Ruby - Feature #18959

Handle gracefully nil kwargs eg. **nil

08/08/2022 04:14 PM - LevLukomskyi (Lev Lukomskyi)

Status:	Open
Priority:	Normal
Assignee:	
Target version:	

Description

The issue:

```
def qwe(a: 1) end

qwe(**nil) #=> fails with `no implicit conversion of nil into Hash (TypeError)` error

{ a:1, **nil } #=>  fails with `no implicit conversion of nil into Hash (TypeError)` error
```

Reasoning:

I found myself that I often want to insert a key/value to hash if a certain condition is met, and it's very convenient to do this inside hash syntax, eg.:

```
{
  some: 'value',
  **({ id: id } if id.present?),
}
```

Such syntax is much more readable than:

```
h = { some: 'value' }
h[:id] = id if id.present?
h
```

Yes, it's possible to write like this:

```
{
  some: 'value',
  **(id.present? ? { id: id } : {}),
}
```

but it adds unnecessary boilerplate noise.

I enjoy writing something like this in ruby on rails:

```
content_tag :div, class: [*('is-hero' if hero), *('is-search-page' if search_page)].presence
```

If no conditions are met then the array is empty, then converted to nil by presence, and class attribute is not rendered if it's nil. It's short and so convenient! There should be a similar way for hashes!

I found this issue here: https://bugs.ruby-lang.org/issues/8507 where "consistency" thing is discussed. While consistency is the right thing to do, I think the main point here is to have fun with programming, and being able to write stuff in a concise and readable way.

Please, add this small feature to the language, that'd be so wonderful! □□

History

#1 - 08/08/2022 04:52 PM - jeremyevans0 (Jeremy Evans)

This was also rejected in #9291. However, since this is a feature request with a use case, I guess it can be considered again.

Note that you can opt in to the behavior you want with one line (def nil.to_hash = {}), so this behavior is already available for users who want it.

#2 - 08/08/2022 07:02 PM - LevLukomskyi (Lev Lukomskyi)

Oh wow, I didn't know that def nil.to_hash = {} works perfectly well! Many thanks!

#3 - 08/10/2022 04:53 PM - Dan0042 (Daniel DeLorme)

Monkeypatching nil.to_hash seems a veeeery bad idea. For example

```
h = {"y"=>"!"}
"xyz".sub(/y/, h) #=> "x!z"

h = nil
"xyz".sub(/y/, h) #=> TypeError (no implicit conversion of nil into String)

def nil.to_hash; {}; end
"xyz".sub(/y/, h) #=> "xz" (segfault in 2.6!)
```

Honestly I would love for this feature to be accepted. Having a special case for **nil would be ok, but ideally the double-splat should call to_h. When we do **obj it means that we're explicitly expecting/asking for a Hash object. So why use the implicit to_hash method? I've always found it disconcerting.

#4 - 08/20/2022 10:01 PM - ioquatix (Samuel Williams)

I have another use case.

I'm receiving options from a web protocol.

Sometimes there are no options.

So I end up with config = [["Thing1", nil], ["Thing2", {option: "value"}]]

To invoke a method with those options, I have to write:

```
config.each do |name, options|
  setup(name, **(options || {}))
end
```

This approach probably immediately calls dup on the empty hash which seems inefficient.

Since **nil is a type error currently, extending this behaviour seems okay to me. But I've only been thinking about my own use case, maybe there are some negative use cases that I haven't thought about.

As another data point, this *does* work:

```
def foo(*arguments)
end

foo(*nil) # okay

a = [1]
b = nil
[*a, *b] # okay
```

So it seems reasonable that this works too:

```
def foo(**options)
end

foo(**nil) # TypeError

a = {x: 1}
b = nil
{**a, **b} # TypeError
```

#5 - 08/21/2022 05:12 AM - sawa (Tsuyoshi Sawada)

In my opinion, putting the whole subhash {id: id} within a condition like that is redundant and ugly.

That is one (but not the only) reason that I had asked for Hash#compactin #11818 (which is available). Using this, your code can be rewritten as:

```
{
  some: 'value',
  id: id if id.present?,
}
.compact
```

although I would actually do:

```
{
  some: 'value',
  id: id.presence,
}
.compact
```

#6 - 08/24/2022 11:09 AM - LevLukomskyi (Lev Lukomskyi)

compact way looks good indeed, although:

1. it removes all nils from the hash, which might be undesired in some cases
2. it creates hash duplicate each time, while kwargs way creates temporary hash only if a condition is met, though I believe interpreter might optimize that in future

#7 - 08/24/2022 02:44 PM - sawa (Tsuyoshi Sawada)

LevLukomskyi (Lev Lukomskyi) wrote in #note-6:

> compact way looks good indeed, although:
>
> 1. it removes all nils from the hash, which might be undesired in some cases

I think such use case would be a design failure. nil generally represents absence of a valid value. Sometimes having nil and sometimes not is inconsistent. (If you apply compact while possibly retaining false values, nothing is wrong with that.)

> 1. it creates hash duplicate each time, while kwargs way creates temporary hash only if a condition is met, though I believe interpreter might optimize that in future

If you want to avoid temporary hashes, you can use compact! instead of compact.

#8 - 08/24/2022 08:16 PM - LevLukomskyi (Lev Lukomskyi)

1. => more specifically .tap(&:compact!) since compact! may return nil if nothing changed.
2. there might be also a case when you want to merge another hash under some conditions, eg.:

```
{
  some: 'value',
  **({ id: id, name: name } if id.present?),
}
```

in this case compact won't help

#9 - 08/25/2022 03:47 AM - sawa (Tsuyoshi Sawada)

LevLukomskyi (Lev Lukomskyi) wrote in #note-8:

> 1. => more specifically .tap(&:compact!) since compact! may return nil if nothing changed.

I think tap(&:foo) is actually not that much efficient, and tap(&:compact!) could be slower than using compact. I meant to simply use compact! in a different statement, not in a chain. Any way, this kind of micro optimization should not matter that much.

> 1. there might be also a case when you want to merge another hash under some conditions, eg.:
>
> ```
> {
>   some: 'value',
>   **({ id: id, name: name } if id.present?),
> }
> ```

If you want to handle the whole subhash as one item, I think you should move that outside of the hash:

```
foo = {
  some: 'value',
}

foo.merge!({id: id, name: name} if id.present?
```

#10 - 08/25/2022 11:36 AM - LevLukomskyi (Lev Lukomskyi)

> I meant to simply use compact! in a different statement, not in a chain.

If use compact! as a different statement then you need to make a variable, you have to make a mental effort to make up the name for that variable and repeat its name to use compact!. Also, you have to repeat that variable at the end to return the resulting hash, since we are building the hash to use it somewhere, so in the end:

```
res = {
  some: 'value',
  id: id.presence,
}
res.compact!
res
```

vs

```
{
  some: 'value',
  **({ id: id } if id.present?),
}
```

While the first option looks like a bunch of instructions, the second option looks like a hash with plugged another hash – the intent is clearer.

The same with another example:

```
res = {
  some: 'value',
}
res.merge!(id: id, name: name) if id.present?
res
```

vs

```
{
  some: 'value',
  **({ id: id, name: name } if id.present?),
}
```

The second option looks more concise to me.

#11 - 08/25/2022 12:14 PM - sawa (Tsuyoshi Sawada)

LevLukomskyi (Lev Lukomskyi) wrote in #note-10:

> If use compact! as a different statement then you need to make a variable, you have to make a mental effort to make up the name for that variable and repeat its name to use compact!. [...] While the first option looks like a bunch of instructions, the second option looks like a hash with plugged another hash – the intent is clearer.

Okay, simply go back to my compact example. Indeed, as you said, that would create a temporal hash. However, please do not forget that your code uses a temporal hash as well, and it is even worse since you use a temporal hash per condition, possibly adding up to multiple temporal hashes. Given that fact, I do not think your concern about the performance using compact even makes sense.

#12 - 08/25/2022 01:17 PM - LevLukomskyi (Lev Lukomskyi)

I'm not that much concerned about the performance but about readability. Your solution with compact is probably good for that particular case but it cannot solve all other cases with an arbitrary hash plugged into the main hash on an arbitrary condition.

#13 - 08/25/2022 03:13 PM - austin (Austin Ziegler)

LevLukomskyi (Lev Lukomskyi) wrote in #note-10:

> ```
> res = {
>   some: 'value',
> }
> res.merge!(id: id, name: name) if id.present?
> res
> ```
>
> vs
>
> ```
> {
>   some: 'value',
> ```

```
  **({ id: id, name: name } if id.present?),
}
```

> The second option looks more concise to me.

I do not find the second option clearer or more concise in any way. It feels much more of line-noise.

In my code, I would opt for an explicit function that does the logic required *for* me, rather than trying to do it in-line, and it would probably be something like this:

```
def default_options
  {
    id: :omit,
    name: :omit
  }
end

def resolve_options(*overrides)
  default_options.merge(*overrides.compact!) { |k, default, value|
    value.nil? : default : value
  }.delete_if { |k, v| v == :omit }
end

# later, at the call-site
some_function(resolve_options(some: 'value', id: id, name: name))
# or
some_function(resolve_options({some: 'value'}, {id: id, name: name} if id.present?))
```

My nose twitches at complex inline if statements as you demonstrated.

It could be nice if **nil resulted in {}, but it has a high probability for compatibility issues, so I think caution is warranted.

#14 - 08/25/2022 10:32 PM - LevLukomskyi (Lev Lukomskyi)

@austin, your example is a perfect example of "overengineering".

We could argue about "clearness", although about "conciseness" it's easy to check – option 1: 72 characters, option 2: 62 characters, so option 2 is 15% more concise, clear winner.

> it has a high probability for compatibility issues

Currently, nobody is using **nil because it throws an error because nil.to_hash is not defined. So I don't see how adding such feature would lead to *"high probability of compatibility issues"*.

#15 - 08/26/2022 12:15 AM - austin (Austin Ziegler)

LevLukomskyi (Lev Lukomskyi) wrote in #note-14:

> @austin, your example is a perfect example of "overengineering".

On this, we disagree. I find the number of sigils required for **({ id: id, name: name } if id.present?) to be a sure sign that someone is trying to be clever, rather than correct.

> We could argue about "clearness", although about "conciseness" it's easy to check – option 1: 72 characters, option 2: 62 characters, so option 2 is 15% more concise, clear winner.

Except that it's not a clear winner at all. It means that I have to look at every single parameter to see the logic that applies to that parameter, which means that there's a *substantially* increased surface for bugs. I can write a *test* for resolve_options to make sure it always does the right thing. I can't do that for inline if id.present? cases. If you don't like the separate method resolve_options, then use ternaries:

```
some_function({
  some: 'value',
  id: id.present? ? id : :omit,
  name: id.present? ? name : :omit
}.delete_if { _2 == :omit })
```

Yes, I'm using == :omit instead of _2.nil? because nil may be a permitted value in the API being called, and I don't know that. But a resolve_options method on the class where some_function is defined could know that.

> it has a high probability for compatibility issues

> Currently, nobody is using **nil because it throws an error because nil.to_hash is not defined. So I don't see how adding such feature would lead to *"high probability of compatibility issues"*.

Daniel DeLorme points out some oddities that would be involved with a monkeypatch above: https://bugs.ruby-lang.org/issues/18959#note-3

Yes, this feature *could* be useful, but I also think that your example use cases are code that I would never permit because it's clever over clear.

#16 - 08/26/2022 02:52 PM - Dan0042 (Daniel DeLorme)

> > Currently, nobody is using **nil because it throws an error because nil.to_hash is not defined. So I don't see how adding such feature would lead to *"high probability of compatibility issues"*.
>
> Daniel DeLorme points out some oddities that would be involved with a monkeypatch above: https://bugs.ruby-lang.org/issues/18959#note-3

Yes, there are problems with adding nil.to_hash, but nil.to_h already exists, and I also fail to see how using that for splatting could have compatibility issues.

#17 - 08/26/2022 08:43 PM - LevLukomskyi (Lev Lukomskyi)

Currently:

```
"xyz".sub(/y/, nil) #=> TypeError: no implicit conversion of nil into String

def nil.to_hash; {} end
"xyz".sub(/y/, nil) #=> "xz"
```

Ok, the behavior is different, but it doesn't mean that we cannot implement the feature, it just means **that ruby should implement better nil handling in sub method**, and instead of raising TypeError it should raise ArgumentError if nil is passed, which is more expected, and also describe this in docs since currently, it says that the second parameter can be either string, hash, or block. If sub method is fixed this way then there will be no compatibility issues.

@austin your latest example has 110 symbols, the condition should be duplicated for each new hash key, as well as the ternary operator, so it's not good. Previous example has 294 symbols – it'll need more tests, because more code == more bugs and maintenance efforts.

#18 - 08/26/2022 09:16 PM - austin (Austin Ziegler)

LevLukomskyi (Lev Lukomskyi) wrote in #note-17:

> @austin your latest example has 110 symbols, the condition should be duplicated for each new hash key, as well as the ternary operator, so it's not good. Previous example has 294 symbols – it'll need more tests, because more code == more bugs and maintenance efforts.

This is — at its absolute most generous interpretation — a simplistic way of looking at code quality. There's a reason that we don't all code in APL, which by your measure, would have the least number of symbols.

Your examples increase the line-noise that detractors of Ruby point at without doing *anything* to increase the maintainability and readability of said code, and IMO do a lot to *decrease* the maintainability of said code because you can't just *see* the configuration, you have to read the entire line of *each* configuration line.

Given how tightly you're defending objectively unreadable code, I think that's more than enough reason to reject this feature, because there will be people who write code as unmaintainable as you've championed on this.

As I've said: there may be a good reason for such graceful handling—preferably by changing kw splatting from #to_hash to #to_h(as @dan0042) suggests—but I can't see anything you've written as example code as being a good argument in favour of it, because it's (1) wasteful, (2) noisy, and (3) *impossible* to write good tests for in application code.

Matz & the core team generally only approve things which have strong real-world use cases. Both @sawa and I have provided *better* options than your examples.

For your example, I think that a far better option would be to make it so that Hash#merge, Hash#merge! and Hash#update ignore nil parameters.

```
{some: 'value'}.update({ id: id, name: name } if id: present?)
```

But that would be an entirely *different* feature request. I still think it's bad code to put the logic in the parameter call, but it's still better than **({ id: id, name: name } if id.present?) which is *clever*, but ultimately unmaintainable for future you.

#19 - 08/26/2022 11:36 PM - LevLukomskyi (Lev Lukomskyi)

> because it's (1) wasteful

It wastes what? I don't understand this argument

> (2) noisy

This I can understand because indeed sometimes programmers can put too much in one line. However there are only two operators in that line – if and **, that's not that many, with IDE highlighting it reads easily.

> (3) *impossible* to write good tests for in application code.

This argument I also don't understand – what makes that piece of code *impossible* to write good tests? Eg.:

```
class Example
  def build_hash(id = nil, name = nil)
    {
      some: 'value',
      **({ id: id, name: name } if id.present?),
    }
  end
end

Rspec.describe Example do
  describe '#build_hash' do
    it 'returns simple hash if id is not set' do
      expect(described_class.build_hash).to eq(some: 'value')
    end

    it 'returns extended hash if id is set' do
      expect(described_class.build_hash(123, 'Lev')).to eq(some: 'value', id: 123, name: 'Lev')
    end
  end
end
```

I've just written simplified tests for application code, no problem.

> Both @sawa and I have provided better options than your examples.

It looks like you guys are ashamed of ** and try to avoid it at all costs, instead of grasping this new language feature.

I see that I'm not alone and there are other similar issues:

https://bugs.ruby-lang.org/issues/9291
https://bugs.ruby-lang.org/issues/8507
https://bugs.ruby-lang.org/issues/15381

and people @rits, @stephencelis, @kainosnoema, @ted, @Dan0042, @ioquatix (Samuel Williams) like this feature. I guess, there are more people frustrated by **nil behavior but they don't have time to write a feature request.

Also, I've found one more good example. Here a programmer has to write this:

```
def add_email_to_list(email:, tracking_info:)
  EmailSubscriber.find_by(email: email) || EmailSubscriber.new(
    email: email, **(tracking_info || {})
  )
  ...
end
```

instead of just writing this:

```
def add_email_to_list(email:, tracking_info:)
  EmailSubscriber.find_by(email: email) || EmailSubscriber.new(
    email: email, **tracking_info
  )
  ...
end
```

#20 - 08/27/2022 12:21 AM - austin (Austin Ziegler)

LevLukomskyi (Lev Lukomskyi) wrote in #note-19:

> > Both @sawa and I have provided better options than your examples.
>
> It looks like you guys are ashamed of ** and try to avoid it at all costs, instead of grasping this new language feature.

Not at all. I've used ** for a *very* long time (since kwargs were introduced). But I also tend to do **(options || {}) and I don't write hard-to-read code like **({id: id, name: name} if id.present?).

> Also, I've found one more good example. Here a programmer has to write this:
>
> ```
> def add_email_to_list(email:, tracking_info:)
>   EmailSubscriber.find_by(email: email) || EmailSubscriber.new(
>     email: email, **(tracking_info || {})
>   )
>   ...
> end
> ```
>
> instead of just writing this:
>
> ```
> def add_email_to_list(email:, tracking_info:)
>   EmailSubscriber.find_by(email: email) || EmailSubscriber.new(
>     email: email, **tracking_info
>   )
>   ...
> end
> ```

I'd write that as:

```
def add_email_to_list(email:, tracking_info: {})
  EmailSubscriber.find_by(email: email) || EmailSubscriber.new(email: email, **tracking_info)
end
```

It *is* a different signature (tracking_info is no longer a required kwarg) but it's a *better* API and doesn't require any special shenanigans.

I'm not arguing against the improvement of ** or even permitting **nil. But the example you've given is completely unconvincing because it is suboptimal, has unnecessary allocations, and is more reminiscent of what I'd see from Perl in 1998 than Ruby of 2022.

I think that there are two things that could help:

1. If we want to allow **nil, the argument should be made to switch ** from calling #to_hash to calling #to_h, because that would give you **nil *for free*. That would require digging a bit deeper to determine why ** calls #to_hash instead of #to_h.
2. Hash#merge and Hash#update could be modified to ignore nil arguments (which would permit your use-case).

#21 - 08/27/2022 03:16 AM - ioquatix (Samuel Williams)

It seems that foo(*arguments) will call arguments.to_a rather than arguments.to_ary. However, I also noticed foo(*Object.new) works as a single argument, which honestly seems a bit odd to me.

So:

> If we want to allow **nil, the argument should be made to switch ** from calling #to_hash to calling #to_h, because that would give you **nil for free. That would require digging a bit deeper to determine why ** calls #to_hash instead of #to_h.

seems consistent to me.

#22 - 08/27/2022 10:13 PM - LevLukomskyi (Lev Lukomskyi)

> I also tend to do **(options || {}) and I don't write hard-to-read code like **({id: id, name: name} if id.present?).

That's interesting, that you are ok for

```
**(options || {})
```

but not ok for

```
**(options if condition)
```

which is kind of the same..

> I'd write that as:
>
> ```
> def add_email_to_list(email:, tracking_info: {})
>   EmailSubscriber.find_by(email: email) || EmailSubscriber.new(email: email, **tracking_info)
> end
> ```

This does not solve the issue, the method still can be called with nil eg.:

```
add_email_to_list(email: 'email@example.com', tracking_info: nil)
```

and it'll raise the error, so you still need that boilerplate code **(tracking_info || {})

> the example you've given is completely unconvincing because it is suboptimal, has unnecessary allocations, and is more reminiscent of what I'd see from Perl in 1998 than Ruby of 2022.

that's true, it has an additional allocation if condition is satisfied, although it's not critical in most cases. Also, it could be probably optimized by interpreter in the future, similar to how [3, 5].max is optimized.

> 1. If we want to allow **nil, the argument should be made to switch ** from calling #to_hash to calling #to_h, because that would give you **nil *for free*. That would require digging a bit deeper to determine why ** calls #to_hash instead of #to_h.

Agree, to_hash method is weird, and it just adds confusion to people about why there are both to_hash and to_h, and which one they should use. I think to_hash should be deprecated and later removed.

BTW the issue with **nil can be also fixed by this code:

```
class NilClass; alias to_hash to_h end
```

> 1. Hash#merge and Hash#update could be modified to ignore nil arguments (which would permit your use-case).

Agree, doing nothing on .merge(nil) would be good. It will remove redundant allocations like

```
hash.merge(other_hash_param || {})
```

Although I guess it's a more radical change..

One note though, your example:

```
{ some: 'value' }.merge({ id: id, name: name } if id.present?)
```

won't work as you expect, it raises *SyntaxError: unexpected ')', expecting end-of-input*
The valid way would be:

```
{ some: 'value' }.update(({ id: id, name: name } if id.present?))
```

which looks not so nice..

#23 - 08/28/2022 05:15 AM - austin (Austin Ziegler)

LevLukomskyi (Lev Lukomskyi) wrote in #note-22:

> > I also tend to do **(options || {}) and I don't write hard-to-read code like **({id: id, name: name} if id.present?).
>
> That's interesting, that you are ok for
>
> ```
> **(options || {})
> ```
>
> but not ok for
>
> ```
> **(options if condition)
> ```
>
> which is kind of the same..

I disagree that it's the same, but more to the point, I mistyped: I tend **not** to do **(options || {}). I disagree that it's the same because putting an if expression *in a parameter position* is less immediately readable than || or a ternary.

> I'd write that as:
>
> ```
> def add_email_to_list(email:, tracking_info: {})
>   EmailSubscriber.find_by(email: email) || EmailSubscriber.new(email: email, **tracking_info)
> end
> ```

> This does not solve the issue, the method still can be called with nil eg.:
>
> ```
> add_email_to_list(email: 'email@example.com', tracking_info: nil)
> ```
>
> and it'll raise the error, so you still need that boilerplate code **(tracking_info || {})

No, you don't. You need to fail when someone passes you garbage (and nil is a garbage value in this case), or you need to ensure better default values (e.g., {}) at your call sites. I'm not sure that **nil converting to {} is sensible behaviour, given that—especially since ** uses implicit conversion (#to_hash) instead of explicit conversion (#to_h).

> the example you've given is completely unconvincing because it is suboptimal, has unnecessary allocations, and is more reminiscent of what I'd see from Perl in 1998 than Ruby of 2022.
> that's true, it has an additional allocation if condition is satisfied, although it's not critical in most cases. Also, it could be probably optimized by interpreter in the future, similar to how [3, 5].max is optimized.

I suspect that it can't be optimized easily, because arbitrarily complex expressions can be specified after if or unless.

1. If we want to allow **nil, the argument should be made to switch ** from calling #to_hash to calling #to_h, because that would give you **nil *for free*. That would require digging a bit deeper to determine why ** calls #to_hash instead of #to_h.

> Agree, to_hash method is weird, and it just adds confusion to people about why there are both to_hash and to_h, and which one they should use. I think to_hash should be deprecated and later removed.

It's present of the same reason that #to_ary and #to_a are present. #to_hash implies "this object *is* a hash", whereas #to_h says "this object can be converted to a hash".

> BTW the issue with **nil can be also fixed by this code:
>
> ```
> class NilClass; alias to_hash to_h end
> ```

That introduces other problems because of implicit hash coercion.

1. Hash#merge and Hash#update could be modified to ignore nil arguments (which would permit your use-case).

> Agree, doing nothing on .merge(nil) would be good. It will remove redundant allocations like
>
> ```
> hash.merge(other_hash_param || {})
> ```
>
> Although I guess it's a more radical change..

I think that it's *generally* a better change for the examples you have provided than **nil support.

> One note though, your example:
>
> ```
> { some: 'value' }.merge({ id: id, name: name } if id.present?)
> ```
>
> won't work as you expect, it raises *SyntaxError: unexpected ')', expecting end-of-input*
>
> The valid way would be:
>
> ```
> { some: 'value' }.update(({ id: id, name: name } if id.present?))
> ```
>
> which looks not so nice..

Fair, but I'll note that I was adapting your example. I would *never* write code where I use an if inline. If we extended the concept for #merge to ignore false or nil values, then we could write it as

```
{some: 'value'}.update(id.present? && {id: id, name: name})
```

I'm still not fond of the pattern, and think that an explicit call to Hash#delete_if to remove defaulted values is clearer and more intentional, but whatever.

I *don't* think that, in general, **nil is a good pattern to permit. There's an easy fix to it by using #to_h instead of #to_hash, if it turns out to be a good pattern. More than that, I think that we need a *better* use case to demonstrate its need than has been shown to date, because the patterns that have been presented so far are (IMO) unconvincing.

#24 - 08/28/2022 03:14 PM - p8 (Petrik de Heus)

LevLukomskyi (Lev Lukomskyi) wrote:

> I enjoy writing something like this in ruby on rails:
>
> ```
> content_tag :div, 'Content', class: [*('is-hero' if hero), *('is-search-page' if search_page)].presence
> ```

Starting with Rails 6.1 you can do:

```
content_tag(:div, 'Content', class: token_list('is-hero': hero, 'is-search-page': search_page))
```

#25 - 08/29/2022 12:40 AM - LevLukomskyi (Lev Lukomskyi)

> You need to fail when someone passes you garbage (and nil is a garbage value in this case), or you need to ensure better default values (e.g., {}) at your call sites.

When you pass nil you make less allocations, also you don't need to put || {} at all calling sites. For an options which are just passed somewhere else inside the method it's a valid usage.

> I suspect that it can't be optimized easily, because arbitrarily complex expressions can be specified after if or unless.

I think it doesn't matter how complex condition there is, because the interpreter must fully evaluate it first anyway to check whether if is satisfied, then if the expression inside if is {...} and ** operation is pending – just store entries in stack, not heap, and pass them to **. One more optimization could be to avoid creating an empty hash on **nil.

> It's present of the same reason that #to_ary and #to_a are present.

Actually yeah, I see now:

```
nil.to_a #=> []
nil.to_ary #=> NoMethodError: undefined method `to_ary' for nil:NilClass
nil.to_s #=> ""
nil.to_str #=> NoMethodError: undefined method `to_str' for nil:NilClass
nil.to_i #=> 0
nil.to_int #=> NoMethodError: undefined method `to_int' for nil:NilClass
nil.to_h #=> {}
nil.to_hash #=> NoMethodError: undefined method `to_hash' for nil:NilClass
```

so that's consistent. But anyway, you get used to good things quickly, I mean *nil somehow calls to_a and not to_ary, despite to_ary is designed for implicit coercions.

> I think that it's *generally* a better change for the examples you have provided than **nil support.

Actually that's not consistent that you are ok for merge(nil) but not ok for add_email_to_list(tracking_info: nil) which is kind of the same.

> I'm still not fond of the pattern, and think that an explicit call to Hash#delete_if to remove defaulted values is clearer and more intentional, but whatever.

I don't think that the way with delete_if is clearer since you mentally are doing two operations – first insert values into hash, and then remove them. In case of ** you are doing only one operation – insert values.

> Starting with Rails 6.1 you can do:
>
> ```
> content_tag(:div, 'Content', class: token_list('is-hero': hero, 'is-search-page': search_page))
> ```

Actually this generates Content if all values are false, so .presence is still needed But yeah, good to know that there is this way

#26 - 08/29/2022 02:17 PM - austin (Austin Ziegler)

LevLukomskyi (Lev Lukomskyi) wrote in #note-25:

> > You need to fail when someone passes you garbage (and nil is a garbage value in this case), or you need to ensure better default values (e.g., {}) at your call sites.
>
> When you pass nil you make less allocations, also you don't need to put || {} at all calling sites. For an options which are just passed somewhere else inside the method it's a valid usage.

You don't need to put || {} at all calling sites. The method and its documentation should be very clear that, if provided, the second parameter must be a Hash. That is, passing nil should be discouraged. Or you make the specific case that you want nil to *clear* the tracking info, in which case you probably shouldn't be using || {} or **kwargs at all.

There may be value in **nil => {}, but I repeat that the examples you have provided are unconvincing, but are instead *likely* to have unintended side effects, especially as the examples you've given seem to be AR-ish in nature, and converting a hash to kwargs is not likely what you actually want with Active Record.

> > I suspect that it can't be optimized easily, because arbitrarily complex expressions can be specified after if or unless.
>
> I think it doesn't matter how complex condition there is, because the interpreter must fully evaluate it first anyway to check whether if is satisfied, then if the expression inside if is {...} and ** operation is pending – just store entries in stack, not heap, and pass them to **. One more optimization could be to avoid creating an empty hash on **nil.

Such an optimization might be possible (or more likely detecting **nil and doing nothing).

> > I think that it's *generally* a better change for the examples you have provided than **nil support.
>
> Actually that's not consistent that you are ok for merge(nil) but not ok for add_email_to_list(tracking_info: nil) which is kind of the same.

Sure it is. The signature for #merge is #merge(*hashes, &block), so the pseudo-code implementation could easily be:

```
class Hash
  def merge(*hashes, &block)
    hashes.each do |hash|
      next if hash.nil?
      # … do merge magic and conflict resolution if block has been given here
    end
  end
end
```

Or, if you prefer

```
class Hash
  def merge(*hashes, &block)
    hashes.compact.each do |hash|
      # … do merge magic and conflict resolution if block has been given here
    end
  end
end
```

Both of these options permit the "clever" code you'd eventually come to regret, but do so cleanly.

> > I'm still not fond of the pattern, and think that an explicit call to Hash#delete_if to remove defaulted values is clearer and more intentional, but whatever.
>
> I don't think that the way with delete_if is clearer since you mentally are doing two operations – first insert values into hash, and then remove them. In case of ** you are doing only one operation – insert values.

You're looking at the concept of a #default_options method wrong. You're not inserting values in the hash, you're merging the *provided* values with defaults.

Elixir has a function Keyword.validate/2 that is called as Keyword.validate(options, default_options) and it ensures that the keyword list *only* has the keys present in default_options and that if those keys are missing, they're set with the provided default value. This is more or less the same for mutable objects, and one additional step: having a way of removing values that should be omitted if not provided. I've done very similar things in JavaScript as well. The pattern is simple: default_options.merge(*provided_options_list).delete_if { |h, k| omit_default?(h, k) }. The use of #merge is intentional, because you don't want to update the defaults (should they be stored in a constant), although

default_options.dup.update(*provided_option_list).delete_if {|h, k| omit_default?(h, k) } would work as well.