facebook / **immutable-js**

Watch 588 | Star 24,588 | Fork 1,431

Code | Issues 69 | Pull requests 13 | Projects 0 | Wiki | Insights

Immutable persistent data collections for Javascript which increase efficiency and simplicity.
http://facebook.github.io/immutable-js/

1,509 commits | **4** branches | **62** releases | **172** contributors | MIT

Branch: **master** | New pull request | Find file | Clone or download

leebyron Improve docs around Record Types and classes … Latest commit `e65e5af` on May 18

.github	Reduce repetition in COC link `CONTRIBUTING.md`	8 months ago
__tests__	Improve docs around Record Types and classes	3 months ago
contrib/cursor	Fixing some minor typos (#1439)	3 months ago
pages	Update eslint & prettier (#1524)	3 months ago
perf	Adds perf tests for `Map.merge`. (#1407)	10 months ago
resources	Update rollup version (#1526)	3 months ago
src	Make isArrayLike check more precise to avoid false positives (#1520)	3 months ago
type-definitions	Improve docs around Record Types and classes	3 months ago
.eslintrc.json	Update eslint & prettier (#1524)	3 months ago
.gitignore	Adds a better gh-pages deploy script (#1348)	10 months ago
.travis.yml	Update travis node version (#1528)	3 months ago
LICENSE	Relicense as MIT (#1320)	10 months ago
README.md	Add documentation around toJS/toJSON	3 months ago
package.json	Update rollup version (#1526)	3 months ago
tslint.json	Prettify: Use es5 trailing commas	10 months ago
yarn.lock	Update rollup version (#1526)	3 months ago

README.md

Immutable collections for JavaScript

build passing

Immutable data cannot be changed once created, leading to much simpler application development, no defensive copying, and enabling advanced memoization and change detection techniques with simple logic. Persistent data presents a mutative API which does not update the data in-place, but instead always yields new updated data.

Immutable.js provides many Persistent Immutable data structures including: `List`, `Stack`, `Map`, `OrderedMap`, `Set`, `OrderedSet` and `Record`.

These data structures are highly efficient on modern JavaScript VMs by using structural sharing via hash maps tries and vector tries as popularized by Clojure and Scala, minimizing the need to copy or cache data.

Immutable.js also provides a lazy `Seq`, allowing efficient chaining of collection methods like `map` and `filter` without creating intermediate representations. Create some `Seq` with `Range` and `Repeat`.

Want to hear more? Watch the presentation about Immutable.js:

Alan Kay
Xerox PARC, Creator of Smalltalk
The last thing you wanted any programmer to do is mess with internal state even if presented figuratively. It is unfortunate that much of what is called "object-oriented programming" today is simply old style programming with fancier constructs.
"The Early History of Smalltalk"

Getting started

Install `immutable` using npm.

```
npm install immutable
```

Then require it into any module.

```
const { Map } = require('immutable')
const map1 = Map({ a: 1, b: 2, c: 3 })
const map2 = map1.set('b', 50)
map1.get('b') + " vs. " + map2.get('b') // 2 vs. 50
```

Browser

Immutable.js has no dependencies, which makes it predictable to include in a Browser.

It's highly recommended to use a module bundler like webpack, rollup, or browserify. The `immutable` npm module works without any additional consideration. All examples throughout the documentation will assume use of this kind of tool.

Alternatively, Immutable.js may be directly included as a script tag. Download or link to a CDN such as CDNJS or jsDelivr.

Use a script tag to directly add `Immutable` to the global scope:

```
<script src="immutable.min.js"></script>
<script>
```

```
  var map1 = Immutable.Map({a:1, b:2, c:3});
  var map2 = map1.set('b', 50);
  map1.get('b'); // 2
  map2.get('b'); // 50
</script>
```

Or use an AMD-style loader (such as RequireJS):

```
require(['./immutable.min.js'], function (Immutable) {
  var map1 = Immutable.Map({a:1, b:2, c:3});
  var map2 = map1.set('b', 50);
  map1.get('b'); // 2
  map2.get('b'); // 50
});
```

Flow & TypeScript

Use these Immutable collections and sequences as you would use native collections in your Flowtype or TypeScript programs while still taking advantage of type generics, error detection, and auto-complete in your IDE.

Installing `immutable` via npm brings with it type definitions for Flow (v0.55.0 or higher) and TypeScript (v2.1.0 or higher), so you shouldn't need to do anything at all!

Using TypeScript with Immutable.js v4

Immutable.js type definitions embrace ES2015. While Immutable.js itself supports legacy browsers and environments, its type definitions require TypeScript's 2015 lib. Include either `"target": "es2015"` or `"lib": "es2015"` in your `tsconfig.json`, or provide `--target es2015` or `--lib es2015` to the `tsc` command.

```
const { Map } = require("immutable");
const map1 = Map({ a: 1, b: 2, c: 3 });
const map2 = map1.set('b', 50);
map1.get('b') + " vs. " + map2.get('b') // 2 vs. 50
```

Using TypeScript with Immutable.js v3 and earlier:

Previous versions of Immutable.js include a reference file which you can include via relative path to the type definitions at the top of your file.

```
///<reference path='./node_modules/immutable/dist/immutable.d.ts'/>
import Immutable from require('immutable');
var map1: Immutable.Map<string, number>;
map1 = Immutable.Map({a:1, b:2, c:3});
var map2 = map1.set('b', 50);
map1.get('b'); // 2
map2.get('b'); // 50
```

The case for Immutability

Much of what makes application development difficult is tracking mutation and maintaining state. Developing with immutable data encourages you to think differently about how data flows through your application.

Subscribing to data events throughout your application creates a huge overhead of book-keeping which can hurt performance, sometimes dramatically, and creates opportunities for areas of your application to get out of sync with each other due to easy to make programmer error. Since immutable data never changes, subscribing to changes throughout the model is a dead-end and new data can only ever be passed from above.

This model of data flow aligns well with the architecture of React and especially well with an application designed using the ideas of Flux.

When data is passed from above rather than being subscribed to, and you're only interested in doing work when something has changed, you can use equality.

Immutable collections should be treated as *values* rather than *objects*. While objects represent some thing which could change over time, a value represents the state of that thing at a particular instance of time. This principle is most important to understanding the appropriate use of immutable data. In order to treat Immutable.js collections as values, it's important to use the `Immutable.is()` function or `.equals()` method to determine *value equality* instead of the `===` operator which determines object *reference identity*.

```
const { Map } = require('immutable')
const map1 = Map({ a: 1, b: 2, c: 3 })
const map2 = Map({ a: 1, b: 2, c: 3 })
map1.equals(map2) // true
map1 === map2 // false
```

Note: As a performance optimization Immutable.js attempts to return the existing collection when an operation would result in an identical collection, allowing for using `===` reference equality to determine if something definitely has not changed. This can be extremely useful when used within a memoization function which would prefer to re-run the function if a deeper equality check could potentially be more costly. The `===` equality check is also used internally by `Immutable.is` and `.equals()` as a performance optimization.

```
const { Map } = require('immutable')
const map1 = Map({ a: 1, b: 2, c: 3 })
const map2 = map1.set('b', 2) // Set to same value
map1 === map2 // true
```

If an object is immutable, it can be "copied" simply by making another reference to it instead of copying the entire object. Because a reference is much smaller than the object itself, this results in memory savings and a potential boost in execution speed for programs which rely on copies (such as an undo-stack).

```
const { Map } = require('immutable')
const map = Map({ a: 1, b: 2, c: 3 })
const mapCopy = map; // Look, "copies" are free!
```

JavaScript-first API

While Immutable.js is inspired by Clojure, Scala, Haskell and other functional programming environments, it's designed to bring these powerful concepts to JavaScript, and therefore has an Object-Oriented API that closely mirrors that of ES2015 Array, Map, and Set.

The difference for the immutable collections is that methods which would mutate the collection, like `push`, `set`, `unshift` or `splice`, instead return a new immutable collection. Methods which return new arrays, like `slice` or `concat`, instead return new immutable collections.

```
const { List } = require('immutable')
const list1 = List([ 1, 2 ]);
const list2 = list1.push(3, 4, 5);
const list3 = list2.unshift(0);
const list4 = list1.concat(list2, list3);
assert.equal(list1.size, 2);
assert.equal(list2.size, 5);
assert.equal(list3.size, 6);
assert.equal(list4.size, 13);
assert.equal(list4.get(0), 1);
```

Almost all of the methods on Array will be found in similar form on `Immutable.List`, those of Map found on `Immutable.Map`, and those of Set found on `Immutable.Set`, including collection operations like `forEach()` and `map()`.

```
const { Map } = require('immutable')
const alpha = Map({ a: 1, b: 2, c: 3, d: 4 });
alpha.map((v, k) => k.toUpperCase()).join();
// 'A,B,C,D'
```

Convert from raw JavaScript objects and arrays.

Designed to inter-operate with your existing JavaScript, Immutable.js accepts plain JavaScript Arrays and Objects anywhere a method expects a `Collection`.

```
const { Map, List } = require('immutable')
const map1 = Map({ a: 1, b: 2, c: 3, d: 4 })
const map2 = Map({ c: 10, a: 20, t: 30 })
const obj = { d: 100, o: 200, g: 300 }
const map3 = map1.merge(map2, obj);
// Map { a: 20, b: 2, c: 10, d: 100, t: 30, o: 200, g: 300 }
const list1 = List([ 1, 2, 3 ])
const list2 = List([ 4, 5, 6 ])
const array = [ 7, 8, 9 ]
const list3 = list1.concat(list2, array)
// List [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
```

This is possible because Immutable.js can treat any JavaScript Array or Object as a Collection. You can take advantage of this in order to get sophisticated collection methods on JavaScript Objects, which otherwise have a very sparse native API. Because Seq evaluates lazily and does not cache intermediate results, these operations can be extremely efficient.

```
const { Seq } = require('immutable')
const myObject = { a: 1, b: 2, c: 3 }
Seq(myObject).map(x => x * x).toObject();
// { a: 1, b: 4, c: 9 }
```

Keep in mind, when using JS objects to construct Immutable Maps, that JavaScript Object properties are always strings, even if written in a quote-less shorthand, while Immutable Maps accept keys of any type.

```
const { fromJS } = require('immutable')

const obj = { 1: "one" }
console.log(Object.keys(obj)) // [ "1" ]
console.log(obj["1"], obj[1]) // "one", "one"

const map = fromJS(obj)
console.log(map.get("1"), map.get(1)) // "one", undefined
```

Property access for JavaScript Objects first converts the key to a string, but since Immutable Map keys can be of any type the argument to `get()` is not altered.

Converts back to raw JavaScript objects.

All Immutable.js Collections can be converted to plain JavaScript Arrays and Objects shallowly with `toArray()` and `toObject()` or deeply with `toJS()`. All Immutable Collections also implement `toJSON()` allowing them to be passed to `JSON.stringify` directly. They also respect the custom `toJSON()` methods of nested objects.

```
const { Map, List } = require('immutable')
const deep = Map({ a: 1, b: 2, c: List([ 3, 4, 5 ]) })
console.log(deep.toObject()) // { a: 1, b: 2, c: List [ 3, 4, 5 ] }
console.log(deep.toArray()) // [ 1, 2, List [ 3, 4, 5 ] ]
console.log(deep.toJS()) // { a: 1, b: 2, c: [ 3, 4, 5 ] }
JSON.stringify(deep) // '{"a":1,"b":2,"c":[3,4,5]}'
```

Embraces ES2015

Immutable.js supports all JavaScript environments, including legacy browsers (even IE8). However it also takes advantage of features added to JavaScript in ES2015, the latest standard version of JavaScript, including Iterators, Arrow Functions, Classes, and Modules. It's inspired by the native Map and Set collections added to ES2015.

All examples in the Documentation are presented in ES2015. To run in all browsers, they need to be translated to ES5.

```
// ES2015
const mapped = foo.map(x => x * x);
// ES5
var mapped = foo.map(function (x) { return x * x; });
```

All Immutable.js collections are Iterable, which allows them to be used anywhere an Iterable is expected, such as when spreading into an Array.

```
const { List } = require('immutable')
const aList = List([ 1, 2, 3 ])
const anArray = [ 0, ...aList, 4, 5 ] // [ 0, 1, 2, 3, 4, 5 ]
```

Note: A Collection is always iterated in the same order, however that order may not always be well defined, as is the case for the `Map` and `Set`.

Nested Structures

The collections in Immutable.js are intended to be nested, allowing for deep trees of data, similar to JSON.

```
const { fromJS } = require('immutable')
const nested = fromJS({ a: { b: { c: [ 3, 4, 5 ] } } })
// Map { a: Map { b: Map { c: List [ 3, 4, 5 ] } } }
```

A few power-tools allow for reading and operating on nested data. The most useful are `mergeDeep`, `getIn`, `setIn`, and `updateIn`, found on `List`, `Map` and `OrderedMap`.

```
const { fromJS } = require('immutable')
const nested = fromJS({ a: { b: { c: [ 3, 4, 5 ] } } })

const nested2 = nested.mergeDeep({ a: { b: { d: 6 } } })
// Map { a: Map { b: Map { c: List [ 3, 4, 5 ], d: 6 } } }

console.log(nested2.getIn([ 'a', 'b', 'd' ])) // 6

const nested3 = nested2.updateIn([ 'a', 'b', 'd' ], value => value + 1)
console.log(nested3);
// Map { a: Map { b: Map { c: List [ 3, 4, 5 ], d: 7 } } }

const nested4 = nested3.updateIn([ 'a', 'b', 'c' ], list => list.push(6))
// Map { a: Map { b: Map { c: List [ 3, 4, 5, 6 ], d: 7 } } }
```

Equality treats Collections as Values

Immutable.js collections are treated as pure data *values*. Two immutable collections are considered *value equal* (via `.equals()` or `is()`) if they represent the same collection of values. This differs from JavaScript's typical *reference equal* (via `===` or `==`) for Objects and Arrays which only determines if two variables represent references to the same object instance.

Consider the example below where two identical `Map` instances are not *reference equal* but are *value equal*.

```
// First consider:
const obj1 = { a: 1, b: 2, c: 3 }
const obj2 = { a: 1, b: 2, c: 3 }
obj1 !== obj2 // two different instances are always not equal with ===

const { Map, is } = require('immutable')
const map1 = Map({ a: 1, b: 2, c: 3 })
const map2 = Map({ a: 1, b: 2, c: 3 })
map1 !== map2 // two different instances are not reference-equal
map1.equals(map2) // but are value-equal if they have the same values
is(map1, map2) // alternatively can use the is() function
```

Value equality allows Immutable.js collections to be used as keys in Maps or values in Sets, and retrieved with different but equivalent collections:

```
const { Map, Set } = require('immutable')
const map1 = Map({ a: 1, b: 2, c: 3 })
const map2 = Map({ a: 1, b: 2, c: 3 })
const set = Set().add(map1)
set.has(map2) // true because these are value-equal
```

Note: `is()` uses the same measure of equality as Object.is for scalar strings and numbers, but uses value equality for Immutable collections, determining if both are immutable and all keys and values are equal using the same measure of equality.

Performance tradeoffs

While value equality is useful in many circumstances, it has different performance characteristics than reference equality. Understanding these tradeoffs may help you decide which to use in each case, especially when used to memoize some operation.

When comparing two collections, value equality may require considering every item in each collection, on an `O(N)` time complexity. For large collections of values, this could become a costly operation. Though if the two are not equal and hardly similar, the inequality is determined very quickly. In contrast, when comparing two collections with reference equality, only the initial references to memory need to be compared which is not based on the size of the collections, which has an `O(1)` time complexity. Checking reference equality is always very fast, however just because two collections are not reference-equal does not rule out the possibility that they may be value-equal.

Return self on no-op optimization

When possible, Immutable.js avoids creating new objects for updates where no change in *value* occurred, to allow for efficient *reference equality* checking to quickly determine if no change occurred.

```
const { Map } = require('immutable')
const originalMap = Map({ a: 1, b: 2, c: 3 })
const updatedMap = originalMap.set('b', 2)
updatedMap === originalMap // No-op .set() returned the original reference.
```

However updates which do result in a change will return a new reference. Each of these operations occur independently, so two similar updates will not return the same reference:

```
const { Map } = require('immutable')
const originalMap = Map({ a: 1, b: 2, c: 3 })
const updatedMap = originalMap.set('b', 1000)
// New instance, leaving the original immutable.
updatedMap !== originalMap
const anotherUpdatedMap = originalMap.set('b', 1000)
// Despite both the results of the same operation, each created a new reference.
anotherUpdatedMap !== updatedMap
```

```
// However the two are value equal.
anotherUpdatedMap.equals(updatedMap)
```

Batching Mutations

> If a tree falls in the woods, does it make a sound?
>
> If a pure function mutates some local data in order to produce an immutable return value, is that ok?
>
> — Rich Hickey, Clojure

Applying a mutation to create a new immutable object results in some overhead, which can add up to a minor performance penalty. If you need to apply a series of mutations locally before returning, Immutable.js gives you the ability to create a temporary mutable (transient) copy of a collection and apply a batch of mutations in a performant manner by using `withMutations`. In fact, this is exactly how Immutable.js applies complex mutations itself.

As an example, building `list2` results in the creation of 1, not 3, new immutable Lists.

```
const { List } = require('immutable')
const list1 = List([ 1, 2, 3 ]);
const list2 = list1.withMutations(function (list) {
  list.push(4).push(5).push(6);
});
assert.equal(list1.size, 3);
assert.equal(list2.size, 6);
```

Note: Immutable.js also provides `asMutable` and `asImmutable`, but only encourages their use when `withMutations` will not suffice. Use caution to not return a mutable copy, which could result in undesired behavior.

Important!: Only a select few methods can be used in `withMutations` including `set`, `push` and `pop`. These methods can be applied directly against a persistent data-structure where other methods like `map`, `filter`, `sort`, and `splice` will always return new immutable data-structures and never mutate a mutable collection.

Lazy Seq

`Seq` describes a lazy operation, allowing them to efficiently chain use of all the higher-order collection methods (such as `map` and `filter`) by not creating intermediate collections.

Seq is immutable — Once a Seq is created, it cannot be changed, appended to, rearranged or otherwise modified. Instead, any mutative method called on a `Seq` will return a new `Seq`.

Seq is lazy — `Seq` does as little work as necessary to respond to any method call. Values are often created during iteration, including implicit iteration when reducing or converting to a concrete data structure such as a `List` or JavaScript `Array`.

For example, the following performs no work, because the resulting `Seq`'s values are never iterated:

```
const { Seq } = require('immutable')
const oddSquares = Seq([ 1, 2, 3, 4, 5, 6, 7, 8 ])
  .filter(x => x % 2 !== 0)
  .map(x => x * x)
```

Once the `Seq` is used, it performs only the work necessary. In this example, no intermediate arrays are ever created, filter is called three times, and map is only called once:

```
oddSquares.get(1); // 9
```

Any collection can be converted to a lazy Seq with `Seq()`.

```
const { Map, Seq } = require('immutable')
const map = Map({ a: 1, b: 2, c: 3 })
const lazySeq = Seq(map)
```

`Seq` allows for the efficient chaining of operations, allowing for the expression of logic that can otherwise be very tedious:

```
lazySeq
  .flip()
  .map(key => key.toUpperCase())
  .flip()
// Seq { A: 1, B: 1, C: 1 }
```

As well as expressing logic that would otherwise seem memory or time limited, for example `Range` is a special kind of Lazy sequence.

```
const { Range } = require('immutable')
Range(1, Infinity)
  .skip(1000)
  .map(n => -n)
  .filter(n => n % 2 === 0)
  .take(2)
  .reduce((r, n) => r * n, 1)
// 1006008
```

Documentation

Read the docs and eat your vegetables.

Docs are automatically generated from Immutable.d.ts. Please contribute!

Also, don't miss the Wiki which contains articles on specific topics. Can't find something? Open an issue.

Testing

If you are using the Chai Assertion Library, Chai Immutable provides a set of assertions to use against Immutable.js collections.

Contribution

Use Github issues for requests.

We actively welcome pull requests, learn how to contribute.

Changelog

Changes are tracked as Github releases.

Thanks

Phil Bagwell, for his inspiration and research in persistent data structures.

Hugh Jackson, for providing the npm package name. If you're looking for his unsupported package, see this repository.

License

Immutable.js is MIT-licensed.