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 #12104: Procs keyword arguments affect value of p...	**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	**Open**	
Related to Ruby trunk - Bug #13647: Some weird behaviour with keyword arguments	**Feedback**	
Related to Ruby trunk - Bug #14130: Keyword arguments are ripped from the mid...	**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.