React の最上位 API

`React` は React ライブラリのエントリポイントです。`<script>` タグから React を読み込む場合、これらの最上位 API をグローバルの `React` から利用できます。npm と ES6 を使う場合、`import React from 'react'` と書けます。npm と ES5 を使う場合、`var React = require('react')` と書けます。

概要

コンポーネント

React コンポーネントを使用すると UI を独立した再利用可能な部分に分割し、各部分を個別に考えることができます。React コンポーネントは `React.Component` または `React.PureComponent` をサブクラス化することで定義できます。

- `React.Component`
- `React.PureComponent`

ES6 クラスを使わない場合は、代わりに `create-react-class` モジュールを使うことができます。詳しくは Using React without ES6 を参照してください。
React コンポーネントは関数で定義でき、その際に以下の関数でラップできます：

- `React.memo`

React 要素を作成する

UI がどのように見えるべきかを記述するために JSX の使用 を推奨します。JSX のそれぞれの要素は `React.createElement()` を呼ぶための単なる糖衣構文です。JSX を使用している場合は、通常、次のメソッドを直接呼び出すことはありません。

- `createElement()`
- `createFactory()`

詳しくは JSX なしで React を使う を参照してください。

要素を変換する

`React` は要素を操作するためのいくつかの API を提供しています。

- `cloneElement()`
- `isValidElement()`
- `React.Children`

フラグメント (Fragment)

`React` はラッパーなしで複数の要素をレンダーするためのコンポーネントを提供しています。

- `React.Fragment`

Refs

- `React.createRef`
- `React.forwardRef`

サスペンス (Suspense)

サスペンスを使用すると、コンポーネントはレンダーの前に何かを「待機」できます。現在、サスペンスは 1 つのユースケースのみをサポートしています：`React.lazy` を使ってコンポーネントを動的に読み込む。将来的にはデータの取得のような他のユースケースもサポートされるでしょう。

- `React.lazy`
- `React.Suspense`

フック (hook)

フック (hook) は React 16.8 で追加された新機能です。state などの React の機能を、クラスを書かずに使えるようになります。フックには専用のセクションと別の API リファレンスがあります。

- 基本的なフック
  - `useState`
  - `useEffect`

  - useContext
- 追加のフック
  - useReducer
  - useCallback
  - useMemo
  - useRef
  - useImperativeHandle
  - useLayoutEffect
  - useDebugValue

リファレンス

React.Component

React コンポーネントが ES6 クラス を用いて定義されている場合、 React.Component はそれらの基底クラスになります。

```
class Greeting extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

基底クラス React.Component に関するメソッドとプロパティの一覧については、React.Component API Reference を参照してください。

React.PureComponent

React.PureComponent は React.Component と似ています。両者の違いは React.Component が shouldComponentUpdate() を実装していないことに対し、 React.PureComponent は props と state を浅く (shallow) 比較することでそれを実装していることです。
React コンポーネントの render() 関数が同じ props と state を与えられたときに同じ結果をレンダーするときは、パフォーマンスを向上させるために React.PureComponent を使用できます。

> **補足**
> React.PureComponent の shouldComponentUpdate() は オブジェクトの浅い比較のみを行います。これらに複雑なデータ構造が含まれていると、深い部分のみに差分があるために、本当は差分があるにも関わらず差分がないと見なされる場合があります。単純な props と state を持つ場合にのみ PureComponent を継承するか、深いデータ構造が変更されたとわかっているときに forceUpdate() を使用してください。あるいは、ネストされたデータ構造の高速な比較を容易にするために イミュータブルなオブジェクト の使用を検討してください。
> さらに、React.PureComponent の shouldComponentUpdate() はサブツリー全体のコンポーネントの props の更新をスキップします。子コンポーネントの全てが「純粋な」コンポーネントであることを確認してください。

React.memo

```
const MyComponent = React.memo(function MyComponent(props) {
  /* render using props */
});
```

React.memo は 高階コンポーネント です。これは React.PureComponent に似ていますが、クラスではなく関数コンポーネント用です。
もしある関数コンポーネントが同じ props を与えられたときに同じ結果をレンダーするなら、結果を記憶してパフォーマンスを向上させるためにそれを React.memo でラップすることができます。つまり、React はコンポーネントのレンダーをスキップし、最後のレンダー結果を再利用します。
デフォルトでは props オブジェクト内の複雑なオブジェクトは浅い比較のみが行われます。比較を制御したい場合は 2 番目の引数でカスタム比較関数を指定できます。

```
function MyComponent(props) {
  /* render using props */
}
function areEqual(prevProps, nextProps) {
  /*
  nextProps を render に渡した結果が
  prevProps を render に渡した結果となるときに true を返し
  それ以外のときに false を返す
  */
}
export default React.memo(MyComponent, areEqual);
```

これは**パフォーマンス最適化**のためだけの方法です。バグを引き起こす可能性があるため、レンダーを「抑止する」ために使用しないでください。

> **注意**

> クラスコンポーネントの `shouldComponentUpdate()` とは異なり、この `areEqual` 関数は props が等しいときに `true` を返し、props が等しくないときに `false` を返します。これは `shouldComponentUpdate` とは逆です。

`createElement()`

```
React.createElement(
  type,
  [props],
  [...children]
)
```

与えられた型の新しい React 要素を作成して返します。`type` 引数はタグ名の文字列（`'div'` や `'span'` など）、React component 型（クラスもしくは関数）、React fragment 型のいずれかです。
JSX で書かれたコードは `React.createElement()` を用いるコードに変換されます。JSX を使っていれば通常 `React.createElement()` を直接呼び出すことはありません。詳しくは JSX なしで React を使うを参照してください。

`cloneElement()`

```
React.cloneElement(
  element,
  [props],
  [...children]
)
```

`element` から新しい React 要素を複製して返します。結果の要素は元の要素の props と新しい props が浅くマージされたものを持ちます。新しい子要素は既存の子要素を置き換えます。`key` と `ref` は元の要素から保持されます。
`React.cloneElement()` は以下のコードとほぼ同等です：

```
<element.type {...element.props} {...props}>{children}</element.type>
```

ただし、`ref` も保持されます。つまり `ref` のある子要素を受け取っても、間違って元の React 要素から `ref` を盗むことはありません。新しい要素にも同じ `ref` が追加されます。
この API は非推奨の `React.addons.cloneWithProps()` の代替として導入されました。

`createFactory()`

```
React.createFactory(type)
```

与えられた型の React 要素を生成する関数を返します。`React.createElement()` と同様に、`type` 引数はタグ名の文字列（`'div'` や `'span'` など）、React コンポーネント型（クラスもしくは関数）、React フラグメント型のいずれかです。
このヘルパーはレガシーだと考えられているため、代わりに JSX か `React.createElement()` を直接使用することをおすすめします。
JSX を使っていれば通常 `React.createFactory()` を直接呼び出すことはありません。詳しくは JSX なしで React を使うを参照してください。

`isValidElement()`

```
React.isValidElement(object)
```

オブジェクトが React 要素であることを確認します。`true` または `false` を返します。

`React.Children`

`React.Children` はデータ構造が非公開の `this.props.children` を扱うためのユーティリティを提供します。

`React.Children.map`

```
React.Children.map(children, function[(thisArg)])
```

this を thisArg に設定して、children 内に含まれるすべての直下の子要素に対して関数を呼び出します。children が配列の場合は走査され、配列の各要素に対して関数が呼び出されます。children が null または undefined の場合はこのメソッドは配列ではなく null または undefined を返します。

補足
children が Fragment の場合、それは 1 つの子要素として扱われ、走査されません。

React.Children.forEach

```
React.Children.forEach(children, function[(thisArg)])
```

React.Children.map() と似ていますが、配列を返しません。

React.Children.count

```
React.Children.count(children)
```

children に含まれるコンポーネントの総数を返します。これは map または forEach に渡したコールバックが呼ばれる回数と同じです。

React.Children.only

```
React.Children.only(children)
```

children が 1 つの子要素しか持たないことを確認し、結果を返します。そうでない場合、このメソッドはエラーを投げます。

補足:
React.Children.map() の返り値は React 要素ではなく配列なため、React.Children.only() はそれを受け付けません。

React.Children.toArray

```
React.Children.toArray(children)
```

データ構造が非公開の children を平坦な配列として返し、それぞれの要素に key を割り当てます。レンダーメソッド内で子の集合を操作したい場合、特に this.props.children を渡す前に並べ替えたりスライスしたい場合に便利です。

補足:
React.Children.toArray() は子のリストを平坦にするときにネストされた配列の意味を保つために key を変更します。つまり、toArray は配列のそれぞれの要素の key に接頭辞を付けて返します。

React.Fragment

React.Fragment コンポーネントを使用すると追加の DOM 要素を作成することなく render() メソッドで複数の要素を返すことができます。

```
render() {
  return (
    <React.Fragment>
      Some text.
      <h2>A heading</h2>
    </React.Fragment>
  );
}
```

また、フラグメントを <></> という短縮構文で使用できます。詳しくは React v16.2.0: Improved Support for Fragments を参照してください。

React.createRef

React.createRef は ref を作成します。ref は ref 属性を介して React 要素に取り付けることができます。

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
```

```
    this.inputRef = React.createRef();
  }

  render() {
    return <input type="text" ref={this.inputRef} />;
  }

  componentDidMount() {
    this.inputRef.current.focus();
  }
}
```

`React.forwardRef`

`React.forwardRef` は ref を配下のツリーの別のコンポーネントに受け渡す React コンポーネントを作成します。この手法はあまり一般的ではありませんが、2 つのシナリオで特に役立ちます：

- Forwarding refs to DOM components
- Forwarding refs in higher-order-components

`React.forwardRef` はレンダー関数を引数として受け入れます。React は props と ref を 2 つの引数として呼び出します。この関数は React ノードを返す必要があります。

```
const FancyButton = React.forwardRef((props, ref) => (
  <button ref={ref} className="FancyButton">
    {props.children}
  </button>
));

// You can now get a ref directly to the DOM button:
const ref = React.createRef();
<FancyButton ref={ref}>Click me!</FancyButton>;
```

上の例では、React は `<FancyButton ref={ref}>` 要素に与えた `ref` を `React.forwardRef` の呼び出し内のレンダー関数の 2 番目の引数として渡します。このレンダー関数は `ref` を `<button ref={ref}>` 要素に渡します。
結果として、React が `ref` を取り付けた後、`ref.current` は `<button>` の DOM 要素のインスタンスを直接指すようになります。
詳しくは forwarding refs を参照してください。

`React.lazy`

`React.lazy()` を使用すると、動的に読み込まれるコンポーネントを定義できます。これにより、バンドルサイズを削減して、最初のレンダー時に使用されないコンポーネントの読み込みを遅らせることができます。
code splitting のドキュメントから使用方法を学ぶことができます。また、使い方をより詳しく説明したこちらの記事もチェックしてみてください。

```
// This component is loaded dynamically
const SomeComponent = React.lazy(() => import('./SomeComponent'));
```

`lazy` コンポーネントをレンダーするには `<React.Suspense>` がレンダリングツリーの上位に必要です。これはローディングインジケータを指定する方法です。

> **補足**
> `React.lazy` を使って動的にインポートするには JS 環境で Promise が使用できる必要があります。これは IE11 以前の環境ではポリフィルが必要だということです。

`React.Suspense`

`React.Suspense` を使用することで、その配下にレンダーする準備ができていないコンポーネントがあるときにローディングインジケータを指定できます。現在、遅延読み込みコンポーネントは `<React.Suspense>` のみによってサポートされています。

```
// This component is loaded dynamically
const OtherComponent = React.lazy(() => import('./OtherComponent'));

function MyComponent() {
  return (
    // Displays <Spinner> until OtherComponent loads
    <React.Suspense fallback={<Spinner />}>
      <div>
        <OtherComponent />
      </div>
    </React.Suspense>
  );
}
```

これは code splitting guide で文書化されています。遅延される (lazy) コンポーネントを `Suspense` ツリーの奥深くに置くことができ、それらを 1 つずつラップする必要はありません。ベストプラクティスは `<Suspense>` をローディングインジケータを表示したい場所に配置することですが、コードを分割したい場合は `lazy()` を使用してください。
これらは現在サポートされていませんが、将来的には `Suspense` にデータの取得などのより多くのシナリオを処理させる予定です。これについてはロードマップで読めます。

> **注意:**

> `React.lazy()` と `<React.Suspense>` は `ReactDOMServer` ではまだサポートされていません。これは既知の制限であり、今後解決されます。

このページを編集する