Версия: 29.7

Mock-функции

Имитационные функции позволяют вам тестировать связи между кодом путем удаления фактической реализации функции, захвата вызовов функции (и параметров, передаваемых в этих вызовах), захвата экземпляров функций конструктора при создании экземпляра с помощью `new` и разрешения настройки возвращаемых значений во время тестирования.

Существует два способа создания макета функции: либо путем создания макетной функции для использования в тестовом коде, либо путем написания `manual mock` для переопределения зависимости модуля.

Использование mock-функции

Давайте представим, что мы тестируем реализацию функции `forEach`, которая выполняет обратный вызов для каждого элемента предоставленного массива.

forEach.js

```
export function forEach(items, callback) {
  for (let index = 0; index < items.length; index++) {
    callback(items[index]);
  }
}
```

Чтобы протестировать эту функцию, мы можем использовать мок-функцию, и посмотреть на состояние мока чтобы убедиться, что функция была вызвана как ожидалось.

forEach.test.js

```
const forEach = require('./forEach');

const mockCallback = jest.fn(x => 42 + x);

test('forEach mock function', () => {
  forEach([0, 1], mockCallback);

  // The mock function was called twice
  expect(mockCallback.mock.calls).toHaveLength(2);
```

```
  // The first argument of the first call to the function was 0
  expect(mockCallback.mock.calls[0][0]).toBe(0);

  // The first argument of the second call to the function was 1
  expect(mockCallback.mock.calls[1][0]).toBe(1);

  // The return value of the first call to the function was 42
  expect(mockCallback.mock.results[0].value).toBe(42);
});
```

`.mock` свойство

У всех мок-функций есть особое свойство `.mock`, где хранятся данные о том как функция была вызвана и что она вернула. Свойство `.mock` также отслеживает значение `this` для каждого вызова, так что как правило это можно посмотреть:

```
const myMock1 = jest.fn();
const a = new myMock1();
console.log(myMock1.mock.instances);
// > [ <a> ]

const myMock2 = jest.fn();
const b = {};
const bound = myMock2.bind(b);
bound();
console.log(myMock2.mock.contexts);
// > [ <b> ]
```

Эти свойства мока очень полезны в тестах чтобы указывать как эти функции вызываются, наследуются, или что они возвращают:

```
// The function was called exactly once
expect(someMockFunction.mock.calls).toHaveLength(1);

// The first arg of the first call to the function was 'first arg'
expect(someMockFunction.mock.calls[0][0]).toBe('first arg');

// The second arg of the first call to the function was 'second arg'
expect(someMockFunction.mock.calls[0][1]).toBe('second arg');

// The return value of the first call to the function was 'return value'
expect(someMockFunction.mock.results[0].value).toBe('return value');

// The function was called with a certain `this` context: the `element` object.
expect(someMockFunction.mock.contexts[0]).toBe(element);

// This function was instantiated exactly twice
```

```
expect(someMockFunction.mock.instances.length).toBe(2);

// The object returned by the first instantiation of this function
// had a `name` property whose value was set to 'test'
expect(someMockFunction.mock.instances[0].name).toBe('test');

// The first argument of the last call to the function was 'test'
expect(someMockFunction.mock.lastCall[0]).toBe('test');
```

Значения возвращаемые имитаторами

Мок-функции также могут использоваться для внедрения тестовых значений в ваш код во время тестирования:

```
const myMock = jest.fn();
console.log(myMock());
// > undefined

myMock.mockReturnValueOnce(10).mockReturnValueOnce('x').mockReturnValue(true);

console.log(myMock(), myMock(), myMock(), myMock());
// > 10, 'x', true, true
```

Имитационные функции также очень эффективны в коде, использующем функциональный стиль передачи продолжения. Код, написанный в этом стиле, помогает избежать необходимости в сложных заглушках, которые воссоздают поведение реального компонента, за который они выступают, в пользу введения значений непосредственно в тест непосредственно перед их использованием.

```
const filterTestFn = jest.fn();

// Make the mock return `true` for the first call,
// and `false` for the second call
filterTestFn.mockReturnValueOnce(true).mockReturnValueOnce(false);

const result = [11, 12].filter(num => filterTestFn(num));

console.log(result);
// > [11]
console.log(filterTestFn.mock.calls[0][0]); // 11
console.log(filterTestFn.mock.calls[1][0]); // 12
```

Большинство реальных примеров на самом деле включают получение доступа к макетной функции зависимого компонента и ее настройку, но техника та же. В этих случаях постарайтесь избежать соблазна реализовать логику внутри любой функции, которая непосредственно не тестируется.

Мокинг (имитации) модулей

Предположим, у нас есть класс, который получает пользователей из нашего API. Этот класс использует axios для вызова API, а затем возвращает атрибут `data`, который содержит всех пользователей:

users.js

```
import axios from 'axios';

class Users {
  static all() {
    return axios.get('/users.json').then(resp => resp.data);
  }
}

export default Users;
```

Теперь, чтобы протестировать этот метод, фактически не обращаясь к API (и, таким образом, создавая медленные и хрупкие тесты), мы можем использовать `jest.mock(...)` функцию для автоматического моделирования модуля axios.

Как только мы создадим макет модуля, мы можем предоставить `mockResolvedValue` for `.get`, который возвращает данные, которые мы хотим, чтобы наш тест подтверждал. По сути, мы говорим, что хотим `axios.get('/users.json')` вернуть поддельный ответ.

users.test.js

```
import axios from 'axios';
import Users from './users';

jest.mock('axios');

test('should fetch users', () => {
  const users = [{name: 'Bob'}];
  const resp = {data: users};
  axios.get.mockResolvedValue(resp);

  // or you could use the following depending on your use case:
  // axios.get.mockImplementation(() => Promise.resolve(resp))

  return Users.all().then(data => expect(data).toEqual(users));
});
```

Насмешливые части

Подмножества модуля могут быть подделаны, а остальная часть модуля может сохранить их фактическую реализацию:

foo-bar-baz.js

```
export const foo = 'foo';
export const bar = () => 'bar';
export default () => 'baz';
```

```
//test.js
import defaultExport, {bar, foo} from '../foo-bar-baz';

jest.mock('../foo-bar-baz', () => {
  const originalModule = jest.requireActual('../foo-bar-baz');

  //Mock the default export and named export 'foo'
  return {
    __esModule: true,
    ...originalModule,
    default: jest.fn(() => 'mocked baz'),
    foo: 'mocked foo',
  };
});

test('should do a partial mock', () => {
  const defaultExportResult = defaultExport();
  expect(defaultExportResult).toBe('mocked baz');
  expect(defaultExport).toHaveBeenCalled();

  expect(foo).toBe('mocked foo');
  expect(bar()).toBe('bar');
});
```

Реализации имитаторов

Тем не менее, есть случаи, когда полезно выйти за рамки возможности указывать возвращаемые значения и полностью заменить реализацию макетной функции. Это можно сделать с помощью `jest.fn` или `mockImplementationOnce` метода для mock-функций.

```
const myMockFn = jest.fn(cb => cb(null, true));
```

```
myMockFn((err, val) => console.log(val));
// > true
```

`mockImplementation` Метод полезен, когда вам нужно определить реализацию по умолчанию для mock-функции, созданной из другого модуля:

foo.js

```
module.exports = function () {
  // some implementation;
};
```

test.js

```
jest.mock('../foo'); // this happens automatically with automocking
const foo = require('../foo');

// foo is a mock function
foo.mockImplementation(() => 42);
foo();
// > 42
```

Когда вам нужно воссоздать сложное поведение макетной функции таким образом, чтобы несколько вызовов функции приводили к разным результатам, используйте `mockImplementationOnce` метод:

```
const myMockFn = jest
  .fn()
  .mockImplementationOnce(cb => cb(null, true))
  .mockImplementationOnce(cb => cb(null, false));

myMockFn((err, val) => console.log(val));
// > true

myMockFn((err, val) => console.log(val));
// > false
```

Когда у издевающейся функции заканчиваются реализации, определенные с помощью `mockImplementationOnce`, она будет выполнять реализацию по умолчанию, установленную с помощью `jest.fn` (если она определена):

```
const myMockFn = jest
  .fn(() => 'default')
  .mockImplementationOnce(() => 'first call')
  .mockImplementationOnce(() => 'second call');
```

```
console.log(myMockFn(), myMockFn(), myMockFn(), myMockFn());
// > 'first call', 'second call', 'default', 'default'
```

Для случаев, когда у нас есть методы, которые обычно связаны цепочкой (и поэтому их всегда нужно возвращать `this`), у нас есть sugary API для упрощения этого в виде `.mockReturnThis()` функции, которая также используется во всех макетах:

```
const myObj = {
  myMethod: jest.fn().mockReturnThis(),
};

// is the same as

const otherObj = {
  myMethod: jest.fn(function () {
    return this;
  }),
};
```

Пародийные имена

При желании вы можете указать имя для своих макетных функций, которое будет отображаться вместо `'jest.fn()'` в выводе тестовой ошибки. Используйте `.mockName()`, если хотите иметь возможность быстро идентифицировать фиктивную функцию, сообщающую об ошибке в результатах вашего теста.

```
const myMockFn = jest
  .fn()
  .mockReturnValue('default')
  .mockImplementation(scalar => 42 + scalar)
  .mockName('add42');
```

Пользовательские матчеры

Наконец, чтобы упростить определение того, как были вызваны фиктивные функции, мы добавили для вас несколько пользовательских функций сопоставления:

```
// The mock function was called at least once
expect(mockFunc).toHaveBeenCalled();

// The mock function was called at least once with the specified args
expect(mockFunc).toHaveBeenCalledWith(arg1, arg2);
```

```
// The last call to the mock function was called with the specified args
expect(mockFunc).toHaveBeenLastCalledWith(arg1, arg2);

// All calls and the name of the mock is written as a snapshot
expect(mockFunc).toMatchSnapshot();
```

Эти программы сопоставления используются для обычных форм проверки `.mock` собственности. Вы всегда можете сделать это вручную самостоятельно, если это вам больше по вкусу или если вам нужно сделать что-то более конкретное:

```
// The mock function was called at least once
expect(mockFunc.mock.calls.length).toBeGreaterThan(0);

// The mock function was called at least once with the specified args
expect(mockFunc.mock.calls).toContainEqual([arg1, arg2]);

// The last call to the mock function was called with the specified args
expect(mockFunc.mock.calls[mockFunc.mock.calls.length - 1]).toEqual([
  arg1,
  arg2,
]);

// The first arg of the last call to the mock function was `42`
// (note that there is no sugar helper for this specific of an assertion)
expect(mockFunc.mock.calls[mockFunc.mock.calls.length - 1][0]).toBe(42);

// A snapshot will check that a mock was invoked the same number of times,
// in the same order, with the same arguments. It will also assert on the name.
expect(mockFunc.mock.calls).toEqual([[arg1, arg2]]);
expect(mockFunc.getMockName()).toBe('a mock name');
```

Для ознакомления с полным списком сопоставлений, обратите внимание на справочную документацию.