テストユーティリティ

インポート

```
import ReactTestUtils from 'react-dom/test-utils'; // ES6
var ReactTestUtils = require('react-dom/test-utils'); // ES5 with npm
```

概要

`ReactTestUtils` はお好みのテストフレームワークで React コンポーネントをテストしやすくするものです。Facebook では快適に JavaScript をテストするために Jest を使用しています。Jest のウェブサイトにある React Tutorial を通して Jest の始め方を学んでください。

> **補足:**
> `react-testing-library` の使用をおすすめします。これは、エンドユーザーがコンポーネントを使用するのと同様の書き方でコンポーネントを使用するテストを書くことを可能にし、かつそれを促進するように設計されています。
> また別の手段として、Airbnb が Enzyme と呼ばれるテストユーティリティをリリースしています。Enzyme は React コンポーネントの出力のアサート、操作、そして横断的な処理を簡単にしてくれます。

- `act()`
- `mockComponent()`
- `isElement()`
- `isElementOfType()`
- `isDOMComponent()`
- `isCompositeComponent()`
- `isCompositeComponentWithType()`
- `findAllInRenderedTree()`
- `scryRenderedDOMComponentsWithClass()`
- `findRenderedDOMComponentWithClass()`
- `scryRenderedDOMComponentsWithTag()`
- `findRenderedDOMComponentWithTag()`
- `scryRenderedComponentsWithType()`
- `findRenderedComponentWithType()`
- `renderIntoDocument()`
- `Simulate`

Reference

`act()`

アサーション用のコンポーネントを準備するために、それをレンダーして更新を実行するコードを `act()` でラップします。これにより、テストはブラウザでの React の動作により近い状態で実行されます。

> **補足**
> `react-test-renderer` を使っている場合、それはこのメソッドと同じように振舞う `act` エクスポートも提供します。

例えば、次のような `Counter` コンポーネントがあるとしましょう：

```
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {count: 0};
    this.handleClick = this.handleClick.bind(this);
  }
  componentDidMount() {
```

```
    document.title = `You clicked ${this.state.count} times`;
  }
  componentDidUpdate() {
    document.title = `You clicked ${this.state.count} times`;
  }
  handleClick() {
    this.setState(state => ({
      count: state.count + 1,
    }));
  }
  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={this.handleClick}>
          Click me
        </button>
      </div>
    );
  }
}
```

これをテストするには次のように書きます：

```
import React from 'react';
import ReactDOM from 'react-dom';
import { act } from 'react-dom/test-utils';
import Counter from './Counter';

let container;

beforeEach(() => {
  container = document.createElement('div');
  document.body.appendChild(container);
});

afterEach(() => {
  document.body.removeChild(container);
  container = null;
});

it('can render and update a counter', () => {
  // Test first render and componentDidMount
  act(() => {
    ReactDOM.render(<Counter />, container);
  });
  const button = container.querySelector('button');
  const label = container.querySelector('p');
  expect(label.textContent).toBe('You clicked 0 times');
  expect(document.title).toBe('You clicked 0 times');

  // Test second render and componentDidUpdate
  act(() => {
    button.dispatchEvent(new MouseEvent('click', {bubbles: true}));
  });
  expect(label.textContent).toBe('You clicked 1 times');
  expect(document.title).toBe('You clicked 1 times');
});
```

DOM イベントのディスパッチは、DOM コンテナが `document` に追加されたときだけ動作することを忘れないでください。`react-testing-library` のようなヘルパーを使えばボイラープレートのコードを減らせます。

`mockComponent()`

```
mockComponent(
  componentClass,
  [mockTagName]
)
```

モック化されたコンポーネントモジュールをこのメソッドに渡すことで、ダミーの React コンポーネントとして使用できるようになる便利なメソッドを追加することができます。通常のレンダーの代わりに、コンポーネントは、与えられた子要素を含んだシンプルな `<div>`（もしくは `mockTagName` が与えられていれば他のタグ）になります。

> **補足:**
> `mockComponent()` はレガシーな API です。その代わりとして shallow rendering や `jest.mock()` の使用をおすすめします。

`isElement()`

```
isElement(element)
```

`element` が任意の React 要素である場合 `true` を返します。

`isElementOfType()`

```
isElementOfType(
  element,
  componentClass
)
```

`element` が `componentClass` 型の React 要素である場合 `true` を返します。

`isDOMComponent()`

```
isDOMComponent(instance)
```

`instance` が DOM コンポーネント（`<div>` や `<span>` など）である場合 `true` を返します。

`isCompositeComponent()`

```
isCompositeComponent(instance)
```

`instance` がクラスや関数のようなユーザ定義のコンポーネントである場合 `true` を返します。

`isCompositeComponentWithType()`

```
isCompositeComponentWithType(
  instance,
  componentClass
)
```

`instance` が React の `componentClass` 型のコンポーネントである場合 `true` を返します。

`findAllInRenderedTree()`

```
findAllInRenderedTree(
  tree,
  test
)
```

`tree` 中のすべてのコンポーネントを横断して `test(component)` が `true` である全てのコンポーネントを集め、その結果を返します。このメソッド自身はそれほど有用ではありませんが、他のテストユーティリティのための基本メソッドとして使用されます。

`scryRenderedDOMComponentsWithClass()`

```
scryRenderedDOMComponentsWithClass(
  tree,
  className
)
```

レンダーされたツリー内に存在する、クラス名が `className` に一致する DOM コンポーネントが持つ全ての DOM 要素を探し、その結果を返します。

`findRenderedDOMComponentWithClass()`

```
findRenderedDOMComponentWithClass(
  tree,
  className
)
```

`scryRenderedDOMComponentsWithClass()` と同様のメソッドですが、このメソッドは結果が 1 つだけであることを期待しており、その 1 つの結果を返すか、一致するものが 1 つでなかった場合には例外をスローします。

`scryRenderedDOMComponentsWithTag()`

```
scryRenderedDOMComponentsWithTag(
  tree,
  tagName
)
```

レンダリングされたツリー内に存在する、タグ名が `tagName` に一致する DOM コンポーネントが持つ全ての DOM 要素を探し、その結果を返します。

`findRenderedDOMComponentWithTag()`

```
findRenderedDOMComponentWithTag(
  tree,
  tagName
)
```

`scryRenderedDOMComponentsWithTag()` と同様のメソッドですが、このメソッドは結果が 1 つだけであることを期待しており、その 1 つの結果を返すか、一致するものが 1 つでなかった場合には例外を返します。

`scryRenderedComponentsWithType()`

```
scryRenderedComponentsWithType(
  tree,
  componentClass
)
```

型が `componentClass` と同じコンポーネントのインスタンスを全て探し、その結果を返します。

`findRenderedComponentWithType()`

```
findRenderedComponentWithType(
  tree,
  componentClass
)
```

`scryRenderedComponentsWithType()` と同様のメソッドですが、このメソッドは結果が 1 つだけであることを期待しており、その 1 つの結果を返すか、一致するものが 1 つでなかった場合には例外を返します。

`renderIntoDocument()`

```
renderIntoDocument(element)
```

React 要素をドキュメントから切り離された DOM ノードにレンダーします。**この関数を実行するには DOM が必要です。**これは以下のコードと実質的に等価です：

```
const domContainer = document.createElement('div');
ReactDOM.render(element, domContainer);
```

> **補足:**

React をインポートする**前**に `window`, `window.document` および `window.document.createElement` をグローバルスコープに持っている必要があります。そうでなければ React は DOM にアクセスできないものと判断し `setState` のようなメソッドが動作しなくなります。

その他のユーティリティ

`Simulate`

```
Simulate.{eventName}(
  element,
  [eventData]
)
```

省略可能な `eventData` イベントデータを使って DOM ノード上のイベントディスパッチをシミュレートします。

`Simulate` は React が理解している全てのイベントそれぞれに対応するメソッドを持っています。

要素をクリックする

```
// <button ref={(node) => this.button = node}>...</button>
const node = this.button;
ReactTestUtils.Simulate.click(node);
```

入力フィールドの値を変更して ENTER キーを押す

```
// <input ref={(node) => this.textInput = node} />
const node = this.textInput;
node.value = 'giraffe';
ReactTestUtils.Simulate.change(node);
ReactTestUtils.Simulate.keyDown(node, {key: "Enter", keyCode: 13, which: 13});
```

補足

React はコンポーネントで使用しているイベントプロパティ（例えば keyCode、which など）を何も作成しないため、あなたはそれらを Simulate が持つメソッドに渡す必要があります。

このページを編集する