Lodash

What is Lodash & why should I use it?

J.P. Strydom

Entelect

Cell C

March 26, 2018

Outline

What Is Lodash?

What Is Lodash?

Lodash, or "_" for short, is a JavaScript helper library for arrays, strings, and objects. It provides a ton of helper functions to help you save time.

Documentation here: `https://lodash.com/docs`

Why Use Lodash?

Why Use Lodash?

- It has over 400 useful helper functions.

Why Use Lodash?

- It has over 400 useful helper functions.
- It leads to clean, easy-to-use, and easy-to-read code.

Why Use Lodash?

- It has over 400 useful helper functions.
- It leads to clean, easy-to-use, and easy-to-read code.
- It automatically handles multiple variable types for you.

Why Use Lodash?

- It has over 400 useful helper functions.
- It leads to clean, easy-to-use, and easy-to-read code.
- It automatically handles multiple variable types for you.
  - `arrays`
  - `strings`
  - `objects`

Why Use Lodash?

- It has over 400 useful helper functions.
- It leads to clean, easy-to-use, and easy-to-read code.
- It automatically handles multiple variable types for you.
  - `arrays`
  - `strings`
  - `objects`
- It's well-documented, well-supported, and well-loved.

Why Use Lodash?

- It has over 400 useful helper functions.
- It leads to clean, easy-to-use, and easy-to-read code.
- It automatically handles multiple variable types for you.
  - `arrays`
  - `strings`
  - `objects`
- It's well-documented, well-supported, and well-loved.
  - 0 issues on GitHub.

Why Use Lodash?

- It has over 400 useful helper functions.
- It leads to clean, easy-to-use, and easy-to-read code.
- It automatically handles multiple variable types for you.
  - `arrays`
  - `strings`
  - `objects`
- It's well-documented, well-supported, and well-loved.
  - 0 issues on GitHub.
  - 30,000+ stars on GitHub.

Why Use Lodash?

- It has over 400 useful helper functions.
- It leads to clean, easy-to-use, and easy-to-read code.
- It automatically handles multiple variable types for you.
  - `arrays`
  - `strings`
  - `objects`
- It's well-documented, well-supported, and well-loved.
  - 0 issues on GitHub.
  - 30,000+ stars on GitHub.
  - 64,000+ dependants on NPM.

Why Use Lodash?

- It has over 400 useful helper functions.
- It leads to clean, easy-to-use, and easy-to-read code.
- It automatically handles multiple variable types for you.
  - `arrays`
  - `strings`
  - `objects`
- It's well-documented, well-supported, and well-loved.
  - 0 issues on GitHub.
  - 30,000+ stars on GitHub.
  - 64,000+ dependants on NPM.
- It's thoroughly tested across multiple platforms and browsers.

Why Use Lodash?

- It has over 400 useful helper functions.
- It leads to clean, easy-to-use, and easy-to-read code.
- It automatically handles multiple variable types for you.
  - `arrays`
  - `strings`
  - `objects`
- It's well-documented, well-supported, and well-loved.
  - 0 issues on GitHub.
  - 30,000+ stars on GitHub.
  - 64,000+ dependants on NPM.
- It's thoroughly tested across multiple platforms and browsers.
- It's very efficient (more efficient than native JS methods).

Why Use Lodash?

- It has over 400 useful helper functions.
- It leads to clean, easy-to-use, and easy-to-read code.
- It automatically handles multiple variable types for you.
  - `arrays`
  - `strings`
  - `objects`
- It's well-documented, well-supported, and well-loved.
  - 0 issues on GitHub.
  - 30,000+ stars on GitHub.
  - 64,000+ dependants on NPM.
- It's thoroughly tested across multiple platforms and browsers.
- It's very efficient (more efficient than native JS methods).
- It's very light (69 KB)

Some Useful Lodash Functions

Let's look at a few useful Lodash methods...

_.map

`_.map(collection, iteratee)`

Creates an array of values by running each element in `collection` through `iteratee`.

`_.map`

`_.map(collection, iteratee)`

Creates an array of values by running each element in `collection` through `iteratee`.

`collection` *(Array | Object)*: The collection to iterate over.
`iteratee` *(Function)*: The function invoked per iteration.

The iteratee is invoked with three arguments: (`value`, `index|key`, `collection`).

_.map

`_.map(collection, iteratee)`

Creates an array of values by running each element in `collection` through `iteratee`.

`collection` *(Array | Object)*: The collection to iterate over.
`iteratee` *(Function)*: The function invoked per iteration.

The iteratee is invoked with three arguments: (`value`, `index|key`, `collection`).

returns *(Array)*: The new mapped array.

_.map

Example

```
function square(n) {
  return n * n;
}

_.map([4, 8], square);
// => [16, 64]

_.map({ 'a': 4, 'b': 8 }, square);
// => [16, 64] (iteration order is not guaranteed)

const users = [
  { 'user': 'barney' },
  { 'user': 'fred' }
];

// Using the `_.property` iteratee shorthand.
_.map(users, 'user');
// => ['barney', 'fred']
```

_.get

`_.get(object, path, [defaultValue])`

Gets the value at `path` of `object`. If the resolved value is undefined, the `defaultValue` is returned in its place.

`_.get`

`_.get(object, path, [defaultValue])`

Gets the value at `path` of `object`. If the resolved value is undefined, the `defaultValue` is returned in its place.

`object` *(Array | Object)*: The object to query.

`path` *(Array | string)*: The path of the property to get.

`defaultValue` *(*)*: The value returned for undefined results.

`_.get`

`_.get(object, path, [defaultValue])`

Gets the value at `path` of `object`. If the resolved value is undefined, the `defaultValue` is returned in its place.

`object` *(Array | Object)*: The object to query.
`path` *(Array | string)*: The path of the property to get.
`defaultValue` *(*)*: The value returned for undefined results.

returns *(*)*: The resolved value.

_.get

Example

```
const object = { 'a': [{ 'b': { 'c': 3 } }] };

_.get(object, 'a[0].b.c');
// => 3

_.get(object, ['a', '0', 'b', 'c']);
// => 3

_.get(object, 'a.b.c', 'default');
// => 'default'
```

`_.set`

`_.set(object, path, value)`

Sets the `value` at `path` of `object`. If a portion of `path` doesn't exist, it's created. Arrays are created for missing index properties while objects are created for all other missing properties.

`_.set`

`_.set(object, path, value)`

Sets the `value` at `path` of `object`. If a portion of `path` doesn't exist, it's created. Arrays are created for missing index properties while objects are created for all other missing properties.

`object` *(Array | Object)*: The object to modify.

`path` *(Array | string)*: The path of the property to set.

`value` *(*)*: The value to set.

`_.set`

`_.set(object, path, value)`

Sets the `value` at `path` of `object`. If a portion of `path` doesn't exist, it's created. Arrays are created for missing index properties while objects are created for all other missing properties.

`object` *(Array | Object)*: The object to modify.

`path` *(Array | string)*: The path of the property to set.

`value` *(*)*: The value to set.

returns *(Object)*: The mutated object.

_.set

Example

```
const object = { 'a': [{ 'b': { 'c': 3 } }] };

_.set(object, 'a[0].b.c', 4);
console.log(object.a[0].b.c);
// => 4

_.set(object, ['x', '0', 'y', 'z'], 5);
console.log(object.x[0].y.z);
// => 5
```

_.assign

`_.assign(object, [sources])`

Assigns own enumerable string keyed properties of source objects to the destination `object`. Source objects are applied from left to right. Subsequent `sources` overwrite property assignments of previous `sources`.

`_.assign`

`_.assign(object, [sources])`

Assigns own enumerable string keyed properties of source objects to the destination `object`. Source objects are applied from left to right. Subsequent `sources` overwrite property assignments of previous `sources`.

`object` *(Object)*: The destination object.

`sources` *(...Object)*: The source objects.

_.assign

`_.assign(object, [sources])`

Assigns own enumerable string keyed properties of source objects to the destination `object`. Source objects are applied from left to right. Subsequent `sources` overwrite property assignments of previous `sources`.

`object` *(Object)*: The destination object.
`sources` *(...Object)*: The source objects.

returns *(Object)*: The mutated object.

_.assign

Example

```
    const a = {'a': 0};

    const b = {'b': 1};

    _.assign({ 'a': 1 }, a, b);
    // => { 'a': 0, 'b': 1 }
```

`_.debounce`

`_.debounce(func, [wait], [options])`

Creates a debounced function that delays invoking `func` until after `wait` milliseconds have elapsed since the last time the debounced function was invoked.

`_.debounce`

`_.debounce(func, [wait], [options])`

Creates a debounced function that delays invoking `func` until after `wait` milliseconds have elapsed since the last time the debounced function was invoked.

`func` *(Function)*: The function to debounce.

`wait` *(number)*: The number of milliseconds to delay.

`options` *(Object)*: The options object.

`_.debounce`

`_.debounce(func, [wait], [options])`

Creates a debounced function that delays invoking `func` until after `wait` milliseconds have elapsed since the last time the debounced function was invoked.

`func` *(Function)*: The function to debounce.
`wait` *(number)*: The number of milliseconds to delay.
`options` *(Object)*: The options object.

returns *(Function)*: The new debounced function.

_.debounce

Example

```
function search(someSearchString) {
    // Preform a search API call
}

const searchValue = document.getElementById(
    "search-field"
);
searchValue.addEventListener(
    "keyup",
    _.debounce(search, 500)
);
```

`_.find`

`_.find(collection, [predicate], [fromIndex])`

Iterates over elements of `collection`, returning the first element `predicate` returns truthy for.

`_.find`

`_.find(collection, [predicate], [fromIndex])`

Iterates over elements of `collection`, returning the first element `predicate` returns truthy for.

`collection` *(Array | Object)*: The collection to inspect.
`iteratee` *(Function)*: The function invoked per iteration.
`fromIndex` *(number)*: The index to search from.

The predicate is invoked with three arguments:
(`value`, `index` | `key`, `collection`).

_.find

`_.find(collection, [predicate], [fromIndex])`

Iterates over elements of `collection`, returning the first element `predicate` returns truthy for.

`collection` *(Array | Object)*: The collection to inspect.

`iteratee` *(Function)*: The function invoked per iteration.

`fromIndex` *(number)*: The index to search from.

The predicate is invoked with three arguments: (`value`, `index` | `key`, `collection`).

returns *(*)*: The matched element, else undefined.

_.find

Example

```
    const users = [
      { 'user': 'barney',  'age': 36, 'active': true },
      { 'user': 'fred',    'age': 40, 'active': false },
      { 'user': 'pebbles', 'age': 1,  'active': true }
    ];

    _.find(users, o => o.age < 40);
    // => object for 'barney'

    // The `_.matches` iteratee shorthand.
    _.find(users, { 'age': 1, 'active': true });
    // => object for 'pebbles'

    // The `_.matchesProperty` iteratee shorthand.
    _.find(users, ['active', false]);
    // => object for 'fred'

    // The `_.property` iteratee shorthand.
    _.find(users, 'active');
    // => object for 'barney'
```

_.deburr

_.deburr([string])

Deburrs `string` by converting Latin-1 Supplement and Latin Extended-A letters to basic Latin letters and removing combining diacritical marks.

_.deburr

`_.deburr([string])`

Deburrs `string` by converting Latin-1 Supplement and Latin Extended-A letters to basic Latin letters and removing combining diacritical marks.

`string` *(string)*: The string to deburr.

_.deburr

`_.deburr([string])`

Deburrs `string` by converting Latin-1 Supplement and Latin Extended-A letters to basic Latin letters and removing combining diacritical marks.

`string` *(string)*: The string to deburr.

returns *(string)*: The deburred string.

_.deburr

Example

```
    _.deburr('déjà vu');
    // => 'deja vu'
```

`_.keyBy`

`_.keyBy(collection, [iteratee])`

Creates an object composed of keys generated from the results of running each element of `collection` through `iteratee`. The corresponding value of each key is the last element responsible for generating the key.

`_.keyBy`

`_.keyBy(collection, [iteratee])`

Creates an object composed of keys generated from the results of running each element of `collection` through `iteratee`. The corresponding value of each key is the last element responsible for generating the key.

`collection` *(Array | Object)*: The collection to iterate over.
`iteratee` *(Function)*: The iteratee to transform keys.

The iteratee is invoked with one argument: (`value`).

_.keyBy

`_.keyBy(collection, [iteratee])`

Creates an object composed of keys generated from the results of running each element of `collection` through `iteratee`. The corresponding value of each key is the last element responsible for generating the key.

`collection` *(Array | Object)*: The collection to iterate over.
`iteratee` *(Function)*: The iteratee to transform keys.

The iteratee is invoked with one argument: (`value`).

returns *(Object)*: The composed aggregate object.

_.keyBy

Example

```
const array = [
  { 'dir': 'left', 'code': 97 },
  { 'dir': 'right', 'code': 100 }
];

_.keyBy(array, o => String.fromCharCode(o.code));
/*
=> {
    'a': { 'dir': 'left', 'code': 97 },
    'd': { 'dir': 'right', 'code': 100 }
}
*/

_.keyBy(array, 'dir');
/*
=> {
    'left': { 'dir': 'left', 'code': 97 },
    'right': { 'dir': 'right', 'code': 100 }
}
*/
```

_.cloneDeep

`_.cloneDeep(value)`

Recursively deeply clones `value`.

`_.cloneDeep`

`_.cloneDeep(value)`

Recursively deeply clones `value`.

`value` *(*)*:	The value to recursively clone.

`_.cloneDeep`

`_.cloneDeep(value)`

Recursively deeply clones `value`.

`value` *(*)*: The value to recursively clone.

returns *(*)*: The deep cloned value.

_.cloneDeep

Example

```
    const objects = [{ 'a': 1 }, { 'b': 2 }];

    const deep = _.cloneDeep(objects);
    console.log(deep[0] === objects[0]);
    // => false
```

Questions?