar to the situation with delegation, in the sense that there's an intermediary that used to just pass through the data to the intended destination, but now this doesn't work anymore. With delegation it was sender->delegator->target, with blocks it's object->method->block. With delegation it's easy enough to fix by adding **kw or ruby2_keywords, but with blocks it looks like the only choice is to **restructure the code**. That's a fair amount more disruptive than the simple find-and-replace operations we were seeing previously.

#110 - 12/11/2019 09:25 AM - koic (Koichi ITO)

Thank you very much. It is solved by **h.
However, in the following complicated case, it seems difficult to solve using **h. This is an interesting use case :-)

```
def do_something(*args, &block)
  yield 'yield_self', {expected: 'then'}
end

do_something do |code, expected:, use: expected, instead_of: code|
  puts "code:       #{code}"
  puts "expected:   #{expected}"
  puts "use:        #{use}"
  puts "instead_of: #{instead_of}"
end
```

```
% ruby exmaple.rb
/tmp/example.rb:2: warning: The last argument is used as the keyword parameter
/tmp/example.rb:5: warning: for `false' defined here; maybe ** should be added to the call?
code:       yield_self
expected:   then
use:        then
instead_of: yield_self
```

I think it will be solved by converting from hash to kwarg. But it can be complicated to solve when implemented by several different libraries.

I met this usage in real-world code. I show it as a use case with block arguments.
https://github.com/rubocop-hq/rubocop/blob/v0.77.0/spec/rubocop/cop/style/numeric_predicate_spec.rb#L12

#111 - 12/11/2019 02:16 PM - Dan0042 (Daniel DeLorme)

This can be solved by refactoring, but yeah it's much more complicated than just adding ** to the appropriate places :-(

```
def do_something(*args, &block)
  yield 'yield_self', {expected: 'then'}
end

def print_something(code, expected:, use: expected, instead_of: code)
  puts "code:       #{code}"
  puts "expected:   #{expected}"
  puts "use:        #{use}"
  puts "instead_of: #{instead_of}"
end

do_something do |code, h|
  print_something(code, **h)
end
```

#112 - 12/11/2019 03:29 PM - jeremyevans0 (Jeremy Evans)

koic (Koichi ITO) wrote:

> Thank you very much. It is solved by **h.
> However, in the following complicated case, it seems difficult to solve using **h. This is an interesting use case :-)
>
> ```
> def do_something(*args, &block)
>   yield 'yield_self', {expected: 'then'}
> end
> ```

```
do_something do |code, expected:, use: expected, instead_of: code|
  puts "code:       #{code}"
  puts "expected:   #{expected}"
  puts "use:        #{use}"
  puts "instead_of: #{instead_of}"
end
```

Unless I'm missing something, it should be sufficient to drop the braces in the yield, so you are calling the block with keywords instead of a hash:

```
def do_something(*args, &block)
  yield 'yield_self', expected: 'then'
end
```

This will still work correctly if the block accepts a hash argument instead of keywords.

#113 - 12/13/2019 05:08 PM - bughit (bug hit)

So after this improvement, nonsense like this:

```
foo(a: 1, 'a' => 2, 1 => 3, nil => 4, true => 5, Object.new => 6)
```

becomes legal

and useful functionality like this destructuring example

```
words
  .select { |paragraph_id:, text:, **| paragraph_id > first_paragraph_id && !text.include?('foo') }
  .map { |text:, timestamp:, **| {at: timestamp.to_i, content: "#{text} [#{timestamp}]"} }
  .group_by { |at:, **| at % 1000 }
```

becomes illegal.

Which programmers is this supposed to make happy, users or implementers? I can't imagine many users will be happy. If you're going to take away whatever rudimentary makeshift destructuring ruby had, then implement proper first-class destructuring. And if you're going to break compat, don't enable nonsense like foo(a: 1, 'a' => 2, 1 => 3, nil => 4, true => 5, Object.new => 6), these are not "Real keyword arguments", which is the title of this issue.

#114 - 12/13/2019 05:45 PM - jeremyevans0 (Jeremy Evans)

bughit (bug hit) wrote:

> So after this improvement, nonsense like this:
>
> ```
> foo(a: 1, 'a' => 2, 1 => 3, nil => 4, true => 5, Object.new => 6)
> ```
>
> becomes legal

That was already legal if foo did not accept keyword arguments. :)

As keyword argument splats have always been just plain hashes, and not a separate type, it makes sense that they can handle any keys a normal hash would handle. I'll admit that this stretches the definition of "keyword arguments", but Ruby has always been more pragmatic than dogmatic.

> and useful functionality like this destructuring example
>
> ```
> words
>   .select { |paragraph_id:, text:, **| paragraph_id > first_paragraph_id && !text.include?('foo') }
>   .map { |text:, timestamp:, **| {at: timestamp.to_i, content: "#{text} [#{timestamp}]"} }
>   .group_by { |at:, **| at % 1000 }
> ```
>
> becomes illegal.

One person's "useful destructuring" is another person's "bad hack". Keyword arguments were not intended for destructuring, they only did so for backwards compatibility with existing callers passing hashes. That was determined to be a mistake, due to the problems it caused.

Also, if you want the above to work, it is possible, you just need to make words be an object where blocks for those methods are called with keywords:

```
module KeywordEnumerable
  %w'select map group_by'.each do |meth|
    define_method(meth) do |&block|
      ret = super() do |h|
        block.call(**h)
```

```
      end
      ret.extend(KeywordEnumerable) if ret.is_a?(Array)
      ret
    end
  end
end
words.extend(KeywordEnumerable)
```

> Which programmers is this supposed to make happy, users or implementers? I can't imagine many users will be happy.

In the long term, both. In the beginning, many users will be happy, and many users will be unhappy. That's true of most changes when you have many users. If you look at all of the bugs we were able to close due to separating keyword arguments, that should be an indication of users who will be happy the change was made.

Ignoring backwards compatibility, do you think the previous way of handling keyword arguments was better?

> If you're going to take away whatever rudimentary makeshift destructuring ruby had, then implement proper first-class destructuring.

That's what pattern matching allows. Still experimental in 2.7, though.

Also, the previous "rudimentary makeshift destructuring" is still possible, but you have to explicitly double splat the hash, it is no longer done implicitly.

#115 - 12/13/2019 07:58 PM - bughit (bug hit)

> That was already legal if foo did not accept keyword arguments. :)

my full example was this

```
def foo(a: nil, **args)
  args
end

foo(a: 1, 'a' => 2, 1 => 3, nil => 4, true => 5, Object.new => 6)
```

This is not legal in 2.6 and will be in 2.7, somehow in the name of "real keyword arguments"

> One person's "useful destructuring" is another person's "bad hack".

destructuring itself can not be called a hack, it's a feature of many languages ruby competes with and will likely make it back to ruby.

> Keyword arguments were not intended for destructuring

It is irrelevant what the intent was, it was present and was useful and is therefore an unwelcome breaking change, unless first-class destructuring is added.

> Ignoring backwards compatibility, do you think the previous way of handling keyword arguments was better?

I did not even suggest reverting it. I said make it even more "real" by blocking foo(a: 1, 'a' => 2, 1 => 3, nil => 4, true => 5, Object.new => 6) and provide first-class hash param destructuring when taking away the current one, in 3.0.

#116 - 12/20/2019 07:50 AM - matz (Yukihiro Matsumoto)

jeremyevans0 (Jeremy Evans) It's OK to merge the patch to the master after the 2.7 release, to see how big the influence is.

Matz.

#117 - 12/20/2019 02:54 PM - nobu (Nobuyoshi Nakada)

- Target version changed from 3.0 to 36

#118 - 09/29/2020 03:37 AM - hsbt (Hiroshi SHIBATA)

- Target version changed from 36 to 3.0

Files

vm_args.diff	4.19 KB	03/25/2019	jeremyevans0 (Jeremy Evans)