
fast.js

build unknown

Faster user-land reimplementations for several common builtin native JavaScript functions.

Note: fast.js is very young and in active development. The current version is optimised for V8 (chrome / node.js) and may not perform well in other JavaScript engines, so you may not want to use it in the browser at this point. Please read the caveats section before using fast.js.

What?

Fast.js is a collection of micro-optimisations aimed at making writing very fast JavaScript programs easier. It includes fast replacements for several built-in native methods such as `.forEach`, `.map`, `.reduce` etc, as well as common utility methods such as `.clone`.

Installation

Via npm:

```
1 npm install --save fast.js
```

Usage

```
1 var fast = require('fast.js');
2 console.log(fast.map([1,2,3], function (a) { return a * a; }));
```

How?

Thanks to advances in JavaScript engines such as V8 there is essentially no performance difference between native functions and their JavaScript equivalents, providing the developer is willing to go the extra mile to write very fast code. In fact, native functions often have to cover complicated edge cases from the ECMAScript specification, which put them at a performance disadvantage.

An example of such an edge case is sparse arrays and the `.map`, `.reduce` and `.forEach` functions:

```
1 var arr = new Array(100); // a sparse array with 100 slots
2
3 arr[20] = 'Hello World';
```

```
4
5 function logIt (item) {
6   console.log(item);
7 }
8
9 arr.forEach(logIt);
```

In the above example, the `logIt` function will be called only once, despite there being 100 slots in the array. This is because 99 of those slots are empty. To implement this behavior according to spec, the native `forEach` function must check whether each slot in the array has ever been assigned or not (a simple `null` or `undefined` check is not sufficient), and if so, the `logIt` function will be called.

However, almost no one actually uses this pattern - sparse arrays are very rare in the real world. But the native function must still perform this check, just in case. If we ignore the concept of sparse arrays completely, and pretend that they don't exist, we can write a JavaScript function which comfortably beats the native version:

```
1 var fast = require('fast.js');
2
3 var arr = [1,2,3,4,5];
4
5 fast.forEach(arr, logIt); // faster than arr.forEach(logIt)
```

By optimising for the 99% use case, `fast.js` methods can be up to 5x faster than their native equivalents.

Caveats

As mentioned above, `fast.js` does not conform 100% to the ECMAScript specification and is therefore not a drop in replacement 100% of the time. There are at least three scenarios where the behavior differs from the spec:

- Sparse arrays are not supported. A sparse array will be treated just like a normal array, with unpopulated slots containing `undefined` values. This means that iteration functions such as `.map()` and `.forEach()` will visit these empty slots, receiving `undefined` as an argument. This is in contrast to the native implementations where these unfilled slots will be skipped entirely by the iterators. In the real world, sparse arrays are very rare. This is evidenced by the very popular `underscore.js`'s lack of support.
- Functions created using `fast.bind()` and `fast.partial()` are not identical to functions created by the native `Function.prototype.bind()`, specifically:
 - The partial implementation creates functions that do not have immutable “poison pill” `caller` and `arguments` properties that throw a `TypeError` upon `get`, `set`, or `deletion`.

-
- The partial implementation creates functions that have a `prototype` property. (Proper bound functions have none.)
 - The partial implementation creates bound functions whose `length` property does not agree with that mandated by ECMA-262: it creates functions with `length 0`, while a full implementation, depending on the length of the target function and the number of pre-specified arguments, may return a non-zero length.

See the documentation for `Function.prototype.bind()` on MDN for more details.

- The behavior of `fast.reduce()` differs from the native `Array.prototype.reduce()` in some important ways.
 - Specifying an `undefined initialValue` is the same as specifying no initial value at all. This differs from the spec which looks at the number of arguments specified. We just do a simple check for `undefined` which may lead to unexpected results in some circumstances - if you're relying on the normal behavior of `reduce` when an initial value is specified, make sure that that value is not `undefined`. You can usually use `null` as an alternative and `null` will not trigger this edge case.
 - A 4th argument is supported - `thisContext`, the context to bind the reducer function to. This is not present in the spec but is provided for convenience.

In practice, it's extremely unlikely that any of these caveats will have an impact on real world code. These constructs are extremely uncommon.

Benchmarks

To run the benchmarks in `node.js`:

```
1 npm run bench
```

To run the benchmarks in SpiderMonkey, you must first download `js-shell`. If you're on linux, this can be done by running:

```
1 npm run install-sm
```

This will download the latest nightly build of the `js-shell` binary and extract it to `ci/environments/sm`. If you're on mac or windows, you should download the appropriate build for your platform and place the extracted files in that directory.

After `js-shell` has been downloaded, you can run the SpiderMonkey benchmarks by running:

```
1 npm run bench-sm
```

Example benchmark output

```
1 > node ./bench/index.js
2
3 Running 55 benchmarks, please wait...
4
5 Native .fill() vs fast.fill() (3 items)✓
6   Array.prototype.fill() x 19,878,369 ops/sec ±1.81% (90 runs
   sampled)✓
7   fast.fill() x 160,929,619 ops/sec ±1.92% (89 runs sampled)
8
9   Result: fast.js is 709.57% faster than Array.prototype.fill().
10
11 Native .fill() vs fast.fill() (10 items)✓
12   Array.prototype.fill() x 11,285,333 ops/sec ±2.13% (86 runs
   sampled)✓
13   fast.fill() x 48,641,408 ops/sec ±4.87% (85 runs sampled)
14
15   Result: fast.js is 331.01% faster than Array.prototype.fill().
16
17 Native .fill() vs fast.fill() (1000 items)✓
18   Array.prototype.fill() x 283,166 ops/sec ±1.65% (89 runs sampled)
   ✓
19   fast.fill() x 476,660 ops/sec ±1.43% (90 runs sampled)
20
21   Result: fast.js is 68.33% faster than Array.prototype.fill().
22
23 Native .reduce() plucker vs fast.pluck()✓
24   Native Array::reduce() plucker x 1,041,300 ops/sec ±2.13% (87
   runs sampled)✓
25   fast.pluck() x 491,417 ops/sec ±0.97% (93 runs sampled)✓
26   underscore.pluck() x 487,872 ops/sec ±1.06% (92 runs sampled)✓
27   lodash.pluck():
28
29   Result: fast.js is 52.81% slower than Native Array::reduce()
   plucker.
30
31 Native Object.keys().map() value extractor vs fast.values()✓
32   Native Object.keys().map() x 5,435,909 ops/sec ±1.47% (90 runs
   sampled)✓
33   fast.values() x 11,500,439 ops/sec ±2.18% (83 runs sampled)✓
34   underscore.values() x 5,543,090 ops/sec ±1.41% (91 runs sampled)✓
35   lodash.values() x 4,081,797 ops/sec ±1.55% (90 runs sampled)
36
37   Result: fast.js is 111.56% faster than Native Object.keys().map().
38
39 Object.assign() vs fast.assign()✓
40   Object.assign() x 250,190 ops/sec ±1.36% (93 runs sampled)✓
41   fast.assign() x 208,612 ops/sec ±1.44% (87 runs sampled)✓
42   fast.assign() v0.0.4c x 212,198 ops/sec ±1.67% (87 runs sampled)✓
43   fast.assign() v0.0.4b x 197,658 ops/sec ±1.34% (89 runs sampled)✓
```

```

44     lodash.assign() x 163,550 ops/sec ±1.34% (87 runs sampled)
45
46     Result: fast.js is 16.62% slower than Object.assign().
47
48     Object.assign() vs fast.assign() (3 arguments)✓
49         Object.assign() x 81,027 ops/sec ±1.61% (88 runs sampled)✓
50         fast.assign() x 72,334 ops/sec ±1.06% (91 runs sampled)✓
51         fast.assign() v0.0.4c x 73,304 ops/sec ±1.05% (87 runs sampled)✓
52         fast.assign() v0.0.4b x 63,523 ops/sec ±1.22% (92 runs sampled)
53
54     Result: fast.js is 10.73% slower than Object.assign().
55
56     Object.assign() vs fast.assign() (10 arguments)✓
57         Object.assign() x 25,287 ops/sec ±3.27% (80 runs sampled)✓
58         fast.assign() x 24,588 ops/sec ±1.63% (91 runs sampled)✓
59         fast.assign() v0.0.4c x 24,207 ops/sec ±2.34% (87 runs sampled)✓
60         fast.assign() v0.0.4b x 20,558 ops/sec ±2.15% (85 runs sampled)
61
62     Result: fast.js is 2.77% slower than Object.assign().
63
64     Native string comparison vs fast.intern() (short)✓
65         Native comparison x 85,392,447 ops/sec ±21.72% (89 runs sampled)✓
66         fast.intern() x 184,548,307 ops/sec ±0.59% (99 runs sampled)
67
68     Result: fast.js is 116.12% faster than Native comparison.
69
70     Native string comparison vs fast.intern() (medium)✓
71         Native comparison x 3,761,682 ops/sec ±20.61% (95 runs sampled)✓
72         fast.intern() x 182,842,946 ops/sec ±0.77% (97 runs sampled)
73
74     Result: fast.js is 4760.67% faster than Native comparison.
75
76     Native string comparison vs fast.intern() (long)✓
77         Native comparison x 19,951 ops/sec ±0.63% (97 runs sampled)✓
78         fast.intern() x 179,058,362 ops/sec ±1.58% (94 runs sampled)
79
80     Result: fast.js is 897376.33% faster than Native comparison.
81
82     Native try {} catch (e) {} vs fast.try()✓
83         try...catch x 163,886 ops/sec ±1.13% (93 runs sampled)✓
84         fast.try() x 2,886,759 ops/sec ±1.93% (86 runs sampled)
85
86     Result: fast.js is 1661.44% faster than try...catch.
87
88     Native try {} catch (e) {} vs fast.try() (single function call)✓
89         try...catch x 174,874 ops/sec ±0.57% (89 runs sampled)✓
90         fast.try() x 2,778,079 ops/sec ±1.60% (86 runs sampled)
91
92     Result: fast.js is 1488.61% faster than try...catch.
93
94     Native .apply() vs fast.apply() (3 items, no context)✓

```

```

95     Function::apply() x 30,268,279 ops/sec ±2.33% (86 runs sampled)✓
96     fast.apply() x 34,548,361 ops/sec ±2.05% (92 runs sampled)
97
98     Result: fast.js is 14.14% faster than Function::apply().
99
100    Native .apply() vs fast.apply() (3 items, with context)✓
101        Function::apply() x 24,829,234 ops/sec ±2.08% (87 runs sampled)✓
102        fast.apply() x 34,777,263 ops/sec ±1.63% (94 runs sampled)
103
104    Result: fast.js is 40.07% faster than Function::apply().
105
106    Native .apply() vs fast.apply() (6 items, no context)✓
107        Function::apply() x 30,864,112 ops/sec ±1.81% (90 runs sampled)✓
108        fast.apply() x 33,857,870 ops/sec ±2.35% (86 runs sampled)
109
110    Result: fast.js is 9.70% faster than Function::apply().
111
112    Native .apply() vs fast.apply() (6 items, with context)✓
113        Function::apply() x 25,550,568 ops/sec ±1.93% (90 runs sampled)✓
114        fast.apply() x 28,453,053 ops/sec ±2.64% (87 runs sampled)
115
116    Result: fast.js is 11.36% faster than Function::apply().
117
118    Native .apply() vs fast.apply() (10 items, no context)✓
119        Function::apply() x 25,256,277 ops/sec ±0.84% (99 runs sampled)✓
120        fast.apply() x 20,836,068 ops/sec ±1.12% (86 runs sampled)
121
122    Result: fast.js is 17.5% slower than Function::apply().
123
124    Native .apply() vs fast.apply() (10 items, with context)✓
125        Function::apply() x 21,261,822 ops/sec ±1.88% (84 runs sampled)✓
126        fast.apply() x 20,416,548 ops/sec ±2.17% (82 runs sampled)
127
128    Result: fast.js is 3.98% slower than Function::apply().
129
130    fast.clone() vs underscore.clone() vs lodash.clone()✓
131        fast.clone() x 3,473,853 ops/sec ±1.52% (87 runs sampled)✓
132        underscore.clone() x 2,652,079 ops/sec ±1.78% (88 runs sampled)✓
133        lodash.clone() x 1,102,502 ops/sec ±1.07% (93 runs sampled)
134
135    Result: fast.js is 215.09% faster than lodash.clone().
136
137    Native .indexOf() vs fast.indexOf() (3 items)✓
138        Array::indexOf() x 27,266,466 ops/sec ±0.64% (98 runs sampled)✓
139        fast.indexOf() x 64,809,105 ops/sec ±2.63% (92 runs sampled)✓
140        fast.indexOf() v0.0.2 x 50,082,737 ops/sec ±1.39% (93 runs
        sampled)✓
141        underscore.indexOf() x 16,758,164 ops/sec ±1.09% (96 runs sampled
        )✓
142        lodash.indexOf() x 23,183,843 ops/sec ±0.95% (95 runs sampled)
143

```

```

144     Result: fast.js is 137.69% faster than Array::indexOf().
145
146     Native .indexOf() vs fast.indexOf() (10 items)✓
147         Array::indexOf() x 19,572,180 ops/sec ±1.02% (93 runs sampled)✓
148         fast.indexOf() x 30,659,406 ops/sec ±1.83% (89 runs sampled)✓
149         fast.indexOf() v0.0.2 x 33,348,396 ops/sec ±2.73% (85 runs
            sampled)✓
150         underscore.indexOf() x 19,318,182 ops/sec ±1.92% (91 runs sampled
            )✓
151         lodash.indexOf() x 10,954,969 ops/sec ±0.69% (98 runs sampled)
152
153     Result: fast.js is 56.65% faster than Array::indexOf().
154
155     Native .indexOf() vs fast.indexOf() (1000 items)✓
156         Array::indexOf() x 548,163 ops/sec ±1.15% (96 runs sampled)✓
157         fast.indexOf() x 650,766 ops/sec ±1.80% (88 runs sampled)✓
158         fast.indexOf() v0.0.2 x 689,307 ops/sec ±1.00% (94 runs sampled)✓
159         underscore.indexOf() x 595,136 ops/sec ±0.65% (97 runs sampled)✓
160         lodash.indexOf() x 164,976 ops/sec ±0.79% (97 runs sampled)
161
162     Result: fast.js is 18.72% faster than Array::indexOf().
163
164     Native .lastIndexOf() vs fast.lastIndexOf() (3 items)✓
165         Array::lastIndexOf() x 24,297,112 ops/sec ±0.72% (94 runs sampled
            )✓
166         fast.lastIndexOf() x 132,052,822 ops/sec ±0.45% (97 runs sampled)
            ✓
167         fast.lastIndexOf() v0.0.2 x 133,679,760 ops/sec ±0.65% (95 runs
            sampled)✓
168         underscore.lastIndexOf() x 58,700,858 ops/sec ±4.10% (90 runs
            sampled)✓
169         lodash.lastIndexOf() x 49,037,304 ops/sec ±0.56% (99 runs sampled
            )
170
171     Result: fast.js is 443.49% faster than Array::lastIndexOf().
172
173     Native .lastIndexOf() vs fast.lastIndexOf() (10 items)✓
174         Array::lastIndexOf() x 6,224,933 ops/sec ±0.72% (92 runs sampled)
            ✓
175         fast.lastIndexOf() x 69,754,081 ops/sec ±0.58% (99 runs sampled)✓
176         fast.lastIndexOf() v0.0.2 x 64,893,167 ops/sec ±2.46% (92 runs
            sampled)✓
177         underscore.lastIndexOf() x 21,080,527 ops/sec ±4.53% (92 runs
            sampled)✓
178         lodash.lastIndexOf() x 17,194,132 ops/sec ±0.55% (92 runs sampled
            )
179
180     Result: fast.js is 1020.56% faster than Array::lastIndexOf().
181
182     Native .lastIndexOf() vs fast.lastIndexOf() (1000 items)✓
183         Array::lastIndexOf() x 128,940 ops/sec ±0.70% (99 runs sampled)✓

```

```

184     fast.lastIndexOf() x 1,409,601 ops/sec ±0.40% (99 runs sampled)✓
185     fast.lastIndexOf() v0.0.2 x 1,265,606 ops/sec ±1.51% (87 runs
      sampled)✓
186     underscore.lastIndexOf() x 1,147,635 ops/sec ±0.46% (97 runs
      sampled)✓
187     lodash.lastIndexOf() x 346,839 ops/sec ±0.89% (95 runs sampled)
188
189     Result: fast.js is 993.22% faster than Array::lastIndexOf().
190
191     Native .bind() vs fast.bind()✓
192     Function::bind() x 36,503,967 ops/sec ±3.38% (78 runs sampled)✓
193     fast.bind() x 10,071,557 ops/sec ±1.79% (84 runs sampled)✓
194     fast.bind() v0.0.2 x 7,823,170 ops/sec ±1.66% (82 runs sampled)✓
195     underscore.bind() x 2,411,650 ops/sec ±1.70% (89 runs sampled)✓
196     lodash.bind() x 465,496 ops/sec ±1.96% (84 runs sampled)
197
198     Result: fast.js is 72.41% slower than Function::bind().
199
200     Native .bind() vs fast.bind() with prebound functions✓
201     Function::bind() x 51,404,179 ops/sec ±2.18% (80 runs sampled)✓
202     fast.bind() x 33,334,432 ops/sec ±3.74% (83 runs sampled)✓
203     fast.bind() v0.0.2 x 22,070,212 ops/sec ±1.50% (93 runs sampled)✓
204     underscore.bind() x 3,776,775 ops/sec ±1.31% (92 runs sampled)✓
205     lodash.bind() x 27,288,182 ops/sec ±1.77% (83 runs sampled)
206
207     Result: fast.js is 35.15% slower than Function::bind().
208
209     Native .bind() vs fast.partial()✓
210     Function::bind() x 52,693,468 ops/sec ±2.22% (82 runs sampled)✓
211     fast.partial() x 12,231,235 ops/sec ±1.95% (89 runs sampled)✓
212     fast.partial() v0.0.2 x 8,803,944 ops/sec ±1.68% (89 runs sampled
      )✓
213     fast.partial() v0.0.0 x 9,035,529 ops/sec ±1.82% (86 runs sampled
      )✓
214     underscore.partial() x 4,463,215 ops/sec ±1.33% (92 runs sampled)
      ✓
215     lodash.partial() x 639,048 ops/sec ±1.76% (86 runs sampled)
216
217     Result: fast.js is 76.79% slower than Function::bind().
218
219     Native .bind() vs fast.partial() with prebound functions✓
220     Function::bind() x 59,131,567 ops/sec ±1.36% (95 runs sampled)✓
221     fast.partial() x 32,818,864 ops/sec ±1.59% (94 runs sampled)✓
222     fast.partial() v0.0.2 x 23,085,249 ops/sec ±2.39% (86 runs
      sampled)✓
223     fast.partial() v0.0.0 x 22,930,109 ops/sec ±1.71% (89 runs
      sampled)✓
224     underscore.partial() x 7,487,595 ops/sec ±1.76% (90 runs sampled)
      ✓
225     lodash.partial() x 31,324,568 ops/sec ±1.54% (94 runs sampled)
226

```



```

227     Result: fast.js is 44.5% slower than Function::bind().
228
229     Native .map() vs fast.map() (3 items)✓
230     Array::map() x 5,640,525 ops/sec ±1.74% (90 runs sampled)✓
231     fast.map() x 10,660,733 ops/sec ±2.27% (87 runs sampled)✓
232     fast.map() v0.0.2a x 9,206,635 ops/sec ±2.15% (85 runs sampled)✓
233     fast.map() v0.0.1 x 10,324,826 ops/sec ±2.49% (85 runs sampled)✓
234     fast.map() v0.0.0 x 10,642,222 ops/sec ±2.35% (89 runs sampled)✓
235     underscore.map() x 9,735,259 ops/sec ±1.75% (93 runs sampled)✓
236     lodash.map() x 5,465,553 ops/sec ±1.33% (92 runs sampled)
237
238     Result: fast.js is 89.00% faster than Array::map().
239
240     Native .map() vs fast.map() (10 items)✓
241     Array::map() x 2,833,498 ops/sec ±1.47% (93 runs sampled)✓
242     fast.map() x 3,688,187 ops/sec ±1.37% (95 runs sampled)✓
243     fast.map() v0.0.2a x 3,637,287 ops/sec ±1.53% (93 runs sampled)✓
244     fast.map() v0.0.1 x 3,779,077 ops/sec ±0.97% (96 runs sampled)✓
245     fast.map() v0.0.0 x 3,867,612 ops/sec ±1.41% (92 runs sampled)✓
246     underscore.map() x 3,076,947 ops/sec ±0.50% (99 runs sampled)✓
247     lodash.map() x 2,763,458 ops/sec ±0.92% (97 runs sampled)
248
249     Result: fast.js is 30.16% faster than Array::map().
250
251     Native .map() vs fast.map() (1000 items)✓
252     Array::map() x 39,835 ops/sec ±1.35% (91 runs sampled)✓
253     fast.map() x 34,221 ops/sec ±1.68% (83 runs sampled)✓
254     fast.map() v0.0.2a x 34,869 ops/sec ±1.67% (83 runs sampled)✓
255     fast.map() v0.0.1 x 38,140 ops/sec ±1.77% (87 runs sampled)✓
256     fast.map() v0.0.0 x 38,223 ops/sec ±2.53% (84 runs sampled)✓
257     underscore.map() x 39,373 ops/sec ±2.07% (87 runs sampled)✓
258     lodash.map() x 35,883 ops/sec ±1.79% (93 runs sampled)
259
260     Result: fast.js is 14.09% slower than Array::map().
261
262     Native .filter() vs fast.filter() (3 items)✓
263     Array::filter() x 6,866,257 ops/sec ±2.81% (82 runs sampled)✓
264     fast.filter() x 6,765,856 ops/sec ±2.56% (82 runs sampled)✓
265     underscore.filter() x 4,905,032 ops/sec ±2.59% (83 runs sampled)✓
266     lodash.filter() x 6,960,451 ops/sec ±2.33% (85 runs sampled)
267
268     Result: fast.js is 1.46% slower than Array::filter().
269
270     Native .filter() vs fast.filter() (10 items)✓
271     Array::filter() x 2,774,149 ops/sec ±2.11% (89 runs sampled)✓
272     fast.filter() x 2,818,009 ops/sec ±1.50% (96 runs sampled)✓
273     underscore.filter() x 2,130,756 ops/sec ±1.72% (96 runs sampled)✓
274     lodash.filter() x 2,585,908 ops/sec ±1.98% (90 runs sampled)
275
276     Result: fast.js is 1.58% faster than Array::filter().
277

```

```

278 Native .filter() vs fast.filter() (1000 items)✓
279   Array::filter() x 21,596 ops/sec ±2.48% (82 runs sampled)✓
280   fast.filter() x 22,255 ops/sec ±2.16% (81 runs sampled)✓
281   underscore.filter() x 20,475 ops/sec ±2.21% (87 runs sampled)✓
282   lodash.filter() x 23,728 ops/sec ±2.09% (90 runs sampled)
283
284   Result: fast.js is 3.05% faster than Array::filter().
285
286 Native .reduce() vs fast.reduce() (3 items)✓
287   Array::reduce() x 12,380,157 ops/sec ±1.75% (89 runs sampled)✓
288   fast.reduce() x 12,730,241 ops/sec ±1.08% (95 runs sampled)✓
289   fast.reduce() v0.0.2c x 8,286,933 ops/sec ±1.65% (92 runs sampled)
290     )✓
291   fast.reduce() v0.0.2b x 12,045,164 ops/sec ±1.40% (90 runs
292     sampled)✓
293   fast.reduce() v0.0.2a x 11,590,488 ops/sec ±2.25% (92 runs
294     sampled)✓
295   fast.reduce() v0.0.1 x 12,495,314 ops/sec ±2.24% (86 runs sampled)
296     )✓
297   fast.reduce() v0.0.0 x 11,694,271 ops/sec ±0.87% (95 runs sampled)
298     )✓
299   underscore.reduce() x 11,522,220 ops/sec ±1.07% (97 runs sampled)
300     ✓
301   lodash.reduce() x 11,100,158 ops/sec ±2.12% (88 runs sampled)
302
303   Result: fast.js is 2.83% faster than Array::reduce().
304
305 Native .reduce() vs fast.reduce() (10 items)✓
306   Array::reduce() x 3,871,590 ops/sec ±1.79% (89 runs sampled)✓
307   fast.reduce() x 3,941,594 ops/sec ±1.69% (93 runs sampled)✓
308   fast.reduce() v0.0.2c x 2,998,224 ops/sec ±1.61% (92 runs sampled)
309     )✓
310   fast.reduce() v0.0.2b x 4,124,821 ops/sec ±1.04% (92 runs sampled)
311     )✓
312   fast.reduce() v0.0.2a x 3,826,263 ops/sec ±2.27% (89 runs sampled)
313     )✓
314   fast.reduce() v0.0.1 x 3,983,053 ops/sec ±1.42% (92 runs sampled)
315     ✓
316   fast.reduce() v0.0.0 x 3,963,823 ops/sec ±1.35% (92 runs sampled)
317     ✓
318   underscore.reduce() x 3,560,315 ops/sec ±1.89% (86 runs sampled)✓
319   lodash.reduce() x 3,897,774 ops/sec ±1.80% (91 runs sampled)
320
321   Result: fast.js is 1.81% faster than Array::reduce().
322
323 Native .reduce() vs fast.reduce() (1000 items)✓
324   Array::reduce() x 44,142 ops/sec ±2.52% (90 runs sampled)✓
325   fast.reduce() x 38,836 ops/sec ±1.46% (86 runs sampled)✓
326   fast.reduce() v0.0.2c x 33,523 ops/sec ±1.59% (90 runs sampled)✓
327   fast.reduce() v0.0.2b x 44,667 ops/sec ±0.90% (97 runs sampled)✓
328   fast.reduce() v0.0.2a x 44,036 ops/sec ±1.30% (94 runs sampled)✓

```

```

318     fast.reduce() v0.0.1 x 38,857 ops/sec ±2.31% (87 runs sampled)✓
319     fast.reduce() v0.0.0 x 41,001 ops/sec ±2.68% (88 runs sampled)✓
320     underscore.reduce() x 44,323 ops/sec ±0.86% (97 runs sampled)✓
321     lodash.reduce() x 44,380 ops/sec ±1.60% (92 runs sampled)
322
323     Result: fast.js is 12.02% slower than Array::reduce().
324
325     Native .reduceRight() vs fast.reduceRight() (3 items)✓
326     Array::reduceRight() x 12,477,745 ops/sec ±2.00% (91 runs sampled)
327         ✓
328     fast.reduceRight() x 12,403,528 ops/sec ±2.00% (85 runs sampled)✓
329     underscore.reduceRight() x 9,818,641 ops/sec ±2.07% (93 runs
330         sampled)✓
331     lodash.reduceRight() x 11,421,199 ops/sec ±2.51% (88 runs sampled)
332
333     Result: fast.js is 0.59% slower than Array::reduceRight().
334
335     Native .reduceRight() vs fast.reduceRight() (10 items)✓
336     Array::reduceRight() x 4,166,641 ops/sec ±0.82% (97 runs sampled)
337         ✓
338     fast.reduceRight() x 4,099,997 ops/sec ±1.78% (86 runs sampled)✓
339     underscore.reduceRight() x 3,385,772 ops/sec ±2.29% (87 runs
340         sampled)✓
341     lodash.reduceRight() x 3,950,796 ops/sec ±1.40% (93 runs sampled)
342
343     Result: fast.js is 1.6% slower than Array::reduceRight().
344
345     Native .reduceRight() vs fast.reduceRight() (1000 items)✓
346     Array::reduceRight() x 36,287 ops/sec ±10.30% (87 runs sampled)✓
347     fast.reduceRight() x 43,700 ops/sec ±1.92% (94 runs sampled)✓
348     underscore.reduceRight() x 41,119 ops/sec ±2.14% (89 runs sampled)
349         ✓
350     lodash.reduceRight() x 38,804 ops/sec ±2.51% (85 runs sampled)
351
352     Result: fast.js is 20.43% faster than Array::reduceRight().
353
354     Native .forEach() vs fast.forEach() (3 items)✓
355     Array::forEach() x 24,961,078 ops/sec ±1.99% (86 runs sampled)✓
356     fast.forEach() x 24,771,711 ops/sec ±1.38% (92 runs sampled)✓
357     fast.forEach() v0.0.2a x 21,779,106 ops/sec ±1.38% (92 runs
358         sampled)✓
359     fast.forEach() v0.0.1 x 21,920,682 ops/sec ±1.22% (95 runs
360         sampled)✓
361     fast.forEach() v0.0.0 x 24,759,663 ops/sec ±1.49% (91 runs
362         sampled)✓
363     underscore.forEach() x 20,265,586 ops/sec ±1.25% (94 runs sampled)
364         ✓
365     lodash.forEach() x 24,858,012 ops/sec ±0.50% (99 runs sampled)
366
367     Result: fast.js is 0.76% slower than Array::forEach().

```

```

359
360 Native .forEach() vs fast.forEach() (10 items)✓
361   Array::forEach() x 8,557,082 ops/sec ±0.37% (97 runs sampled)✓
362   fast.forEach() x 8,799,272 ops/sec ±0.41% (97 runs sampled)✓
363   fast.forEach() v0.0.2a x 7,268,647 ops/sec ±0.65% (97 runs
      sampled)✓
364   fast.forEach() v0.0.1 x 7,949,741 ops/sec ±0.67% (92 runs sampled
      )✓
365   fast.forEach() v0.0.0 x 8,493,600 ops/sec ±0.66% (94 runs sampled
      )✓
366   underscore.forEach() x 7,375,416 ops/sec ±0.52% (94 runs sampled)
      ✓
367   lodash.forEach() x 8,606,963 ops/sec ±0.47% (99 runs sampled)
368
369   Result: fast.js is 2.83% faster than Array::forEach().
370
371 Native .forEach() vs fast.forEach() (1000 items)✓
372   Array::forEach() x 103,201 ops/sec ±0.24% (100 runs sampled)✓
373   fast.forEach() x 102,629 ops/sec ±0.34% (102 runs sampled)✓
374   fast.forEach() v0.0.2a x 94,386 ops/sec ±0.64% (94 runs sampled)✓
375   fast.forEach() v0.0.1 x 95,651 ops/sec ±0.37% (97 runs sampled)✓
376   fast.forEach() v0.0.0 x 99,667 ops/sec ±0.49% (100 runs sampled)✓
377   underscore.forEach() x 102,414 ops/sec ±0.26% (99 runs sampled)✓
378   lodash.forEach() x 102,077 ops/sec ±0.35% (96 runs sampled)
379
380   Result: fast.js is 0.55% slower than Array::forEach().
381
382 Native .some() vs fast.some() (3 items)✓
383   Array::some() x 23,673,539 ops/sec ±1.00% (93 runs sampled)✓
384   fast.some() x 24,480,193 ops/sec ±1.25% (96 runs sampled)✓
385   underscore.some() x 16,143,239 ops/sec ±2.63% (91 runs sampled)✓
386   lodash.some() x 24,973,245 ops/sec ±0.68% (97 runs sampled)
387
388   Result: fast.js is 3.41% faster than Array::some().
389
390 Native .some() vs fast.some() (10 items)✓
391   Array::some() x 8,275,557 ops/sec ±0.89% (94 runs sampled)✓
392   fast.some() x 8,589,278 ops/sec ±0.46% (95 runs sampled)✓
393   underscore.some() x 7,637,286 ops/sec ±0.43% (101 runs sampled)✓
394   lodash.some() x 8,565,231 ops/sec ±0.36% (97 runs sampled)
395
396   Result: fast.js is 3.79% faster than Array::some().
397
398 Native .some() vs fast.some() (1000 items)✓
399   Array::some() x 97,934 ops/sec ±1.11% (98 runs sampled)✓
400   fast.some() x 102,638 ops/sec ±0.71% (95 runs sampled)✓
401   underscore.some() x 102,123 ops/sec ±1.03% (95 runs sampled)✓
402   lodash.some() x 102,114 ops/sec ±0.57% (97 runs sampled)
403
404   Result: fast.js is 4.80% faster than Array::some().
405

```

```
406 Native .every() vs fast.every() (3 items)✓
407   Array::every() x 22,865,101 ops/sec ±0.47% (98 runs sampled)✓
408   fast.every() x 26,275,582 ops/sec ±0.97% (94 runs sampled)✓
409   underscore.every() x 19,320,783 ops/sec ±0.52% (97 runs sampled)✓
410   lodash.every() x 24,804,998 ops/sec ±0.84% (94 runs sampled)
411
412   Result: fast.js is 14.92% faster than Array::every().
413
414 Native .every() vs fast.every() (10 items)✓
415   Array::every() x 7,955,651 ops/sec ±0.76% (98 runs sampled)✓
416   fast.every() x 8,672,263 ops/sec ±1.16% (94 runs sampled)✓
417   underscore.every() x 7,466,047 ops/sec ±0.74% (98 runs sampled)✓
418   lodash.every() x 8,249,248 ops/sec ±1.80% (95 runs sampled)
419
420   Result: fast.js is 9.01% faster than Array::every().
421
422 Native .every() vs fast.every() (1000 items)✓
423   Array::every() x 94,830 ops/sec ±0.51% (96 runs sampled)✓
424   fast.every() x 98,676 ops/sec ±1.33% (94 runs sampled)✓
425   underscore.every() x 100,531 ops/sec ±0.27% (97 runs sampled)✓
426   lodash.every() x 98,853 ops/sec ±1.07% (94 runs sampled)
427
428   Result: fast.js is 4.06% faster than Array::every().
429
430 Native .concat() vs fast.concat() (3 items)✓
431   Array::concat() x 2,408,488 ops/sec ±1.49% (92 runs sampled)✓
432   fast.concat() x 15,798,707 ops/sec ±1.59% (93 runs sampled)
433
434   Result: fast.js is 555.96% faster than Array::concat().
435
436 Native .concat() vs fast.concat() (10 items)✓
437   Array::concat() x 1,912,184 ops/sec ±0.66% (98 runs sampled)✓
438   fast.concat() x 6,977,148 ops/sec ±1.35% (90 runs sampled)
439
440   Result: fast.js is 264.88% faster than Array::concat().
441
442 Native .concat() vs fast.concat() (1000 items)✓
443   Array::concat() x 427,875 ops/sec ±1.44% (94 runs sampled)✓
444   fast.concat() x 119,531 ops/sec ±1.10% (93 runs sampled)
445
446   Result: fast.js is 72.06% slower than Array::concat().
447
448 Native .concat() vs fast.concat() (1000 items, using apply)✓
449   Array::concat() x 88,438 ops/sec ±0.98% (94 runs sampled)✓
450   fast.concat() x 105,193 ops/sec ±1.56% (90 runs sampled)
451
452   Result: fast.js is 18.95% faster than Array::concat().
453
454 Finished in 1352 seconds
```

Credits

Fast.js is written by codemix, a small team of expert software developers who specialise in writing very fast web applications. This particular library is heavily inspired by the excellent work done by Petka Antonov, especially the superb bluebird Promise library.

License

MIT, see LICENSE.md.