Ruby master - Misc #16188

What are the performance implications of the new keyword arguments in 2.7 and 3.0?

09/29/2019 06:27 PM - Eregon (Benoit Daloze)

Status:	Open
Priority:	Normal
Assignee:	jeremyevans0 (Jeremy Evans)

Description

In #14183, keyword arguments became further separated from positional arguments.

Contrary to the original design though, keyword and positional arguments are not fully separated for methods not accepting keyword arguments.
Example: foo(key: :value) will def foo(hash) will pass a positional argument.
This is of course better for compatibility, but I wonder what are the performance implications.

The block argument is completely separate in all versions, so no need to concern ourselves about that.

In Ruby <= 2.6:

- The caller never needs to know about the callee's arguments, it can just take all arguments and pass them as an array. The last argument might be used to extract keyword, but this is all done at the callee side.
- Splitting kwargs composed of Symbol and non-Symbol keys can be fairly expensive, but it is a rare occurrence. If inlining the callee and kwargs are all passed as a literal Hash at the call site, there shouldn't be any overhead compared to positional arguments once JIT'ed.

In Ruby 2.7:

- The caller needs to pass positional and keyword arguments separately, at least when calling a method accepting kwargs. But, if it calls a methods not accepting kwargs, then the "kwargs" (e.g. foo(key: :value)) should be treated just like a final Hash positional argument.
- (If we had complete separation, then we could always pass positional and keyword arguments separately, so the caller could once again ignore the callee)

How is the logic implemented in MRI for 2.7?

Specializing the caller for a given callee is a well-known technique.
However, it becomes more difficult if different methods are called from the same callsite (polymorphic call), especially if one accepts kwargs and another does not.
In that case, I think we will see a performance cost to this approach, by having to pass arguments differently based on the method to be called.

What about delegation using ruby2_keywords?
Which checks does that add (compared to 2.6) in the merged approach with the Hash flag?

Related issues:

Related to Ruby master - Feature #14183: "Real" keyword argument	**Closed**
Related to Ruby master - Misc #16157: What is the correct and *portable* way ...	**Open**
Related to Ruby master - Feature #16897: General purpose memoizer in Ruby 3 w...	**Open**
Related to Ruby master - Feature #16463: Fixing *args-delegation in Ruby 2.7:...	**Closed**

History

#1 - 09/29/2019 06:27 PM - Eregon (Benoit Daloze)

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

#2 - 09/29/2019 06:29 PM - Eregon (Benoit Daloze)

- Description updated

#3 - 09/29/2019 09:01 PM - jeremyevans0 (Jeremy Evans)

Eregon (Benoit Daloze) wrote:

> In #14183, keyword arguments became further separated from positional arguments.

> Contrary to the original design though, keyword and positional arguments are not fully separated for methods not accepting keyword arguments.
> Example: foo(key: :value) will def foo(hash) will pass a positional argument.
> This is of course better for compatibility, but I wonder what are the performance implications.

Internally, there are not really performance implications for the choice to treat keyword arguments as a last positional hash. If we not do so, the foo(key: value) call would be an ArgumentError. I don't believe we had to give up any optimizations in CRuby when I took mame's original branch and added backwards compatibility to treat keyword arguments as a last positional hash.

> In Ruby <= 2.6:
>
> - The caller never needs to know about the callee's arguments, it can just take all arguments and pass them as an array. The last argument might be used to extract keyword, but this is all done at the callee side.
> - Splitting kwargs composed of Symbol and non-Symbol keys can be fairly expensive, but it is a rare occurrence. If inlining the callee and kwargs are all passed as a literal Hash at the call site, there shouldn't be any overhead compared to positional arguments once JIT'ed.

If you pass a literal hash to a method that accepts a keyword splat (where the literal hash is treated as keywords), I assume there still must be overhead compared to passing a literal hash to a method that it does not accept keyword arguments, as it must allocate a new hash. I think passing a literal hash to a method that accepts explicit keywords but no keyword hash can be allocation-less. I'm not sure if the JIT engine change things in this area, maybe k0kobun knows.

> In Ruby 2.7:
>
> - The caller needs to pass positional and keyword arguments separately, at least when calling a method accepting kwargs. But, if it calls a methods not accepting kwargs, then the "kwargs" (e.g. foo(key: :value)) should be treated just like a final Hash positional argument.
> - (If we had complete separation, then we could always pass positional and keyword arguments separately, so the caller could once again ignore the callee)
>
> How is the logic implemented in MRI for 2.7?

The general structure remains the same. There are warnings added for methods that accept keyword arguments that will break in Ruby 3:

- Treating a last hash as keywords.
- Treat keywords as a final mandatory positional argument.
- Splitting last hash or keywords when explicit keywords given and keyword splat not accepted.

There are some behavior changes compared to 2.7:

- No need to split keyword hash (or last hash treated as keywords) if a keyword splast is accepted, as keyword splats accept non-Symbol keys in 2.7.
- **nil is now supported in method definitions for treating the method like it accepts keywords, but not have any keywords accepted.
- Passing an empty keyword splat to a method no longer passes an argument, unless this argument is a required positional argument, in which case it is warned.

This last point is the one behavior in CRuby 2.7 that has significant performance implications. What happens is an empty hash argument is not passed, but the call is flagged that empty keyword hash was passed. If the empty keyword hash turns out to be required to fulfill a positional argument, an empty hash is added back. This requires allocating a temporary buffer to store the new argument array.

> Specializing the caller for a given callee is a well-known technique.
> However, it becomes more difficult if different methods are called from the same callsite (polymorphic call), especially if one accepts kwargs and another does not.
> In that case, I think we will see a performance cost to this approach, by having to pass arguments differently based on the method to be called.

I don't think this is true in CRuby. It certainly could be true in other Ruby implementations. However, I would think you could always treat things as passing keywords in the caller code, and just have the callee convert keywords to a positional hash if the method does not accept keywords.

> What about delegation using ruby2_keywords?
> Which checks does that add (compared to 2.6) in the merged approach with the Hash flag?

For methods that are flagged with ruby2_keywords (which can only happen if the method accepts a regular splat but no keywords or keyword splat), if the method is called with keywords, we set a flag so that the keyword hash is treated as a last positional hash with the keyword flag set. Additionally, in this case the empty keyword hash is not removed, it is treated the same as a non-empty keyword hash.

For all method calls that use an argument splat and pass no keywords or keyword splat, if the last element of the argument splat array is a hash with the keyword flag set, that argument is treated as keywords instead of a positional hash.

#4 - 10/14/2019 05:56 PM - Eregon (Benoit Daloze)

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

#5 - 10/15/2019 07:13 AM - Eregon (Benoit Daloze)

jeremyevans0 (Jeremy Evans) wrote:

Thank you for your detailed reply.

> Internally, there are not really performance implications for the choice to treat keyword arguments as a last positional hash. If we not do so, the foo(key: value) call would be an ArgumentError.

I think there are. If args and kwargs are completely separated, we can choose to represent kwargs as we wish, and only need to build a Hash if the method takes a **kwrest.
But in the current approach, for a call site calling both a method taking kwargs and another not taking kwargs, we'll need logic to handle both the kwargs case and the conversion to a last positional Hash. If they were completely separated, the positional Hash would just be an ArgumentError and we wouldn't need that logic at all.
I agree that would be too incompatible though, so let's focus on optimizing the semantics in Ruby 2.7 and for Ruby 3.

> I don't believe we had to give up any optimizations in CRuby when I took mame's original branch and added backwards compatibility to treat keyword arguments as a last positional hash.

I think CRuby does not do many optimizations for keyword arguments except for the simplest case, I'd like to consider optimizations and JITs in general.
A few extra checks in MRI might not be noticeable if some form of kwargs (e.g., **kwargs) is generally slow, but could very well be significant on a more optimizing Ruby implementation.

> If you pass a literal hash to a method that accepts a keyword splat (where the literal hash is treated as keywords), I assume there still must be overhead compared to passing a literal hash to a method that it does not accept keyword arguments, as it must allocate a new hash.

**kwrest allocates conceptually, but the allocation does not need to happen, for instance if the Hash instance is not stored anywhere and escape analysis can figure it out.
This is how keyword arguments are currently optimized in TruffleRuby: we always create the Hash, but the escape analysis in the JIT figures it does not need to actually allocate it in many cases.

> I think passing a literal hash to a method that accepts explicit keywords but no keyword hash can be allocation-less. I'm not sure if the JIT engine change things in this area, maybe k0kobun knows.

When I mentioned JIT, I was thinking specifically to TruffleRuby, JRuby and highly-optimizing JITs.
I think MJIT doesn't optimize kwargs better than the interpreter currently.

> The general structure remains the same. There are warnings added for methods that accept keyword arguments that will break in Ruby 3:

Right, what about Ruby 3?
We should aim something that is both semantically simpler and easier to optimize then.

> There are some behavior changes compared to 2.7:
>
> - No need to split keyword hash (or last hash treated as keywords) if a keyword splast is accepted, as keyword splats accept non-Symbol keys in 2.7.

That should be good for performance, no more "crazy" splitting in argument handling and complicated checks for whether splitting should be tried :)

> - Passing an empty keyword splat to a method no longer passes an argument, unless this argument is a required positional argument, in which case it is warned.
>
> This last point is the one behavior in CRuby 2.7 that has significant performance implications. What happens is an empty hash argument is not passed, but the call is flagged that empty keyword hash was passed. If the empty keyword hash turns out to be required to fulfill a positional argument, an empty hash is added back. This requires allocating a temporary buffer to store the new argument array.

Right, so we need to pass an extra flag with the call for that case at least.

How are *args and **kwargs actually represented in MRI?
Is it like a flat array of "arguments", and the last one can be a keyword Hash, and there is some flag passed to the callee to differentiate a last positional Hash from keyword arguments?
Then methods taking kwargs need to check the flag to know how many positional args are passed, and whether kwargs are passed.
Methods not taking kwargs don't even need to check the flag, and can consider the entire flat array as positional arguments.

I guess cases like foo(a: 1, b: 2) are not represented with a Hash to avoid allocations in MRI, how does it look like then for such a case?
That makes the case for methods not taking kwargs quite more complicated, as they need to read the "passed optimized kwargs" flag and if set convert those optimized kwargs to a Hash, and add + 1 to the count of positional arguments.

> I don't think this is true in CRuby. It certainly could be true in other Ruby implementations. However, I would think you could always treat things as passing keywords in the caller code, and just have the callee convert keywords to a positional hash if the method does not accept keywords.

Right, that sounds like the easiest approach.

> For methods that are flagged with ruby2_keywords (which can only happen if the method accepts a regular splat but no keywords or keyword splat), if the method is called with keywords, we set a flag so that the keyword hash is treated as a last positional hash with the keyword flag set. Additionally, in this case the empty keyword hash is not removed, it is treated the same as a non-empty keyword hash.

That should be fine, because it means only ruby2_keywords flagged methods would have a little overhead for that extra logic (i.e., writing a flag or @ivar on the Hash).
Deciding to keep or remove the empty keyword hash case should itself be free once JITed, as it's a never-changing property of a method, if we make ruby2_keywords define a new method internally.

> For all method calls that use an argument splat and pass no keywords or keyword splat, if the last element of the argument splat array is a hash with the keyword flag set, that argument is treated as keywords instead of a positional hash.

That's the one I'm concerned about, it means every foo(some arguments) with def foo(*args); bar(*args); end will have to check if the last positional argument is a Hash with the flag set:
args.size >= 1 && Hash === args[-1] && flagged?(args[-1]). That's a quite a few branches and it's not just reading one bit from memory, it's at least 4 memory loads (size, [-1], .class, .flag).
I think this is another motivation for having ruby2_keywords only in Ruby 2.7, as it's clearly not free for performance.

We could skip that if we know currently no method uses ruby2_keywords, but as soon as it's used we have to perform the check for every method call with the method taking a rest argument (and no explicit kwargs).
I would think there are quite a few methods taking a rest argument and no explicit kwargs, i.e. def foo(*args), def foo(req, *args), def foo(opt=default, *args), etc all need the extra check
but def foo(*args, **kwargs)/def foo(*args, kwarg:)/def foo(*args, kwarg: default) do not need the check.

Maybe we should be more explicit about when to convert a flagged Hash to keyword arguments.
For instance we could have, using ActionDispatch::MiddlewareStack::Middleware's example:

```
class Middleware
  attr_reader :args, :block, :klass

  def initialize(klass, args, block)
    @klass = klass
    @args  = args
    @block = block
  end
  ruby2_keywords :initialize if respond_to?(:ruby2_keywords, true)

  # Current code
  def build(app)
    klass.new(app, *args, &block)
  end

  # idea to be explicit when converting to keyword args
  # and avoid extra checks in other call to methods with a *rest argument.
  def build(app)
    klass.send_keyword_hash(:new, app, *args, &block)
  end
end
```

That would limit that complicated check to only send_pass_kwargs calls, and not any other calls.
It's also more explicit and less magic, which might be easier to understand and explain.
I'm unsure about a good name, maybe send_pass_kwargs, send_pass_flagged_kwargs, send_pass_flagged_hash, send_flagged_hash_as_kwargs, send_kwhash, send_keyword_hash?
What do you think?

#6 - 10/15/2019 06:42 PM - jeremyevans0 (Jeremy Evans)

Eregon (Benoit Daloze) wrote:

> jeremyevans0 (Jeremy Evans) wrote:
>
> > Internally, there are not really performance implications for the choice to treat keyword arguments as a last positional hash. If we not do so, the foo(key: value) call would be an ArgumentError.
>
> I think there are. If args and kwargs are completely separated, we can choose to represent kwargs as we wish, and only need to build a Hash if the method takes a **kwrest.

> But in the current approach, for a call site calling both a method taking kwargs and another not taking kwargs, we'll need logic to handle both the kwargs case and the conversion to a last positional Hash. If they were completely separated, the positional Hash would just be an ArgumentError and we wouldn't need that logic at all.
> I agree that would be too incompatible though, so let's focus on optimizing the semantics in Ruby 2.7 and for Ruby 3.

I should preface this by saying all of my comments related to performance implications were and are implicitly refering to CRuby's implementation. I have no knowledge of the internals of other Ruby implementations and no idea what the performance implications of the differences are on other implementations.

Note that when discussing performance implications of keyword arguments, we should be clear what we are discussing. In Ruby 2.7 and 3.0, the caller still doesn't need to know about the callee's argument handling, it just passes all arguments through. It is up to the callee to raise an ArgumentError if the arguments are not correct. This is the same as older Ruby versions. Just as in older Ruby versions, the callee handles treating keywords as a final positional hash if the method does not accept keyword arguments. The only thing that is really changed is how the callee is handling the arguments, the caller code is generally unaffected.

> > The general structure remains the same. There are warnings added for methods that accept keyword arguments that will break in Ruby 3:
>
> Right, what about Ruby 3?
> We should aim something that is both semantically simpler and easier to optimize then.

If you want to see what Ruby 3 keyword argument handling will look like, I have a branch prepared for that already:
https://github.com/jeremyevans/ruby/tree/r3

I have been using this for testing my apps and libraries to be sure that any case that will break in Ruby 3 will be warned in Ruby 2.7. Most of the diff for the core changes is code removal, there are very few additions:
https://github.com/jeremyevans/ruby/commit/6aa7d021f694537ad15d3f72941e4816639ddfd5

The code does end up being semantically simpler, and it should be faster, though I don't expect the performance to be significantly better in real-world cases.

> How are *args and **kwargs actually represented in MRI?
> Is it like a flat array of "arguments", and the last one can be a keyword Hash, and there is some flag passed to the callee to differentiate a last positional Hash from keyword arguments?
> Then methods taking kwargs need to check the flag to know how many positional args are passed, and whether kwargs are passed.
> Methods not taking kwargs don't even need to check the flag, and can consider the entire flat array as positional arguments.

In most of the C-API, int argc, VALUE *argv, with the addition of int kw_splat in Ruby 2.7 for whether the final argument is a hash that should be treated as keywords.

> I guess cases like foo(a: 1, b: 2) are not represented with a Hash to avoid allocations in MRI, how does it look like then for such a case?

There is an optimization for literal keyword use in both the caller and callee (if both are written in Ruby) so that a hash is not allocated in that case if no keyword splats are present in the caller and callee.

> That makes the case for methods not taking kwargs quite more complicated, as they need to read the "passed optimized kwargs" flag and if set convert those optimized kwargs to a Hash, and add + 1 to the count of positional arguments.

In CRuby, it's not significantly more complicated. The callee may need to add the literal keywords to a hash anyway (if callee uses a keyword splat). Moving that hash to the last positional argument is very little work.

> > For methods that are flagged with ruby2_keywords (which can only happen if the method accepts a regular splat but no keywords or keyword splat), if the method is called with keywords, we set a flag so that the keyword hash is treated as a last positional hash with the keyword flag set. Additionally, in this case the empty keyword hash is not removed, it is treated the same as a non-empty keyword hash.
>
> That should be fine, because it means only ruby2_keywords flagged methods would have a little overhead for that extra logic (i.e., writing a flag or @ivar on the Hash).

Correct, the check to add the flag is only on ruby2_keywords flagged methods.

> > For all method calls that use an argument splat and pass no keywords or keyword splat, if the last element of the argument splat array is a hash with the keyword flag set, that argument is treated as keywords instead of a positional hash.
>
> That's the one I'm concerned about, it means every foo(some arguments) with def foo(*args); bar(*args); end will have to check if the last positional argument is a Hash with the flag set:
> args.size >= 1 && Hash === args[-1] && flagged?(args[-1]). That's a quite a few branches and it's not just reading one bit from memory, it's at least 4 memory loads (size, [-1], .class, .flag).

> I think this is another motivation for having ruby2_keywords only in Ruby 2.7, as it's clearly not free for performance.

It's not free in CRuby. However, if you look at all of the surrounding code, I don't think the performance difference in CRuby will be significant.

> I would think there are quite a few methods taking a rest argument and no explicit kwargs, i.e. def foo(*args), def foo(req, *args), def foo(opt=default, *args), etc all need the extra check
> We could skip that if we know currently no method uses ruby2_keywords, but as soon as it's used we have to perform the check for every method call with the method taking a rest argument (and no explicit kwargs).

A sufficiently advanced compiler in a more optimized Ruby implementation should be able to determine in most cases whether the method is ever called with such a hash, and optimize the check out, correct? :)

Consider:

```
ruby2_keywords def foo(*a) bar(*a) end
```

Other than the check at this specific bar call-site, you don't need to check other call sites. Because when bar is called here, the hash with the flag is duped and the flag removed (as if the hash was keyword splatted), so the flagged hash never escapes this method. This is not how the master branch works in CRuby, but I consider that a bug, and I'll be fixing it shortly.

Handling more involved cases where the arguments are stored and can escape may require checking all call sites with splats and no explicit keywords, though.

> but def foo(*args, **kwargs)/def foo(*args, kwarg:)/def foo(*args, kwarg: default) do not need the check.

correct.

> Maybe we should be more explicit about when to convert a flagged Hash to keyword arguments.
> For instance we could have, using ActionDispatch::MiddlewareStack::Middleware's example:
>
> ```
> class Middleware
>   attr_reader :args, :block, :klass
>
>   def initialize(klass, args, block)
>     @klass = klass
>     @args  = args
>     @block = block
>   end
>   ruby2_keywords :initialize if respond_to?(:ruby2_keywords, true)
>
>   # Current code
>   def build(app)
>     klass.new(app, *args, &block)
>   end
>
>   # idea to be explicit when converting to keyword args
>   # and avoid extra checks in other call to methods with a *rest argument.
>   def build(app)
>     klass.send_keyword_hash(:new, app, *args, &block)
>   end
> end
> ```
>
> That would limit that complicated check to only send_pass_kwargs calls, and not any other calls.
> It's also more explicit and less magic, which might be easier to understand and explain.
> I'm unsure about a good name, maybe send_pass_kwargs, send_pass_flagged_kwargs, send_pass_flagged_hash, send_flagged_hash_as_kwargs, send_kwhash, send_keyword_hash?
> What do you think?

send_keyword_hash is more explicit, but it requires significantly more code or a global monkey patch of Object#send_keyword_hash to be backwards compatible. The main goal for ruby2_keywords is to allow existing method definitions to work without changing them, just by flagging the method.

I assume in your example, initialize should accept *args and not args, as otherwise, you can't use ruby2_keywords.

It seems odd to recommend we add another feature (Kernel#send_keyword_hash) to compliment a feature (Module#ruby2_keywords) that you are strongly recommending we remove in Ruby 3.0.

#7 - 10/22/2019 09:42 PM - Eregon (Benoit Daloze)

jeremyevans0 (Jeremy Evans) wrote:

> In CRuby, it's not significantly more complicated. The callee may need to add the literal keywords to a hash anyway (if callee uses a keyword

> splat). Moving that hash to the last positional argument is very little work.

It's still a condition in the callee which needs to handle both kinds of callers: passing a Hash and passing literal keywords, with completely different ways to read keyword arguments from that.
That's likely rather inefficient if it happens in practice.

> > That's the one I'm concerned about, it means every foo(some arguments) with def foo(*args); bar(*args); end will have to check if the last positional argument is a Hash with the flag set:
> > args.size >= 1 && Hash === args[-1] && flagged?(args[-1]). That's a quite a few branches and it's not just reading one bit from memory, it's at least 4 memory loads (size, [-1], .class, .flag).
> > I think this is another motivation for having ruby2_keywords only in Ruby 2.7, as it's clearly not free for performance.
>
> It's not free in CRuby. However, if you look at all of the surrounding code, I don't think the performance difference in CRuby will be significant.

Maybe not in CRuby. But with a sufficiently optimized Ruby implementation, arguments handling should be free whenever passed from caller to inlined callee, and should be cheap when not inlined.

> A sufficiently advanced compiler in a more optimized Ruby implementation should be able to determine in most cases whether the method is ever called with such a hash, and optimize the check out, correct? :)

I don't think so. As soon as one case stores a flagged Hash we'd have to check *all* call sites using a *rest argument and no kwargs.
ActionDispatch already has such a case.

I think that check can only be optimized out if we create the Hash in the same compilation unit. If not, we'll have to check the flag, on every such call.
I think you need to believe my word here that this is not free and this is significant enough that we don't want to shoot ourselves for all future Ruby versions with that overhead.
For me, it ruins a good part of the performance advantages of separating positional and keyword arguments, because once again there is automatic conversion and "keywords split out from the last Array argument".
And that is not cheap, it's a pretty ugly merge of control flow and changing which arguments go where if both cases occur. And even if they don't there is the extra check.

It'd be interesting to implement this in TruffleRuby to measure, but there is no way this will happen before preview2.

> Handling more involved cases where the arguments are stored and can escape may require checking all call sites with splats and no explicit keywords, though.

Yes, exactly and that is the problem.
If we have a compatibility workaround with an overhead, at least let's localize it and not affect all call sites with a rest argument (and without kwargs).

> I assume in your example, initialize should accept *args and not args, as otherwise, you can't use ruby2_keywords.

Then ruby2_keywords needs to be moved to the 3 callers of build_middleware in
https://github.com/rails/rails/blob/1811e841166198bf86ae6de18d0971df77b932b4/actionpack/lib/action_dispatch/middleware/stack.rb#L92-L122
then.

> It seems odd to recommend we add another feature (Kernel#send_keyword_hash) to compliment a feature (Module#ruby2_keywords) that you are strongly recommending we remove in Ruby 3.0.

I think it's a straightforward way to mark exactly which call sites need the legacy conversion behavior.
There are likely rather few of those, and it seems worth to avoid degrading the performance of all other call sites with a *rest argument and no kwargs.

#8 - 10/23/2019 02:50 AM - jeremyevans0 (Jeremy Evans)

Eregon (Benoit Daloze) wrote:

> jeremyevans0 (Jeremy Evans) wrote:
>
> > In CRuby, it's not significantly more complicated. The callee may need to add the literal keywords to a hash anyway (if callee uses a keyword splat). Moving that hash to the last positional argument is very little work.
>
> It's still a condition in the callee which needs to handle both kinds of callers: passing a Hash and passing literal keywords, with completely different ways to read keyword arguments from that.
> That's likely rather inefficient if it happens in practice.

As I stated last time, it's not that inefficient in CRuby. I'll try to benchmark to compare it tomorrow, but I expect it would make only a small difference in a microbenchmark, and probably no significant difference in a real world benchmark.

> A sufficiently advanced compiler in a more optimized Ruby implementation should be able to determine in most cases whether the method is ever called with such a hash, and optimize the check out, correct? :)

I don't think so. As soon as one case stores a flagged Hash we'd have to check *all* call sites using a *rest argument and no kwargs. ActionDispatch already has such a case.

Agreed.

> I think that check can only be optimized out if we create the Hash in the same compilation unit. If not, we'll have to check the flag, on every such call.
> I think you need to believe my word here that this is not free and this is significant enough that we don't want to shoot ourselves for all future Ruby versions with that overhead.

Sorry, but I won't take your word as to the extent of the performance difference. You shouldn't even take your word, as you haven't implemented it yet. I'll post a benchmark, and I ask you to do the same.

> For me, it ruins a good part of the performance advantages of separating positional and keyword arguments, because once again there is automatic conversion and "keywords split out from the last Array argument".

Separating positional and keyword arguments was not done for performance reasons, it was done to avoid the issues that not separating them caused.

> And that is not cheap, it's a pretty ugly merge of control flow and changing which arguments go where if both cases occur. And even if they don't there is the extra check.

Again, please post a benchmark so we can discuss actual and not theoretical performance differences.

> It'd be interesting to implement this in TruffleRuby to measure, but there is no way this will happen before preview2.

Considering preview2 was released before this was posted, I agree. :)

> Handling more involved cases where the arguments are stored and can escape may require checking all call sites with splats and no explicit keywords, though.

> Yes, exactly and that is the problem.
> If we have a compatibility workaround with an overhead, at least let's localize it and not affect all call sites with a rest argument (and without kwargs).

The initial proposal for ruby2_keywords used a VM frame flag and not a hash object flag, and was thus localized. Unfortunately, it didn't handle cases that people wanted it to handle. I think Ruby's goal of programmer happiness should outweigh minor theoretical performance issues.

> I assume in your example, initialize should accept *args and not args, as otherwise, you can't use ruby2_keywords.

> Then ruby2_keywords needs to be moved to the 3 callers of build_middleware in https://github.com/rails/rails/blob/1811e841166198bf86ae6de18d0971df77b932b4/actionpack/lib/action_dispatch/middleware/stack.rb#L92-L122 then.

Correct.

> It seems odd to recommend we add another feature (Kernel#send_keyword_hash) to compliment a feature (Module#ruby2_keywords) that you are strongly recommending we remove in Ruby 3.0.

> I think it's a straightforward way to mark exactly which call sites need the legacy conversion behavior.
> There are likely rather few of those, and it seems worth to avoid degrading the performance of all other call sites with a *rest argument and no kwargs.

Kernel#send_keyword_hash requires changing the method definition or adding a separate method definition, when one of the main benefits of ruby2_keywords is that you do not need to modify an existing method definition.

#9 - 10/23/2019 11:35 AM - mame (Yusuke Endoh)

NuriYuri (Youri Nouri) Thank you for your comment, but this ticket is not a good place to discuss it. Please add it to #14183 or create a new ticket.

#10 - 10/23/2019 04:05 PM - Eregon (Benoit Daloze)

jeremyevans0 (Jeremy Evans) wrote:

> You shouldn't even take your word, as you haven't implemented it yet. I'll post a benchmark, and I ask you to do the same.

Fair enough, I'll try to benchmark this soon in TruffleRuby then.
If I can have a call with you it would be helpful to discuss the conceptual implementation for that logic and what examples are good to compare (and ideas related to delegation).

> > For me, it ruins a good part of the performance advantages of separating positional and keyword arguments, because once again there is automatic conversion and "keywords split out from the last Array argument".
>
> Separating positional and keyword arguments was not done for performance reasons, it was done to avoid the issues that not separating them caused.

I know, but I think separating arguments can be an opportunity for better performance performance and cheaper method/block calls, and this partially removes that opportunity.

Also on the semantics level, I think it would be cleaner if there are no ways to automatically convert a positional Hash to kwargs or vice-versa.
Don't you agree?

By having ruby2_keywords in Ruby 3, we're probably going to see some abuse of it, and some confusion about it.

If the goal is to separate positional and keyword arguments, why are we leaving a backdoor?

> The initial proposal for ruby2_keywords used a VM frame flag and not a hash object flag, and was thus localized. Unfortunately, it didn't handle cases that people wanted it to handle. I think Ruby's goal of programmer happiness should outweigh minor theoretical performance issues.

Yes the frame flag approach was better performance-wise in that it would only affect a known small set of call sites.

> > I think it's a straightforward way to mark exactly which call sites need the legacy conversion behavior.
> > There are likely rather few of those, and it seems worth to avoid degrading the performance of all other call sites with a *rest argument and no kwargs.
>
> Kernel#send_keyword_hash requires changing the method definition or adding a separate method definition, when one of the main benefits of ruby2_keywords is that you do not need to modify an existing method definition.

Yes, it requires changing the method definition, but in a fairly obvious way.
I would argue everyone changing a delegation method knows very well which is the delegating call site, and marking it with send_keyword_hash is completely trivial.

I'd think it's even useful to mark and correspondingly to not mark some call sites which should not pass kwargs.

The current logic where Array#dup dup's the last argument if it's a tagged Hash (6081ddd6e6f2297862b3c7e898d28a76b8f9240b)
seems a particularly not nice workaround for the fact a tagged Hash might be used as kwargs multiple times.
#send_keyword_hash would solve this in a more clean, explicit and clear way.

I don't think there is much of a difference to add ruby2_keywords, a "visibility modifier" before def vs modifying a call site.
Both need modifications of the file.

#11 - 10/24/2019 10:46 PM - jeremyevans0 (Jeremy Evans)

Eregon (Benoit Daloze) wrote:

> jeremyevans0 (Jeremy Evans) wrote:
>
> > You shouldn't even take your word, as you haven't implemented it yet. I'll post a benchmark, and I ask you to do the same.
>
> Fair enough, I'll try to benchmark this soon in TruffleRuby then.

As we discussed, here is a link to the benchmark: https://pastebin.com/raw/inwr6GvW

You'll want to compare the master branch with a branch that removes ruby2_keywords. When I ran this yesterday (before some ruby2_keywords changes were merged today), this was the diff I used for for removal: https://pastebin.com/raw/izk2qesi . This isn't a complete removal, just a removal of the code from the hot paths so we can see the performance difference.

On CRuby master branch, in the worst possible case I could design, the difference was about 1%.

Note that ruby2_keywords approach for delegation is over twice as fast as explicit keyword delegation due to reduced object allocation (see below). If you have 300 method(*args) calls for every 1 call to a method that uses ruby2_keywords for delegation instead of explicit keyword delegation, keeping ruby2_keywords still improves performance.

> > For me, it ruins a good part of the performance advantages of separating positional and keyword arguments, because once again there is automatic conversion and "keywords split out from the last Array argument".
>
> Separating positional and keyword arguments was not done for performance reasons, it was done to avoid the issues that not separating them caused.

I know, but I think separating arguments can be an opportunity for better performance performance and cheaper method/block calls, and this partially removes that opportunity.

In CRuby, ruby2_keywords actually results in better performance. It is over twice as fast as explicit delegation using keywords. Here's a benchmark showing that: https://pastebin.com/raw/TNFJJSCs. So in CRuby, ruby2_keywords in general increases performance, not decreases it.

Argument delegation using ... now uses ruby2_keywords internally. This was done to avoid warnings and not for performance reasons, but it does increase performance substantially.

> Also on the semantics level, I think it would be cleaner if there are no ways to automatically convert a positional Hash to kwargs or vice-versa. Don't you agree?

Technically, **hash automatically converts a positional hash to keywords. So I definitely think we want a way to automatically convert from positional hash to keywords, otherwise all keyword arguments would need to be specified explicitly in every call.

> By having ruby2_keywords in Ruby 3, we're probably going to see some abuse of it, and some confusion about it.

I don't think we'll see major problems due to it, but my precognition has never been 100%.

> If the goal is to separate positional and keyword arguments, why are we leaving a backdoor?

This isn't a backdoor, it's a tunnel. It allows you to tunnel keyword arguments through a method that doesn't explicitly list itself as accepting keyword arguments. It doesn't have any of the issues that caused us to separate positional and keyword arguments. For the following code:

```
ruby2_keywords def foo(*args, &block)
  bar(*args)
end
```

Method bar knows that it is passed a positional hash if you call foo(h), and knows it is passed keywords if you call foo(**h).

Stating that ruby2_keywords violates or backtracks on keyword argument separation is wrong.

> > I think it's a straightforward way to mark exactly which call sites need the legacy conversion behavior.
> > There are likely rather few of those, and it seems worth to avoid degrading the performance of all other call sites with a *rest argument and no kwargs.
>
> Kernel#send_keyword_hash requires changing the method definition or adding a separate method definition, when one of the main benefits of ruby2_keywords is that you do not need to modify an existing method definition.

> Yes, it requires changing the method definition, but in a fairly obvious way.
> I would argue everyone changing a delegation method knows very well which is the delegating call site, and marking it with send_keyword_hash is completely trivial.

This would require adding a method to Kernel in addition to the ruby2_keywords method added to Module. It would also make the method slower on older versions of Ruby (and possibly also the current version of Ruby), since they would not be able to use the optimized version of send.

> I'd think it's even useful to mark and correspondingly to not mark some call sites which should not pass kwargs.
>
> The current logic where Array#dup dup's the last argument if it's a tagged Hash (6081ddd6e6f2297862b3c7e898d28a76b8f9240b)
> seems a particularly not nice workaround for the fact a tagged Hash might be used as kwargs multiple times.
> #send_keyword_hash would solve this in a more clean, explicit and clear way.

The commit referenced is not related to Array#dup. It calls rb_hash_dup on the last element, because that is the equivalent of what **hash does.

send_keyword_hash would need to dup the hash in the exact same way, otherwise the callee could modify the hash and have the changes reflected in the caller, which is not how keyword hashes are supposed to work.

> I don't think there is much of a difference to add ruby2_keywords, a "visibility modifier" before def vs modifying a call site.
> Both need modifications of the file.

Note that you cannot have send_keyword_hash without ruby2_keywords. At the very least, you are at least doubling the work required to get correct delegation if you add it.

Your arguments against ruby2_keywords in general boil down to these things:

- It is bad for performance. I have proven this not to be true in CRuby, and we are not yet sure the extent to which it affects other implementations. On CRuby, ruby2_keywords actually helps performance significantly due to fewer object allocations during delegation.
- Users will have to modify their code twice. First, this is only true if ruby2_keywords is removed. Removal has not been decided on, yet, though the reason it was renamed from pass_keywords to ruby2_keywords was because it was expected to be removed after Ruby 2.6 EOL. Second, your proposal for duplicate method definitions with a version check still would result in almost all users modifying their code twice, as almost no users want to have dead code in their codebase for ruby versions they no longer support. So there is no practical difference in the number of times users will need to modify their code, even if ruby2_keywords is removed at some future point.
- It's not semantic, going against keyword argument separation. As I explained above, it does not go against keyword argument separation, it works with it to handle delegation in a backwards compatible manner.
- It looks strange, and will look more strange in Ruby 4+. I kind of agree. I think the pass_keywords name was better. This doesn't seem like a very strong reason, though.

#12 - 10/27/2019 10:41 AM - Eregon (Benoit Daloze)

I want to expand on my semantics concern, for the performance concern I should get some numbers first.

jeremyevans0 (Jeremy Evans) wrote:

> Technically, **hash automatically converts a positional hash to keywords. So I definitely think we want a way to automatically convert from positional hash to keywords, otherwise all keyword arguments would need to be specified explicitly in every call.

Yes of course. What I meant is ruby2_keywords is the only thing which allows passing keyword arguments without explicitly using ** or literal keywords (foo(a: 1)) at the call site (those 2 are explicit and fine).
For instance, bar(*args) from the code above passes keyword arguments even though no Ruby code should be able to do that in 3.0, but ruby2_keywords is an exception here.

It's not so nice that somecall(*args) behaves differently depending on whether Hash === args[-1] && tagged?(args[-1]) (which is not a property of the lexical code, but a dynamic runtime flag).
So looking at some Ruby code in some gem, I cannot know if somecall(*args) may pass or never passes keyword arguments.
For example if my code defines somecall, that might make it unclear whether I should accept keyword arguments or not.
Having an explicit send_keyword_hash would fix that and mark the only places that might take *args but actually pass keyword arguments too.

mame (Yusuke Endoh) What do you think of that specific concern?

#13 - 11/24/2019 08:54 PM - Eregon (Benoit Daloze)

jeremyevans0 (Jeremy Evans) wrote:

> On CRuby master branch, in the worst possible case I could design, the difference was about 1%.

Why is def a(x) end; a(*arr) the worst case? Would it not be more expensive to call a method accepting keyword arguments, since then further checks might be needed?

I benchmarked MRI, comparing MRI 2.6.5 with MRI 2.7.0preview3, and I see overheads far above 1%, more in the 10%.
There is a bit of noise, even with turbo boost disabled and the performance CPU governor and using CLOCK_THREAD_CPUTIME_ID.
However, 2.7 is consistently slower for that benchmark in my measurements.

Comparing the last measurement for each:

- baseline: 2.7 is 1.3% faster
- req: 2.7 is 10.6% slower
- kw: 2.7 is 9.7% slower
- kwrest: 2.7 is 8% slower

See https://gist.github.com/eregon/15ebe02ff8f42c0ab964e1066a783f9d for all numbers and the benchmark.

Those overheads are probably not entirely due to ruby2_keywords (I would think a good part is, though).
The fact the baseline results are quite close seems to indicate that at least req(1) is not slower on Ruby 2.7 (in fact, it's a bit faster).
However, all 3 foo(*args) seem to show a general 8-10% slowdown in Ruby 2.7, which I'd guess is due to ruby2_keywords.

#14 - 11/24/2019 09:56 PM - Eregon (Benoit Daloze)

I also measured on TruffleRuby, and there the diff is minimal, just adding the ruby2_keywords check on *splat call sites:
https://github.com/oracle/truffleruby/commit/d143af3626aae009e2414bfe61833565fe3a0476

The results are similar (details on https://gist.github.com/eregon/15ebe02ff8f42c0ab964e1066a783f9d):

- req: 4.6% slower
- kw: 10.4% slower
- kwrest: 10.9% slower

In summary, I see about 10% slowdown on this micro benchmark, representative of foo(*args) calls, just by the extra ruby2_keywords check.

#15 - 11/24/2019 10:38 PM - Eregon (Benoit Daloze)

Here is another benchmark, where no keyword arguments are used, yet we see a slowdown of up to 11.5% in MRI 2.7:
https://gist.github.com/eregon/31e155901c995925bd1c661dfa1a71d8

length([1], [2], [3]) and length({a: 1}, {b: 2}, {c: 3}) are essentially the baselines.
On TruffleRuby, those two incur no overhead because they cause no allocation (escape analysis) and
the JIT can see arguments don't have the ruby2_keywords flag.

The other cases cannot be optimized that way, and need to check for the ruby2_keywords flag.

TruffleRuby:

- length(*ARRAYS): 5.9% slower
- length(*HASHES): 4.4% slower

MRI 2.7.0preview3 vs MRI 2.6.5:

- length(*ARRAYS): 2.7 is 11.5% slower
- length(*HASHES): 2.7 is 10.3% slower

Do we want all foo(*args) calls to get that overhead in Ruby 3+?

I think we should either:

- Remove ruby2_keywords in Ruby 3.0, just have it in Ruby 2.7 where it's needed. Ruby 3.0 (with the keyword arg separation) doesn't need ruby2_keywords.
- Combine ruby2_keywords with send_keyword_hash, which solves the performance issue and is explicit, therefore improving readability and debug-ability.
- Use another way for delegation in Ruby 2.7 (e.g. the lexical pass_keywords or ..., see https://eregon.me/blog/2019/11/10/the-delegation-challenge-of-ruby27.html)

cc matz (Yukihiro Matsumoto)mame (Yusuke Endoh)jeremyevans0 (Jeremy Evans)

#16 - 11/24/2019 10:41 PM - jeremyevans0 (Jeremy Evans)

Eregon (Benoit Daloze) wrote:

> jeremyevans0 (Jeremy Evans) wrote:
>
> > On CRuby master branch, in the worst possible case I could design, the difference was about 1%.
>
> Why is def a(x) end; a(*arr) the worst case? Would it not be more expensive to call a method accepting keyword arguments, since then further checks might be needed?

def a(x) end; a(*arr) end is the worst case performance decrease, assuming that what we are measuring is the effect of supporting ruby2_keywords when not actually using the feature. If the final hash does not have the keyword flag, no changes are made. Methods accepting keyword arguments have more overhead than methods not accepting keyword arguments in CRuby, so there will be a larger percentage difference when calling methods that do not accept keyword arguments.

> I benchmarked MRI, comparing MRI 2.6.5 with MRI 2.7.0preview3, and I see overheads far above 1%, more in the 10%.

Comparing 2.6 to 2.7 is irrelevant in regards to the discussion of the effect of ruby2_keywords. There are many other changes between 2.6 and 2.7 that have a much larger effect than ruby2_keywords. If you want to measure the effect of ruby2_keywords, you need to benchmark the master branch against the master branch with the removal of ruby2_keywords, as I did in an earlier comment, and as your TruffleRuby benchmark did.

> However, all 3 foo(*args) seem to show a general 8-10% slowdown in Ruby 2.7, which I'd guess is due to ruby2_keywords.

I think your guess is wrong. If you want to test the effect of ruby2_keywords, you have to benchmark with that change in isolation, not in combination with all of the other changes between 2.6 and 2.7.

Eregon (Benoit Daloze) wrote:

> I also measured on TruffleRuby, and there the diff is minimal, just adding the ruby2_keywords check on *splat call sites:
> https://github.com/oracle/truffleruby/commit/d143af3626aae009e2414bfe61833565fe3a0476
>
> The results are similar (details on https://gist.github.com/eregon/15ebe02ff8f42c0ab964e1066a783f9d):
>
> - req: 4.6% slower
> - kw: 10.4% slower
> - kwrest: 10.9% slower
>
> In summary, I see about 10% slowdown on this micro benchmark, representative of foo(*args) calls, just by the extra ruby2_keywords check.

Thanks for working on a microbenchmark for TruffleRuby. This shows about a maximum of 11% slowdown for calls using splats without keyword splats. Considering the percentage of calls using splats without keyword splats, compared to all other calls, it seems unlikely this change will have a significant effect in a real world benchmark on TruffleRuby.

#17 - 11/24/2019 10:57 PM - jeremyevans0 (Jeremy Evans)

Eregon (Benoit Daloze) wrote:

> Here is another benchmark, where no keyword arguments are used, yet we see a slowdown of up to 11.5% in MRI 2.7:
> https://gist.github.com/eregon/31e155901c995925bd1c661dfa1a71d8
>
> length([1], [2], [3]) and length({a: 1}, {b: 2}, {c: 3}) are essentially the baselines.
> On TruffleRuby, those two incur no overhead because they cause no allocation (escape analysis) and
> the JIT can see arguments don't have the ruby2_keywords flag.
>
> The other cases cannot be optimized that way, and need to check for the ruby2_keywords flag.
>
> TruffleRuby:
>
> - length(*ARRAYS): 5.9% slower
> - length(*HASHES): 4.4% slower
>
> MRI 2.7.0preview3 vs MRI 2.6.5:
>
> - length(*ARRAYS): 2.7 is 11.5% slower
> - length(*HASHES): 2.7 is 10.3% slower
>
> Do we want all foo(*args) calls to get that overhead in Ruby 3+?

As I explained in my previous comment, the majority of the slowdown is not related to ruby2_keywords. Remove the ruby2_keywords code and you are likely to see roughly the same slowdown. You need to backout ruby2_keywords from the CRuby master branch in order to get a proper benchmark of the effect of ruby2_keywords. I did this in my earlier benchmark and showed that ruby2_keywords in isolation has only about a ~1% effect in CRuby.

> I think we should either:
>
> - Remove ruby2_keywords in Ruby 3.0, just have it in Ruby 2.7 where it's needed. Ruby 3.0 (with the keyword arg separation) doesn't need ruby2_keywords.

ruby2_keywords is about 2x faster than explicit keyword arguments in CRuby, so this would actually decrease performance in CRuby.

> - Combine ruby2_keywords with send_keyword_hash, which solves the performance issue and is explicit, therefore improving readability and debug-ability.

This requires modifying the internals of methods instead of just flagging the methods, and is much more invasive to the user.

> - Use another way for delegation in Ruby 2.7 (e.g. the lexical pass_keywords or ..., see
> https://eregon.me/blog/2019/11/10/the-delegation-challenge-of-ruby27.html)

The lexical pass_keywords is not truly lexical, as only the current VM frame was flagged, so the behavior inside blocks in a method was not what the user would expect. Modifying the implementation to handle lexical VM frames could possibly result in more slowdown, and I'm not sure how to

implement it. Additionally, there are cases where non-lexical passing is used (e.g. in Rails), and a lexical approach would not handle those cases. ruby2_keywords handles that case, and many other real world cases that the lexical approach does not handle.

... doesn't handle all delegation cases, it only handles a subset where all arguments are passed and no arguments are added/removed/changed. There are many cases where it cannot be used.

#18 - 11/24/2019 11:04 PM - Eregon (Benoit Daloze)

jeremyevans0 (Jeremy Evans) wrote:

> Comparing 2.6 to 2.7 is irrelevant in regards to the discussion of the effect of ruby2_keywords. There are many other changes between 2.6 and 2.7 that have a much larger effect than ruby2_keywords.

Which other changes would affect calls like req(*arr)?
AFAIK, there is no change for that in 2.7 except ruby2_keywords, is it?

> If you want to measure the effect of ruby2_keywords, you need to benchmark the master branch against the master branch with the removal of ruby2_keywords, as I did in an earlier comment, and as your TruffleRuby benchmark did.

I'm concerned your diff above still keeps some overhead, e.g., there are still references to the ruby2_keywords flag.

> Considering the percentage of calls using splats without keyword splats compared to all other calls,

How much do you think is that percentage? Right now I would expect far more *args than *args, **kwargs.

> it seems unlikely this change will have a significant effect in a real world benchmark on TruffleRuby.

I think it can actually, every foo(*args) call, typically used for delegating arguments to another method is can be up to ~11% slower.
Some gems use foo(*args) quite often.
Typically, method_missing uses it too, and method_missing can be performance-sensitive (https://twitter.com/headius/status/1197972352488767489).

#19 - 11/25/2019 12:00 AM - Eregon (Benoit Daloze)

https://github.com/ruby/ruby/compare/master...eregon:no-ruby2_keywords is a more extensive removal of ruby2_keywords, although not complete yet.
It's actually a lot of extra logic in setup_parameters_complex.

#20 - 11/25/2019 12:22 AM - Eregon (Benoit Daloze)

jeremyevans0 (Jeremy Evans) wrote:

- Remove ruby2_keywords in Ruby 3.0, just have it in Ruby 2.7 where it's needed. Ruby 3.0 (with the keyword arg separation) doesn't need ruby2_keywords.

> ruby2_keywords is about 2x faster than explicit keyword arguments in CRuby, so this would actually decrease performance in CRuby.

I don't think anyone is seriously considering using ruby2_keywords in Ruby 3+ (or at least in Ruby 3.3+).
So it seems CRuby should improve performance for *args, **kwargs, or propose another way to delegate more efficiently (e.g., ... is one way I think many Rubyists would like).

- Combine ruby2_keywords with send_keyword_hash, which solves the performance issue and is explicit, therefore improving readability and debug-ability.

> This requires modifying the internals of methods instead of just flagging the methods, and is much more invasive to the user.

It requires extra modifications, yes, but IMHO very easy modifications: marking where positional args should be converted to kwargs for delegation.
Since users have to think where they need ruby2_keywords, I would argue placing send_keyword_hash is no harder than adding ruby2_keywords, and makes the whole model a lot easier to understand, less magic and fixes performance.

- Use another way for delegation in Ruby 2.7 (e.g. the lexical pass_keywords or ..., see https://eregon.me/blog/2019/11/10/the-delegation-challenge-of-ruby27.html)

> The lexical pass_keywords is not truly lexical, as only the current VM frame was flagged, so the behavior inside blocks in a method was not what the user would expect. Modifying the implementation to handle lexical VM frames could possibly result in more slowdown, and I'm not sure how to implement it.

I do mean lexical, including calls inside blocks.

I would think it's straightforward to implement pass_keywords, including blocks, without overhead on unrelated code:
Modify every call site inside pass_keywords methods with *args, including those in blocks inside the method, to use a different bytecode.
In that bytecode's implementation, do the extra logic needed.
We might also need some prelude logic in the marked method to remember if it was passed kwargs.
And that's all there is to it, isn't it?

> Additionally, there are cases where non-lexical passing is used (e.g. in Rails), and a lexical approach would not handle those cases.

I showed it can be done with a block: https://github.com/eregon/rails/commit/8b0625ed68

> ruby2_keywords handles that case, and many other real world cases that the lexical approach does not handle.

Can you give an example the above approach (capturing the call inside the lambda) cannot handle?

> ... doesn't handle all delegation cases, it only handles a subset where all arguments are passed and no arguments are added/removed/changed.
> There are many cases where it cannot be used.

Yes, we should at least allow leading arguments as that's so frequent.
I'll file an issue for it if there isn't one already.
I believe everyone expects def m(a, ...) to work.

I think ... is actually what most Rubyists want.
I think the only real difficulty with ... is how to use in older Ruby versions (eval is one possibility).
Maybe we should backport ... to 2.4/2.5/2.6.

#21 - 11/25/2019 02:42 AM - jeremyevans0 (Jeremy Evans)

Eregon (Benoit Daloze) wrote:

> jeremyevans0 (Jeremy Evans) wrote:
>
> > Comparing 2.6 to 2.7 is irrelevant in regards to the discussion of the effect of ruby2_keywords. There are many other changes between 2.6 and 2.7 that have a much larger effect than ruby2_keywords.
>
> Which other changes would affect calls like req(*arr)?
> AFAIK, there is no change for that in 2.7 except ruby2_keywords, is it?

There are a ton of other changes in 2.7 that could affect performance besides ruby2_keywords.

> If you want to measure the effect of ruby2_keywords, you need to benchmark the master branch against the master branch with the removal of ruby2_keywords, as I did in an earlier comment, and as your TruffleRuby benchmark did.

> I'm concerned your diff above still keeps some overhead, e.g., there are still references to the ruby2_keywords flag.

I don't think those references are in a case the benchmark would hit, but I could be wrong. As you mentioned in a later comment, you prepared your own diff for just removing ruby2_keywords.

> Considering the percentage of calls using splats without keyword splats compared to all other calls,

> How much do you think is that percentage? Right now I would expect far more *args than *args, **kwargs.

There are probably more *args than *args, **kwargs. However, that is not what is important. I would guess that method calls with no arguments or only explicit arguments outnumber calls with splats at least 10-1 if not 100-1. So a 10% performance difference would be 0.1-1% difference in a real world benchmark.

> it seems unlikely this change will have a significant effect in a real world benchmark on TruffleRuby.

> I think it can actually, every foo(*args) call, typically used for delegating arguments to another method is can be up to ~11% slower.
> Some gems use foo(*args) quite often.
> Typically, method_missing uses it too, and method_missing can be performance-sensitive (
> https://twitter.com/headius/status/1197972352488767489).

Well, we could guess, or you could pick an existing real world benchmark and run it and report the results for:

- CRuby master
- CRuby master with ruby2_keywords removed
- TruffleRuby master
- TruffleRuby master with ruby2_keywords added

I would be surprised if there was a measurable difference.

> https://github.com/ruby/ruby/compare/master...eregon:no-ruby2_keywords is a more extensive removal of ruby2_keywords, although not complete yet.
> It's actually a lot of extra logic in setup_parameters_complex.

Do all non-ruby2_keywords tests still pass with that? If so, what are the benchmark results with the patch?

> I don't think anyone is seriously considering using ruby2_keywords in Ruby 3+ (or at least in Ruby 3.3+).
> So it seems CRuby should improve performance for *args, **kwargs, or propose another way to delegate more efficiently (e.g., ... is one way I think many Rubyists would like).

I disagree. I think that assuming it works better, most Ruby programmers will consider using it. Now, if explicit keyword argument performance is improved so that ruby2_keywords is no longer the faster way, then I could see there no longer be a need for it. However, that is far from a forgone conclusion.

> It requires extra modifications, yes, but IMHO very easy modifications: marking where positional args should be converted to kwargs for delegation.

Marking methods that do delegation is much easier than modifying the internals of methods, especially in more complex cases such as when the delegating method stores the arguments and they are sent to a target method in a separate method.

> Since users have to think where they need ruby2_keywords, I would argue placing send_keyword_hash is no harder than adding ruby2_keywords, and makes the whole model a lot easier to understand, less magic and fixes performance.

It's definitely more difficult, especially in complex cases. I don't think it is easier to understand or less magic, it is just more explicit. As to the performance issues, they are very minor in CRuby and based on your own benchmark fairly minor in TruffleRuby.

> - Use another way for delegation in Ruby 2.7 (e.g. the lexical pass_keywords or ..., see https://eregon.me/blog/2019/11/10/the-delegation-challenge-of-ruby27.html)
>
> The lexical pass_keywords is not truly lexical, as only the current VM frame was flagged, so the behavior inside blocks in a method was not what the user would expect. Modifying the implementation to handle lexical VM frames could possibly result in more slowdown, and I'm not sure how to implement it.
>
> I do mean lexical, including calls inside blocks.
>
> I would think it's straightforward to implement pass_keywords, including blocks, without overhead on unrelated code:
> Modify every call site inside pass_keywords methods with *args, including those in blocks inside the method, to use a different bytecode.
> In that bytecode's implementation, do the extra logic needed.
> We might also need some prelude logic in the marked method to remember if it was passed kwargs.
> And that's all there is to it, isn't it?

It sounds so simple. I look forward to your patch implementing it, as I'm sure I could learn much from it.

> Additionally, there are cases where non-lexical passing is used (e.g. in Rails), and a lexical approach would not handle those cases.
>
> I showed it can be done with a block: https://github.com/eregon/rails/commit/8b0625ed68

That's not pass_keywords handling the case, that's you modifying the code so that pass_keywords can handle it. Surely you can see how that is a more difficult change to make.

> ruby2_keywords handles that case, and many other real world cases that the lexical approach does not handle.
>
> Can you give an example the above approach (capturing the call inside the lambda) cannot handle?

No. Certainly there is a way to always move code into the lexical scope. It is a more difficult change and it makes the resulting code harder to understand, but it is possible.

#22 - 11/27/2019 03:38 PM - Eregon (Benoit Daloze)

Eregon (Benoit Daloze) wrote:

> https://github.com/ruby/ruby/compare/master...eregon:no-ruby2_keywords is a more extensive removal of ruby2_keywords, although not complete yet.
> It's actually a lot of extra logic in setup_parameters_complex.

With that change applied I could see a slowdown of 3-5% locally, although with quite some noise between runs.

I'm also not sure if that is the full picture, because there is still ruby2_keywords-related code with that diff, so it could be more too.

#23 - 11/27/2019 03:49 PM - Eregon (Benoit Daloze)

jeremyevans0 (Jeremy Evans) wrote:

> There are a ton of other changes in 2.7 that could affect performance besides ruby2_keywords.

Any specific guess?

> I don't think those references are in a case the benchmark would hit, but I could be wrong. As you mentioned in a later comment, you prepared your own diff for just removing ruby2_keywords.

Your original diff kept quite a lot more complicated code in setup_parameters_complex, so I think that diff is already more precise and seems to show an higher overhead (see my previous comment).

> So a 10% performance difference would be 0.1-1% difference in a real world benchmark.

Still, if there are foo(*args) calls in a performance-sensitive part of a real world benchmark I'd expect it can be noticeable.

> Well, we could guess, or you could pick an existing real world benchmark and run it and report the results for:

OptCarrot might be affected, it uses send(*DISPATCH[@opcode]) in the main loop of the CPU:
https://github.com/mame/optcarrot/blob/ded9d9379324d968867d1e052dbbc811d45afd4d/lib/optcarrot/cpu.rb#L939
However the CPU is only a small part compared to the PPU in OptCarrot.

> Do all non-ruby2_keywords tests still pass with that? If so, what are the benchmark results with the patch?

I'd think so, I just manually removed dead code based on RHASH_PASS_AS_KEYWORDS never happening.

> > Can you give an example the above approach (capturing the call inside the lambda) cannot handle?
>
> No. Certainly there is a way to always move code into the lexical scope. It is a more difficult change and it makes the resulting code harder to understand, but it is possible.

I think it's only more difficult in rather rare cases, and it makes it significantly easier to understand and use than ruby2_keywords.

#24 - 11/27/2019 03:59 PM - Eregon (Benoit Daloze)

jeremyevans0 (Jeremy Evans) wrote:

> Do all non-ruby2_keywords tests still pass with that? If so, what are the benchmark results with the patch?

Yes, all specs pass and test-all has just 2 expected failures:

```
  1) Failure:
TestKeywordArguments#test_ruby2_keywords [/home/eregon/code/ruby/test/ruby/test_keyword.rb:2794]:
<[[{:a=>1}], {}]> expected but was
<[[], {:a=>1}]>.

[ 92/226] TestSyntax#test_argument_forwarding = 0.00 s
  2) Failure:
TestSyntax#test_argument_forwarding [/home/eregon/code/ruby/test/ruby/test_syntax.rb:1515]:
```

```
--- expected
+++ actual
@@ -1 +1,4 @@
-""
+"/home/eregon/code/ruby/test/ruby/test_syntax.rb:1485: warning: The last argument is used as the keyword para
meter
+" +
+"/home/eregon/code/ruby/test/ruby/test_syntax.rb:1478: warning: for `bar' defined here
+"

20722 tests, 2699145 assertions, 2 failures, 0 errors, 114 skips
```

#25 - 11/27/2019 04:19 PM - jeremyevans0 (Jeremy Evans)

Eregon (Benoit Daloze) wrote:

> jeremyevans0 (Jeremy Evans) wrote:
>
> > There are a ton of other changes in 2.7 that could affect performance besides ruby2_keywords.
>
> Any specific guess?

After discussion with you and mame (Yusuke Endoh) , it appears my benchmarking was flawed and the performance difference is closer to your measurements.

> Your original diff kept quite a lot more complicated code in setup_parameters_complex, so I think that diff is already more precise and seems to show an higher overhead (see my previous comment).

Actually, the only relevant difference is the same in both your and my removal diffs, the code in CALLER_SETUP_ARG, not the code in setup_parameters_complex.

> So a 10% performance difference would be 0.1-1% difference in a real world benchmark.

Still, if there are foo(*args) calls in a performance-sensitive part of a real world benchmark I'd expect it can be noticeable.

Possibly. A real world benchmark would be beneficial to determine that.

> Well, we could guess, or you could pick an existing real world benchmark and run it and report the results for:

OptCarrot might be affected, it uses send(*DISPATCH[@opcode]) in the main loop of the CPU:
https://github.com/mame/optcarrot/blob/ded9d9379324d968867d1e052dbbc811d45afd4d/lib/optcarrot/cpu.rb#L939
However the CPU is only a small part compared to the PPU in OptCarrot.

OptCarrot would be a good choice for a real world benchmark for this.

> Do all non-ruby2_keywords tests still pass with that? If so, what are the benchmark results with the patch?

I'd think so, I just manually removed dead code based on RHASH_PASS_AS_KEYWORDS never happening.

From your later testing, all tests pass, so your removal diff appears correct.

> > Can you give an example the above approach (capturing the call inside the lambda) cannot handle?
>
> No. Certainly there is a way to always move code into the lexical scope. It is a more difficult change and it makes the resulting code harder to understand, but it is possible.

I think it's only more difficult in rather rare cases, and it makes it significantly easier to understand and use than ruby2_keywords.

It is always more difficult to move code compared to not moving code. How much more difficult depends on the situation.

The discussion of lexical pass_keywords is academic anyway without a proposed plan/diff for implementing it. Your proposal only works if you can move code into blocks in the lexical scope, and pass_keywords as implemented in a previous pull request did not support that.

#26 - 11/27/2019 04:45 PM - Eregon (Benoit Daloze)

mame (Yusuke Endoh)'s measurements:

```
def req(x)
  x
end
hash = {a: 1}
arr = [hash]
100000000.times { req(*arr) }
```

gives

```
$ time ./miniruby.check bench.rb
real    0m5.726s
user    0m5.689s
sys 0m0.025s
$ time ./miniruby.nocheck bench.rb
real    0m5.318s
user    0m5.318s
sys 0m0.000s
$ time ./miniruby.check bench.rb
real    0m5.849s
user    0m5.845s
sys 0m0.004s
$ time ./miniruby.nocheck bench.rb
real    0m5.395s
user    0m5.392s
sys 0m0.000s
```

That's 7% and 8.4% overhead on user time.

#27 - 06/05/2020 03:51 PM - Eregon (Benoit Daloze)

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

#28 - 06/05/2020 03:52 PM - Eregon (Benoit Daloze)

*- Related to Feature #16463: Fixing *args-delegation in Ruby 2.7: ruby2_keywords semantics by default in 2.7.1 added*