Ruby trunk - Feature #14183

"Real" keyword argument

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

Status:	Open
Priority:	Normal
Assignee:	
Target version:	Next Major

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 trunk - Bug #8316: Can't pass hash to first positional argume...	**Closed**	
Related to Ruby trunk - Bug #9898: Keyword argument oddities	**Closed**	**06/03/2014**
Related to Ruby trunk - Bug #10856: Splat with empty keyword args gives unexp...	**Closed**	
Related to Ruby trunk - Bug #11236: inconsistent behavior using ** vs hash as...	**Open**	
Related to Ruby trunk - Bug #11967: Mixing kwargs with optional parameters ch...	**Rejected**	
Related to Ruby trunk - Bug #12717: Optional argument treated as kwarg	**Open**	
Related to Ruby trunk - Bug #12821: Object converted to Hash unexpectedly und...	**Closed**	
Related to Ruby trunk - Bug #13336: Default Parameters don't work	**Closed**	
Related to Ruby trunk - Bug #13647: Some weird behaviour with keyword arguments	**Open**	
Related to Ruby trunk - Bug #14130: Keyword arguments are ripped from the mid...	**Open**	
Related to Ruby trunk - Bug #15078: Hash splat of empty hash should not creat...	**Open**	

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.