25.09.2020

Unit Test Recorder - Автоматически генерирует модульные тесты в процессе использования вашего приложения

Введение

Это описание для пакета npm unit-test-recorder или UTR для краткости.

Пакет представляет собой инструмент CLI, который позволяет пользователю записывать действия, проходящие через каждую функцию в приложении nodejs. Затем это может быть преобразовано в рабочие случаи модульного тестирования с помощью интерполяции строк.

Этот инструмент предназначен для регрессионного тестирования с использованием снимков.

Сгенерированные снимки представляют собой читаемый и редактируемый человеком код. Пользователям рекомендуется взять на себя инициативу и внести изменения непосредственно в этот код.

Ввод кода слева и генерированный код справа.

Методология

В тестировании мы передаем некоторые аргументы функции, которую тестируем, и ожидаем в ответ некоторый результат. Если мы уверены, что тестируемая функция стабильна, то мы можем записать параметры и результат в качестве истинного значения.

Для записи параметров и результатов нам нужно внести некоторые изменения в сам код. Этот процесс называется инструментированием. Инструменты используются во многих местах, включая запускатели покрытия кода, такие как istanbuljs, и службы мониторинга, такие как new relic.

UTR использует babeljs для внедрения регистратора в код.

Запись функций

Возьмем эту функцию в качестве примера

```
const foo = (a, b) => a + b;
```

UTR использует babel для того, чтобы явно инструментировать его для

```
const foo = (...p) => recorderWrapper(
  { name: 'foo', fileName: 'bar' },
  (a, b) => a + b,
  ...p,
);
```

Функция recorderWrapper предоставляется UTR. Функция ниже - это упрощенная версия ее реализации.

```
const recorderWrapper = (meta, fn, ...p) => {
  const result = fn(...p);
  Recorder.record({ meta, result, params: p });
  return result;
};
```

Свойство meta хранит дополнительную информацию о характере функции. Например, имя файла, где располагается функция, возвращает ли она обещание, является ли она экспортом по умолчанию или именованным экспортом и т.д.

Эта функция записывает все вызовы этой функции в состоянии. Вот упрощенное представление состояния.

```
{
  "fileName":{
    "functionName":{
      "captures":[
        { "params": [1, 2], "result": 3 },
        { "params": [2, 3], "result": 5 },
      ]
    }
  }
}
```

Теперь, используя интерполяцию строк, мы можем генерировать тестовые случаи.

```
describe('fileName', () => {
  describe('functionName', () => {
    it('test 1', () => {
      expect(foo(1, 2)).toEqual(3);
    });
    it('test 2', () => {
      expect(foo(2, 3)).toEqual(5);
    });
  });
});
```

Запись моков

Для записи моков нам нужно записать все вызовы импортированной функции и связать их с функцией, которую мы в данный момент записываем.

```
const fs = require('fs');
const foo = (fileName) => fs.readFileSync(fileName).toString().toUpperCase();
const bar = (fileName) => fs.readFileSync(fileName).toString().toLowerCase();
const baz = (f1, f2) => foo(f1) + bar(f2);
```

Проблема в том, что если мы инструментируем имитируемую функцию наивно, у нас не будет способа узнать, какая функция вызвала имитируемую функцию.

```
fs.wrappedReadFileSync = (...p) => recorderWrapper(
  { name: 'fs.readFileSync' },
  fs.readFileSync,
  ...p,
);
const foo = (...p) => recorderWrapper(
  { name: 'foo' },
  (fileName) => fs.wrappedReadFileSync(fileName).toString().toUpperCase(),
  ...p,
);
const bar = (...p) => recorderWrapper(
  { name: 'bar' },
  (fileName) => fs.wrappedReadFileSync(fileName).toString().toUpperCase(),
  ...p,
);
const baz = (...p) => recorderWrapper(
  { name: 'baz' },
  (f1, f2) => foo(f1) + bar(f2),
  ...p,
```

);

Поэтому recorderWrapper для fs.readFileSync должен быть осведомлен о стеке вызовов JavaScript или продолжении. Локальное хранилище продолжений - это популярная библиотека JavaScript, созданная для этой цели. Однако в nodejs 13 и более поздних версиях существует встроенная поддержка в виде AsyncLocalStorage.

Для иллюстрации мы теперь используем два отдельных оболочки mockRecorderWrapper и functionRecorderWrapper.

```
const functionRecorderWrapper = (meta, fn, ...p) => {
  // Установите мета-информацию таким образом, чтобы все вызываемые функции имели к ней доступ
  cls.set('meta', meta);
  return recorderWrapper(meta, fn, ...p);
};
const mockRecorderWrapper = (mockMeta, fn, ...p) => {
  // Получите метаданные функции, которая вызвала этот мок
  const invokerMeta = cls.get('meta');
  // Используйте метаданные вызывающей функции, чтобы убедиться, что записи правильно коррелируются.
  return recorderWrapper({ ...mockMeta, ...invokerMeta }, fn, ...p);
};
```

Теперь, если имитируемая функция вызывается несколько раз, мы можем сохранить последовательность в массиве.

```
{
  "foo": [
    {"moduleName": "fs", "functionName": "readFileSync", "params": [ "file1" ], "result": "file1_contents" },
  ],
  "bar": [
    {"moduleName": "fs", "functionName": "readFileSync", "params": [ "file1" ], "result": "file1_contents" },
  ],
  "baz": [
    {"moduleName": "fs", "functionName": "readFileSync", "params": [ "file1" ], "result": "file1_contents" },
    {"moduleName": "fs", "functionName": "readFileSync", "params": [ "file2" ], "result": "file2_contents" },
  ]
}
```

Сгенерированные тесты выглядят так

```
const fs = require('fs');
jest.mock('fs');
describe('fileName', () => {
  describe('baz', () => {
    it('test 1', () => {
      fs.readFileSync.mockImplementationOnce('file1_contents');
      fs.readFileSync.mockImplementationOnce('file2_contents');
      // baz вызывает foo и bar. В свою очередь, они используют моки.
      expect(baz('file1', 'file2')).toEqual('FILE1_CONTENTSfile2_contents');
    });
  });
});
```

Внедрение зависимостей

Другой распространенный шаблон программирования в JavaScript включает использование внедрения зависимостей. Часто клиенты баз данных передаются функциям, как в приведенном ниже примере.

```
const foo = async (dbClient) => {
  const rows = await dbClient.query('SELECT COUNT(*) as usercount FROM users;');
  return rows[0].usercount;
};
```

Чтобы сгенерировать тесты для функции foo, нам нужно записать функцию query аргумента dbClient. Однако, если мы изменяем саму функцию dbClient.query, это может привести к нежелательным побочным эффектам. Вместо этого мы создаем новую функцию во время выполнения и используем babel для изменения исходного кода, чтобы убедиться, что будет использоваться новая созданная функция.

```
const foo = async (dbClient) => {
  const rows = await dbClient.wrappedQuery('SELECT COUNT(*) as usercount FROM users;');
  return rows[0].usercount;
};
```

Код ниже - это упрощенная версия реальной реализации.

```
// Выполняется перед выполнением функции.
const preExecutionHook = (wrappingMeta) => {
  // wrappingMeta выглядит подобно { objectName: 'dbClient', fnName: 'query' };
  const { fnName } = wrappingMeta;
  const newFnName = generateNewFnName(wrappingMeta); // "wrappedQuery" в этом случае
  // dbClient.wrappedQuery создан во время выполнения
  dbClient[newFnName] = (...p) => {
    const invokerMeta = cls.get('meta');
    const fn = dbClient[fnName];
    return recorderWrapper({ ...wrappingMeta, ...invokerMeta }, fn, ...p);
  };
};
```

В сгенерированных тестах мы имитируем методы внедренного объекта, как показано ниже

```
describe('fileName', () => {
  describe('foo', () => {
    it('test 1', () => {
      const dbClient = {};
      dbClient.query.mockImplementationOnce([{ usercount: 20 }]);
      expect(foo(dbClient)).toEqual(20);
    });
  });
});
```

Продвижения внедрений

Чтобы процесс генерации тестов был эффективным с точки зрения данных, мы используем продвижение внедрений. В приведенном ниже примере, даже если функция bar никогда не вызывается отдельно, мы можем использовать вызов foo для генерации тестов как для foo, так и для bar.

```
const bar = (client, status) => client.query('SELECT COUNT(*) FROM users WHERE status=$1', status);
const foo = async (dbClient) => {
  const offlineUsers = await bar(dbClient, 'offline');
  const activeUsers = await dbClient.query('SELECT COUNT(*) FROM users WHERE status=$1', 'active');
  const awayUsers = await bar(dbClient, 'away');
  return { offlineUsers, activeUsers, awayUsers };
};
```

Хотя тестирование foo также охватывает bar, отдельные тесты для bar выступают в качестве опоры для любых будущих модификаций функции bar. Поэтому это принятое проектировочное решение.

Сгенерированные тесты выглядят так:

```
describe('fileName', () => {
  describe('bar', () => {
    it('test 1', () => {
      const dbClient = {};
      dbClient.query.mockImplementationOnce(10);
      expect(bar(dbClient, 'offline')).toEqual(10);
    });
    it('test 2', () => {
      const dbClient = {};
      dbClient.query.mockImplementationOnce(30);
      expect(bar(dbClient, 'away')).toEqual(30);
    });
  });
  describe('foo', () => {
    it('test 1', () => {
      const dbClient = {};
      dbClient.query.mockImplementationOnce(10);
      dbClient.query.mockImplementationOnce(20);
      dbClient.query.mockImplementationOnce(30);
      const expected = { offlineUsers: 10, activeUsers: 20, awayUsers: 30 };
      expect(foo(dbClient)).toEqual(expected);
    });
  });
});
```

Продвижение внедрений обрабатывается внутренне путем поддержания неявного стека. Когда область видимости функции заканчивается, все записанные внедрения зависимостей передаются ее родителю.

Мы используем uuidv4 для корректной идентификации внедренных функций для продвижения, так как имена переменных могут отличаться между родителем и потомком.

Первый вызов dbClient.query

```
fooCaptures = {
  captures: [], // Функция foo еще не вызывала dbClient.query
};
barCaptures = {
  captures: [
    {
      injectedFunctionUUid: 'uuid1',
      params: [
        'SELECT COUNT(*) FROM users WHERE status=$1',
        'offline',
      ],
      result: 10,
    },
  ],
};

// Когда bar завершается, захваты (captures) переносятся на его родительскую функцию.
fooCaptures = {
  captures: [
    {
      injectedFunctionUUid: 'uuid1',
```

```
      params: [
        'SELECT COUNT(*) FROM users WHERE status=$1',
        'offline',
      ],
      result: 10,
    },
  ],
};
```

Второй вызов dbClient.query

```
// Когда foo напрямую вызывает dbClient.query, она добавляется в массив captures
fooCaptures = {
  captures: [
    {
      injectedFunctionUUid: 'uuid1',
      params: [
        'SELECT COUNT(*) FROM users WHERE status=$1',
        'offline',
      ],
      result: 10,
    },
    {
      injectedFunctionUUid: 'uuid1',
      params: [
        'SELECT COUNT(*) FROM users WHERE status=$1',
        'active',
      ],
      result: 20,
    },
  ],
};
```

Третий вызов dbClient.query

```
fooCaptures = {
  captures: [
    {
      injectedFunctionUUid: 'uuid1',
      params: [
        'SELECT COUNT(*) FROM users WHERE status=$1',
        'offline',
      ],
      result: 10,
    },
    {
      injectedFunctionUUid: 'uuid1',
      params: [
        'SELECT COUNT(*) FROM users WHERE status=$1',
        'active',
      ],
      result: 20,
    },
  ],
};
```

```
barCaptures = {
  captures: [
    {
      injectedFunctionUUid: 'uuid1',
      params: [
        'SELECT COUNT(*) FROM users WHERE status=$1',
        'away',
      ],
      result: 30,
    },
  ],
};

fooCaptures = {
  captures: [
    {
      injectedFunctionUUid: 'uuid1',
      params: [
        'SELECT COUNT(*) FROM users WHERE status=$1',
        'offline',
      ],
      result: 10,
    },
    {
      injectedFunctionUUid: 'uuid1',
      params: [
        'SELECT COUNT(*) FROM users WHERE status=$1',
        'active',
      ],
      result: 20,
    },
    {
      injectedFunctionUUid: 'uuid1',
      params: [
        'SELECT COUNT(*) FROM users WHERE status=$1',
        'away',
      ],
      result: 30,
    },
  ],
};
```

Заключение

Помимо обычных преимуществ тестирования программного обеспечения, это также быстрый и недорогой способ генерирования набора тестового кода, соответствующего корпусу существующего кода. Это может быть использовано исследователями, работающими над инструментами генерации кода на основе глубокого обучения.