Should we get rid of the enumerate trap and Reflect.enumerate? #1

Edit New issue

Open **erights** opened this issue on Jan 29 · 31 comments

erights commented on Jan 29

TC39 discussion at https://public.etherpad-mozilla.org/p/tc39-jan-28 recorded resolution that we need to ask Tom **@tvcutsem** before deciding. Based on what we could remember in the room without Tom's input, we all agreed that we would like to get rid of these. Tom's feedback:

> A few things come to mind:
>
> 1) Faithful virtualization of the prototype chain. We decided that the Proxy should always be in control of operations involving prototype inheritance. This includes has, get, set and enumerate. For all of these operations, when executed on a normal object, they will walk the prototype chain until they hit a Proxy. After that point, full control is delegated to the Proxy. If a Proxy would only be able to override get() and set() but not enumerate(), this may lead to inconsistencies (without an enumerate() trap, the VM will have to externally walk the prototype chain of the Proxy via getPrototypeOf()).
>
> 2) When we designed Proxies originally, we made the design decision to map many built-in operations directly to specific traps, even if these operations could be expressed in terms of more fundamental operations (get, set, has, enumerate are all examples here). The design argument here was always improved performance by reducing the number of temporary allocations.
>
> Much later in the design, others started pushing back on the number of traps, leading us to reconsider some traps, e.g. I recall the "hasOwn" trap was dropped in favor of checking whether (getOwnProperty() !== undefined), even though it is strictly less efficient. I don't have a view on the actual cost of expressing enumerate() in terms of lower-level MOP operations, but it seems obvious that there will be more allocation costs involved. The question is then: if enumerate() gets dropped on these grounds, why not get(), set() and has()?
>
> 3) As far as I recall, enumerate() is the only trap that actually returns an *iterator* over the property names, allowing efficient virtualization of objects with a large (or potentially, infinite) number of properties. The only other trap that returns a list of property keys, "ownKeys", returns a manifest array of property names.
>
> 4) enumerate() had weaker invariants than ownKeys() [just like all the traps that deal with prototype inheritance, since a frozen object can still inherit from a non-frozen object, leading to weaker observable invariants in general]. If one would express for-in in terms of repeated calls to ownKeys(), the invariant check overhead may be substantial compared to just calling enumerate().
>
> Unless there are really good reasons to get rid of enumerate(), I think it should remain in. For me, (1) and (3) are the killer arguments: consistency with has(),get(),set() and efficient enumeration via iteration.
>
> Feel free to forward this mail to TC39. I'm happy to engage in follow-up discussions, just ping me when needed.

Labels
None yet

Milestone
No milestone

Assignee
No one—assign yourself

Notifications
Unsubscribe
You're receiving notifications because you were mentioned.

7 participants

Lock conversation

erights commented on Jan 29

To facilitate discussion of this issue (and likely many other issues) I would like us to invite Tom to join the delegates group.

bterlson commented on Jan 29

I have given **@tvcutsem** the appropriate permissions.

bterlson commented on Jan 29

It is early and I have not had my coffee yet. Sorry for any errors below :)

I don't believe efficiency is particularly compelling. For-in on a proxy may be slow.

It is true that enumerate is more efficient and at least partially useful with a huge number or infinite number of property keys. But ownKeys won't work with these sorts of objects. And enumerate only considers enumerable properties. It doesn't seem too compelling to hold on to enumerate so that for-in can iterate infinitely. Maybe I'm missing something.

The reason to get rid of enumerate is that we don't know how to fix tc39/ecma262#161. The options are 1) for-in or [[enumerate]] exhausts the iterator before entry into the loop, potentially doing has checks to not visit deleted keys (and therefore removing any benefit of [[enumerate]] being an iterator), or 2) leave spec as-is and require implementations to not pre-compute the list of keys as they do today as pre-computation is now observable.

Point 1 seems most compelling to me. Not sure what problems would look like in practice so it's hard for me to theorize how fatal it is.

rossberg-chromium commented on Jan 29

@bterlson, agreed on all your points. Point 1 is a valid concern, although it probably is more philosophical than practical. Not sure if JavaScript is the language to seek perfect philosophical purity. :)

bterlson commented on Jan 29

Also worth pointing out that going with option 1 above means specifying some semantics for reconfiguring enumerable keys during iteration which presently is implementation defined (and differs across implementations).

https://gist.github.com/bterlson/893619d0766008e93885

erights commented on Jan 29

I find point 1 important and not to be dismissed lightly. I wish I had remembered it during the meeting. Point 3 should no longer concern us, now that we've decided that the for/in client always starts by doing a full snapshot anyway. In any case, I'll defer to others on efficiency issues; my concern is semantics. Point 1 is an important semantic property. Perhaps there is a lighter weight way to accommodate it?

That said, if we drop the iterator trap, it will not impede any of the concrete plans I have for using proxies, nor any concrete plans I have heard anyone else suggest. So I am willing to drop it.

erights commented on Jan 29

Regarding the choice between Brian's options 1 and 2, I strongly prefer 1, so that we can all agree on snapshot + has checks. This takes us vastly closer to a deterministic for/in spec. Thus, there is certainly no reason for the enumerate trap to be an iterator.

rossberg-chromium commented on Jan 29

Yes, AFAICT, everybody in the room yesterday agreed that option 1 would be preferable in almost every way.

bterlson commented on Jan 29

@rossberg-chromium true, iff:

1. we can't remove [[Enumerate]] trap, and
2. we are ok standardizing that keys reconfigured to be non-enumerable during for-in enumeration are still visited (both for proxies and for regular objects).

I'm not sure I'm ok with 2, yet. And I'm not sure we had consensus on this. Am I remembering wrong?

erights commented on Jan 29

What I think I remember regarding **@bterlson** 's question 2: I remember that when we tried to declare consensus, you (**@bterlson**) said you had to check first what the implementations issues were with Chakra. But if there was no cause for concern there, I think you provisionally agreed that this would be ok.

I do not remember anyone else expressing reservations. But we did not try to officially declare conditional consensus (as we did with the enumerate trap itself), so not everyone with reservations may have felt the need to voice them.

Does this sound right?

bterlson commented on Jan 29

That sounds right **@erights**. Also I have discovered (see above) that Chakra semantics with re-configured non-enumerable keys are handled same as delete (ie. will not be visted). IMO this seems like better semantics to me. I will need to investigate more to see if this is something we could get away with changing.

erights commented on Jan 29

I agree the semantics are better. If we can all agree on snapshot + has and enumerable check, so much the better.

But I am much more interested in all of us agreeing on the same behavior than I am at improving this kind of semantic issue. +1 to whatever more deterministic behavior we can all agree on. Ideal would be to agree to a deterministic behavior.

allenwb commented on Jan 29

I think the place to virtualize the prototype chain (or to implement unique inheritance semantics) is in [[GetPrototypeOf]], [[OwnPropertyKeys]], [[GetOwnProperty]], etc. Having to have [[Enumerate]] separately implement it is a bug farm. For example, the part of [ordinary object [Enumerate] that says

> The enumerable property names of prototype objects must be obtained as if by invoking the prototype object's [[Enumerate]] internal method. [[Enumerate]] must obtain the own property keys of the target object as if by calling its [[OwnPropertyKeys]] internal method. Property attributes of the target object must be obtained as if by calling its [[GetOwnProperty]] internal method.

was added very late in response to a bug report.

I think it is just fine for-in to define a standard property enumeration in terms of [[GetOwnProperty]], [[OwnPropertyKeys]], [[GetOwnProperty]]. If a virtual object really wants to synthesize some other view of inherited enumerable properties it should do it using those fundamental operations.

erights commented on Jan 29

> I think it is just fine for-in to define a standard property enumeration in terms of [[GetOwnProperty]], [[OwnPropertyKeys]], [[GetOwnProperty]]

and, as you mentioned earlier, [[GetPrototypeOf]].

I do not disagree with the point of your note.

allenwb commented on Jan 29

@bterlson
WRT reconfiguration 9.1.11 says:

> The values of [[Enumerable]] attributes are not considered when determining if a property of a prototype object has already been processed.

and

> The enumerable property names of prototype objects must be obtained as if by invoking the prototype object's [[Enumerate]] internal method.

and (at least for ordinary objects) [[Enumerate]] only returns enumerable, non-symbol property names.

So what do you think is under-specified?

bterlson commented on Jan 29

@erights continuing our chain of agreement followed by qualification, I agree we should decide on a deterministic behavior but I would much rather do that as part of a separate process. I would like to include whatever we decide here in ES2016 if at all possible and making a quick decision on a stricter enumeration semantics seems ill-advised.

@allenwb see the gist I posted above: https://gist.github.com/bterlson/893619d0766008e93885. I believe 9.1.11 says nothing about whether `c` should be visited or not.

erights commented on Jan 29

@bterlson Agreed

rossberg-chromium commented on Jan 29

In principle, I agree that excluding properties that turned non-enumerable would be the more logical semantics. However, I also suspect that it is going to cost, since it requires another check on each iteration. Given that this is a corner case with no practical relevance, I'd much rather not impose that extra cost on every user of for-in.

erights referenced this issue on Jan 29

Deterministic for/in enumeration order and sequence of proxy traps #2 **Open**

allenwb commented on Jan 29

Ah, well the informative generator definition provided would not produce 'c' because the yield of each name is guarded by an enumerable check.

You're right that the the normative text is not explicit about that case. However, I think it is reasonable to read the normative text

> A property that is deleted before it is processed by the iterator's next method is ignored. If new

> properties are added to the target object during enumeration, the newly added properties are not guaranteed to be processed in the active enumeration.

as appling to properties that have had their enumerable changed in addition to added/deleted properties. Approximately that same language appeared in both ES3 and ES5 as part of the normative semantics of for-in. During ES5 when we originally specified Object.defineProperty semantics our conceptual model of changing the attributes of an existing property was that it was equivalent to atomically deleting the old property and adding back the same property with revised attribute values.

ljharb commented on Jan 29

Regarding an enumerable check, please note that as much as I love that check and think it should be included, `var obj = { get a() { Object.defineProperty(this, 'b', { enumerable: false }); }, b: 42 }; for (var k in obj) { console.log(k, obj[k]); }` does indeed log "b, 42" in Safari 9, Chrome Canary, Firefox Nightly, Edge, and IE 11 (and presumably all sorts of others).

tvcutsem commented on Jan 29

It took me a while to ingest all the various debates that lead to this issue, but I think I now have a better understanding of the constraints.

I agree with the counter-arguments to (2), (3) and (4). I was unaware that the iterator must be exhausted anyway for other reasons.

That leaves point (1). To make it easier to see if there would be potential issues, let's consider a concrete use case of virtualizing the proto-chain: implementing an object that pretends to support multiple inheritance or mixin-style inheritance. Obviously, getPrototypeOf will betray the illusion as it can return only 1 single object, but get(), set(), has() and enumerate() can virtualize multiple inheritance just fine. Without an enumerate() trap, I guess the properties of all prototypes can still be enumerated by having getPrototypeOf return either a synthesized "aggregate" prototype (containing the union of all inherited properties) or by synthesizing a linearized chain of prototypes. This is not as efficient or elegant as the direct enumerate() approach, but it seems like it could be made to work.

Would it make sense to leave `enumerate()` in but change its return type from an iterator to an array? If the iterator is exhausted and its results kept in memory anyway, specifying that the result must be an array will make that cost clear to developers. It would also ease the invariant checking.

allenwb commented on Jan 29

@tvcutsem

> Without an enumerate() trap, I guess the properties of all prototypes can still be enumerated by having getPrototypeOf return either a synthesized "aggregate" prototype (containing the union of all inherited properties) or by synthesizing a linearized chain of prototypes. This is not as efficient or elegant as the direct enumerate() approach, but it seems like it could be made to work.

that's precisely what I had been assuming. And I think it is just fine. If somebody is trying to create something like a prototype that encapsulates MI lookup then they should want its synthesized inheritance model to any client that introspects on inherited properties using [[GetPrototypeOf]], not just for-in enumeration.

domenic commented on Jan 29

My take on this issue is that it (1) is indeed a reasonable point from a purity perspective. However, we have to remember what exact feature is getting the shaft here: namely, for-in. For-in is in general considered a legacy feature, so saying that it is not as easy to virtualize as get/set/has seems OK. You can still do it; you still maintain full control of the getPrototype trap, after all. You'll just have to invest a little more work if you want for-in to work consistently with your get/set/has.

tvcutsem commented on Jan 30

@allenwb agreed that a good MI Proxy implementation will need to synthesize the proto-chain anyway for [[GetPrototypeOf]] to work.

Ok, so removing `enumerate()` would work (it was never a fundamental trap anyway), but I'd still like to know whether the option of having `enumerate()` return an array rather than an iterator would solve the most pressing issues and give us the best of both. It doesn't have the problem of causing observable iterator side-effects (other than invoking the trap, which is fine), it makes the memory cost manifest, it provides the VM with all property keys at once (supporting the snapshotting semantics), and it still allows easy virtualization of inherited properties. Thoughts?

rossberg-chromium commented on Feb 2

@tvcutsem, [[Enumerate]] introduces quite a bit of complexity and weirdness. It makes the separation of responsibilities between for-in and the different [[Enumerate]] methods rather fuzzy, see e.g. bugs ecma262/#160 and #161 and associated discussions and PRs. It exposes an underspecified method in the reflection API that is really just a hack for for-in. It would simplify matters significantly (and fix all open issues almost immediately) if we just got rid of it altogether and move all hacks to for-in.

tvcutsem commented on Feb 2

@rossberg-chromium Fair enough. If removing the [[Enumerate]] internal method would signficantly clean up the spec, that's also a good indicator it doesn't belong in the reflection API.

bterlson commented on Feb 2

I will work up a PR for this.

@tvcutsem thanks so much for working through this with us :-D

bterlson commented on Feb 5

Proposed fix is here: tc39/ecma262#367.

ljharb commented on Feb 5

Linking to tc39/proposal-object-values-entries#12 as well (minor spec text update in the proposal)

allenwb commented 16 minutes ago

Is there any way to transfer this issue to ecma262 repo? It has background info that should be public.

Probably a lesson that "Reflector" issues are best for getting attention of TC39 delegates but shouldn't be used for discussions that can/should be publicly visible.

ljharb commented 10 minutes ago

The only way would be to make this repo public, afaik.

Alternatively we could laboriously reconstruct the issue thread in order, and then only the timestamps would be wrong?

Write Preview

Leave a comment

Attach files by dragging & dropping or selecting them.

Styling with Markdown is supported

Close issue Comment