should Symbol.iterator fallback be a callable check or an undefined/null check?

iterator helpers stage 3 update

Michael Ficarra • May 2023

Normative: change Symbol.iterator fallback from callable check to undefined/null check #272

Edit | Code

Open | main ← michaelficarra-patch-1

Conversation 4 | Commits 1 | Checks 1 | Files changed 1 | +2 −2

michaelficarra 5 days ago • edited — Member

This better matches what we already do in `GetIterator` (used everywhere else we get an iterator from an iterable). If `obj` has a `Symbol.iterator` property that is non-callable, it will now throw instead of falling back to treating `obj` as an iterator.

Normative: change Symbol.iterator fallback from callable check to und… — Verified ✓ 9b3631a

michaelficarra mentioned this pull request 5 days ago

remove hint from GetIteratorFlattenable #271 — Merged

ljharb (Jordan Harband) 3 days ago — Member

What sort of code would be affected by this change?

Reviewers – review now

No reviews—at least 1 approving review is required.

Still in progress? Convert to draft

Assignees

No one—assign yourself

Labels

None yet

Projects

None yet

Milestone

No milestone

Development

Successfully merging this pull request may close these issues.

iterator helpers
(stage 3 proposal)

2.1.2 GetIteratorFlattenable (*obj*)

The abstract operation GetIteratorFlattenable takes argument *obj* (an ECMAScript language value) and returns either a normal completion containing an Iterator Record or a throw completion. It performs the following steps when called:

1. If *obj* is not an Object, throw a **TypeError** exception.
2. Let *method* be ? Get(*obj*, @@iterator).
3. If IsCallable(*method*) is **false**, then
    a. Let *iterator* be *obj*.
4. Else,
    a. Let *iterator* be ? Call(*method*, *obj*).
5. If *iterator* is not an Object, throw a **TypeError** exception.
6. Return ? GetIteratorDirect(*iterator*).

Symbol.iterator ⇒ "next"

ECMA-262

7.3.11 GetMethod (*V*, *P*)

1. Let *func* be ? GetV(*V*, *P*).
2. If *func* is either **undefined** or **null**, return **undefined**.
3. If IsCallable(*func*) is **false**, throw a **TypeError** exception.
4. Return *func*.

7.4.3 GetIterator (*obj*, *kind*)

The abstract operation GetIterator takes arguments *obj* (an ECMAScript language value) and *kind* (sync or async) and returns either a normal completion containing an Iterator Record or a throw completion. It performs the following steps when called:

1. If *kind* is async, then
    a. Let *method* be ? GetMethod(*obj*, @@asyncIterator).
    b. If *method* is **undefined**, then
        i. Let *syncMethod* be ? GetMethod(*obj*, @@iterator).
        ii. If *syncMethod* is **undefined**, throw a **TypeError** exception.
        iii. Let *syncIteratorRecord* be ? GetIteratorFromMethod(*obj*, *syncMethod*).
        iv. Return CreateAsyncFromSyncIterator(*syncIteratorRecord*).
2. Otherwise, let *method* be ? GetMethod(*obj*, @@iterator).
3. If *method* is **undefined**, throw a **TypeError** exception.
4. Return ? GetIteratorFromMethod(*obj*, *method*).

Symbol.asyncIterator ⇒ Symbol.iterator

Options

1. change Symbol.iterator ⇒ "next" fallback from IsCallable to undefined/null check
2. change existing Symbol.asyncIterator ⇒ Symbol.iterator fallback from undefined/null check to IsCallable
    a. unsure of impact on existing programs
3. leave them alone; inconsistency here is fine

Option 1

fall back on undefined/null

Example: non-callable values do not cause fallback

consider:

```
Iterator.from({
  [Symbol.iterator]: 0,
  next() { ... },
});
```

before

- `Iterator.from` does not consider the passed object to be iterable
- falls back to treating it as an iterator

after

- `Iterator.from` considers the passed object to be iterable
- throws when it tries to call `0` to get the iterator

Option 1

Option 2

fall back on non-callable

Example: non-callable values cause fallback

consider:

```
const brokenIterable = {
  [Symbol.asyncIterator]: 0,
  [Symbol.iterator]() { ... },
};

for await (let a of brokenIterable);
```

before

- for-await will consider `brokenIterable` to be async iterable because `Symbol.asyncIterator` is not undefined/null
- will try to call `0` to get an iterator, and it will throw

after

- for-await will not consider `brokenIterable` to be async iterable because `Symbol.asyncIterator` is not callable
- will fall back to `Symbol.iterator`, call it, and get an iterator

Option 2

Additional Considerations

1. async iterator helpers (stage 2) will add the async-to-sync fallback
   - Symbol.asyncIterator ⇒ Symbol.iterator ⇒ "next"
   - if we choose option 3, Symbol.asyncIterator ⇒ Symbol.iterator fallback will be inconsistent with either
     - existing Symbol.asyncIterator ⇒ Symbol.iterator fallback in GetIterator
     - the Symbol.iterator ⇒ "next" fallback added in GetIteratorFlattenable
2. this will likely set precedent for the first-class protocols proposal
3. **7.1.1.1 OrdinaryToPrimitive (*O*, *hint*)**

   The abstract operation OrdinaryToPrimitive takes arguments *O* (an Object) and *hint* (string or number) and returns either a normal completion containing an ECMAScript language value or a throw completion. It performs the following steps when called:

   1. If *hint* is string, then
      a. Let *methodNames* be « **"toString"**, **"valueOf"** ».
   2. Else,
      a. Let *methodNames* be « **"valueOf"**, **"toString"** ».
   3. For each element *name* of *methodNames*, do
      a. Let *method* be ? Get(*O*, *name*).
      b. If IsCallable(*method*) is **true**, then
         i. Let *result* be ? Call(*method*, *O*).
         ii. If *result* is not an Object, return *result*.
   4. Throw a **TypeError** exception.

Champion's Opinion

- option 3 is unacceptable
- option 2 has the most desirable semantics
  - but may not be worth the risk/effort
- hope for option 2, but option 1 is also acceptable
- reminder: we're only considering how already-broken programs break