Function helpers

J. S. Choi
Indiana University
2021-10-25
GitHub repository

There are useful, common helper functions are defined a lot, used a lot, downloaded a lot.
We should standardize at least some of them.

This proposal is seeking Committee consensus for Stage 1: that standardizing at least some Function helpers is "worth investigating".

It is not seeking to standardize every imaginable helper function—just some selected frequently used functions.

Choosing which functions to standardize would be bikeshedding for Stage 2.

Why standardize helper functions

High frequency, universal usefulness: Each function is a frequently trodden cowpath and is frequently re-implemented.
Every developer needs to manipulate callbacks.

Developer ergonomics: When a helper function is standardized, we can readily use it in REPL or script, instead of downloading an external package or pasting a definition.

Code clarity: Standardization gives one standard name to each helper function, rather than various names from various libraries that refer to the same thing. Even for simple functions, a standard name may often be clearer than an inline declaration, e.g., `identity` vs. `x => x`.

Unlike new syntax, standardized helper functions are relatively lightweight ways to improve the experience of all developers.

These helper functions are well-trodden cowpaths, each of which may deserve standardization.

The following functions are only **possibilities**. Choosing **which** functions to standardize in **this** proposal would be bikeshedding for before Stage **2**.

flow flowAsync pipe pipeAsync
constant identity noop
once debounce throttle
aside unThis

Possibility: flow and flowAsync

The Function.flow static method would create a new function by composing several callbacks. Composition flows LTR: leftmost callback is called first, and rightmost callback is called last.

If Function.flow receives no arguments, then, by default, it will return Function.identity.

Function.flowAsync would be the same as flow, except it would work on async functions.

The name "flow" comes from lodash.flow. (The name `compose` would be confusing with other languages' RTL function composition.) The following real-world examples originally used lodash.flow.

```
Function.flow(...fns)
Function.flowAsync(...fns)

const { flow } = Function;
const f = flow(f0, f1, f2);
f(5, 7); // f2(f1(f0(5, 7)))
```

From strapi@3.6.8:

```
const transform = flow(
  flattenDeep,
  cleanupUnwantedProperties
);
```

From semantic-ui-react@v2.0.4:

```
const getInfoForSeeTags = flow(
  _.get('docblock.tags'),
  _.filter(tag => tag.title === 'see'),
  _.map(tag => { /* … */ }),
);
```

Possibility: pipe and pipeAsync

The Function.pipe static method would apply several callbacks to an initial input value. Application goes LTR: leftmost callback is called first, and rightmost callback is called last.

Function.pipeAsync would be the same except it would work on async functions.

`pipe(x, f0, f1)` would be equivalent to `flow(f0, f1)(x)`.

`pipeAsync(x, f0, f1)` would be equivalent to `flowAsync(f0, f1)(Promise.resolve(x))`.

The following real-world example originally used fp-ts's pipe.

```
Function.pipe(input, ...fns)
Function.pipeAsync(input, ...fns)
```

```
const { pipe } = Function;
pipe(x, f0, f1, f2); // f2(f1(f0(5)))
```

From @gripeless/pico@1.0.1:

```
return pipe(
  download(absoluteURL),
  mapRej(downloadErrorToDetailedError),
  chainFluture(responseToBlob),
  chainFluture(blobToDataURL),
  mapFluture(dataURL => `url(${dataURL})`)
);
```

There has been a lot of community feedback from developers who have desired standardized unary-function application and are unhappy with the pipe operator |>'s topic syntax. Standard pipe functions would help ameliorate their concerns.

Possibility: constant and identity

The Function.constant static method would create a new function that always returns a given constant value.

The Function.identity static method would always return its first argument.

`constant(x)` is equivalent to `() => x`.
`identity` is equivalent to `x => x`.

```
const { constant, identity } = Function;
[ 0, 1 ].map(constant(5)) // [ 5, 5 ]
[ 0, 1 ].map(identity) // [ 0, 1 ]
```

The following real-world examples originally used lodash.constant and lodash.identity.

From Cypress v8.6.0:

```
setDefaultHeader(
  'access-control-expose-headers',
  constant('*'))
```

```
function findTestInSuite (suite, fn = identity) {
  for (const test of suite.tests) {
    if (fn(test)) { return test; } } }
```

From Odoo v15.0:

```
url.toJSON = constant(this.url);
```

From Meteor v2.5.0:

```
const boilerplate = new Boilerplate(
  CORDOVA_ARCH, manifest, {
    urlMapper: identity, /* … */
  });
```

Possibility: noop

The Function.noop static method would always return undefined.

noop is equivalent to `() => {}`.

```
const { noop } = Function;
[ 0, 1 ].map(noop)
// [ undefined, undefined ]
```

This function is already available and frequently used both from jQuery and from Lodash, generally to fill a required callback argument or to disable a callback property.

The following real-world examples originally used jQuery's $.noop or lodash.noop.

From Twitter Typeahead.js v0.11.1:

```
this.cancel = function cancel() {
  canceled = true;
  that.cancel = noop;
  that.async &&
    that.trigger('asyncCanceled', query);
};
```

From Three.js 0.133.1:

```
SuiteUI.prototype.run = function() {
  this.runButton.click = noop;
  this.runButton.innerText = "Running..."
  this.suite.run({ async: true });
}
```

From Wordpress v5.1.11:

```
{ /* … */
  defaultExpandedArguments: {
    duration: 'fast',
    completeCallback: noop }
  /* … */ }
```

Possibility: once

The Function.prototype.once method would create a new function that calls the original function at most once, no matter how much the new function is called.

```
const initialize = createApplication.once();
initialize();
initialize();
// createApplication is invoked only once.
```

lodash.once is one of the most popular Lodash functions. jQuery also has a similar .one method. Many other APIs have similar methods, such as Node's events API.

The following real-world examples originally used lodash.once.

From Meteor v2.2.1:

```
// "Are we running Meteor from a git checkout?"
export const inCheckout = (function () {
  try { /* … */ } catch (e) { console.log(e); }
  return false;
}).once();
```

From cypress@8.6.0:

```
cy.on('command:retry', _.after(2, (() => {
  button.remove() /* … */
}).once()))
```

From Jitsi Meet v6482:

```
this._hangup = (() => {
 sendAnalytics(createToolbarEvent('hangup'));
 /* … */
}).once()
```

Possibility: debounce and throttle

The Function.prototype.debounce method would create a new function that, when called, calls the original function—but only after a given length of time has elapsed since the last time the new function was called.

The Function.prototype.throttle method would create a new function that, when called, calls the original function—but only at most once within a given length of time.

These two methods may come with options that could be bikeshedded in Stage 1.

There are multiple articles (e.g., from CSS Tricks and from Ben Alman) that further explain the difference between debounce and throttle and why both are useful.

In this example, logging happens on keyup events from `inputEl`, but only after the user has stopped typing for at least 250 ms:

```
inputEl.addEventListener('keyup',
  console.log.debounce(250));
```

In this example, logging happens on window scroll, but no more than once every 250 ms:

```
window.addEventListener(scroll',
  console.log.throttle(250));
```

lodash.debounce and lodash.throttle are also two of the most popular Lodash functions; they are also popular non-Lodash packages (debounce, throttle-debounce, etc.). They are useful in virtually every end-user-facing graphical application.

Possibility: aside

The Function.prototype.aside method would a new function that calls the original function but always returns the first argument given to the new function.

`fn.aside()` would be equivalent to `x => { fn(x); return x; }`.

This would be useful for debugging with `console.log` or `debugger`, as well as performing other side effects.

Libraries or languages also call similar functions "tap", "trace", or "peek". The following real-world example originally used lodash.tap.

```
g(console.log.aside(f(input)))
```

```
const data = await
Promise.resolve('intro.txt')
  .then(Deno.open)
  .then(Deno.readAll)
  .then(console.log.aside())
  .then(data => new
TextDecoder('utf-8').decode(data));
```

From IBM/report-toolkit v0.6.1.

```
function filterEnabledRules(config) {
  return pipe(
    config,
    /* … */
    (ruleIds => {
      debug(/* … */);
    }).aside(); }
```

Possibility: unThis

The Function.prototype.unThis method would create a new function that calls the original function, supplying its first argument as the original function's `this` receiver, and supplying the rest of its arguments as the original function's ordinary arguments.

`fn.unThis()` would be equivalent to `fn.call.bind(fn)`.

This would be useful for converting functions that rely on the dynamic `this` binding into functions that only use their arguments.

```
const $slice =
Array.prototype.slice.unThis();
$slice([ 0, 1, 2 ], 1); // [ 1, 2 ]
```

The following real-world examples originally used the bind-this library or a similar function.

From chrome-devtools-frontend:

```
runTests(implementation.unThis(), t);
```

From string.prototype.trimstart:

```
var bound = getPolyfill().unThis();
```

From andreasgal/dom.js:

```
const /* … */
  join = A.join || Array.prototype.join.unThis(),
  map = A.map || Array.prototype.map.unThis(),
  /* … */;
```

This function would not be a substitute for a bind-this syntax, which would allow developers to change the receiver of functions without creating a wrapper function.