Rails has ActiveSupport::HashWithIndifferentAccess, which is widely used in Rails to handle Request, Session, ActionView's form construction, ActiveRecord's DB communication, and so on. It receives String or Symbol and normalize them to fetch the value. But it is implemented with Ruby. If we provide C implementation of that, Rails will gain the performance improvement.

summary of previous discussion: https://github.com/rails/rails/pull/40182#issuecomment-687607812

#1 - 12/26/2020 11:06 AM - matsuda (Akira Matsuda)

❤️

#2 - 12/26/2020 12:53 PM - Eregon (Benoit Daloze)

Isn't a C extension in a gem enough?

Also what specifically would writing it in C instead of Ruby gain?

Intuitively I'd think there would be no significant gain to write it in C.

Is there any profile showing a significant amount of time is spent in HashWithIndifferentAccess?

Also, the semantics of HashWithIndifferentAccess IMHO don't fit the collection design of the core library. And the name doesn't really fit with other class names in the core library.

#3 - 12/27/2020 10:44 AM - mame (Yusuke Endoh)

+1, if Rails people really want it, and if it brings performance improvement. We need to experiment, but in principle, it looks to me a good idea to provide small but simple improvements that we can use immediately in Ruby 3.1. We mainly focused on big language improvements until 3.0, but many of them still require some time to be practically useful.

#4 - 12/27/2020 04:14 PM - Eregon (Benoit Daloze)

I don't think C code will be more efficient for things like https://github.com/rails/rails/blob/914caca2d31bd753f47f9168f2a375921d9e91cc/activesupport/lib/active_support/hash_with_indifferent_access.rb#L367.

And translating the rest of the logic to C would just make it harder to read, maintain, and not gain anything.

Are we going to rewrite Rails in C too? ;)

Honestly, I think it's a waste of time to even experiment, it's bound to insignicicant gains for large efforts and maintenance cost.

IMHO, HashWithIndifferentAccess in core would be a mistake, as plain as a mistake can be.

Also, I see no valid reason to have this in core, if people really want to try this, they can make a C extension.

#5 - 12/27/2020 05:13 PM - jeremyevans0 (Jeremy Evans)

I am against adding this in principle. One of the harder things for new Ruby programmers to understand is the difference between symbols and strings. This is even more difficult for programmers learning Ruby and Rails at the same time, due to the fact that Rails treats symbols and strings the same in most places.

I'm also against adding this for the reasons that Eregon (Benoit Daloze) mentioned. If a speed-up is desired, HashWithIndifferentAccess can easily be a C-extension gem, there is no pressing reason to have it in core or stdlib.

If we do decide to add this to Ruby, it should have a better name. HashWithIndifferentAccess is not indifferent in regards to type (1 and "1" are different). HashWithIndifferentAccess is not indifferent in regards to case ("a" and "A" are different). HashWithIndifferentAccess is not indifferent in regards to encoding, at least some of the time ("u1234" and "u1234".b are different). "WithDifferentAccess" is too vague. HashCovertingSymbolKeysToStringKeys is more accurate, though quite long.

#6 - 12/27/2020 07:09 PM - Dan0042 (Daniel DeLorme)
I'm also against this, but I think a more general-purpose version might be ok. In my code I use a NormalizedHash class which calls key = normalize(key) for every method with a key argument (as well as merge, etc.) I use this base class to define subclasses with specific normalize methods; HeaderHash converts keys to Http-Header-Camelcase, OptionsHash raises an error if the key is not a Symbol. Something that can be used as the basis of different implementations (including HashWithIndifferentAccess) might make sense in the stdlib. Might.

#7 - 12/27/2020 08:08 PM - marcandre (Marc-Andre Lafortune)
Count me in the "No" camp.

HashWithIndifferentAccess has horrible semantics. It may have had a reason to be when Ruby didn't support symbol garbage collection and Rails didn't require an explicit mapping of HTTP Request to params, but not today.

#8 - 12/28/2020 03:14 AM - naruse (Yui NARUSE)
- Description updated

#9 - 12/28/2020 03:41 AM - naruse (Yui NARUSE)
My intention is

- A feature to implement Rails's ActiveSupport::HashWithIndifferentAccess, not providing HashWithIndifferentAccess itself.
- the key of internal hash should be symbol though ActiveSupport::HashWithIndifferentAccess uses String as key.
- rails doesn't depend C extension.

I added a link to kamipo's summary as more context
https://github.com/rails/rails/pull/40182#issuecomment-687607812

Dan0042 (Daniel DeLorme) wrote in #note-6:

I'm also against this, but I think a more general-purpose version might be ok. In my code I use a NormalizedHash class which calls key = normalize(key) for every method with a key argument (as well as merge, etc.) I use this base class to define subclasses with specific normalize methods; HeaderHash converts keys to Http-Header-Camelcase, OptionsHash raises an error if the key is not a Symbol. Something that can be used as the basis of different implementations (including HashWithIndifferentAccess) might make sense in the stdlib. Might.

You can use Hash#compare_by_identity (https://ruby-doc.org/core-2.7.1/Hash.html#method-i-compare_by_identity).

#10 - 12/28/2020 11:22 AM - byroot (Jean Boussier)
A feature to implement Rails's ActiveSupport::HashWithIndifferentAccess

Would it be possible to have a "hook" akin to convert_key?

e.g. something like:

```ruby
hash = {}
hash.coerce_key = ->(key) { key.is_a?(Symbol) ? key.name : key }
```

#11 - 12/28/2020 11:24 AM - naruse (Yui NARUSE)
byroot (Jean Boussier) wrote in #note-10:

A feature to implement Rails's ActiveSupport::HashWithIndifferentAccess

Would it be possible to have a "hook" akin to convert_key?

I think it will not fast.

#12 - 12/28/2020 11:53 AM - Eregon (Benoit Daloze)
naruse (Yui NARUSE) wrote in #note-11:

I think it will not fast.

Before we start considering performance trade-offs, do we even have a benchmark where time spent in HashWithIndifferentAccess is significant for a Rails app?

#13 - 12/29/2020 08:34 PM - zverok (Victor Shepelev)
I believe that HashWithIndifferentAccess is one of the very false ideas in Rails -- and that as of today, it is a more or less common understanding in the community.

The distinction of Symbol (as a controlled internal name) and String (as an input/output user data) is one of the very powerful concepts in Ruby, and the "I don't want to think whether it is Symbol or String, internal or external" is explicitly against this distinction (as far as I can understand, the holders of this ideas would actually prefer to have ONLY strings, but have a shorter syntax for string keys, like in JS, and that's how they perceive symbols).

Eventually, even Rails started to make a clearer distinction between internal/external (see StrongParams), but HWIA is so omnipresent there, that they doubtfully get rid of it anytime soon. But I don't believe it is a reason to introduce it in a core language (not even mentioning the fact that the reason "Rails uses it, and it should be implemented in C, let's add it to the language core" feels quite weird).

I am really surprised that Ruby core developers feel so positive about it.

### #14 - 01/02/2021 10:53 AM - nobu (Nobuyoshi Nakada)

I had a vague original idea for this proposal, which extends the Hash class generically.

First I though about case-insensitive string hashes, it had been able by using $= in old days. The special variable was removed, still there are that use cases, e.g., HTTP headers, command completions, etc.

As I glanced st.c again this time, confirmed that customizing key conversion per instances isn't possible as far as keeping the backward compatibility. I think case-insensitive (only for String) Hash, like hashes compared by identity, would be possible, though.

### #15 - 01/17/2021 12:57 PM - hcatlin (Hampton Catlin)

During my 15 years of Ruby programming, I can't remember a single time that the difference between string-and-symbol with regards to Hashes was used on purpose. Instead, it's the source of countless bugs, extra type checking code, and difficulty when you install a new library-- ("hmmm, is this options hash going to be stringed or symboled?")

Would anyone here think the code below was acceptable?

```ruby
options[:host] = "ruby-lang.org"
options[:port] = 3000
```

Not only would we likely flag that code, but chances are that it might not work! And this is why Rails has HashWithIndifferentAccess and most of us would prefer to use it in our libraries, and just don't want to have the awkwardness of having to type out that horribly long name every time.

In almost every use of Hash in my career, the keys have being either symbols or strings, and I can't think of production code that even uses the fact that you can use an object as a key as a feature. I think I've attempted it myself a couple times, but usually refactored the code after, berating myself for trying to be a little too clever.

Which brings me to #11982!

I opened this proposal 5 years ago that would basically mean that Rails wouldn't need HashWithIndifferentAccess, as Ruby would have it's own stringed, symbol-or-string agnostic Map implementation.

In the last year I've come back to full time Ruby programming, and this remains one of my biggest frustrations with the language. Hash was cute when I first learned the language and it was an early differentiator, but other languages have improved this paradigm and I think we should too!

A first-level Ruby syntax for easily creating Hash-like objects that you can use as a Dictionary/Map/etc without worrying about if it's a string or a symbol is my 2021 wish! :)

### #16 - 01/23/2021 11:54 AM - timcraft (Tim Craft)

hcatlin (Hampton Catlin) wrote in #note-15:

> Would anyone here think the code below was acceptable?

Mixing symbol keys and string keys together like that would no doubt be very confusing, but why would you choose to do that instead of just using symbol keys or string keys? With modern Ruby the use case handled by options hashes is typically better handled by keyword arguments, and sometimes an options/config object might be a more appropriate choice. Keyword argument hashes use symbol keys, and I would expect options/config objects to be implemented using symbol keys.

We now also have Hash#transform_keys, so transforming a hash with string keys from outside data sources to symbol keys is straightforward. That can easily handle use cases like transforming keys in config/database.yml to symbols. JSON has symbolize_names and so on. There doesn't need to be confusion if library authors make a clear decision on whether to use strings or symbols for a specific use case, instead of trying to support both.

And this is why Rails has HashWithIndifferentAccess and most of us would prefer to use it in our libraries, and just don't want to have the awkwardness of having to type out that horribly long name every time.

Maybe I'm in the minority, but I would never use HashWithIndifferentAccess in my own libraries, and I try hard not to use it directly in Rails because I've seen it become the source of confusion and subtle bugs!

I think the critical use case for HashWithIndifferentAccess is params; where you want to be using symbol keys because it's cleaner syntax, but the
keys are coming from untrusted input. But HashWithIndifferentAccess is just the historical solution that exists for that case. There's no reason there
couldn't be an ActiveSupport::Params class which allowed for params.title instead of params[:title], making the choice between whether to use string
keys or symbol keys disappear (in the calling code at least).

In almost every use of Hash in my career, the keys have being either symbols or strings, and I can't think of production code that even uses the
fact that you can use an object as a key as a feature. I think I've attempted it myself a couple times, but usually refactored the code after,
berating myself for trying to be a little too clever.

I have a lot of code which does, plenty of it in production. Hashes that represent a set of attributes, hashes that represent a set of variables (e.g. for
template locals), and keyword argument hashes are relatively frequent examples for using symbol keys. The syntax for hash literals with symbol keys
is so much nicer to read and write than the hashrocket syntax with string keys.

Apart from symbols the other common use case I encounter is time series data. I frequently use date keys, integer keys for years, month keys, and
quarter keys. Having this as an option instead of only being able to use strings is one of the reasons I prefer Ruby to other languages. And I don't
consider that kind of code to be clever. Quite the opposite, it feels like "boring" code which just works.

In the last year I've come back to full time Ruby programming, and this remains one of my biggest frustrations with the language.

Welcome back to Ruby! :) Are you working on Rails apps? If so it seems a little unfair to get frustrated at the language for choices made by the web
framework. If not can you share an example of a Ruby use case that would be significantly improved by HashWithIndifferentAccess?

FWIW I'm against this proposal for the reasons already mentioned:

- The name is terrible, which apart from making it clumsy to use and not seeming very Ruby-ish I think demonstrates that the semantics of the
class are a bit unclear. As Jeremy points out it is not indifferent in many respects, just with symbols and strings.

- There isn't any profiling data which shows this is a significant bottleneck for a significant number of Rails projects. Optimizing performance for
  Date, Time, BigDecimal, JSON, URI, YAML etc would benefit Rails and a broader range of Ruby applications.

- Re-implementing in C adds more work for maintainers of alternative Ruby implementations, for seemingly little benefit.

The original motivation here seems to be for improving performance of Rails applications. Why does it need to be in Ruby core/stdlib? Why can't it
just be implemented as a C extension packaged in a gem?

#17 - 03/27/2021 12:31 AM - joelb (Joel Blum)

I think the critical use case for HashWithIndifferentAccess is params; where you want to be using symbol keys because it's cleaner syntax, but
the keys are coming from untrusted input

We can agree the vast majority of Ruby devs are web programmers, mostly with Rails. So they also write a lot of javascript and obviously they prefer
the js syntax [name: 'joe'] to rocket ['name' => 'joe'], so they mostly use symbols under the hood (emphasized because the developer 99% of the
time doesn't care whether it's a symbol or a string during normal, routine web/Rails work. The developer simply wants a dictionary with string like keys
and prefers to use the js object notation).

So if 99% of the time you don't care whether it's a symbol or a string, yet the popular hash syntax goes for symbol keys, what happens is every time
you do JSON.parse you will get stringified keys and it's very easy to see why from a user point a view some indifferent construct makes sense. The
truth is most of us ARE indifferent, we just want a dictionary with string like names.

#18 - 03/27/2021 01:03 AM - marcandre (Marc-Andre Lafortune)

joelb (Joel Blum) wrote in #note-17:

[...] what happens is every time you do JSON.parse you will get stringified keys

Use JSON.parse(data, symbolize_names: true)

#19 - 03/27/2021 06:49 AM - joelb (Joel Blum)

Use JSON.parse(data, symbolize_names: true)

I know that. Yet the fact is these bugs happen again and again (not only to new Ruby devs, would you agree it's quite easy to forget to
symbolize_keys or stringify or what have you). I don't know if this suggestion is the right solution for the problem but I was hoping we could at least
agree there's a problem for a sizeable segment of Ruby users.
A small thought: I think the original sin in Ruby was making name in `{name: 'joe'}` a symbol, e.g. have the new "javascript syntax" use symbol keys. I think it would have been better if it was a string / frozen string instead. And if a developer really needed symbol keys (which us Rails devs actually never do), he could have perhaps written `{name => 'joe'}` or whatever. And yes it means we would have had to access the hash so: `hsh['name']` and not `hsh[name]` (I kinda like the latter for saving a character) but we wouldn't have been having this discussion over and over again. By default hash keys would have been strings and that's that (correct me if I'm wrong but that's how it is in most programming languages; I don't see js or python devs insisting on symbol hash keys).

I don't know if anyone would agree with me here, and I know it's too late to do that anyway, but maybe it's worth mentioning.

#20 - 03/28/2021 01:56 AM - duerst (Martin Dürst)

joelb (Joel Blum) wrote in #note-19:

Use JSON.parse(data, symbolize_names: true)

I know that. Yet the fact is these bugs happen again and again (not only to new Ruby devs, would you agree it's quite easy to forget to symbolize_keys or stringify or what have you). I don't know if this suggestion is the right solution for the problem but I was hoping we could at least agree there's a problem for a sizeable segment of Ruby users.

Maybe we should change the default on JSON.parse? That would probably lead to too much backwards compatibility issues.

Maybe we should introduce a new method where symbol keys are the default. That could be done without backwards compatibility issues, just by spreading the word to use the new method.

Another option may be a global setting that projects could use to change the default.

A small thought: I think the original sin in Ruby was making name in `{name: 'joe'}` a symbol, e.g. have the new "javascript syntax" use symbol keys.

No. Using a symbol is the right thing to do in Ruby. Having :name be a symbol, but name: be a string would be highly confusing. And keys (i.e. member names) in data structures usually are identifers rather than data, so in Ruby, symbols are more appropriate.

If you really want an "original sin", it's that Ruby distinguishes between identifiers (symbols) and strings (see also below).

I think it would have been better if it was a string / frozen string instead. And if a developer really needed symbol keys (which us Rails devs actually never do), he could have perhaps written `{name => 'joe'}` or whatever. And yes it means we would have had to access the hash so: `hsh['name']` and not `hsh[name]` (I kinda like the latter for saving a character) but we wouldn't have been having this discussion over and over again. By default hash keys would have been strings and that's that (correct me if I'm wrong but that's how it is in most programming languages; I don't see js or python devs insisting on symbol hash keys).

Javascript doesn't have symbols in the first place, so I don't see how JS devs could insist on symbol hash keys anyway. I also haven't found symbols in Python, so I guess the same thing applies there, too. When searching for "Python symbol", the only stuff I get is sympy. To see whether it's more natural to treat JSON keys as symbols or as strings, you would have to look at other languages that distinguish symbols and strings, e.g. Lisp.

I don't know if anyone would agree with me here, and I know it's too late to do that anyway, but maybe it's worth mentioning.

Different programming languages just handle different things differently, not only in syntax but also in semantics. If you work with Javascript and Ruby, you have to deal with the fact that in conditions, the empty string and 0 are treated differently. There are other issues that you have to be aware of. Unfortunately, no way of getting around this.

#21 - 03/28/2021 03:17 AM - marcandre (Marc-Andre Lafortune)

duerst (Martin Dürst) wrote in #note-20:


String are meant for input/output of text. Symbols are identifier for developers. A program written in English for French users would have its symbols in English and its strings in French...

In any case, that some people disagree with the validity of the distinction of Symbols and Strings, it is not going away. There's not much point discussing that.

#22 - 03/28/2021 05:29 PM - joelb (Joel Blum)

I find it very unlikely any language will make symbols the default hash keys like Ruby did.

String are meant for input/output of text. Symbols are identifier for developers.

Why do you need that distinction?

#23 - 03/31/2021 12:19 AM - duerst (Martin Dürst)
marcandre (Marc-Andre Lafortune) wrote in #note-21:

duerst (Martin Dürst) wrote in #note-20:

Javascript doesn't have symbols in the first place


Thanks for the pointer. I haven't fully understood that page, but it seems to me that in JS, the 'uniqueness' aspect is much more important, and there are less similarities with strings than in Ruby. Also, the interface on JS symbols is much smaller than in Ruby, and quite inconvenient except for some very narrow use cases.

joelb (Joel Blum) wrote in #note-22:


I find it very unlikely any language will make symbols the default hash keys like Ruby did.

Definitely not JS, see above. But what about various Lisp variants?

String are meant for input/output of text. Symbols are identifier for developers.

Why do you need that distinction?

You don't need that distinction. There is also no need to distinguish integers and floats. Most programming languages make that distinction, but some, including JS, don't (or didn't?).

There is also no strict need to distinguish between strings and numbers. Almost all programming languages have such a distinction, but in Perl (+most shells, awk,...), a string that looks like a number is just a number, and any number is a string. Some languages (e.g. Python, Haskell) provide arrays and tuples, others (e.g. Ruby, JS last time I checked) have only arrays. Some languages distinguish between characters and strings (Ruby up to version 1.8, C), others don't (Ruby from version 1.9,...).

Each choice has its reasons, advantages, and disadvantages. If all programming languages were the same, it would be very boring.

#24 - 04/01/2021 09:49 PM - lamont (Lamont Granquist)
+1 on adding this.

There are any number of bugs which are caused by reading mixed symbols and strings into Hashes, then round tripping them through JSON and having them change one way or the other.

I mildly disliked the use of Mash structures in Chef for about the first 5 years I used them, thinking they were too clever, until I came around to this realization that it let you export a Hash-like structure as an API which consumers could then use and not have to worry about injecting some JSON that had symbolized keys on it and getting into deep symbols-vs-strings confusion. That whole issue goes away, which is very novice-friendly. What is left is that mixed syntax when writing code is considered to be ugly, but that can generally be fixed byrubocop automatic linting rules and community standards.

And if you don't like that, its cool, you don't have to use it. I would love to see it as a core language feature though. Although I really need to be able to inherit from the class and wrap convert_key/convert_value and it would be useful to have a private regular_writer/regular_update bypass (for performance) that subclasses could use for already-converted access.

#25 - 04/02/2021 04:11 AM - sawa (Tsuyoshi Sawada)
I am surprised and am sad that so many of the core developers are positive on this.

#26 - 04/02/2021 01:45 PM - joelb (Joel Blum)
sawa (Tsuyoshi Sawada) wrote in #note-25:
I am surprised and am sad that so many of the core developers are positive on this.

Why? I understand you passionately disagree with this. But it could be the people having a problem with hash symbol vs strings aren't explaining themselves well enough. Is there a point to try to convince you / clarify the pain more? I think we are not being needlessly stubborn, we really do run into symbol/string hash key bugs enough to make us want to use HashWithIndifferentAccess. Even if I run into these issues only once in 6 months (and I'm experienced enough, for Ruby newcomers this will happen more frequently) it's enough for me to turn into HashWithIndifferentAccess. I simply see no value in thinking about the type of named keys of my hashes, I want them to have names and whether they are strings/symbols/frozen strings doesn't really matter to most users I suspect.

I think language designers should generally care about the average user and average use case more than anything else.

#27 - 04/02/2021 02:13 PM - Dan0042 (Daniel DeLorme)

lamont (Lamont Granquist) wrote in #note-24:

Although I really need to be able to inherit from the class and wrap convert_key/convert_value and it would be useful to have a private regular_writer/regular_update bypass (for performance) that subclasses could use for already-converted access.

I don't think the current proposal would allow that. Although it's not super clear what the proposal is, specifically. It's "something" to make HashWithIndifferentAccess more performant via C implementation, but that "something" is quite vague.

joelb (Joel Blum) wrote in #note-26:

I think we are not being needlessly stubborn, we really do run into symbol/string hash key bugs enough to make us want to use HashWithIndifferentAccess.

Then by all means use it! What's the problem with HashWithIndifferentAccess if that's what you need?

Several people have expressed their doubt that a C implementation would provide any significant speedup, and I agree. But as mame said, "we need to experiment" to see if that's the case or not. But before that we need a concrete proposal. Is it to add Hash#compare_by_symbol=?

#28 - 04/02/2021 07:34 PM - lamont (Lamont Granquist)

Any implementation should probably consider recursive structures under arrays and all the mutator methods on Array:

```ruby
% pry
[1] pry(main)> require 'active_support/hash_with_indifferent_access'
=> true
[2] pry(main)> test = ActiveSupport::HashWithIndifferentAccess.new
=> {}
[3] pry(main)> test['foo'] = []
=> []
[4] pry(main)> test['foo'] << { bar: "baz" }
=> [{:bar=>"baz"]
[5] pry(main)> test.class
=> ActiveSupport::HashWithIndifferentAccess
[6] pry(main)> test['foo'][0].class
=> Hash
[7] pry(main)> test['foo'] = [{:bar=>"baz"]
NoMethodError: undefined method `nested_under_indifferent_access' for {:bar=>"baz"]:Hash
from /Users/lamont/.asdf/installs/ruby/3.0.0/lib/ruby/gems/3.0.0/gems/activesupport-6.1.3.1/lib/active_support/hash_with_indifferent_access.rb:381:in `convert_value'
```
[8] pry(main)>

#29 - 04/02/2021 07:42 PM - lamont (Lamont Granquist)

And people should likely recall the reasons for the existence of symbols and strings. Originally strings were very expensive and not deduplicated, but were garbage collected. While symbols were very cheap, deduplicated and frozen and not garbage collected. Over time they've mutated to the point where frozen string literals and symbols look pretty much identical. Given a time machine, they probably should be literally identical such that :foo.equal?("foo".freeze) would be true which would just render symbols syntactic sugar for string literals. I believe some languages don't have any symbol-like datastructures and instead have frozen strings so trying to extend the JSON standard to have symbols is probably the wrong way around to deal with this problem (those languages would not be made better by introducing symbol-like objects, nobody needs them).

That is all way too much to do to the language at this point, though, since the backcompat break would be extensive, but features like this would be good to better opt-in to closer to that kind of behavior.