Версия: 29.7

Класс ES6 издевается

Jest можно использовать для издевательства над классами ES6, которые импортируются в файлы, которые вы хотите протестировать.

Классы ES6 - это функции-конструкторы с некоторым синтаксическим сахаром. Следовательно, любой макет для класса ES6 должен быть функцией или фактическим классом ES6 (который, опять же, является другой функцией). Таким образом, вы можете издеваться над ними, используя фиктивные функции.

Пример класса ES6

Мы будем использовать надуманный пример класса, который воспроизводит звуковые файлы, `SoundPlayer` и потребительского класса, который использует этот класс, `SoundPlayerConsumer`. Мы будем издеваться `SoundPlayer` в наших тестах для `SoundPlayerConsumer`.

sound-player.js

```
export default class SoundPlayer {
  constructor() {
    this.foo = 'bar';
  }

  playSoundFile(fileName) {
    console.log('Playing sound file ' + fileName);
  }
}
```

sound-player-consumer.js

```
import SoundPlayer from './sound-player';

export default class SoundPlayerConsumer {
  constructor() {
    this.soundPlayer = new SoundPlayer();
  }

  playSomethingCool() {
    const coolSoundFileName = 'song.mp3';
    this.soundPlayer.playSoundFile(coolSoundFileName);
```

```
  }
}
```

4 способа создания макета класса ES6

Автоматическая насмешка

Вызов `jest.mock('./sound-player')` возвращает полезный "автоматический макет", который вы можете использовать для слежки за вызовами конструктора класса и всех его методов. Он заменяет класс ES6 фиктивным конструктором и заменяет все его методы фиктивными функциями, которые всегда возвращают `undefined`. Вызовы методов сохраняются в `theAutomaticMock.mock.instances[index].methodName.mock.calls`.

> **ПРИМЕЧАНИЕ**
>
> Если вы используете функции со стрелками в своих классах, они *не* будут частью макета. Причина этого в том, что функции arrow отсутствуют в прототипе объекта, это просто свойства, содержащие ссылку на функцию.

Если вам не нужно заменять реализацию класса, это самый простой вариант для настройки. Например:

```
import SoundPlayer from './sound-player';
import SoundPlayerConsumer from './sound-player-consumer';
jest.mock('./sound-player'); // SoundPlayer is now a mock constructor

beforeEach(() => {
  // Clear all instances and calls to constructor and all methods:
  SoundPlayer.mockClear();
});

it('We can check if the consumer called the class constructor', () => {
  const soundPlayerConsumer = new SoundPlayerConsumer();
  expect(SoundPlayer).toHaveBeenCalledTimes(1);
});

it('We can check if the consumer called a method on the class instance', () => {
  // Show that mockClear() is working:
  expect(SoundPlayer).not.toHaveBeenCalled();

  const soundPlayerConsumer = new SoundPlayerConsumer();
  // Constructor should have been called again:
  expect(SoundPlayer).toHaveBeenCalledTimes(1);

  const coolSoundFileName = 'song.mp3';
  soundPlayerConsumer.playSomethingCool();
```

```
  // mock.instances is available with automatic mocks:
  const mockSoundPlayerInstance = SoundPlayer.mock.instances[0];
  const mockPlaySoundFile = mockSoundPlayerInstance.playSoundFile;
  expect(mockPlaySoundFile.mock.calls[0][0]).toBe(coolSoundFileName);
  // Equivalent to above check:
  expect(mockPlaySoundFile).toHaveBeenCalledWith(coolSoundFileName);
  expect(mockPlaySoundFile).toHaveBeenCalledTimes(1);
});
```

Ручная насмешка

Создайте макет вручную, сохранив макет реализации в `__mocks__` папке. Это позволяет вам указать реализацию, и ее можно использовать во всех тестовых файлах.

__mocks__/sound-player.js

```
// Import this named export into your test file:
export const mockPlaySoundFile = jest.fn();
const mock = jest.fn().mockImplementation(() => {
  return {playSoundFile: mockPlaySoundFile};
});

export default mock;
```

Импортируйте макет и метод mock, общий для всех экземпляров:

sound-player-consumer.test.js

```
import SoundPlayer, {mockPlaySoundFile} from './sound-player';
import SoundPlayerConsumer from './sound-player-consumer';
jest.mock('./sound-player'); // SoundPlayer is now a mock constructor

beforeEach(() => {
  // Clear all instances and calls to constructor and all methods:
  SoundPlayer.mockClear();
  mockPlaySoundFile.mockClear();
});

it('We can check if the consumer called the class constructor', () => {
  const soundPlayerConsumer = new SoundPlayerConsumer();
  expect(SoundPlayer).toHaveBeenCalledTimes(1);
});

it('We can check if the consumer called a method on the class instance', () => {
  const soundPlayerConsumer = new SoundPlayerConsumer();
  const coolSoundFileName = 'song.mp3';
  soundPlayerConsumer.playSomethingCool();
```

```
  expect(mockPlaySoundFile).toHaveBeenCalledWith(coolSoundFileName);
});
```

Вызов `jest.mock()` с заводским параметром модуля

`jest.mock(path, moduleFactory)` принимает аргумент **фабрики модулей**. Фабрика модулей - это функция, которая возвращает макет.

Чтобы издеваться над функцией-конструктором, фабрика модулей должна возвращать функцию-конструктор. Другими словами, фабрика модулей должна быть функцией, которая возвращает функцию - функцию более высокого порядка (HOF).

```
import SoundPlayer from './sound-player';
const mockPlaySoundFile = jest.fn();
jest.mock('./sound-player', () => {
  return jest.fn().mockImplementation(() => {
    return {playSoundFile: mockPlaySoundFile};
  });
});
```

> **⚠ ОСТОРОЖНО**
>
> Поскольку вызовы `jest.mock()` помещаются в начало файла, Jest предотвращает доступ к переменным, находящимся вне области видимости. По умолчанию вы не можете сначала определить переменную, а затем использовать ее на заводе. Jest отключит эту проверку для переменных, начинающихся со слова `mock`. Однако вы по-прежнему должны гарантировать, что они будут инициализированы вовремя. Помните о временной мертвой зоне.

Например, следующее приведет к ошибке вне области видимости из-за использования `fake` вместо `mock` в объявлении переменной.

```
// Note: this will fail
import SoundPlayer from './sound-player';
const fakePlaySoundFile = jest.fn();
jest.mock('./sound-player', () => {
  return jest.fn().mockImplementation(() => {
    return {playSoundFile: fakePlaySoundFile};
  });
});
```

Следующее приведет к появлению а, `ReferenceError` несмотря на использование `mock` в объявлении переменной, поскольку `mockSoundPlayer` она не заключена в функцию со стрелкой и, следовательно, доступна до инициализации после подъема.

```
import SoundPlayer from './sound-player';
const mockSoundPlayer = jest.fn().mockImplementation(() => {
  return {playSoundFile: mockPlaySoundFile};
});
// results in a ReferenceError
jest.mock('./sound-player', () => {
  return mockSoundPlayer;
});
```

Замена издевательства с помощью `mockImplementation()` или `mockImplementationOnce()`

Вы можете заменить все вышеперечисленные макеты, чтобы изменить реализацию для одного теста или всех тестов, вызвав `mockImplementation()` существующий макет.

Вызовы jest.mock помещаются в начало кода. Вы можете указать макет позже, например, в `beforeAll()`, вызвав `mockImplementation()` (или `mockImplementationOnce()`) существующий макет вместо использования заводского параметра. Это также позволяет вам изменять макет между тестами, если это необходимо:

```
import SoundPlayer from './sound-player';
import SoundPlayerConsumer from './sound-player-consumer';

jest.mock('./sound-player');

describe('When SoundPlayer throws an error', () => {
  beforeAll(() => {
    SoundPlayer.mockImplementation(() => {
      return {
        playSoundFile: () => {
          throw new Error('Test error');
        },
      };
    });
  });

  it('Should throw an error when calling playSomethingCool', () => {
    const soundPlayerConsumer = new SoundPlayerConsumer();
    expect(() => soundPlayerConsumer.playSomethingCool()).toThrow();
  });
});
```

Подробнее: Понимание функций макетного конструктора

Создание макета функции конструктора с использованием `jest.fn().mockImplementation()` делает макеты более сложными, чем они есть на самом деле. В этом разделе показано, как вы можете создавать свои собственные макеты, чтобы проиллюстрировать, как работает макетирование.

Ручная насмешка над другим классом ES6

Если вы определите класс ES6, используя то же имя файла, что и издеваемый класс в `__mocks__` папке, он будет использоваться как макет. Этот класс будет использоваться вместо реального класса. Это позволяет вам внедрить тестовую реализацию для класса, но не предоставляет способа следить за вызовами.

Для надуманного примера макет может выглядеть следующим образом:

__mocks__/sound-player.js

```
export default class SoundPlayer {
  constructor() {
    console.log('Mock SoundPlayer: constructor was called');
  }

  playSoundFile() {
    console.log('Mock SoundPlayer: playSoundFile was called');
  }
}
```

Издевается с использованием заводского параметра модуля

Заводская функция модуля, переданная в `jest.mock(path, moduleFactory)`, может быть HOF, которая возвращает функцию *. Это позволит вызывать `new` макет. Опять же, это позволяет вам вводить другое поведение для тестирования, но не предоставляет способа следить за вызовами.

*** Заводская функция модуля должна возвращать функцию**

Чтобы издеваться над функцией-конструктором, фабрика модулей должна возвращать функцию-конструктор. Другими словами, фабрика модулей должна быть функцией, которая возвращает функцию - функцию более высокого порядка (HOF).

```
jest.mock('./sound-player', () => {
  return function () {
    return {playSoundFile: () => {}};
  };
});
```

> **ПРИМЕЧАНИЕ**

Макет не может быть функцией со стрелкой, потому что вызов `new` функции со стрелкой запрещен в JavaScript. Так что это не сработает:

```
jest.mock('./sound-player', () => {
  return () => {
    // Does not work; arrow functions can't be called with new
    return {playSoundFile: () => {}};
  };
});
```

Это вызовет ***ошибку типа: _soundPlayer2.default не является конструктором***, если только код не перенесен в ES5, например, с помощью `@babel/preset-env`. (В ES5 нет ни функций со стрелками, ни классов, поэтому оба будут преобразованы в простые функции.)

Издевается над определенным методом класса

Допустим, вы хотите издеваться или шпионить за методом `playSoundFile` внутри класса `SoundPlayer`. Простой пример:

```
// your jest test file below
import SoundPlayer from './sound-player';
import SoundPlayerConsumer from './sound-player-consumer';

const playSoundFileMock = jest
  .spyOn(SoundPlayer.prototype, 'playSoundFile')
  .mockImplementation(() => {
    console.log('mocked function');
  }); // comment this line if just want to "spy"

it('player consumer plays music', () => {
  const player = new SoundPlayerConsumer();
  player.playSomethingCool();
  expect(playSoundFileMock).toHaveBeenCalled();
});
```

Статические методы получения и установки

Давайте представим, что наш класс `SoundPlayer` имеет метод получения `foo` и статический метод `brand`

```
export default class SoundPlayer {
  constructor() {
    this.foo = 'bar';
  }
```

```
  playSoundFile(fileName) {
    console.log('Playing sound file ' + fileName);
  }

  get foo() {
    return 'bar';
  }
  static brand() {
    return 'player-brand';
  }
}
```

Вы можете легко издеваться над ними / шпионить за ними, вот пример:

```
// your jest test file below
import SoundPlayer from './sound-player';

const staticMethodMock = jest
  .spyOn(SoundPlayer, 'brand')
  .mockImplementation(() => 'some-mocked-brand');

const getterMethodMock = jest
  .spyOn(SoundPlayer.prototype, 'foo', 'get')
  .mockImplementation(() => 'some-mocked-result');

it('custom methods are called', () => {
  const player = new SoundPlayer();
  const foo = player.foo;
  const brand = SoundPlayer.brand();

  expect(staticMethodMock).toHaveBeenCalled();
  expect(getterMethodMock).toHaveBeenCalled();
});
```

Отслеживание использования (слежка за макетом)

Внедрение тестовой реализации полезно, но вы, вероятно, также захотите проверить, вызываются ли конструктор класса и методы с правильными параметрами.

Шпионит за конструктором

Чтобы отслеживать вызовы конструктора, замените функцию, возвращаемую HOF, на Jest mock function. Создайте ее с помощью `jest.fn()`, а затем укажите ее реализацию с помощью `mockImplementation()`.

```
import SoundPlayer from './sound-player';
jest.mock('./sound-player', () => {
  // Works and lets you check for constructor calls:
  return jest.fn().mockImplementation(() => {
    return {playSoundFile: () => {}};
  });
});
```

Это позволит нам проверить использование нашего издеваемого класса, используя `SoundPlayer.mock.calls`: `expect(SoundPlayer).toHaveBeenCalled();` или почти эквивалентный: `expect(SoundPlayer.mock.calls.length).toBeGreaterThan(0);`

Издевается над экспортом класса не по умолчанию

Если класс **не** экспортируется из модуля по умолчанию, то вам нужно вернуть объект с ключом, который совпадает с именем экспорта класса.

```
import {SoundPlayer} from './sound-player';
jest.mock('./sound-player', () => {
  // Works and lets you check for constructor calls:
  return {
    SoundPlayer: jest.fn().mockImplementation(() => {
      return {playSoundFile: () => {}};
    }),
  };
});
```

Шпионит за методами нашего класса

Наш издеваемый класс должен будет предоставить любые функции-члены (`playSoundFile` в примере), которые будут вызываться во время наших тестов, иначе мы получим ошибку при вызове функции, которая не существует. Но мы, вероятно, захотим также проследить за вызовами этих методов, чтобы убедиться, что они были вызваны с ожидаемыми параметрами.

Новый объект будет создаваться каждый раз, когда во время тестов вызывается функция mock constructor. Чтобы отслеживать вызовы методов во всех этих объектах, мы заполняем `playSoundFile` другую фиктивную функцию и сохраняем ссылку на ту же фиктивную функцию в нашем тестовом файле, чтобы она была доступна во время тестов.

```
import SoundPlayer from './sound-player';
const mockPlaySoundFile = jest.fn();
jest.mock('./sound-player', () => {
  return jest.fn().mockImplementation(() => {
    return {playSoundFile: mockPlaySoundFile};
    // Now we can track calls to playSoundFile
```

```
  });
});
```

Ручной макет, эквивалентный этому, был бы:

__mocks__/sound-player.js

```
// Import this named export into your test file
export const mockPlaySoundFile = jest.fn();
const mock = jest.fn().mockImplementation(() => {
  return {playSoundFile: mockPlaySoundFile};
});

export default mock;
```

Использование аналогично заводской функции модуля, за исключением того, что вы можете опустить второй аргумент из `jest.mock()`, и вы должны импортировать издевательский метод в свой тестовый файл, поскольку он там больше не определен. Используйте для этого исходный путь к модулю; не включайте `__mocks__`.

Уборка между тестами

Чтобы очистить запись о вызовах функции-конструктора-макета и ее методов, мы вызываем `mockClear()` в `beforeEach()` функции:

```
beforeEach(() => {
  SoundPlayer.mockClear();
  mockPlaySoundFile.mockClear();
});
```

Полный пример

Вот полный тестовый файл, в котором используется заводской параметр модуля для `jest.mock`:

sound-player-consumer.test.js

```
import SoundPlayer from './sound-player';
import SoundPlayerConsumer from './sound-player-consumer';

const mockPlaySoundFile = jest.fn();
jest.mock('./sound-player', () => {
  return jest.fn().mockImplementation(() => {
    return {playSoundFile: mockPlaySoundFile};
  });
```

```
});

beforeEach(() => {
  SoundPlayer.mockClear();
  mockPlaySoundFile.mockClear();
});

it('The consumer should be able to call new() on SoundPlayer', () => {
  const soundPlayerConsumer = new SoundPlayerConsumer();
  // Ensure constructor created the object:
  expect(soundPlayerConsumer).toBeTruthy();
});

it('We can check if the consumer called the class constructor', () => {
  const soundPlayerConsumer = new SoundPlayerConsumer();
  expect(SoundPlayer).toHaveBeenCalledTimes(1);
});

it('We can check if the consumer called a method on the class instance', () => {
  const soundPlayerConsumer = new SoundPlayerConsumer();
  const coolSoundFileName = 'song.mp3';
  soundPlayerConsumer.playSomethingCool();
  expect(mockPlaySoundFile.mock.calls[0][0]).toBe(coolSoundFileName);
});
```


*Последнее обновление от **12 сентября 2023** года **от Симэна Бекхуса в 2023 году***