

---

## memoize

Memoize functions - An optimization used to speed up consecutive function calls by caching the result of calls with identical input

Memory is automatically released when an item expires or the cache is cleared.

By default, **only the memoized function's first argument is considered** via strict equality comparison. If you need to cache multiple arguments or cache *objects by value*, have a look at alternative caching strategies below.

If you want to memoize Promise-returning functions (like *async* functions), you might be better served by p-memoize.

### Install

```
1 npm install memoize
```

### Usage

```
1 import memoize from 'memoize';
2
3 let index = 0;
4 const counter = () => ++index;
5 const memoized = memoize(counter);
6
7 memoized('foo');
8 //=> 1
9
10 // Cached as it's the same argument
11 memoized('foo');
12 //=> 1
13
14 // Not cached anymore as the argument changed
15 memoized('bar');
16 //=> 2
17
18 memoized('bar');
19 //=> 2
20
21 // Only the first argument is considered by default
22 memoized('bar', 'foo');
23 //=> 2
```

---

**Works well with Promise-returning functions** But you might want to use p-memoize for more Promise-specific behaviors.

```
1 import memoize from 'memoize';
2
3 let index = 0;
4 const counter = async () => ++index;
5 const memoized = memoize(counter);
6
7 console.log(await memoized());
8 //=> 1
9
10 // The return value didn't increase as it's cached
11 console.log(await memoized());
12 //=> 1
```

```
1 import memoize from 'memoize';
2 import got from 'got';
3 import delay from 'delay';
4
5 const memoizedGot = memoize(got, {maxAge: 1000});
6
7 await memoizedGot('https://sindresorhus.com');
8
9 // This call is cached
10 await memoizedGot('https://sindresorhus.com');
11
12 await delay(2000);
13
14 // This call is not cached as the cache has expired
15 await memoizedGot('https://sindresorhus.com');
```

## Caching strategy

By default, only the first argument is compared via exact equality (===) to determine whether a call is identical.

```
1 import memoize from 'memoize';
2
3 const pow = memoize((a, b) => Math.pow(a, b));
4
5 pow(2, 2); // => 4, stored in cache with the key 2 (number)
6 pow(2, 3); // => 4, retrieved from cache at key 2 (number), it's wrong
```

You will have to use the `cache` and `cacheKey` options appropriate to your function. In this specific case, the following could work:

```
1 import memoize from 'memoize';
```

---

```
2
3 const pow = memoize((a, b) => Math.pow(a, b), {
4   cacheKey: arguments_ => arguments_.join(',')
5 });
6
7 pow(2, 2); // => 4, stored in cache with the key '2,2' (both arguments
   as one string)
8 pow(2, 3); // => 8, stored in cache with the key '2,3'
```

More advanced examples follow.

**Example: Options-like argument** If your function accepts an object, it won't be memoized out of the box:

```
1 import memoize from 'memoize';
2
3 const heavyMemoizedOperation = memoize(heavyOperation);
4
5 heavyMemoizedOperation({full: true}); // Stored in cache with the
   object as key
6 heavyMemoizedOperation({full: true}); // Stored in cache with the
   object as key, again
7 // The objects appear the same, but in JavaScript, they're different
   objects
```

You might want to serialize or hash them, for example using `JSON.stringify` or something like `serialize-javascript`, which can also serialize `RegExp`, `Date` and so on.

```
1 import memoize from 'memoize';
2
3 const heavyMemoizedOperation = memoize(heavyOperation, {cacheKey: JSON.
   stringify});
4
5 heavyMemoizedOperation({full: true}); // Stored in cache with the key
   '["full":true]' (string)
6 heavyMemoizedOperation({full: true}); // Retrieved from cache
```

The same solution also works if it accepts multiple serializable objects:

```
1 import memoize from 'memoize';
2
3 const heavyMemoizedOperation = memoize(heavyOperation, {cacheKey: JSON.
   stringify});
4
5 heavyMemoizedOperation('hello', {full: true}); // Stored in cache with
   the key '["hello",{"full":true}]' (string)
6 heavyMemoizedOperation('hello', {full: true}); // Retrieved from cache
```

---

**Example: Multiple non-serializable arguments** If your function accepts multiple arguments that aren't supported by `JSON.stringify` (e.g. DOM elements and functions), you can instead extend the initial exact equality (`===`) to work on multiple arguments using `many-keys-map`:

```
1 import memoize from 'memoize';
2 import ManyKeysMap from 'many-keys-map';
3
4 const addListener = (emitter, eventName, listener) => emitter.on(
  eventName, listener);
5
6 const addOneListener = memoize(addListener, {
7   cacheKey: arguments_ => arguments_, // Use *all* the arguments as
    key
8   cache: new ManyKeysMap() // Correctly handles all the arguments for
    exact equality
9 });
10
11 addOneListener(header, 'click', console.log); // `addListener` is run,
    and it's cached with the `arguments` array as key
12 addOneListener(header, 'click', console.log); // `addListener` is not
    run again because the arguments are the same
13 addOneListener(mainContent, 'load', console.log); // `addListener` is
    run, and it's cached with the `arguments` array as key
```

Better yet, if your function's arguments are compatible with `WeakMap`, you should use `deep-weak-map` instead of `many-keys-map`. This will help avoid memory leaks.

## API

### **memoize(fn, options?)**

**fn** Type: `Function`

The function to be memoized.

**options** Type: `object`

**maxAge** Type: `number`

Default: `Infinity`

Milliseconds until the cache entry expires.

---

**cacheKey** Type: `Function`

Default: `arguments_ => arguments_[0]`

Example: `arguments_ => JSON.stringify(arguments_)`

Determines the cache key for storing the result based on the function arguments. By default, **only the first argument is considered**.

A `cacheKey` function can return any type supported by `Map` (or whatever structure you use in the `cache` option).

Refer to the caching strategies section for more information.

**cache** Type: `object`

Default: `new Map()`

Use a different cache storage. Must implement the following methods: `.has(key)`, `.get(key)`, `.set(key, value)`, `.delete(key)`, and optionally `.clear()`. You could for example use a `WeakMap` instead or `quick-lru` for a LRU cache.

Refer to the caching strategies section for more information.

### **memoizeDecorator(options)**

Returns a decorator to memoize class methods or static class methods.

Notes:

- Only class methods and getters/setters can be memoized, not regular functions (they aren't part of the proposal);
- Only TypeScript's decorators are supported, not Babel's, which use a different version of the proposal;
- Being an experimental feature, they need to be enabled with `--experimentalDecorators`; follow TypeScript's docs.

**options** Type: `object`

Same as options for `memoize()`.

```
1 import {memoizeDecorator} from 'memoize';
2
3 class Example {
4   index = 0
5
6   @memoizeDecorator()
```

---

```
7     counter() {
8         return ++this.index;
9     }
10 }
11
12 class ExampleWithOptions {
13     index = 0
14
15     @memoizeDecorator({maxAge: 1000})
16     counter() {
17         return ++this.index;
18     }
19 }
```

### memoizeClear(fn)

Clear all cached data of a memoized function.

**fn** Type: `Function`

The memoized function.

### Tips

#### Cache statistics

If you want to know how many times your cache had a hit or a miss, you can make use of stats-map as a replacement for the default cache.

#### Example

```
1 import memoize from 'memoize';
2 import StatsMap from 'stats-map';
3 import got from 'got';
4
5 const cache = new StatsMap();
6 const memoizedGot = memoize(got, {cache});
7
8 await memoizedGot('https://sindresorhus.com');
9 await memoizedGot('https://sindresorhus.com');
10 await memoizedGot('https://sindresorhus.com');
11
12 console.log(cache.stats);
13 //=> {hits: 2, misses: 1}
```

---

## Related

- [p-memoize](#) - Memoize promise-returning & async functions