Ruby master - Feature #16891

Restore Positional Argument to Keyword Conversion

05/14/2020 08:30 PM - jeremyevans0 (Jeremy Evans)

Status:	Rejected
Priority:	Normal
Assignee:	matz (Yukihiro Matsumoto)
Target version:	

Description

Based on feedback from Rails-core, Matz has decided to postpone full separation of keyword and positional arguments (see https://discuss.rubyonrails.org/t/new-2-7-3-0-keyword-argument-pain-point/74980). I think most Ruby-core developers were aware that fixing keyword argument separation issues can be fairly painful for libraries and applications that use keyword arguments extensively.

If we are going to discuss reversion or partial reversion of keyword argument separation, I think it would be best to address each aspect of keyword argument separation separately and decide how we want to handle each. There are 4 aspects I am aware of.

Positional Hash Argument to Keyword Argument Conversion

This is by far the most common issue when dealing with keyword argument separation. I would estimate 95% of keyword argument separation issues/warnings are related to this change, and if we are to restore any behavior it should be this behavior. Restoring positional hash argument to keyword argument conversion would allow the following code to continue to work:

```
def foo(key: 42); end
foo({key: 42})

def bar(*a)
  foo(*a)
end
bar({key: 42})
```

Attached is a patch that restores the behavior (see below).

Keyword Argument to Mandatory Positional Argument Conversion

This is probably the next most common issue with dealing with keyword argument separation. I'm on the fence about whether we should try to restore this behavior. Restoring keyword argument to mandatory positional argument conversion would allow the following code to continue to work:

```
def foo(h, **kw); end
foo(key: 42)
```

I don't think it would be difficult to restore this behavior, but I'm not sure it is worth doing so.

Splitting of Keyword or Positional Hash Argument During Conversion

In Ruby 2.0-2.6, Ruby would split a hash with both symbol keys and non-symbol keys into two hashes, one symbol keyed hash used for keyword arguments, and one non-symbol keyed hash to be passed as a positional argument.

The need for this splitting appears to be rare. I have heard the splitting was not matz's intended behavior originally. I know of only one user relying on this. mame (Yusuke Endoh)'s original keyword argument separation commit removed the splitting, but I added it back with deprecation warnings to ease the transition. There's actually two types of splitting, one type when splitting keyword argument to positional argument, and another type when splitting positional argument to keyword argument. Restoring the splitting would allow this code to continue to work:

```
def foo(h={}, key: 42); [h, key] end
foo("key" => 43, key: 42)
foo({"key" => 43, key: 42})
```

Due to the very low usage of this behavior and questionable semantics, I recommend we do not restore this behavior. Especially because the splitting no longer happens in 2.7 if arbitrary keywords are accepted by the method, since arbitrary keywords support

non-symbol keys in 2.7. However, I don't think it is that difficult to restore this support if we decide to do so.

Empty Keyword Argument Splat to Mandatory Positional Argument Conversion

In Ruby 2.7, empty keyword splats are removed and do not pass arguments. However, if the method requires a positional argument, for compatibility Ruby 2.7 passes a new empty positional hash argument. Restoring this behavior would allow the following code to continue to work:

```
h = {}
def foo(a) a end
foo(**h)
```

I think very little code relies on this feature, and it would very painful and invasive to restore it, so I recommend we do not restore this behavior.

Proposed behavior for Ruby 3.0

My preference would be to not change behavior at all. However, assuming we want to restore some backwards compatibility in regards to keyword argument separation, my proposal for Ruby 3.0 is to only restore positional argument to keyword argument conversion. We convert positional hash argument to keyword arguments if:

- Method accepts keyword arguments
- Method is not passed keyword arguments
- Method is passed a hash as the final positional argument
- Method is passed more arguments than the minimum required

If the method does not accept arbitrary keywords, we only convert positional argument to keyword argument if the above are true and the final positional hash is empty or contains a symbol key. It doesn't make sense to convert a positional hash to keywords if it has no symbol keys and the method does not accept arbitrary keywords.

When converting positional arguments to keyword arguments, we issue a warning in verbose mode in 3.0. In 3.1 or 3.2, we can switch the warning to non-verbose mode, before removing it and enforcing full separation in the following version.

I propose to keep the current master branch behavior other than this change.

Proposed behavior for Ruby 2.7.2

I propose we change the positional argument to keyword argument conversion warning to only issue in verbose mode and keep all other behavior the same.

Patch

Attached is a patch that implements my proposal. It does not include test or spec changes, as those will be extensive even for just the change I am proposing. I think we should agree on a proposal before working on changes to tests or specs.

Related issues:

Related to Ruby master - Feature #16378: Support leading arguments together w...	**Closed**

History

#1 - 05/15/2020 01:23 AM - Dan0042 (Daniel DeLorme)

IMHO there's one consideration that's more important than all others. The following code must work in ruby 2.8/3.0

```
def foo(h={}, **kw)
  [h, kw]
end
foo({x:1}) #=> [{x:1}, {}]
foo(x:1)   #=> [{}, {x:1}]
```

This is the #1 reason why keyword separation is needed. The fix for the numerous bug reports like #14130 etc. Without this we might as well go back to 2.6 behavior. And from what I can understand it seems the "Proposed behavior for Ruby 3.0" doesn't support this.

Also I believe there's a fifth aspect that deserves consideration for reverting:

Non-Symbol Keyword Argument

Previously one could trust that foo(1=>2) would be treated as a positional hash, but 2.7 changed the behavior. In addition to being backward incompatible, honestly... this doesn't seem to have much benefit? The issue has been reported in various (rejected) bug reports, and the pain point is described better than I ever could here:
https://discuss.rubyonrails.org/t/new-2-7-3-0-keyword-argument-pain-point/74980/13

#2 - 05/15/2020 03:45 AM - sawa (Tsuyoshi Sawada)

Dan0042 (Daniel DeLorme) wrote in #note-1:

> ## Non-Symbol Keyword Argument
>
> Previously one could trust that foo(1=>2) would be treated as a positional hash, but 2.7 changed the behavior. In addition to being backward incompatible, honestly... this doesn't seem to have much benefit? The issue has been reported in various (rejected) bug reports, and the pain point is described better than I ever could here:
> https://discuss.rubyonrails.org/t/new-2-7-3-0-keyword-argument-pain-point/74980/13

The core of that problem actually has nothing to do with allowing non-symbol keyword argument. That developer ad hocly used string vs. symbol distinction in hope that hash keys within a positional hash argument would be distinguished from keyword arguments, but that is specific to the DSL used there. If symbol were used as hash keys in positional hash argument, then the same problem would be met.

The real problem that lies there is the **argument precedence**. When there is a method call like:

```
def foo(bar = {}, **baz)
  #...
end

foo(a: 1)
```

in previous behavior, the ** applied before the optionality in = {}, and gobbled the argument {a: 1}, making bar to be always {}. But now, that problem has been solved by distinguishing {a: 1} and a: 1.

#3 - 05/15/2020 04:03 AM - marcandre (Marc-Andre Lafortune)

Dan0042 (Daniel DeLorme) wrote in #note-1:

> Also I believe there's a fifth aspect that deserves consideration for reverting:
>
> ## Non-Symbol Keyword Argument
>
> Previously one could trust that foo(1=>2) would be treated as a positional hash, but 2.7 changed the behavior. In addition to being backward incompatible, honestly... this doesn't seem to have much benefit? The issue has been reported in various (rejected) bug reports, and the pain point is described better than I ever could here:
> https://discuss.rubyonrails.org/t/new-2-7-3-0-keyword-argument-pain-point/74980/13

Totally agree. I probably missed that conversation, but I am very curious as to what the use-cases were supposed to be. I see only downsides to that particular decision and would love to see that reversed.

As for the rest, I still believe that easy generic call delegation is key; I'd like to have that without ....

#4 - 05/15/2020 11:20 AM - mame (Yusuke Endoh)

First of all, I'm really sorry for bothering many people about this change. And I'd like to really thank you jeremyevans0 (Jeremy Evans) for the great work to this issue. Honestly, I'm sorry to see this change reverted because I still believe this change is good. Anyway, I declare I will respect matz's final decision.

I tried Jeremy's patch, and in my first impression, it works great. Thank you. But I think that it is too early to decide how we should do. I'd like to wait for more "pain point" reports.

What we should not miss is that quite a few people already worked for Ruby 2.7 warnings, and many programs (including Rails itself) are running on 2.7 without warnings. I'm afraid if this revert breaks them.

This is just my current thinking.

Though I'm waiting for "pain point" reports, I presume that the hardest part is delegation. Passing keywords to delegation method (foo(k: 1) -> def foo(*args)) silently converts keywords to a hash, and then, passing the args to a keyword-aware method (bar(*args) -> def bar(k:)) emit a warning.

```
def bar(k:)
  p k
end

def foo(*args) # we should fix this: ruby2_keywords def foo(*args)
```

```
  ...
  bar(*args)    # emits a warning
  ...
end

foo(k: 1)
```

Worse, the argument array is sometimes kept to an instance variable.

```
def bar(k:)
  p k
end

def foo(*args) # we should fix this: ruby2_keywrods def foo(*args)
  @args = args
end

foo(k: 1)

...

bar(*@args) # emits a warning
```

We can simply fix this issue by adding ruby2_keywords to def foo(*args). This is the most painful to find where to fix.

IMO, the root cause is that passing keywords to a rest-argument method (foo(k: 1) -> def foo(*args)) silently converts keywords to a hash. This behavior will be kept in 3.0, so we cannot show a "deprecation" warning, but I wonder if it may be good to emit on the event in question, say, "keywords are passed to a rest-argument method; maybe ruby2_keywords should be added?" only on debug mode.

However, I know it would be never acceptable to add more opt-out warnings. Now I agree that all keyword-related warnings should be opt-in, anyway.

#5 - 05/16/2020 06:21 PM - jeremyevans0 (Jeremy Evans)

Dan0042 (Daniel DeLorme) wrote in #note-1:

> IMHO there's one consideration that's more important than all others. The following code must work in ruby 2.8/3.0
>
> ```
> def foo(h={}, **kw)
>   [h, kw]
> end
> foo({x:1}) #=> [{x:1}, {}]
> foo(x:1)   #=> [{}, {x:1}]
> ```
>
> This is the #1 reason why keyword separation is needed. The fix for the numerous bug reports like #14130 etc. Without this we might as well go back to 2.6 behavior. And from what I can understand it seems the "Proposed behavior for Ruby 3.0" doesn't support this.

You are correct that behavior I proposed doesn't support this. However, this is the most common issue with keyword argument separation. Either we keep the 2.7 behavior for compatibility or we move to the 3.0 behavior for consistency. There is no way to keep compatibility with 2.7 while keeping the benefits of separation.

In regards to "we might as well go back to 2.6 behavior", I think that notion is shortsighted at best. As I mentioned, there are 4 separate issues in regards to separation. You should have justifications for keeping all 4 aspects separately if you are seriously making a recommendation to revert all keyword argument separation changes.

> Also I believe there's a fifth aspect that deserves consideration for reverting:
>
> ## Non-Symbol Keyword Argument
>
> Previously one could trust that foo(1=>2) would be treated as a positional hash, but 2.7 changed the behavior. In addition to being backward incompatible, honestly... this doesn't seem to have much benefit? The issue has been reported in various (rejected) bug reports, and the pain point is described better than I ever could here:
> https://discuss.rubyonrails.org/t/new-2-7-3-0-keyword-argument-pain-point/74980/13

This aspect is not directly related to keyword argument separation, though it was introduced at the same time. This change is also different in nature from what Matz has proposed. Matz has discussed postponing the full separation. The full separation will still happen later.

Matz is considering keeping some Ruby 3.0 behavior the same as 2.7. As far as I know, he is not considering reverting 2.7 behavior to match 2.6, with the exception of removing warnings or lowering the verbosity of them. Removing non-symbol keyword arguments would make Ruby 3.0 behavior different from Ruby 2.7 behavior (though similar with Ruby 2.6 behavior). You could argue that we could remove non-symbol keyword arguments in 2.7.2, but I think that change is way too large to consider in a patch version.

#6 - 05/17/2020 03:44 AM - marcandre (Marc-Andre Lafortune)

> You could argue that we could remove non-symbol keyword arguments in 2.7.2, but I think that change is way too large to consider in a patch version.

I'd say revert now, but other possibility is warn in 2.7.2 and revert in 3.0

#7 - 05/17/2020 04:04 PM - zw963 (Wei Zheng)

IMHO, Update from ruby 2 to ruby 3, it should be a BIG major version, it was expected to breaking some things, right?

I don't know why we discuss about this, but, if the main issue come from ruby on rails, rails not stand for ruby, right?

I have to say, Keyword arguments is totally not so useful in ruby 2, in fact, anyone use it? even since we introduce this feature 7 years later? i guess the main reason cause this, i think is, it so confusing about HASH arguments with keyword arguments, In fact, because we always ambiguous on this, which lead to keyword argument a **Chicken Ribs **, no one use it, even it have several advantage, e.g. it can told you which name argument is missing.

So, i really propose, follow the thought at first, keep the evolution of ruby a clean way, we really need only one solution for this chaotic thing on ruby 3, if we don't want change this in ruby 3, maybe we never have chance to change even on ruby 4, maybe never introduce keyword arguments is a better solution then current, so, why not remove "keyword arguments" completely in ruby 3 ?

Just a personal opinion, thank you.

#8 - 05/18/2020 07:10 AM - sam.saffron (Sam Saffron)

To me the big question matz (Yukihiro Matsumoto) needs to answer is "do we want gems to work on both 3.X and 2.X?"

Personally I 100% support breaking kwargs and args over time, but I think there are massive advantages in making the break happen when 2.6 goes EOL as long as 2.7 has a clean replacement.

```
def bar(baz:)
end

def foo(*a)
  bar(*a)
end
```

To me ruby2_keywords is not a long term solution.

A long term solution addresses https://bugs.ruby-lang.org/issues/16897 cleanly.

When Ruby 2.6 goes EOL we can simply ask gem authors to convert code that looks like:

```
def foo(*args)
  do_stuff_with_args(args)
  bar(*args)
end
```

to

```
def foo(...)
  args = ...
  do_stuff_args(args)
  bar(...)
end
```

The change is mechanical and trivial. A new Arguments object can be introduced which is what ... is, it can have a to_a and a bunch of helpers to access the Arguments + an efficient C based equality method.

I support the revert, but I think it is far more important to have 2.7 ready to handle the final separation here and breaking change. At least then there is an end in sight to the various semantic issues we had over the years.

#9 - 05/18/2020 02:45 PM - jeremyevans0 (Jeremy Evans)

sam.saffron (Sam Saffron) wrote in #note-8:

> To me the big question matz (Yukihiro Matsumoto) needs to answer is "do we want gems to work on both 3.X and 2.X?"

You are implying that there needs to be changes for that to happen, which is not the case. Many popular gems already run without warnings on 2.7 and will run fine in 3.0. If not already, by the release of 3.0, the majority of popular gems will run correctly on it without restoring positional hash to

keyword conversion.

> Personally I 100% support breaking kwargs and args over time, but I think there are massive advantages in making the break happen when 2.6 goes EOL as long as 2.7 has a clean replacement.
>
> ```
> def bar(baz:)
> end
>
> def foo(*a)
>   bar(*a)
> end
> ```
>
> To me ruby2_keywords is not a long term solution.
>
> A long term solution addresses https://bugs.ruby-lang.org/issues/16897 cleanly.

I think the plan was to keep ruby2_keywords at least until 2.6 is EOL, after which it could be removed and everyone could switch to explicit keyword delegation. With this approach, there is always an approach that will work with all currently supported Ruby versions.

Personally, I don't think we should remove ruby2_keywords until there is an alternative approach that is equally efficient.

> When Ruby 2.6 goes EOL we can simply ask gem authors to convert code that looks like:
>
> ```
> def foo(*args)
>   do_stuff_with_args(args)
>   bar(*args)
> end
> ```
>
> to
>
> ```
> def foo(...)
>   args = ...
>   do_stuff_args(args)
>   bar(...)
> end
> ```
>
> The change is mechanical and trivial. A new Arguments object can be introduced which is what ... is, it can have a to_a and a bunch of helpers to access the Arguments + an efficient C based equality method.

It sounds like such an approach will be slower than the ruby2_keywords approach due to the additional object allocation.

#10 - 05/19/2020 12:21 AM - sam.saffron (Sam Saffron)

I agree many gems already work, but we have built ourselves a time bomb here.

ruby2_keywords is certainly going to go away one day, it is too ugly to keep permanently. What happens when it goes away? We need to then swap our implementation to a backwards compatible way that 2.7 supports. But the only alternative it supports is often a slower alternative.

> Personally, I don't think we should remove ruby2_keywords until there is an alternative approach that is equally efficient.
>
> It sounds like such an approach will be slower than the ruby2_keywords approach due to the additional object allocation.

Let me expand with a slightly alternative syntax for clarity:

```
def foo_old(*args)
  @track = args
  delegate(*args)
end

def foo_new(...args)
  @track = args
  delegate(...)
end

def foo_now(*args, **kwargs)
  @track = [args, kwargs]
  delegate(*args, **kwargs)
end
```

Now lets consider allocations:

foo

- old .. 1 Array allocation
- new .. 1 Arguments object allocation
- now .. 1 Array allocation + 1 Hash allocation + 1 Array allocation

foo(1,2)

- old .. 1 Array allocation
- new .. 1 Arguments object allocation
- now .. 1 Array allocation + 1 Hash allocation + 1 Array allocation

foo(1, a: 1)

- old .. 1 Array allocation
- new .. 1 Arguments object allocation
- now .. 1 Array allocation + 1 Hash allocation + 1 Array allocation

Old syntax and new syntax both allocate exactly the same amount ... 1 wrapping RVALUE.

New syntax sparks a lot more joy for quite a few reasons:

```
def foo(...a)
  puts a[:bar]
  puts a[0]
  puts a.length
  puts a.to_a
end

foo(1, 2, a: 1, bar: 2)
a[:bar] == 2
a[0] == 1
a.length == 4 # delightful cause *args would be length 3 which is confusing since there really are 4 args.
a.to_a == [1, 2, {a: 1, bar: 2}] # arguably we could make it `[1, 2, {a: 1} , {bar: 2}]
```

Arguments would be implemented in C which provides some big advantages like deferred materialization of the list of args. It would also implement its own version of an rb_ary_equal like function (rb_arguments_equal) which can be tuned to be very fast.

End result is that ...a would not only be as fast as *a is today but a bit faster in certain use cases.

#11 - 05/19/2020 01:07 AM - jeremyevans0 (Jeremy Evans)

sam.saffron (Sam Saffron) wrote in #note-10:

> I agree many gems already work, but we have built ourselves a time bomb here.
>
> ruby2_keywords is certainly going to go away one day, it is too ugly to keep permanently. What happens when it goes away? We need to then swap our implementation to a backwards compatible way that 2.7 supports. But the only alternative it supports is often a slower alternative.

Which I why I think it should not be removed until an equally efficient replacement exists. Assuming we did not remove ruby2_keywords until a equally efficient replacement exists (which I'm assuming will be after 2.6 is EOL), what problems do you see? Switching to a newer syntax that could possibly be introduced in 3.0 would not allow the same code to work with Ruby 2.0-2.6, which is what ruby2_keywords allows.

> Personally, I don't think we should remove ruby2_keywords until there is an alternative approach that is equally efficient.
>
> It sounds like such an approach will be slower than the ruby2_keywords approach due to the additional object allocation.

Let me expand with a slightly alternative syntax for clarity:

```
def foo_old(*args)
  @track = args
  delegate(*args)
end

def foo_new(...args)
  @track = args
  delegate(...)
end

def foo_now(*args, **kwargs)
  @track = [args, kwargs]
  delegate(*args, **kwargs)
end
```

Now lets consider allocations:

foo

- old .. 1 Array allocation
- new .. 1 Arguments object allocation
- now .. 1 Array allocation + 1 Hash allocation + 1 Array allocation

foo(1,2)

- old .. 1 Array allocation
- new .. 1 Arguments object allocation
- now .. 1 Array allocation + 1 Hash allocation + 1 Array allocation

foo(1, a: 1)

- old .. 1 Array allocation
- new .. 1 Arguments object allocation
- now .. 1 Array allocation + 1 Hash allocation + 1 Array allocation

Old syntax and new syntax both allocate exactly the same amount ... 1 wrapping RVALUE.

New syntax sparks a lot more joy for quite a few reasons:

```
def foo(...a)
  puts a[:bar]
  puts a[0]
  puts a.length
  puts a.to_a
end

foo(1, 2, a: 1, bar: 2)
a[:bar] == 2
a[0] == 1
a.length == 4
# delightful cause *args would be length 3 which is confusing since there really are 4 args.
a.to_a == [1, 2, {a: 1, bar: 2}] # arguably we could make it `[1, 2, {a: 1} , {bar: 2}]
```

Arguments would be implemented in C which provides some big advantages like deferred materialization of the list of args. It would also implement its own version of rb_ary_equal which can be tuned to be very fast.

End result is that ...a would not only be as fast as *a is today but a bit faster in certain use cases.

I'm not sure the deferred materialization you describe is feasible in terms of limiting allocations in CRuby. It sounds like you would want to keep the VALUE* argv passed instead if materializing (and similar for keyword arguments). However, such argv are often allocated on the stack, and you cannot rely on the contents of the memory after the call returns. You could copy the memory, but that will cause some allocation even if it isn't object allocation. Consider:

```
def foo(...a)
  proc{a}
end
foo(1).call[0]
```

In regards to your proposed Arguments#[], a[0] is ambiguous in the case of foo(1, 0=>2).

#12 - 05/19/2020 01:43 AM - Dan0042 (Daniel DeLorme)

An idea: it may be worth sub-dividing "Positional Hash Argument to Keyword Argument Conversion" into

a) Overflow Positional Hash Argument to Keyword Argument Conversion

When number of positional arguments is greater than maximum arity, keeping as positional would result in ArgumentError. Converting it to Keyword Argument is relatively safe.

```
def foo(key: 42); end
foo({key: 42})
```

b) Optional Positional Hash Argument to Keyword Argument Conversion

When number of positional arguments is greater than minimum arity but less or equal to maximum arity, it introduces the conditions for issues #14130 etc. Keeping those positional hash arguments as positional would prevent those issues. *rest delegation would convert the keywords to positional (unless you use #16463 or #16511) but then at the end of the delegation chain I think in many/most cases it would be re-converted to keywords because of a), as in this example:

```
def bar(*a)
  foo(*a)
```

```
end
bar({key: 42})
```

This is hard to quantify, but my intuition is that reverting a) while keeping b) would handle a large part of that "95% of keyword argument separation issues/warnings".
Just my intuition though.

#13 - 05/19/2020 03:19 AM - jeremyevans0 (Jeremy Evans)

Dan0042 (Daniel DeLorme) wrote in #note-12:

> An idea: it may be worth sub-dividing "Positional Hash Argument to Keyword Argument Conversion" into
>
> ### a) Overflow Positional Hash Argument to Keyword Argument Conversion
>
> When number of positional arguments is greater than maximum arity, keeping as positional would result in ArgumentError. Converting it to Keyword Argument is relatively safe.
>
> ```
> def foo(key: 42); end
> foo({key: 42})
> ```
>
> ### b) Optional Positional Hash Argument to Keyword Argument Conversion
>
> When number of positional arguments is greater than minimum arity but less or equal to maximum arity, it introduces the conditions for issues #14130 etc. Keeping those positional hash arguments as positional would prevent those issues. *rest delegation would convert the keywords to positional (unless you use #16463 or #16511) but then at the end of the delegation chain I think in many/most cases it would be re-converted to keywords because of a), as in this example:
>
> ```
> def bar(*a)
>   foo(*a)
> end
> bar({key: 42})
> ```
>
> This is hard to quantify, but my intuition is that reverting a) while keeping b) would handle a large part of that "95% of keyword argument separation issues/warnings".
> Just my intuition though.

I could be wrong, but I get the feeling the distinction you are trying to make is roughly the same as sawa (Tsuyoshi Sawada)'s argument precedence argument:

sawa (Tsuyoshi Sawada) wrote in #note-2:

> The real problem that lies there is the **argument precedence**. When there is a method call like:
>
> ```
> def foo(bar = {}, **baz)
>   #...
> end
>
> foo(a: 1)
> ```
>
> in previous behavior, the ** applied before the optionality in = {}, and gobbled the argument {a: 1}, making bar to be always {}. But now, that problem has been solved by distinguishing {a: 1} and a: 1.

This argument wouldn't apply to def bar(*a) as that doesn't accept keywords, but would apply to def bar(*a, **kw) or def bar(a={}, **kw).

Implementing this behavior is a simple modification to my patch, though it would make the 2.7 behavior more challenging since you would have to verbose warn in some cases and non-verbose warn in others.

This behavior can make more sense in some cases, but makes less sense in others. Consider def foo(a = true, **kw). foo({bar: 1}) in earlier Ruby versions would treat bar: 1 as keywords, and arguably that makes more sense for this method definition if you are doing positional hash to keyword conversion.

Changing the argument precedence when restoring behavior makes the behavior different from 2.7. I think if we are going to restore this behavior to keep backwards compatibility, we should probably do so in a manner compatible with 2.7. However, I don't have strong feelings about it.

#14 - 05/19/2020 03:56 AM - sam.saffron (Sam Saffron)

> Assuming we did not remove ruby2_keywords until a equally efficient replacement exists (which I'm assuming will be after 2.6 is EOL), what problems do you see?

Honestly ... no problem at all, the only issue I see is that we are in a rush to come up with an efficient replacement.

If we come up with an efficient replacement for 3.0 then when finally we are ready to EOL 2.7 we can remove ruby2_keywords. So for example in Ruby 3.4 we could raise a loud deprecation on ruby2_keywords. If this waits till Ruby 3.1 ... then we are stuck carrying ruby2_keywords for another release.

To me the big thing we missed here is that certain apps only target latest Ruby so the preference there for application authors is to pick the efficient replacement for *args, over ruby2_keywords. They can move earlier than gem authors.

As to the OP here.

My vote is

1. Restore positional Hash Argument to Keyword Argument Conversion
2. Warning in verbose mode only for Ruby 3
3. 1 release prior to removal of ruby2_keywords warn more loudly

Don't touch any of the other edge cases for now.

We are stuck with ruby2_keywords for a while sadly, so I don't see any reason to rush people to change code to use it.

I also wonder is ruby2_keywords even necessary if we are still keeping restoration of positional hash around for a few more releases?

Another bigger question is ... what if for: def(*x); p x.class; end # x class is Arguments

Then lots of code needs no change. Especially if Arguments behaves mostly like the Array did in the past, we don't even need new syntax. Yes it is a big change ... but it keeps syntax of ruby cleaner.

#15 - 05/19/2020 09:45 AM - kamipo (Ryuta Kamizono)

Instead of restoring all positional hash argument to keyword argument conversion, how about adding new implicit maybe kwargs flag for opt-in warning (i.e. implicit maybe kwargs flag by default)?

```
def foo(key: 42); end
foo({key: 42})

def bar(*a)
  foo(*a)
end
bar({key: 42})
```

I agree that the warning for the case bar({key: 42}) is'nt quite helpful for multiple nested delegation where should be fixed, so there is room for improving the warning message.

On the other hand, the warning for the case foo({key: 42}) is clear enough what we should do, so is it worth postponing this case even after a lot of code has already been addressed?

#16 - 05/20/2020 08:29 PM - ziggythehamster (Keith Gable)

A use case that seems to have been forgotten that I want to explicitly point out (and I'm not sure if this is the ticket for it, given the discussion here vs. the linked thread, so I'm happy to open a new issue if needed) is that it seems to be currently impossible to use keyword arguments in 2.7 with "operator" syntax without triggering a deprecation warning.

An example:

```
class SomeProcessor
  def <<(foo:, bar:, baz:)
    # do something with an explicit keyword argument list
  end
end


i = SomeProcessor.new
i << { foo: 1, bar: 2, baz: 3 }
# deprecation warning, but reasonable syntax, which explodes as expected when the hash is not conforming to th
e expected keys
i << ( foo: 1, bar: 2, baz: 3 )
# illegal syntax but unambiguous and reasonable, if you wanted to introduce something like an "argument list"
type because you want implicit conversion to go away
i << foo: 1, bar: 2, baz: 3        # illegal syntax and ambiguous
i << **{ foo: 1, bar: 2, baz: 3 }  # illegal syntax and not very good looking even if it did work
i.<<(**{ foo: 1, bar: 2, baz: 3 })
# works without deprecation warnings but not good looking, which defeats the vision of kwargs
i.<<(foo: 1, bar: 2, baz: 3)
# works without deprecation warnings and the least bad looking of the "not beautiful" ones ... we could call t
```

```
his "add" instead of "<<" but that feels un-Ruby-like
```

Given the discussion here, I think that most would agree that the i << {} syntax should work fine and be preferred in Ruby 3.x (and thus 2.7), and it should be able to be unambiguously supported, but I don't immediately understand what I'm doing "wrong" that makes Ruby unhappy about this given the error message, since it feels idiomatic and "Ruby" to me (as someone who's been programming in Ruby since 1.8.2).

In general, it feels strange to have to add ** when calling if I want my method signature to use keyword arguments, but I don't have a better suggestion that is more intuitive, and I understand the desire to make them a separate "thing". Keyword arguments (which aren't just wrapping a hash) would make static analysis and optimization much better, and implicit hash conversion brings along baggage, some of which is historical.

Maybe the better approach is to have a pragma that people can add that officially declares that all argument lists in the file aren't relying on legacy behavior and then tune the deprecation schedule accordingly?

#17 - 05/21/2020 12:58 AM - mame (Yusuke Endoh)

Hi, ziggythehamster (Keith Gable), thank you for your feedback, and I'm sorry for troubling you.

In this case, I think you can accept the Hash argument as a positional one. How about this?

```
class SomeProcessor
  def <<(hash)
    foo = hash[:foo]
    bar = hash[:bar]
    baz = hash[:baz]

    # Alternatively, you can use pattern matching since 2.7:
    hash in { foo:, bar:, baz:, **nil }

    # do something
  end
end
```

An explicit check may be needed if you want to raise an ArgumentError when an extra (unknown) argument is passed.

We have receive some related reports:

- https://bugs.ruby-lang.org/issues/14183#change-83014
- https://bugs.ruby-lang.org/issues/16494

This is just an idea, but we may allow pattern matching in an argument to accept a positional argument and decomposes it based on the pattern.

```
class SomeProcessor
  def <<(_hash in { foo:, bar:, baz: })
    # do something
  end
end

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

This is a bit tedious than the old style, but in my opinion it is reasonably clean and explicit. I'll try to implement and then create another ticket.

#18 - 05/21/2020 06:26 AM - ziggythehamster (Keith Gable)

A sole positional argument does in fact work, and I am a big fan of the pattern matching feature (enough that once we're on 2.7, I'm considering -W:no-experimental), but I hadn't thought about the combination of the two, and I certainly hadn't thought about it in the form of a match in the argument list... and I really like that. :) My original thinking was that in this case we know we can avoid creating a Hash in the first place, but end up having to in order to make the callers have a nice syntax to use. With a non-operator method, we can just use kwargs and be done, but that's not possible with operator methods. Perhaps there's some optimization where e.g., hashes never reused but used as keyword arguments are treated specially as a language construct, but I don't know if the compiler has any way of knowing that the hash is being passed to a kwargs-taking method.

Reading up on this issue a bit more since this morning, I think (as a user), the best course of action is to make the previous syntax work fine without deprecations except when $VERBOSE or whatever. I also think more cases where arguments are unambiguous should remain supported >=3 if you are comfortable assuming people were following the old rules and haven't begun using the new ones ("x" => "y", foo: "bar" is two arguments since kwargs didn't formerly take stuff that wasn't symbols). Maybe a pragma can opt in to the live 2.7 behavior of converting non-string args?

#19 - 05/28/2020 11:06 AM - ioquatix (Samuel Williams)

I like the current plan for keyword argument separation. Promotion of hash to keywords is very confusing and cause many problems designing clean interfaces in Ruby. Kudos to jeremyevans0 (Jeremy Evans) and everyone else here for providing calm and rational discussion. Dan0042 (Daniel DeLorme) really hits the nail on the head with the first reply.

In 2.x the following behaviour is **very** confusing:

```
def with(*arguments, **options)
  p arguments
  p options
end

with({foo: 10}) # arguments = [], options = {foo: 10}
with({foo: 10}, bar: 20) # arguments = [{foo: 10}], options = {bar: 20}
```

This design is impossible to work around and absolutely must be fixed in Ruby 3. It cause me many frustrations and confuse users when hash arguments become options, e.g. https://github.com/socketry/async/issues/67 is just one of many issues I've had with this behaviour.

I personally don't care **how** we fix it. It can be a per-file pragma, we could introduce ruby3_keywords like ruby2_keywords. However, the current implementation is clean and I've upgraded many gems and had no issues. The warnings are clear, explain how to fix the problem and in several issues revealed bugs in my own code (incorrect conversion of options to required positional argument).

I have no problems with the current path, I'm the maintainer of lots of gems, I've successfully prepared them for Ruby 3, and I look forward to a future where we can take full advantage of keyword arguments.

With respect to users and developers, I believe that if they cannot handle upgrading their code, they should stay on the version of Ruby that works for them. It will be supported for many more years. There is nothing wrong with Ruby 2.6 for code bases that must continue to operate with old code. In some of my consulting work, some applications are still running on 1.8.7 (but very few, thankfully).

Looking forward, the delegation problem is better solved by an argument tail operator. I think ... and extended forms of it are a great way forward, e.g.

```
def delegate(context, ...)
  context.call(...)
end
```

This is far better as a pattern for delegation than any of the other options we've explored here.

#20 - 05/28/2020 11:41 AM - Eregon (Benoit Daloze)

- Related to Feature #16378: Support leading arguments together with ... added

#21 - 05/28/2020 11:47 AM - Eregon (Benoit Daloze)

Regarding delegation, I agree we need to support at least leading arguments together with ....
I filed #16378 6 months ago (before the 2.7 release), it was rejected and then later accepted.
I think we should backport that to 2.7 as it's the only clean way to do delegation in 2.7 (considering that ruby2_keywords feels like a hack for many, and * and *,** have subtle bugs in 2.7).

#22 - 06/16/2020 12:21 AM - sylvain.joyeux (Sylvain Joyeux)

sawa (Tsuyoshi Sawada) wrote in #note-2:

> The core of that problem actually has nothing to do with allowing non-symbol keyword argument. That developer ad hocly used string vs. symbol distinction in hope that hash keys within a positional hash argument would be distinguished from keyword arguments, but that is specific to the DSL used there. If symbol were used as hash keys in positional hash argument, then the same problem would be met.

I'm this user ...

Yes. But it was not. The strings are not keywords-like and would make no sense as symbols. Plus some calls use strings, some calls use objects that are part of the domain being modeled. Which really can't even be confused with keyword arguments.

> The real problem that lies there is the **argument precedence**. When there is a method call like:

Up until 2.6.x, **attempting** to pass non-keyword arguments to a keyword splat would be an error. So, no, it would not be a matter of precedence. I was assuming that at the very worst (for me), the way I was using Ruby splitting the two hashes would give me an error. Because without allowing arbitrary objects as keys in keyword arguments, there was no other way.

Basically, I was assuming that once the real keyword arguments come, the *syntax* k: 10, v: 20 at the end of a method would be interpreted as keyword arguments, leaving the rest as not-keyword-arguments. This was my assumption and of course the core developers can do as they please. But I stand by my analysis that, without this allowing of arbitrary arguments into keywords, I wouldn't have to deal with so many landmines.

In addition, the fact that the two signatures

```
def m(positional, k: 10); end
def m(positional, k: 10, **kw); end
```

behave differently makes the whole thing horribly fragile.

So, yes, it might be a corner-case. But if I came up with ways to leverage this old behavior, you can be sure that I'm certainly not the only one. Ruby still has enough users to virtually guarantee that.

#23 - 06/23/2020 09:49 AM - Eregon (Benoit Daloze)

I think even just doing the first one, "Positional Hash Argument to Keyword Argument Conversion", is only going to make everything more confusing and delay migration.
In other words, I think there will be far more pain points that way than currently.

If delegation is a frequent problem, I think we should do #16463 which specifically targets delegation and doesn't really compromise the semantic model.
That also gives us an easy way to warn later on for delegation and do that migration at the time where it's safe to use *args, **kwargs, ..., or something new without compromising backward compatibility.
If delegation is not a frequent problem, then I think the original plan for 2.7/3.0 is fine.

Going back to converting a positional hash to kwargs is IMHO going so much backward we might as well give up on separation entirely.

#24 - 06/30/2020 02:58 PM - jeremyevans0 (Jeremy Evans)

- Status changed from Open to Rejected

Files

keyword-hash-integration.diff	5.2 KB	05/14/2020	jeremyevans0 (Jeremy Evans)