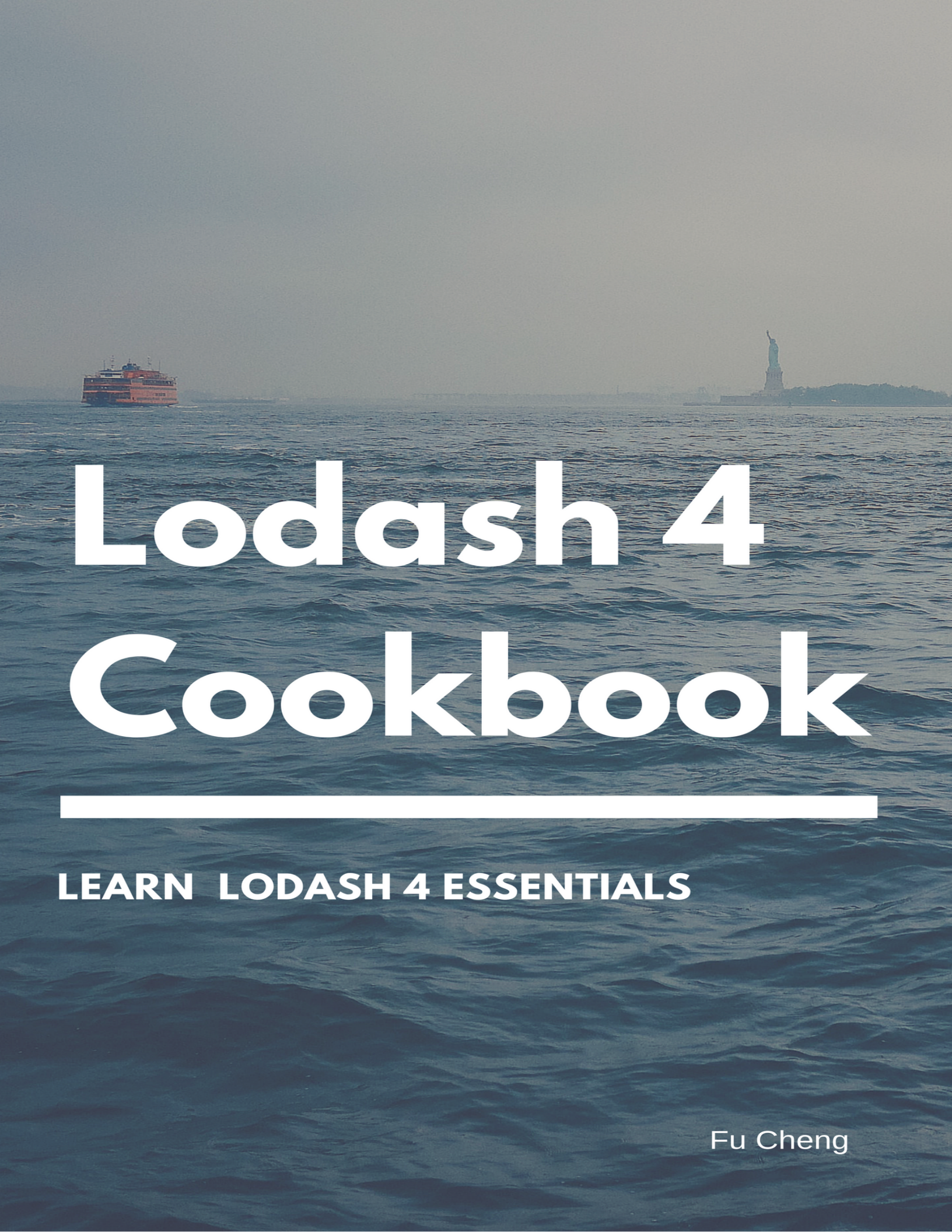


Lodash 4 Cookbook

LEARN LODASH 4 ESSENTIALS

Fu Cheng

Lodash 4 Cookbook

For lodash 4.17.10

Fu Cheng

This book is for sale at http://leanpub.com/lodashcookbook

This version was published on 2020-05-27

Tweet This Book!

Please help Fu Cheng by spreading the word about this book on Twitter!

The suggested hashtag for this book is #lodashcookbook.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#lodashcookbook

Also By Fu Cheng

JUnit 5 Cookbook

A Practical Guide for Java 8 Lambdas and Streams

ES6 Generators

To my wife Andrea and my daughters Olivia and Erica

Contents

1. Introduction

This book is about the popular JavaScript utilities library lodash[1]. Before we discuss lodash, we should understand why we need a JavaScript utilities library. With the prevalence of Web 2.0[2], Ajax[3] and NodeJS[4], JavaScript has become a very important programming language in both browser-side and server-side. Besides of the bad parts[5] of JavaScript language, JavaScript itself doesn't have a rich set of high-level API for developers to use, which makes common programming tasks hard to complete.

For example, it's a very common task to iterate an array and process all elements in this array in sequence. In some old browsers, the JavaScript `Array` object doesn't have the method `forEach()`. To iterate an array, `for` loop is required as in Listing 1.1. `process` is the function to process elements in the array.

Listing 1.1 Traditional approach to iterate an array

```
for (var i = 0, n = array.length; i < n; i++) {
  process(array[i]);
}
```

When using the method `forEach()`, the code in Listing 1.1 can be simplified as in Listing 1.2.

Listing 1.2 Use `forEach()` to iterate an array

```
array.forEach(process);
```

Comparing code snippets in Listing 1.1 and Listing 1.2, it's obvious that Listing 1.2 is much simpler to understand and easier to write and maintain than the code in Listing 1.1. That's why developers want more high-level APIs. JavaScript itself is evolving to add more language features and APIs, but the process is not fast enough. ECMAScript[6], the specification behind JavaScript, includes nine new methods for searching and manipulating array contents in 5th edition[7]. This means developers can use the method `forEach()` when JavaScript engine supports ECMAScript 5. But some old browsers, like IE 8, don't support ECMAScript 5, which means developers need to consider about cross-platform compatibility issues if supporting old browsers is a must. ECMAScript 6[8] specification is published in June 2015 with a lot of new features and enhancements.

[1] http://lodash.com/
[2] http://en.wikipedia.org/wiki/Web_2.0
[3] http://en.wikipedia.org/wiki/Ajax_(programming)
[4] http://nodejs.org
[5] Find out the good parts of JavaScript in Douglas Crockford's excellent book JavaScript: The Good Parts
[6] http://en.wikipedia.org/wiki/ECMAScript
[7] http://www.ecma-international.org/ecma-262/5.1/
[8] http://www.ecma-international.org/ecma-262/6.0/

See a comprehensive ECMAScript 5 & 6 compatibility table at here[9] and here[10].

Developers rely on JavaScript libraries to make daily development easier. The goal of libraries is to become the bridge between JavaScript runtime and developers. Developers can enjoy the high-level APIs provided by those libraries. Libraries are responsible for handling implementation details about how to use the low-level JavaScript APIs efficiently.

You may have heard about or even used another JavaScript utilities library Underscore[11]. Underscore provides a rich set of common APIs in the namespace _. Lodash also uses namespace _ and it's a drop-in replacement of Underscore with more features and performance improvements. If you already use Underscore, you can simply replace the Underscore with lodash, everything should just work.

This book is for the latest lodash 4.17.15 version.

1.1 Installation

Lodash is just a plain old JavaScript library, so it's very easy to install and use.

1.1.1 Web

In a web application, we can just download the lodash release JavaScript file and include it in the HTML page, then use _ in the JavaScript code.

Listing 1.3 Install lodash in HTML page

```
<script src="lodash.js"></script>
```

We can also use links provided by CDN servers to load lodash. CDN servers usually have different versions of lodash to choose. Listing 1.4 shows how to use cdnjs[12] to load lodash. cdnjs also provides the minified JavaScript version with source mapping file.

Listing 1.4 Load lodash from cdnjs

```
<script src="//cdnjs.cloudflare.com/ajax/libs/lodash.js/4.17.15/lodash.js">
</script>
```

No Bower support in v4

Bower support has been removed in v4 in favor of npm. But we can still use Bower to download lodash v4.

[9] http://kangax.github.io/compat-table/es5/
[10] http://kangax.github.io/compat-table/es6/
[11] http://underscorejs.org/
[12] https://cdnjs.com/libraries/lodash.js

1.1.2 NodeJS

In NodeJS, we can install lodash using npm[13] or yarn[14]; see Listing 1.5 and Listing 1.6.

Listing 1.5 Install lodash using npm

```
$ npm install --save lodash

$ npm install --save lodash@4.17.15
```

Listing 1.6 Install lodash using yarn

```
$ yarn add lodash

$ yarn add lodash@4.17.15
```

Then we can use `require` to import lodash package, see Listing 1.7.

Listing 1.7 Use lodash in NodeJS

```
var _ = require('lodash'); // Require the whole lodash package

var forEach = require('lodash/forEach'); // Require only forEach
```

It's recommended to only install NodeJS modules of actually used modules. For example, if the code only uses `_.forEach` method, then install the `lodash.foreach` module only.

Listing 1.8 Use lodash modules

```
$ npm install --save lodash.foreach

var forEach = require('lodash.foreach');
```

1.2 Lodash features

Lodash focuses on providing core features that are frequently used for JavaScript built-in objects, including:

- Arrays
- Objects

[13] http://npmjs.org
[14] https://yarnpkg.com/en/

- Functions
- Strings

Some of those features may have been included in latest version of ECMAScript specification. Some platforms may have also implemented extra features. If the underlying platform already supports a certain feature, lodash just uses the native implementation to improve performance.

1.3 Code sample convention

All code samples in this book are written in ECMAScript 6 JavaScript syntax and tested on NodeJS 6.9.4. Most of the code are written as Jest[15] test cases to verify the result. For those code that are not written as Jest code, the result of execution is provided below the actual code as a comment; see Listing 1.9.

Listing 1.9 Code sample convention

```
_.min([1, 2, 3]);
// -> 1
```

As in the Listing 1.9 above, `_.min([1, 2, 3]);` is the actual code, `1` after `// ->` is the execution result.

The complete source code of this book can be found on GitHub[16].

1.4 About this book

Lodash is a well-documented JavaScript library with comprehensive official documentation[17]. This book is a simple and concise guide on how to use lodash in practice. It covers core features and most frequently used functions.

[15] https://facebook.github.io/jest
[16] https://github.com/VividcodeIO/lodash4cookbook
[17] https://lodash.com/docs

2. Common concepts

Before diving into details of lodash functions, we start from some common concepts in lodash.

2.1 Truthy and falsy

Truthy and *falsy* values are very important when using lodash predicates. `false`, `0`, `""`(empty string), `null`, `undefined` and `NaN` are falsy values in JavaScript. All other values are truthy values.

2.2 SameValueZero

SameValueZero[1] is the algorithm of how to compare two values in lodash. It's similar with JavaScript "strict equality comparison" (===), except the handling of `NaN`. It always make developers confused as `NaN === NaN` returns `false`. *SameValueZero* removes that confusion, so `NaN` is same to `NaN` in *SameValueZero* algorithm.

2.3 Predicates

Predicate functions only return truthy or falsy values. They are used frequently in lodash. For example, when filtering a collection, a predicate function is required to determine what kind of elements should be kept.

Predicate functions can be written as plain old JavaScript functions. Lodash also provides some helper functions to generate predicate functions for common use cases.

2.3.1 `matches`

`_.matches(source)` takes a source object and creates a new function which performs a deep comparison between the given object and the source object. `_.matches` supports comparison of different types of data, including booleans, numbers, strings, `Date` objects, `RegExp` objects, `Object` objects and arrays. Listing 2.1 shows how `_.matches` works by comparing strings and objects.

[1] http://www.ecma-international.org/ecma-262/6.0/#sec-samevaluezero

Listing 2.1 Match by object comparison

```
const matches = require('lodash/matches');

describe('matches', () => {
  it('should match strings', () => {
    let f = matches('hello');
    expect(f('world')).toBe(false);
    expect(f('hello')).toBe(true);
  });

  it('should match objects', () => {
    let f = matches([{a: 1}, {b: 2}]);
    expect(f([{a: 1}, {b: 3}])).toBe(false);
  });
});
```

2.3.2 matchesProperty

`_.matchesProperty(path, value)` takes a property path and the expected value of this property path to create a new function that checks if the given object's value of the same property path matches the expected value. Listing 2.2 shows how `_.matchesProperty` works by matching simple property name, built-in property and nested property path.

Listing 2.2 Match by comparing property value

```
const matchesProperty = require('lodash/matchesProperty');

describe('matchesProperty', () => {
  it('should match property name', () => {
    let f = matchesProperty('name', 'Alex');
    expect(f({name: 'Alex'})).toBe(true);
  });

  it('should match built-in property', () => {
    let f = matchesProperty('length', 5);
    expect(f('hello')).toBe(true);
  });

  it('should match nested path', () => {
    let f = matchesProperty('user.name', 'Alex');
    expect(f({user: {name: 'Alex'}})).toBe(true);
  });
});
```

2.3.3 property

`_.property(path)` takes a property path and creates a new function which returns the value of this property path in the given object. `_.property` can be used to create predicate functions with property values converted to truthy or falsy values.

Listing 2.3 Extract property value

```
const property = require('lodash/property');

describe('property', () => {
  it('should extract property value', () => {
    let f = property('name');
    expect(f({name: 'Alex'})).toBe('Alex');
  });
});
```

For lodash functions which accept predicates, e.g. `_.find` and `_.filter`, predicates can be specified using functions, strings, and objects.

- If a function is provided, it's used directly. The predicate matches if the function returns a truthy value.
- If only a string is provided, it's used to create a function using `_.property` as the predicate.
- If an array that contains a string and a value is provided, the string and the value are used to create a function using `_matchesProperty` as the predicate.
- If an object is provided, it's used to create a function using `_.matches` as the predicate.

For example, given an array shown in Listing 2.4,

Listing 2.4 Example input JSON array

```
[
  {
    "name": "Alex",
    "age": 30,
    "is_premium": false
  },
  {
    "name": "Bob",
    "age": 20,
    "is_premium": true
  },
  {
    "name": "Mary",
```

```
    "age": 25,
    "is_premium": false
  }
]
```

`_.find` returns the first matching element in the array. A JavaScript function can be used as the predicate to `_.find`. In Listing 2.5, we finds the first element with `age` greater than `18` in the array `users`. The result is the first element with name `Alex`.

Listing 2.5 Find using a function

```
const find = require('lodash/find');

describe('find with different predicates', () => {
  it('should find with a function', () => {
    let user = find(users, user => user.age > 18);
    expect(user).toBeDefined();
    expect(user.name).toBe('Alex');
  });
});
```

If a string is passed as the predicate, it's treated as a property name of objects in the array. In Listing 2.6, we finds the first element with truthy value of the property `is_premium` in the array. The actual used predicate is `_.property('is_premium')`. The result is the second element with name `Bob`.

Listing 2.6 Find using a property value

```
it('should find with a property value', () => {
  let user = find(users, 'is_premium');
  expect(user).toBeDefined();
  expect(user.name).toBe('Bob');
});
```

If an object is passed as the predicate, it's treated as a search example. Objects in returned results must have exactly the same values for all the corresponding properties provided in the search example. In Listing 2.7, we finds the first element with the value of the property `name` equals to `Alex` in the array. The actual used predicate is `_.matches({ name: 'Alex' })`.

Listing 2.7 Find using an object

```
it('should find with an object', () => {
  let user = find(users, { name: 'Alex' });
  expect(user).toBeDefined();
  expect(user.name).toBe('Alex');
});
```

If the name of an argument of a lodash function is `predicate`, it means this argument supports the predicate syntax described above.

2.4 Iteratees

Iteratees are used by lodash functions which require iterating through a collection. Iteratee is invoked for each element in the collection and the result is used instead of the original element. Iteratees are typically used to transform collections. A typical usage of iteratee is in the function `_.map`. The second argument of `_.map` is the iteratee. The result of applying iteratee to each element in the collection is collected and returned. In Listing 2.8, we use a function to transform input array `[1, 2, 3]` to `[3, 6, 9]`.

Listing 2.8 `map` using an iteratee function

```
const map = require('lodash/map');

describe('map with iteratees', () => {
  it('should map with an iteratee function', () => {
    let result = map([1, 2, 3], n => n * 3);
    expect(result).toEqual([3, 6, 9]);
  });
});
```

If the name of an argument of a lodash function is `iteratee`, it means this argument is an iteratee function.

2.4.1 Iteratee shorthand

When iteratee functions are required, we can also use the similar syntax as predicate functions to quickly create them. These iteratee shorthands use methods `_.matches`, `_.matchesProperty` or `_.property` behind the scene.

In the second invocation of `_.map` in Listing 2.9, the second argument of `_.map` must be an array to indicate that it uses `_.matchesProperty`.

Listing 2.9 `map` using an iteratee shorthand

```
it('should map with iteratee shorthands', () => {
  let result = map(users, {name: 'Alex'});
  expect(result).toEqual([true, false, false]);

  result = map(users, ['name', 'Alex']);
  expect(result).toEqual([true, false, false]);

  result = map(users, 'name');
  expect(result).toEqual(['Alex', 'Bob', 'Mary']);
});
```

2.5 `this` binding

In Lodash 3, we can use the argument `thisArg` to specify the value of `this` binding. In Lodash 4, `thisArg` has been removed in most methods. To specify the binding object, `_.bind` should be used explicitly. In Listing 2.10, when the function `add` is invoked, `this` value is bound to `obj`.

Listing 2.10 `map` with `this` binding using `_.bind`

```
const map = require('lodash/map');
const bind = require('lodash/bind');

describe('this binding', () => {
  it('should bind to this', () => {
    const obj = {
      val: 10,
      add: function(n) {
        return this.val + n;
      }
    };
    let result = map([1, 2, 3], bind(obj.add, obj));
    expect(result).toEqual([11, 12, 13]);
  });
});
```

If the name of an argument of a lodash function is `thisArg`, then this function supports binding `this` value.

3. Collections

A collection is an object that contains iterable elements. In lodash, collections can be arrays, objects, and strings. Lodash has a rich set of functions to work with collections.

In this chapter, we use the following JSON array as the sample data `fruits` for some code samples.

Listing 3.1 Sample data `fruits`

```
[
  {
    "name": "apple",
    "price": 0.99,
    "onSale": true
  },
  {
    "name": "orange",
    "price": 1.99,
    "onSale": false
  },
  {
    "name": "passion fruit",
    "price": 4.99,
    "onSale": false
  }
]
```

3.1 Each

`_.each(collection, [iteratee=_.identity])` and `_.eachRight(collection, [iteratee=_.identity])` iterate over elements in the collection and invoke the iteratee function. The difference is that `_.eachRight` iterates from right to left. `_.forEach` is an alias of `_.each`, while `_.forEachRight` is an alias of `_.eachRight`.

Listing 3.2 Iterate collections

```
const each = require('lodash/each');

describe('each', () => {
  it('should support basic iteration', () => {
    let sum = 0;
    each([1, 2, 3], val => sum += val);
    expect(sum).toEqual(6);
  });
});
```

3.2 Every and some

`_.every(collection, [predicate=_.identity])` checks if **all** elements in the collection match the given predicate.

Listing 3.3 Check if all elements match certain condition

```
const every = require('lodash/every');

describe('every', () => {
  it('should support arrays with functions', () => {
    let result = every([1, 2, 3, 4], n => n % 2 === 0);
    expect(result).toBe(false);
  });

  it('should support arrays with property value', () => {
    const fruits = [
      {
        name: 'apple',
        price: 1.99,
        onSale: true
      },
      {
        name: 'orange',
        price: 0.99,
        onSale: true
      }
    ];
    let result = every(fruits, ['onSale', true]);
    expect(result).toBe(true);
```

```
  });

  it('should support objects', () => {
    const obj = {
      a: 1,
      b: 2,
      c: 3
    };
    let result = every(obj, n => n % 2 === 0);
    expect(result).toBe(false);
  });

  it('should support strings', () => {
    let result = every('aaaa', c => c === 'a');
    expect(result).toBe(true);
  });
});
```

`_.some(collection, [predicate=_.identity])` is the opposite of `_.every` which checks if **any** element in the collection matches the given predicate. `_.some` doesn't need to iterate the entire collection and the iteration exits as soon as a matching element is found.

Listing 3.4 Check if any element matches certain condition

```
const some = require('lodash/some');

describe('some', () => {
  it('should support arrays', () => {
    let result = some([1, 2, 3, 4], n => n % 2 === 0);
    expect(result).toBe(true);
  });

  it('should support strings', () => {
    let result = some('hello', c => c === 'x');
    expect(result).toBe(false);
  });
});
```

3.3 Filter and reject

`_.filter(collection, [predicate=_.identity])` filters a collection by returning elements matching the given predicate. `_.reject(collection, [predicate=_.identity])` is the opposite of

`_.filter` that returns elements **not** matching the given predicate. When `_.filter` is used to filter objects, only values of matching properties are returned. If you want to keep the original object structure, use `_.pick` or `_.omit` instead. When `_.filter` is used on strings, matching characters are returned in an array.

Listing 3.5 Filter a collection

```
const filter = require('lodash/filter');

describe('filter', () => {
  it('should support arrays', () => {
    let result = filter(['a', 'b', 'c'], c => c > 'b');
    expect(result).toEqual(['c']);
  });

  it('should support objects', () => {
    const obj = {
      a: 1,
      b: 2,
      c: 3,
    };
    let result = filter(obj, n => n > 1);
    expect(result).toEqual([2, 3]);
  });

  it('should support strings', () => {
    let result = filter('hello', c => c !== 'l');
    expect(result).toEqual(['h', 'e', 'o']);
  });
});
```

Listing 3.6 shows the examples of `_.reject`.

Listing 3.6 Reject elements in a collection

```
const reject = require('lodash/reject');

describe('reject', () => {
  it('should support arrays', () => {
    let result = reject(['a', 'b', 'c'], c => c > 'b');
    expect(result).toEqual(['a', 'b']);
  });
});
```

`_.filter` and `_.reject` always return a new array. The input collection is not modified. A new array of filtered or rejected elements, object property values or characters is returned.

3.4 Size

`_.size(collection)` gets the size of a collection. For arrays, the size is the array's length, same as the array's property `length`. For objects, the size is the number of own enumerable properties, i.e. the length of the array returned by `_.keys`. For strings, the size is the string's length.

Listing 3.7 Get the size of a collection

```
const size = require('lodash/size');

describe('size', () => {
  it('should support arrays', () => {
    expect(size([1, 2])).toEqual(2);
  });

  it('should support objects', () => {
    expect(size({
      a: 1,
      b: 2,
      c: 3,
    })).toEqual(3);
  });

  it('should support strings', () => {
    expect(size('hello')).toEqual(5);
  });
});
```

3.5 Includes

`_.includes(collection, value, [fromIndex=0])` checks if a collection contains the given value. An optional index can be provided as the starting position to search. If the collection is an object, values of this object's properties, i.e. the result of `_.values`, are searched instead. `_.includes` uses the `SameAsZero` algorithm to check equality.

Listing 3.8 Check if a collection contains the given value

```
const includes = require('lodash/includes');

describe('includes', () => {
  it('should support arrays', () => {
    expect(includes(['a', 'b', 'c'], 'a')).toBe(true);
  });

  it('should support arrays with index', () => {
    expect(includes(['a', 'b', 'c'], 'a', 1)).toBe(false);
  });

  it('should support objects', () => {
    expect(includes({
      a: 1,
      b: 2,
      c: 3
    }, 1)).toBe(true);
  });

  it('should support strings', () => {
    expect(includes('hello', 'h')).toBe(true);
  });
});
```

3.6 Sample

`_.sample(collection)` gets a single random element from a collection. `_.sampleSize(collection, [n=1])` gets `n` random elements with unique keys from a collection.

Listing 3.9 Get random elements from a collection

```
_.sample(['a', 'b', 'c']);
// -> 'a'

_.sample({
  a: 1,
  b: 2,
  c: 3
});
// -> 1
```

```
_.sample('hello');
// -> 'h'

_.sampleSize('hello', 2)
// -> ['h', 'l']
```

3.7 Shuffle

`_.shuffle(collection)` shuffles a collection by generating a random permutation. Lodash uses the Fisher-Yates shuffle[1] algorithm to shuffle the collection. For objects, the return value of `_.shuffle` is a random permutation of the property values.

Listing 3.10 Shuffle a collection

```
_.shuffle(['a', 'b', 'c']);
// -> ['b', 'c', 'a']

_.shuffle({
  a: 1,
  b: 2,
  c: 3
});
// -> [1, 2, 3]

_.shuffle('hello');
// -> ['l', 'l', 'o', 'h', 'e']
```

Due to the random nature of `_.sample`, `_.sampleSize` and `_.shuffle`, most likely the result will be different when running Listing 3.10 on your local machine.

3.8 Partition

`_.partition(collection, [predicate=_.identity])` splits a collection into two groups based on the result of invoking the predicate on each element. The first group contains elements for which the predicate returns a truthy value, while the second group contains elements for which the predicate returns a falsy value.

[1] https://en.wikipedia.org/wiki/Fisher%E2%80%93Yates_shuffle

Listing 3.11 Split a collection into two groups

```
const partition = require('lodash/partition');
const fruits = require('../data/fruits.json');

describe('partition', () => {
  it('should support arrays', () => {
    let result = partition(['a', 'b', 'c'], char => char > 'a');
    expect(result.length).toBe(2);
    expect(result[0]).toEqual(['b', 'c']);
    expect(result[1]).toEqual(['a']);
  });

  it('should support predicate syntax', () => {
    let result = partition(fruits, 'onSale');
    expect(result.length).toBe(2);
    expect(result[0].length).toBe(1);
    expect(result[1].length).toBe(2);
  });

  it('should support strings', () => {
    let result = partition('hello', char => char > 'l');
    expect(result.length).toBe(2);
    expect(result[0]).toEqual(['o']);
    expect(result[1]).toEqual(['h', 'e', 'l', 'l']);
  });
});
```

3.9 Count by

`_.countBy(collection, [iteratee=_.identity])` applies a function to each element in the collection and counts the number of occurrences of each result. The counting result is returned as an object with the applied result as the keys and the count as the corresponding values.

Listing 3.12 Count the number of occurrences

```
const countBy = require('lodash/countBy');

describe('countBy', () => {
  it('should support arrays', () => {
    expect(countBy([1, 2, 3], n => n > 1)).toEqual({
      true: 2,
      false: 1,
    });
  });

  it('should support objects', () => {
    expect(countBy({
      a: 1,
      b: 1,
      c: 2,
    }, val => val / 2)).toEqual({
      1: 1,
      0.5: 2,
    });
  });

  it('should support strings', () => {
    expect(countBy('hello', char => char === 'l')).toEqual({
      true: 2,
      false: 3,
    });
  });
});
```

3.10 Group by and key by

`_.groupBy(collection, [iteratee=_.identity])` applies a function to each element in the collection and groups the elements by the result. Elements have the same result will be in the same group. The grouping result is returned as an object. The keys in the object are the applied results, while the values are arrays of elements which generate the corresponding result.

Listing 3.13 Group elements

```
const groupBy = require('lodash/groupBy');

describe('groupBy', () => {
  it('should support arrays', () => {
    expect(groupBy([1, 2, 3], n => n > 1)).toEqual({
      true: [2, 3],
      false: [1],
    });
  });

  it('should support objects', () => {
    expect(groupBy({
      a: 1,
      b: 1,
      c: 2,
    }, val => val / 2)).toEqual({
      1: [2],
      0.5: [1, 1],
    });
  });

  it('should support strings', () => {
    expect(groupBy('hello', char => char === 'l')).toEqual({
      true: ['l', 'l'],
      false: ['h', 'e', 'o'],
    });
  });
});
```

The difference between `_.countBy` and `_.groupBy` is that `_.countBy` only returns the number of grouped elements.

`_.keyBy(collection, [iteratee=_.identity])`'s behavior is similar with `_.groupBy`, but `_.keyBy` only keeps the last element for each key.

Listing 3.14 Get the last element of grouping

```
const keyBy = require('lodash/keyBy');

describe('keyBy', () => {
  it('should support arrays', () => {
    expect(keyBy([1, 2, 3], n => n > 1)).toEqual({
      true: 3,
      false: 1,
    });
  });

  it('should support objects', () => {
    expect(keyBy({
      a: 1,
      b: 1,
      c: 2,
    }, val => val / 2)).toEqual({
      1: 2,
      0.5: 1,
    });
  });

  it('should support strings', () => {
    expect(keyBy('hello', char => char === 'l')).toEqual({
      true: 'l',
      false: 'o',
    });
  });
});
```

3.11 `invokeMap`

`_.invokeMap(collection, path, [args])` invokes a method on each element in the collection and returns the results in an array. The method to invoke is specified by the path, can be the function's name or the function itself. Additional arguments can also be provided for the method invocation. In Listing 3.14, when the function is invoked, `this` references to the current element.

Listing 3.15 Invoke a method on each element in the collection

```
const invokeMap = require('lodash/invokeMap');

describe('invokeMap', () => {
  it('should support method names', () => {
    expect(invokeMap(['a', 'b', 'c'], 'toUpperCase')).toEqual(['A', 'B', 'C']);
  });

  it('should support extra arguments', () => {
    expect(invokeMap([['a', 'b'], ['c', 'd']], 'join', ''))
      .toEqual(['ab', 'cd']);
  });

  it('should support functions', () => {
    expect(invokeMap([{a: 1}, {a: 2}], function(toAdd) {
      return this.a + toAdd;
    }, 3)).toEqual([4, 5]);
  });
});
```

3.12 Map and reduce

Map and reduce are common operations when processing collections. *Map* transforms a collection into another collection by applying an operation to each element in the collection. *Reduce* transforms a collection into a single value by accumulating results of applying an operation to each element. The result of the last operation is used as the input of the current operation.

3.12.1 Map

`_.map(collection, [iteratee=_.identity])` is the generic map function. We can use the different iteratee syntax.

Listing 3.16 Generic map operation

```
const map = require('lodash/map');

describe('map', () => {
  it('should support arrays', () => {
    expect(map([1, 2, 3], n => n * 2)).toEqual([2, 4, 6]);
  });

  it('should support iteratee syntax', () => {
    const users = [
      {
        name: 'Alex',
      },
      {
        name: 'Bob',
      }
    ];
    expect(map(users, 'name')).toEqual(['Alex', 'Bob']);
    expect(map(users, {name: 'Alex'})).toEqual([true, false]);
  });
});
```

3.12.2 Reduce

`_.reduce(collection, [iteratee=_.identity], [accumulator])` has similar arguments list with `_.map`, except that it accepts an optional value as the initial input of the first reduce operation. If the initial value is not provided, the first element in the collection is used instead. The provided iteratee function will be invoked with four arguments, `accumulator`, `value`, `index/key` and `collection`. `accumulator` is the current reduced value, while `value` is the current element in the collection. The returned result of the iteratee function invocation is passed as the `accumulator` value of the next invocation.

Listing 3.17 Use `_.reduce` to sum the values in an array

```
const reduce = require('lodash/reduce');

describe('reduce', () => {
  it('should support no initial value', () => {
    let result = reduce([1, 2, 3],
      (accumulator, value) => accumulator + value);
    expect(result).toEqual(6);
  });

  it('should support initial value', () => {
    let result = reduce([1, 2, 3],
      (accumulator, value) => accumulator + value, 100);
    expect(result).toEqual(106);
  });
});
```

`_.reduceRight(collection, [iteratee=_.identity], [accumulator]` is similar with `_.reduce)` except `_.reduceRight` iterates all the elements from right to left.

Listing 3.18 Reduce elements from right to left

```
const reduceRight = require('lodash/reduceRight');
const reduce = require('lodash/reduce');

describe('reduceRight', () => {
  it('should support strings', () => {
    let result = reduceRight('hello',
      (accumulator, value) => accumulator.toUpperCase() + value);
    expect(result).toEqual('OLLEh');

    result = reduce('hello',
      (accumulator, value) => accumulator.toUpperCase() + value);
    expect(result).toEqual('HELLo');
  });
});
```

3.13 Search

Search is a very common task in programming. Search is performed on iterable collections with given conditions. The return result is the first element in the collection matching the condition, or `undefined` if no matching element is found.

3.13.1 `find`

`_.find(collection, [predicate=_.identity], [fromIndex=0])` is the generic function to search in collections. When invoking `_.find`, the collection itself and the search condition should be provided. We can also provide an optional starting index for the search. `_.find` supports the same predicate syntax. If a function is provided as the predicate, the function is invoked for each element in the array until the function returns a truthy value. The function is invoked with three arguments: the currently iterated element, index or key of the element and the collection itself.

Listing 3.19 Find

```
const find = require('lodash/find');
const fruits = require('../data/fruits.json');

describe('find', () => {
  it('should support function predicates', () => {
    let result = find(fruits, fruit => fruit.price <= 2);
    expect(result).toBeDefined();
    expect(result.name).toEqual('apple');
  });

  it('should support property predicates', () => {
    let result = find(fruits, 'onSale');
    expect(result).toBeDefined();
    expect(result.name).toEqual('apple');

    result = find(fruits, ['name', 'orange']);
    expect(result).toBeDefined();
    expect(result.name).toEqual('orange');
  });

  it('should support object predicates', () => {
    let result = find(fruits, {
      name: 'passion fruit',
      onSale: false,
    });
    expect(result).toBeDefined();
    expect(result.name).toEqual('passion fruit');
  });
});
```

3.13.2 `findLast`

`_.findLast(collection, [predicate=_.identity], [fromIndex=collection.length-1])` is similar with `_.find`, but `_.findLast` iterates over all elements of the collection in reverse order. For arrays, it searches from the last element. For strings, it searches from the last character. For objects, it searches from the last element of the array of property names returned by `_.keys`.

Listing 3.20 Find in reverse order

```
const findLast = require('lodash/findLast');

describe('findLast', () => {
  it('should support strings', () => {
    expect(findLast('hello', char => char < 'f')).toEqual('e');
  });
});
```

3.14 Sort

`_.sortBy(collection, [iteratee=_.identity])` sorts a collection in ascending order with results after applying the iteratee function to each element in the collection. The sort is stable, which means it preserves original order for elements with equality. We can use multiple iteratees as sort conditions. If multiple elements in the collection have the same value for the first property name, those elements are sorted using the second property name, and so on.

Listing 3.21 Sort a collection

```
const sortBy = require('lodash/sortBy');

describe('sortBy', () => {
  it('should support simple sort', () => {
    expect(sortBy([3, 2, 1])).toEqual([1, 2, 3]);
  });

  it('should support function predicates', () => {
    let result = sortBy([-3, 2, 1], val => Math.abs(val));
    expect(result).toEqual([1, 2, -3]);
  });

  it('should support multiple conditions', () => {
    const users = [
      {
        name: 'David',
```

```
        age: 28,
      },
      {
        name: 'Alex',
        age: 30,
      },
      {
        name: 'Bob',
        age: 28,
      }
    ];
    let result = sortBy(users, 'age', 'name');
    expect(result[0].name).toEqual('Bob');
    expect(result[1].name).toEqual('David');
    expect(result[2].name).toEqual('Alex');
  });
});
```

3.15 `flatMap`

`_.flatMap(collection, [iteratee=_.identity])` invokes an iteratee function to each element in a collection. The result of each iteratee function invocation is an array. All result arrays are concatenated and flattened into a single array as the final result.

`_.flatMapDeep(collection, [iteratee=_.identity])` is similar with `_.flatMap` except that `_.flatMapDeep` recursively flattens the result array until it's completely flattened.

`_.flatMapDepth(collection, [iteratee=_.identity], [depth=1])` is similar with `_.flatMapDeep` except that it only flattens the result the given times. The default value of `depth` is `1`, so `_.flatMapDepth(array, iteratee)` is the same as `_.flatMap(array, iteratee)`.

Listing 3.22 Example of `_.flatMap` and `_.flatMapDeep`

```
const flatMap = require('lodash/flatMap');
const flatMapDeep = require('lodash/flatMapDeep');

describe('flatMap', () => {
  it('should support basic operation', () => {
    const map = value => [value + 1, value - 1];
    let result = flatMap([1, 2], map);
    expect(result).toEqual([2, 0, 3, 1]);
  });
```

```
  it('should support recursion', () => {
    const map = value => [[value + 1], [value - 1]];
    let result = flatMap([1, 2], map);
    expect(result).toEqual([[2], [0], [3], [1]]);

    result = flatMapDeep([1, 2], map);
    expect(result).toEqual([2, 0, 3, 1]);
  });
});
```

4. String templates

If you want to generate strings from a template, `_.template([string=''], [options={}])` is a simple yet powerful function to do that. It can be used by libraries and applications to avoid long string concatenations. Grunt uses `_.template` to support templates in configuration file[1]. It can also be used in applications to generate HTML markups, messages, emails and more.

Listing 10.1 Basic usage of string templates

```
let tpl = _.template('Hello, <%= name %>. Current time is <%= new Date() %>.');
tpl({
  name: 'Alex'
});
// -> 'Hello, Alex. Current time is Fri Jun 23 2017 20:07:19 GMT+1200 (NZST).'
```

In Listing 10.1, the input argument of `_.template` is the template itself. In the template, `<%=` and `%>` are the delimiters to wrap variables to be evaluated at runtime. `_.template` returns a new function. After invoking the returned function with a context object that contains actual values of template variables, it returns the generated string. If no value is assigned to a variable, an empty string will be used.

`_.template` supports three types of delimiters, *interpolate*, *escape* and *evaluate*.

4.1 interpolate

interpolate delimiters allow to interpolate variables. The default regular expression pattern to declare interpolated variables is `/<%=([\s\S]+?)%>/g`. Simple variables and complex expressions are both supported. The regular expression pattern of *interpolate* delimiters can be customized by the property `interpolate` of the options object.

[1] http://gruntjs.com/api/grunt.template

Listing 10.2 Use interpolate delimiters

```
let tpl = _.template('Hello, <%= name %>, the total amount is <%= order.amount + 10 \
%>.');
tpl({
  name: 'Alex',
  order: {
    amount: 100,
  }
});
// -> 'Hello, Alex, the total amount is 110.'
```

4.2 escape

It's common to use `_.template` to generate HTML markups. *escape* delimiters allow to interpolate variables and escape the result values. The default regular expression pattern to declare escaped variables is `/<%-([\s\S]+?)%>/g`. The pattern can be customized by the property `escape` of the options object.

Listing 10.3 Use escape delimiters

```
let tpl = _.template('<div><%- markup %></div>');
tpl({
  markup: '<span>Hello</span>'
});
// -> '<div>&lt;span&gt;Hello&lt;/span&gt;</div>'
```

4.3 evaluate

evaluate delimiters allow to execute JavaScript code. This kind of delimiters is useful when adding logic to templates, e.g. adding condition checks or loops to the template. The default regular expression pattern to declare JavaScript code is `/<%([\s\S]+?)%>/g`. The pattern can be customized by the property `evaluate` of the options object.

Listing 10.4 Use evaluate delimiters

```
let tpl = _.template('<% if (a > 0) { %> Good! <% } else { %> Bad! <% } %>');
tpl({
  a: 1
});
// -> ' Good! '
tpl({
  a: -1
});
// -> ' Bad! '
```

4.4 imports

Besides from the context object passed to the function created by `_.template`, a default object can also to passed to `_.template` as an additional source when evaluating variables. The object is specified using the property `imports` of the options object. The default values in the imported object can be overridden by values in the context object.

Listing 10.5 Use extra imports

```
let tpl = _.template('Hi, <%= user %>, you should pay <%= amount * discount %>.',
  {
    imports: {
      discount: 0.8,
  }
});
tpl({
  user: 'Alex',
  amount: 100
});
// -> 'Hi, Alex, you should pay 80.'
tpl({
  user: 'Bob',
  amount: 100,
  discount: 0.9
});
// -> 'Hi, Bob, you should pay 90.'
```

The default imports object contains only lodash object itself with the key _, so lodash methods can be used directly in the expressions.

4.5 Data object name

By default, when evaluating variables in the template, variables use the same names as in the context object. The property `variable` of the options object sets a name to the context object, then variable names should be changed accordingly.

Listing 10.6 Use different variable object name

```
var tpl = _.template('Hello, <%= user.name %>.', {
  variable: 'user',
});
tpl({
  name: 'Alex'
});
// -> 'Hello, Alex.'
```

In Listing 10.6, the context object's name is set to `user`, so `user.name` accesses the property `name` of the context object.

5. Recipes

5.1 Filter an object's properties

5.1.1 Scenario

Filter a given object by removing certain properties.

5.1.2 Solution

Although `_.filter` and `_.reject` can be applied to objects, they cannot be used for this scenario, because `_.filter` and `_.reject` return an array of property values after filtering. `_.pick`, `_.pickBy`, `_.omit` and `_.omitBy` should be used instead.

Listing 11.1 Filter a given object by removing certain properties

```
let fruits = {
  apple: {
    name: 'Apple',
    price: 2.99
  },
  orange: {
    name: 'Orange',
    price: 1.99
  },
  banana: {
    name: 'Banana',
    price: 0.5
  }
};

_.pickBy(fruits, fruit => fruit.price > 2);
// -> { apple: { name: 'Apple', price: 2.99 } }

_.pickBy(fruits, (fruit, key) =>  key != 'apple');
// -> { orange: { name: 'Orange', price: 1.99 },
//      banana: { name: 'Banana', price: 0.5 } }

_.pick(fruits, 'apple');
// -> { apple: { name: 'Apple', price: 2.99 } }
```

When a predicate function is passed to `_.pickBy` or `_.omitBy`, it's invoked with three arguments: property value, property name and the object itself.

5.2 Push an array of elements into an array

5.2.1 Scenario

Given an array of elements, push those elements into another array.

5.2.2 Solution

If using `Array`'s `push` method, the whole array will be pushed as a single element.

Listing 11.2 Use `Array`'s `push` method

```
let array = [1, 2, 3];
array.push([4, 5, 6]);
console.log(array);
// -> [1, 2, 3, [4, 5, 6]]
```

The first solution is to use `_.spread` to wrap `push` method to accept arrays as arguments.

Listing 11.3 Use `_.spread` to wrap `push` method

```
let array = [1, 2, 3];
let push = _.bind(_.spread(Array.prototype.push), array);
push([4, 5, 6]);
console.log(array);
// -> [1, 2, 3, 4, 5, 6]
```

The second solution is to push the array first, then use `_.flatten` to flatten the array.

Listing 11.4: Use `_.flatten` to flatten the array

```
let array = [1, 2, 3];
array.push([4, 5, 6]);
_.flatten(array);
// -> [1, 2, 3, 4, 5, 6]
```

5.3 Process data for C3.js pie chart

5.3.1 Scenario

C3.js[1] is a popular chart library based on d3.js[2]. C3.js can create pie chart[3] based on data input. But when there are many items in the data set, the pie chart itself becomes very hard to read.

Listing 11.5 is the basic code to create a pie chart with `100` items.

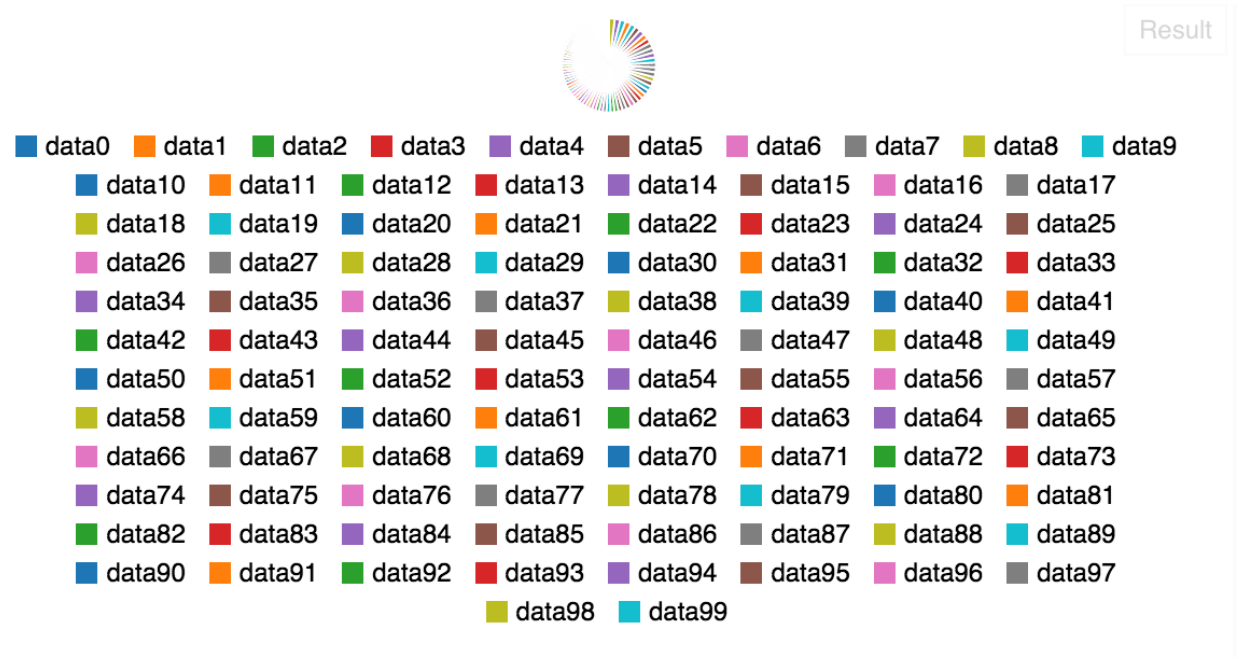
Listing 11.5 Basic code of create a pie chart

```
function generateData(num) {
    var data = [];
    for (var i = 0; i < num; i++) {
        data.push(['data' + i, (i <= 20 ? 1000 : 0) +  Math.random() * 10]);
    }
    return data;
}

let chart = c3.generate({
    bindto: '#chart',
    data: {
      columns: generateData(100),
      type: 'pie'
    }
});
```

Below is how this chart looks like.

[1] http://c3js.org/
[2] http://d3js.org/
[3] http://c3js.org/samples/chart_pie.html



Unreadable pie chart with 100 items

5.3.2 Solution

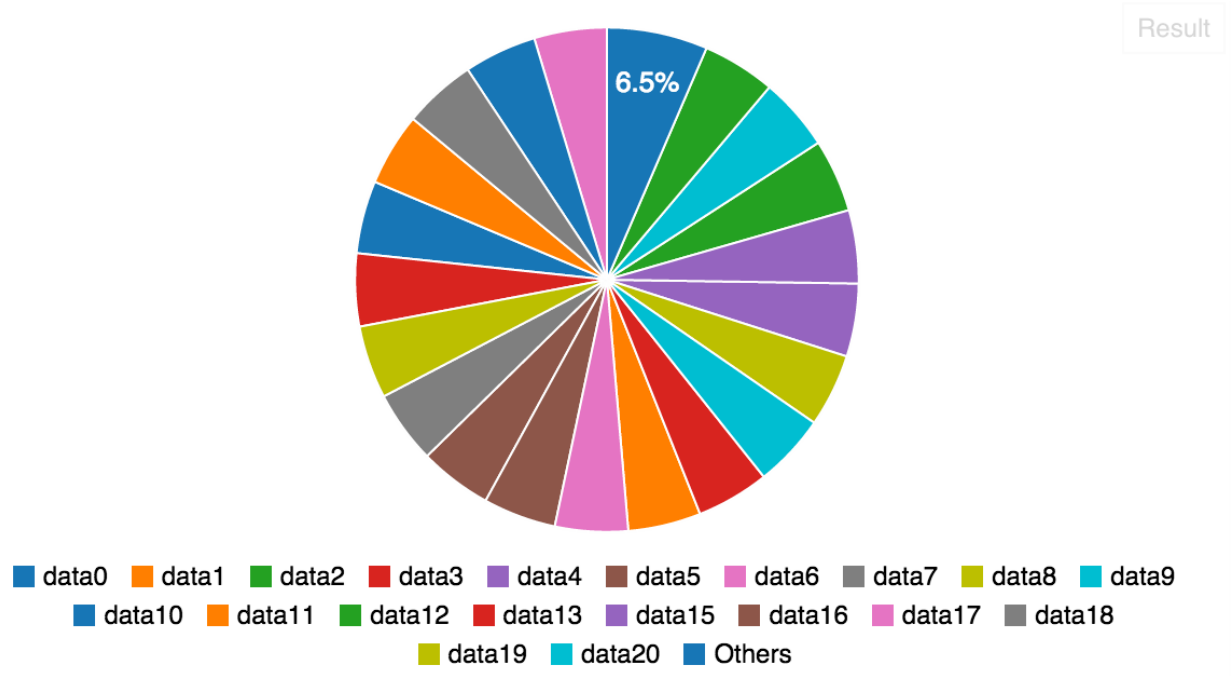
One solution is to process the data set first by limiting the number of items. For example, we can only get top `20` items from the data set and all the rest items are summed into a new item called `Others`. By doing this, the created chart will be more readable.

In Listing 11.6, use `_.sortBy` to sort the `data` array first based on the second element of the item array. Items in the `data` array are all arrays, `[1]` can be used to access the second element in the item array. After the `data` array is sorted, use `_.take` to find the top `20` items in the sorted array, then use `_.last` to find the last one. This last item is used as the threshold to partition the `data` array. Then we use `partition` to divide the `data` array into two groups. The first group `groups[0]` contains items we want to keep, the second group `groups[1]` contains items we want to merge. For the second group, we use `_.sum` to calculate the sum of all the items to merge. The merged item is pushed to the result data array with name `Others` and sum.

Listing 11.6 Use lodash to process data

```
function processData(data) {
  var threshold = _(data).sortBy('[1]').take(20).last();
  if (threshold) {
    var groups = _.partition(data, function(item) {
      return item[1] >= threshold[1];
    });
    if (_.size(groups[1]) > 0) {
      groups[0].push(['Others', _.sum(groups[1], '[1]')]);
    }
    return groups[0];
  }
  return data;
}
```

After using Listing 11.6 code to process the data first, the chart is much easier to read, see below.



Pie chart with items merged

5.4 Create a unique array of objects

5.4.1 Scenario

Given an array of objects in Listing 11.7, remove duplicate values from the array.

Listing 11.7 An array of objects with duplicate values

```
[
  {
    "name": "Alex",
    "age": 30
  },
  {
    "name": "Bob",
    "age": 28
  },
  {
    "name": "Alex",
    "age": 30
  }
]
```

5.4.2 Solution

`_.uniq` and `uniqBy` functions can be used to remove duplicate values from an array, but it only uses *SameAsZero* algorithm to compare values. To perform the deep comparison for elements in the array of Listing 11.7, we need to convert each element into a single value. For example, if the property `name` is the unique key for each element, use `_.uniqBy(array, 'name')`. If there is no unique key, you can convert the element into a JSON string.

Listing 11.8 Compare array elements as JSON strings

```
_.uniqBy(array, element => JSON.stringify(element));
```

JSON serialization may generate different results for objects with the same value due to the undermined property enumeration order. For a more consistent result, we should create our own object serialization format. In Listing 11.9, we concatenate `name` and `age` properties as the serialization format to determine uniqueness.

Listing 11.9 Compare array elements using custom serialization format

```
_.uniq(array, element => element.name + element.age);
```

5.5 Convert an array to an object

5.5.1 Scenario

Given an array of objects with IDs, convert the array to an object with IDs as the keys and array elements as the values.

For example, given an array in Listing 11.10, convert it to an object shown in Listing 11.11.

List 11.10 An array of objects with IDs

```
[
  {
    "id": "user001",
    "name": "Alex"
  },
  {
    "id": "user002",
    "name": "Bob"
  }
]
```

List 11.11 Conversion result of Listing 11.10

```
{
  "user001": {
    "id": "user001",
    "name": "Alex"
  },
  "user002": {
    "id": "user002",
    "name": "Bob"
  }
}
```

5.5.2 Solution

One solution is to use `_.each` to iterate the array and set each property in the result object, see Listing 11.12.

List 11.12 Solution to use `_.each`

```
let result = {};
_.each(array, function(obj) {
  result[obj.id] = obj;
});
```

A better solution is to use `_.reduce`, see Listing 11.13.

List 11.13 Solution to use `_.reduce`

```
_.reduce(array, function(result, obj) {
  result[obj.id] = obj;
  return result;
}, {});
```

6. Thank you

Thank you for reading sample chapters of this book. You can purchase the complete book at Leanpub[1].

[1] https://leanpub.com/lodashcookbook