ReverseIterable

https://github.com/leebyron/ecmascript-reverse-iterable

Lee Byron
Facebook

Iteration (old school)

```
// Forward iteration
for (var i = 0; i < a.length; i++) {
  var v = a[i];
  doSomething(v, i);
}

// Reverse iteration
for (var i = a.length - 1; i >= 0; i--) {
  var v = a[i];
  doSomething(v, i);
}
```

Iteration (ES6)

```
// Forward iteration
for (let [i, v] of a.entries()) {
  doSomething(v, i);
}


// Reverse iteration
// ???
```

Iterable Interface

```
// get iterator
var iterator = myIterable[Symbol.iterator]()

// many kinds of iterator for collections
var keys = myMap.keys()
var values = myMap.values()
var entries = myMap.entries()

// iterators are Iterable
var iterator = entries[Symbol.iterator]()
```

ReverseIterable Interface

- *ReverseIterable* is an *Iterable* which can produce an *Iterator* which yields values in the opposite order of it's forward-iterating `Symbol.iterator` method.
- New well-known symbol: `Symbol.reverseIterator` produces an *Iterator.*
- Objects which implement *Iterable* do not also have to implement *ReverseIterable.*
  - Example: Generators are not ReverseIterable.
- A *ReverseIterable* should be efficiently and incrementally iterated (no buffering!)

Reverse Iteration (ES7)

```
// Forward iteration
for (let [i, v] of a.entries()) {
  doSomething(v, i);
}

// Reverse iteration
for (let v of a[Symbol.reverseIterator]()) {
  doSomething(v);
}
```

Add *ReverseIterable* to existing Collections

Array, Map, Set, String, List

22.1.3 Properties of the Array Prototype Object

22.1.3.X Array.prototype [@@reverseIterator] ( )

> This property is new

1. Let *O* be the result of calling ToObject with the **this** value as its argument.
2. ReturnIfAbrupt(*O*).
3. Return CreateArrayReverseIterator(*O*, **"value"**).

22.1.X.1 CreateArrayReverseIterator Abstract Operation

1. Assert: Type(*array*) is Object
2. Let *iterator* be ObjectCreate(%ArrayIteratorPrototype%, ([[IteratedObject]], [[ArrayReverseIteratorNextIndex]], [[ArrayIterationKind]])).
3. Set *iterator's* [[IteratedObject]] internal slot to *array*.
4. If *array* has a [[TypedArrayName]] internal slot, then
    - a. Let *len* be the value of the [[ArrayLength]] internal slot of *array*.
5. Else,
    - a. Let *len* be ToLength(Get(*array*, **"length"**)).
    - b. ReturnIfAbrupt(*len*).
6. Set *iterator's* [[ArrayReverseIteratorNextIndex]] internal slot to *len*-1.
7. Set *iterator's* [[ArrayIteratorKind]] internal slot to *kind*.
8. Return *iterator*.

22.1.X.2 The %ArrayReverseIteratorPrototype% Object

All Array Reverse Iterator Objects inherit properties from the %ArrayReverseIteratorPrototype% intrinsic object. The %ArrayReverseIteratorPrototype% object is an ordinary object and its [[Prototype]] internal slot is the %IteratorPrototype% intrinsic object (25.1.2). In addition, %ArrayReverseIteratorPrototype% has the following properties:

22.1.X.2.1 %ArrayReverseIteratorPrototype%.next ( )

1. Let *O* be the **this** value.
2. If Type(*O*) is not Object, throw a **TypeError** exception.

Collection Reverse Iterators

```
// Forward iteration
for (let [i, v] of a.entries()) {
  doSomething(v, i);
}

// Reverse iteration
for (let v of a[Symbol.reverseIterator]()) {
  doSomething(v);
}
```

Collection Reverse Iterators

- Reverse equivalents of keys(), values(), and entries()?
- We have a few options:

Explicit reverse iterator functions

```
// Forward iteration
for (let [i, v] of a.entries()) {
  doSomething(v, i);
}

// Reverse iteration
for (let [i, v] of a.reverseEntries()) {
  doSomething(v, i);
}
```

Explicit reverse iterator functions

- Pro: Straight-forward and obvious.
- Meh: Expands collection API.
- Con: Couple concept of choosing key/value/entries to choosing reversal.
- Con: Must only start a "chain" of methods, excludes some "itertool" possibilities:
  - `myArray.reverseEntries().map(fn).filter(fn)`

Iterators may be *ReverseIterable*

```
// Forward iteration
for (let [i, v] of a.entries()) {
  doSomething(v, i);
}

// Reverse iteration
for (let [i, v] of a.entries()[Symbol.reverseIterable]()) {
  doSomething(v, i);
}
```

Iterators may be *ReverseIterable*

- Con: Not a great User API.
- Pro: Separate concepts of choosing key/value/entries to choosing reversal.
- Pro: Can exist anywhere in a chain (if previous *Iterator* is *ReverseIterable*)
  - `myArray.entries().map(fn)[Symbol.reverseIterator]().filter(fn)`
- Pro: "Itertools" style utilities desiring reversibility can operate on *Iterators* which are *ReverseIterable*.
  - `takeLast = (num, iter) => take(num, iter[Symbol.reverseIterator]())`
  - `takeLast(5, map(myArray.entries()))`

Iterators may be *ReverseIterable*, with explicit reverse method

```
// Forward iteration
for (let [i, v] of a.entries()) {
  doSomething(v, i);
}

// Reverse iteration
for (let [i, v] of a.entries().reverse()) {
  doSomething(v, i);
}
```

Iterators may be *ReverseIterable*, with explicit reverse method

```
%IteratorPrototype%.reverse = function() {
  let O = Object(this)
  let usingReverseIterator = O[Symbol.reverseIterator]
  if (usingReverseIterator === undefined) {
    // only ReverseIterable can be reversed
    throw new TypeError('Iterator is not reversible.');
  }
  let iterator = usingReverseIterator.call(O)
  return iterator
}
```

Iterators may be *ReverseIterable*, with explicit reverse method

```
%IteratorPrototype%.reverse = function() {
  let O = Object(this)
  let usingReverseIterator = O[Symbol.reverseIterator]
  if (usingReverseIterator === undefined) {
    // any Iterable can be reversed by buffering
    return [...this][Symbol.reverseIterator]();
  }
  let iterator = usingReverseIterator.call(O)
  return iterator
}
```

Iterators may be *ReverseIterable*, with explicit reverse method

- Pro: Expected user-level API
- Pro: API similar to `array.reverse()` (similar semantics, similar name!)
- Con: API similar to `array.reverse()` (different semantics, unfortunately.)
- Con: `iterator.reverse()` exists on all Iterators, but is only valid/cheap on *ReverseIterable* Iterators.
  - Throw *TypeError* when used on non-*ReverseIterable*? Or buffer first, which could have difficult to understand perf?
  - May not be obvious to user when non-optimal case occurs.
- Con: semantically ambiguous when `iterator.reverse()` called after `iterator.next()` - how much state is preserved? Buffering vs Non-buffering cases must remain equivalent which is more burden on implementers. Should likely throw.

Examples from other languages

Python

```
seq = ["a", "b", "c"]

// seq is indexible, and thus a reversed iterator is
// efficiently created.
for i in reversed(seq)
  print i

// since iterators are not reversible, list() must be
// used to buffer the enumerate() iterator.
for i, e in reversed(list(enumerate(a))):
  print i, e
```

ObjC

```
a = @["a", "b", "c"]

// NSArray has a method which produces a reverse
// "enumerator"
for (id i in [a reverseObjectEnumerator]) {
   NSLog(@"%@", i);
}

d = [NSDictionary new];

// NSDictionary has separate methods for enumerating
// keys. NSDictionary has no order, if reversible, it
// would likely be called reverseKeyEnumerator
for (id i in [d keyEnumerator]) {
   NSLog(@"%@", i);
}
```

C#

```
// IEnumerable : Iterable ::
// IEnumerator : Iterator

// IEnumerable has a method Reverse() which returns
// another IEnumerable. It uses a stack to buffer
// values before yielding to protect against edits while
// iterating.
// Some user-land implementations have added
// non-buffering.
foreach (var el in enumerable.Reverse())
{
  // do work with el
}
```

Scala

```
def reverseIterator: Iterator[A]
// An iterator yielding elements in reversed order.

Note: will not terminate for infinite-sized collections.

Note: xs.reverseIterator is the same as
xs.reverse.iterator but might be more efficient.

Optimizes when possible, but falls back to buffering.
```

Ruby

```
a = [ "a", "b", "c" ]
a.reverse_each {|x| print x, " " }
// c b a

reverse_each is a method on Array which
returns an Enumerator.

e = a.each
e.reverse_each {|x| print x, " " }

reverse_each is a method on Enumerable which creates
a temporary array as a buffer.
```

Java

```
// No common interface for this
// reverse iterate List, but not in common-lib

// org.apache.commons.collections.iterators.ReverseListIterator
reverseIterator = ReverseListIterator(list)
```

C++

```
// std::reverse_iterator
// Operates on another Iterator
// Expects underlying Iterator to be bi-directional
template <class Iterator> class reverse_iterator;

// std::vector::rbegin
// Produces a reverse iterator from a vector.
std::vector<int>::reverse_iterator rit = myvector.rbegin();
for (; rit != myvector.rend(); ++rit)
  std::cout << ' ' << *rit;
```

"rbegin" and "rend" are convention in all std collections.