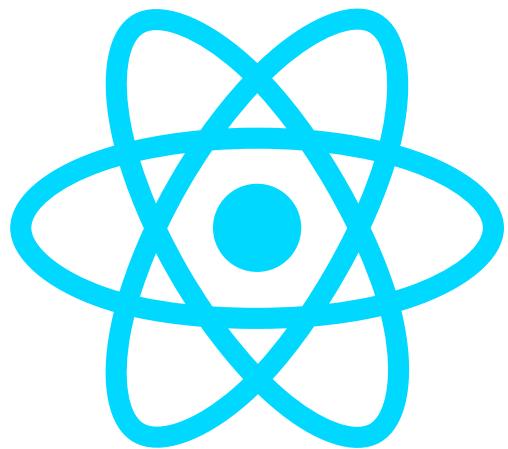


# REACT Docs - 日本語



## Table of contents

• Accessibility	5
• Add react /to /a /website	17
• Addons animation	22
• Addons create /fragment	29
• Addons perf	31
• Addons pure /render /mixin	35
• Addons shallow /compare	36
• Addons shallow /renderer	37
• Addons test /utils	39
• Addons two /way /binding /helpers	46
• Addons update	49
• Addons	52
• Cdn links	54
• Code splitting	55
• Codebase overview	60
• Components and /props	64
• Composition vs /inheritance	69
• Conditional rendering	73
• Context	78
• Create a /new /react /app	84
• Cross origin /errors	87
• Design principles	89
• Error boundaries	95
• Error decoder	100
• Faq ajax	101
• Faq build	104
• Faq functions	105
• Faq internals	112
• Faq state	113
• Faq structure	116
• Faq styling	118
• Faq versioning	119
• Forms	122
• Forwarding refs	128

• Fragments	131
• Getting started	134
• Handling events	138
• Hello world	141
• Higher order /components	143
• Hooks custom	151
• Hooks effect	156
• Hooks faq	167
• Hooks intro	188
• Hooks overview	193
• Hooks reference	199
• Hooks rules	215
• Hooks state	218
• How to /contribute	224
• Implementation notes	229
• Integrating with /other /libraries	247
• Introducing jsx	256
• Jsx in /depth	260
• Legacy context	269
• Legacy event /pooling	274
• Lifting state /up	275
• Lists and /keys	283
• Optimizing performance	288
• Portals	297
• React without /es6	300
• React without /jsx	304
• Reconciliation	305
• Reference dom /elements	309
• Reference events	314
• Reference glossary	322
• Reference javascript /environment /requirements	327
• Reference profiler	328
• Reference pure /render /mixin	331
• Reference react /component	333
• Reference react /dom /client	348
• Reference react /dom /server	350

• Reference react /dom	355
• Reference react	360
• Reference test /renderer	369
• Refs and /the /dom	375
• Release channels	381
• Render props	384
• Rendering elements	390
• State and /lifecycle	392
• Static type /checking	400
• Strict mode	407
• Testing environments	413
• Testing recipes	415
• Testing	426
• Thinking in /react	427
• Typechecking with /proptypes	432
• Uncontrolled components	437
• Web components	439

# アクセシビリティ

## なぜアクセシビリティが必要なのか？

Web アクセシビリティ（[a11y](#) とも呼ばれます）とは、誰にでも使えるようウェブサイトを設計・構築することです。ユーザ補助技術がウェブページを解釈できるようにするために、サイトでアクセシビリティをサポートする必要があります。

React はアクセシビリティを備えたウェブサイトの構築を全面的にサポートしており、大抵は標準の HTML の技術が用いられます。

## 標準およびガイドライン

### WCAG

[Web Content Accessibility Guidelines](#) はアクセシビリティを備えたウェブサイトを構築するためのガイドラインを提供しています。

以下の WCAG のチェックリストはその概要を示します。

- [WCAG checklist from Wuhcag](#)
- [WCAG checklist from WebAIM](#)
- [Checklist from The A11Y Project](#)

### WAI-ARIA

[Web Accessibility Initiative - Accessible Rich Internet Applications](#) には十分なアクセシビリティを持つ JavaScript ウィジェットの構築テクニックが含まれています。

補足として、JSX ではすべての `aria-*` で始まる HTML 属性がサポートされています。Reactにおいてほとんどの DOM プロパティと属性がキャメルケースである一方で、これらの属性は純粋な HTML と同じようにハイフンケース（ケバブケースやリスプケースなどとも言われる）である必要があります。

```
<input
  type="text"
  aria-label={labelText}
  aria-required="true"
  onChange={onchangeHandler}
  value={inputValue}
  name="name"
/>
```

## セマンティックな HTML

セマンティック（意味論的）な HTML はウェブアプリケーションにおけるアクセシビリティの基礎となります。ウェブサイト内の情報の意味を明確にするための多様な HTML 要素を使うことにより、大抵の場合は少ない手間でアクセシビリティを手に入れられます。

- [MDN HTML 要素リファレンス](#)

ときおり、React コードを動くようにするために JSX に `<div>` を追加すると、HTML のセマンティックが崩れることができます。とりわけ、リスト (`<ol>`, `<ul>`, `<dl>`) や `<table>` タグと組み合わせるときに問題になります。そんなときは複数の要素をグループ化するために [React フラグメント](#)を使う方がよいでしょう。

具体例です。

```
import React, { Fragment } from 'react';

function ListItem({ item }) {
  return (
    <Fragment>
      <dt>{item.term}</dt>
      <dd>{item.description}</dd>
    </Fragment>
  );
}

function Glossary(props) {
  return (
    <dl>
      {props.items.map(item => (
        <ListItem item={item} key={item.id} />
      ))}
    </dl>
  );
}
```

項目の集合をフラグメントの配列に変換することができますし、他の任意の要素でも同様です。

```
function Glossary(props) {
  return (
    <dl>
      {props.items.map(item => (
        // Fragments should also have a `key` prop when mapping collections
        <Fragment key={item.id}>
          <dt>{item.term}</dt>
          <dd>{item.description}</dd>
        </Fragment>
      ))}
    </dl>
  );
}
```

もし、フラグメントタグに `props` を渡す必要がなく、かつ使用しているツールがサポートしているのであれば、[省略記法](#)が使えます。

```
function ListItem({ item }) {
  return (
    <>
      <dt>{item.term}</dt>
      <dd>{item.description}</dd>
    </>
  );
}
```

より詳細な情報は[フラグメントドキュメント](#)にあります。

## アクセシブルなフォーム

### ラベル付け

`<input>` や `<textarea>` のような各 HTML フォームコントロールには、アクセシブルな形でのラベル付けが必要です。スクリーンリーダに公開される、説明的なラベルを提供する必要があります。

以下の資料にはその方法が示されています：

- [W3C による要素にラベルを付ける方法の解説](#)
- [WebAIM による要素にラベルを付ける方法の解説](#)
- [The Paciello Group によるアクセシブルな名前についての解説](#)

React でこれらの標準的な HTML の実践知識を直接使用できますが、JSX では `for` 属性は `htmlFor` として記述されることに注意してください。

```
<label htmlFor="namedInput">Name:</label>
<input id="namedInput" type="text" name="name"/>
```

### ユーザへのエラー通知

すべてのユーザがエラーの起きた状況を理解できる必要があります。以下のリンクはどのようにエラーテキストをユーザと同じくスクリーンリーダにも公開するかを解説しています。

- [W3C によるユーザへの通知方法の例示](#)
- [WebAIM によるフォームバリデーションの解説](#)

### フォーカス制御

あなたのウェブアプリケーションが完全にキーボードだけで操作できることを確かめてください：

- [WebAIM によるキーボードアクセシビリティの解説](#)

### キーボードフォーカスとフォーカス時のアウトライン(輪郭線)

キーボードフォーカスは DOM の中でキーボードからの入力を受け付けるために選択されている要素を示します。フォーカスを輪郭線で示した以下の画像のような例を、様々な場所で見かけることができます：



例えば `outline: 0` のようにしてこのアウトラインを CSS で削除できますが、これは他の実装でフォーカス線を置き換える場合にのみ行うようにしてください。

## 目的のコンテンツまで飛べる仕組み

キーボードによる操作を補助して高速化するために、あなたのアプリケーションのナビゲーション（メニューや目次）部分をユーザが読み飛ばせるような仕組みを提供しましょう。

スキップリンク ("skiplink") やスキップナビゲーションリンク ("skip navigation link") とは、ユーザがキーボードでページを操作する場合にのみ出現する、隠れたナビゲーションリンクです。これらのリンクはページ内アンカーといくらかのスタイルを用いて、とても簡単に実装できます：

- [WebAIM - Skip Navigation Links](#)

`<main>` や `<aside>` のようなランドマーク要素とロール属性も活用してページの領域を区切り、補助技術を使うユーザが素早くこれらのセクションに移動できるようにしてください。

アクセシビリティを強化する、これらの要素の使い方についての詳細は以下を読んでください：

- [Accessible Landmarks](#)

## プログラムによりフォーカスを管理する

React アプリケーションは実行されている間、継続的に HTML の DOM を変更するため、時にキーボードフォーカスが失われたり、予期しない要素にセットされたりすることがあります。これを修正するためには、プログラムによってキーボードフォーカスを正しい位置に移動させる必要があります。例えばモーダルウィンドウを閉じた後には、モーダルを開いたボタンにキーボードフォーカスを戻すことなどです。

MDN のウェブドキュメントには、[キーボードで移動可能な JavaScript ウィジェット](#) の作り方が解説されています。

React でフォーカスをセットするには、[DOM 要素への Ref](#) が使えます。

これを使って、まずコンポーネントクラスの JSX 要素に `ref` を作成します：

```
class CustomTextInput extends React.Component {
  constructor(props) {
    super(props);
    // Create a ref to store the TextInput DOM element
    this.textInput = React.createRef();
  }
  render() {
    // Use the `ref` callback to store a reference to the text input DOM
    // element in an instance field (for example, this.textInput).
    return (
      <input
        type="text"
        ref={this.textInput}
      />
    );
  }
}
```

これで必要な場合にはコンポーネントのほかの場所からその要素にフォーカスすることができます。

```
focus() {
  // Explicitly focus the text input using the raw DOM API
  // Note: we're accessing "current" to get the DOM node
  thistextInput.current.focus();
}
```

ときおり、親コンポーネントは子コンポーネント内の要素にフォーカスをセットする必要があります。これは、親の ref を子の DOM ノードに転送する特別なプロパティを通して親コンポーネントに DOM の ref を公開することで可能になります。

```
function CustomTextInput(props) {
  return (
    <div>
      <input ref={props.inputRef} />
    </div>
  );
}

class Parent extends React.Component {
  constructor(props) {
    super(props);
    this.inputElement = React.createRef();
  }
  render() {
    return (
      <CustomTextInput inputRef={this.inputElement} />
    );
  }
}

// Now you can set focus when required.
this.inputElement.current.focus();
```

コンポーネントを拡張するのに高階コンポーネント (HOC) を使う場合は、React の `forwardRef` 関数を用いて、関数に囲われたコンポーネントに `ref` をフォワーディング (forwarding) することをおすすめします。もし、サードパーティの高階コンポーネントが `ref` フォワーディングを実装していないときでも、上記のパターンはフォールバックとして使えます。

良いフォーカス管理の例は `react-aria-modal` です。これは完全にアクセシブルなモーダルウィンドウの比較的珍しい例です。このライブラリは、最初のフォーカスをキャンセルボタンに設定し（これは、キーボードを使っているユーザがうっかり次のアクションに移ってしまうのを防ぎます）、モーダルの中でキーボードフォーカスが閉じているだけでなく、最初にモーダルを開いた要素にフォーカスを戻してくれます。

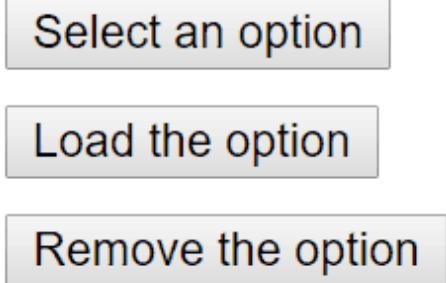
補足：

これはとても重要なアクセシビリティ機能ですが、慎重に使用されるべきテクニックでもあります。このテクニックはキーボードフォーカスの流れが妨げられた場合の修正に使用し、ユーザがアプリケーションをどのように使いたいかを試したり予測するのに使わないでください。

## マウスとポインタのイベント

マウスまたは、ポインタのイベントを通じて使われる機能がキーボード単体でも同じように使用できるようにしてください。ポインタデバイスだけに依存した実装は、多くの場合にキーボードユーザがアプリケーションを使えない原因になります。

これを説明するために、クリックイベントによってアクセシビリティが損なわれるよくある例を見てみましょう。以下の画像はアウトサイドクリックパターンというユーザが要素の外側をクリックして開いている要素を閉じられるパターンです。



これは通常、ポップアップを閉じる役割をもつ `click` イベントを `window` オブジェクトに付与することで実装します。

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

    this.state = { isOpen: false };
    this.toggleContainer = React.createRef();

    this.onClickHandler = this.onClickHandler.bind(this);
    this.onClickOutsideHandler = this.onClickOutsideHandler.bind(this);
  }

  componentDidMount() {
    window.addEventListener('click', this.onClickOutsideHandler);
  }

  componentWillUnmount() {
    window.removeEventListener('click', this.onClickOutsideHandler);
  }

  onClickHandler() {
    this.setState(currentState => ({
      isOpen: !currentState.isOpen
    }));
  }

  onClickOutsideHandler(event) {
    if (this.state.isOpen && !this.toggleContainer.current.contains(event.target))
    {
      this.setState({ isOpen: false });
    }
  }

  render() {
```

```

return (
  <div ref={this.toggleContainer}>
    <button onClick={this.onClickHandler}>Select an option</button>
    {this.state.isOpen && (
      <ul>
        <li>Option 1</li>
        <li>Option 2</li>
        <li>Option 3</li>
      </ul>
    )}
  </div>
);
}
}

```

これはマウスのようなポインタデバイスでは問題なく機能しますが、キーボード単体で操作しようとした場合、タブキーによって次の要素に移動しても `window` オブジェクトは `click` イベントを受け取らないため、うまく機能しません。一部のユーザはあなたのアプリを利用できなくなってしまうでしょう。

Select an option

Load the option

Remove the option

これと同じ機能は `onBlur` と `onFocus` のような適切なイベントハンドラを代わりに用いることで実現できます。

```

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

    this.state = { isOpen: false };
    this.timeOutId = null;

    this.onClickHandler = this.onClickHandler.bind(this);
    this.onBlurHandler = this.onBlurHandler.bind(this);
    this.onFocusHandler = this.onFocusHandler.bind(this);
  }

  onClickHandler() {
    this.setState(currentState => ({
      isOpen: !currentState.isOpen
    }));
  }

  // We close the popover on the next tick by using setTimeout.
  // This is necessary because we need to first check if
  // another child of the element has received focus as
  // the blur event fires prior to the new focus event.
  onBlurHandler() {
    this.timeOutId = setTimeout(() => {

```

```

        this.setState({
          isOpen: false
        });
      });
    }

    // If a child receives focus, do not close the popover.
    onFocusHandler() {
      clearTimeout(this.timeOutId);
    }

    render() {
      // React assists us by bubbling the blur and
      // focus events to the parent.
      return (
        <div onBlur={this.onBlurHandler}
              onFocus={this.onFocusHandler}>
          <button onClick={this.onClickHandler}
                  aria-haspopup="true"
                  aria-expanded={this.state.isOpen}>
            Select an option
          </button>
          {this.state.isOpen && (
            <ul>
              <li>Option 1</li>
              <li>Option 2</li>
              <li>Option 3</li>
            </ul>
          )}
        </div>
      );
    }
  }
}

```

上記のコードは、ポインタデバイスを使うユーザとキーボードを使うユーザの双方にこの機能を公開します。さらに、`aria-* props` を加えるとスクリーンリーダを使うユーザもサポートできます。話を簡単にするため、ポップアップの選択肢を `矢印キー` で操作できるようにするキーボードイベントは実装していません。

Select an option

Load the option

Remove the option

これはポインタデバイスとマウスイベントだけに依存するとキーボードを使うユーザにとって機能が損なわれてしまう数多くの具体例のうちのひとつです。つねにキーボードによるテストをすれば、キーボードに対応するイベントを使うことで解決できる問題領域をすばやく発見できるでしょう。

## より複雑なウィジェット

ユーザ体験がより複雑であるほど、よりアクセシビリティが損なわれるということがあってはいけません。できるだけ HTML に近くなるようコーディングすればアクセシビリティを最も簡単に達成できますが、一方でかなり複雑なウィジェットでもアクセシビリティを保ってコーディングすることができます。

ここでは [ARIA のロール](#) や [ARIA のステートとプロパティ](#)についての知識も必要となります。これらは JSX で完全にサポートされている HTML 属性が詰まったツールボックスであり、十分にアクセシブルで高機能な React コンポーネントの構築を可能してくれます。

それぞれの種類のウィジェットはそれぞれ特定のデザインパターンを持っており、ユーザやユーザエージェントはそれらが特定の方法で機能することを期待します：

- [WAI-ARIA Authoring Practices - Design Patterns and Widgets](#)
- [Heydon Pickering - ARIA Examples](#)
- [Inclusive Components](#)

## その他に考慮すべきポイント

### 言語設定

ページテキストで使用する自然言語を明示して、読み上げソフトが適切な音声設定を選ぶために利用できるようにしてください：

- [WebAIM - Document Language](#)

### ドキュメントの title の設定

ドキュメントの `<title>` は、ユーザが現在いるページのコンテキストを認識していられるように、そのページのコンテンツを正しく説明するものにしてください：

- [WCAG - Understanding the Document Title Requirement](#)

React では [React Document Title Component](#) を使用することで title を設定できます。

### 色のコントラスト

あなたのウェブサイトにある全ての読めるテキストが、色弱のユーザにも最大限読めるように配慮した色のコントラストがあることを確認してください：

- [WCAG - Understanding the Color Contrast Requirement](#)
- [Everything About Color Contrast And Why You Should Rethink It](#)
- [A11yProject - What is Color Contrast](#)

適切な色の組み合わせをウェブサイト内の全てのケースについて手作業で行うのは面倒になりがちなので、代わりに [アクセシブルなカラーパレット全体を Colorable で計算することができます。](#)

以下に述べる aXe および WAVE ツールのどちらも同じように色のコントラストのテストを備えておりコントラストの違反を報告してくれます。

コントラストをチェックする能力を拡張したい場合は、以下のツールが利用できます：

- [WebAIM - Color Contrast Checker](#)
- [The Paciello Group - Color Contrast Analyzer](#)

## 開発とテストのツール

アクセシブルなウェブアプリケーションの作成を支援するために利用できる様々なツールがあります。

### キーボード

最も簡単で最も重要なチェックのうちのひとつは、ウェブサイト全体がキーボード単体でうまく探索でき、使えるかどうかのテストです。これは以下の手順でチェックできます。

1. マウスを外します。
2. `Tab` と `Shift+Tab` を使ってブラウズします。
3. 要素を起動するのに `Enter` を使用します。
4. 必要に応じて、キーボードの矢印キーを使ってメニュー や ドロップダウンリストなどの要素を操作します。

### 開発支援

アクセシビリティ機能には JSX のコード内で直接チェックできるものもあります。JSX に対応した IDE では、ARIA ロールやステートやプロパティに対する intellisense によるチェックが既に提供されていることが多いでしょう。他にも以下のツールを使うこともできます：

#### **eslint-plugin-jsx-a11y**

ESLint の [eslint-plugin-jsx-a11y](#) プラグインはあなたの JSX コードのアクセシビリティに対して、AST による lint のフィードバックを提供します。多くの IDE はコード解析とソースコードのウィンドウに直接そのフィードバックを統合できるようになっています。

[Create React App](#) はこのプラグインを備えており、一部のルールを有効化しています。もし、より多くのアクセシビリティルールを有効化したいときは、プロジェクトルートに `.eslintrc` ファイルを作成し、以下の内容を書き込んでください：

```
{
  "extends": ["react-app", "plugin:jsx-a11y/recommended"],
  "plugins": ["jsx-a11y"]
}
```

### ブラウザでアクセシビリティをテストする

ブラウザからウェブページのアクセシビリティを検査できるツールは沢山あります。それらのツールはあなたが作成した HTML の技術的なアクセシビリティしかチェックできないため、ここで言及した他のアクセシビリティ確認法と組み合わせて使用してください。

#### **aXe, aXe-core and react-axe**

Deque System はアプリケーションの自動化された E2E アクセシビリティテストを行う [aXe-core](#) を提供しています。このモジュールは Selenium に統合できます。

The Accessibility Engine もしくは aXe は、`aXe-core` により構築されたアクセシビリティを検査するブラウザ拡張機能です。

`@axe-core/react` モジュールを使用して、開発時やデバッグ時にこれらによるアクセシビリティの検査結果を直接コンソールへ出力させることもできます。

## WebAIM WAVE

[Web Accessibility Evaluation Tool](#) はアクセシビリティに関する別のブラウザ拡張機能です。

### アクセシビリティ検査ツールとアクセシビリティツリー

アクセシビリティツリー (The Accessibility Tree) は、スクリーンリーダのような補助技術に公開されるべきすべての要素についてアクセス可能なオブジェクトを含んだ DOM ツリーのサブセットです。

一部のブラウザではアクセシビリティツリー内の各要素のアクセシビリティに関する情報を簡単に見ることができます：

- [Using the Accessibility Inspector in Firefox](#)
- [Using the Accessibility Inspector in Chrome](#)
- [Using the Accessibility Inspector in OS X Safari](#)

## スクリーンリーダ

アクセシビリティのテストの一環として、スクリーンリーダによるテストを行うべきです。

ブラウザとスクリーンリーダの相性に注意してください。選択したスクリーンリーダに最適なブラウザでアプリケーションのテストをすることをおすすめします。

### よく使われるスクリーンリーダ

#### NVDA と FireFox

[NonVisual Desktop Access](#) または NVDA は広く利用されているオープンソースの Windows 向けスクリーンリーダーです。

NVDA を最大限に活用する方法は以下のガイドを参照してください：

- [WebAIM - Using NVDA to Evaluate Web Accessibility](#)
- [Deque - NVDA Keyboard Shortcuts](#)

#### VoiceOver と Safari

VoiceOver は Apple 社製品に統合されたスクリーンリーダです。

VoiceOver を有効化して使用する方法は以下のガイドを参照してください：

- [WebAIM - Using VoiceOver to Evaluate Web Accessibility](#)
- [Deque - VoiceOver for OS X Keyboard Shortcuts](#)
- [Deque - VoiceOver for iOS Shortcuts](#)

## JAWS と Internet Explorer

[Job Access With Speech](#) もしくは JAWS は、Windows 上での使用例が豊富なスクリーンリーダーです。

JAWS を最大限に活用する方法は以下のガイドを参照してください：

- [WebAIM - Using JAWS to Evaluate Web Accessibility](#)
- [Deque - JAWS Keyboard Shortcuts](#)

## 他のスクリーンリーダー

### ChromeVox と Google Chrome

ChromeVox は Chromebook に統合されたスクリーンリーダーで、Google Chrome では拡張機能として利用可能です。

ChromeVox を最大限に活用する方法は以下のガイドを参照してください：

- [Google Chromebook Help - Use the Built-in Screen Reader](#)
- [ChromeVox Classic Keyboard Shortcuts Reference](#)

---

[Go to TOC](#)

# 既存のウェブサイトに React を追加する

まずは必要なぶんだけ使ってみましょう。

React は当初から、段階的に導入することができるようデザインされています。つまり **最小限の部分で React を利用することも、あるいは大規模に React を利用することも可能です**。既存のページにちょっとしたインターラクティブ性をもたらしたいだけでも構いません。React コンポーネントを使えばお手の物です。

多くのウェブサイトはシングルページアプリケーションではありませんし、そうする必要もありません。まずは **たった数行のコード** から、あなたのウェブサイトに React を取り入れてみましょう。**ビルドツールは必要ありません**。そこから徐々に React の使用範囲を広げていくのもいいですし、あるいは少しの動的なウィジェットだけにとどめておくのもいいでしょう。

- [1 分で React を追加する](#)
- [オプション：React で JSX を使う（バンドルツールは不要です！）](#)

## 1分で React を導入する

このセクションでは、既存の HTML ページに React コンポーネントを導入する方法を説明します。以下の部分では自分のウェブサイトを利用して進めてもいいですし、練習用に空の HTML ファイルを用意するのもいいでしょう。

複雑なツール類や事前にインストールしておかなければいけないものはありません。**インターネットへの接続さえあれば、1 分間でこのセクションを終わらせることができます**。

オプション：[サンプルをダウンロードする \(2KB ZIP 圧縮\)](#)

### ステップ 1:HTML に DOM コンテナを追加する

まずは編集したい HTML ファイルを開きましょう。React で描画したい箇所を決めて、空の `<div>` 要素を追加しましょう。例えばこんな感じです。

```
<!-- ... 既存の HTML ... -->
<div id="like_button_container"></div>
<!-- ... 既存の HTML ... -->
```

ここでは `<div>` 要素にユニークな `id` 属性を指定しています。こうしておけば、後から JavaScript のコードでこの `<div>` 要素を探し出し、この中に React コンポーネントを表示できます。

#### ヒント

「コンテナ」としての `<div>` 要素は `<body>` タグの中であれば **どこにでも置くことができます**。また空の `<div>` はひとつのページにひとつだけでも、あるいは必要なだけたくさんあっても大丈夫です。`<div>` 要素は空のことが多いですが、それはたとえ `<div>` の中に他の要素があったとしても、React が結局その中身を置き換えてしまうからです。

## ステップ 2:script タグを追加する

次に、同じ HTML ファイルの `</body>` タグの直前に、3 つの `<script>` タグを追加しましょう。

```
<!-- ... other HTML ... -->

<!-- Load React. -->
<!-- Note: when deploying, replace "development.js" with "production.min.js". -->
<script src="https://unpkg.com/react@18/umd/react.development.js" crossorigin>
</script>
<script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js" crossorigin></script>

<!-- Load our React component. -->
<script src="like_button.js"></script>

</body>
```

最初のふたつのタグは React を読み込んでおり、最後のタグはこれから書くコンポーネントのコードを読み込んでいます。

## ステップ 3:React コンポーネントを作成する

`like_button.js` という名前の新しいファイルを作成し、HTML ファイルのすぐ隣に置きましょう。

[サンプルコード](#)を開いて、自分のファイルにコピーアンドペーストしてください。

### ヒント

このコードは `LikeButton` という React コンポーネントを定義しています。まだわからなくとも心配しなくて大丈夫です。こういった React の構成要素については、[チュートリアル](#) と [Hello World](#) のページで後ほど見ていくことにして、まずはサンプルコードを画面に表示させてみましょう！

[サンプルコード](#)の末尾に次の 3 行を追加してみましょう。

```
// ... コピーアンドペーストしたサンプルコード ...

const domContainer = document.querySelector('#like_button_container');
const root = ReactDOM.createRoot(domContainer);
root.render(e(LikeButton));
```

この 3 行のコードは、ステップ 1 で追加した空の `<div>` 要素を見つけてきて、そこに React アプリを作成し、その中に React コンポーネントの「いいね」ボタンを表示します。

## これだけです！

ステップ 4 はありません。これであなたは自分のウェブサイトにはじめての React コンポーネントを導入できました。

React の導入についてもっと知るには、次のセクションも見てみてください。

[完成したソースコードを見る](#)

[完成したソースコードをダウンロードする \(2KB ZIP 圧縮\)](#)

## ヒント:コンポーネントを再利用する

React コンポーネントを HTML ページの一箇所だけではなくいろいろな箇所で使いたくなることがあるかもしれません。そこで「いいね」ボタンを 3 回繰り返し表示し、さらにそこにちょっとしたデータを渡すプログラムを用意しました。

[ソースコードを見る](#)

[ソースコードをダウンロードする \(2KB ZIP 圧縮\)](#)

### 補足

このようなやり方は、主に React を利用する DOM コンテナがページ内でお互いに干渉していない場合において便利な手段です。React 単体のコードとしては、[コンポーネントを組み合わせる](#) やり方のほうが手軽です。

## ヒント:本番環境用に JavaScript を圧縮する

ウェブサイトを本番環境にデプロイするにあたって、圧縮していない JavaScript はページの速度を著しく落としてしまうということに配慮してください。

自分のスクリプトの圧縮が完了していて、デプロイ後の HTML が `production.min.js` で終わる React スクリプトを読み込んでいることが検証できていれば、**あなたのウェブサイトは本番環境にリリースする準備ができています。**

```
<script src="https://unpkg.com/react@18/umd/react.production.min.js" crossorigin>
</script>
<script src="https://unpkg.com/react-dom@18/umd/react-dom.production.min.js"
crossorigin></script>
```

自分のスクリプトを圧縮することがまだできていないのであれば、[例えばこんなやり方があります。](#)

## オプション:React で JSX を使う

今までの例では、ブラウザにもともと備わっている機能のみ使ってきました。React コンポーネントを表示するために次のような JavaScript の関数を呼び出していたのはそのためです。

```
const e = React.createElement;

// Display a "Like" <button>
return e(
  'button',
  { onClick: () => this.setState({ liked: true }) },
  'Like'
);
```

ただし、React においては `JSX` を利用することもできます。

```
// Display a "Like" <button>
return (
  <button onClick={() => this.setState({ liked: true })}>
```

```
Like
  </button>
);
```

これらふたつのスニペットはまったく同じ内容です。JSX の使用は完全にオプションですが、React はもちろん他のライブラリで UI を記述する際にも、JSX は多くの人に支持されています。

[このコンバータ](#)上で JSX を使って遊んでみてください。

## JSX を手軽に試してみる

手っ取り早く JSX を自分のプロジェクトで試してみるには、次の `<script>` タグを追加してみてください。

```
<script src="https://unpkg.com/babel-standalone@6/babel.min.js"></script>
```

この状態で、任意の `<script>` タグに `type="text/babel"` 属性を持たせることで、その `<script>` タグの中では JSX が使えるようになります。[サンプル用 HTML ファイル](#)をダウンロードして遊んでみてください。

この方法は学習やシンプルなデモの作成にはいいですが、これをそのまま使うとウェブサイトは重くなってしまい、**本番環境には向きません**。次のレベルに進む準備ができたら、先ほど追加した `<script>` タグと `type="text/babel"` 属性は削除してしまいましょう。そして次のセクションに進み、JSX プリプロセッサを設定して `<script>` タグを自動変換するようしましょう。

## JSX をプロジェクトに追加する

JSX をプロジェクトに追加するためには、バンドルツールや開発用サーバといった複雑なツールは必要ありません。つまるところ、JSX を追加することは **CSS プリプロセッサを追加することにとてもよく似ています**。唯一必要となるのは、コンピューターに [Node.js](#) がインストールされていることだけです。

ターミナルを開き、プロジェクトのディレクトリに移動した上で、次のふたつのコマンドを実行してください。

1. ステップ 1: `npm init -y` (うまくいかなければ[こうやってみてください](#))
2. ステップ 2: `npm install babel-cli@6 babel-preset-react-app@3`

### ヒント

ここでは **JSX プリプロセッサをインストールするためだけに npm を使っています**。それ以外の用途では必要ありません。React のソースコードもアプリケーションコードも引き続き `<script>` タグの中にそのまま書くことができます。

お疲れ様です！ これで**本番環境用の JSX の設定**をプロジェクトに追加することができました。

## JSX プリプロセッサを実行する

`src` というディレクトリを作成したうえで、次のコマンドをターミナルから実行してみてください。

```
npx babel --watch src --out-dir . --presets react-app/prod
```

## 補足

`npx` はタイプミスではありません。[npm 5.2 以上で利用可能なパッケージ実行ツール](#)です。

万が一 "You have mistakenly installed the `babel` package" というエラーが表示されたのであれば、[JSX をプロジェクトに追加する](#)のステップがうまく実行できていなかったのかもしれません。今いるディレクトリと同じディレクトリで改めて実行してみてください。

このコマンドは JSX を継続的に監視するため、実行が完了するのを待つ必要はありません。

[このお手本の JSX コード](#)を参考に `src/like_button.js` というファイルを作成すると、先ほど起動したコマンドがブラウザでの実行に適した `like_button.js` に変換してくれます。JSX ファイルを編集したら、自動的に再変換してくれます。

さらにこの変換コマンドのおかげで、古いブラウザの互換性を気にすることなく、クラス構文といったモダンな JavaScript の構文を使うこともできるようになります。このツールは Babel というもので、もっと詳しく知りたければ[公式ドキュメント](#)をご覧になってみてください。

ビルドツールの便利さを体感して、もっとたくさんのことツールに任せたいと思っていただけなら、[次のセクション](#)ではさらにいくつかの人気で扱いやすいツールチェーンを紹介しています。そうでもない場合は... `<script>` タグだけでも十分な機能を果たせます！

# Animation Add-Ons

Note:

`ReactTransitionGroup` and `ReactCSSTransitionGroup` have been moved to the `react-transition-group` package that is maintained by the community. Its `1.x` branch is completely API-compatible with the existing addons. Please file bugs and feature requests in the [new repository](#).

The `ReactTransitionGroup` add-on component is a low-level API for animation, and `ReactCSSTransitionGroup` is an add-on component for easily implementing basic CSS animations and transitions.

## High-level API: ReactCSSTransitionGroup

`ReactCSSTransitionGroup` is a high-level API based on `ReactTransitionGroup` and is an easy way to perform CSS transitions and animations when a React component enters or leaves the DOM. It's inspired by the excellent [ng-animate](#) library.

### Importing

```
import ReactCSSTransitionGroup from 'react-transition-group'; // ES6
var ReactCSSTransitionGroup = require('react-transition-group'); // ES5 with npm

class TodoList extends React.Component {
  constructor(props) {
    super(props);
    this.state = {items: ['hello', 'world', 'click', 'me']};
    this.handleAdd = this.handleAdd.bind(this);
  }

  handleAdd() {
    const newItems = this.state.items.concat([
      prompt('Enter some text')
    ]);
    this.setState({items: newItems});
  }

  handleRemove(i) {
    let newItems = this.state.items.slice();
    newItems.splice(i, 1);
    this.setState({items: newItems});
  }

  render() {
    const items = this.state.items.map((item, i) => (
      <div key={i} onClick={() => this.handleRemove(i)}>
        {item}
      </div>
    ));
    return (
      <div>
```

```

        <button onClick={this.handleAdd}>Add Item</button>
        <ReactCSSTransitionGroup
          transitionName="example"
          transitionEnterTimeout={500}
          transitionLeaveTimeout={300}>
          {items}
        </ReactCSSTransitionGroup>
      </div>
    );
  }
}

```

#### Note:

You must provide the key `attribute` for all children of `ReactCSSTransitionGroup`, even when only rendering a single item. This is how React will determine which children have entered, left, or stayed.

In this component, when a new item is added to `ReactCSSTransitionGroup` it will get the `example-enter` CSS class and the `example-enter-active` CSS class added in the next tick. This is a convention based on the `transitionName` prop.

You can use these classes to trigger a CSS animation or transition. For example, try adding this CSS and adding a new list item:

```

.example-enter {
  opacity: 0.01;
}

.example-enter.example-enter-active {
  opacity: 1;
  transition: opacity 500ms ease-in;
}

.example-leave {
  opacity: 1;
}

.example-leave.example-leave-active {
  opacity: 0.01;
  transition: opacity 300ms ease-in;
}

```

You'll notice that animation durations need to be specified in both the CSS and the render method; this tells React when to remove the animation classes from the element and -- if it's leaving -- when to remove the element from the DOM.

## Animate Initial Mounting

`ReactCSSTransitionGroup` provides the optional prop `transitionAppear`, to add an extra transition phase at the initial mount of the component. There is generally no transition phase at the initial mount as the default value of `transitionAppear` is `false`. The following is an example which passes the prop `transitionAppear` with the value `true`.

```

render() {
  return (
    <ReactCSSTransitionGroup
      transitionName="example"
      transitionAppear={true}
      transitionAppearTimeout={500}
      transitionEnter={false}
      transitionLeave={false}>
      <h1>Fading at Initial Mount</h1>
    </ReactCSSTransitionGroup>
  );
}

```

During the initial mount `ReactCSSTransitionGroup` will get the `example-appear` CSS class and the `example-appear-active` CSS class added in the next tick.

```

.example-appear {
  opacity: 0.01;
}

.example-appear.example-appear-active {
  opacity: 1;
  transition: opacity .5s ease-in;
}

```

At the initial mount, all children of the `ReactCSSTransitionGroup` will `appear` but not `enter`. However, all children later added to an existing `ReactCSSTransitionGroup` will `enter` but not `appear`.

Note:

The prop `transitionAppear` was added to `ReactCSSTransitionGroup` in version `0.13`. To maintain backwards compatibility, the default value is set to `false`.

However, the default values of `transitionEnter` and `transitionLeave` are `true` so you must specify `transitionEnterTimeout` and `transitionLeaveTimeout` by default. If you don't need either enter or leave animations, pass `transitionEnter={false}` or `transitionLeave={false}`.

## Custom Classes

It is also possible to use custom class names for each of the steps in your transitions. Instead of passing a string into `transitionName` you can pass an object containing either the `enter` and `leave` class names, or an object containing the `enter`, `enter-active`, `leave-active`, and `leave` class names. If only the enter and leave classes are provided, the enter-active and leave-active classes will be determined by appending '`-active`' to the end of the class name. Here are two examples using custom classes:

```

// ...
<ReactCSSTransitionGroup
  transitionName={{ 
    enter: 'enter',
    enterActive: 'enterActive',
    leave: 'leave',
    leaveActive: 'leaveActive',
  }}

```

```

        appear: 'appear',
        appearActive: 'appearActive'
    } }>
{item}
</ReactCSSTransitionGroup>

<ReactCSSTransitionGroup
    transitionName={ {
        enter: 'enter',
        leave: 'leave',
        appear: 'appear'
    } }>
{item2}
</ReactCSSTransitionGroup>
// ...

```

## Animation Group Must Be Mounted To Work

In order for it to apply transitions to its children, the `ReactCSSTransitionGroup` must already be mounted in the DOM or the prop `transitionAppear` must be set to `true`.

The example below would **not** work, because the `ReactCSSTransitionGroup` is being mounted along with the new item, instead of the new item being mounted within it. Compare this to the [Getting Started](#) section above to see the difference.

```

render() {
  const items = this.state.items.map((item, i) => (
    <div key={item} onClick={() => this.handleRemove(i)}>
      <ReactCSSTransitionGroup transitionName="example">
        {item}
      </ReactCSSTransitionGroup>
    </div>
  )));
  return (
    <div>
      <button onClick={this.handleAdd}>Add Item</button>
      {items}
    </div>
  );
}

```

## Animating One or Zero Items

In the example above, we rendered a list of items into `ReactCSSTransitionGroup`. However, the children of `ReactCSSTransitionGroup` can also be one or zero items. This makes it possible to animate a single element entering or leaving. Similarly, you can animate a new element replacing the current element. For example, we can implement a simple image carousel like this:

```

import ReactCSSTransitionGroup from 'react-transition-group';

function ImageCarousel(props) {
  return (
    <div>
      <ReactCSSTransitionGroup
        transitionName="carousel"
        transitionEnterTimeout={300}

```

```

        transitionLeaveTimeout={300}>
          <img src={props.imageSrc} key={props.imageSrc} />
        </ReactCSSTransitionGroup>
      </div>
    );
}

```

## Disabling Animations

You can disable animating `enter` or `leave` animations if you want. For example, sometimes you may want an `enter` animation and no `leave` animation, but `ReactCSSTransitionGroup` waits for an animation to complete before removing your DOM node. You can add `transitionEnter={false}` or `transitionLeave={false}` props to `ReactCSSTransitionGroup` to disable these animations.

Note:

When using `ReactCSSTransitionGroup`, there's no way for your components to be notified when a transition has ended or to perform any more complex logic around animation. If you want more fine-grained control, you can use the lower-level `ReactTransitionGroup` API which provides the hooks you need to do custom transitions.

## Low-level API: `ReactTransitionGroup`

### Importing

```

import ReactTransitionGroup from 'react-addons-transition-group' // ES6
var ReactTransitionGroup = require('react-addons-transition-group') // ES5 with
npm

```

`ReactTransitionGroup` is the basis for animations. When children are declaratively added or removed from it (as in the [example above](#)), special lifecycle methods are called on them.

- `componentWillAppear()`
- `componentDidAppear()`
- `componentWillEnter()`
- `componentDidEnter()`
- `componentWillLeave()`
- `componentDidLeave()`

### Rendering a Different Component

`ReactTransitionGroup` renders as a `span` by default. You can change this behavior by providing a `component` prop. For example, here's how you would render a `<ul>`:

```

<ReactTransitionGroup component="ul">
  {/* ... */}
</ReactTransitionGroup>

```

Any additional, user-defined, properties will become properties of the rendered component. For example, here's how you would render a `<ul>` with CSS class:

```
<ReactTransitionGroup component="ul" className="animated-list">
  {/* ... */}
</ReactTransitionGroup>
```

Every DOM component that React can render is available for use. However, `component` does not need to be a DOM component. It can be any React component you want; even ones you've written yourself! Just write `component={List}` and your component will receive `this.props.children`.

## Rendering a Single Child

People often use `ReactTransitionGroup` to animate mounting and unmounting of a single child such as a collapsible panel. Normally `ReactTransitionGroup` wraps all its children in a `span` (or a custom `component` as described above). This is because any React component has to return a single root element, and `ReactTransitionGroup` is no exception to this rule.

However if you only need to render a single child inside `ReactTransitionGroup`, you can completely avoid wrapping it in a `<span>` or any other DOM component. To do this, create a custom component that renders the first child passed to it directly:

```
function FirstChild(props) {
  const childrenArray = React.Children.toArray(props.children);
  return childrenArray[0] || null;
}
```

Now you can specify `FirstChild` as the `component` prop in `<ReactTransitionGroup>` props and avoid any wrappers in the result DOM:

```
<ReactTransitionGroup component={FirstChild}>
  {someCondition ? <MyComponent /> : null}
</ReactTransitionGroup>
```

This only works when you are animating a single child in and out, such as a collapsible panel. This approach wouldn't work when animating multiple children or replacing the single child with another child, such as an image carousel. For an image carousel, while the current image is animating out, another image will animate in, so `<ReactTransitionGroup>` needs to give them a common DOM parent. You can't avoid the wrapper for multiple children, but you can customize the wrapper with the `component` prop as described above.

## Reference

### `componentWillAppear()` [\[#componentwillappear\]](#)

```
componentWillAppear(callback)
```

This is called at the same time as `componentDidMount()` for components that are initially mounted in a `TransitionGroup`. It will block other animations from occurring until `callback` is called. It is only called on the initial render of a `TransitionGroup`.

## `componentDidAppear()` **{#componentdidappear}**

`componentDidAppear()`

This is called after the `callback` function that was passed to `componentWillAppear` is called.

## `componentWillEnter()` **{#componentwillenter}**

`componentWillEnter(callback)`

This is called at the same time as `componentDidMount()` for components added to an existing `TransitionGroup`. It will block other animations from occurring until `callback` is called. It will not be called on the initial render of a `TransitionGroup`.

## `componentDidEnter()` **{#componentdidenter}**

`componentDidEnter()`

This is called after the `callback` function that was passed to `componentWillEnter()` is called.

## `componentWillLeave()` **{#componentwillleave}**

`componentWillLeave(callback)`

This is called when the child has been removed from the `ReactTransitionGroup`. Though the child has been removed, `ReactTransitionGroup` will keep it in the DOM until `callback` is called.

## `componentDidLeave()` **{#componentdidleave}**

`componentDidLeave()`

This is called when the `willLeave` `callback` is called (at the same time as `componentWillUnmount()`).

---

[Go to TOC](#)

# Keyed Fragments

Note:

`React.addons` entry point is deprecated as of React v15.5. We now have first class support for fragments which you can read about [here](#).

## Importing

```
import createFragment from 'react-addons-create-fragment'; // ES6
var createFragment = require('react-addons-create-fragment'); // ES5 with npm
```

## Overview

In most cases, you can use the `key` prop to specify keys on the elements you're returning from `render`. However, this breaks down in one situation: if you have two sets of children that you need to reorder, there's no way to put a key on each set without adding a wrapper element.

That is, if you have a component such as:

```
function Swapper(props) {
  let children;
  if (props.swapped) {
    children = [props.rightChildren, props.leftChildren];
  } else {
    children = [props.leftChildren, props.rightChildren];
  }
  return <div>{children}</div>;
}
```

The children will unmount and remount as you change the `swapped` prop because there aren't any keys marked on the two sets of children.

To solve this problem, you can use the `createFragment` add-on to give keys to the sets of children.

`Array<ReactNode> createFragment(object children)` **#arrayreactnode-createfragmentobject-children**

Instead of creating arrays, we write:

```
import createFragment from 'react-addons-create-fragment';

function Swapper(props) {
  let children;
  if (props.swapped) {
    children = createFragment({
      right: props.rightChildren,
      left: props.leftChildren
    });
  } else {
```

```
children = createFragment({
  left: props.leftChildren,
  right: props.rightChildren
});
}
return <div>{children}</div>;
}
```

The keys of the passed object (that is, `left` and `right`) are used as keys for the entire set of children, and the order of the object's keys is used to determine the order of the rendered children. With this change, the two sets of children will be properly reordered in the DOM without unmounting.

The return value of `createFragment` should be treated as an opaque object; you can use the `React.Children` helpers to loop through a fragment but should not access it directly. Note also that we're relying on the JavaScript engine preserving object enumeration order here, which is not guaranteed by the spec but is implemented by all major browsers and VMs for objects with non-numeric keys.

# Performance Tools

Note:

As of React 16, `react-addons-perf` is not supported. Please use [your browser's profiling tools](#) to get insight into which components re-render.

## Importing

```
import Perf from 'react-addons-perf'; // ES6
var Perf = require('react-addons-perf'); // ES5 with npm
```

## Overview

React is usually quite fast out of the box. However, in situations where you need to squeeze every ounce of performance out of your app, it provides a `shouldComponentUpdate()` method where you can add optimization hints to React's diff algorithm.

In addition to giving you an overview of your app's overall performance, `Perf` is a profiling tool that tells you exactly where you need to put these methods.

See these articles for an introduction to React performance tooling:

- ["How to Benchmark React Components"](#)
- ["Performance Engineering with React"](#)
- ["A Deep Dive into React Perf Debugging"](#)

## Development vs. Production Builds

If you're benchmarking or seeing performance problems in your React apps, make sure you're testing with the [minified production build](#). The development build includes extra warnings that are helpful when building your apps, but it is slower due to the extra bookkeeping it does.

However, the perf tools described on this page only work when using the development build of React. Therefore, the profiler only serves to indicate the *relatively* expensive parts of your app.

## Using Perf

The `Perf` object can be used with React in development mode only. You should not include this bundle when building your app for production.

### Getting Measurements

- `start()`
- `stop()`

- `getLastMeasurements()`

## Printing Results

The following methods use the measurements returned by `Perf.getLastMeasurements()` to pretty-print the result.

- `printInclusive()`
  - `printExclusive()`
  - `printWasted()`
  - `printOperations()`
  - `printDOM()`
- 

## Reference

### `start() {#start}`

### `stop() {#stop}`

```
Perf.start()  
// ...  
Perf.stop()
```

Start/stop the measurement. The React operations in-between are recorded for analyses below. Operations that took an insignificant amount of time are ignored.

After stopping, you will need `Perf.getLastMeasurements()` to get the measurements.

### `getLastMeasurements() {#getlastmeasurements}`

```
Perf.getLastMeasurements()
```

Get the opaque data structure describing measurements from the last start-stop session. You can save it and pass it to the other print methods in `Perf` to analyze past measurements.

#### Note

Don't rely on the exact format of the return value because it may change in minor releases. We will update the documentation if the return value format becomes a supported part of the public API.

### `printInclusive() {#printinclusive}`

```
Perf.printInclusive(measurements)
```

Prints the overall time taken. When no arguments are passed, `printInclusive` defaults to all the measurements from the last recording. This prints a nicely formatted table in the console, like so:

(index)	Owner > component	Inclusive time (ms)	Instances
0	"<root> > App"	15.31	37
1	"App > Box"	6.47	37
2	"Box > Box2"	1.96	37

## printExclusive() `{#printexclusive}`

`Perf.printExclusive(measurements)`

"Exclusive" times don't include the times taken to mount the components: processing props, calling `componentWillMount` and `componentDidMount`, etc.

(index)	Component c...	Total inclu...	Exclusive m...	Exclusive r...	Mount time ...	Render time...	Instances
0	"App"	15.31	1.76	1.23	0.04	0.03	37

## printWasted() `{#printwasted}`

`Perf.printWasted(measurements)`

**The most useful part of the profiler.**

"Wasted" time is spent on components that didn't actually render anything, e.g. the render stayed the same, so the DOM wasn't touched.

(index)	Owner > component	Wasted time (ms)	Instances
0	"<root> > App"	15.317999947001226	37
1	"App > Box"	6.479999952716753	37
2	"Box > Box2"	1.9630000460892916	37

## printOperations() `{#printoperations}`

`Perf.printOperations(measurements)`

Prints the underlying DOM manipulations, e.g. "set innerHTML" and "remove".

(index)	data-reactid	type	args
0	"0"	"set innerHTML"	"<div data-reactid=".0\..."

## printDOM() `{#printdom}`

`Perf.printDOM(measurements)`

This method has been renamed to `printOperations()`. Currently `printDOM()` still exists as an alias but it prints a deprecation warning and will eventually be removed.

---

[Go to TOC](#)

# PureRenderMixin

Note:

`PureRenderMixin` is a legacy add-on. Use `React.PureComponent` instead.

## Importing

```
import PureRenderMixin from 'react-addons-pure-render-mixin'; // ES6
var PureRenderMixin = require('react-addons-pure-render-mixin'); // ES5 with npm
```

## Overview

If your React component's render function renders the same result given the same props and state, you can use this mixin for a performance boost in some cases.

Example:

```
const createReactClass = require('create-react-class');

createReactClass({
  mixins: [PureRenderMixin],

  render: function() {
    return <div className={this.props.className}>foo</div>;
  }
});
```

Under the hood, the mixin implements `shouldComponentUpdate`, in which it compares the current props and state with the next ones and returns `false` if the equalities pass.

Note:

This only shallowly compares the objects. If these contain complex data structures, it may produce false-negatives for deeper differences. Only mix into components which have simple props and state, or use `forceUpdate()` when you know deep data structures have changed. Or, consider using [immutable objects](#) to facilitate fast comparisons of nested data.

Furthermore, `shouldComponentUpdate` skips updates for the whole component subtree. Make sure all the children components are also "pure".

---

[Go to TOC](#)

# Shallow Compare

Note:

`shallowCompare` is a legacy add-on. Use `React.memo` or `React.PureComponent` instead.

## Importing

```
import shallowCompare from 'react-addons-shallow-compare'; // ES6
var shallowCompare = require('react-addons-shallow-compare'); // ES5 with npm
```

## Overview

Before `React.PureComponent` was introduced, `shallowCompare` was commonly used to achieve the same functionality as `PureRenderMixin` while using ES6 classes with React.

If your React component's render function is "pure" (in other words, it renders the same result given the same props and state), you can use this helper function for a performance boost in some cases.

Example:

```
export class SampleComponent extends React.Component {
  shouldComponentUpdate(nextProps, nextState) {
    return shallowCompare(this, nextProps, nextState);
  }

  render() {
    return <div className={this.props.className}>foo</div>;
  }
}
```

`shallowCompare` performs a shallow equality check on the current `props` and `nextProps` objects as well as the current `state` and `nextState` objects.

It does this by iterating on the keys of the objects being compared and returning true when the values of a key in each object are not strictly equal.

`shallowCompare` returns `true` if the shallow comparison for props or state fails and therefore the component should update.

`shallowCompare` returns `false` if the shallow comparison for props and state both pass and therefore the component does not need to update.

---

[Go to TOC](#)

# Shallow Renderer

## インポート

```
import ShallowRenderer from 'react-test-renderer/shallow'; // ES6
var ShallowRenderer = require('react-test-renderer/shallow'); // ES5 with npm
```

## 概要

React の単体テストを実装するとき、Shallow Renderer が役立つでしょう。浅いレンダー (shallow rendering) を使用すると、インスタンス化またはレンダーされていない子コンポーネントの振る舞いを心配することなく、「1 階層深く」レンダーしてレンダーメソッドが返すものを assert できます。これに DOM は必要ありません。

たとえば、以下のコンポーネントがある場合：

```
function MyComponent() {
  return (
    <div>
      <span className="heading">Title</span>
      <Subcomponent foo="bar" />
    </div>
  );
}
```

以下のように assert できます：

```
import ShallowRenderer from 'react-test-renderer/shallow';

// in your test:
const renderer = new ShallowRenderer();
renderer.render(<MyComponent />);
const result = renderer.getRenderOutput();

expect(result.type).toBe('div');
expect(result.props.children).toEqual([
  <span className="heading">Title</span>,
  <Subcomponent foo="bar" />
]);
```

浅いレンダーによるテストには現在いくつかの制限があります。すなわち ref をサポートしていません。

補足：

また Enzyme の [Shallow Rendering API](#) をチェックすることをお勧めします。それは同じ機能上でより良くより高いレベルの API を提供します。

## リファレンス

### shallowRenderer.render() {#shallowrendererrender}

`shallowRenderer.render()` は、テストしているコンポーネントをレンダーするための「場所」と捉えることができ、そこからコンポーネントの出力を抽出できます。

`shallowRenderer.render()` は `root.render()` に似ていますが、DOM を必要とせず、1 階層だけレンダーします。つまり、テスト対象のコンポーネントが持つ子コンポーネントの実装から分離してテストを実行できます。

### shallowRenderer.getRenderOutput() {#shallowrenderergetrenderoutput}

`shallowRenderer.render()` が呼び出された後、`shallowRenderer.getRenderOutput()` を使用して浅くレンダーされた出力を取得できます。

そして出力から得た結果の `assert` を開始できます。

# 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](#) を通じて 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、エンドユーザがコンポーネントを使用するときと同様の方法でコンポーネントを使用するような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。

バージョン 16 以下の React を使用している場合、[Enzyme](#) ライブラリを使うことで 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行えます。

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

## Reference

### act() {#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/client';
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.createRoot(container).render(<Counter />);
  });
  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。
- `recipes` ドキュメントには `act()` がどのように動作するのかについて例や使用法を交えた詳しい解説があります。

## mockComponent() [\[#mockcomponent\]](#)

```

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 `jest.mock()` の使用をおすすめします。

## isElement() [\[#iselement\]](#)

```
isElement(element)
```

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

## isElementOfType() {#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}

```
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}

```
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}

```
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}

```
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}

```
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}

```
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}

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

レンダー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}

```
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}

```
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}

```
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}

```
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.createRoot(domContainer).render(element);
```

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 `useState` 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}

```
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。

# Two-way Binding Helpers

Note:

`LinkedStateMixin` is deprecated as of React v15. The recommendation is to explicitly set the value and change handler, instead of using `LinkedStateMixin`.

## Importing

```
import LinkedStateMixin from 'react-addons-linked-state-mixin'; // ES6
var LinkedStateMixin = require('react-addons-linked-state-mixin'); // ES5 with npm
```

## Overview

`LinkedStateMixin` is an easy way to express two-way binding with React.

In React, data flows one way: from owner to child. We think that this makes your app's code easier to understand. You can think of it as "one-way data binding."

However, there are lots of applications that require you to read some data and flow it back into your program. For example, when developing forms, you'll often want to update some React `state` when you receive user input. Or perhaps you want to perform layout in JavaScript and react to changes in some DOM element size.

In React, you would implement this by listening to a "change" event, read from your data source (usually the DOM) and call `setState()` on one of your components. "Closing the data flow loop" explicitly leads to more understandable and easier-to-maintain programs. See [our forms documentation](#) for more information.

Two-way binding -- implicitly enforcing that some value in the DOM is always consistent with some React `state` -- is concise and supports a wide variety of applications. We've provided `LinkedStateMixin`: syntactic sugar for setting up the common data flow loop pattern described above, or "linking" some data source to React `state`.

Note:

`LinkedStateMixin` is just a thin wrapper and convention around the `onChange / setState()` pattern. It doesn't fundamentally change how data flows in your React application.

## LinkedStateMixin: Before and After

Here's a simple form example without using `LinkedStateMixin`:

```
var createReactClass = require('create-react-class');

var NoLink = createReactClass({
  getInitialState: function() {
    return {message: 'Hello!'};
  },
  handleChange: function(event) {
    this.setState({message: event.target.value});
  },
  render: function() {
    var message = this.state.message;
    return <input type="text" value={message} onChange={this.handleChange}>;
  }
});
```

This works really well and it's very clear how data is flowing, however, with a lot of form fields it could get a bit verbose. Let's use `LinkedStateMixin` to save us some typing:

```
var createReactClass = require('create-react-class');

var WithLink = createReactClass({
  mixins: [LinkedStateMixin],
  getInitialState: function() {
    return {message: 'Hello!'};
  },
  render: function() {
    return <input type="text" valueLink={this.linkState('message')} />;
  }
});
```

`LinkedStateMixin` adds a method to your React component called `linkState()`. `linkState()` returns a `valueLink` object which contains the current value of the React state and a callback to change it.

`valueLink` objects can be passed up and down the tree as props, so it's easy (and explicit) to set up two-way binding between a component deep in the hierarchy and state that lives higher in the hierarchy.

Note that checkboxes have a special behavior regarding their `value` attribute, which is the value that will be sent on form submit if the checkbox is checked (defaults to `on`). The `value` attribute is not updated when the checkbox is checked or unchecked. For checkboxes, you should use `checkedLink` instead of `valueLink`:

```
<input type="checkbox" checkedLink={this.linkState('booleanValue')} />
```

## Under the Hood

There are two sides to `LinkedStateMixin`: the place where you create the `valueLink` instance and the place where you use it. To prove how simple `LinkedStateMixin` is, let's rewrite each side separately to be more explicit.

### valueLink Without LinkedStateMixin

```
var createReactClass = require('create-react-class');

var WithoutMixin = createReactClass({
  getInitialState: function() {
    return {message: 'Hello!'};
```

```

},
handleChange: function(newValue) {
  this.setState({message: newValue});
},
render: function() {
  var valueLink = {
    value: this.state.message,
    requestChange: this.handleChange
  };
  return <input type="text" valueLink={valueLink} />;
}
);

```

As you can see, `valueLink` objects are very simple objects that just have a `value` and `requestChange` prop. And `LinkedStateMixin` is similarly simple: it just populates those fields with a value from `this.state` and a callback that calls `this.setState()`.

## LinkedStateMixin Without valueLink

```

var LinkedStateMixin = require('react-addons-linked-state-mixin');
var createReactClass = require('create-react-class');

var WithoutLink = createReactClass({
  mixins: [LinkedStateMixin],
  getInitialState: function() {
    return {message: 'Hello!'};
  },
  render: function() {
    var valueLink = this.linkState('message');
    var handleChange = function(e) {
      valueLink.requestChange(e.target.value);
    };
    return <input type="text" value={valueLink.value} onChange={handleChange} />;
  }
});

```

The `valueLink` prop is also quite simple. It simply handles the `onChange` event and calls `this.props.valueLink.requestChange()` and also uses `this.props.valueLink.value` instead of `this.props.value`. That's it!

# Immutability Helpers

Note:

`update` is a legacy add-on. Use `immutability-helper` instead.

## Importing

```
import update from 'react-addons-update'; // ES6
var update = require('react-addons-update'); // ES5 with npm
```

## Overview

React lets you use whatever style of data management you want, including mutation. However, if you can use immutable data in performance-critical parts of your application it's easy to implement a fast `shouldComponentUpdate()` method to significantly speed up your app.

Dealing with immutable data in JavaScript is more difficult than in languages designed for it, like [Clojure](#). However, we've provided a simple immutability helper, `update()`, that makes dealing with this type of data much easier, *without* fundamentally changing how your data is represented. You can also take a look at Facebook's [Immutable-js](#) and the [Advanced Performance](#) section for more detail on Immutable-js.

## The Main Idea

If you mutate data like this:

```
myData.x.y.z = 7;
// or...
myData.a.b.push(9);
```

You have no way of determining which data has changed since the previous copy has been overwritten. Instead, you need to create a new copy of `myData` and change only the parts of it that need to be changed. Then you can compare the old copy of `myData` with the new one in `shouldComponentUpdate()` using triple-equals:

```
const newData = deepCopy(myData);
newData.x.y.z = 7;
newData.a.b.push(9);
```

Unfortunately, deep copies are expensive, and sometimes impossible. You can alleviate this by only copying objects that need to be changed and by reusing the objects that haven't changed. Unfortunately, in today's JavaScript this can be cumbersome:

```
const newData = extend(myData, {
  x: extend(myData.x, {
    y: extend(myData.x.y, {z: 7}),
```

```
  },
  a: extend(myData.a, {b: myData.a.b.concat(9)})
});
```

While this is fairly performant (since it only makes a shallow copy of `log n` objects and reuses the rest), it's a big pain to write. Look at all the repetition! This is not only annoying, but also provides a large surface area for bugs.

## update() {#update}

`update()` provides simple syntactic sugar around this pattern to make writing this code easier. This code becomes:

```
import update from 'react-addons-update';

const newData = update(myData, {
  x: {y: {z: {$set: 7}}},
  a: {b: {$push: [9]}}
});
```

While the syntax takes a little getting used to (though it's inspired by [MongoDB's query language](#)) there's no redundancy, it's statically analyzable and it's not much more typing than the mutative version.

The `$`-prefixed keys are called *commands*. The data structure they are "mutating" is called the *target*.

## Available Commands

- `={$push: array}` `push()` all the items in `array` on the target.
- `={$unshift: array}` `unshift()` all the items in `array` on the target.
- `={$splice: array of arrays}` for each item in `arrays` call `splice()` on the target with the parameters provided by the item.
- `={$set: any}` replace the target entirely.
- `={$merge: object}` merge the keys of `object` with the target.
- `={$apply: function}` passes in the current value to the function and updates it with the new returned value.

## Examples

### Simple push

```
const initialArray = [1, 2, 3];
const newArray = update(initialArray, {$push: [4]}); // => [1, 2, 3, 4]
```

`initialArray` is still `[1, 2, 3]`.

### Nested collections

```
const collection = [1, 2, {a: [12, 17, 15]}];
const newCollection = update(collection, {2: {a: {$splice: [[1, 1, 13, 14]]}}});
// => [1, 2, {a: [12, 13, 14, 15]}]
```

This accesses `collection's index 2`, key `a`, and does a splice of one item starting from index `1` (to remove `17`) while inserting `13` and `14`.

## Updating a value based on its current one

```
const obj = {a: 5, b: 3};
const newObj = update(obj, {b: {$apply: function(x) {return x * 2;}}});
// => {a: 5, b: 6}
// This is equivalent, but gets verbose for deeply nested collections:
const newObj2 = update(obj, {b: {$set: obj.b * 2}});
```

## (Shallow) Merge

```
const obj = {a: 5, b: 3};
const newObj = update(obj, {$merge: {b: 6, c: 7}}); // => {a: 5, b: 6, c: 7}
```

# Add-Ons

Note:

`React.addons` entry point is deprecated as of React v15.5. The add-ons have moved to separate modules, and some of them have been deprecated.

The React add-ons are a collection of useful utility modules for building React apps. **These should be considered experimental** and tend to change more often than the core.

- `createFragment`, to create a set of externally-keyed children.

The add-ons below are in the development (unminified) version of React only:

- `Perf`, a performance profiling tool for finding optimization opportunities.
- `ReactTestUtils`, simple helpers for writing test cases.

## Legacy Add-ons

The add-ons below are considered legacy and their use is discouraged. They will keep working in observable future, but there is no further development.

- `PureRenderMixin`. Use `React.PureComponent` instead.
- `shallowCompare`, a helper function that performs a shallow comparison for props and state in a component to decide if a component should update. We recommend using `React.PureComponent` instead.
- `update`. Use `kolodny/immutability-helper` instead.
- `ReactDOMFactories`, pre-configured DOM factories to make React easier to use without JSX.

## Deprecated Add-ons

- `LinkStateMixin` has been deprecated.
- `TransitionGroup` and `CSSTransitionGroup` have been deprecated in favor of [their drop-in replacements](#).

# Using React with Add-ons

You can install the add-ons individually from npm (e.g. `npm install react-addons-create-fragment`) and import them:

```
import createFragment from 'react-addons-create-fragment'; // ES6
var createFragment = require('react-addons-create-fragment'); // ES5 with npm
```

When using React 15 or earlier from a CDN, you can use `react-with-addons.js` instead of `react.js`:

```
<script src="https://unpkg.com/react@15/dist/react-with-addons.js"></script>
```

The add-ons will be available via the `React.addons` global (e.g. `React.addons.TestUtils`).

# CDN リンク

React および ReactDOM は CDN を介して利用することができます。

```
<script crossorigin src="https://unpkg.com/react@18/umd/react.development.js">
</script>
<script crossorigin src="https://unpkg.com/react-dom@18/umd/react-
dom.development.js"></script>
```

上記のバージョンは開発のためだけのものであり、本番環境には適していません。圧縮・最適化された本番バージョンの React は下記のリンクから利用できます。

```
<script crossorigin src="https://unpkg.com/react@18/umd/react.production.min.js">
</script>
<script crossorigin src="https://unpkg.com/react-dom@18/umd/react-
dom.production.min.js"></script>
```

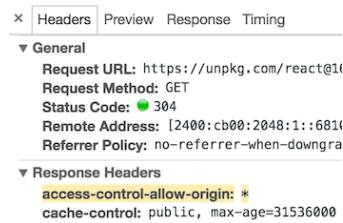
特定のバージョンの `react` と `react-dom` をロードする場合は、`18` の部分をバージョン番号で置き換えてください。

## なぜ `crossorigin` 属性が必要なのか？

CDN を使って React の機能を提供する場合、`crossorigin` 属性を設定することをお勧めします。

```
<script crossorigin src="..."></script>
```

利用している CDN が `Access-Control-Allow-Origin: *` という HTTP ヘッダを設定していることを確認することもお勧めします。



`Access-Control-Allow-Origin: *`

これにより React 16 以降でより優れたエラーハンドリングを利用できます。

---

[Go to TOC](#)

# コード分割

## バンドル

多くの React アプリケーションは、[Webpack](#)、[Rollup](#) や [Browserify](#) などのツールを使ってファイルを「バンドル」しています。バンドルはインポートされたファイルをたどって、それらを 1 つのファイルにまとめるプロセスです。このバンドルされたファイルを Web ページ内に置くことによって、アプリ全体を一度に読み込むことができます。

### 例

#### App:

```
// app.js
import { add } from './math.js';

console.log(add(16, 26)); // 42
```

```
// math.js
export function add(a, b) {
  return a + b;
}
```

#### Bundle:

```
function add(a, b) {
  return a + b;
}

console.log(add(16, 26)); // 42
```

補足：

実際のバンドルはこれとは大幅に異なった見た目になります。

もし [Create React App](#)、[Next.js](#)、[Gatsby](#) やこれらに類するツールを使用している場合、アプリケーションをバンドルするための Webpack のセットアップが最初から含まれています。

そうでない場合は、自分でバンドルを設定する必要があります。設定方法に関しては、Webpack のドキュメントにある [Installation](#) や [Getting Started](#) などを参照してみてください。

# コード分割

バンドルは確かに素晴らしいですが、アプリが大きくなるにつれて、バンドルのサイズも大きくなります。特にサイズの大きなサードパーティ製のライブラリを含む場合は顕著にサイズが増大します。不用意に大きなバンドルを作成してしまいアプリの読み込みに多くの時間がかかるってしまうという事態にならないためにも、常に注意を払い続けなければなりません。

大きなバンドルを不注意に生成してしまわないように、あらかじめコードを「分割」して問題を回避しましょう。Code-Splitting は、[Webpack](#)、[Rollup](#) や [Browserify](#) (`factor-bundle` を使用) などのバンドラによってサポートされている機能であり、実行時に動的にロードされる複数のバンドルを生成することができます。

コード分割は、ユーザが必要とするコードだけを「遅延読み込み」する手助けとなり、アプリのパフォーマンスを劇的に向上させることができます。アプリの全体的なコード量を減らすことはできませんが、ユーザが必要としないコードを読み込まなくて済むため、初期ロードの際に読む込むコード量を削減できます。

## import() {#import}

コード分割をアプリに導入する最も良い手段は動的な `import()` 構文を使用することです。

### Before:

```
import { add } from './math';
console.log(add(16, 26));
```

### After:

```
import("./math").then(math => {
  console.log(math.add(16, 26));
});
```

Webpack がこの構文を見つけると、自動的にアプリのコードを分割します。Create React App を使用している場合はすでに設定がされているため、[すぐに使用を開始することができます](#)。Next.js も同様です。

もし Webpack を自分でセットアップしていた場合には、[Webpack のコード分割に関するガイド](#)を読むと良いでしょう。あなたの Webpack の設定はだいたい[このよう](#)になると思います。

もし Babel を使用している場合は、Babel が動的インポート構文をパースできても変換してしまわないようにする必要があります。そのためには [babel-plugin-syntax-dynamic-import](#) を利用すると良いでしょう。

## React.lazy {#reactlazy}

`React.lazy` 関数を使用すると、動的インポートを通常のコンポーネントとしてレンダーすることができます。

### Before:

```
import OtherComponent from './OtherComponent';
```

### After:

```
const OtherComponent = React.lazy(() => import('./OtherComponent'));
```

このコンポーネントがはじめてレンダーされる時、`OtherComponent` を含むバンドルを自動的にロードしてくれます。

`React.lazy` は動的インポート構文 `import()` を呼び出す関数を引数として取ります。この関数は React コンポーネントを含む `default export` を持つモジュールに解決される `Promise` を返さなければなりません。

遅延コンポーネントは、`Suspense` コンポーネント内でレンダーされる必要があります。これによって、遅延コンポーネントのローディングの待機中にフォールバック用のコンテンツ（ローディングインジケータなど）を表示できます。

```
import React, { Suspense } from 'react';

const OtherComponent = React.lazy(() => import('./OtherComponent'));

function MyComponent() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <OtherComponent />
      </Suspense>
    </div>
  );
}
```

`fallback` プロパティはコンポーネントがロードされるのを待っている間に表示したいあらゆる React 要素を受け取ります。`Suspense` コンポーネントは遅延コンポーネントより上位のどこにでも配置することができます。また、複数の遅延コンポーネントを単一の `Suspense` コンポーネントでラップすることもできます。

```
import React, { Suspense } from 'react';

const OtherComponent = React.lazy(() => import('./OtherComponent'));
const AnotherComponent = React.lazy(() => import('./AnotherComponent'));

function MyComponent() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <section>
          <OtherComponent />
          <AnotherComponent />
        </section>
      </Suspense>
    </div>
  );
}
```

## フォールバックを避ける

既にユーザに表示されているものも含むあらゆるコンポーネントは、レンダーの結果としてサスPENDする可能性があります。画面に表示される内容の一貫性を保つため、既に表示されているコンポーネントがサスPENDした場合、React はツリーを直近の `<Suspense>` バウンダリまで非表示にする必要があります。しかしユーザの観点からはこれは不親切です。

このタブ切り替えの例で考えてみましょう：

```
import React, { Suspense } from 'react';
import Tabs from './Tabs';
import Glimmer from './Glimmer';

const Comments = React.lazy(() => import('./Comments'));
const Photos = React.lazy(() => import('./Photos'));

function MyComponent() {
  const [tab, setTab] = React.useState('photos');
```

```

function handleTabSelect(tab) {
  setTab(tab);
};

return (
  <div>
    <Tabs onTabSelect={handleTabSelect} />
    <Suspense fallback=<Glimmer />>
      {tab === 'photos' ? <Photos /> : <Comments />}
    </Suspense>
  </div>
);
}

```

この例では、タブが 'photos' から 'comments' に切り替わると Comments がサスペンドするため、ユーザは Glimmer (点滅) を見てしまうことになります。これは当然で、ユーザはもう Photos を見たいわけではないし Comments コンポーネントはまだ何もレンダーできないのですから、React はユーザ体験を一貫させるために Glimmer を表示するしかないわけです。

しかしこのようなユーザ体験は望ましくないことがあります。具体的には、新しい UI の準備を行っている間は「古い」UI を表示し続けるほうが望ましい場合があります。新たに導入された `startTransition` API を使うことで、React にこれをさせることができます。

```

function handleTabSelect(tab) {
  startTransition(() => {
    setTab(tab);
  });
}

```

このコードは React に、タブを 'comments' に切り替えるのは緊急性の高い更新ではなく、少し時間のかかる [トランジション](#) である、と伝えています。これにより React は、既存の UI をその場でインタラクティブに保ち、<Comments /> の準備ができたところでそちらを表示するよう切り替えるようになります。詳細は [トランジション](#) を参照してください。

## Error Boundary

もし他のモジュールがロードに失敗した場合（例えば、ネットワークの障害など）、エラーが発生します。その際には `error boundary` を使用することによってこれらのエラーをハンドリングし、エラーの回復やユーザ体験の向上に繋げることができます。error boundary を作成したら、遅延コンポーネントより上位のあらゆる場所で使用でき、ネットワークエラーが発生した際にエラー内容を表示することができます。

```

import React, { Suspense } from 'react';
import MyErrorBoundary from './MyErrorBoundary';

const OtherComponent = React.lazy(() => import('./OtherComponent'));
const AnotherComponent = React.lazy(() => import('./AnotherComponent'));

const MyComponent = () => (
  <div>
    <MyErrorBoundary>
      <Suspense fallback=<div>Loading...</div>>
        <section>
          <OtherComponent />
          <AnotherComponent />
        </section>
    </Suspense>
  </div>
);

```

```
</MyErrorBoundary>
</div>
);
```

## ルーティング単位でのコード分割

アプリ内のどこにコード分割を導入するかを決めるのは少し面倒です。バンドルを均等に分割する場所を確実に選択したいところですが、ユーザ体験を妨げてはなりません。

コード分割を導入するにあたって適している場所はルーティングです。Web を使用するほとんどの人は、多少のロード時間がかかるページ遷移に慣れています。また、ユーザがページ上の他の要素を同時に操作する可能性を減らすよう、ページ全体を一度に再レンダーすることが多いでしょう。

これは [React Router](#) のようなライブラリを使ったアプリに `React.lazy` を使用することでルーティングベースのコード分割を導入する方法の例です。

```
import React, { Suspense, lazy } from 'react';
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';

const Home = lazy(() => import('./routes/Home'));
const About = lazy(() => import('./routes/About'));

const App = () => (
  <Router>
    <Suspense fallback={<div>Loading...</div>}>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
      </Routes>
    </Suspense>
  </Router>
);
```

## 名前付きエクスポート

`React.lazy` は現在デフォルトエクスポートのみサポートしています。インポートしたいモジュールが名前付きエクスポートを使用している場合、それをデフォルトとして再エクスポートする中間モジュールを作成できます。これにより、tree shaking が機能し未使用のコンポーネントを取り込まず済むようにできます。

```
// ManyComponents.js
export const MyComponent = /* ... */;
export const MyUnusedComponent = /* ... */;
```

```
// MyComponent.js
export { MyComponent as default } from "./ManyComponents.js";
```

```
// MyApp.js
import React, { lazy } from 'react';
const MyComponent = lazy(() => import("./MyComponent.js"));
```

# コードベースの概要

このセクションでは React コードベースの構成や規則そして実装についての概要を説明します。

あなたが [React にコントリビュート](#) したい場合に、このガイドがあなたがより快適に変更を行えるように手助けとなる事を願っています。

これらの規約のいずれかをあなたの React アプリケーションで推奨しているというわけではありません。規約の多くは歴史的な理由で存在しており、時間とともに変化する可能性があります。

## 最上位フォルダ

React リポジトリをクローンした後、プロジェクトのルートディレクトリに複数のフォルダがあることに気がつくでしょう：

- `packages` には React リポジトリの全てのパッケージのメタデータ (`package.json` など) や、ソースコード (`src` サブディレクトリ) が含まれています。あなたの変更がコードに関するものなら、各パッケージの `src` サブディレクトリが作業時間のほとんどを過ごす場所となります。
- `fixtures` にはコントリビューター向けの React の小さなテスト用アプリケーションが含まれています。
- `build` は React のビルド出力です。リポジトリには存在しませんが、最初に [React をビルドした](#) 後に `clone` した React ディレクトリに現れます。

ドキュメントは [React 本体とは異なるリポジトリ](#) にホスティングされています。

他にも最上位にあるフォルダがありますが、ほとんどはツールが使用するためのもので、コントリビュートする場合にそれらに直面することはほぼありません。

## テストコードをソースコードのそばに格納する

ユニットテスト用の最上位ディレクトリはありません。代わりに、テスト対象のファイルのすぐ隣の `__tests__` というディレクトリに置いています。

例えば、`setInnerHTML.js` のテストは、すぐ隣の `__tests__/setInnerHTML-test.js` にあります。

## 警告と不变数 (Invariant)

React のコードベースは警告の表示に `console.error` を使用しています：

```
if (__DEV__) {
  console.error('Something is wrong.');
}
```

警告は開発時にのみ有効になります。本番時には警告は完全に取り除かれます。コードの一部の実行を禁止する必要がある場合は、代わりに `invariant` モジュールを使用してください：

```
var invariant = require('invariant');

invariant(
```

```
2 + 2 === 4,
'You shall not pass!'
);
```

`invariant` は `invariant` 内の条件が `false` のときスローされます。

"Invariant" とは "この条件は常に `true` である" ということの単なる言い換えです。アサーションを作成していると考えることができます。

開発バージョンと本番バージョンで同じ動作となるようにしておく事は重要ですので、`invariant` は両方の環境でスローされます。本番バージョンではエラーメッセージは自動的にエラーコードに変換され、バイト数に悪影響が出ることを避けます。

## 開発バージョンと本番バージョン

コードベース内で `__DEV__` 擬似グローバル変数を使用して、開発用のコードブロックを保護することができます。

コンパイル時にインライン化され、CommonJS のビルド時には `process.env.NODE_ENV !== 'production'` というチェックに変換されます。

スタンドアロンなビルドにおいては、minify されないビルドでは `true` となり、minify されたビルドではそれを囲む `if` ブロックごと完全に除去されます。

```
if (__DEV__) {
  // This code will only run in development.
}
```

## Flow

最近私たちはコードベースに `Flow` によるチェックを導入し始めました。ライセンスヘッダで `@flow` アノテーションがマークされているファイルは型チェックをされています。

既存のコードに `Flow` アノテーションを追加するプルリクエストを受けつけています。`Flow` アノテーションは以下のようになります：

```
ReactRef.detachRefs = function(
  instance: ReactInstance,
  element: ReactElement | string | number | null | false,
): void {
  // ...
}
```

可能であれば、新しいコードは `Flow` アノテーションを使うべきです。`yarn flow` を実行することでローカル環境で `Flow` を使ってコードをチェックできます。

## 複数のパッケージ

React は [単一リポジトリ \(monorepo\)](#) です。そのリポジトリには複数の別々のパッケージが含まれているので、それらの変更はまとめて調整でき、issue も 1箇所にまとまっています。

## React コア (core)

React の「コア (core)」は最上位の React API を全て含んでいます。例えば：

- `React.createElement()`
- `React.Component`
- `React.Children`

**React core** にはコンポーネントの定義に必要な API のみが含まれています。リコンシリエーションのアルゴリズムやあらゆるプラットフォーム固有のコードは含まれません。コアは React DOM や React Native コンポーネントに使われています。

React core のコードはソースツリーの `packages/react` に格納されています。npm では `react` パッケージとして利用可能です。対応するスタンドアロンなブラウザ向けのビルドは `react.js` と呼ばれ、`React` というグローバル変数をエクスポートします。

## レンダラ

React は元々 DOM のために作成されましたが、後になって React Native によりネイティブなプラットフォームもサポートするようになりました。これにより React の内部に "レンダラ (renderer)" の概念が導入されました。

レンダラは React ツリーを、基盤となるプラットフォーム固有の呼び出しへと変換する方法を管理します。

レンダラは `packages/` にも格納されています：

- **ReactDOM Renderer** は React コンポーネントを DOM に変換します。トップレベルの `ReactDOM API` を実装し、npm では `react-dom` パッケージとして利用可能です。`react-dom.js` と呼ばれるスタンドアロンなブラウザ向けバンドルとしても使用でき、`ReactDOM` というグローバル変数をエクスポートします。
- **React Native Renderer** は React コンポーネントをネイティブのビューに変換します。React Native の内部で使われています。
- **React Test Renderer** は React コンポーネントを JSON ツリーに変換します。Jest のスナップショットテスト機能で使用され、npm では `react-test-renderer` パッケージとして利用可能です。

その他に公式にサポートしているレンダラは `react-art` だけです。個別の GitHub リポジトリにありましたでしたが、現在はメインのソースツリーに移動しました。

### 補足：

技術的に、`react-native-renderer` は React に React Native の実装とやり取りの方法を教える非常に薄い層です。ネイティブのビューを管理する実際のプラットフォーム固有のコードは、コンポーネントと共に [React Native リポジトリ](#) にあります。

## リコンサイラ (reconciler)

React DOM と React Native のように大幅に異なるレンダラでも多くのロジックを共有する必要があります。特に、ツリーのリコンシリエーションのアルゴリズムは可能なかぎり同じものにして、宣言型レンダリング、独自コンポーネント、state、ライフサイクルメソッドおよび ref がプラットフォーム間で一貫した動作となるようにするべきです。

異なるレンダラ間でコードの一部を共有することでこの課題を解決します。React のこの箇所を "リコンサイラ (reconciler, 差分検出処理)" と呼んでいます。`setState()` のような更新がスケジュールされると、差分検出処理ではツリー内のコンポーネントで `render()` を呼び出して、コンポーネントをマウント、更新、もしくはアンマウントします。

リコンサイラは、パブリック API がないため、個別にパッケージ化されていません。代わりに、それらは React DOM や React Native などのレンダラーによってのみ使用されます。

## スタック (stack) リコンサイラ

"stack" リコンサイラは React 15 およびそれ以前で実装されています。それ以降は使用されませんが、[次のセクション](#) では詳細に説明されています。

## Fiber リコンサイラ

"fiber" リコンサイラは stack リコンサイラが内在的に抱える問題を解決し、いくつかの長年の問題を修正することを目的とした新しい取り組みです。React 16 以降はデフォルトのリコンサイラとなっています。

主な目的は以下の通りです：

- 中止可能な作業を小分けに分割する機能
- 進行中の作業に優先順位を付けたり、再配置や再利用をする機能
- React でレイアウトをサポートするための親子間を行き来しながらレンダーする機能
- `render()` から複数の要素を返す機能
- `error boundary` のサポートの向上

React Fiber のアーキテクチャに関して[ここ](#)や[ここ](#)で読むことができます。React 16 と共にリリースされていますが、非同期機能についてはデフォルトではまだ有効化されていません。

ソースコードは `packages/react-reconciler` に格納されています。

## イベントシステム

React はブラウザ間でのイベント挙動の差異を吸収するため、ネイティブイベントの上にレイヤーを実装しています。ソースコードは `packages/react-dom/src/events` にあります。

## この次は？

React 16 以前のリコンサイラについてより詳細に学ぶには[次のセクション](#)を読んでください。新しいリコンサイラの内部についてはまだドキュメント化していません。

# コンポーネントと props

コンポーネントにより UI を独立した再利用できる部品に分割し、部品それぞれを分離して考えることができます。このページではコンポーネントという概念の導入を行います。[詳細な API リファレンスはこちらで参照できます。](#)

概念的には、コンポーネントは JavaScript の関数と似ています。（"props" と呼ばれる）任意の入力を受け取り、画面上に表示すべきものを記述する React 要素を返します。

## 関数コンポーネントとクラスコンポーネント

コンポーネントを定義する最もシンプルな方法は JavaScript の関数を書くことです：

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

この関数は、データの入った "props"（「プロパティ」の意味）というオブジェクトを引数としてひとつ受け取り、React 要素を返すので、有効な React コンポーネントです。これは文字通り JavaScript の関数ですので、このようなコンポーネントのことを "関数コンポーネント (function component)" と呼びます。

コンポーネントを定義するために [ES6 クラス](#) も使用できます：

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

上記 2 つのコンポーネントは React の視点からは等価です。

関数コンポーネントとクラスコンポーネントには、[次のセクション](#)で説明するようにそれぞれ幾つかの追加機能があります。

## コンポーネントのレンダー

前節では、DOM のタグを表す React 要素のみを扱いました：

```
const element = <div />;
```

しかし、要素はユーザ定義のコンポーネントを表すこともできます：

```
const element = <Welcome name="Sara" />;
```

React がユーザ定義のコンポーネントを見つけた場合、JSX に書かれている属性と子要素を单一のオブジェクトとしてこのコンポーネントに渡します。このオブジェクトのことを "props" と呼びます。

例えば以下のコードではページ上に "Hello, Sara" を表示します：

```

function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}

const root = ReactDOM.createRoot(document.getElementById('root'));
const element = <Welcome name="Sara" />;
root.render(element);

```

### Try it on CodePen

この例で何が起こるのかおさらいしてみましょう。

1. `<Welcome name="Sara" />` という要素を引数として `root.render()` を呼び出します。
2. React は `Welcome` コンポーネントを呼び出し、そのときに `props` として `{name: 'Sara'}` を渡します。
3. `Welcome` コンポーネントは出力として `<h1>Hello, Sara</h1>` 要素を返します。
4. React DOM は `<h1>Hello, Sara</h1>` に一致するよう、DOM を効率的に更新します。

**補足:** コンポーネント名は常に大文字で始めてください。

React は小文字で始まるコンポーネントを DOM タグとして扱います。例えば、`<div />` は HTML の `div` タグを表しますが、`<Welcome />` はコンポーネントを表しており、スコープ内に `Welcome` が存在する必要があります。

この規約の背後にある理由については [JSX を深く理解する](#) を参照してください。

## コンポーネントを組み合わせる

コンポーネントは自身の出力の中で他のコンポーネントを参照できます。これにより、どの詳細度のレベルにおいても、コンポーネントという単一の抽象化を利用できます。ボタン、フォーム、ダイアログ、画面：React アプリでは、これらは共通してコンポーネントとして表現されます。

例えば、`Welcome` を何回もレンダーする `App` コンポーネントを作成できます：

```

function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}

function App() {
  return (
    <div>
      <Welcome name="Sara" />
      <Welcome name="Cahal" />
      <Welcome name="Edite" />
    </div>
  );
}

```

### Try it on CodePen

典型的には、新規の React アプリは階層の一番上に单一の `App` コンポーネントを持っています。しかし、既存のアプリに React を統合する場合は、`Button` のような小さなコンポーネントからボトムアップで始め、徐々にビューの階層構造の頂上に向かって進んでいってよいでしょう。

## コンポーネントの抽出

コンポーネントをより小さなコンポーネントに分割することを恐れないでください。

例えば、この `Comment` コンポーネントについて考えましょう：

```
function Comment(props) {
  return (
    <div className="Comment">
      <div className="UserInfo">
        <img className="Avatar"
          src={props.author.avatarUrl}
          alt={props.author.name}>
      />
      <div className="UserInfo-name">
        {props.author.name}
      </div>
    </div>
    <div className="Comment-text">
      {props.text}
    </div>
    <div className="Comment-date">
      {formatDate(props.date)}
    </div>
  </div>
);
}
```

[Try it on CodePen](#)

これは `props` として `author` (オブジェクト)、`text` (文字列)、および `date` (日付) を受け取り、ソーシャルメディアサイトにおける 1 つのコメントを表します。

これだけのネストがあるため、このコンポーネントの変更には苦労を伴い、また内部の個々の部品を再利用することも困難です。ここからいくつかのコンポーネントを抽出しましょう。

まず、`Avatar` を抽出します：

```
function Avatar(props) {
  return (
    <img className="Avatar"
      src={props.user.avatarUrl}
      alt={props.user.name}>
  );
}
```

`Avatar` は、自身が `Comment` の内側でレンダーされているということを知っている必要はありません。なので `props` の名前として、`author` ではなく `user` というもっと一般的な名前を付けました。

コンポーネントが使用される文脈ではなく、コンポーネント自身からの観点で `props` の名前を付けることをお勧めします。

これで `Comment` をほんの少しシンプルにできます：

```
function Comment(props) {
  return (
    <div className="Comment">
      <div className="UserInfo">
        <Avatar user={props.author} />
        <div className="UserInfo-name">
          {props.author.name}
        </div>
      </div>
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}
```

次に、ユーザ名の隣の `Avatar` をレンダーするために使われる、`UserInfo` コンポーネントを抽出しましょう。

```
function UserInfo(props) {
  return (
    <div className="UserInfo">
      <Avatar user={props.user} />
      <div className="UserInfo-name">
        {props.user.name}
      </div>
    </div>
  );
}
```

これにより `Comment` をさらにシンプルにできます：

```
function Comment(props) {
  return (
    <div className="Comment">
      <UserInfo user={props.author} />
      <div className="Comment-text">
        {props.text}
      </div>
      <div className="Comment-date">
        {formatDate(props.date)}
      </div>
    </div>
  );
}
```

### Try it on CodePen

コンポーネントの抽出は最初は面倒な仕事のように思えますが、再利用できるコンポーネントをパレットとして持つておくことは、アプリケーションが大きくなれば努力に見合った利益を生みます。役に立つ経験則として、UI の一部（`Button`、`Panel`、`Avatar` など）が複数回使われている場合、またはその UI 自体が複雑（`App`、`FeedStory`、`Comment` など）である場合、それらは別のコンポーネントに抽出する有力な候補であるといえます。

## Props は読み取り専用

コンポーネントを [関数で宣言する](#)か [クラスで宣言する](#)かに関わらず、自分自身の `props` は決して変更してはいけません。この `sum` 関数を考えましょう：

```
function sum(a, b) {  
  return a + b;  
}
```

このような関数は入力されたものを変更しようとせず、同じ入力に対し同じ結果を返すので "純粋 (pure)" であると言われます。

対照的に、以下の関数は自身への入力を変更するため純関数ではありません：

```
function withdraw(account, amount) {  
  account.total -= amount;  
}
```

React は柔軟ですが、1 つだけ厳格なルールがあります：

**全ての React コンポーネントは、自己の `props` に対して純関数のように振る舞わねばなりません。**

もちろんアプリケーションの UI は動的で、時間に応じて変化するものです。[次の章](#)では、"state" という新しい概念を紹介します。state により React コンポーネントは上述のルールを壊すことなく、時間と共にユーザのアクション、ネットワークのレスポンスや他の様々な事に反応して、出力を変更することができます。

---

[Go to TOC](#)

# コンポジション vs 繙承

React は強力なコンポジションモデルを備えており、コンポーネント間のコードの再利用には継承よりもコンポジションをお勧めしています。

この章では、React を始めて間もない開発者が継承に手を出した時に陥りがちないいくつかの問題と、コンポジションによりその問題がどのように解決できるのかについて考えてみます。

## 子要素の出力 (Containment)

コンポーネントの中には事前には子要素を知らないものもあります。これは `Sidebar` や `Dialog` のような汎用的な "入れ物" をあらわすコンポーネントではよく使われています。

このようなコンポーネントでは特別な `children` という `props` を使い、以下のようにして受け取った子要素を出力することができます。

```
function FancyBorder(props) {
  return (
    <div className={'FancyBorder FancyBorder-' + props.color}>
      {props.children}
    </div>
  );
}
```

これにより他のコンポーネントから JSX をネストすることで任意の子要素を渡すことができます。

```
function WelcomeDialog() {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">
        Welcome
      </h1>
      <p className="Dialog-message">
        Thank you for visiting our spacecraft!
      </p>
    </FancyBorder>
  );
}
```

[Try it on CodePen](#)

`<FancyBorder>` JSX タグの内側のあらゆる要素は `FancyBorder` に `children` という `props` として渡されます。`FancyBorder` は `<div>` の内側に `{props.children}` をレンダーするので、渡された要素が出力されます。

あまり一般的ではありませんが、複数箇所に子要素を追加したいケースも考えられます。そのようなケースでは以下のように `children` の `props` の代わりに独自の `props` を作成して渡すことができます。

```
function SplitPane(props) {
  return (
    <div className="SplitPane">
      <div className="SplitPane-left">
        {props.left}
```

```

        </div>
      <div className="SplitPane-right">
        {props.right}
      </div>
    </div>
  );
}

function App() {
  return (
    <SplitPane
      left={
        <Contacts />
      }
      right={
        <Chat />
      }
    >
  );
}

```

[Try it on CodePen](#)

`<Contacts />` や `<Chat />` のような React の要素はただのオブジェクトなので、他のあらゆるデータと同様に `props` として渡すことができます。このアプローチは他のライブラリで言うところの `slot` に似ていると感じるかもしれません。React のコンポーネントに `props` として渡せるものに制限はありません。

## 特化したコンポーネント (Specialization)

コンポーネントを他のコンポーネントの "特別なケース" として考えることができます。例えば、`WelcomeDialog` は `Dialog` の特別なケースと言えるでしょう。

React ではこれもコンポジションで実現できます。汎用的なコンポーネントに `props` を渡して設定することで、より特化したコンポーネントを作成することができます。

```

function Dialog(props) {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">
        {props.title}
      </h1>
      <p className="Dialog-message">
        {props.message}
      </p>
    </FancyBorder>
  );
}

function WelcomeDialog() {
  return (
    <Dialog
      title="Welcome"
      message="Thank you for visiting our spacecraft!" />
  );
}

```

[Try it on CodePen](#)

コンポジションはクラスとして定義されたコンポーネントでも同じように動作します。

```
function Dialog(props) {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">
        {props.title}
      </h1>
      <p className="Dialog-message">
        {props.message}
      </p>
      {props.children}
    </FancyBorder>
  );
}

class SignUpDialog extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.handleSignUp = this.handleSignUp.bind(this);
    this.state = {login: ''};
  }

  render() {
    return (
      <Dialog title="Mars Exploration Program"
        message="How should we refer to you?">
        <input value={this.state.login}
          onChange={this.handleChange} />
        <button onClick={this.handleSignUp}>
          Sign Me Up!
        </button>
      </Dialog>
    );
  }

  handleChange(e) {
    this.setState({login: e.target.value});
  }

  handleSignUp() {
    alert(`Welcome aboard, ${this.state.login}!`);
  }
}
```

[Try it on CodePen](#)

## 継承はどうなの？

Facebook では、何千というコンポーネントで React を使用していますが、コンポーネント継承による階層構造が推奨されるケースは全く見つかっていません。

props とコンポジションにより、コンポーネントの見た目と振る舞いを明示的かつ安全にカスタマイズするのに十分な柔軟性が得られます。コンポーネントはどのような props でも受け付けることができ、それはプリミティブ値でも、React 要素でも、あるいは関数であってもよい、ということに留意してください。

コンポーネント間で非 UI 機能を再利用したい場合は、それを別の JavaScript モジュールに抽出することをお勧めします。コンポーネントはその関数やオブジェクト、クラスなどを継承することなくインポートすることで使用することができるでしょう。

---

[Go to TOC](#)

# 条件付きレンダー

React ではあなたの必要なふるまいをカプセル化した独立したコンポーネントを作ることができます。そして、あなたのアプリケーションの状態に応じて、その一部だけを描画することが可能です。

React における条件付きレンダーは JavaScript における条件分岐と同じように動作します。`if` もしくは[条件演算子](#)のような JavaScript 演算子を使用して現在の状態を表す要素を作成すれば、React はそれに一致するように UI を更新します。

以下の 2 つのコンポーネントを考えてみましょう：

```
function UserGreeting(props) {
  return <h1>Welcome back!</h1>;
}

function GuestGreeting(props) {
  return <h1>Please sign up.</h1>;
}
```

ユーザがログインしているかどうかによって、これらのコンポーネントの一方だけを表示する `Greeting` コンポーネントを作成しましょう：

```
function Greeting(props) {
  const isLoggedIn = props.isLoggedIn;
  if (isLoggedIn) {
    return <UserGreeting />;
  }
  return <GuestGreeting />;
}

const root = ReactDOM.createRoot(document.getElementById('root'));
// Try changing to isLoggedIn={true}:
root.render(<Greeting isLoggedIn={false} />);
```

[Try it on CodePen](#)

この例では `isLoggedIn` プロパティの値によって異なる挨拶メッセージを表示します。

## 要素変数

要素を保持しておくために変数を使うことができます。これは、出力の他の部分を変えずにコンポーネントの一部を条件付きでレンダーしたい時に役立ちます。

ログアウトとログインボタンを表す以下の 2 つの新しいコンポーネントを考えましょう：

```
function LoginButton(props) {
  return (
    <button onClick={props.onClick}>
      Login
    </button>
  );
}
```

```
function LogoutButton(props) {
  return (
    <button onClick={props.onClick}>
      Logout
    </button>
  );
}
```

以下の例では、`LoginControl` というステート付きコンポーネントを作成します。

`LoginControl` は現在の state によって `<LoginButton />` もしくは `<LogoutButton />` の一方をレンダーします。加えて、前の例の `<Greeting />` もレンダーします：

```
class LoginControl extends React.Component {
  constructor(props) {
    super(props);
    this.handleLoginClick = this.handleLoginClick.bind(this);
    this.handleLogoutClick = this.handleLogoutClick.bind(this);
    this.state = {isLoggedIn: false};
  }

  handleLoginClick() {
    this.setState({isLoggedIn: true});
  }

  handleLogoutClick() {
    this.setState({isLoggedIn: false});
  }

  render() {
    const isLoggedIn = this.state.isLoggedIn;
    let button;

    if (isLoggedIn) {
      button = <LogoutButton onClick={this.handleLogoutClick} />;
    } else {
      button = <LoginButton onClick={this.handleLoginClick} />;
    }

    return (
      <div>
        <Greeting isLoggedIn={isLoggedIn} />
        {button}
      </div>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<LoginControl />);
```

### Try it on CodePen

変数を宣言して `if` 文を使用することはコンポーネントを条件的にレンダーするなかなか良い方法ではありますが、より短い構文を使いたくなる時もあります。以下で述べるように、JSX でインラインで条件を記述する方法がいくつか存在します。

## 論理 `&&` 演算子によるインライン If

中括弧で囲むことで、[JSX に式を埋め込む](#) ことができます。これには JavaScript の論理 `&&` 演算子も含まれます。これは条件に応じて要素を含めたいというときに便利です。

```
function Mailbox(props) {
  const unreadMessages = props.unreadMessages;
  return (
    <div>
      <h1>Hello!</h1>
      {unreadMessages.length > 0 &&
        <h2>
          You have {unreadMessages.length} unread messages.
        </h2>
      }
    </div>
  );
}

const messages = ['React', 'Re: React', 'Re:Re: React'];

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Mailbox unreadMessages={messages} />);
```

[Try it on CodePen](#)

これが動作するのは、JavaScript では `true && expression` は必ず `expression` と評価され、`false && expression` は必ず `false` と評価されるからです。

従って、条件部分が `true` であれば、`&&` の後に書かれた要素が出力に現れます。もし `false` であれば、React はそれを無視して飛ばします。

`falsy` な値を返した場合、`&&` の後の要素の評価はスキップされますが、`falsy` な値そのものは返されるということに注意してください。以下の例では `<div>0</div>` がレンダーメソッドから返されます。

```
render() {
  const count = 0;
  return (
    <div>
      {count && <h1>Messages: {count}</h1>}
    </div>
  );
}
```

## 条件演算子によるインライン If-Else

条件的に要素をレンダーするもうひとつの方法は JavaScript の `condition ? true : false` 条件演算子を利用することです。

以下の例では条件演算子を用いて、条件に応じてテキストの小さなブロックをレンダーします。

```
render() {
  const isLoggedIn = this.state.isLoggedIn;
  return (
    <div>
      The user is <b>{isLoggedIn ? 'currently' : 'not'}</b> logged in.
    </div>
  );
}
```

```

        </div>
    );
}

```

より大きな式にも適用することができますが、何が起こっているのか分かりづらくなります：

```

render() {
  const isLoggedIn = this.state.isLoggedIn;
  return (
    <div>
      {isLoggedIn
        ? <LogoutButton onClick={this.handleLogoutClick} />
        : <LoginButton onClick={this.handleLoginClick} />
      }
    </div>
  );
}

```

普通の JavaScript を書くときと同様、あなたとチームが読みやすいと思えるものに合わせて、適切なスタイルを選択してください。条件が複雑になりすぎたら、[コンポーネントを抽出](#)するべきタイミングかもしれない、ということにも留意してください。

## コンポーネントのレンダーを防ぐ

稀なケースですが、他のコンポーネントによってレンダーされているにも関わらず、コンポーネントが自分のことを隠したい、ということがあるかもしれません。その場合はレンダー出力の代わりに `null` を返すようにしてください。

以下の例では、`<WarningBanner />` バナーは `warn` と呼ばれるプロパティの値に応じてレンダーされます。そのプロパティの値が `false` なら、コンポーネントはレンダーされません：

```

function WarningBanner(props) {
  if (!props.warn) {
    return null;
  }

  return (
    <div className="warning">
      Warning!
    </div>
  );
}

class Page extends React.Component {
  constructor(props) {
    super(props);
    this.state = {showWarning: true};
    this.handleToggleClick = this.handleToggleClick.bind(this);
  }

  handleToggleClick() {
    this.setState(state => ({
      showWarning: !state.showWarning
    }));
  }

  render() {
    return (
      <div>

```

```
<WarningBanner warn={this.state.showWarning} />
<button onClick={this.handleToggleClick}>
  {this.state.showWarning ? 'Hide' : 'Show'}
</button>
</div>
);
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Page />);
```

### Try it on CodePen

コンポーネントの `render` メソッドから `null` を返してもコンポーネントのライフサイクルメソッドの発火には影響しません。例えば `componentDidMount` は変わらず呼び出されます。

# コンテクスト

コンテクストは各階層で手動でプロパティを下に渡すことなく、コンポーネントツリー内でデータを渡す方法を提供します。

典型的な React アプリケーションでは、データは `props` を通してトップダウン（親から子）で渡されますが、このようなやり方は、アプリケーション内の多くのコンポーネントから必要とされる特定のタイプのプロパティ（例：ロケール設定、UI テーマ）にとっては面倒です。コンテクストはツリーの各階層で明示的にプロパティを渡すことなく、コンポーネント間でこれらの様な値を共有する方法を提供します。

- [コンテクストをいつ使用すべきか](#)
- [コンテクストを使用する前に](#)
- [API](#)
  - [React.createContext](#)
  - [Context.Provider](#)
  - [Class.contextType](#)
  - [Context.Consumer](#)
  - [Context.displayName](#)
- [例](#)
  - [動的なコンテクスト](#)
  - [ネストしたコンポーネントからコンテクストを更新する](#)
  - [複数のコンテクストを使用する](#)
- [注意事項](#)
- [レガシーな API](#)

## コンテクストをいつ使用すべきか

コンテクストは、ある React コンポーネントのツリーに対して「グローバル」とみなすことができる、現在の認証済みユーザー・テーマ・優先言語といったデータを共有するために設計されています。例えば、以下のコードでは `Button` コンポーネントをスタイルする為に、手動で `"theme"` プロパティを通してしています。

```
embed:context/motivation-problem.js
```

コンテクストを使用することで、中間の要素群を経由してプロパティを渡すことを避けることができます。

```
embed:context/motivation-solution.js
```

## コンテクストを使用する前に

コンテクストは主に、何らかのデータが、ネストレベルの異なる多くのコンポーネントからアクセスできる必要がある時に使用されます。コンテクストはコンポーネントの再利用をより難しくする為、慎重に利用してください。

もし多くの階層を経由していくつかの `props` を渡すことを避けたいだけであれば、[コンポーネントコンポジション](#)は多くの場合、コンテクストよりシンプルな解決策です。

例えば、深くネストされた `Link` と `Avatar` コンポーネントがプロパティを読み取ることが出来るように、`user` と `avatarSize` プロパティをいくつかの階層下へ渡す `Page` コンポーネントを考えてみましょう。

```
<Page user={user} avatarSize={avatarSize} />
// ... Page コンポーネントは以下をレンダー ...
<PageLayout user={user} avatarSize={avatarSize} />
// ... PageLayout コンポーネントは以下をレンダー ...
<NavigationBar user={user} avatarSize={avatarSize} />
// ... NavigationBar コンポーネントは以下をレンダー ...
<Link href={user.permalink}>
  <Avatar user={user} size={avatarSize} />
</Link>
```

最終的に `Avatar` コンポーネントだけがプロパティを必要としているのであれば、多くの階層を通して `user` と `avatarSize` プロパティを下に渡すことには冗長に感じるかもしれません。また、`Avatar` コンポーネントが上のコンポーネントから追加のプロパティを必要とする時はいつでも、全ての間の階層にも追加しないといけないことも厄介です。

**コンテクストを使用せずに**この問題を解決する 1 つの手法は、`Avatar` コンポーネント自身を渡すようにするというもので、そうすれば間のコンポーネントは `user` や `avatarSize` プロパティを知る必要はありません。

```
function Page(props) {
  const user = props.user;
  const userLink = (
    <Link href={user.permalink}>
      <Avatar user={user} size={props.avatarSize} />
    </Link>
  );
  return <PageLayout userLink={userLink} />;
}

// これで以下のようになります。
<Page user={user} avatarSize={avatarSize} />
// ... Page コンポーネントは以下をレンダー ...
<PageLayout userLink={...} />
// ... PageLayout コンポーネントは以下をレンダー ...
<NavigationBar userLink={...} />
// ... NavigationBar コンポーネントは以下をレンダー ...
{props.userLink}
```

この変更により、一番上の `Page` コンポーネントだけが、`Link` と `Avatar` コンポーネントの `user` と `avatarSize` の使い道について知る必要があります。

この制御の反転はアプリケーション内で取り回す必要のあるプロパティの量を減らし、ルートコンポーネントにより多くの制御を与えることにより、多くのケースでコードを綺麗にすることができます。しかし、このような制御の反転がすべてのケースで正しい選択となるわけではありません。ツリー内の上層に複雑性が移ることは、それら高い階層のコンポーネントをより複雑にして、低い階層のコンポーネントに必要以上の柔軟性を強制します。

コンポーネントに対して 1 つの子までという制限はありません。複数の子を渡したり、子のために複数の別々の「スロット」を持つことさえできます。[ドキュメントはここにあります。](#)

```
function Page(props) {
  const user = props.user;
  const content = <Feed user={user} />;
  const topBar = (
    <NavigationBar>
```

```

    <Link href={user.permalink}>
      <Avatar user={user} size={props.avatarSize} />
    </Link>
  </NavigationBar>
);
return (
<PageLayout
  topBar={topBar}
  content={content}
/>
);
}
}

```

このパターンは、そのすぐ上の親から子を切り離す必要がある多くのケースにとって十分です。レンダーの前に子が親とやり取りする必要がある場合、さらにレンダープロップと合わせて使うことができます。

しかし、たまに同じデータがツリー内の異なるネスト階層にある多くのコンポーネントからアクセス可能であることが必要となります。コンテクストはそのようなデータとその変更を以下の全てのコンポーネントへ「ブロードキャスト」できます。コンテクストを使うことが他の手法よりシンプルである一般的な例としては、現在のロケール、テーマ、またはデータキャッシュの管理が挙げられます。

## API

### React.createContext {#reactcreatecontext}

```
const MyContext = React.createContext(defaultValue);
```

コンテクストオブジェクトを作成します。React がこのコンテクストオブジェクトが登録されているコンポーネントをレンダーする場合、ツリー内の最も近い上位の一貫する `Provider` から現在のコンテクストの値を読み取ります。

`defaultValue` 引数は、コンポーネントがツリー内の上位に対応するプロバイダを持っていない場合のみ使用されます。このようなデフォルト値は、ラップしない単独でのコンポーネントのテストにて役に立ちます。補足：`undefined` をプロバイダの値として渡しても、コンシューマコンポーネントが `defaultValue` を使用することはありません。

### Context.Provider {#contextprovider}

```
<MyContext.Provider value={/* 何らかの値 */}>
```

全てのコンテクストオブジェクトにはプロバイダ (Provider) コンポーネントが付属しており、これによりコンシューマコンポーネントはコンテクストの変更を購読できます。

プロバイダコンポーネントは `value` プロパティを受け取り、これが子孫であるコンシューマコンポーネントに渡されます。1 つのプロバイダは多くのコンシューマと接続することができます。プロバイダはネストしてツリー内のより深い位置で値を上書きすることができます。

プロバイダの子孫の全てのコンシューマは、プロバイダの `value` プロパティが変更されるたびに再レンダーされます。プロバイダからその子孫コンシューマ (`.contextType` や `useContext` を含む) への伝播は `shouldComponentUpdate` メソッドの影響を受けないため、コンシューマは祖先のコンポーネントが更新をスキップしている場合でも更新されます。

変更は、`Object.is` と同じアルゴリズムを使用し、新しい値と古い値の比較によって判断されます。

## 補足

この方法で変更の有無を判断するため、オブジェクトを `value` として渡した場合にいくつかの問題が発生する可能性があります。詳細は[注意事項](#)を参照してください。

## Class.contextType {#classcontexttype}

```
class MyClass extends React.Component {
  componentDidMount() {
    let value = this.context;
    /* MyContextの値を使用し、マウント時に副作用を実行します */
  }
  componentDidUpdate() {
    let value = this.context;
    /* ... */
  }
  componentWillUnmount() {
    let value = this.context;
    /* ... */
  }
  render() {
    let value = this.context;
    /* MyContextの値に基づいて何かをレンダーします */
  }
}
MyClass.contextType = MyContext;
```

クラスの `contextType` プロパティには `React.createContext()` により作成されたコンテクストオブジェクトを指定することができます。これにより、`this.context` を使って、そのコンテクストタイプの最も近い現在値を利用できます。レンダーカスタマイズ機能を含むあらゆるライフサイクルメソッドで参照できます。

## 補足:

この API では、1 つのコンテクストだけ登録することができます。もし 2 つ以上を読み取る必要がある場合、[複数のコンテクストを使用する](#)を参照してください。

実験的な `public class fields syntax` を使用している場合、**static** クラスフィールドを使用することで `contextType` を初期化することができます。

```
class MyClass extends React.Component {
  static contextType = MyContext;
  render() {
    let value = this.context;
    /* 値に基づいて何かをレンダーします */
  }
}
```

## Context.Consumer {#contextconsumer}

```
<MyContext.Consumer>
  {value => /* コンテクストの値に基づいて何かをレンダーリングします */}
</MyContext.Consumer>
```

コンテクストの変更を購読する React コンポーネントです。このコンポーネントを使うことで、[関数コンポーネント](#)内でコンテクストを購読することができます。

`function as a child` が必要です。この関数は現在のコンテクストの値を受け取り、React ノードを返します。この関数に渡される引数 `value` は、ツリー内の上位で一番近いこのコンテクスト用のプロバイダの `value` プロパティと等しくなります。このコンテクスト用のプロバイダが上位に存在しない場合、引数の `value` は `createContext()` から渡された `defaultValue` と等しくなります。

補足

"function as a child" パターンについてさらに情報が必要な場合は[レンダープロップ](#)を参照してください。

## Context.displayName {#contextdisplayname}

コンテクストオブジェクトは `displayName` という文字列型のプロパティを有しています。React DevTools はこの文字列を利用してコンテクストの表示のしかたを決定します。

例えば以下のコンポーネントは DevTools で `MyDisplayName` と表示されます。

```
const MyContext = React.createContext(/* some value */);
MyContext.displayName = 'MyDisplayName';

<MyContext.Provider> // "MyDisplayName.Provider" in DevTools
<MyContext.Consumer> // "MyDisplayName.Consumer" in DevTools
```

## 例

### 動的なコンテクスト

テーマに動的な値を使用したより複雑な例：

**theme-context.js** [embed:context/theme-detailed-theme-context.js](#)

**themed-button.js** [embed:context/theme-detailed-themed-button.js](#)

**app.js** [embed:context/theme-detailed-app.js](#)

### ネストしたコンポーネントからコンテクストを更新する

コンポーネントツリーのどこか深くネストされたコンポーネントからコンテクストを更新することはよく必要になります。このケースでは、コンテクストを通して下に関数を渡すことで、コンシューマがコンテクストを更新可能にすることができます。

**theme-context.js** `embed:context/updating-nested-context-context.js`**theme-toggler-button.js** `embed:context/updating-nested-context-theme-toggler-button.js`**app.js** `embed:context/updating-nested-context-app.js`

## 複数のコンテクストを使用する

コンテクストの再レンダーを高速に保つために、React は各コンテクストのコンシューマをツリー内の別々のノードにする必要があります。

`embed:context/multiple-contexts.js`

2 つ以上のコンテクストの値が一緒に使用されることが多い場合、両方を提供する独自のレンダープロップコンポーネントの作成を検討した方が良いかもしれません。

## 注意事項

コンテクストは参照の同一性を使用していつ再レンダーするかを決定するため、プロバイダの親が再レンダーするときにコンシューマで意図しないレンダーを引き起こす可能性があるいくつかの問題があります。例えば以下のコードでは、新しいオブジェクトが `value` に対して常に作成されるため、プロバイダが再レンダーするたびにすべてのコンシューマを再レンダーしてしまいます。

`embed:context/reference-caveats-problem.js`

この問題を回避するためには、親の `state` に値をリフトアップします。

`embed:context/reference-caveats-solution.js`

## レガシーな API

### 補足

React は以前に実験的なコンテクスト API を公開していました。その古い API は全ての 16.x 系のリリースでサポートされる予定ですが、アプリケーションで使用しているのであれば、新しいバージョンにマイグレーションすべきです。レガシーな API は将来の React メジャー版で削除されます。[レガシーなコンテクストのドキュメントはここにあります。](#)

# 新しい React アプリを作る

もっとも良いユーザ・開発体験を得るために統合されたツールチェインを使いましょう。

このページではいくつかの人気のある React ツールチェインを説明します。これは次のようなタスクに役立ちます：

- 大量のファイルとコンポーネントでスケールする
- npm を通してサードパーティライブラリを利用する
- よくある間違いを早期に発見する
- 開発環境で CSS と JS をライブ編集する
- 本番用の出力を最適化する

このページで推奨されているツールチェインは始めるにあたって設定が不要です。

## ツールチェインが必要ない場合

あなたが上記のような問題を経験していなかったり、まだ JavaScript のツールを利用するのに慣れていない場合、HTML ページに簡単な `<script>` タグで React を追加することを検討してください（JSX の利用も検討してみてください）。

これは既存のウェブサイトに React を統合する最も簡単な方法でもあります。あなたが役立つと思えばいつでもより大きなツールチェインを追加できます。

## 推奨するツールチェイン

React チームは主に以下のソリューションを推奨します：

- React を学習中か、新しいシングルページアプリケーションを作成したい場合、Create React App を利用してください
- Node.js でサーバサイドでレンダーされたウェブサイトを構築するなら、Next.js を試してください
- 静的なコンテンツ中心のウェブサイトを構築するなら、Gatsby を試してください
- コンポーネントライブラリの構築や既存のコードベースへの統合をするなら、その他の柔軟なツールチェインを試してください

## Create React App

Create React App は React を学習するのに快適な環境であり、React で新しいシングルページアプリケーションを作成するのに最も良い方法です。

開発環境をセットアップして最新の JavaScript の機能を使えるようにし、快適な開発体験を提供し、そして本番環境用の最適化を行います。あなたのマシンに `Node >= 14.0.0` 及び `npm >= 5.6` の環境が必要です。プロジェクトを作成するには次を実行します：

```
npx create-react-app my-app  
cd my-app  
npm start
```

## 補足

最初の行の `npx` は打ち間違いではありません -- これは `npm 5.2` から利用できるパッケージランナーツールです。

Create React App はバックエンドのロジックやデータベース接続は扱いません。フロントエンドのビルドパイプラインを構築するだけであり、バックエンドに関しては好きなものを組み合わせて使って構いません。内部では [Babel](#) と [webpack](#) を利用していますが、それらについて知る必要はありません。

本番環境にデプロイする準備ができたら、`npm run build` を実行すれば、`build` フォルダ内に最適化されたアプリケーションのビルドが生成されます。Create React App の詳細については、[該当ツールの README](#) および [ユーザガイド](#) を参照してください。

## Next.js

[Next.js](#) は React を使って静的なサイトやサーバサイドでレンダーされるアプリケーションを構築する場合に人気のある軽量フレームワークです。すぐに使えるスタイルおよびルーティングのソリューションを含み、サーバ環境として [Node.js](#) を利用することを想定しています。

Next.js の[オフィシャルガイド](#)を参照してください。

## Gatsby

[Gatsby](#) は React で静的なウェブサイトを作成するのに最も良い方法です。React コンポーネントを使用しながらも、事前レンダーされた HTML と CSS を出力することで最速のロード時間を保証します。

Gatsby の[オフィシャルガイド](#) および [スターターキットのギャラリー](#)を参照してください。

## 他の柔軟なツールチェイン

以下のツールチェインはより大きな柔軟性や選択肢を提供します。経験豊富なユーザにこれらを推奨します。

- [Neutrino](#) は [webpack](#) のパワーとプリセットのシンプルさを兼ね備えています。プリセットには [React アプリ](#) と [React コンポーネント](#)用のものがあります。
- [Nx](#) はフルスタックの monorepo 開発用ツールキットであり、React、Next.js、[Express](#) などのビルドインサポートを有しています。
- [Parcel](#) は高速な、ゼロ設定のウェブアプリケーションバンドラであり、[React](#) と共に利用できます。
- [Razzle](#) は設定不要のサーバレンダリングフレームワークでありながら、Next.js よりも柔軟性があります。

## ゼロからツールチェインを作成する

JavaScript ビルドツールチェインは一般的に次から成ります：

- **パッケージマネジャ。** [Yarn](#) や [npm](#) など。サードパーティのパッケージの広大なエコシステムを利用でき、それらを簡単にインストールしたりアップデートしたりできます。

- **バンドラ。** [webpack](#) や [Parcel](#) など。モジュール化されたコードを書けるようになり、それを小さなパッケージにまとめてバンドルしてロード時間の最適化を行います。
- **コンパイラー。** [Babel](#) など。未だ動作している古いブラウザでもモダンな JavaScript コードを書いて動作させることができます。

ゼロから独自の JavaScript ツールチェインを設定したい場合、[こちらのガイドをチェック](#)すると Create React App の機能の一部を再現できます。

カスタムしたツールチェインは忘れずに[本番環境用に正しく設定](#)してください。

# Cross-origin Errors

Note:

The following section applies only to the development mode of React. Error handling in production mode is done with regular try/catch statements.

In [development mode](#), React uses a global `error` event handler to preserve the "pause on exceptions" behavior of browser DevTools. It also logs errors to the developer console.

If an error is thrown from a [different origin](#) the browser will mask its details and React will not be able to log the original error message. This is a security precaution taken by browsers to avoid leaking sensitive information.

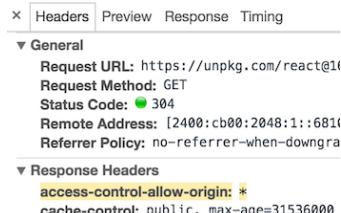
You can simplify the development/debugging process by ensuring that errors are thrown with a same-origin policy. Below are some common causes of cross-origin errors and ways to address them.

## CDN

When loading React (or other libraries that might throw errors) from a CDN, add the `crossorigin` attribute to your `<script>` tags:

```
<script crossorigin src="..."></script>
```

Also ensure the CDN responds with the `Access-Control-Allow-Origin: *` HTTP header:



`Access-Control-Allow-Origin: *`

## Webpack

### Source maps

Some JavaScript bundlers may wrap the application code with `eval` statements in development. (For example Webpack will do this if `devtool` is set to any value containing the word "eval".) This may cause errors to be treated as cross-origin.

If you use Webpack, we recommend using the `cheap-module-source-map` setting in development to avoid this problem.

## Code splitting

If your application is split into multiple bundles, these bundles may be loaded using JSONP. This may cause errors thrown in the code of these bundles to be treated as cross-origin.

To resolve this, use the `crossOriginLoading` setting in development to add the `crossorigin` attribute to the `<script>` tags generated for the JSONP requests.

---

[Go to TOC](#)

# 設計原則

このドキュメントを作成したのは、React が何をして何をしないのか、そして開発理念がどのようなものなのかをより理解できるようにするためです。私たちはコミュニティへの貢献を見ることにわくわくしています。しかしこれらの原則の 1つ以上に違反する道を選ぶことはありません。

## 注意:

このドキュメントは React を強く理解していることを前提としています。React のコンポーネントやアプリケーションではなく、React 自体の設計原則について説明しています。

React の紹介については、代わりに [React の流儀](#)を調べてください。

## コンポジション

React の主な機能はコンポーネントのコンポジションです。異なる人々によって書かれたコンポーネントは一緒にうまく動くべきです。コードベース全体に波及する変更を引き起こすことなく、コンポーネントに機能を追加できるということが重要です。

たとえば、コンポーネントを使用する側を変更せずに、コンポーネントにローカル state を導入することが可能であるべきです。同様に、必要に応じて初期化と終了処理を任意のコンポーネントに追加できるべきです。

コンポーネントで state またはライフサイクルメソッドを使用することについて「悪い」ことは何もありません。他の強力な機能と同様に適度に使用する必要がありますが、私たちはそれらを削除するつもりはありません。それどころか、それらは React を有用にするものとして不可欠な部分であると思います。将来的には[より関数型的なパターン](#)を使用可能にするかもしれません、ローカルステートとライフサイクルメソッドの両方がそのモデルの一部になるでしょう。

コンポーネントはしばしば「単なる関数」と表現されますが、私たちの見解では、それらは有用であるために単なる関数以上である必要があります。React では、コンポーネントは組み合わせ可能な動作を記述します。そしてこれには、レンダリング、ライフサイクル、および state が含まれます。[Relay](#) のようないくつかの外部ライブラリは、データの依存関係を記述するといったような他の責任をコンポーネントに付け加えます。それらのアイデアが何らかの形で React に改めて取り込まれるということも有り得ます。

## 共通の抽象化

一般的に私たちはユーザ側で実装できる[機能の追加をできるだけ避けます](#)。使用されない無駄なライブラリコードであなたのアプリを肥大化させたくありません。ただし、これには例外があります。

たとえば、もし React がローカル state やライフサイクルメソッドをサポートしなかったなら、人々はそれらに対して独自の抽象化を作成したことでしょう。複数の抽象化が競合している場合、React はどちらかの利用を勧めたりどちらか一方のみにある特性を利用したりすることができます。最小公倍数の機能でやっていかなければなりません。

これが、React 自体に機能を追加することがある理由です。多くのコンポーネントが互換性のない、または非効率的な方法で特定の機能を実装していることに気づいたら、React にそれを織り込むかもしれません。しかし私たちは気軽にはそれをしません。それをするときは抽象化レベルを上げることがエコシステム全体に利益をもたらすと確信しているからです。state、ライフサイクルメソッド、クロスプラウザのイベント正規化などがその好例です。

私たちは常にそのような改善提案をコミュニティと話し合います。それらの議論のいくつかは React イシュートラッカーの "big picture" ラベルで見つけることができます。

## 避難ハッチ

React は実用主義的です。それは Facebook で書かれた製品のニーズによって推進されています。関数型プログラミングなど、まだ完全には主流になっていない、いくつかのパラダイムによって影響を受けますが、さまざまなスキルや経験レベルを持つ幅広い開発者がアクセスしやすいことがプロジェクトの明確な目標です。

私たちが好みないパターンを非推奨にしたい場合は非推奨にする前に、既存のすべてのユースケースを検討し、[コミュニティに代替案について教育](#)することが私たちの責任です。アプリを構築するのに有用なパターンを宣言的に表現するのが難しい場合は、[命令的な API を提供](#)します。多くのアプリで必要と思われるものに最適な API を見つけられないときは、[一時的な標準未満の作業用 API](#) を提供します（後で取り除くことが可能で、将来の改善の余地がある場合に限ります）。

## 安定性

私たちは API の安定性を重視しています。Facebook には、React を使った 5 万以上のコンポーネントがあります。Twitter や Airbnb を含む他の多くの会社もまた React のヘビーユーザです。これが私たちが公開された API や振る舞いを変更することに大抵消極的である理由です。

しかし、「何も変わらない」という意味での安定性を人々は過大評価している、と私たちは考えます。それはすぐに停滞に変わってしまいます。代わりに、「本番環境では頻繁に使用され、何かが変わったときには明確な（できれば自動化された）移行方法がある」という意味での安定性を優先します。

あるパターンを非推奨にすることは、Facebook 内部でその使用法を調べ、非推奨警告を追加します。これにより変化の影響が評価できます。変更が時期尚早であり、準備が整うところまでコードベースを持っていくのにさらに戦略的に検討する必要があると分かった場合、変更を差し戻すことがあります。

変更があまり破壊的でなく、移行戦略がすべてのユースケースで実行可能であると確信している場合は、非推奨の警告をオープンソースコミュニティに公開します。私たちは Facebook 以外の多くの React ユーザと密接に連絡を取り合っており、人気のあるオープンソースプロジェクトを監視し、非推奨警告の修正を手引きしています。

Facebook の React コードベースのサイズが非常に大きいことを考えると、社内での移行が成功することは、他の企業でも問題ないことを示す良い指標となります。それでも、時々人々は私達が考えていなかった新たなユースケースを指摘するため、私たちはそれらのための避難ハッチを追加したり、アプローチを再考したりします。

正当な理由がない限り、私たちは何も非推奨にすることはありません。非推奨警告は時々ライラさせられるということを認識していますが、それを追加するのは、私たちとコミュニティの多くの人々が価値あると考える改善と新機能追加のための道が、非推奨化によって開かれるからです。

たとえば、React 15.2.0 で[未知の DOM props に関する警告](#)を追加しました。多くのプロジェクトがこの影響を受けました。しかしこの警告を修正することは、React に[カスタム属性](#)のサポートを導入できるようにするために重要でした。私たちが追加するすべての非推奨化の背景には、このような理由があります。

非推奨警告を追加すると、現在のメジャーバージョンの残りの部分では警告のまま残り、[次のメジャーバージョンでは動作が変更されます](#)。繰り返し行われる手作業が多い場合は、変更の大部分を自動化する [codemod](#) スクリプトをリリースします。Codemod を使用すると、大規模なコードベースでも移行を滞りなく進めることができます。それらを使用することをお勧めします。

私たちがリリースした codemod は、[react-codemod](#) リポジトリで見つけることができます。

## 相互運用性

私たちは既存のシステムとの相互運用性と段階的な導入に高い価値を置いています。Facebook は巨大な非 React のコードベースを持っています。Facebook の Web サイトでは、XHP と呼ばれるサーバサイドのコンポーネントシステム、React よりも前に開発された内部 UI ライブラリ、そして React 自体を組み合わせて使用しています。私たちにとって重要なことは、どの製品チームでも、賭けるようにコードを書き換えるのではなく、[小さな機能に対して React を使い始める](#)ことができるということです。

これが、React がミュータブルなモデルを扱うための避難ハッチを提供し、他の UI ライブラリと一緒にうまく機能しようとする理由です。既存の命令的な UI を宣言型コンポーネントにラップすることも、その逆も可能です。これは段階的な採用には不可欠です。

## スケジューリング

コンポーネントが関数として記述されても、React を使うときはそれらを直接呼び出さないでください。すべてのコンポーネントは[何をレンダーする必要があるかの説明書き](#)を返し、その説明書きには `<LikeButton>` のようなユーザ作成のコンポーネントと `<div>` のようなプラットフォーム固有のコンポーネントの両方を含めることができます。将来のある時点で `<LikeButton>` を「展開」して、コンポーネントのレンダー結果に従って UI ツリーを実際に再帰的に変更するのを、React の責務です。

これは微妙な違いですが強力なものです。あなたがコンポーネント関数を呼び出さずに React が呼び出します。それは React は必要に応じて呼び出しを遅らせる権限があるということを意味します。現在の React の実装ではツリーを再帰的に調べて、1 回のイベントループの間に更新されたツリー全体のレンダー関数を呼び出します。しかし、将来的には[フレームのドロップを避けるため](#)にいくつかの更新を遅らせるかもしれません。

これは React の設計の共通テーマです。いくつかの人気のあるライブラリは、新しいデータが利用可能になったときに計算が実行される「プッシュ」アプローチを実装しています。しかし React は、計算が必要になるまで遅らせることができる「プル」アプローチを採用しています。

React は汎用的なデータ処理ライブラリではありません。ユーザインターフェイスを構築するためのライブラリです。アプリ内において React は、どの計算が今すぐ必要でどの計算がそうでないのかを知ることができる特殊な位置づけにある、と私たちは考えています。

何かが画面外にある場合は、それに関連するロジックを遅らせることができます。データがフレームレートよりも早く到着する場合は、合体してバッチ更新することができます。フレームを落とさないように、重要度の低いバックグラウンド作業（ネットワークからロードされたばかりの新しいコンテンツのレンダーなど）よりも、ユーザの操作による操作（ボタンクリックによるアニメーションなど）を優先できます。

念のために言うと、今の時点ではこれらの可能性は実現していません。しかし、このようなことを自由に行いたいということが、私たちがスケジューリングを制御したい理由であり、`setState()` が非同期である理由です。概念的には、`setState()` を「更新のスケジュール」だと考えています。

ユーザが関数型リアクティブプログラミングのいくつかのバリエーションで一般的な「プッシュ」ベースのパラダイムでビューを直接構成させた場合、スケジューリングに対する制御を得るのが難しくなります。「糊付け部分」のコードは私たちが管理したいのです。

React の主な目標は、React 内部に戻る前に実行されるユーザコードの量を最小限にすることです。これは React が UI について知っていることに従ってスケジュールしたり小分けに作業を分割したりする能力を保持することを保証します。

React は完全に「リアクティブ」であることを望んでいないため、React は "Schedule" と呼ばれるべきだったというチーム内の冗談があります。

## 開発体験

良い開発者経験を提供することは私達にとって重要です。

たとえば、Chrome と Firefox で React コンポーネントツリーを調べることができる [React DevTools](#) をメンテしています。私たちは、それが Facebook エンジニアとコミュニティの両方に大きな生産性向上をもたらしていると聞いています。

私たちは開発者向けの有用な警告を提供するために、一層の努力をしています。たとえば、React は開発中にブラウザが理解できない方法でタグをネストした場合、または API で一般的な入力ミスをした場合に警告します。開発者向けの警告とそれに関連するチェックが、React の開発版が製品版より遅い主な理由です。

Facebook で内部的に見られる使用パターンは、よくある間違いとは何か、そしてそれらを早期に防ぐ方法を理解するのに役立ちます。私たちが新しい機能を追加するとき、私たちはよくある間違いを予想してそれらについて警告しようとします。

私たちは開発者の体験を向上させる方法を常に探しています。体験をさらに良くするために、ぜひ皆さんの提案を聞き、貢献を受け入れたいと思っています。

## デバッグ

問題が発生した場合は、コードベースでその間違いの原因を突き止めるための "パンくず" (breadcrumb) を作成することが重要です。React では、props と state がそのようなパンくずです。

画面に問題がある場合は、React DevTools を開いてレンダーを担当するコンポーネントを見つけ、次に props と state が正しいかどうかを確認できます。そうであれば、問題はコンポーネントの `render()` 関数、または `render()` によって呼び出される関数にあることがわかります。これで問題が切り分けられました。

state が間違っている場合、問題はこのファイル内の `setState()` の呼び出しの 1 つによって引き起こされていることがわかります。これも、通常は 1 つのファイル内に `setState()` の呼び出しは数回しかないため、検索と修正が比較的簡単です。

props が間違っている場合は、インスペクタでツリーを上にたどり、悪い props を渡して最初に「井戸に毒を入れた」犯人のコンポーネントを探します。

任意の UI を生成した元データまで props と state の現在値を使って追跡できるというこの能力は React にとって非常に重要です。state がクロージャやコンビネータに「閉じ込め」られておらず、React で直接利用できることは明示的な設計目標です。

UI は動的ですが、`props` と `state` の同期的な `render()` 関数により、デバッグ作業が単なる当て推量から、退屈ながら有限の手順になると信じています。これにより複雑なアニメーションのようないくつかのユースケースがより困難になりますが、React ではこの制約を保持したいと思います。

## 設定

グローバルな実行時設定オプションは問題があることがわかりました。

例えば、`React.configure(options)` や `React.register(component)` のような関数を実装することが時々要求されます。しかし、これは複数の問題を引き起こし、私たちはそれらに対する良い解決策を知りません。

誰かがサードパーティのコンポーネントライブラリからそのような関数を呼び出すとどうなりますか？ある React アプリに別の React アプリが埋め込まれていて、それらの望ましい設定に互換性がない場合はどうなりますか？サードパーティコンポーネントは特定の設定をどのように必須にしますか？グローバル設定はコンポジションではうまく機能しないと考えています。コンポジションは React の中心であるため、コードでグローバル設定を提供しません。

ただし、ビルドレベルでグローバル設定をいくつか提供します。たとえば、開発ビルトと本番ビルトを別々に提供しています。将来 [プロファイリングビルトを追加する](#)かもしれませんし、また、他のビルドフラグの検討を受け入れています。

## DOM を超えて

私たちは React の価値を、バグの少ないコンポーネントを書いてうまく構成することができるという点に見てています。DOM は React のオリジナルのレンダーターゲットですが、[React Native](#) は Facebook とコミュニティの両方において同じくらい重要です。

レンダラーに依存しないことは React の重要な設計上の制約です。それは内部表現にいくらかのオーバーヘッドを追加します。その一方で、コアへの改善はすべてのプラットフォームに行きわたります。

単一のプログラミングモデルを持つことで、プラットフォームではなく製品を中心エンジニアリングチームを形成できます。これまでのところ、そのトレードオフは私たちにとって価値があります。

## 実装

可能な限り洗練された API を提供しようとしています。しかし実装がエレガントであることにはそれほど関心がありません。現実の世界は完璧には程遠いので、ユーザが醜いコードを書かなくて済むのであれば、合理的な範囲で醜いコードをライブラリに入れることを選びます。新しいコードを評価するときには、正しく、パフォーマンスが高く、優れた開発者体験を提供する実装を求めます。優雅さは二の次です。

私たちは賢いコードより退屈なコードを好みます。コードは使い捨てであり、しばしば変更されます。したがって、それが絶対に必要でない限り、新しい内部抽象化を導入しないことが重要です。移動、変更、削除が容易な冗長コードは、時期尚早に抽象化され変更が難しいエレガントなコードよりも優先されます。

## ツールへの最適化

いくつかの一般的に使用される API は冗長な名前を持っています。例えば、`didMount()` や `onMount()` の代わりに `componentDidMount()` を使います。これは意図的です。目的は、ライブラリとのやり取りのポイントをよく見えるようにすることです。

Facebook のような大規模なコードベースでは、特定の API の使用を検索できることが非常に重要です。他と区別しやすい冗長な名前を大切にしています。特に、控えめに使用する必要がある機能についてはそれが重要です。例えば、`dangerouslySetInnerHTML` をコードレビューで見逃すことは難しいでしょう。

私たちは破壊的な変更を加える際に [codemods](#) に依存しているため、検索を最適化することも重要です。膨大な自動化された変更をコードベース全体に適用するのが簡単で安全であることを望みます。区別しやすい冗長な名前を使用すると、これを実現できます。同様に、他と区別可能な名前を使用すると、潜在的な誤検知を心配することなく、React の使用に関するカスタムの [lint ルール](#)を簡単に作成できます。

[JSX](#) も同様の役割を果たします。React では必須ではありませんが、美観上および実用上の理由から、Facebook で広く使用されています。

私たちのコードベースでは、JSX はそれらが React 要素ツリーを扱っているというツールへの明白なヒントを提供します。これにより、[定数要素の巻き上げ](#)、安全な [lint](#) および [codemod](#) 内部コンポーネントの使用などのビルト時の最適化を追加したり、[JSX ソースの場所](#)を警告に含めることができます。

## ドックフーディング

私たちはコミュニティによって提起された問題に取り組むために最善を尽くします。しかし、私たちは Facebook も内部で経験している問題を優先する可能性があります。直感に反するかもしれません、私たちはこれこそがコミュニティが React に賭けることができる主な理由であると思います。

大量に内部で使用していることで、React が明日消えないという自信が得られます。React は Facebook の問題を解決するために Facebook で作成されました。それは Facebook に確かなビジネス価値をもたらし、多くの製品で使用されています。[ドックフーディング](#)によって、私たちのビジョンが鮮明なままであり、私たちが今後も焦点を絞った方向性を持てるようになります。

これは、コミュニティが提起した問題を無視しているという意味ではありません。たとえば、Facebook 内部で依存していないにも関わらず、React に [Web Components](#) および [SVG](#) のサポートを追加しました。私達は積極的に[皆さんの問題点を聞き](#)、私たちの能力の及ぶ限りで[それらに対処](#)します。コミュニティは React を私たちにとって特別なものにしており、私たちもお返しとして喜んでコミュニティに貢献したいと考えています。

Facebook で多くのオープンソースプロジェクトをリリースした後、みんなを同時に幸せにすることを試みるが焦点が不十分なプロジェクトを生み出してもうまく成長しないということを学びました。代わりに、少人数の観客を選んで満足させることに集中すると最終的には良い効果があることがわかりました。これこそまさに私たちが React を使って行ったことであり、そして今のところ Facebook 製品チームが遭遇した問題を解決することは、オープンソースコミュニティにもうまく還元されています。

このアプローチの欠点は、React の初期体験の良し悪しなど、Facebook チームが対処する必要がないことに十分に焦点を当てることができないことです。私たちはこのことを強く認識しており、以前にオープンソースプロジェクトで行ったのと同じ過ちを犯さずに、コミュニティのすべての人に利益をもたらすようなやり方でどのような改善ができるかを考えています。

---

[Go to TOC](#)

# Error Boundary

かつて、コンポーネント内で発生した JavaScript エラーは React の内部状態を破壊し、以降のレンダーで不可解なエラーを引き起こしていました。このようなエラーはアプリケーションコード中のどこか前の段階で発生したエラーによって引き起こされますが、React はエラーをコンポーネント内で適切に処理する方法を提供していなかったため回復できませんでした。

## error boundary とは

UI の一部に JavaScript エラーがあってもアプリ全体が壊れてはいけません。React ユーザがこの問題に対応できるように、React 16 では “error boundary” という新しい概念を導入しました。

error boundary は自身の子コンポーネントツリーで発生した JavaScript エラーをキャッチし、エラーを記録し、クラッシュしたコンポーネントツリーの代わりにフォールバック用の UI を表示する React コンポーネントです。error boundary は配下のツリー全体のレンダー中、ライフサイクルメソッド内、およびコンストラクタ内で発生したエラーをキャッチします。

### 補足

error boundary は以下のエラーをキャッチしません：

- イベントハンドラ（[詳細](#)）
- 非同期コード（例：`setTimeout` や `requestAnimationFrame` のコールバック）
- サーバサイドレンダリング
- （子コンポーネントではなく）error boundary 自身がスローしたエラー

クラスコンポーネントに、ライフサイクルメソッドの `static getDerivedStateFromError()` か `componentDidCatch()` のいずれか（または両方）を定義すると、error boundary になります。`static getDerivedStateFromError()` はエラーがスローされた後にフォールバック UI をレンダーするために使用します。`componentDidCatch()` はエラー情報をログに記録するために使用します。

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    // Update state so the next render will show the fallback UI.
    return { hasError: true };
  }

  componentDidCatch(error, errorInfo) {
    // You can also log the error to an error reporting service
    logErrorToMyService(error, errorInfo);
  }

  render() {
```

```

    if (this.state.hasError) {
      // You can render any custom fallback UI
      return <h1>Something went wrong.</h1>;
    }

    return this.props.children;
}
}

```

使用する際は通常のコンポーネントとして扱います：

```

<ErrorBoundary>
  <MyWidget />
</ErrorBoundary>

```

error boundary はコンポーネントに対して JavaScript の `catch {}` ブロックのように動作します。error boundary になれるのはクラスコンポーネントだけです。実用上、一度だけ error boundary を定義してそれをアプリケーションの至るところで使用することがよくあります。

**error boundary** は配下のツリー内のコンポーネントで発生したエラーのみをキャッチすることに注意してください。error boundary は自身で起こるエラーをキャッチできません。error boundary がエラーメッセージのレンダーに失敗した場合、そのエラーは最も近い上位の error boundary に伝搬します。この動作もまた、JavaScript の `catch {}` ブロックの動作と似ています。

## ライブデモ

[error boundary を宣言して利用する例](#)を確認してください。

## error boundary を配置すべき場所

error boundary の粒度はあなた次第です。サーバサイドフレームワークがクラッシュを処理する際によく見られるように、最上位のルートコンポーネントをラップしてユーザーに “Something went wrong” メッセージを表示してもいいでしょう。各ウィジェットを個別にラップしてアプリケーションの残りの部分をクラッシュから守るのもいいでしょう。

## エラーがキャッチされなかった場合の新しい動作

この変更には重要な意味があります。React 16 から、どの **error boundary** でもエラーがキャッチされなかった場合に React コンポーネントツリー全体がアンマウントされるようになりました。

この決定については議論がありましたが、我々の経験上、壊れた UI をそのまま表示しておくことは、完全に削除してしまうよりももっと悪いことです。例えば、Messenger のような製品において壊れた UI を表示したままにしておくと、誰かが誤って別の人へメッセージを送ってしまう可能性があります。同様に、支払いアプリで間違った金額を表示することは、何も表示しないよりも悪いことです。

この変更のため、React 16 に移行すると、これまで気付かれていなかったアプリケーションの既存の不具合が明らかになることでしょう。error boundary を追加することで、問題が発生したときのユーザ体験を向上できます。

例えば、Facebook Messenger はサイドバー、情報パネル、会話ログ、メッセージ入力欄といったコンテンツを個別の error boundary でラップしています。これらの UI エリアの一部のコンポーネントがクラッシュしても、残りの部分はインタラクティブなままであります。

また、本番環境で発生したキャッチされなかった例外について知つて修正できるように、JS エラー報告サービスを利用（もしくは自身で構築）することもお勧めします。

## コンポーネントのスタックトレース

React 16 は開発時に、レンダー中に起こった全てのエラーをコンソールに出力します（アプリケーションが誤ってエラーを握り潰してしまっても出力します）。そこではエラーメッセージと JavaScript のスタックに加えて、コンポーネントのスタックトレースも提供します。これにより、コンポーネントツリーのどこでエラーが発生したのかが正確にわかります：

```
▶ React caught an error thrown by BuggyCounter. You should fix this error in your code. react-dom.development.js:7708
React will try to recreate this component tree from scratch using the error boundary you provided, ErrorBoundary.

Error: I crashed!

The error is located at:
  in BuggyCounter (created by App)
  in ErrorBoundary (created by App)
  in div (created by App)
  in App
```

コンポーネントスタックトレースにはファイル名と行番号も出力できます。Create React App のプロジェクトではこれがデフォルトで有効になっています：

```
▶ React caught an error thrown by BuggyCounter. You should fix this error in your code. react-dom.development.js:7708
React will try to recreate this component tree from scratch using the error boundary you provided, ErrorBoundary.

Error: I crashed!

The error is located at:
  in BuggyCounter (at App.js:26)
  in ErrorBoundary (at App.js:21)
  in div (at App.js:8)
  in App (at index.js:5)
```

Create React App を使用しない場合は、[このプラグイン](#)を手動で Babel の設定に追加してください。ただし、この機能は開発専用であり、**本番では必ず無効化しなければならない**ことに注意してください。

### 補足

スタックトレースで表示されるコンポーネント名は `Function.name` プロパティに依存します。このプロパティをネイティブで提供しない古いブラウザやデバイス（IE 11 など）をサポートする場合は、アプリケーションバンドルに `Function.name` のポリフィル（`function.name-polyfill` など）を含めることを検討してください。もしくは、全てのコンポーネントに `displayName` プロパティを明示的に設定することもできます。

## try/catch について

`try` / `catch` は素晴らしいですが、命令型のコードでのみ動作します：

```
try {
  showButton();
} catch (error) {
  // ...
}
```

一方、React コンポーネントは宣言型であり、何がレンダーされるべきなのかを指定します：

<Button />

error boundary は React の宣言型という性質を保持しつつ、期待通りの動作をします。例えば、`componentDidUpdate` メソッドで発生したエラーがツリー内のどこか深い場所にある `setState` によって引き起こされていた場合でも、最も近い error boundary にそのことが正しく伝播します。

## イベントハンドラについて

error boundary はイベントハンドラ内で発生したエラーをキャッチしません。

イベントハンドラ内のエラーから回復するのに error boundary は不要です。レンダーメソッドやライフサイクルメソッドとは異なり、イベントハンドラはレンダー中には実行されません。そのためイベントハンドラ内でエラーが発生しても、React が画面に表示する内容は変わりません。

イベントハンドラ内のエラーをキャッチする必要がある場合は、普通の JavaScript の `try / catch` 文を使用してください：

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = { error: null };
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    try {
      // Do something that could throw
    } catch (error) {
      this.setState({ error });
    }
  }

  render() {
    if (this.state.error) {
      return <h1>Caught an error.</h1>
    }
    return <button onClick={this.handleClick}>Click Me</button>
  }
}
```

上記の例は標準の JavaScript の動作説明であって error boundary を使用していないことに注意してください。

## React 15 からの命名の変更

React 15 は error boundary を異なるメソッド名 (`unstable_handleError`) で非常に限定的にサポートしていました。このメソッドはもう動作しないため、16 ベータ版リリース以降はコードを `componentDidCatch` に変更する必要があります。

この変更について、自動的にコードを移行できる [codemod](#) が提供されています。

# Error Decoder

In the minified production build of React, we avoid sending down full error messages in order to reduce the number of bytes sent over the wire.

We highly recommend using the development build locally when debugging your app since it tracks additional debug info and provides helpful warnings about potential problems in your apps, but if you encounter an exception while using the production build, this page will reassemble the original text of the error.

# AJAX と API

## AJAX コールをする方法は？

任意の AJAX ライブラリを React と共に利用可能です。人気のあるものとしては、[Axios](#)、[jQuery AJAX](#)、ブラウザ組み込みの `window.fetch` などがあります。

## コンポーネントのどのライフサイクルで AJAX コールすべきか？

AJAX コールによるデータ取得は `componentDidMount` のライフサイクルメソッドで行うべきです。データ取得後に `setState` でコンポーネントを更新できるようにするためです。

## 例：ローカル state に AJAX の通信結果をセットする

下記のコンポーネントは、`componentDidMount` で AJAX コールして得られたデータをローカルコンポーネントの `state` に流し込んでいます。

このサンプル API が返す JSON オブジェクトはこのようになります：

```
{
  "items": [
    { "id": 1, "name": "Apples", "price": "$2" },
    { "id": 2, "name": "Peaches", "price": "$5" }
  ]
}
```

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      error: null,
      isLoading: false,
      items: []
    };
  }

  componentDidMount() {
    fetch("https://api.example.com/items")
      .then(res => res.json())
      .then(
        (result) => {
          this.setState({
            isLoading: true,
            items: result.items
          });
        },
        // 補足：コンポーネント内のバグによる例外を隠蔽しないためにも
        // catch() ブロックの代わりにここでエラーハンドリングすることが重要です
        (error) => {
          this.setState({
            isLoading: true,
            error
          });
        }
      );
  }
}
```

```

}

render() {
  const { error, isLoading, items } = this.state;
  if (error) {
    return <div>Error: {error.message}</div>;
  } else if (!isLoading) {
    return <div>Loading...</div>;
  } else {
    return (
      <ul>
        {items.map(item => (
          <li key={item.id}>
            {item.name} {item.price}
          </li>
        ))}
      </ul>
    );
  }
}

```

こちらがフックによる同等版です：

```

function MyComponent() {
  const [error, setError] = useState(null);
  const [isLoading, setIsLoaded] = useState(false);
  const [items, setItems] = useState([]);

  // Note: the empty deps array [] means
  // this useEffect will run once
  // similar to componentDidMount()
  useEffect(() => {
    fetch("https://api.example.com/items")
      .then(res => res.json())
      .then(
        (result) => {
          setIsLoaded(true);
          setItems(result);
        },
        // Note: it's important to handle errors here
        // instead of a catch() block so that we don't swallow
        // exceptions from actual bugs in components.
        (error) => {
          setIsLoaded(true);
          setError(error);
        }
      )
    }, [])

  if (error) {
    return <div>Error: {error.message}</div>;
  } else if (!isLoading) {
    return <div>Loading...</div>;
  } else {
    return (
      <ul>
        {items.map(item => (
          <li key={item.id}>
            {item.name} {item.price}
          </li>
        ))}
      </ul>
    );
  }
}

```

```
)});  
  </ul>  
};  
}
```

# Babel、JSX、ビルドステップ

## React では JSX を使用する必要がありますか？

いいえ！ 詳細は "JSX なしで React を使う" をご覧ください。

## React では ES6(もしくはそれ以降のバージョン)を使用する必要がありますか？

いいえ！ 詳細は "ES6 なしで React を使う" をご覧ください。

## JSX の中にコメントを記述するには？

```
<div>
  /* Comment goes here */
  Hello, {name}!
</div>
```

```
<div>
  /* It also works
   for multi-line comments. */
  Hello, {name}!
</div>
```

---

[Go to TOC](#)

# コンポーネントに関数を渡す

## コンポーネントに(clickのような)イベントハンドラを渡すには?

イベントハンドラやその他の関数を `props` として、子コンポーネントに渡してください。

```
<button onClick={this.handleClick}>
```

ハンドラ内で親コンポーネントにアクセスする必要がある場合は、関数をコンポーネントインスタンスにバインドする必要があります（以下を参照）。

## 関数をコンポーネントインスタンスにバインドするには?

使用する構文やビルドステップにより、`this.props` や `this.state` のようなコンポーネントの属性に、関数がアクセスできるようにする方法がいくつかあります。

### コンストラクタでバインドする (ES2015)

```
class Foo extends Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() {
    console.log('Click happened');
  }
  render() {
    return <button onClick={this.handleClick}>Click Me</button>;
  }
}
```

### クラスプロパティ (ES2022)

```
class Foo extends Component {
  handleClick = () => {
    console.log('Click happened');
  };
  render() {
    return <button onClick={this.handleClick}>Click Me</button>;
  }
}
```

### レンダー内でバインドする

```
class Foo extends Component {
  handleClick() {
    console.log('Click happened');
  }
  render() {
    return <button onClick={this.handleClick.bind(this)}>Click Me</button>;
  }
}
```

**補足：**

レンダー内で `Function.prototype.bind` を利用すると、コンポーネントがレンダーされるたびに新しい関数が作成され、パフォーマンスに影響を与える可能性があります（下記参照）。

**レンダー内でアロー関数を使用する**

```
class Foo extends Component {
  handleClick() {
    console.log('Click happened');
  }
  render() {
    return <button onClick={() => this.handleClick()}>Click Me</button>;
  }
}
```

**注意：**

レンダー内でアロー関数を利用するとコンポーネントがレンダーされるたびに新しい関数が作成されるため、厳密な一致による比較に基づいた最適化が動作しなくなる可能性があります。

**レンダー内でアロー関数を使用しても良いのか？**

一般的には問題ありません。コールバック関数にパラメータを渡すのに最も簡単な方法です。

しかしパフォーマンス上の問題が出た際には、ぜひ最適化しましょう！

**そもそもバインドはなぜ必要なのか？**

JavaScriptにおいて、以下の2つのコードは同等ではありません。

```
obj.method();
```

```
var method = obj.method;
method();
```

メソッドはバインドすることで、2つ目のコードが1つ目と同様に動作するようになります。

Reactでは、一般的に他のコンポーネントに渡すメソッドしかバインドする必要はありません。たとえば、`<button onClick={this.handleClick}>` というコードは、`this.handleClick` を渡しているので、バインドする必要があります。しかし、`render` メソッドやライフサイクルメソッドをバインドする必要はありません。それらは他のコンポーネントに渡すことがないからです。

[Yehuda Katzの記事](#) はバインドとは何か、またJavaScriptにおける関数の動作について詳細に説明してくれています。

**どうしてコンポーネントをレンダーするたびに、関数が呼ばれるのか？**

関数をコンポーネントに渡すときにその関数を呼び出していないことを確認してください：

```
render() {
  // Wrong: handleClick is called instead of passed as a reference!
  return <button onClick={this.handleClick()}>Click Me</button>
}
```

代わりに、(カッコなしの) 関数自体を返すようにしてください:

```
render() {
  // Correct: handleClick is passed as a reference!
  return <button onClick={this.handleClick}>Click Me</button>
}
```

## イベントハンドラやコールバックにパラメータを渡すには?

イベントハンドラをラップするアロー関数を作成してパラメータを渡すことができます:

```
<button onClick={() => this.handleClick(id)} />
```

これは `.bind` を呼び出すのと同じことです:

```
<button onClick={this.handleClick.bind(this, id)} />
```

### 例: アロー関数を利用してパラメータを渡す

```
const A = 65 // ASCII character code

class Alphabet extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      justClicked: null,
      letters: Array.from({length: 26}, (_, i) => String.fromCharCode(A + i))
    };
  }
  handleClick(letter) {
    this.setState({ justClicked: letter });
  }
  render() {
    return (
      <div>
        Just clicked: {this.state.justClicked}
        <ul>
          {this.state.letters.map(letter =>
            <li key={letter} onClick={() => this.handleClick(letter)}>
              {letter}
            </li>
          )}
        </ul>
      </div>
    );
  }
}
```

### 例: データ属性を利用してパラメータを渡す

ほかにも、DOM API をイベントハンドラに必要なデータの保存に使用することができます。大量の要素を最適化したり、React.PureComponent による等価性チェックに依存したレンダーツリーがある場合に、この方法を検討してください。

```

const A = 65 // ASCII character code

class Alphabet extends React.Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
    this.state = {
      justClicked: null,
      letters: Array.from({length: 26}, (_, i) => String.fromCharCode(A + i))
    };
  }

  handleClick(e) {
    this.setState({
      justClicked: e.target.dataset.letter
    });
  }

  render() {
    return (
      <div>
        Just clicked: {this.state.justClicked}
        <ul>
          {this.state.letters.map(letter =>
            <li key={letter} data-letter={letter} onClick={this.handleClick}>
              {letter}
            </li>
          )}
        </ul>
      </div>
    );
  }
}

```

## 関数があまりに頻繁にあるいは連続して呼ばれる場合の対処法は？

`onClick` や `onScroll` のようなイベントハンドラがあって、それが高頻度で呼び出されるのを防ぎたい場合、コールバックが実行される頻度を制御することができます。これは以下の方法で可能です：

- スロットル：時間ベースの頻度に基づくサンプルの制御（例：`_.throttle`）
- デバウンス：一定時間アクティブでなかった場合の出力の制御（例：`_.debounce`）
- `requestAnimationFrame` スロットル：`requestAnimationFrame` に基づくサンプルの制御（例：`raf-schd`）

スロットルとデバウンスの比較は[このビジュアルデモ](#)をご覧ください。

補足：

`_.debounce`、`_.throttle`、`raf-schd` は遅延されるコールバックの呼び出しをキャンセルするための `cancel` メソッドを提供します。`componentWillUnmount` でこのキャンセルメソッドを呼び出すか、あるいは遅延されるコールバック関数内でコンポーネントがまだマウントされていることを確認するようにしてください。

## スロットル

スロットルは、特定の時間枠内に関数が複数回呼ばれるのを防ぎます。以下の例では、「クリック」ハンドラをスロットルして、1秒間に2回以上呼ばれるのを防ぎます。

```
import throttle from 'lodash.throttle';

class LoadMoreButton extends React.Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
    this.handleClickThrottled = throttle(this.handleClick, 1000);
  }

  componentWillMount() {
    this.handleClickThrottled.cancel();
  }

  render() {
    return <button onClick={this.handleClickThrottled}>Load More</button>;
  }

  handleClick() {
    this.props.loadMore();
  }
}
```

## デバウンス

デバウンスは、関数が最後に呼ばれてから一定時間経過するまで実行されないようにします。これは、高頻度で発生するイベント（例：スクロールやキーボードイベントなど）に応じた高コストな計算処理が必要なときに役立ちます。下記の例は、250ミリ秒の遅延でテキスト入力をデバウンスします。

```
import debounce from 'lodash.debounce';

class Searchbox extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.emitChangeDebounced = debounce(this.emitChange, 250);
  }

  componentWillMount() {
    this.emitChangeDebounced.cancel();
  }

  render() {
    return (
      <input
        type="text"
        onChange={this.handleChange}
        placeholder="Search..."
        defaultValue={this.props.value}
      />
    );
  }

  handleChange(e) {
    this.emitChangeDebounced(e.target.value);
  }
}
```

```

    }
    emitChange(value) {
      this.props.onChange(value);
    }
}

```

### requestAnimationFrame スロットル

`requestAnimationFrame` は描画パフォーマンスが最適化されるタイミングでブラウザで実行される関数をキューに入れる方法です。`requestAnimationFrame` でキューに入れられている関数は、次のフレームで実行されます。ブラウザは、毎秒 60 フレーム (60fps) を確保するように努めます。しかし、それができなかった場合には、自然に 1 秒間のフレーム数が制限されます。たとえば、デバイスが 30 fps しか処理できなければ、1 秒間に 30 フレームしか取得できません。`requestAnimationFrame` スロットルを使うことで、1 秒間に 60 回以上になる更新を防ぐことができます。1 秒間に 100 回の更新をする場合、ユーザには確認できない追加作業をブラウザが作成することになります。

#### 注意：

このテクニックにより、フレーム内の最後に公開された値のみがキャプチャされます。この最適化がどのように動作するのかについての例は MDN で確認できます。

```

import rafSchedule from 'raf-schd';

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

    this.handleScroll = this.handleScroll.bind(this);

    // Create a new function to schedule updates.
    this.scheduleUpdate = rafSchedule(
      point => this.props.onScroll(point)
    );
  }

  handleScroll(e) {
    // When we receive a scroll event, schedule an update.
    // If we receive many updates within a frame, we'll only publish the latest
    // value.
    this.scheduleUpdate({ x: e.clientX, y: e.clientY });
  }

  componentWillUnmount() {
    // Cancel any pending updates since we're unmounting.
    this.scheduleUpdate.cancel();
  }

  render() {
    return (
      <div
        style={{ overflow: 'scroll' }}
        onScroll={this.handleScroll}
      >
        
      </div>
    );
  }
}

```

```
    );
}
```

## レート制限をテストする

レート制限コードを正しくテストするとき、早送り機能があると便利です。もし `jest` を使っているのであれば、`mock timers` を使って時間を早送りすることができます。`requestAnimationFrame` スロットルを使っているのであれば、`raf-stub` がアニメーションフレームの間隔を制御するのに役立つでしょう。

# 仮想 DOM と内部処理

## 仮想 DOM とは？

仮想 DOM (virtual DOM; VDOM) は、インメモリに保持された想像上のまたは「仮想の」 UI 表現が、ReactDOM のようなライブラリによって「実際の」 DOM と同期されるというプログラミング上の概念です。このプロセスは差分検出処理 (reconciliation) と呼ばれます。

このアプローチにより React の宣言型 API が可能になっています。あなたは UI をどのような状態にしたいのか React に伝え、React は必ず DOM をその状態と一致させます。これにより、React なしではアプリケーションを構築するためには避けて通れない属性の操作やイベントハンドリング、および手動での DOM 更新が抽象化されます。

"仮想 DOM" は特定の技術というよりむしろ 1 つのパターンなので、時たま違う意味で使われることがあります。React の世界において "仮想 DOM" という用語は通常、ユーザインターフェースを表現するオブジェクトである [React 要素](#) と結びつけて考えられます。React は一方で、コンポーネントツリーに関する追加情報を保持するため "ファイバー (fiber)" と呼ばれる内部オブジェクトも使用します。これらも React における "仮想 DOM" 実装の一部と見なすことができます。

## Shadow DOM は仮想 DOM と同じもの？

いいえ、違います。Shadow DOM は、本来 web components において変数や CSS をスコープ化するために設計されたブラウザ技術です。仮想 DOM は JavaScript のライブラリによってブラウザ API の上に実装された概念です。[さらに読む](#)

## 「React Fiber」とは？

Fiber は React 16 の新しい差分検出処理エンジンです。その主な目的は仮想 DOM の逐次レンダーを可能にすることです。[さらに読む](#)

# コンポーネントの state

## setState は何をしているのですか？

`setState()` はコンポーネントの `state` オブジェクト更新をスケジュールします。`state` が更新されると、コンポーネントはそれに再レンダーで応じます。

## state と props の違いは何ですか？

`props` ("properties" を短くしたもの) と `state` は、両方ともplainな JavaScript のオブジェクトです。どちらもレンダー結果に影響を及ぼす情報を持ってはいますが、ある重要な一点が異なっています。つまり、`props` は（関数引数のように）コンポーネントへ渡されるのに対し、`state` は（関数内で宣言された変数のように）コンポーネントの内部で制御されます。

`props` と `state` のどちらをいつ使うべきかについて、こちらでより詳しく読むことができます。

- [Props vs State](#)
- [ReactJS: Props vs. State](#)

## setState が誤った値を返すのはなぜですか？

React では、`this.props` と `this.state` のいずれも、レンダーされたもの、つまりスクリーン上の値を表しています。

`setState` 呼び出しは非同期です。呼び出し直後から `this.state` が新しい値を反映することを期待しないでください。もし現在の `state` に基づいた値を計算する必要がある場合は、オブジェクトの代わりに更新関数を渡してください（詳しくは以下を参照）。

このコード例は期待した通りには動きません。

```
incrementCount() {
  // 補足：これは意図通りに*動きません*
  this.setState({count: this.state.count + 1});
}

handleSomething() {
  // `this.state.count` は 0 から始まるとします。
  this.incrementCount();
  this.incrementCount();
  this.incrementCount();
  // React がコンポーネントを再レンダーしても、`this.state.count` は意図通りの 3 ではなく 1
  // になります。

  // これは、上記の `incrementCount()` 関数は `this.state.count` の値を読むのですが、
  // しかしコンポーネントが再レンダーされるまで React が `this.state.count` を更新しないため
  // です。
  // そして `incrementCount()` は値が 0 のままの `this.state.count` を毎回読み、そして 1
  // をセットしてしまいます。

  // 対処法は下で説明しています！
}
```

この問題を解決するには以下を見てください。

## どうやって現在の state に依存する値を更新したらいいですか？

`setState` ヘオブジェクトを渡す代わりに関数を渡してください。その関数は常に最新の状態の `state` を使って呼ばれることが保証されています（次項参照）。

### .setState ヘオブジェクトを渡すのと関数を渡すとのでは何が違いますか？

更新関数を渡すと、その関数内で現在の `state` の値へアクセスできるようになります。`setState` 呼び出しはバッチ処理されるため、更新処理を連結して、それぞれの更新が競合せずに順序だって動作することが保証されます。

```
incrementCount() {
  this.setState((state) => {
    // 重要：更新には `this.state` ではなく `state` を使います。
    return {count: state.count + 1}
  });
}

handleSomething() {
  // `this.state.count` は 0 から始まるとします。
  this.incrementCount();
  this.incrementCount();
  this.incrementCount();

  // ここで `this.state.count` を読んでもまだ 0 のままで。
  // しかし React がコンポーネントを再レンダーするとき、値は 3 になります。
}
```

[setState についてもっと学ぶ](#)

### いつ `setState` は非同期になりますか？

現在、`setState` はイベントハンドラの内側では非同期です。

例えばクリックイベントの間に `Parent` と `Child` の両方が `setState` を呼ぶとき、非同期処理のおかげで `Child` が 2 度レンダーされないことが保証されます。その代わりに React はブラウザイベントの最後に `state` の更新を「フラッシュ(flush)」します。これにより大規模アプリのパフォーマンスが大幅に向上升します。

これは実装の詳細ですので、この仕組みに直接依存しないようにしてください。将来のバージョンにおいて、React はより多くの場合にバッチ更新するようになります。

### どうして React は `this.state` を同期的に更新しないのですか？

前項で説明したように、全てのコンポーネントがそのイベントハンドラ内で `setState()` を呼ぶまで、React は再レンダー前に意図的に「待つ」ようになっています。これにより不必要的再レンダーが防がれ、パフォーマンスが向上します。

とはいっても、React がどうして再レンダーなしに `this.state` を即時更新しないのか、まだ疑問に思っているかもしれません。

これには主に 2 つの理由があります。

- 同期的更新が `props` と `state` の間の一貫性を破壊し、非常にデバッグが難しい問題を引き起こしうるため。
- 同期的更新が、我々が取り組んでいる新機能のいくつかを実装不可能にしうるため。

この [GitHub コメント](#) は特定の例について詳しく解説しています。

## Redux や MobX のような state 管理ライブラリを使うべきでしょうか？

[時には必要かもしれません。](#)

まずは他のライブラリを追加する前に React を理解することをお勧めします。React だけでも非常に複雑なアプリケーションを作り上げることができます。

---

[Go to TOC](#)

# ファイル構成

## お勧めの React プロジェクトの構成は？

React はファイルをどのようにフォルダ分けするかについての意見を持っていません。とはいえ、あなたが検討したいかもしれないエコシステム内でよく用いられる共通の方法があります。

### 機能ないしルート別にグループ化する

プロジェクトを構成する一般的な方法の 1 つは、CSS や JS やテストをまとめて、機能別ないしルート別のフォルダにグループ化するというものです。

```
common/
  Avatar.js
  Avatar.css
  APIUtils.js
  APIUtils.test.js
feed/
  index.js
  Feed.js
  Feed.css
  FeedStory.js
  FeedStory.test.js
  FeedAPI.js
profile/
  index.js
  Profile.js
  ProfileHeader.js
  ProfileHeader.css
  ProfileAPI.js
```

ここでの「機能」の定義は普遍的なものではないので、粒度の選択はあなた次第です。トップレベルのフォルダの名前が思いつかない場合は、ユーザに「この製品の主な構成部品は何か」と聞いてみて、ユーザの思考モデルを青写真として使いましょう。

### ファイルタイプ別にグループ化する

プロジェクトを構築する別の人気の方法は、例えば以下のようにして類似ファイルをグループ分けするというものです。

```
api/
  APIUtils.js
  APIUtils.test.js
  ProfileAPI.js
  UserAPI.js
components/
  Avatar.js
  Avatar.css
  Feed.js
  Feed.css
  FeedStory.js
  FeedStory.test.js
  Profile.js
  ProfileHeader.js
  ProfileHeader.css
```

人によってはこの方法をさらに推し進め、コンポーネントをアプリケーション内の役割に応じてフォルダ分けすることを好みます。例として、[Atomic Design](#) はこのような原則の下に作られたデザインの方法論です。ただこのような方法論は、従わなければならない厳格なルールとして扱うよりも、役に立つ見本として扱う方が多くの場合生産的であるということを忘れないでください。

### ネストのしすぎを避ける

深くネストされた JavaScript プロジェクトには様々な痛みを伴います。相対パスを使ったインポートが面倒になりますし、ファイルが移動したときにそれらを更新するのも大変です。よほど強い理由があって深いファルダ構造を使う場合を除き、1 つのプロジェクト内では 3 段か 4 段程度のフォルダ階層に留めることを考慮してください。もちろんこれはお勧めにすぎず、あなたのプロジェクトには当てはまらないかもしれません。

### 考えすぎない

まだプロジェクトを始めたばかりなら、ファイル構成を決めるのに [5 分以上かけない](#) ようにしましょう。上述の方法の 1 つを選ぶか、自分自身の方法を考えて、コードを書き始めましょう！おそらく実際のコードをいくらか書けば、なんにせよ考え直したくなる可能性が高いでしょう。

もしも完全に詰まった場合は、すべて 1 フォルダに入れるところから始めましょう。そのうち十分に数が増えれば、いくつかのファイルを分離したくなってくるでしょう。そのころには、どのファイルと一緒に編集している頻度が高いのか、十分わかるようになっているでしょう。一般的には、よく一緒に変更するファイルを近くに置いておくのは良いアイディアです。この原則は、「コロケーション」と呼ばれます。

プロジェクトが大きくなるにつれ、実際にはしばしば上記両方の方法が組み合わされて使用されます。ですので、「正しい」方法を最初から選択することはさほど重要ではありません。

# CSS とスタイルの使用

## CSS のクラスをコンポーネントに適用するにはどうすれば？

このようにクラス名を文字列として `className` プロパティに与えてください。

```
render() {
  return <span className="menu navigation-menu">Menu</span>
}
```

このようにコンポーネントの `props` や `state` の状態を元に CSS クラスを割り当てる方法もよく使用されています。

```
render() {
  let className = 'menu';
  if (this.props.isActive) {
    className += ' menu-active';
  }
  return <span className={className}>Menu</span>
}
```

### ヒント

もしあなたが上記のようなコードを書くことが多い場合、`classnames` という npm パッケージを使うことで簡略化できます。

## インラインスタイルは使えますか？

はい、スタイルの書き方については[この資料](#)をご覧ください。

## インラインスタイルは悪なの？

パフォーマンス観点から言えば、基本的に CSS クラスを使う方が、インラインスタイルを用いるよりも優れています。

## CSS-in-JS とは？

"CSS-in-JS" とは外部ファイルでスタイルを定義するのとは違い、JavaScript を用いて CSS を生成するパターンのことを指します。

注意：この機能は React の一部ではありません。サードパーティのライブラリ群により提供される機能です。React はスタイルがどのように定義されているかには関心を持ちません。判断に困った場合は、まずは別の `*.css` にスタイルを定義して、`className` を使って参照するところからはじめると良いでしょう。

## React でアニメーションは使えますか？

React を使ってアニメーションを動かすことは可能です。例として、[React Transition Group](#)、[React Motion](#)、[React Spring](#) や [Framer Motion](#) をご覧ください。

---

[Go to TOC](#)

# バージョニングポリシー

React は [セマンティック バージョニング \(semantic versioning; semver\)](#) の原則に従います。

すなわちバージョン番号は **x.y.z** になります。

- **バグ修正**をする時、**z** の番号を変更することで**パッチリリース**をします（例 15.6.2 から 15.6.3）。
- **新機能追加**をする時、**y** の番号を変更することで**マイナーリリース**をします（例 15.6.2 から 15.7.0）。
- **破壊的変更**をする時、**x** の番号を変更することで**メジャーリリース**をします（例 15.6.2 から 16.0.0）。

メジャーリリースには新機能を含むことができ、全てのリリースにバグ修正を含められます。

マイナーリリースは、最も一般的なリリースです。

このバージョニングポリシーは Next と Experimental チャンネルのプレリリースビルトには適用されません。 [プレリリースについての詳細](#)

## 破壊的変更

破壊的変更は誰にとっても不便なので、私たちはメジャーリリースを最小限にするようにしています。例えば React 15 は 2016 年 4 月にリリースされており、React 16 は 2017 年の 9 月にリリースされています。React 17 は 2020 年 10 月にリリースされました。

その代わり、新機能のリリースをマイナーバージョンでしています。つまりマイナーリリースは控えめな名前にも関わらず、メジャーリリースよりしばしば興味深く魅力的です。

## 安定性への取り組み

React が徐々に変化していく中で、私たちは新機能を取り入れるために必要な労力を最小限にするようにしています。別のパッケージに入れることはあっても、可能な限り古い API が動作するように保ちます。例えば [ミックスインは長年推奨されていません](#)が [create-react-class](#) を通じて今日までサポートされており、多くのレガシーコードが安定してそれらを継続使用しています。

100 万人を超える開発者が React を使用し、合わせると何百万ものコンポーネントを管理しています。Facebook のコードベースだけでも 5 万以上の React コンポーネントがあります。なので React はできるだけ簡単に新バージョンにアップグレードできるようにする必要があります。もし移行手段なしに React へ大きな変更を行えば、開発者は古いバージョンにとどまるでしょう。私たちはアップグレード方法を Facebook 自体でテストしています。10 人以下の私たちのチームが単独で 5 万以上のコンポーネントをアップデートできるなら、React を使用している全ての人にとって管理しやすいアップグレードであると見込めます。多くの場合、私たちはコンポーネントの構文をアップグレードするための[自動化スクリプト](#)を書き、オープンソースのリリースに含め誰でも使用できるようにしています。

## 警告による段階的アップグレード

React の開発ビルトは多くの有益な警告を含みます。可能な限り、私たちは将来の破壊的変更に備える警告を追加します。最新のリリースでもしあなたのアプリが警告を出さないのであれば、次期メジャーリリースとの互換性があるでしょう。これによりアプリを 1 つのコンポーネントずつアップグレードすることが可能になります。

開発時の警告はあなたのアプリの実行に影響しません。なので開発ビルトと本番ビルトでアプリの動作は同じであると確信できます。違いは、本番ビルトは警告をロギングしないこと、そして本番ビルトはより効率的に動作すること、の 2 点のみです（万一そうなっていないことに気づいた場合は、issue を作成してください）。

## 何を破壊的変更とみなすのか？

通常、私たちは下記の変更ではメジャー番号を上げません。

- **開発時の警告。**これらは本番環境の動作に影響を与えないで、新しい警告の追加や既存の警告の修正はメジャーバージョンの間で行います。これにより次期の破壊的変更を確実に警告することができるのです。
- **`unstable_` から始まる API。**これらは、API をまだ信頼することができない、実験的な機能として提供されます。`unstable_` という接頭語をつけてリリースすることで、より速く開発サイクルを進め、より早く安定した API にすることができます。
- **React のアルファバージョンとカナリア (canary) バージョン。**新機能を早くテストするために React のアルファバージョンを提供しますが、アルファで学んだことを基に柔軟に変更を加える必要があります。もしこれらのバージョンを使用する場合は、安定版のリリース前に API が変わる可能性に注意してください。
- **ドキュメント化されていない API と内部データ構造。**もし内部プロパティである `__SECRET_INTERNALS_DO_NOT_USE_OR_YOU_WILL_BE_FIRED` や `__reactInternalInstance$uk43rzhitjg` などにアクセスしたいのであれば保証はされません。自己責任で行ってください。

このポリシーはみなさんの頭痛の種とならないよう、実用的に構成されています。上記の全ての変更のためにメジャーバージョンを上げると、より多くメジャーリリースが必要になり、最終的により多くのバージョニングの問題をコミュニティに對して引き起こすことになります。それは React の改善を私たちが望むほど早くできないことも意味します。

それでも、上記のリストのような変更がコミュニティ内で広域に渡る問題を引き起こすと予想される場合は、私たちは段階的な移行手段を提供するように最善を尽くします。

## 新機能がない場合でもパッチではなくマイナーリリースになる理由は？

マイナーリリースに新機能がないということはあります。これは semver で許容されており、具体的には「(マイナーバージョンは) 非公開のコード内で大きな機能追加や改善があった場合に上げても良い (MAY)。パッチレベルの変更を含んでいても良い (MAY)」ことになっています。

それでも、そのようなリリースがパッチリリースとしてバージョニングされないのは何故かというもっともな疑問が湧いてきます。

答えは、React やその他のソフトウェアに対するあらゆる変更には、想定外にコードが壊れるリスクがあるからです。とあるバグを修正するためのパッチリリースが別のバグを引き起こすという状況を想像してください。開発者を混乱させるだけでなく、将来のパッチリリースに対する信頼を失わせることになります。元の修正が現実には滅多に発生しないバグに対するものだった場合は、特に残念なことになります。

React のリリースにバグを含めないことに関して我々はかなり良い成績を残してきていますが、ほとんどの開発者はパッチリリースは有害な副作用なしに採用できるものと当然に考えているので、パッチリリースに関しては信頼性のハードルがさらに上がります。

このような理由により、我々はパッチリリースを重大なバグやセキュリティの脆弱性の修正のみに利用することにしています。

リリースがあまり本質的でない変更、例えば内部のリファクタリング、実装の詳細の変更、パフォーマンス改善、小さなバグ修正といったものに関する場合、我々は新機能がなくともマイナーバージョンを上げます。

# フォーム

HTML のフォーム要素は当然のこととして内部に何らかの状態を持っていますので、フォーム要素は React において他の DOM 要素とちょっと異なる動作をします。例えば、このプレーン HTML によるフォームは 1 つの名前を受け付けます：

```
<form>
  <label>
    Name:
    <input type="text" name="name" />
  </label>
  <input type="submit" value="Submit" />
</form>
```

このフォームは、ユーザがフォームを送信した際に新しいページに移動する、という、HTML フォームとしてのデフォルトの動作をします。React でこの振る舞いが必要なら、そのまま動きます。しかし大抵のケースでは、フォームの送信に応答してユーザがフォームに入力したデータにアクセスするような JavaScript 関数があった方が便利です。これを実現する標準的な方法は、"制御された (controlled) コンポーネント" と呼ばれるテクニックを使うことです。

## 制御されたコンポーネント

HTML では `<input>`、`<textarea>`、そして `<select>` のようなフォーム要素は通常、自身で状態を保持しており、ユーザの入力に基づいてそれを更新します。React では、変更される状態は通常はコンポーネントの `state` プロパティに保持され、`setState()` 関数でのみ更新されます。

React の `state` を "信頼できる唯一の情報源 (single source of truth)" とすることで、上述の 2 つの状態を結合させることができます。そうすることで、フォームをレンダーしている React コンポーネントが、後続するユーザ入力でフォームで起きることも制御できるようになります。このような方法で React によって値が制御される入力フォーム要素は「制御されたコンポーネント」と呼ばれます。

例えば、前述のフォームの例において、フォーム送信時に名前をログに残すようにしたい場合、フォームを制御されたコンポーネントとして書くことができます：

```
class NameForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: ''};

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({value: event.target.value});
  }

  handleSubmit(event) {
    alert('A name was submitted: ' + this.state.value);
    event.preventDefault();
  }

  render() {
```

```

    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name:
          <input type="text" value={this.state.value} onChange={this.handleChange}>
        />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}

```

### Try it on CodePen

フォーム要素の `value` 属性が設定されているので、表示される値は常に `this.state.value` となり、React の state が信頼できる情報源となります。`handleChange` はキーストロークごとに実行されて React の state を更新するので、表示される値はユーザがタイプするたびに更新されます。

制御されたコンポーネントを使うと、ユーザ入力の値は常に React の state によって制御されるようになります。これによりタイプするコード量は少し増えますが、その値を他の UI 要素に渡したり、他のイベントハンドラからリセットしたりできるようになります。

## textarea タグ

HTML では、`<textarea>` 要素はテキストを子要素として定義します。

```

<textarea>
  Hello there, this is some text in a text area
</textarea>

```

React では、`<textarea>` は代わりに `value` 属性を使用します。こうすることで、`<textarea>` を使用するフォームは單一行の入力フォームと非常に似た書き方ができるようになります：

```

class EssayForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: 'Please write an essay about your favorite DOM element.'
    };

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({value: event.target.value});
  }

  handleSubmit(event) {
    alert('An essay was submitted: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (

```

```

        <form onSubmit={this.handleSubmit}>
          <label>
            Essay:
            <textarea value={this.state.value} onChange={this.handleChange} />
          </label>
          <input type="submit" value="Submit" />
        </form>
      );
    }
}

```

`this.state.value` がコンストラクタで初期化されているので、テキストエリアには始めからテキストが入っていることに注意してください。

## select タグ

HTML では、`<select>` はドロップダウンリストを作成します。例えばこの HTML は味についてのドロップダウンリストを作成しています：

```

<select>
  <option value="grapefruit">Grapefruit</option>
  <option value="lime">Lime</option>
  <option selected value="coconut">Coconut</option>
  <option value="mango">Mango</option>
</select>

```

`selected` 属性があるため Coconut オプションが最初に選択されていることに注意してください。この `selected` 属性の代わりに React は `value` 属性を親の `select` タグで使用します。一箇所で更新すればよいだけなので、制御されたコンポーネントを使う場合にはこちらがより便利です。例えば：

```

class FlavorForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: 'coconut'};

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({value: event.target.value});
  }

  handleSubmit(event) {
    alert('Your favorite flavor is: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Pick your favorite flavor:
          <select value={this.state.value} onChange={this.handleChange}>
            <option value="grapefruit">Grapefruit</option>
            <option value="lime">Lime</option>
            <option value="coconut">Coconut</option>
          </select>
        </label>
      </form>
    );
  }
}

```

```

        <option value="mango">Mango</option>
    </select>
</label>
<input type="submit" value="Submit" />
</form>
);
}
}

```

### Try it on CodePen

全体的に見て `<input type="text">`、`<textarea>`、そして `<select>` が非常に似た動作をするようになっています。これらはすべて、制御されたコンポーネントを実装する時に使うことができる `value` 属性を受け取ります。

#### 補足

`value` 属性に配列を渡すことで、`select` タグ内の複数のオプションを選択することができます：

```
<select multiple={true} value={['B', 'C']}>
```

```
# #file input タグ {#the-file-input-tag}
```

HTML では、`<input type="file">` によってユーザにデバイス内の 1 つ以上のファイルを選ばせて、それをサーバにアップロードしたり [File API](#) を使って JavaScript で操作したりすることができます。

```
<input type="file" />
```

この値は読み取り専用ですので、これは非制御コンポーネントになります。[このドキュメントの後の方](#)で、他の非制御コンポーネントと併せて説明しています。

## 複数の入力の処理

複数の制御された `input` 要素を処理する必要がある場合、それぞれの入力要素に `name` 属性を追加すれば、ハンドラ関数に `event.target.name` に基づいて処理を選択させることができます。

例えば：

```

class Reservation extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      isGoing: true,
      numberOfGuests: 2
    };
    this.handleInputChange = this.handleInputChange.bind(this);
  }

  handleInputChange(event) {
    const target = event.target;
    const value = target.type === 'checkbox' ? target.checked : target.value;
    const name = target.name;
  }
}

```

```

        this.setState({
          [name]: value
        });
      }

      render() {
        return (
          <form>
            <label>
              Is going:
              <input
                name="isGoing"
                type="checkbox"
                checked={this.state.isGoing}
                onChange={this.handleInputChange} />
            </label>
            <br />
            <label>
              Number of guests:
              <input
                name="numberOfGuests"
                type="number"
                value={this.state.numberOfGuests}
                onChange={this.handleInputChange} />
            </label>
          </form>
        );
      }
    }
  
```

### Try it on [CodePen](#)

渡された入力名に対応する state のキーを更新するのに用いた ES6 の [computed property name](#) 構文の使い方に注意してください：

```

        this.setState({
          [name]: value
        });
      
```

これは以下の ES5 のコードと同等です：

```

var partialState = {};
partialState[name] = value;
this.setState(partialState);
      
```

また、`setState()` は自動的に部分的な state を現在の state にマージするので、変更された部分のみで呼び出せば大丈夫です。

## 制御された入力における null 値

[制御されたコンポーネント](#) で `value` プロパティに値を指定することで、変更させたくない場合にユーザが値を変更できなくなります。もしも `value` を指定したのに入力フィールドが依然変更可能であるという場合は、`value` を誤って `undefined` もしくは `null` に設定してしまったのかもしれません。

以下のコードでこれを示しています（入力フィールドは最初はロックされていますが、短い遅延の後に編集可能になります）。

```
ReactDOM.createRoot(mountNode).render(<input value="hi" />);

setTimeout(function() {
  ReactDOM.createRoot(mountNode).render(<input value={null} />);
}, 1000);
```

## 制御されたコンポーネントの代替手段

制御されたコンポーネントは、あらゆる種類のデータの変更に対してイベントハンドラを書き、あらゆる入力状態を React コンポーネントに通してやる必要があるため、時としてうんざりすることがあります。このことは既存のコードベースを React に変換する場合や、React アプリケーションを非 React のライブラリと統合する場合に、特に問題化します。これらの状況においては、入力フォームを実装する代替手段である[非制御コンポーネント](#)を検討してみてください。

## 本格的なソリューション

入力値のバリデーション、訪問済みフィールドの追跡やフォーム送信を含む完全なソリューションをお探しの場合は、[Formik](#) が人気のある選択肢のひとつです。しかしながらこれは制御されたコンポーネントや state の管理と同じ原理で作成されていますので、これらについて学ぶことを無視しないようにしましょう。

# ref のフォワーディング

`ref` のフォワーディングはあるコンポーネントを通じてその子コンポーネントのひとつに `ref` を自動的に渡すテクニックです。これは基本的にはアプリケーション内のほとんどのコンポーネントで必要ありません。しかし、コンポーネントの種類によっては、特に再利用可能なコンポーネントライブラリにおいては、便利なものとなるかもしれません。一般的なシナリオについて以下で述べます。

## DOM コンポーネントに ref をフォワーディングする

ネイティブの `button` DOM 要素をレンダーする `FancyButton` というコンポーネントを考えてみましょう：

```
embed:forwarding-refs/fancy-button-simple.js
```

React コンポーネントは、レンダーの結果も含め、実装の詳細を隠蔽します。`FancyButton` を使用する他のコンポーネントは内側の `button` DOM 要素に対する `ref` を取得する必要はありません。これは、互いのコンポーネントの DOM 構造に過剰に依存することを防ぐので、良いことです。

そういったカプセル化は `FeedStory` や `Comment` のようなアプリケーションレベルのコンポーネントでは望ましいことはありますが、`FancyButton` や `MyTextInput` といった非常に多くのところで再利用可能な "末梢の" コンポーネントでは不便である可能性があります。このようなコンポーネントは、アプリケーションのいたるところで通常の DOM である `button` や `input` と同様に扱われる傾向にあり、フォーカス、要素の選択、アニメーションをこなすにはそれら DOM ノードにアクセスすることが避けられないかもしれません。

`ref` のフォワーディングはオプトインの機能であり、それにより、コンポーネントが `ref` を受け取って、それをさらに下層の子に渡せる（つまり、`ref` を "転送" できる）ようになります。

下の例では、`FancyButton` は渡された `ref` を取得して、それをレンダーする `button` DOM にフォワーディングするために、`React.forwardRef` を使っています。

```
embed:forwarding-refs/fancy-button-simple-ref.js
```

このように、`FancyButton` を使ったコンポーネントは下層の `button` DOM ノードの `ref` を取得することができ、必要であれば `button` DOM を直接使うかのように、DOM にアクセスすることができます。

上の例で、何が起こっているかを順々に説明します。

1. `React.createRef` を呼び、`React ref` をつくり、それを `ref` 変数に代入します。
2. `ref` を `<FancyButton ref={ref}>` に JSX の属性として指定することで渡します。
3. `React` は `ref` を、`forwardRef` 内の関数 `(props, ref) => ...` の 2 番目の引数として渡します。
4. この引数として受け取った `ref` を `<button ref={ref}>` に JSX の属性として指定することで渡します。
5. この `ref` が紐付けられると、`ref.current` は `<button>` DOM ノードのことを指すようになります。

### 補足

2 番目の引数 `ref` は `React.forwardRef` の呼び出しを使ってコンポーネントを定義したときにだけ存在します。通常の関数またはクラスコンポーネントは `ref` 引数を受け取らず、`ref` は `props` からも利用できません。

ref のフォワーディング先は DOM コンポーネントだけにとどまりません。クラスコンポーネントインスタンスに対しても ref をフォワーディングできます。

## コンポーネントライブラリのメンテナ向けの補足

コンポーネントライブラリの中で、`forwardRef` を使い始めた場合、破壊的変更として扱い、新しいメジャーバージョンをリリースすべきです。ライブラリが外から見て今までと違う挙動（例えば、どの値が ref に代入されるかや、どの型がエクスポートされるのか）をする可能性があり、古い挙動に依存しているアプリケーションや他のライブラリを壊す可能性があるからです。

`React.forwardRef` が存在する場合だけ、条件的に `React.forwardRef` を適用することも同じ理由で推奨されません：そのような実装は、React そのものを更新したとき、ライブラリがどのように振る舞うかを変えてしまい、ユーザのアプリケーションを破壊する可能性があるからです。

## 高階コンポーネントにおける ref のフォワーディング

このテクニックは高階コンポーネント（HOC としても知られています）においても特に便利です。コンポーネントの props をコンソールにログ出力する HOC を例として考えてみましょう。`embed:forwarding-refs/log-props-before.js`

"logProps" HOC はすべての `props` をラップするコンポーネントに渡すので、レンダーされる出力は同じになるでしょう。例えば、"fancy button" コンポーネントに渡されるすべての props をログとして記録するために、この HOC を使用することができます。`embed:forwarding-refs/fancy-button.js`

ところが上記の例には欠陥があります。これでは ref が渡されないのです。`ref` は `props` のひとつではないからです。`key` と同様に `ref` は React では `props` とは違う扱いになります。HOC に対する `ref` を追加した場合、ラップされたコンポーネントではなく、一番外側のコンテナコンポーネントを参照します。

これは `FancyButton` コンポーネントに紐付けられることを意図した `ref` が、実際には `LogProps` コンポーネントに紐付けられてしまうことを意味します。`embed:forwarding-refs/fancy-button-ref.js`

幸いにも、`React.forwardRef` API を使って、内側の `FancyButton` コンポーネントに対して `ref` を明示的に転送することができます。`React.forwardRef` は `render` 関数を受け取り、その関数は `props` と `ref` を引数として取り、React ノードを返します。例えば、`embed:forwarding-refs/log-props-after.js`

## DevTools でのカスタム名表示

`React.forwardRef` は `render` 関数を受け取ります。React DevTools は `ref` をフォワーディングしているコンポーネントとして何を表示すべきかを決定するために、この関数を使います。

例えば、次のコンポーネントは "ForwardRef" として DevTools に表示されます。

`embed:forwarding-refs/wrapped-component.js`

`render` 関数に名前をつけると、DevTools はその名前を含めるようになります（例： "ForwardRef(myFunction)"）：

`embed:forwarding-refs/wrapped-component-with-function-name.js`

ラップしているコンポーネントを含めるために、`render` 関数の `displayName` を設定することもできます：

```
embed:forwarding-refs/customized-display-name.js
```

---

[Go to TOC](#)

# フラグメント

React でよくあるパターンの 1 つに、コンポーネントが複数の要素を返すというものがあります。フラグメント(fragment) を使うことで、DOM に余分なノードを追加することなく子要素をまとめることができます。

```
render() {
  return (
    <React.Fragment>
      <ChildA />
      <ChildB />
      <ChildC />
    </React.Fragment>
  );
}
```

このようなものを宣言するための[短い記法](#)もあります。

## 動機

コンポーネントが子要素のリストを返すというのはよくあるパターンです。この React スニペットを例にしましょう：

```
class Table extends React.Component {
  render() {
    return (
      <table>
        <tr>
          <Columns />
        </tr>
      </table>
    );
  }
}
```

レンダーされる HTML が正しいものであるためには、`<Columns />` は複数の `<td>` 要素を返す必要があります。`<Columns />` 中の `render()` 内で親として `div` 要素を使ってしまうと、結果として出力される HTML は不正なものとなってしまいます。

```
class Columns extends React.Component {
  render() {
    return (
      <div>
        <td>Hello</td>
        <td>World</td>
      </div>
    );
  }
}
```

上記では、以下のような `<Table />` の出力となってしまいます：

```
<table>
  <tr>
    <div>
```

```
<td>Hello</td>
<td>World</td>
</div>
</tr>
</table>
```

フラグメントはこのような問題を解決します。

## 使い方

```
class Columns extends React.Component {
  render() {
    return (
      <React.Fragment>
        <td>Hello</td>
        <td>World</td>
      </React.Fragment>
    );
  }
}
```

上記は、以下のような正しい `<Table />` の出力となります：

```
<table>
  <tr>
    <td>Hello</td>
    <td>World</td>
  </tr>
</table>
```

## 短い記法

フラグメントを宣言するための新しい短縮記法があります。それは空のタグのようにも見えます：

```
class Columns extends React.Component {
  render() {
    return (
      <>
        <td>Hello</td>
        <td>World</td>
      </>
    );
  }
}
```

この `<></>` は他の要素と同じように使うことが可能ですが、key や属性のサポートはありません。

## key 付きフラグメント

明示的に `<React.Fragment>` と宣言したフラグメントでは key を持つことができます。これはコレクションをフラグメントの配列に変換するときに有用です。たとえば定義リストを作成する時に利用します：

```
function Glossary(props) {
  return (
    <dl>
      {props.items.map(item => (
```

```
// Without the `key`, React will fire a key warning
<React.Fragment key={item.id}>
  <dt>{item.term}</dt>
  <dd>{item.description}</dd>
</React.Fragment>
) );
</dl>
);
}
```

`key` はフラグメントに渡すことができる唯一の属性です。将来的には、イベントハンドラのような他の属性を渡すこともサポートするかもしれません。

## ライブデモ

この [CodePen](#) で新しい JSX フラグメントの記法を試すことができます。

# Getting Started

- "download.html"
- "downloads.html"

このページは React のドキュメント、および関連する資料の概要となります。

**React** はユーザインターフェースを作成する為の JavaScript のライブラリです。このホームページ、またはこのチュートリアルから React がどういうものかを学びましょう。

- [React を試す](#)
- [React を学ぶ](#)
- [最新の情報を追いたい人へ](#)
- [バージョン管理されたドキュメント](#)
- [足りないものがある?](#)

## React を試す

React は当初より、既存のプロジェクトに徐々に追加していくようなデザインとなっています。たくさん React を使うのでも、少しだけ使うのでも構いません。触りだけやってみるもよし、シンプルな HTML のページにインタラクティブな機能を追加するのに使うもよし、React をフル活用した複雑なアプリを作成するもよし。どのような目的にしても、このページにあるリンクが役に立つでしょう。

## Web 上で試せるオンラインエディタ

React をちょっと試してみたい場合、Web 上のコードエディタが使えます。[CodePen](#)、[CodeSandbox](#) や [Stackblitz](#) で Hello World のテンプレートを使って試してみましょう。

自前のテキストエディタを使いたい場合は、[この HTML ファイル](#)をダウンロード・編集して、ブラウザを使ってからローカルファイルシステムから開くことができます。ランタイムでの遅いコード変換が行われる為、簡単なデモに留めておくことをおすすめします。

## React を Web サイトに追加する

React は HTML ページにすぐに追加することができます。そのあと徐々に全体に反映させていくか、少数のダイナミックなウィジェットに留めるかはあなたの自由です。

## 新規 React アプリの作成

新しく React のプロジェクトを始めたい場合でも、まずは[シンプルな HTML ページに script タグを追加](#)するのがおすすめです。数分でセットアップできます！

アプリが成長するにつれて、より統合されたセットアップを行うことを考慮していきましょう。大きいアプリの場合におすすめの様々な JavaScript を用いたツールチェインが存在します。それぞれわずかな設定、もしくは設定要らずでリッチな React のエコシステムをフル活用できます。[詳細はこちら。](#)

# React を学ぶ

React を学びたい人には様々な背景があり、それぞれいろんな学び方のスタイルがあるでしょう。理論から学びたい人も、実際に手を動かしながら学びたい人も、このセクションが役に立てれば幸いです。

- 実際に手を動かしながら学びたい人は[チュートリアル](#)からはじめましょう。
- コンセプトからひとつひとつ学んでいきたい人は[こちらのガイド](#)からはじめていきましょう。

他の新しい技術を新しく学ぶ時と同様、React にも学習コストがあります。しかし、辛抱強く、地道にコツコツやっていくことで、絶対にできるようになります。

## 最初の例

[React のホームページ](#)にはライブエディタにて記入できる小さな React のサンプルがいくつか載っています。まだ React のことを何も知らなくても、実際に触ってみてどのように変化するか見てみましょう。

## React 初心者向けのガイド

もし、React の公式ドキュメントのペースが早すぎると感じた場合は、[Tania Rascia 氏によって書かれた React の概要](#)を読んでみましょう。React の重要なコンセプトについて詳細に説明されており、初心者にもとても易しい作りとなっています。それが終わったら、もう一度公式ドキュメントを読んでみましょう！

## デザイナ向けのガイド

もしあなたがデザイナのバックグラウンドを持っているなら、[これらの資料](#)から始めることをおすすめします。

## JavaScript 資料

React の公式ドキュメントはあなたがある程度 JavaScript について知っていることを想定しています。JavaScript の達人である必要は全くありませんが、JavaScript と React を同時に習得するのは難しいでしょう。

JavaScript について自分がどれだけ知っているかを知るために、[この JavaScript の概要](#)を読んでおくことをおすすめします。読むのに 30 分から 1 時間程かかるかもしれませんが、React を学ぶための自信へと繋がってくるでしょう。

### ヒント

もし JavaScript で何か詰まった時は、[MDN](#) と [javascript.info](#) を確認すると良いでしょう。また、[コミュニティ運営の掲示板](#)を活用するのもおすすめします。

## 実践チュートリアル

実際に手を動かしながら学びたい人は[チュートリアル](#)からはじめましょう。このチュートリアルでは React を使って三目並べゲームを作成します。ゲーム作成に興味が無い方は飛ばそうと思うかもしれません、試してみてください。ここで学ぶテクニックは全ての React アプリの基礎となることであり、習得することで React に関する深い理解が得られます。

## 一步づつ進めるためのガイド

コンセプトからひとつひとつ学んでいきたい人は[こちらのガイド](#)からはじめていきましょう。このガイドは章ごとに分かれていて、各章がそれまでに習ったことを上乗せする知識となっているため、余すことなく学ぶことができます。

## React 的な考え方

多くの React のユーザは [React の流儀](#)を読んだ時に React についてピンときたと高く評価しています。この世に存在する一番古い React に関する資料ですが、今でも現役で使えます。

## おすすめの資料

時には第三者が執筆した技術書やオンラインコース等が公式ドキュメントよりもわかりやすいという人もいます。我々は[よく勧められている資料集](#)についてもまとめています。無料のものもあります。

## 高度なコンセプト

React の[基本コンセプト](#)について慣れてきて、実際に手で動かしてみたら、今度はさらに高度な内容にも手を出したいと思うかもしれません。この資料では強力な、しかし普段使われることのない [コンテクスト](#) や [ref](#) といった React のさらなる機能について紹介します。

## API リファレンス

この資料は特定の React API の詳細について学びたいときに役に立つでしょう。例えば、[React.Component API リファレンス](#)では [setState\(\)](#) がどう機能しているかについてより詳細に記載されており、種々のライフサイクルメソッドがどのように役に立つかについて書かれています。

## 用語集と FAQ

この[用語集](#)にはこのドキュメントにてよく使われる単語等が記載されています。FAQ もあり、[AJAX リクエストについて](#)、[コンポーネントの state](#)、[ファイル構成](#)などのよくある質問とそれらに対する回答が記載されています。

## 最新の情報を追いたい人へ

[React ブログ](#)は React チームからのアップデートが記載されている公式ブログです。リリースノートや機能の非推奨化の告知など、何か大事な発表がある時にはこのブログに真っ先に記載されます。

他にも [@reactjs](#) のツイッターアカウントをフォローすることができますが、公式ブログを追っていれば大事なことを見落とすことはないでしょう。

全ての React のリリースにブログ記事があるわけではありませんが、リリースごとに詳細に書かれた [changelog](#) が [React リポジトリ](#) の [CHANGELOG.md](#)、および [Releases ページ](#)に記載されています。

## バージョン管理されたドキュメント

このドキュメントは常に最新の stable 版の React に準拠しています。React 16 からは古いバージョンのドキュメントも[別のページ](#)から閲覧できます。古いバージョンのドキュメントは各バージョンリリース時のスナップショットであり、更新をかけることはありません。

## 他に聞きたいことは？

ドキュメントに記載されていないことがあったり、わかりにくい箇所があったりしたら、困ったことや改善案等を[この公式ドキュメントのリポジトリ](#)に issue として立てるか、もしくは公式の [@reactjs](#) までご連絡ください。あなたの意見をお待ちしております！

---

[Go to TOC](#)

# イベント処理

React でのイベント処理は DOM 要素のイベントの処理と非常に似ています。いくつかの文法的な違いがあります：

- React のイベントは小文字ではなく camelCase で名付けられています。
- JSX ではイベントハンドラとして文字列ではなく関数を渡します。

例えば、以下の HTML :

```
<button onclick="activateLasers()">
  Activate Lasers
</button>
```

は、React では少し異なります：

```
<button onClick={activateLasers}>
  Activate Lasers
</button>
```

別の違いとして、React では `false` を返してもデフォルトの動作を抑止することができません。明示的に `preventDefault` を呼び出す必要があります。例えば、plainな HTML では、「フォームをサブミットする」という form 要素のデフォルト動作を抑止するために次のように書くことができます。

```
<form onsubmit="console.log('You clicked submit.'); return false">
  <button type="submit">Submit</button>
</form>
```

React では、代わりに次のようにになります：

```
function Form() {
  function handleSubmit(e) {
    e.preventDefault();
    console.log('You clicked submit.');
  }

  return (
    <form onSubmit={handleSubmit}>
      <button type="submit">Submit</button>
    </form>
  );
}
```

ここで、`e` は合成 (synthetic) イベントです。React はこれらの合成イベントを [W3C の仕様](#) に則って定義しているので、ブラウザ間の互換性を心配する必要はありません。React のイベントはネイティブのイベントと全く同様に動作するわけではありません。詳細については、[SyntheticEvent](#) のリファレンスガイドを参照してください。

React を使う場合、一般的には DOM 要素の生成後に `addEventListener` を呼び出してリスナを追加する必要はありません。代わりに、要素が最初にレンダーされる際にリスナを指定するようにしてください。

コンポーネントを [ES6 のクラス](#)を使用して定義した場合、一般的なパターンではイベントハンドラはクラスのメソッドになります。例えば、以下の `Toggle` コンポーネントはユーザが "ON" 状態 "OFF" 状態を切り替えられるようなボタンをレンダーします。

```
class Toggle extends React.Component {
  constructor(props) {
    super(props);
    this.state = {isToggleOn: true};

    // This binding is necessary to make `this` work in the callback
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState(prevState => ({
      isToggleOn: !prevState.isToggleOn
    }));
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        {this.state.isToggleOn ? 'ON' : 'OFF'}
      </button>
    );
  }
}
```

### Try it on [CodePen](#)

JSX のコールバックにおける `this` の意味に注意しなければなりません。JavaScript では、クラスのメソッドはデフォルトでは [バインド](#)されません。`this.handleClick` へのバインドを忘れて `onClick` に渡した場合、実際に関数が呼ばれた時に `this` は `undefined` となってしまいます。

これは React に限った動作ではなく、[JavaScript における関数の仕組み](#)の一部です。一般的に、`onClick={this.handleClick}` のように `○` を末尾に付けずに何らかのメソッドを参照する場合、そのメソッドはバインドしておく必要があります。

`bind` の呼び出しが苦痛なら、それを回避する方法が 2 つあります。[パブリッククラスフィールド構文](#)を使えば、コールバックを正しくバインドすることができます：

```
class LoggingButton extends React.Component {
  // This syntax ensures `this` is bound within handleClick.
  handleClick = () => {
    console.log('this is:', this);
  };

  render() {
    return (
      <button onClick={this.handleClick}>
        Click me
      </button>
    );
  }
}
```

この構文は、[Create React App](#) ではデフォルトで有効です。

クラスフィールド構文を使用しない場合、コールバック内でアロー関数を使用することもできます：

```
class LoggingButton extends React.Component {
  handleClick() {
    console.log('this is:', this);
  }

  render() {
    // This syntax ensures `this` is bound within handleClick
    return (
      <button onClick={() => this.handleClick()}>
        Click me
      </button>
    );
  }
}
```

この構文での問題は、`LoggingButton` がレンダーされるたびに異なるコールバック関数が毎回作成されるということです。大抵のケースではこれは問題ありません。しかし、このコールバックが `props` の一部として下層のコンポーネントに渡される場合、それら下層コンポーネントが余分に再描画されることになります。一般的にはコンストラクタでバインドするかクラスフィールド構文を使用して、この種のパフォーマンスの問題を避けるようおすすめします。

## イベントハンドラに引数を渡す

ループ内では、イベントハンドラに追加のパラメータを渡したくなることがあります。例えば、`id` という行の ID がある場合、以下のどちらでも動作します：

```
<button onClick={(e) => this.deleteRow(id, e)}>Delete Row</button>
<button onClick={this.deleteRow.bind(this, id)}>Delete Row</button>
```

上記の 2 行は等価であり、上側ではアロー関数が、下側では `Function.prototype.bind` が使われています。

どちらの場合でも、React イベントを表す `e` という引数は ID の次の 2 番目の引数として渡されることになります。アロー関数では `e` を明示的に渡す必要がありますが、`bind` の場合には `id` 以降の追加の引数は自動的に転送されます。

---

[Go to TOC](#)

# Hello World

React のいちばん短い例はこのようになります：

```
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<h1>Hello, world!</h1>);
```

これは "Hello, world!" という見出しをページに表示します。

## Try it on CodePen

上記のリンクをクリックしてオンラインエディタを開いてください。好きなように書き換えて、出力にどう影響するのかを確認してください。このガイドのほとんどのページにはこのような編集可能な例が出てきます。

```
## このガイドの読み方 {#how-to-read-this-guide}
```

このガイドでは、React アプリケーションの構成部品である React 要素やコンポーネントの使い方を見ていきます。一度それらをマスターすると、小さくて再利用可能な部品から複雑なアプリケーションを作成できるようになります。

### ヒント

このガイドはコンセプトを一段階ずつ学んでいきたい人向けに構成されています。手を動かして学びたい方は[実践的なチュートリアル](#)を参照してください。このガイドとチュートリアルは互いに相補的なものです。

このページは、React のコンセプトをステップバイステップで学ぶためのガイドのうち最初の章です。全ての章のリストはナビゲーション用のサイドバーにあります。モバイルデバイスで読んでいる場合は、画面の右下にあるボタンを押すことでナビゲーションにアクセスできます。

それぞれの章は、前の章までに学んだ知識を前提として構成されています。["Main Concepts" にある章をサイドバーに並んで](#)いる順番に読んでいくことで、React のほとんどを学ぶことができます。例えば["JSX の導入"](#)がこの章の次の章です。

## 前提となる知識

React は JavaScript ライブラリなので、JavaScript 言語の基本的な理解があることを想定しています。[あまり自信がない場合は、JavaScript のチュートリアル](#)を一通り読んで知識レベルを確認し、このガイドを迷わず読み進められるようにしてください。そうすれば 30 分から 1 時間ほどかかるかもしれません、JavaScript と React を同時に学んでいるような気分にならずにすむでしょう。

### 補足

このガイドでは時折比較的新しい JavaScript の構文を例の中で使用しています。ここ数年 JavaScript を使った仕事をしていなかったという場合は、[この 3 点](#)を理解すればだいたい理解したことになるでしょう。

## 始めましょう！

下にスクロールすればウェブサイトのフッターのすぐ手前に[このガイドの次の章へのリンク](#)が出てきます。

---

[Go to TOC](#)

# 高階 (Higher-Order) コンポーネント

高階コンポーネント (higher-order component; HOC) はコンポーネントのロジックを再利用するための React における応用テクニックです。HOC それ自体は React の API の一部ではありません。HOC は、React のコンポジションの性質から生まれる設計パターンです。

具体的には、高階コンポーネントとは、あるコンポーネントを受け取って新規のコンポーネントを返すような関数です。

```
const EnhancedComponent = higherOrderComponent(WrappedComponent);
```

コンポーネントが props を UI に変換するのに対して、高階コンポーネントはコンポーネントを別のコンポーネントに変換します。

HOC は Redux における `connect` や Relay における `createFragmentContainer` のように、サードパーティ製の React ライブリでは一般的なものです。

このドキュメントでは、なぜ高階コンポーネントが便利で、自分でどのように記述するのかを説明します。

## 横断的関心事に HOC を適用する

### 補足

以前に横断的関心事を処理する方法としてミックスインをお勧めしました。私たちはその後にミックスインはそれが持つ価値以上の問題を引き起こすことに気づきました。ミックスインから離れる理由と、既存のコンポーネントを移行する方法については[こちらの詳細な記事を読んでください](#)。

コンポーネントは React のコード再利用における基本単位です。しかし、いくつかのパターンの中には、これまでのコンポーネントが素直に当てはまらないことがあることに気づいたかもしれません。

例えば、コメントのリストを描画するのに外部のデータソースの購読を行う `CommentList` コンポーネントがあるとしましょう：

```
class CommentList extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {
      // "DataSource" is some global data source
      comments: DataSource.getComments()
    };
  }

  componentDidMount() {
    // Subscribe to changes
    DataSource.addChangeListener(this.handleChange);
  }

  componentWillMount() {
```

```
// Clean up listener
DataSource.removeChangeListener(this.handleChange);
}

handleChange() {
// Update component state whenever the data source changes
this.setState({
  comments: DataSource.getComments()
});
}

render() {
  return (
    <div>
      {this.state.comments.map((comment) => (
        <Comment comment={comment} key={comment.id} />
      ))}
    </div>
  );
}
}
```

後になって、前述のパターンと似たような形で、1件のブログ記事に関する情報を購読するコンポーネントを書くとします：

```
class BlogPost extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {
      blogPost: DataSource.getBlogPost(props.id)
    };
  }

  componentDidMount() {
    DataSource.addChangeListener(this.handleChange);
  }

  componentWillUnmount() {
    DataSource.removeChangeListener(this.handleChange);
  }

  handleChange() {
    this.setState({
      blogPost: DataSource.getBlogPost(this.props.id)
    });
  }

  render() {
    return <TextBlock text={this.state.blogPost} />;
  }
}
```

`CommentList` と `BlogPost` は同一ではありません。`DataSource` に対して異なるメソッドを呼び出し、異なる出力を描画します。しかし、それらの実装の大部分は同じです：

- コンポーネントのマウント時に、`DataSource` にイベントリスナを登録する。
- リスナの内部で、`setState` をデータソースが変更されるたびに呼び出す。
- コンポーネントのアンマウント時には、イベントリスナを削除する。

大規模なアプリケーションにおいては、`DataSource` を購読して `setState` を呼び出すという同様のパターンが何度も発生することが想像できるでしょう。1つの場所にロジックを定義し、多数のコンポーネントを横断してロジックを共有可能にするような抽象化が欲しいところです。このような場合には高階コンポーネントが有効です。

コンポーネントを作成するような関数を書いて、`DataSource` からデータを受け取る、`CommentList` や `BlogPost` のようなコンポーネントを作り出せます。その関数は引数の1つとして子コンポーネントを受け取り、その子コンポーネントは購読したデータを `props` の一部として受け取ります。この関数を `withSubscription` と呼ぶことにしましょう。

```
const CommentListWithSubscription = withSubscription(
  CommentList,
  (DataSource) => DataSource.getComments()
);

const BlogPostWithSubscription = withSubscription(
  BlogPost,
  (DataSource, props) => DataSource.getBlogPost(props.id)
);
```

1つの引数はラップされるコンポーネントです。2つの引数は、与えられた `DataSource` と現在の `props` をもとに、関心のあるデータを取り出します。

`CommentListWithSubscription` と `BlogPostWithSubscription` が描画されると、`CommentList` と `BlogPost` は `DataSource` から取得した最新データを `data` プロパティとして受け取ります：

```
// This function takes a component...
function withSubscription(WrappedComponent, selectData) {
  // ...and returns another component...
  return class extends React.Component {
    constructor(props) {
      super(props);
      this.handleChange = this.handleChange.bind(this);
      this.state = {
        data: selectData(dataSource, props)
      };
    }

    componentDidMount() {
      // ... that takes care of the subscription...
      dataSource.addChangeListener(this.handleChange);
    }

    componentWillUnmount() {
      dataSource.removeChangeListener(this.handleChange);
    }

    handleChange() {
      this.setState({
        data: selectData(dataSource, this.props)
      });
    }

    render() {
      // ... and renders the wrapped component with the fresh data!
      // Notice that we pass through any additional props
      return <WrappedComponent data={this.state.data} {...this.props} />;
    }
  };
}
```

HOC は入力のコンポーネントを改変したり、振る舞いをコピーするのに継承を利用したりしません。むしろ HOC は元のコンポーネントをコンテナコンポーネント内にラップすることで組み合わせるので、HOC は副作用のない純関数です。

それだけです！ ラップされたコンポーネントはコンテナの `props` のすべてに加えて新規のプロパティである `data` を受け取り、出力の描画に使用します。外側にある HOC は渡すデータが使われる方法や理由には関心がありませんし、ラップされたコンポーネントの側はデータがどこからやって来たのかには関心を持ちません。

`withSubscription` は通常の関数なので、引数を好きなだけ増やしたり減らしたりできます。例えば、`data` プロパティの名前を変更可能にして、HOC をラップされるコンポーネントから更に分離させることもできるでしょう。もしくは `shouldComponentUpdate` を設定する引数を受け取ったり、データソースを設定する引数を受け取りたいこともあるかもしれません。HOC ではコンポーネントがどのように定義されるかを完全に管理できるため、上述のことは全て実現できます。

コンポーネントのように、`withSubscription` とラップされるコンポーネントの間の契約は完全に `props` に基づいています。これにより同じ `props` をラップされるコンポーネントに与える限りは、ある HOC を他の HOC と簡単に交換できます。このことは例えばデータ取得ライブラリを変更する場合に便利でしょう。

## 元のコンポーネントを変更するのではなく、コンポジションを使うこと

HOC の中でコンポーネントのプロトタイプを変更したり、あるいは何にせよコンポーネントに変更を加えたりしたくなる誘惑に負けてはいけません。

```
function logProps(InputComponent) {
  InputComponent.prototype.componentDidUpdate = function(prevProps) {
    console.log('Current props: ', this.props);
    console.log('Previous props: ', prevProps);
  };
  // The fact that we're returning the original input is a hint that it has
  // been mutated.
  return InputComponent;
}

// EnhancedComponent will log whenever props are received
const EnhancedComponent = logProps(InputComponent);
```

このコードにはいくつかの問題があります。1つは入力のコンポーネントを改変されたコンポーネントとは別に再利用できなくなってしまうことです。さらに悪いことに、もしこの `EnhancedComponent` に別の HOC を適用し、それが同様に `componentDidUpdate` に変更を加えるものであった場合、最初の HOC が加えた機能は上書きされてしまいます！ またこの HOC はライフサイクルメソッドを持たない関数コンポーネントには機能しません。

コンポーネントの改変を行うような HOC は不完全な抽象化です。つまり、利用する側は他の HOC との競合を避けるため、どのように実装されているかを知っておく必要があるのです。

改変を行う代わりに、HOC はコンテナコンポーネントで入力されたコンポーネントをラップすることによるコンポジションを使用すべきです：

```
function logProps(WrappedComponent) {
  return class extends React.Component {
    componentDidUpdate(prevProps) {
      console.log('Current props: ', this.props);
      console.log('Previous props: ', prevProps);
    }
  }
}
```

```

    render() {
      // Wraps the input component in a container, without mutating it. Good!
      return <WrappedComponent {...this.props} />;
    }
  }
}

```

この HOC は改変を行うバージョンと同等の機能を持ちつつ、衝突の可能性を回避しています。クラス型と関数コンポーネントのどちらでも同様にうまく動作します。そして純関数なので、自分自身を含めた他の HOC と組み合わせることができます。

おそらく HOC とコンテナコンポーネントと呼ばれるパターンの類似性に気づいたでしょう。コンテナコンポーネントは高レベルと低レベルの関心事の責任を分離する戦略の一部です。コンテナはデータ購読や state を管理してコンポーネントに props を渡し、渡された側のコンポーネントは UI の描画などの事柄を取り扱います。HOC はコンテナをその実装の一部として使用します。HOC をパラメータ化されたコンテナコンポーネントの定義であると考えることができます。

## 規則: 自身に関係のない props はラップされるコンポーネントにそのまま渡すこと

HOC はコンポーネントに機能を追加するものです。その props にもとづく契約は大きく変更すべきではありません。HOC の返り値のコンポーネントはラップされたコンポーネントと似たようなインターフェースを持つことが期待されます。

HOC はその特定の関心とは関係のない props はラップされる関数に渡すべきです。大抵の HOC はこのような描画メソッドを持ちます：

```

render() {
  // Filter out extra props that are specific to this HOC and shouldn't be
  // passed through
  const { extraProp, ...passThroughProps } = this.props;

  // Inject props into the wrapped component. These are usually state values or
  // instance methods.
  const injectedProp = someStateOrInstanceMethod;

  // Pass props to wrapped component
  return (
    <WrappedComponent
      injectedProp={injectedProp}
      {...passThroughProps}
    />
  );
}

```

この決まり事により、HOC が可能な限り柔軟で再利用しやすいものになります。

## 規則: 組み立てやすさを最大限保つこと

すべての HOC が同じ見た目になるわけではありません。引数としてラップされるコンポーネント 1 つだけを受け取ることがあります。

```
const NavbarWithRouter = withRouter(Navbar);
```

通常、HOC は追加の引数を受け取ります。この Relay からの例では、config オブジェクトがコンポーネントのデータ依存を指定するために使われています：

```
const CommentWithRelay = Relay.createContainer(Comment, config);
```

もっとも一般的な HOC の型シグネチャはこのようなものです：

```
// React Redux's `connect`
const ConnectedComment = connect(commentSelector, commentActions)(CommentList);
```

これは何なのでしょう?! バラバラにしてみると、何が起こっているのかを理解しやすくなります。

```
// connect is a function that returns another function
const enhance = connect(commentListSelector, commentListActions);
// The returned function is a HOC, which returns a component that is connected
// to the Redux store
const ConnectedComment = enhance(CommentList);
```

言い換えば、`connect` は高階コンポーネントを返す高階関数なのです！

この形式は分かりにくかったり不要なものに思えるかもしれません、便利な性質を持っています。`connect` 関数によって返されるもののような単一引数の HOC は、`Component => Component` という型シグネチャを持ちます。入力の型と出力の型が同じ関数は一緒に組み合わせるのが大変簡単なのです。

```
// Instead of doing this...
const EnhancedComponent = withRouter(connect(commentSelector)(WrappedComponent))

// ... you can use a function composition utility
// compose(f, g, h) is the same as (...args) => f(g(h(...args)))
const enhance = compose(
  // These are both single-argument HOCs
  withRouter,
  connect(commentSelector)
)
const EnhancedComponent = enhance(WrappedComponent)
```

(この性質を使えば、`connect` や他の機能追加方式の HOC をデコレータ（提唱中の JavaScript の実験的機能）で使用することも可能になります)

`compose` ユーティリティ関数は `lodash` (`lodash.flowRight` として)、`Redux`、そして `Ramda` といった多くのサードパーティ製ライブラリから提供されています。

## 規則：デバッグしやすくするため表示名をラップすること

HOC により作成されたコンテナコンポーネントは他のあらゆるコンポーネントと同様、`React Developer Tools` に表示されます。デバッグを容易にするため、HOC の結果だと分かるよう表示名を選んでください。

最も一般的な手法は、ラップされるコンポーネントの表示名をラップすることです。つまり高階コンポーネントが `withSubscription` と名付けられ、ラップされるコンポーネントの表示名が `CommentList` である場合、`WithSubscription(CommentList)` という表示名を使用しましょう：

```

function withSubscription(WrappedComponent) {
  class WithSubscription extends React.Component {/* ... */}
  WithSubscription.displayName =
    `WithSubscription(${getDisplayName(WrappedComponent)})`;
  return WithSubscription;
}

function getDisplayName(WrappedComponent) {
  return WrappedComponent.displayName || WrappedComponent.name || 'Component';
}

```

## 注意事項

高階コンポーネントには、あなたが React を始めて間もないならすぐには分からぬよう、いくつかの注意事項があります。

### render メソッド内部で HOC を使用しないこと

React の差分アルゴリズム (Reconciliation と呼ばれる) は、既存のサブツリーを更新すべきかそれを破棄して新しいものをマウントすべきかを決定する際に、コンポーネントの型が同一かどうかの情報を利用します。`render` メソッドから返されるコンポーネントが以前の描画から返されたコンポーネントと (`==` で検証して) 同一だった場合、React はサブツリーを新しいツリーとの差分を取りながら再帰的に更新します。コンポーネントが同一でなければ、以前のサブツリーは完全にアンマウントされます。

通常このことを考慮する必要はありません。ですが HOC に関しては考えるべきことです。このことが、`render` メソッド中でコンポーネントに HOC を適用してはいけないということを意味しているからです：

```

render() {
  // A new version of EnhancedComponent is created on every render
  // EnhancedComponent1 !== EnhancedComponent2
  const EnhancedComponent = enhance(MyComponent);
  // That causes the entire subtree to unmount/remount each time!
  return <EnhancedComponent />;
}

```

ここでの問題はパフォーマンスだけではありません。コンポーネントの再マウントによりコンポーネントとその子要素全ての state が失われるのです。

こうするのではなく、結果としてのコンポーネントが 1 回だけつくられるようにするために、コンポーネント定義の外で HOC を適用してください。そうすれば、レンダー間でその同一性が保たれるようになるでしょう。何にせよ、通常の場合これが望ましい実装になります。

HOC を動的に適用する必要があるような稀なケースでも、コンポーネントのライフサイクルメソッドやコンストラクタの中で行うようにしましょう。

### 静的メソッドは必ずコピーすること

React のコンポーネントで静的メソッドを定義することは便利であることがあります。例えば、Relay のコンテナは GraphQL fragment のコンポジションを容易に実現するため、`getFragment` という静的メソッドを公開しています。

しかし、HOC をコンポーネントに適用すると、元のコンポーネントはコンテナコンポーネントにラップされます。つまり新しいコンポーネントは元のコンポーネントの静的メソッドを 1 つも持っていないということになってしまいます。

```
// Define a static method
WrappedComponent.staticMethod = function() {/*...*/}
// Now apply a HOC
const EnhancedComponent = enhance(WrappedComponent);

// The enhanced component has no static method
typeof EnhancedComponent.staticMethod === 'undefined' // true
```

この問題を解決するために、コンテナコンポーネントを返す前にメソッドをコピーすることができます。

```
function enhance(WrappedComponent) {
  class Enhance extends React.Component {/*...*/}
  // Must know exactly which method(s) to copy :(
  Enhance.staticMethod = WrappedComponent.staticMethod;
  return Enhance;
}
```

しかし、この方法ではどのメソッドがコピーされる必要があるのか正確に知っておく必要があります。[hoist-non-react-statics](#) を使用することで、全ての非 React の静的メソッドを自動的にコピーできます：

```
import hoistNonReactStatic from 'hoist-non-react-statics';
function enhance(WrappedComponent) {
  class Enhance extends React.Component {/*...*/}
  hoistNonReactStatic(Enhance, WrappedComponent);
  return Enhance;
}
```

もう 1 つの解決策となりうる方法はコンポーネント自身とは分離して静的メソッドをエクスポートすることです。

```
// Instead of...
MyComponent.someFunction = someFunction;
export default MyComponent;

// ...export the method separately...
export { someFunction };

// ...and in the consuming module, import both
import MyComponent, { someFunction } from './MyComponent.js';
```

## ref 属性は渡されない

高階コンポーネントの通例としては、すべての props はラップされたコンポーネントに渡されますが、ref に関してはそうではありません。これは `ref` 属性が（`key` と同様）実際のプロパティではなく、React によって特別に処理されているものだからです。HOC から出力されたコンポーネントの要素に `ref` 属性を追加する場合、`ref` 属性はラップされた内側のコンポーネントではなく、最も外側のコンテナコンポーネントを参照します。

この問題の解決方法は（React 16.3 で導入された）`React.forwardRef` API を使うことです。[詳しくは ref のフォワーディングの章をご覧ください。](#)

# 独自フックの作成

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。

自分独自のフックを作成することで、コンポーネントからロジックを抽出して再利用可能な関数を作ることが可能です。

以下のコンポーネントは[副作用フックの使い方](#)について学んだ際に見たチャットアプリのコンポーネントであり、フレンドがオンラインかオフラインかを示すメッセージを表示します。

```
import React, { useState, useEffect } from 'react';

function FriendStatus(props) {
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}
```

さて、このチャットアプリには連絡先リストもあって、そこではオンラインのユーザを緑色で表示したいとしましょう。新しい `FriendListItem` コンポーネントに似たようなロジックをコピーペーストしても構いませんが、それは理想的ではないでしょう：

```
import React, { useState, useEffect } from 'react';

function FriendListItem(props) {
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  return (
    <li style={{ color: isOnline ? 'green' : 'black' }}>
      {props.friend.name}
    </li>
  );
}
```

```

        </li>
    );
}

```

代わりに、このロジックを `FriendStatus` と `FriendListItem` 間で共有したいと思います。

これまで React には、ステートを有するロジックをコンポーネント間で共有するための人気の手法が 2 つありました。[レンダープロップ](#)と[高階コンポーネント](#)です。ツリーに新しいコンポーネントを加える必要なしに、フックが同じ問題をどのように解決するのかを見ていきましょう。

## カスタムフックの抽出

2 つの JavaScript の関数間でロジックを共有したい場合、それを別の関数に抽出します。コンポーネントもフックも関数ですので、同じ方法が使えます！

**カスタムフックとは、名前が "use" で始まり、ほかのフックを呼び出せる JavaScript の関数のことです。**例えば、以下の `useFriendStatus` が我々の最初のカスタムフックの例です：

```

import { useState, useEffect } from 'react';

function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    ChatAPI.subscribeToFriendStatus(friendID, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(friendID, handleStatusChange);
    };
  });

  return isOnline;
}

```

新しいことは何もありません。ロジックは上記のコンポーネントからコピーしてきただけです。コンポーネントのときと同様に、他のフックを呼ぶときはカスタムフックのトップレベルで無条件に呼び出していることを確認してください。

React のコンポーネントと違い、カスタムフックは特定のシグネチャを持つ必要はありません。何を引数として受け取り、そして（必要なら）何を返すのか、といったことは自分で決めることができます。別の言い方をすると、普通の関数と同じだということです。一目で [フックのルール](#)が適用されるものだと分かるようにするために、名前は `use` で始めるべきです。

この `useFriendStatus` の目的はフレンドのステータスを購読するというものです。ですので `friendID` を引数として持ち、そのフレンドがオンラインかどうかを返します。

```

function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);
  // ...
}

```

```
    return isOnline;
}
```

ではこのカスタムフックの使い方を見ていきましょう。

## カスタムフックを使う

そもそも我々の目的は `FriendStatus` と `FriendListItem` コンポーネントでの重複したロジックを取り除くことでした。どちらのコンポーネントもフレンドがオンラインかどうかを知りたいのです。

既にロジックを `useFriendStatus` フックへと抽出したので、それをただ単に使えばいいのです：

```
function FriendStatus(props) {
  const isOnline = useFriendStatus(props.friend.id);

  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}

function FriendListItem(props) {
  const isOnline = useFriendStatus(props.friend.id);

  return (
    <li style={{ color: isOnline ? 'green' : 'black' }}>
      {props.friend.name}
    </li>
  );
}
```

**このコードは元のコードと同等？** はい、全く同じように動作します。注意深く見れば、ふるまいに何の変更も加えていないということが分かります。やったのは、共通のコードを別の関数に抽出したということだけです。カスタムフックは `React` の機能というよりは、フックの設計から自然と導かれる慣習のようなものです。

**カスタムフックは "use" という名前で始めるべき？** ゼひそうしてください。この規約はとても重要です。この規約がなければ、ある関数が内部でフックを呼んでいるかどうかを知る方法がなくなり、[フックのルール](#)の違反を自動でチェックすることができなくなります。

**同じフックを使うコンポーネントは state を共有する？** いいえ。カスタムフックは `state` を使うロジック（データの購読を登録したり現在の値を覚えておいたり）を共有するためのものですが、カスタムフックを使う場所ごとで、内部の `state` や副作用は完全に分離しています。

**どのようにしてカスタムフックは独立したステートを得るのか？** それぞれのフックの呼び出しが独立した `state` を得ます。`useFriendStatus` を直接呼びだしていますので、React から見れば我々のコンポーネントが `useState` や `useEffect` を呼んだ場合と変わりません。すでに[ここ](#)や[ここ](#)で学んだ通り、`useState` や `useEffect` はひとつのコンポーネント内で複数回呼ぶことができ、それらは完全に独立しています。

### ヒント: フック間で情報を受け渡す

フックは関数ですので、フック間で情報を受け渡すことができます。

これを例示するため、我々のチャットの例で、別のコンポーネントを使うことにしましょう。これはチャットの受信者を選ぶ画面であり、現在選択中のフレンドがオンラインかどうかを表示します。

```
const friendList = [];

function ChatRecipientPicker() {
  const [recipientID, setRecipientID] = useState(1);
  const isRecipientOnline = useFriendStatus(recipientID);

  return (
    <>
      <Circle color={isRecipientOnline ? 'green' : 'red'} />
      <select
        value={recipientID}
        onChange={e => setRecipientID(Number(e.target.value))}>
        >
          {friendList.map(friend => (
            <option key={friend.id} value={friend.id}>
              {friend.name}
            </option>
          )));
        </select>
      </>
    );
}
```

現在選択中のフレンド ID を `recipientID` という state 変数に保持し、`<select>` ピッカー内で別のフレンドが選択されるごとにそれを更新します。

`useState` フックは `recipientID` という state 変数の最新の値を返しますので、それを `useFriendStatus` カスタムフックに引数として渡すことができます。

```
const [recipientID, setRecipientID] = useState(1);
const isRecipientOnline = useFriendStatus(recipientID);
```

これにより現在選択中のフレンドがオンラインかどうかが分かります。別のフレンドを選択して `recipientID` 変数が更新された場合、`useFriendStatus` フックはこれまで選択されていたフレンドを購読解除して、新しく選択されたフレンドのステータスを購読開始します。

## useYourImagination() {#useyourimagination}

カスタムフックにより、これまでの React コンポーネントでは不可能であった、ロジック共有に関する柔軟性が得られます。フォーム操作、アニメーション、宣言的データ購読、タイマー、さらには我々が思いついたことのない多様なユースケースに対するカスタムフックを記述することが可能です。何より、作ったカスタムフックは React の組み込み機能と同じくらい簡単に使えるようになります。

あまり焦って抽象化を加えないようにしましょう。関数コンポーネントがやれることが増えたので、平均的な関数コンポーネントはこれまでより長いものになるでしょう。それは普通のことですので、いますぐカスタムフックに分割しないといけないとは考えないでください。一方で、カスタムフックをどこで使えば複雑なロジックをシンプルなインターフェースに置き換えたり、ごちゃっとしたコンポーネントを整理したりできるのか、考え始めることをお勧めします。

一例として、その場しのぎで多くのローカル state が含まれるようになった複雑なコンポーネントをお持ちかもしれません。`useState` を使っても更新ロジックの集中化が簡単になるわけではありませんので、それを [Redux](#) のリデューサ (reducer) で書きたくなることがあるでしょう：

```
function todosReducer(state, action) {
  switch (action.type) {
    case 'add':
      return [...state, {
        text: action.text,
        completed: false
      }];
    // ... other actions ...
    default:
      return state;
  }
}
```

リデューサは単独でのテストが非常にやりやすく、複雑な更新ロジックを表現する場合でもスケールします。必要に応じて後でより小さなリデューサに分割することも可能です。しかし、React のローカル state による手軽さの方が好ましい場合もあるでしょうし、他のライブラリをインストールしたくない場合もあるでしょう。

そこで、`useReducer` というフックを書いて、コンポーネントのローカル state をリデューサで管理できるとしたらどうでしょうか？簡略版では以下のようになるでしょう：

```
function useReducer(reducer, initialState) {
  const [state, setState] = useState(initialState);

  function dispatch(action) {
    const nextState = reducer(state, action);
    setState(nextState);
  }

  return [state, dispatch];
}
```

これをコンポーネント内で使うことができ、リデューサを活用してステート管理ができるようになります：

```
function Todos() {
  const [todos, dispatch] = useReducer(todosReducer, []);

  function handleAddClick(text) {
    dispatch({ type: 'add', text });
  }

  // ...
}
```

ローカルステートをリデューサで管理したいという要求はとてもよくあるので、React にその機能を含めています。[フック API リファレンス](#)で他の組み込みフックと共に解説しています。

# 副作用フックの利用法

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。

副作用 (effect) フックにより、関数コンポーネント内で副作用を実行することができるようになります：

```
import React, { useState, useEffect } from 'react';

function Example() {
  const [count, setCount] = useState(0);

  // Similar to componentDidMount and componentDidUpdate:
  useEffect(() => {
    // Update the document title using the browser API
    document.title = `You clicked ${count} times`;
  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

このスニペットは [1つ前のページのカウンターの例](#)に基づいていますが、新しい機能をひとつ加えてあります。クリック回数を含んだカスタムのメッセージをドキュメントのタイトルにセットしているのです。

データの取得、購読 (subscription) の設定、あるいは React コンポーネント内の DOM の手動での変更、といったものはすべて副作用の例です。これらを "副作用 (side-effect)" (あるいは単に "作用 (effect)") と呼ぶことに慣れているかどうかはともかくとしても、これらのようなことをコンポーネントの中で行ったことはあるでしょう。

## ヒント

React のライフサイクルに馴染みがある場合は、useEffect フックを componentDidMount と componentDidUpdate と componentWillUnmount がまとまったものだと考えることができます。

React コンポーネントにおける副作用には 2 種類あります。クリーンアップコードを必要としない副作用と、必要とする副作用です。これらの違いについて詳しく見ていきましょう。

# クリーンアップを必要としない副作用

時に、**React** が DOM を更新した後で追加のコードを実行したいという場合があります。ネットワークリクエストの送信、手動での DOM 改変、ログの記録、といったものがクリーンアップを必要としない副作用の例です。なぜかというとそれらのコードが実行されたあとすぐにそのことを忘れて構わないからです。クラスとフックとでそのような副作用をどのように表現するのか比較してみましょう。

## クラスを使った例

React のクラスコンポーネントでは、`render` メソッド自体が副作用を起こすべきではありません。そこで副作用を起こすのは早すぎます -- 典型的には、副作用は React が DOM を更新したあとに起こすようにしたいのです。

そのため React のクラスでは、副作用は `componentDidMount` と `componentDidUpdate` に記載します。例に戻ると、以下が React のクラスで実装したカウンターコンポーネントであり、React が DOM に変更を加えた後に、ドキュメントのタイトルを更新しています。

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

  componentDidMount() {
    document.title = `You clicked ${this.state.count} times`;
  }

  componentDidUpdate() {
    document.title = `You clicked ${this.state.count} times`;
  }

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

ここで同じコードを 2 回書かなければならなかったことに注意してください。

これは、コンポーネントがマウント直後なのか更新後なのかに関係なく、大抵の場合は同じ副作用を起こしたいからです。概念的には、毎回のレンダー時に起こってほしいのですが、React のクラスコンポーネントにはそのようなメソッドは存在していません。副作用のコードを別のメソッドに抽出することは可能ですが、2 か所でそのメソッドを呼ばなければいけないことに変わりはありません。

では、同じことが `useEffect` フックを用いるとどのように記述できるのか見てていきましょう。

## フックを使った例

以下の例は既にこのページの最初で見たものですが、改めて詳しく見ていきましょう。

```
import React, { useState, useEffect } from 'react';

function Example() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

`useEffect` は何をやっているのか？ このフックを使うことで、レンダー後に何かの処理をしないといけない、ということを React に伝えます。React はあなたが渡した関数を覚えており（これを「副作用（関数）」と呼ぶこととします）、DOM の更新の後にそれを呼び出します。この副作用の場合はドキュメントのタイトルをセットしていますが、データを取得したりその他何らかの命令型の API を呼び出したりすることも可能です。

`useEffect` がコンポーネント内で呼ばれるのはなぜか？ コンポーネント内で `useEffect` を記述することで、副作用内から state である `count`（や任意の props）にアクセスできるようになります。それらは既に関数スコープ内に存在するので、参照するための特別な API は必要ありません。フックは JavaScript のクロージャを活用しており、JavaScript で解決できることに対して React 特有の API を導入することはしません。

`useEffect` は毎回のレンダー後に呼ばれるのか？ その通りです！ デフォルトでは、副作用関数は初回のレンダー時および毎回の更新時に呼び出されます。あとで[カスタマイズする方法](#)について説明します。「マウント」と「更新」という観点で考えるのではなく、「レンダーの後」に副作用は起こる、というように考える方が簡単かもしれません。React は、副作用が実行される時点では DOM が正しく更新され終わっていることを保証します。

## 詳しい説明

副作用について学んだので、以下の行の意味はお分かりかと思います。

```
function Example() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });
}
```

ここでは `count` という state 変数を宣言し、次に副作用を使うことを React に伝えています。`useEffect` フックには関数を渡しています。この関数こそが副作用関数です。この副作用関数内で `document.title` というブラウザ API を使ってドキュメントのタイトルを設定しています。副作用関数は関数スコープ内にあるため最新の `count` の値は副作用内から参照可能です。React がコンポーネントをレンダーする際に React はこの副作用を覚えておき、DOM を更新した後に呼び出します。これは初回を含む毎回のレンダー時に発生します。

経験のある JavaScript 開発者であれば、`useEffect` に渡される関数は毎回のレンダーごとに異なっているということに気付くかもしれません。これは意図的なものです。むしろ、そのようにすることで、古い値を参照してしまう心配なしに副作用関数内から `count` を読むことができるのです。再レンダーごとに、React は違う副作用関数をスケジュールし、前のものを置き換えます。ある意味で、こうすることで副作用はレンダーの結果の一部のようにふるまうようになります -- それぞれの副作用は特定の 1 回のレンダーと結びついているのです。これがなぜ便利なのかについては[このページの後半](#)で明らかになるでしょう。

### ヒント

`componentDidMount` や `componentDidUpdate` と異なり、`useEffect` でスケジュールされた副作用はブラウザによる画面更新をブロックしません。このためアプリの反応がより良く感じられます。大部分の副作用は同期的に行われる必要がありません。同期的に行う必要がある稀なケース（レイアウトの測定など）のために、`useEffect` と同一の API を有する `useLayoutEffect` という別のフックがあります。

## クリーンアップを有する副作用

ここまでに、クリーンアップを必要としない副作用の表現のしかたについて見てきました。しかし幾つかの副作用ではそれが必要です。例えば何らかの外部のデータソースへの購読をセットアップしたいことがあります。そのような場合、メモリリークが発生しないようにクリーンアップが必要です！ クラスの場合とフックの場合とでクリーンアップをどのように行えるのか比較しましょう。

### クラスを使った例

React のクラスでは、典型的にはデータの購読を `componentDidMount` で行い、クリーンアップを `componentWillUnmount` で行います。例えば、フレンドのオンライン状態を購読することができる [ChatAPI](#) というモジュールがあるとしましょう。以下がクラスを使ってその状態を購読し、表示する例です。

```
class FriendStatus extends React.Component {
  constructor(props) {
    super(props);
    this.state = { isOnline: null };
    this.handleStatusChange = this.handleStatusChange.bind(this);
  }

  componentDidMount() {
    ChatAPI.subscribeToFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }

  componentWillUnmount() {
    ChatAPI.unsubscribeFromFriendStatus(
  }
}
```

```

        this.props.friend.id,
        this.handleStatusChange
    );
}

handleStatusChange(status) {
    this.setState({
        isOnline: status.isOnline
    });
}

render() {
    if (this.state.isOnline === null) {
        return 'Loading...';
    }
    return this.state.isOnline ? 'Online' : 'Offline';
}
}

```

ここで、`componentDidMount` と `componentWillUnmount` とがお互いに鏡のように対応していないといけないことに注意してください。ライフサイクルメソッドを使うと、2つのメソッドに書かれているコードが概念上は同一の副作用に関連しているとしても、それらを分割して書かないといけません。

### 補足

目ざとい読者なら、この例が完全に正しいものであるためには `componentDidUpdate` も必要だと気付くかもしれません。今のところは気にしないでおきますが、このページの[後に出てくる節](#)で改めて説明します。

## フックを使った例

このコンポーネントをフックを使ってどのように書けるのか見ていきましょう。

クリーンアップ用の別の副作用が必要だとお考えかもしれません。しかし購読を開始するコードと解除するコードとは密に関連しているため、`useEffect` はそれらを一緒に書けるようにデザインされています。あなたの副作用が関数を返した場合、React はクリーンアップのタイミングが来たらそれを実行するのです。

```

import React, { useState, useEffect } from 'react';

function FriendStatus(props) {
    const [isOnline, setIsOnline] = useState(null);

    useEffect(() => {
        function handleStatusChange(status) {
            setIsOnline(status.isOnline);
        }

        ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
        // Specify how to clean up after this effect:
        return function cleanup() {
            ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
        };
    });

    if (isOnline === null) {
        return 'Loading...';
    }
}

```

```

    }
    return isOnline ? 'Online' : 'Offline';
}

```

**副作用内からなぜ関数を返したのか？** これこそが副作用のクリーンアップのためのオプションの仕組みです。すべての副作用は、それをクリーンアップするための関数を返すことができます。これにより購読を開始するためのロジックと解除するためのロジックを並べて書くことができます。両方とも同じ副作用の一部なのです！

**React は具体的には副作用のクリーンアップをいつ発生させるのか？** React はコンポーネントがアンマウントされるときにクリーンアップを実行します。しかし、すでに学んだ通り、副作用は 1 回だけでなく毎回のレンダー時に実行されます。このため React は、ひとつ前のレンダーによる副作用を、次の副作用を実行する前にもクリーンアップします。この後で、これが**なぜバグの回避につながるのか**、そして**これがパフォーマンスの問題を引き起こしている場合にどのようにしてこの挙動を止めるのか**について説明します。

#### 補足

副作用から名前付きの関数を返す必要はありません。ここでは目的を明示するために `cleanup` という名前にしましたが、アロー関数を返すことも別の名前を付けることも可能です。

## まとめ

`useEffect` を用いることで、コンポーネントのレンダー後に実行される様々な種類の副作用を表現できることを学びました。いくつかの副作用はクリーンアップが必要である可能性があり、その場合は関数を返します：

```

useEffect(() => {
  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
  return () => {
    ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
  };
});

```

クリーンアップフェーズが必要ない副作用もあり、その場合は何も返す必要はありません。

```

useEffect(() => {
  document.title = `You clicked ${count} times`;
});

```

副作用フックは両方のユースケースをひとつの API に統合します。

---

副作用フックの動作について十分わかったと感じる場合や、逆にもううんざりだという場合は、ここで[次のページ（フックのルールについて）](#)に進んでも構いません。

---

## 副作用を使う場合のヒント

このページの残りの部分では、経験のある React 利用者が興味を持つかもしれない `useEffect` の深い概念について説明します。今すぐ読み進める必要があるとは思わないでください。副作用フックについて詳細が知りたくなったらいつでもこのページに戻ってくればいいのです。

### ヒント: 関心を分離するために複数の副作用を使う

フックを導入する動機のページで述べた問題のひとつは、しばしばそれぞれのライフサイクルメソッドに関連のないロジックが含まれ、一方で関連するロジックが複数のメソッドに分割されてしまう、ということです。以下に示すのは、これまでの例で挙げたカウンタとフレンド状態インジケータとを組み合わせたコンポーネントです。

```
class FriendStatusWithCounter extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0, isOnline: null };
    this.handleStatusChange = this.handleStatusChange.bind(this);
  }

  componentDidMount() {
    document.title = `You clicked ${this.state.count} times`;
    ChatAPI.subscribeToFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }

  componentDidUpdate() {
    document.title = `You clicked ${this.state.count} times`;
  }

  componentWillUnmount() {
    ChatAPI.unsubscribeFromFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }

  handleStatusChange(status) {
    this.setState({
      isOnline: status.isOnline
    });
  }
  // ...
}
```

ここで、`document.title` を設定するためのロジックが `componentDidMount` と `componentDidUpdate` に分離していることに注意してください。データ購読のためのロジックもやはり `componentDidMount` と `componentWillUnmount` とに分離しています。そして `componentDidMount` には両方の仕事のためのコードが含まれています。

ではフックはどのようにこの問題を解決するのでしょうか？ [ステートフックを複数回呼べる](#) のと同様の方法で、副作用を複数回利用することができます。このため、互いに関係のないロジックは別の副作用に分離することが可能です。

```
function FriendStatusWithCounter(props) {
  const [count, setCount] = useState(0);
  useEffect(() => {
    document.title = `You clicked ${count} times`;
```

```

});
```

```

const [isOnline, setIsOnline] = useState(null);
useEffect(() => {
  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
  return () => {
    ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
  };
});
// ...
}

```

フックを使うことで、ライフサイクルのメソッド名に基づくのではなく、**実際に何をやっているのか**に基づいてコードを分割できます。React はコンポーネントで利用されているすべての副作用を、指定されている順番で適用していきます。

## 解説:なぜ副作用は毎回の更新ごとに実行されるのか

クラスに慣れていれば、なぜクリーンアップフェーズは、アンマウント時の 1 度だけではなく再レンダー時に毎回発生するのか、と不思議に思っているかもしれません。実践的な例で、この設計によりなぜバグの少ないコンポーネントが作れるようになるのか見てみましょう。

[このページの前の部分](#)で、フレンドがオンラインかどうかを表示する `FriendStatus` コンポーネントの例を示しました。このクラスでは `this.props` の中にある `friend.id` を参照して、コンポーネントがマウントした後にフレンドのステータスを購読し、アンマウント時には購読を解除します：

```

componentDidMount() {
  ChatAPI.subscribeToFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}

componentWillUnmount() {
  ChatAPI.unsubscribeFromFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}

```

ですがコンポーネントが表示されている最中に `friend` プロパティが変わったらどうなるのでしょうか？ このコンポーネントは間違ったフレンドのオンラインステータスを表示し続けてしまいます。これはバグです。しかも誤ったフレンド ID を使って購読解除を呼び出してしまうため、アンマウント時にメモリリークやクラッシュを引き起こしてしまうでしょう。

クラスコンポーネントの場合は、このようなケースに対処するために `componentDidUpdate` を加える必要がありました。

```

componentDidMount() {
  ChatAPI.subscribeToFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}

```

```

componentDidUpdate(prevProps) {
  // Unsubscribe from the previous friend.id
  ChatAPI.unsubscribeFromFriendStatus(
    prevProps.friend.id,
    this.handleStatusChange
  );
  // Subscribe to the next friend.id
  ChatAPI.subscribeToFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}

componentWillUnmount() {
  ChatAPI.unsubscribeFromFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
);
}

```

適切な `componentDidUpdate` 処理をし忘れることが、React アプリケーションにおけるよくあるバグの原因となっていました。

ではこのコンポーネントのフックを利用したバージョンを見てみましょう。

```

function FriendStatus(props) {
  // ...
  useEffect(() => {
    // ...
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });
}

```

動作は変わっておらず、前述のバグも起こらなくなります。

`useEffect` はデフォルトで更新を処理するため、更新のための特別なコードは不要です。新しい副作用を適用する前に、ひとつ前の副作用をクリーンアップします。これを例示するため、このコンポーネントが経時に発生させる可能性のある購読登録と購読解除のシーケンスを示します：

```

ChatAPI.subscribeToFriendStatus(100, handleStatusChange); // Run first effect
ChatAPI.unsubscribeFromFriendStatus(100, handleStatusChange); // Clean up previous effect
ChatAPI.subscribeToFriendStatus(200, handleStatusChange); // Run next effect
ChatAPI.unsubscribeFromFriendStatus(200, handleStatusChange); // Clean up previous effect
ChatAPI.subscribeToFriendStatus(300, handleStatusChange); // Run next effect
// Unmount
ChatAPI.unsubscribeFromFriendStatus(300, handleStatusChange); // Clean up last effect

```

この挙動によりデフォルトで一貫性を保証することができ、クラスコンポーネントでよく見られた更新ロジック書き忘れによるバグを防止することができます。

## ヒント:副作用のスキップによるパフォーマンス改善

いくつかの場合では、副作用のクリーンアップと適用とをレンダーごとに毎回行うことはパフォーマンスの問題を引き起こす可能性があります。クラスコンポーネントの場合、この問題は `componentDidUpdate` の内部で `prevProps` や `prevState` と比較するコードを加えることで解決できました。

```
componentDidUpdate(prevProps, prevState) {
  if (prevState.count !== this.state.count) {
    document.title = `You clicked ${this.state.count} times`;
  }
}
```

これはよくある要求なので、`useEffect` フックの API にはこの動作が組み込まれています。再レンダー間で特定の値が変わっていない場合には副作用の適用をスキップするよう、React に伝えることができます。そのためには、`useEffect` のオプションの第 2 引数として配列を渡してください。

```
useEffect(() => {
  document.title = `You clicked ${count} times`;
}, [count]); // Only re-run the effect if count changes
```

上記の例では、第 2 引数として `[count]` を渡しています。どういう意味でしょうか？もし `count` が `5` で、次回のコンポーネントのレンダー時にも `count` がまだ `5` であった場合、React は前回のレンダー時に覚えておいた `[5]` と今回のレンダーの `[5]` とを比較します。配列内のすべての要素が同一 (`5 === 5`) ですので、React は副作用をスキップします。これが最適化です。

再レンダー時に `count` が `6` に変更されている場合、前回レンダー時に覚えておいた `[5]` と今回のレンダー時の `[6]` という配列とを比較します。今回は `5 !== 6` ですので React は副作用を再適用します。配列内に複数の要素がある場合、React は配列内の要素のうちひとつでも変わっている場合に副作用を再実行します。

クリーンアップフェーズがある副作用でも同様に動作します：

```
useEffect(() => {
  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
  return () => {
    ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
  };
}, [props.friend.id]); // Only re-subscribe if props.friend.id changes
```

将来的には、ビルド時の変換で第 2 引数を自動で加えられるようになるかもしれません。

### 補足

この最適化を利用する場合、**時間の経過とともに変化し副作用によって利用される、コンポーネントスコープの値（`props` や `state` など）** がすべて配列に含まれていることを確認してください。さもないとあなたのコードは以前のレンダー時の古い値を参照してしまうことになります。その他の最適化のオプションについては [フック API リファレンス](#) で説明しています。

もしも副作用とそのクリーンアップを 1 度だけ（マウント時とアンマウント時にのみ）実行したいという場合、空の配列（`[]`）を第 2 引数として渡すことができます。こうすることで、あなたの副作用は `props` や `state` の値のいずれにも依存していないため再実行する必要が一切ない、ということを React に伝えることができます。これは特別なケースとして処理されているわけではなく、依存配列を普通に処理すればそうなるというだけの話です。

空の配列（`[]`）を渡した場合、副作用内では `props` と `state` の値は常にその初期値のままになります。`[]` を渡すことはおなじみの `componentDidMount` と `componentWillUnmount` による概念と似ているように感じるでしょうが、通常は[こちら](#)や[こちら](#)のように、副作用を過度に再実行しないためのよりよい解決方法があります。また `useEffect` はブラウザが描画し終えた後まで遅延されますので、追加の作業をしてもそれほど問題にならないということをお忘れなく。

`eslint-plugin-react-hooks` パッケージの `exhaustive-deps` ルールを有効にすることをお勧めします。これは依存の配列が正しく記述されていない場合に警告し、修正を提案します。

## 次のステップ

おめでとうございます！長いページでしたが、最終的に副作用に関するほとんどの疑問が解決していることを望みます。これでステートフックと副作用フックの両方を学んだので、それらを組み合わせてやれることがたくさんあります。クラスコンポーネントにおけるほとんどのユースケースがカバーされていますが、足りない部分については[他のフック](#)が役立つかかもしれません。

また、[動機](#)のところで述べた問題をフックがどのように解決するのかについてもわかり始めてきたでしょう。副作用のクリーンアップがどのようにして `componentDidUpdate` と `componentWillUnmount` との間でのコードの重複を防ぎ、関係したコードを並べて書くことができるようになり、バグの少ないコードを記述できるようにするのかを見てきました。また目的別に副作用を分割する方法も学びましたが、これはクラスでは全く不可能なことでした。

この時点で、一体フックがどのように動作しているのか疑問に感じているかもしれません。`useState` のそれぞれの呼び出しがどの `state` 変数に対応しているのかを、React はどうやって知るのでしょうか？更新のたびに、前回と今回の副作用とを React はどうやって対応付けるのでしょうか？次のページでは[フックのルール](#)について学びます -- このルールはフックが動作するために必須のものです。

# フックに関するよくある質問

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。

このページではフックに関するよくある質問にいくつかお答えします。

- 導入の指針

- フックが使える React のバージョンはどれですか？
- クラスコンポーネントを全部書き換える必要があるのですか？
- クラスではできず、フックができるようになることは何ですか？
- これまでの React の知識はどの程度使えますか？
- フック、クラスのいずれを使うべきですか、あるいはその両方でしょうか？
- フックはクラスのユースケースのすべてをカバーしていますか？
- フックはレンダープロップや高階コンポーネントを置き換えるものですか？
- Redux の connect() や React Router といった人気の API はフックによりどうなりますか？
- フックは静的型付けと組み合わせてうまく動きますか？
- フックを使ったコンポーネントはどのようにテストするのですか？
- Lint ルールは具体的に何を強制するのですか？

- クラスからフックへ

- 個々のライフサイクルメソッドはフックとどのように対応するのですか？
- フックでデータの取得をどのように行うのですか？
- インスタンス変数のようなものにはありますか？
- state 変数は 1 つにすべきですか、たくさん使うべきですか？
- コンポーネントの更新のときだけ副作用を実行することは可能ですか？
- 前回の props や state はどうすれば取得できますか？
- 関数内で古い props や state が見えているのはなぜですか？
- どうすれば getDerivedStateFromProps を実装できますか？
- forceUpdate のようなものにはありますか？
- 関数コンポーネントへの ref を作ることは可能ですか？
- DOM ノードの位置やサイズの測定はどのように行うのですか？
- const [thing, setThing] = useState() というのはどういう意味ですか？

- パフォーマンス最適化

- 更新時に副作用をスキップすることはできますか？
- 依存の配列から関数を省略しても大丈夫ですか？
- 副作用の依存リストが頻繁に変わりすぎる場合はどうすればよいですか？
- どうすれば `shouldComponentUpdate` を実装できますか？
- 計算結果のメモ化はどのように行うのですか？
- 計算量の大きいオブジェクトの作成を遅延する方法はありますか？
- レンダー内で関数を作るせいでフックは遅くなるのではないですか？
- どうすれば複数のコールバックを深く受け渡すのを回避できますか？
- useCallback からの頻繁に変わる値を読み出す方法は？

- **内部の仕組み**
  - React はフック呼び出しとコンポーネントとをどのように関連付けているのですか？
  - フックの先行技術にはどのようなものがありますか？

## 導入の指針

### フックが使える React のバージョンはどれですか？

React バージョン 16.8.0 より、以下においてフックの安定版の実装が含まれています。

- React DOM
- React Native
- React DOM Server
- React Test Renderer
- React Shallow Renderer

フックを利用するには、すべての React のパッケージが **16.8.0** 以上である必要があります。例えば React DOM の更新を忘れた場合、フックは動作しません。

React Native はバージョン 0.59 以降でフックをサポートします。

### クラスコンポーネントを全部書き換える必要があるのですか？

いいえ。React からクラスを削除する[予定はありません](#) -- 我々はみなプロダクトを世に出し続ける必要があり、クラスを書き換えていたる余裕はありません。新しいコードでフックを試すことをお勧めします。

### クラスではできず、フックができるようになることは何ですか？

フックにより、コンポーネント間で機能を再利用するためのパワフルで表現力の高い手段が得られます。["独自フックの作成"](#)を読めばできることの概要が掴めるでしょう。React のコアチームメンバーによって書かれた[この記事](#)により、フックによって新たにもたらされる可能性についての洞察が得られます。

### これまでの React の知識はどの程度使えますか？

フックとは、state やライフサイクル、コンテキストや ref といった、あなたが既に知っている React の機能をより直接的に利用できるようにする手段です。React の動作が根本的に変わるものではありませんし、コンポーネントや props、トップダウンのデータの流れについての知識はこれまでと同様に重要です。

もちろんフックにはフックなりの学習曲線があります。このドキュメントに足りないを見つければ、お手伝いします。

### フック、クラスのいずれを使うべきですか、あるいはその両方でしょうか？

準備ができしだい、新しいコンポーネントでフックを試すことをお勧めします。チームの全員の準備が完了し、このドキュメントに馴染んでいることを確かめましょう。(例えばバグを直すなどの理由で) 何にせよ書き換える予定の場合を除いては、既存のクラスをフックに書き換えることはお勧めしません。

クラスコンポーネントの定義内でフックを使うことはできませんが、クラス型コンポーネントとフックを使った関数コンポーネントとを 1 つのコンポーネントツリー内で混在させることは全く問題ありません。あるコンポーネントがクラスで書かれているかフックを用いた関数で書かれているかというのは、そのコンポーネントの実装の詳細です。長期的には、フックが React のコンポーネントを書く際の第一選択となることを期待しています。

## フックはクラスのユースケースのすべてをカバーしていますか？

我々の目標はできるだけ早急にフックがすべてのクラスのユースケースをカバーできるようにすることです。まだ使用頻度の低い `getSnapshotBeforeUpdate`、`getDerivedStateFromError` および `componentDidCatch` についてはフックでの同等物が存在していませんが、すぐに追加する予定です。

## フックはレンダープロップや高階コンポーネントを置き換えるものですか？

レンダープロップや高階コンポーネントは、ひとつの子だけをレンダーすることがよくあります。フックはこのようなユースケースを実現するより簡単な手段だと考えています。これらのパターンには引き続き利用すべき場面があります（例えば、バーチャルスクローラーコンポーネントは `renderItem` プロパティを持つでしょうし、コンテナコンポーネントは自分自身の DOM 構造を有しているでしょう）。とはいえた抵の場合ではフックで十分であり、フックがツリーのネストを減らすのに役立つでしょう。

## Redux の `connect()` や React Router といった人気の API はフックによりどうなりますか？

これまでと同様に全く同じ API を使用し続けることができます。それらは動作し続けます。

React Redux は v7.1.0 より [フック API をサポート](#)しており、`useDispatch` や `useSelector` といったフックを提供しています。

React Router は v5.1 より [フック API をサポート](#) しています。

他のライブラリも、将来的にフックをサポートするかもしれません。

## フックは静的型付けと組み合わせてうまく動きますか？

フックは静的型付けを念頭に設計されました。フックは関数ですので、高階コンポーネントのようなパターンと比較しても正しく型付けするのは容易です。最新版の Flow と TypeScript における React の型定義には、React のフックについてのサポートが含まれています。

重要なことですが、もしより厳密に型付けしたい場合は、カスタムフックを使うことで React API に何らかの制約を加えることが可能ですが、React は基本部品を提供しますが、最初から提供されているものと違う方法でそれらを様々な組み合わせることができます。

## フックを使ったコンポーネントはどのようにテストするのですか？

React の観点から見れば、フックを使ったコンポーネントは単なる普通のコンポーネントです。あなたのテストソリューションが React の内部動作に依存しているのでない場合、フックを使ったコンポーネントのテストのやり方は、あなたが普段コンポーネントをテストしているやり方と変わらないはずです。

補足

テストのレシピ集にコピー・ペーストで使えるたくさんの例が掲載されています。

例えばこのようなカウンタコンポーネントがあるとしましょう：

```
function Example() {
  const [count, setCount] = useState(0);
  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

これを React DOM を使ってテストします。ブラウザでの挙動と確実に合致させるため、これをレンダーしたり更新したりするコードを `ReactTestUtils.act()` でラップします：

```
import React from 'react';
import ReactDOM from 'react-dom/client';
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 effect
  act(() => {
    ReactDOM.createRoot(container).render(<Counter />);
  });
  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 effect
  act(() => {
    button.dispatchEvent(new MouseEvent('click', {bubbles: true}));
  });
  expect(label.textContent).toBe('You clicked 1 times');
  expect(document.title).toBe('You clicked 1 times');
});
```

`act()` を呼び出すと内部の副作用も処理されます。

カスタムフックをテストしたい場合は、テスト内でコンポーネントを作つて中でそのカスタムフックを使うようにしてください。そうすればそのコンポーネントをテストできます。

ボイラープレートを減らすため、エンドユーザが使うのと同じ形でコンポーネントを使ってテストが記述できるように設計されている、[React Testing Library](#) の利用をお勧めします。

詳細については、[テストのレシピ集](#)をご覧ください。

## Lint ルールは具体的に何を強制するのですか？

我々は [ESLint プラグイン](#) を提供しており、これにより [フックのルール](#) を強制してバグを減らすことができます。このルールは、`use` で始まり大文字が続くような名前の関数はすべてフックであると仮定します。これは不完全な推測手段であり過剰検出があるかもしれないことは認識していますが、エコシステム全体での規約なくしてはフックはうまく動作しません。また名前を長くするとフックを利用したり規約を守ったりしてくれなくなるでしょう。

具体的には、このルールは以下を強制します：

- フックの呼び出しが `PascalCase` 名の関数内（コンポーネントと見なされます）か、あるいは他の `useSomething` 式の名前の関数内（カスタムフックと見なされます）にあること。
- すべてのレンダー間でフックが同じ順番で呼び出されること。

これ以外にも幾つかの推測を行っており、また、バグ検出と過剰検出抑制とのバランスを調整していくなかで、これらは将来的に変わる可能性があります。

## クラスからフックへ

### 個々のライフサイクルメソッドはフックとどのように対応するのですか？

- `constructor`：関数コンポーネントはコンストラクタを必要としません。`state` は `useState` を呼び出すときに初期化します。初期 `state` の計算が高価である場合、`useState` に関数を渡すことができます。
- `getDerivedStateFromProps`：レンダー中に更新をスケジューリングします。
- `shouldComponentUpdate`：ページ下部の `React.memo` についての説明を参照してください。
- `render`：これは関数コンポーネントの本体そのものです。
- `componentDidMount`, `componentDidUpdate`, `componentWillUnmount`：これらのあらゆる組み合わせは `useEffect` フックで表現できます（これやこれのような頻度の低いケースも含め）。
- `getSnapshotBeforeUpdate`、`componentDidCatch` および `getDerivedStateFromError`：フックによる同等物はまだ存在していませんが、近日中に追加される予定です。

### フックでデータの取得をどのように行うのですか？

まずはこちらの[小さなデモ](#)をご覧ください。フックを使ってデータの取得をする方法について詳しく学ぶには[こちらの記事](#)を参照してください。

## インスタンス変数のようなものはありませんか？

はい！`useRef()` フックは DOM への参照を保持するためだけにあるのではありません。“ref” オブジェクトは汎用のコンテナであり、その `current` プロパティの値は書き換え可能かつどのような値でも保持することができますので、クラスのインスタンス変数と同様に利用できます。

例えば `useEffect` 内から “ref” オブジェクトを書き換えることができます。

```
function Timer() {
  const intervalRef = useRef();

  useEffect(() => {
    const id = setInterval(() => {
      // ...
    });
    intervalRef.current = id;
    return () => {
      clearInterval(intervalRef.current);
    };
  });
  // ...
}
```

単にインターバルをセットしたいだけであれば（`id` はこの副作用内でローカルでよいので）この `ref` は必要ないところですが、もしもイベントハンドラ内でインターバルをクリアしたい場合には便利です：

```
// ...
function handleCancelClick() {
  clearInterval(intervalRef.current);
}
// ...
```

概念的には、`ref` はクラスにおけるインスタンス変数と似たものだと考えることができます。[遅延初期化](#)をしたい場合を除き、レンダーの最中に `ref` を書き換えることはしないでください。その代わり、通常 `ref` はイベントハンドラや副作用内でだけ書き換えるようにしましょう。

## state 変数は 1 つにすべきですか、たくさん使うべきですか？

これまでクラスを使っていたなら、`useState()` を 1 回だけ呼んで、1 つのオブジェクト内にすべての `state` を入れたくなるかもしれません。そうしたければそのようにすることもできます。以下はマウスの動作を追跡するコンポーネントの例です。位置やサイズの情報を 1 つのローカル `state` に保持しています。

```
function Box() {
  const [state, setState] = useState({ left: 0, top: 0, width: 100, height: 100 });
  // ...
}
```

ここで例えば、ユーザがマウスを動かしたときに `left` と `top` を変更したいとしましょう。この際、これらのフィールドを古い `state` に手動でマージしないといけないことに注意してください：

```
// ...
useEffect(() => {
  function handleWindowMouseMove(e) {
    // Spreading "...state" ensures we don't "lose" width and height
    setState(state => ({ ...state, left: e.pageX, top: e.pageY }));
  }
  // Note: this implementation is a bit simplified
  window.addEventListener('mousemove', handleWindowMouseMove);
  return () => window.removeEventListener('mousemove', handleWindowMouseMove);
}, []);
// ...
```

これは state 変数を更新する時には変数の値が置換されるからです。これは更新されるフィールドがオブジェクトにマージされるというクラスでの `this.setState` の挙動とは異なります。

自動マージがないとつらい場合は、`useLegacyState` のようなカスタムフックを書いてオブジェクト型の state の更新をマージするようにすることはできます。しかし、我々は代わりに、**どの値が一緒に更新されやすいのかに基づいて、state を複数の state 変数に分割することをお勧めします。**

例えば、コンポーネントの state を `position` と `size` という複数のオブジェクトに分割して、マージを行わなくとも `position` を常に新たな値で置換するようにできるでしょう。

```
function Box() {
  const [position, setPosition] = useState({ left: 0, top: 0 });
  const [size, setSize] = useState({ width: 100, height: 100 });

  useEffect(() => {
    function handleWindowMouseMove(e) {
      setPosition({ left: e.pageX, top: e.pageY });
    }
  }, []);
```

互いに独立した state 変数を分割することには別の利点もあります。そうすることで後からそのロジックをカスタムフックに抽出しやすくなるのです。例えば：

```
function Box() {
  const position = useWindowPosition();
  const [size, setSize] = useState({ width: 100, height: 100 });
  // ...

}

function useWindowPosition() {
  const [position, setPosition] = useState({ left: 0, top: 0 });
  useEffect(() => {
    // ...
  }, []);
  return position;
}
```

`position` の state 変数に対する `useState` の呼び出しとそれに関連する副作用を、それらのコードを変えずにカスタムフックに移行できたことに注意してください。もしすべての state が单一のオブジェクトに入っていたら、抽出するのはもっと困難だったでしょう。

すべての state を 1 つの `useState` 呼び出しに含めても動作しますし、フィールドごとに別に `useState` を持たせることでも動作はします。しかしこれらの両極端の間でうまくバランスを取り、少数の独立した state 変数に関連する state をグループ化することで、コンポーネントは最も読みやすくなります。state のロジックが複雑になった場合は、それを [リデューサで管理する](#)か、カスタムフックを書くことをお勧めします。

## コンポーネントの更新のときだけ副作用を実行することは可能ですか？

これは稀なユースケースです。必要であれば、[変更可能な ref](#) を使って、初回レンダー中なのか更新中なのかに対応する真偽値を手動で保持し、副作用内でその値を参照するようにすることができます（このようなことを何度もやる場合は、そのためのカスタムフックを書くことができます）。

## 前回の props や state はどうすれば取得できますか？

前回の props や state が欲しくなるというケースは 2 つあります。

ひとつは、前回の props を [副作用のクリーンアップ](#)に使用したいという場合です。例えば、`userId` プロパティに基づいてソケットを購読する副作用を書いている場合などです。`userId` プロパティが変化した場合、[ひとつの `useId`](#) の購読を解除して次のものを購読したくなるでしょう。ですがこれを実現するのに、特別なことをする必要はありません：

```
useEffect(() => {
  ChatAPI.subscribeToSocket(props.userId);
  return () => ChatAPI.unsubscribeFromSocket(props.userId);
}, [props.userId]);
```

上記の例では、`userId` が `3` から `4` に変わった場合、`ChatAPI.unsubscribeFromSocket(3)` が最初に走り、その後に `ChatAPI.subscribeToSocket(4)` が走ります。クリーンアップ関数は「前回」の `userId` をクロージャとしてキャプチャしていますので、前回の値を取得する必要はありません。

別の場面では、[props や他の state の変更に基づいて state を調整したい](#)ということがあるかもしれません。これはめったに必要なものではありませんし、通常はコードに重複した冗長な state があるというサインです。しかしこれが必要な稀なパターンでは、[前の state や props を state に保存してレンダー中に更新することができます](#)。

これまで `usePrevious` というカスタムフックを使うことを提案していましたが、ほとんどのユースケースは上記の 2 つのパターンのいずれかに当てはまることがわかりました。もしもこのユースケースが当てはまらない場合は、[値を ref に保持](#)し、必要に応じて手作業でアップデートすることができます。レンダー中に `ref` を読み出したり変更したりするとコンポーネントの挙動を予想・理解するのが難しくなるため、避けるようにしてください。

## 関数内で古い props や state が見えているのはなぜですか？

イベントハンドラにせよ副作用関数にせよ、コンポーネント内に書かれた関数からは、その関数が作成された時の props や state が「見え」ます。以下のようない例を考えてみましょう：

```
function Example() {
  const [count, setCount] = useState(0);

  function handleAlertClick() {
    setTimeout(() => {
      alert('You clicked on: ' + count);
    }, 3000);
  }
}
```

```

return (
  <div>
    <p>You clicked {count} times</p>
    <button onClick={() => setCount(count + 1)}>
      Click me
    </button>
    <button onClick={handleAlertClick}>
      Show alert
    </button>
  </div>
);
}

```

最初に "Show alert" ボタンをクリックして、次にカウンタを増加させた場合、アラートダイアログに表示されるのは **"Show alert" ボタンをクリックした時点での count** 変数の値になります。これにより props や state が変わらないことを前提として書かれたコードによるバグが防止できます。

非同期的に実行されるコールバック内で、意図的に state の最新の値を読み出したいという場合は、その値を `ref` 内に保持して、それを書き換えて読み出したりすることができます。

最後に、古い props や state が見えている場合に考えられる他の理由は、「依存の配列」による最適化を使った際に正しく依存する値の全部を指定しなかった、というものです。例えば副作用フックの第 2 引数に `□` を指定したにも関わらず副作用内で `someProps` を読み出しているという場合、副作用関数内では `someProps` の初期値がずっと見え続けることになります。解決方法は依存配列自体を削除するか、配列の中身を修正することです。[関数の扱い方](#)、および依存する値の変化を誤って無視することなく副作用の実行回数を減らすための[よくある手法](#)についてもご覧ください。

#### 補足

`exhaustive-deps` という ESLint のルールを `eslint-plugin-react-hooks` パッケージの一部として提供しています。依存配列が正しく指定されていない場合に警告し、修正を提案します。

## どうすれば `getDerivedStateFromProps` を実装できますか？

おそらくそのようなものは**必要ない**のですが、これが本当に必要になる稀なケースでは（例えば `<Transition>` コンポーネントを実装するときなど）、レンダーの最中に state を更新することができます。React は最初のレンダーの終了直後に更新された state を使ってコンポーネントを再実行しますので、計算量は高くなりません。

以下の例では、`row` プロパティの前回の値を state 変数に格納し後で比較できるようにしています：

```

function ScrollView({row}) {
  const [isScrollingDown, setIsScrollingDown] = useState(false);
  const [prevRow, setPrevRow] = useState(null);

  if (row !== prevRow) {
    // Row changed since last render. Update isScrollingDown.
    setIsScrollingDown(prevRow === null || row > prevRow);
    setPrevRow(row);
  }

  return `Scrolling down: ${isScrollingDown}`;
}

```

これは最初は奇妙に見えるかもしれません、これまでも概念的には `getDerivedStateFromProps` はレンダー中に更新を行うというのがまさに目的でした。

## forceUpdate のようなものはありますか？

`useState` フックと `useReducer` フックのいずれも、前回と今回で値が同じである場合は更新を回避します。適当な場所で `state` を変化させた後に `setState` を呼び出しても再レンダーは発生しません。

通常、React でローカル `state` を直接変更すべきではありません。しかし避難ハッチとして、カウンターを使って `state` が変化していない場合でも再レンダーを強制することが可能です。

```
const [ignored, forceUpdate] = useReducer(x => x + 1, 0);

function handleClick() {
  forceUpdate();
}
```

可能であればこのパターンは避けるようにしてください。

## 関数コンポーネントへの ref を作ることは可能ですか？

このようなことをする必要はありませんが、命令型のメソッドを親コンポーネントに公開するために `useImperativeHandle` フックを利用することができます。

## DOM ノードの位置やサイズの測定はどのように行うのですか？

DOM ノードの位置やサイズを測定するための基本的な方法として、[コールバック形式の ref](#) が利用できます。React は `ref` が異なるノードに割り当てられるたびにコールバックを呼び出します。こちらの[小さなデモ](#)をご覧ください。

```
function MeasureExample() {
  const [height, setHeight] = useState(0);

  const measuredRef = useCallback(node => {
    if (node !== null) {
      setHeight(node.getBoundingClientRect().height);
    }
  }, []);

  return (
    <>
      <h1 ref={measuredRef}>Hello, world</h1>
      <h2>The above header is {Math.round(height)}px tall</h2>
    </>
  );
}
```

この例で `useRef` を使わなかったのは、オブジェクト型の `ref` には現在値が変わった時にそれを通知する機能がないためです。コールバック `ref` を使うことで、[子コンポーネントが測定されたノードを（例えばクリックに応じて）後から表示する場合でも、親コンポーネントの側でその変更について通知を受け取り、測定値を反映させることができます](#)。

`useCallback` の依存値の配列として `[]` を渡したことに注意してください。これにより我々の `ref` コールバックが再レンダーごとに変化しないことが保証され、React が不必要にその関数を呼ばないで済みます。

この例では、レンダーされている `<h1>` はどの再レンダー間でも同じように存在するため、コールバック `ref` はコンポーネントのマウント時とアンマウント時にのみ呼び出されます。コンポーネントのリサイズが発生した際に毎回通知を受け取りたい場合は、`ResizeObserver` や、これを使って作成されているサードパーティのフックの利用を検討してください。

お望みであれば再利用可能なフックとして [このロジックを抽出](#) できます。

```
function MeasureExample() {
  const [rect, ref] = useClientRect();
  return (
    <>
      <h1 ref={ref}>Hello, world</h1>
      {rect !== null &&
        <h2>The above header is {Math.round(rect.height)}px tall</h2>
      }
    </>
  );
}

function useClientRect() {
  const [rect, setRect] = useState(null);
  const ref = useCallback(node => {
    if (node !== null) {
      setRect(node.getBoundingClientRect());
    }
  }, []);
  return [rect, ref];
}
```

`const [thing, setThing] = useState()` というのはどういう意味ですか？

この構文に馴染みがない場合はステートフックのドキュメント内の[説明](#)をご覧ください。

## パフォーマンス最適化

### 更新時に副作用をスキップすることはできますか？

はい。[条件付きで副作用を実行する](#) を参照してください。これがデフォルトの動作になっていないのは、更新時の対応を忘れることが[バグの元になる](#) からです。

### 依存の配列から関数を省略しても大丈夫ですか？

いいえ、一般的には省略できません。

```
function Example({ someProp }) {
  function doSomething() {
    console.log(someProp);
  }

  useEffect(() => {
    doSomething();
  }, []); // 🔴 This is not safe (it calls `doSomething` which uses `someProp`)
}
```

副作用関数の外側にある関数内でどの props や state が使われているのか覚えておくのは大変です。ですので副作用関数内で使われる関数は副作用関数内で宣言するのがよいでしょう。そうすればコンポーネントスコープ内のどの値に副作用が依存しているのかを把握するのは容易です。

```
function Example({ someProp }) {
  useEffect(() => {
    function doSomething() {
      console.log(someProp);
    }

    doSomething();
  }, [someProp]); // ✅ OK (our effect only uses `someProp`)
}
```

このようにした後で、やはりコンポーネントスコープ内のどの値も使用していないのであれば、`□` を指定することは安全です：

```
useEffect(() => {
  function doSomething() {
    console.log('hello');
  }

  doSomething();
}, []); // ✅ OK in this example because we don't use *any* values from component scope
```

ユースケースによっては、以下に述べるような選択肢もあります。

### 補足

`eslint-plugin-react-hooks` パッケージの一部として `exhaustive-deps` という ESLint のルールを提供しています。更新の一貫性が保たれていないコンポーネントを見つけるのに役立ちます。

これがなぜ重要なのか説明します。

`useEffect`、`useLayoutEffect`、`useMemo`、`useCallback` あるいは `useImperativeHandle` の最後の引数として [依存する値のリスト](#) を渡す場合、コールバック内部で使われ React のデータの流れに関わる値が、すべて含まれている必要があります。すなわち `props` や `state` およびそれらより派生するあらゆるものです。

関数を依存のリストから安全に省略できるのは、その関数（あるいはその関数から呼ばれる関数）が `props`、`state` ないしそれらから派生する値のいずれも含んでいない場合のみです。以下の例にはバグがあります。

```
function ProductPage({ productId }) {
  const [product, setProduct] = useState(null);

  async function fetchProduct() {
    const response = await fetch(`http://myapi/product/${productId}`); // Uses
    const json = await response.json();
    setProduct(json);
  }
}
```

```
useEffect(() => {
  fetchProduct();
}, []); // 🚫 Invalid because `fetchProduct` uses `productId`
// ...
}
```

**推奨される修正方法は、この関数を副作用内に移動することです。**これにより、副作用がどの props や state を利用しているのか把握しやすくなり、それらが指定されていることを保証しやすくなります。

```
function ProductPage({ productId }) {
  const [product, setProduct] = useState(null);

  useEffect(() => {
    // By moving this function inside the effect, we can clearly see the values it
    // uses.
    async function fetchProduct() {
      const response = await fetch('http://myapi/product/' + productId);
      const json = await response.json();
      setProduct(json);
    }

    fetchProduct();
  }, [productId]); // ✅ Valid because our effect only uses productId
  // ...
}
```

これにより、要らなくなったレスポンスに対して副作用内でローカル変数を使って対処することも可能になります。

```
useEffect(() => {
  let ignore = false;
  async function fetchProduct() {
    const response = await fetch('http://myapi/product/' + productId);
    const json = await response.json();
    if (!ignore) setProduct(json);
  }

  fetchProduct();
  return () => { ignore = true };
}, [productId]);
```

副作用内に関数を移動したことでの依存リスト内にこの関数を含めないでよくなりました。

### ヒント

フックでデータを取得する方法について[こちらの小さなデモ](#)および[こちらの記事](#)をご覧ください。

**何らかの理由で副作用内に関数を移動できないという場合、他にとりうる選択肢がいくつかあります。**

- **そのコンポーネントの外部にその関数を移動できないか考えましょう。**その場合、関数は props や state を参照していないことが保証されるので、依存のリストに含まざに済むようになります。
- 使おうとしている関数が純粋な計算のみを行い、レンダー中に呼んで構わないものであるなら、その関数を代わりに**副作用の外部で呼ぶ**ようにして、副作用中ではその返り値に依存するようにします。

- 最終手段として、関数を依存リストに加えつつ、`useCallback` を使ってその定義をラップすることができます。これにより、関数自体の依存が変わらない限り関数も変化しないことを保証できます。

```
function ProductPage({ productId }) {
  // ✅ Wrap with useCallback to avoid change on every render
  const fetchProduct = useCallback(() => {
    // ... Does something with productId ...
  }, [productId]); // ✅ All useCallback dependencies are specified

  return <ProductDetails fetchProduct={fetchProduct} />;
}

function ProductDetails({ fetchProduct }) {
  useEffect(() => {
    fetchProduct();
  }, [fetchProduct]); // ✅ All useEffect dependencies are specified
  // ...
}
```

上記の例では関数を依存リストに含める必要があることに注意してください。これにより `ProductPage` の `productId` プロパティが変化した場合に自動的に `ProductDetail` コンポーネント内でデータの再取得が発生するようになります。

## 副作用の依存リストが頻繁に変わりすぎる場合はどうすればよいですか？

しばしば、ある副作用がとても頻繁に変化する `state` を利用しないといけない場合があります。依存のリストからその `state` を省略したくなるかもしれません、通常それはバグになります。

```
function Counter() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    const id = setInterval(() => {
      setCount(count + 1); // This effect depends on the `count` state
    }, 1000);
    return () => clearInterval(id);
  }, []); // ⚡ Bug: `count` is not specified as a dependency

  return <h1>{count}</h1>;
}
```

依存のリストが空であるということ（□）は、コンポーネントのマウント時に副作用が一度のみ実行され、毎回の再レンダー時には実行されないということを意味します。ここでの問題は、副作用コールバックが実行された時点で `count` の値が `0` に設定されたクロージャを作成したため、`setInterval` 内のコールバックで `count` の値が変わらなくなってしまう、ということです。毎秒ごとにこのコールバックは `setCount(0 + 1)` を呼び出すので、カウントは 1 のまま変わらなくなってしまいます。

依存のリストとして `[count]` を指定すればバグは起きなくなりますが、その場合値が変化するたびにタイマーがリセットされることになります。事実上それぞれの `setInterval` は一度しか実行されずに（`setTimeout` のように）クリアされてしまうのです。これは望ましい動作ではありません。これを修正するため、`useState 関数形式による更新` を利用することができます。これにより `state` の現在値を参照せずに `state` がどのように更新されるべきかを指定できます。

```
function Counter() {
  const [count, setCount] = useState(0);

  useEffect(() => {
```

```

const id = setInterval(() => {
  setCount(c => c + 1); // ✅ This doesn't depend on `count` variable outside
}, 1000);
return () => clearInterval(id);
[], []); // ✅ Our effect doesn't use any variables in the component scope

return <h1>{count}</h1>;
}

```

(`setCount` 関数については同一性が保たれることができるので、省略して構いません)

これで、`setInterval` のコールバックは 1 秒に 1 回実行されますが、内部の `setCount` は `count` の最新の値（この例では `c`）を参照できるようになります。

より複雑なケース（ある `state` が別の `state` に依存している場合など）においては、`state` 更新のロジックを `useReducer` フックを使って副作用の外部に移動することを考慮してください。[こちらの記事](#)にこのやり方についての例があります。`useReducer` から返される `dispatch` 関数は常に同一性が保たれます。これはリデューサ (reducer) 関数がコンポーネント内で宣言されており `props` を読み出している場合でも同様です。

最終手段として、クラスにおける `this` のようなものが欲しい場合は、`ref` を使ってミュータブルな値を保持させることができます。そうすればその値を読み書き可能です。例えば：

```

function Example(props) {
  // Keep latest props in a ref.
  const latestProps = useRef(props);
  useEffect(() => {
    latestProps.current = props;
  });

  useEffect(() => {
    function tick() {
      // Read latest props at any time
      console.log(latestProps.current);
    }

    const id = setInterval(tick, 1000);
    return () => clearInterval(id);
  }, []);
}

```

ミュータブルな値に依存することでコンポーネントの挙動が予測しづらくなるため、これは代替手段が思いつかない場合にのみ利用してください。うまくフックに移行できないパターンがあった場合は動作するコード例を添えて [issue](#) を作成していただければお手伝いします。

## どうすれば `shouldComponentUpdate` を実装できますか？

関数コンポーネントを `React.memo` でラップして `props` を浅く比較するようにしてください。

```

const Button = React.memo((props) => {
  // your component
});

```

これがフックになっていないのは、フックと違って組み合わせ可能ではないからです。`React.memo` は `PureComponent` の同等物ですが、`props` のみを比較するという違いがあります（新旧の `props` を受け取るカスタムの比較関数を 2 つめの引数として加えることができます。その関数が `true` を返した場合はコンポーネントの更新はスキップされます）。

`React.memo` は `state` を比較しませんが、これは比較可能な単一の `state` オブジェクトが存在しないからです。しかし子コンポーネント側も純粋にしておくことや、`useMemo` を使って個々のコンポーネントを最適化することができます。

## 計算結果のメモ化はどのように行うのですか？

`useMemo` フックを使うと、前の計算結果を「記憶」しておくことで、複数のレンダー間で計算結果をキャッシュすることができます。

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

このコードは `computeExpensiveValue(a, b)` を呼び出します。しかし依存である `[a, b]` の組み合わせが前回の値と変わっていない場合は、`useMemo` はこの関数の 2 回目の呼び出しをスキップし、単に前回返したのと同じ値を返します。

`useMemo` に渡した関数はレンダー中に実行されるということを覚えておいてください。レンダー中に通常やらないようなことをやらないようにしましょう。例えば副作用は `useMemo` ではなく `useEffect` の仕事です。

`useMemo` はパフォーマンス最適化のために使うものであり、意味上の保証があるものだと考えないでください。将来的に React は、例えば画面外のコンポーネント用のメモリを解放する、などの理由で、メモ化された値を「忘れる」ようにする可能性があります。`useMemo` なしでも動作するコードを書き、パフォーマンス最適化のために `useMemo` を加えるようにしましょう（値が絶対に再計算されなければいけないというような稀なケースでは、`ref` の [遅延初期化](#)を行うことができます）。

便利なことに、`useMemo` は子コンポーネントの計算量の高い再レンダーをスキップするのにも使えます：

```
function Parent({ a, b }) {
  // Only re-rendered if `a` changes:
  const child1 = useMemo(() => <Child1 a={a} />, [a]);
  // Only re-rendered if `b` changes:
  const child2 = useMemo(() => <Child2 b={b} />, [b]);
  return (
    <>
      {child1}
      {child2}
    </>
  )
}
```

フック呼び出しはループ内に配置できないため、このアプローチはループ内では動作しないことに注意してください。ただしリストのアイテムの部分を別のコンポーネントに抽出してその中で `useMemo` を呼び出すことは可能です。

## 計算量の大きいオブジェクトの作成を遅延する方法はありますか？

`useMemo` を使えば同じ値に依存している高価な計算の結果をメモ化することができます。しかしこれはあくまでヒントとして使われるものであり、計算が再実行されないということを保証しません。しかし時にはオブジェクトが一度しか作られないことを保証したい場合があります。

まずよくあるユースケースは `state` の初期値を作成することが高価な場合です。

```
function Table(props) {
  // △ createRows() is called on every render
  const [rows, setRows] = useState(createRows(props.count));
  // ...
}
```

次回以降のレンダーでは無視される初期 state を毎回作成しなおすことを防ぐため、`useState` に関数を渡すことができます。

```
function Table(props) {
  // ✅ createRows() is only called once
  const [rows, setRows] = useState(() => createRows(props.count));
  // ...
}
```

React は初回レンダー時のみこの関数を呼び出します。`useState` の API リファレンスを参照してください。

また、まれに `useRef()` の初期値を毎回再作成することを避けたいということもあります。例えば、命令型で作成するクラスのインスタンスが一度しか作成されないことを保証したいということがあるかもしれません。

```
function Image(props) {
  // △ IntersectionObserver is created on every render
  const ref = useRef(new IntersectionObserver(onIntersect));
  // ...
}
```

`useRef` は `useState` のような関数を渡す形式のオーバーロード記法が使えません。代わりに、自分で関数を書いて高価なオブジェクトを遅延型で `ref` に設定することができます。

```
function Image(props) {
  const ref = useRef(null);

  // ✅ IntersectionObserver is created lazily once
  function getObserver() {
    if (ref.current === null) {
      ref.current = new IntersectionObserver(onIntersect);
    }
    return ref.current;
  }

  // When you need it, call getObserver()
  // ...
}
```

これにより、本当に必要になるまで高価なオブジェクトの作成を避けることができます。Flow や TypeScript を使っているなら、`getObserver()` を non-nullable な型にしておくと便利でしょう。

## レンダー内で関数を作るせいでフックは遅くなるのではないですか？

いいえ。モダンブラウザでは、特殊な場合を除いて、クラスと比較してクロージャーの生の性能はそれほど変わりません。

しかも、フックの設計は幾つかの点においてはより効率的です。

- フックを使えば、クラスインスタンスの作成や、コンストラクタでのイベントハンドラのバインドといった、クラスの場合に必要な様々なオーバーヘッドを回避できます。
- フックをうまく用いたコードは、高階コンポーネントやレンダープロップやコンテクストを多用するコードベースで見られるような深いコンポーネントのネストを必要としません。コンポーネントツリーが小さければ、React がやるべき仕事を減ります。

過去には、インライン関数によるパフォーマンスの懸念というのは、レンダー毎に新しいコールバック関数を作つて渡すと子コンポーネントでの `shouldComponentUpdate` による最適化が動かなくなる、という問題と関連していました。フックではこの問題について 3 つの側面から対応します。

- `useCallback` フックを使えば再レンダーをまたいで同じコールバックを保持できるので、`shouldComponentUpdate` がうまく動作し続けます

```
// Will not change unless `a` or `b` changes
const memoizedCallback = useCallback(() => {
  doSomething(a, b);
}, [a, b]);
```

- `useMemo` フックを使うことで個々の子コンポーネントをいつ更新するのかを制御しやすくなるため、コンポーネントが純粋である必要性は低くなっています
- 最後に、以下で説明されているように、`useReducer` フックを使えば、複数のコールバックを深い階層に受け渡していく必要があまりなくなります

## どうすれば複数のコールバックを深く受け渡すのを回避できますか？

我々が見たところ、ほとんどの人はコンポーネントツリーの各階層で手作業でコールバックを受け渡していく作業が好きではありません。それはより明示的ではありますか、面倒な『配管工事』をしている気分になります。

大きなコンポーネントツリーにおいて我々がお勧めする代替手段は、`useReducer` で `dispatch` 関数を作つて、それをコンテクスト経由で下の階層に渡す、というものです。

```
const TodosDispatch = React.createContext(null);

function TodosApp() {
  // Note: `dispatch` won't change between re-renders
  const [todos, dispatch] = useReducer(todosReducer);

  return (
    <TodosDispatch.Provider value={dispatch}>
      <DeepTree todos={todos} />
    </TodosDispatch.Provider>
  );
}
```

`TodosApp` ツリーの中にいるあらゆる子コンポーネントはこの `dispatch` 関数を使うことができ、上位にいる `TodosApp` にアクションを伝えることができます。

```
function DeepChild(props) {
  // If we want to perform an action, we can get dispatch from context.
  const dispatch = useContext(TodosDispatch);

  function handleClick() {
    dispatch({ type: 'add', text: 'hello' });
  }

  return (
    <button onClick={handleClick}>Add todo</button>
  );
}
```

これは（複数のコールバックを何度も受け渡しする必要がないので）メンテナンスの観点から便利だ、というだけではなく、コールバックにまつわる問題をすべて回避できます。深い更新においてはこのように `dispatch` を渡すのがお勧めのパターンです。

アプリケーションの `state` については、`props` として渡していくか（より明示的）、あるいはコンテクスト経由で渡すか（深い更新ではより便利）を選ぶ余地が依然あります。もしもコンテクストを使って `state` も渡すことにする場合は、2つの別のコンテクストのタイプを使ってください -- `dispatch` のコンテクストは決して変わらないため、`dispatch` だけを使うコンポーネントは（アプリケーションの `state` も必要でない限り）再レンダーする必要がなくなります。

## useCallback からの頻繁に変わる値を読み出す方法は？

### 補足

我々は個別のコールバックを `props` として渡すのではなく、[コンテクスト経由で dispatch を渡すことを推奨](#)しています。以下のアプローチは網羅性と避難ハッシュの目的で掲載しているものです。

稀なケースですが、コールバックを `useCallback` でメモ化しているにも関わらず、内部関数を何度も再作成しないといけないためメモ化がうまく働かない、ということがあります。あなたがメモ化しようとしている関数がレンダー最中には使われないイベントハンドラなのであれば、[インスタンス変数としての ref を使って最後に使われた値を手動で保持しておく](#)ことができます。

```
function Form() {
  const [text, updateText] = useState('');
  const textRef = useRef();

  useEffect(() => {
    textRef.current = text; // Write it to the ref
  });

  const handleSubmit = useCallback(() => {
    const currentText = textRef.current; // Read it from the ref
    alert(currentText);
  }, [textRef]); // Don't recreate handleSubmit like [text] would do

  return (
    <>
      <input value={text} onChange={e => updateText(e.target.value)} />
      <ExpensiveTree onSubmit={handleSubmit} />
    </>
  );
}
```

これはやや複雑なパターンですが、このような避難ハッシュ的最適化は必要であれば可能だということです。カスタムフックに抽出すれば多少は読みやすくなります：

```
function Form() {
  const [text, updateText] = useState('');
  // Will be memoized even if `text` changes:
  const handleSubmit = useEventCallback(() => {
    alert(text);
  }, [text]);
```

```

    return (
      <>
        <input value={text} onChange={e => updateText(e.target.value)} />
        <ExpensiveTree onSubmit={handleSubmit} />
      </>
    );
}

function useEventCallback(fn, dependencies) {
  const ref = useRef(() => {
    throw new Error('Cannot call an event handler while rendering.');
  });

  useEffect(() => {
    ref.current = fn;
  }, [fn, ...dependencies]);

  return useCallback(() => {
    const fn = ref.current;
    return fn();
  }, [ref]);
}

```

いずれにせよ、このパターンは薦められず、網羅性のために示しているに過ぎません。代わりに [コールバックを深く受け渡していくことを回避する](#)のが望ましいパターンです。

## 内部の仕組み

### React はフック呼び出しとコンポーネントとをどのように関連付けているのですか？

React は現在どのコンポーネントがレンダー中なのかを把握しています。[フックのルール](#)のお陰で、フックは React のコンポーネント内（あるいはそれから呼び出されるカスタムフック内）でのみ呼び出されるということが分かっています。

それぞれのコンポーネントに関連付けられる形で、React 内に「メモリーセル」のリストが存在しています。それらは単に何らかのデータを保存できる JavaScript のオブジェクトです。あなたが `useState()` のようなフックを呼ぶと、フックは現在のセルの値を読み出し（あるいは初回レンダー時はセル内容を初期化し）、ポインタを次に進めます。これが複数の `useState()` の呼び出しが個別のローカル state を得る仕組みです。

### フックの先行技術にはどのようなものがありますか？

フックは複数の異なる出典からのアイディアを総合したものです：

- [react-future](#) リポジトリにおける関数型 API の古い実験。
- [Ryan Florence](#) の [Reactions Component](#) を含む、React コミュニティのレンダープロップ API に関する実験。
- [Dominic Gannaway](#) によって提案された、レンダープロップの糖衣構文としての `adopt keyword`。
- [DisplayScript](#) のステート変数とステートセル。
- [ReasonReact](#) の [Reducer components](#)。
- Rx の [Subscriptions](#)。
- Multicore OCaml の [Algebraic effects](#)。

フックは [Sebastian Markbåge](#) が最初のデザインを作り、[Andrew Clark](#)、[Sophie Alpert](#)、[Dominic Gannaway](#) およびその他の React チームのメンバーが洗練させました。

---

[Go to TOC](#)

# フックの導入

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。

```
import React, { useState } from 'react';

function Example() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

この `useState` という関数が、これから説明する最初のフック (Hook) です。これは単なるチラ見せですので、まだ意味が分からなくても問題ありません！

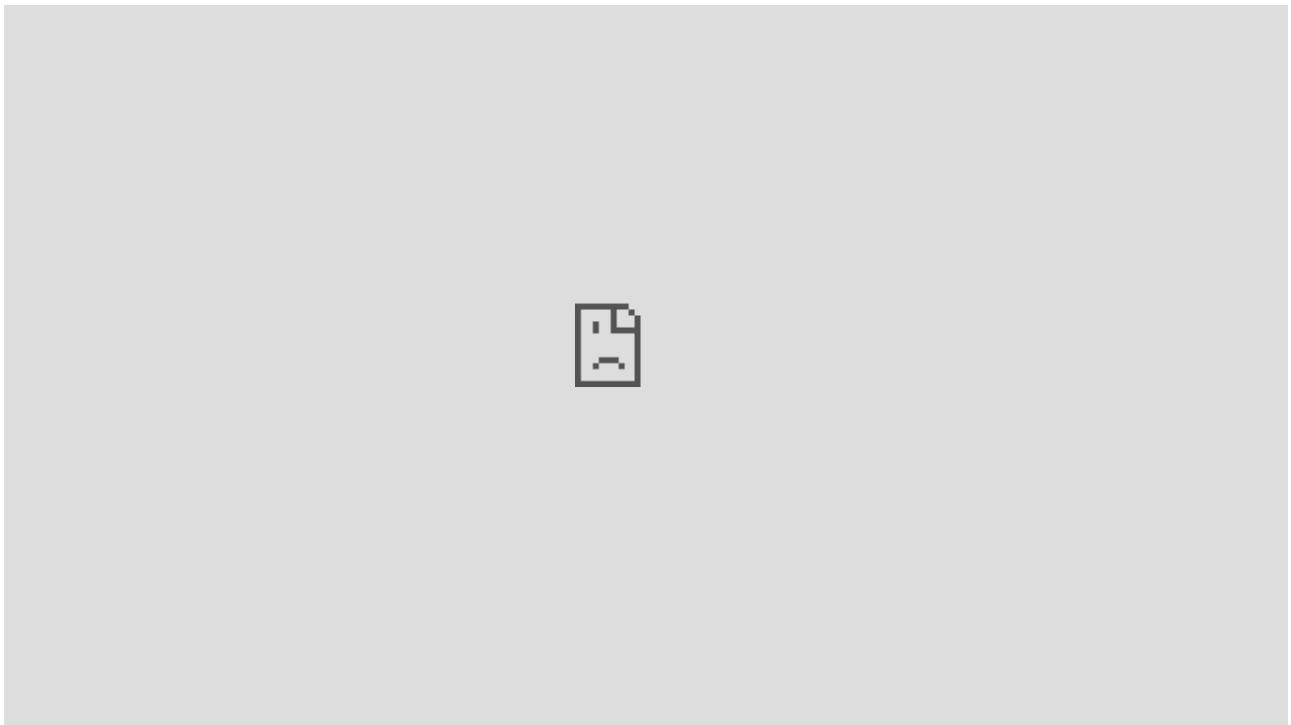
[次のページからフックについて学び始めることができます。](#) このページの残りの部分では、我々がなぜ React にフックの仕組みを加えることにしたのか、そして素晴らしいアプリケーションを作るためにどのように便利なのかについて説明していきます。

## 補足

React 16.8.0 がフックをサポートする最初のバージョンです。アップグレードの際は、React DOM を含むすべてのパッケージの更新を忘れないようにしてください。React Native は [React Native 0.59 リリース](#) 以降でフックをサポートします。

## ビデオによる紹介

React Conf 2018 にて Sophie Alpert と Dan Abramov がフックについての発表を行い、続いて Ryan Florence がアプリケーションでフックを使うようにリファクタリングする方法についてのデモを行いました。ビデオは以下で見ることができます：



## 互換性のない変更はありません

先に進む前に注意すべきこととして、フックは：

- 完全にオプトインです。既存のコードを書き換えずに一部のコンポーネントでフックを試すことができます。またやりたくないのであれば、今すぐに学んだり使ったりする必要もありません。
- 100% 後方互換です。フックには破壊的な変更は一切含まれていません。
- 今すぐ利用可能です。フックは v16.8.0 のリリースから利用可能です。

**React からクラス型コンポーネントを削除する予定はありません。**このページの[下部](#)で段階的にフックを採用していく方法について読むことができます。

フックは既にあなたが知っている React のコンセプトを置き換えるものではありません。むしろ、フックはあなたが既に知っている props、state、コンテクスト、ref、ライフサイクルといったコンセプトに対してより直接的な API を提供するものです。後でお見せするように、フックによって、これらを組み合わせるパワフルな手段も得ることができます。

とにかくフックを学び始めたいという方は、どうぞ[直接次のページに進んでください！](#) 以下には、なぜ React にフックを導入することにしたのか、アプリケーションを書き換えずにどのようにしてフックを使い始めるができるのかについて解説しています。

## 動機

フックによって、過去 5 年で何万というコンポーネントを作成・メンテナンスする中で我々が遭遇してきた、一見互いにあまり関係なさそうに見える様々な問題が解決されます。あなたが React を学習中の場合でも、毎日使っている場合でも、似たようなコンポーネントモデルを持つ別のライブラリが好きな場合でも、これらの問題の幾つかを認識しているかもしれません。

### ステートフルなロジックをコンポーネント間で再利用するのは難しい

React は再利用可能な振る舞いをコンポーネントに「付加する」方法（例えばストアオブジェクトを `connect` するなど）を提供していません。React をしばらく使った事があれば、この問題を解決するためのレンダープロップや高階コンポーネントといったパターンをご存じかもしれません。しかしこれらのパターンを使おうとするとコンポーネントの再構成が必要であり、面倒なうえにコードを追うのが難しくなります。典型的な React アプリを React DevTools で見てみると、おそらくプロバイダやらコンシューマやら高階コンポーネントやらレンダープロップやら、その他諸々の抽象化が多層に積み重なった『ラッパー地獄』を見ることになるでしょう。`DevTools` でそれらをフィルタして隠すことはできますが、この背景にはもっと根本的な問題があるということがわかります：React にはステートフルなロジックを共有するためのよりよい基本機能が必要なのです。

フックを使えば、ステートを持ったロジックをコンポーネントから抽出して、単独でテストしたり、また再利用したりすることができます。フックを使えば、ステートを持ったロジックを、コンポーネントの階層構造を変えることなしに再利用できるのです。このため、多数のコンポーネント間で、あるいはコミュニティ全体で、フックを共有することが簡単になります。

この点については[独自フックの作成](#)にてより詳しく述べています。

### 複雑なコンポーネントは理解しづらくなる

我々はよく、最初はシンプルだったのに、`state` を使うロジックや副作用によって管理不能なごちゃ混ぜ状態に陥ってしまったコンポーネントをメンテナンスさせられてきました。それぞれのライフサイクルメソッドには、しばしば互いに関係のないロジックが混在してしまいます。例えばあるコンポーネントは `componentDidMount` と `componentDidUpdate` で何かデータを取得しているかもしれません。しかし同じ `componentDidMount` 内には、イベントリスナーを登録する何か無関係なロジックがあるかもしれませんし、そのクリーンアップのコードは `componentWillUnmount` に書かれているかもしれません、といった具合です。一緒に更新されるべき互いに関連したコードがバラバラにされ、一方でまったく無関係なコードが 1 つのメソッド内に書かれています。このような状態は簡単にバグや非整合性を引き起こします。

多くの場合、`state` を使ったロジックはコンポーネント内のあらゆる場所にあるため、小さなコンポーネントに分割することは不可能です。テストも困難になります。これが、多くの人が単体の状態管理ライブラリの利用を好む理由のひとつでもあります。しかし、そのようなライブラリを利用するとしばしば過剰な抽象化を引き起こしたり、様々なファイルにジャンプさせられたり、コンポーネントの再利用がより困難になってしまったりします。

この問題を解決するため、関連する機能（例えばデータの購読や取得）をライフサイクルメソッドによって無理矢理分割する代わりに、**フックは関連する機能に基づいて、1 つのコンポーネントを複数の小さな関数に分割することを可能にします**。より `state` 管理を予測しやすくするため、必要に応じてリデューサ (reducer) を使って管理するようにしてもよいでしょう。

これについては[副作用フックの利用法](#)で詳しく述べます。

## クラスは人間と機械の両方を混乱させる

コードの再利用や整頓が難しくなるということに加えてクラスについて我々が学んだことは、クラスが React を学ぶ上で障壁となっているということです。JavaScript で `this` がどのように動作するのか理解しなければなりませんが、それは他の多くの言語での動作とは非常に異なっています。イベントハンドラを `bind` するよう覚えておく必要があります。[ES2022 のパブリッククラスフィールド構文](#)を使わない限り、コードは非常に冗長になってしまいます。開発者は `props` や `state` やトップダウンのデータフローについて完璧に理解できても、クラスの部分でつまづいてしまいます。React における関数コンポーネントとクラスコンポーネントの違いや使い分けについては経験のある React 開発者の間でも意見の差異が出てきます。

加えて、React は登場から約 5 年が経ちましたが、これからの 5 年間も使える選択肢のままであって欲しいと考えています。[Svelte](#) や [Angular](#) や [Glimmer](#) などのライブラリが示したように、コンポーネントの事前コンパイルには大きな将来性があります。特に使用法がテンプレートに限られていらない場合はそうです。最近我々は [Prepack](#) を使った [component folding](#) を試しており、有望な初期結果が得られています。しかし、クラスコンポーネントを使うことで、これらの最適化機能が遅い経路にフォールバックしてしまうようなパターンを助長してしまうことが分かりました。クラスは今まさに使われているツール群でも問題を引き起こします。例えば、クラスはあまりよく minify されませんし、ホットリローディングも不安定で信頼できないものになってしまいます。我々は、コードが最適化しやすい状態でいられる可能性を高くできるような API を提示したいのです。

これらの問題を解決するため、フックは、より多くの React の機能をクラスを使わずに利用できるようにします。コンセプト的には、React のコンポーネントは常に関数に近いものでした。フックは関数を活用しながらも、React の実用性を犠牲にしません。フックは命令型コードへの避難ハッチへのアクセスを提供しますし、複雑な関数型プログラミングやアクティブプログラミングの技法を学ばせることもありません。

### 例

[フック早わかり](#)はフックを学び始めるのに良い記事です。

## 段階的な採用戦略

**TLDR: React からクラスを削除する予定はありません。**

React 開発者はプロダクト開発に注力する必要があり、リリースされるあらゆる新しい API を確かめている時間はない、ということを、我々は理解しています。フックはとても新しい機能ですので、多くの例やチュートリアルが揃うまで、学んだり採用したりするのを待つ方がいいかもしれません。

また、React に新しい基本機能を付け加えるハードルが非常に高いということも理解しています。興味ある読者のために我々は[詳しい RFC](#) を用意しています。そこではより詳しく動機を掘り下げており、関連する先行技術や個別の設計上の選択についての概要が述べられています。

肝心なことですが、フックは既存のコードと併用することができるので、段階的に採用していくことが可能です。フックへの移行を急ぐ必要はありません。特に既存の複雑なコンポーネントについては、「大幅な書き換え」は避けることを推奨します。『フックで考えられる』ようになるには若干の思考の転換が必要です。我々の経験上は、あまり重要でない新しいコンポーネントでまずフックの使い方を練習し、チームの全員が慣れるようにすることが最良です。フックを試してみたら、どうぞお気軽に[フィードバックを送ってください](#)。ポジティブなものでもネガティブなものでも構いません。

クラスコンポーネントのユースケースをすべてフックがカバーできるようにする予定ではいますが、[クラスコンポーネントのサポートも予見可能な将来にわたって続けていきます](#)。Facebook では何万というコンポーネントがクラスとして書かれており、それらを書き換える予定は全くありません。代わりに、クラスと併用しながら新しいコードでフックを使っていく予定でいます。

## よくある質問

[Hook の FAQ ページ](#)では、フックに関するよくある質問にお答えしています。

## 次のステップ

このページを読み終えたことで、フックがどのような問題を解決しようとしているのか大まかに知ることはできたと思いますが、おそらく細かい部分についてはまだ分からぬと思います。心配は要りません。[次のページに進み、例を使ってフックについて学び始めましょう](#)。

---

[Go to TOC](#)

# フック早わかり

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[後方互換性](#)があります。このページでは React 経験者向けにフックの概要を述べていきます。このページはかなり端折った概要となっています。困ったときには以下のようない色のボックスを参照してください。

詳しくは

React にフックを導入する動機については[動機](#)を参照してください。

↑↑↑ それぞれの節の終わりに上のような黄色いボックスがあります。より詳しい説明へのリンクとなっています。

## 📌 ステートフック

この例ではカウンターを表示します。ボタンをクリックすると、カウンターの値が増えます：

```
import React, { useState } from 'react';

function Example() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

この例の `useState` が フック（この単語の意味するところはすぐ後で説明します）です。関数コンポーネントの中でローカルな `state` を使うために呼び出しています。この `state` は以降の再レンダーの間も React によって保持されます。`useState` は現在の `state` の値と、それを更新するための関数とをペアにして返します。この関数はイベントハンドラやその他の場所から呼び出すことができます。クラスコンポーネントにおける `this.setState` と似ていますが、新しい `state` が古いものとマージされないという違いがあります（`useState` と `this.state` の違いについては[ステートフックの利用法](#)で例を挙げて説明します）。

`useState` の唯一の引数は `state` の初期値です。上記の例では、カウンターがゼロからスタートするので `0` を渡しています。`this.state` と違い、`state` はオブジェクトである必要はないことに注意してください（オブジェクトにしたい場合はそうすることも可能です）。引数として渡された `state` の初期値は最初のレンダー時にのみ使用されます。

### 複数の `state` 変数の宣言

1 つのコンポーネント内で 2 回以上ステートフックを使うことができます：

```
function ExampleWithManyStates() {
  // Declare multiple state variables!
  const [age, setAge] = useState(42);
  const [fruit, setFruit] = useState('banana');
  const [todos, setTodos] = useState([{ text: 'Learn Hooks' }]);
  // ...
}
```

配列の分割代入構文を使うことで、`useState` を呼び出して宣言した state 変数に、異なる名前をつけることができます。これらの名前自体は `useState` の API の一部ではありません。その代わりに、React は `useState` を何度も呼び出す場合は、それらが全てのレンダー間で同じ順番で呼び出されるものと仮定します。この後で、どうしてこれが上手く動作し、どのような場合に便利なのが改めて説明します。

### 要するにフックとは？

フックとは、関数コンポーネントに state やライフサイクルといった React の機能を "接続する (hook into)" ための関数です。フックは React をクラスなしに使うための機能ですので、クラス内では機能しません。今すぐに既存のコンポーネントを書き換えることはお勧めしませんが、新しく書くコンポーネントで使いたければフックを利用し始めることができます。

React は `useState` のような幾つかのビルトインのフックを提供します。異なるコンポーネント間でステートフルな振る舞いを共有するために自分自身のフックを作成することもできます。まずは組み込みのフックから見ていきましょう。

詳しくは

ステートフックについてはこちらのページを参照してください：[ステートフックの利用法](#).

## ⚡ 副作用フック

これまでに React コンポーネントの内部から、外部データの取得や購読 (subscription)、あるいは手動での DOM 更新を行ったことがおりでしょう。これらの操作は他のコンポーネントに影響することがあり、またレンダーの最中に実行することができないので、われわれはこのような操作を "副作用 (side-effects)"、あるいは省略して "作用 (effects)" と呼んでいます。

`useEffect` は副作用のためのフックであり、関数コンポーネント内で副作用を実行することを可能にします。クラスコンポーネントにおける `componentDidMount`, `componentDidUpdate` および `componentWillUnmount` と同様の目的で使うものですが、1 つの API に統合されています（これらのメソッドと `useEffect` との違いについては[副作用フックの利用法](#)で例を挙げて説明します）。

例えば、このコンポーネントは React が DOM を更新した後で、HTML ドキュメントのタイトルを設定します。

```
import React, { useState, useEffect } from 'react';

function Example() {
  const [count, setCount] = useState(0);

  // Similar to componentDidMount and componentDidUpdate:
  useEffect(() => {
    // Update the document title using the browser API
    document.title = `You clicked ${count} times`;
  });
}
```

```

});;

return (
  <div>
    <p>You clicked {count} times</p>
    <button onClick={() => setCount(count + 1)}>
      Click me
    </button>
  </div>
);
}

```

`useEffect` を呼ぶことで、DOM への更新を反映した後にあなたが定義する「副作用関数」を実行するように React に指示します。副作用はコンポーネント内で宣言されるので、`props` や `state` にアクセスすることが可能です。デフォルトでは初回のレンダーも含む毎回のレンダー時にこの副作用関数が呼び出されます（クラスにおけるライフサイクルメソッドとの比較は[副作用フックの利用法](#)で詳しく述べます）。

副作用は自分を「クリーンアップ」するためのコードを、オプションとして関数を返すことで指定できます。例えば、以下のコンポーネントでは副作用を利用して、フレンドのオンラインステータスを購読し、またクリーンアップとして購読の解除を行っています。

```

import React, { useState, useEffect } from 'react';

function FriendStatus(props) {
  const [isOnline, setIsOnline] = useState(null);

  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  useEffect(() => {
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);

    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}

```

この例では、コンポーネントがアンマウントされる時や再レンダーによって副作用が再実行される時に `ChatAPI` の購読解除を行っています（必要なら、`ChatAPI` に渡すための `props.friend.id` が変わっていない場合には[毎回購読しなおす処理をスキップする](#)方法があります）。

`useState` の場合と同様、1 つのコンポーネント内で 2 つ以上の副作用を使用することができます。

```

function FriendStatusWithCounter(props) {
  const [count, setCount] = useState(0);
  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });

  const [isOnline, setIsOnline] = useState(null);

```

```

useEffect(() => {
  ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
  return () => {
    ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
  };
});

function handleStatusChange(status) {
  setIsOnline(status.isOnline);
}
// ...

```

フックを利用することで、ライフサイクルメソッドの場合は分離して書かざるを得なかったコンポーネント内の副作用を、関連する部分（リソースの購読とその解除、など）同士で整理して記載することが可能になります。

詳しくは

`useEffect` についての更なる詳細は[副作用フックの利用法](#)を参照してください。

## 👉 フックのルール

フックは JavaScript の関数ですが、2 つの追加のルールがあります。

- フックは関数のトップレベルのみで呼び出してください。ループや条件分岐やネストした関数の中でフックを呼び出さないでください。
- フックは React の関数コンポーネントの内部のみで呼び出してください。通常の JavaScript 関数内では呼び出さないでください（ただしフックを呼び出していい場所がもう 1 力所だけあります — 自分のカスタムフックの中です。これについてはすぐ後で学びます）。

これらのルールを自動的に強制するための [linter plugin](#) を用意しています。これらのルールは最初は混乱しやすかったり制限のように思えたりするかもしれません、フックがうまく動くためには重要なものです。

詳しくは

これらのルールについての詳細は[フックのルール](#)を参照してください。

## 💡 独自フックの作成

`state` を用いたロジックをコンポーネント間で再利用したいことがあります。これまで、このような問題に対して 2 種類の人気の解決方法がありました。[高階コンポーネント](#) と [レンダープロップ](#) です。カスタムフックを利用してすることで、同様のことが、ツリー内のコンポーネントを増やすことなく行えるようになります。

このページの上側で、`useState` と `useEffect` の両方のフックを呼び出してフレンドのオンライン状態を購読する `FriendStatus` というコンポーネントを紹介しました。この購読ロジックを別のコンポーネントでも使いたくなったとしましょう。

まず、このロジックを `useFriendStatus` というカスタムフックへと抽出します。

```
import React, { useState, useEffect } from 'react';

function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);

  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  useEffect(() => {
    ChatAPI.subscribeToFriendStatus(friendID, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(friendID, handleStatusChange);
    };
  });

  return isOnline;
}
```

これは `friendID` を引数として受け取り、フレンドがオンラインかどうかを返します。

これで両方のコンポーネントからこのロジックが使えます：

```
function FriendStatus(props) {
  const isOnline = useFriendStatus(props.friend.id);

  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}

function FriendListItems(props) {
  const isOnline = useFriendStatus(props.friend.id);

  return (
    <li style={{ color: isOnline ? 'green' : 'black' }}>
      {props.friend.name}
    </li>
  );
}
```

それぞれのコンポーネントの `state` は完全に独立しています。フックは `state` を用いたロジックを再利用するものであって、`state` そのものを再利用するものではありません。実のところ、フックのそれぞれの呼び出しが独立した `state` を持っていますので、全く同じカスタムフックを 1 つのコンポーネント内で 2 回呼び出すことも可能です。

カスタムフックは、機能というよりはむしろ慣習のようなものです。関数の名前が "use" から始まって、その関数が他のフックを呼び出しているなら、それをカスタムフックと言うことにする、ということです。この `useSomething` という命名規約によって、我々の linter プラグインはフックを利用しているコードのバグを見つけることができます。

カスタムフックは、フォーム管理や、アニメーションや、宣言的なデータソースの購読や、タイマーや、あるいはおそらく我々が考えたこともない様々なユースケースに利用可能です。React のコミュニティがどのようなカスタムフックを思いつくのか楽しみにしています。

詳しくは

カスタムフックについての詳しい情報は[独自フックの作成](#)を参照してください。

## ⚡ その他のフック

その他にもいくつか、使用頻度は低いものの便利なフックが存在しています。例えば、`useContext` を使えば React のコンテキストをコンポーネントのネストなしに利用できるようになります：

```
function Example() {
  const locale = useContext(LocaleContext);
  const theme = useContext(ThemeContext);
  // ...
}
```

また `useReducer` を使えば複雑なコンポーネントのローカル state をリデューサ (reducer) を用いて管理できるようになります：

```
function Todos() {
  const [todos, dispatch] = useReducer(todosReducer);
  // ...
}
```

詳しくは

全てのビルトインフックについての詳細は[フック API リファレンス](#)を参照してください。

## 次のステップ

かなり駆け足の説明でしたね！ まだよく分からなかったり、より詳しく学びたいと思った場合は、[ステートフック](#)から始まるこの先のページに進んでください。

[フック API リファレンス](#)と[よくある質問](#)も参照してください。

最後になりましたが、[フックの紹介ページ](#)もお忘れなく！ そこではなぜ我々がフックを導入することにしたのか、またアプリ全体を書き換えずにクラスと併用してどのようにフックを使っていくのか、について説明しています。

---

[Go to TOC](#)

# フック API リファレンス

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。

このページでは React 組み込みのフックについて説明します。

フックが初めての方は、先に[概要ページ](#)を確認してください。[よくある質問](#)にも有用な情報が掲載されています。

- [基本のフック](#)
  - `useState`
  - `useEffect`
  - `useContext`
- [追加のフック](#)
  - `useReducer`
  - `useCallback`
  - `useMemo`
  - `useRef`
  - `useImperativeHandle`
  - `useLayoutEffect`
  - `useDebugValue`
  - `useDeferredValue`
  - `useTransition`
  - `useId`
- [ライブラリ製作者用フック](#)
  - `useSyncExternalStore`
  - `useInsertionEffect`

## 基本のフック

### `useState` [\(#usestate\)](#)

```
const [state, setState] = useState(initialState);
```

ステートフルな値と、それを更新するための関数を返します。

初回のレンダー時に返される `state` は第 1 引数として渡された値 (`initialState`) と等しくなります。

`setState` 関数は `state` を更新するために使用します。新しい `state` の値を受け取り、コンポーネントの再レンダーをスケジューリングします。

```
setState(newState);
```

後続の再レンダー時には、`useState` から返される 1 番目の値は常に、更新を適用した後の最新版の `state` になります。

## 補足

React は再レンダー間で `setState` 関数の同一性が保たれ、変化しないことを保証します。従って `useEffect` や `useCallback` の依存リストにはこの関数を含めないでも構いません。

## 関数型の更新

新しい `state` が前の `state` に基づいて計算される場合は、`setState` に関数を渡すことができます。この関数は前回の `state` の値を受け取り、更新された値を返します。以下は、`setState` の両方の形式を用いたカウントコンポーネントの例です。

```
function Counter({initialCount}) {
  const [count, setCount] = useState(initialCount);
  return (
    <>
      Count: {count}
      <button onClick={() => setCount(initialCount)}>Reset</button>
      <button onClick={() => setCount(prevCount => prevCount - 1)}>-</button>
      <button onClick={() => setCount(prevCount => prevCount + 1)}>+</button>
    </>
  );
}
```

"+" と "-" のボタンは、更新後の値が更新前の値に基づいて計算されるため、関数形式を使っています。"Reset" ボタンは常にカウントを初期値に戻すので、通常の形式を使っています。

この更新用関数が現在の `state` と全く同じ値を返す場合は、後続する再レンダーは完全にスキップされます。

## 補足

クラスコンポーネントの `setState` メソッドとは異なり、`useState` は自動的な更新オブジェクトのマージを行いません。この動作は関数型の更新形式をスプレッド構文と併用することで再現可能です：

```
const [state, setState] = useState({});
setState(prevState => {
  // Object.assign would also work
  return {...prevState, ...updatedValues};
});
```

別の選択肢としては `useReducer` があり、これは複数階層の値を含んだ `state` オブジェクトを管理する場合にはより適しています。

## state の遅延初期化

`initialState` 引数は初回レンダー時に使われる `state` 値です。後続のレンダー時にはその値は無視されます。もし初期 `state` が高価な計算をして求める値である場合は、代わりに関数を渡すことができます。この関数は初回のレンダー時にのみ実行されます。

```
const [state, setState] = useState(() => {
  const initialState = someExpensiveComputation(props);
  return initialState;
});
```

## state 更新の回避

現在値と同じ値で更新を行った場合、React は子のレンダーや副作用の実行を回避して処理を終了します（React は `Object.is` による比較アルゴリズムを使用します）。

更新の回避が起きる前に React により該当のコンポーネント自体はレンダーされるかもしれない、ということに注意してください。ツリーのそれ以上「深く」にまで処理は及ばないためこれは問題ではないはずです。もしレンダー中にコストの高い計算を行っている場合は `useMemo` を使った最適化が可能です。

## state 更新のバッチ処理

React はパフォーマンス改善のため、複数の state 更新を 1 回の再レンダーにまとめることができます。通常、これによりパフォーマンスが改善しますが、あなたのアプリケーションの挙動には影響がないはずです。

React 18 より前では、React のイベントハンドラ内部で起きた更新のみがまとめて処理されていました。React 18 からは、[バッチ処理はすべての更新でデフォルトで有効になります](#)。ただし React はユーザが引き起こした複数の異なるイベント（例えばボタンの 2 回のクリック）に伴う更新をまとめることはできません。これにより論理的な誤りを防ぐことができます。

DOM への更新が同期的に適用されるよう強制しないといけないという稀な状況においては、`flushSync` で更新をラップすることができます。ただしこれはパフォーマンスに悪影響を及ぼしますので必要な場合にのみ利用するようにしてください。

## useEffect {#useeffect}

```
useEffect(didUpdate);
```

副作用を有する可能性のある命令型のコードを受け付けます。

DOM の書き換え、データの購読、タイマー、ロギング、あるいはその他の副作用を、関数コンポーネントの本体（React のレンダーフェーズ）で書くことはできません。それを行うと UI にまつわるややこしいバグや非整合性を引き起こします。

代わりに `useEffect` を使ってください。`useEffect` に渡された関数はレンダーの結果が画面に反映された後に動作します。副作用とは React の純粋に関数的な世界から命令型の世界への避難ハッチであると考えてください。

デフォルトでは副作用関数はレンダーが終了した後に毎回動作しますが、[特定の値が変化した時のみ](#)動作させるようにすることもできます。

## エフェクトのクリーンアップ

副作用はしばしば、コンポーネントが画面から消える場合にクリーンアップする必要があるようなリソース（例えば購読やタイマー ID など）を作成します。これを実現するために、`useEffect` に渡す関数はクリーンアップ用関数を返すことができます。例えば、データ購読を作成する場合は以下のようにになります。

```
useEffect(() => {
  const subscription = props.source.subscribe();
  return () => {
    // Clean up the subscription
    subscription.unsubscribe();
  };
});
```

メモリリークを防止するため、コンポーネントが UI から削除される前にクリーンアップ関数が呼び出されます。それに加えて、コンポーネントが複数回レンダーされる場合（大抵はそうですが）、**新しい副作用を実行する前に前回の副作用はクリーンアップされます**。この例では、更新が発生する度に新しい購読が作成される、ということです。毎回の更新で副作用が実行されるのを抑制するためには、後の節をご覧ください。

## 副作用のタイミング

`componentDidMount` や `componentDidUpdate` と異なり、`useEffect` に渡された関数はレイアウトと描画の後で遅延されたイベントとして実行されます。ほとんどの作業はブラウザによる画面への描画をブロックするべきではないため、購読やイベントハンドラのセットアップといったよくある副作用のほとんどにとって、この動作は適切です。

しかしすべての副作用が遅延できるわけではありません。例えばユーザーに見えるような DOM の改変は、ユーザーが見た目の不整合性を感じずに済むよう、次の描画が発生する前に同期的に発生する必要があります（この違いは概念的には受動的なイベントリスナーと能動的なイベントリスナーの違いに似ています）。このようなタイプの副作用のため、React は `useLayoutEffect` という別のフックを提供しています。これは `useEffect` と同じシグネチャを持っており、実行されるタイミングのみが異なります。

加えて、React 18 以降、`useEffect` に渡された関数は、クリックのような個々のユーザー入力の結果としてレイアウト・描画が起こる場合や、`flushSync` でラップされた更新の結果としてレイアウト・描画が起こる場合には、そのようなレイアウト・描画の前に同期的に呼び出されるようになります。この動作により、副作用の結果をイベントシステムや `flushSync` の呼び出し元が確認できるようになります。

### 補足

これは `useEffect` に渡された関数が呼び出されるタイミングにのみ影響します。副作用内でスケジュールされた更新が遅延されることに変わりはありません。`useLayoutEffect` はこれとは異なり、関数を呼び出したあと内部での更新を即座に処理します。

ブラウザが描画を終えるまで `useEffect` が遅延される場合でも、次のレンダーが起こるより前に実行されることは保証されています。React は新しい更新を始める前に常にひとつ前のレンダーの副作用をクリーンアップします。

## 条件付きで副作用を実行する

デフォルトの動作では、副作用関数はレンダーの完了時に毎回実行されます。これにより、コンポーネントの依存配列のうちのいずれかが変化した場合に毎回副作用が再作成されます。

しかし、上述のデータ購読の例でもそうですが、これは幾つかのケースではやりすぎです。新しい購読を設定する必要があるのは毎回の更新ごとではなく、`source` プロパティが変化した場合のみです。

これを実装するためには、`useEffect` の第 2 引数として、この副作用が依存している値の配列を渡します。変更後の例は以下のようになります。

```
useEffect(
  () => {
    const subscription = props.source.subscribe();
    return () => {
      subscription.unsubscribe();
    };
  },
  [props.source],
);
```

これで、データの購読は `props.source` が変更された場合にのみ再作成されるようになります。

## 補足

この最適化を利用する場合、**時間の経過とともに変化し副作用によって利用される、コンポーネントスコープの値（`props` や `state` など）** がすべて配列に含まれていることを確認してください。さもないあなたのコードは以前のレンダー時の古い値を参照してしまうことになります。[関数の扱い方とこの配列の値が頻繁に変わる場合の対処法](#)も参照してください。

もしも副作用とそのクリーンアップを 1 度だけ（マウント時とアンマウント時にのみ）実行したいという場合、空の配列（`[]`）を第 2 引数として渡すことができます。こうすることで、あなたの副作用は `props` や `state` の値のいずれにも依存していないため再実行する必要が一切ない、ということを React に伝えることができます。これは特別なケースとして処理されているわけではなく、入力配列を普通に処理すればそうなるというだけの話です。

空の配列（`[]`）を渡した場合、副作用内では `props` と `state` の値は常にその初期値のままになります。`[]` を渡すことはおなじみの `componentDidMount` と `componentWillUnmount` による概念と似ているように感じるでしょうが、通常は[こちら](#)や[こちら](#)のように、副作用を過度に再実行しないためのよりよい解決方法があります。また `useEffect` はブラウザが描画し終えた後まで遅延されますので、追加の作業をしてもそれほど問題にならないこともあります。

`eslint-plugin-react-hooks` パッケージの `exhaustive-deps` ルールを有効にすることをお勧めします。これは依存の配列が正しく記述されていない場合に警告し、修正を提案します。

依存の配列は副作用関数に引数として渡されるわけではありません。しかし概念としては、この記法は副作用関数の引数が何なのかを表現しています。副作用関数の内部で参照されているすべての値は入力の配列内にも現れるべきです。将来的には、コンパイラが発達すればこの配列を自動で作成することも可能であるはずです。

## useContext {#usecontext}

```
const value = useContext(MyContext);
```

コンテキストオブジェクト（`React.createContext` からの戻り値）を受け取り、そのコンテキストの現在値を返します。コンテキストの現在値は、ツリー内でこのフックを呼んだコンポーネントの直近にある `<MyContext.Provider>` の `value` の値によって決定されます。

直近の `<MyContext.Provider>` が更新された場合、このフックはその `MyContext` プロバイダに渡された最新の `value` の値を使って再レンダーを発生させます。祖先コンポーネントが `React.memo` や `shouldComponentUpdate` を使っている場合でも、`useContext` を使っているコンポーネント自体から再レンダーが発生します。

`useContext` に渡す引数は **コンテキストオブジェクト自体** であることを忘れないでください。

- **正しい:** `useContext(MyContext)`
- **間違い:** `useContext(MyContext.Consumer)`
- **間違い:** `useContext(MyContext.Provider)`

`useContext` を呼び出すコンポーネントはコンテキストの値が変化するたびに毎回再レンダーされます。再レンダーが高価である場合は [メモ化を使って最適化](#) が可能です。

### ヒント

フック以前のコンテキスト API に馴染みがある場合は、`useContext(MyContext)` はクラスにおける `static contextType = MyContext`、あるいは `<MyContext.Consumer>` と同等のものであると考えることができます。

`useContext(MyContext)` は現在のコンテキストの値を読み取り、その変化を購読することしかできません。コンテキストの値を設定するために、今後もツリーの上の階層で `<MyContext.Provider>` が必要です。

## Context.Provider と組み合わせて使用する方法

```
const themes = {
  light: {
    foreground: "#000000",
    background: "#eeeeee"
  },
  dark: {
    foreground: "#ffffff",
    background: "#222222"
  }
};

const ThemeContext = React.createContext(themes.light);

function App() {
  return (
    <ThemeContext.Provider value={themes.dark}>
      <Toolbar />
    </ThemeContext.Provider>
  );
}

function Toolbar(props) {
  return (
    <div>
      <ThemedButton />
    </div>
  );
}

function ThemedButton() {
  const theme = useContext(ThemeContext);
```

```

    return (
      <button style={{ background: theme.background, color: theme.foreground }}>
        I am styled by theme context!
      </button>
    );
}

```

この例は[コンテキストのガイド](#)をフック用に変更したものです。コンテキストをいつどのように使うべきかについては同記事を参照してください。

## 追加のフック

以下のフックは前節で説明した基本のフックの変種であったり、特定の稀なケースでのみ必要となったりするものです。最初から無理に学ぼうとしなくて構いません。

### `useReducer` [\(#usereducer\)](#)

```
const [state, dispatch] = useReducer(reducer, initialState);
```

`useState` の代替品です。`(state, action) => newState` という型のリデューサ (reducer) を受け取り、現在の `state` を `dispatch` メソッドと一緒に返します（もし Redux に馴染みがあれば、これがどう動作するのかはご存じでしょう）。

通常、`useReducer` が `useState` より好ましいのは、複数の値にまたがる複雑な `state` ロジックがある場合や、前の `state` に基づいて次の `state` を決める必要がある場合です。また、`useReducer` を使えば[コールバックの代わりに dispatch を下位コンポーネントに渡せる](#)ようになるため、複数階層にまたがって更新を発生させるようなコンポーネントではパフォーマンスの最適化にもなります。

以下は `useState` の部分で挙げたカウンタの例を、リデューサを使うよう書き換えたものです。

```

const initialState = {count: 0};

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return {count: state.count + 1};
    case 'decrement':
      return {count: state.count - 1};
    default:
      throw new Error();
  }
}

function Counter() {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <>
      Count: {state.count}
      <button onClick={() => dispatch({type: 'decrement'})}>-</button>
      <button onClick={() => dispatch({type: 'increment'})}>+</button>
    </>
  );
}

```

## 補足

`React` は再レンダー間で `dispatch` 関数の同一性が保たれ、変化しないことを保証します。従って `useEffect` や `useCallback` の依存リストにはこの関数を含めないでも構いません。

## 初期 state の指定

`useReducer` の初期化の方法には 2 種類あります。ユースケースによりどちらかを選択してください。最も単純な方法は第 2 引数として初期 state を渡すものです。

```
const [state, dispatch] = useReducer(
  reducer,
  {count: initialCount}
);
```

## 補足

`React` では、リデューサの引数で `state = initialState` のようにして初期値を示すという、`Redux` で普及した慣習を使用しません。初期値は `props` に依存している可能性があるため、フックの呼び出し部分で指定します。強いこだわりがある場合は `useReducer(reducer, undefined, reducer)` という呼び出し方で `Redux` の振る舞いを再現できますが、お勧めはしません。

## 遅延初期化

初期 state の作成を遅延させることもできます。そのためには `init` 関数を第 3 引数として渡してください。初期 state が `init(initialArg)` に設定されます。

これにより初期 state の計算をリデューサの外部に抽出することができます。これはアクションに応じて state をリセットしたい場合にも便利です。

```
function init(initialCount) {
  return {count: initialCount};
}

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return {count: state.count + 1};
    case 'decrement':
      return {count: state.count - 1};
    case 'reset':
      return init(action.payload);
    default:
      throw new Error();
  }
}

function Counter({initialCount}) {
  const [state, dispatch] = useReducer(reducer, initialCount, init);
  return (
    <>
  );
}
```

```

Count: {state.count}
<button
  onClick={() => dispatch({type: 'reset', payload: initialCount})}>
  Reset
</button>
<button onClick={() => dispatch({type: 'decrement'})}>-</button>
<button onClick={() => dispatch({type: 'increment'})}>+</button>
</>
);
}

```

## dispatch による更新の回避

このフックから state の現在値として同じ値を返した場合、React は子のレンダーや副作用の実行を回避して処理を終了します（React は `Object.is` による比較アルゴリズムを使用します）。

更新の回避が起きる前に React により該当のコンポーネント自体はレンダーされるかもしれない、ということに注意してください。ツリーのそれ以上「深く」にまで処理は及ばないためこれは問題ではないはずです。もしレンダー中にコストの高い計算を行っている場合は `useMemo` を使った最適化が可能です。

## useCallback {#usecallback}

```

const memoizedCallback = useCallback(
() => {
  doSomething(a, b);
},
[a, b],
);

```

`メモ化`されたコールバックを返します。

インラインのコールバックとそれが依存している値の配列を渡してください。`useCallback` はそのコールバックをメモ化したものを持ち戻し、その関数は依存配列の要素のいずれかが変化した場合にのみ変化します。これは、不必要的レンダーを避けるために（例えば `shouldComponentUpdate` などを使って）参照の同一性を見るよう最適化されたコンポーネントにコールバックを渡す場合に便利です。

`useCallback(fn, deps)` は `useMemo(() => fn, deps)` と等価です。

### 補足

依存する値の配列はコールバックに引数として渡されるわけではありません。しかし概念としては、この記法はコールバックの引数が何なのかを表現しています。コールバックの内部で参照されているすべての値は依存の配列内にも現れるべきです。将来的には、コンパイラが発達すればこの配列を自動で作成することも可能であるはずです。

`eslint-plugin-react-hooks` パッケージの `exhaustive-deps` ルールを有効にすることをお勧めします。これは依存の配列が正しく記述されていない場合に警告し、修正を提案します。

## useMemo {#usememo}

```

const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);

```

**メモ化**された値を返します。

"作成用" 関数とそれが依存する値の配列を渡してください。`useMemo` は依存配列の要素のいずれかが変化した場合にのみメモ化された値を再計算します。この最適化によりレンダー毎に高価な計算が実行されるのを避ることができます。

`useMemo` に渡した関数はレンダー中に実行されるということを覚えておいてください。レンダー中に通常やらないようなことをこの関数内でやらないようにしましょう。例えば副作用は `useMemo` ではなく `useEffect` の仕事です。

配列が渡されなかった場合は、新しい値がレンダーごとに毎回計算されます。

`useMemo` はパフォーマンス最適化のために使うものであり、意味上の保証があるものだと考えないでください。将来的に React は、例えば画面外のコンポーネント用のメモリを解放するため、などの理由で、メモ化された値を「忘れる」ようになる可能性があります。`useMemo` なしでも動作するコードを書き、パフォーマンス最適化のために `useMemo` を加えるようにしましょう。

### 補足

依存する値の配列は第 1 引数の関数に引数として渡されるわけではありません。しかし概念としては、この記法は関数の引数が何なのかを表現しています。関数の内部で参照されているすべての値は依存の配列内にも現れるべきです。将来的には、コンパイラが発達すればこの配列を自動で作成することも可能であるはずです。

`eslint-plugin-react-hooks` パッケージの `exhaustive-deps` ルールを有効にすることをお勧めします。これは依存の配列が正しく記述されていない場合に警告し、修正を提案します。

## useRef {#useref}

```
const refContainer = useRef(initialValue);
```

`useRef` は、`.current` プロパティが渡された引数 (`initialValue`) に初期化されているミュータブルな `ref` オブジェクトを返します。返されるオブジェクトはコンポーネントの存在期間全体にわたって存在し続けます。

よくあるユースケースは、子コンポーネントに命令型でアクセスするというものです：

```
function TextInputWithFocusButton() {
  const inputEl = useRef(null);
  const onButtonClick = () => {
    // `current` points to the mounted text input element
    inputEl.current.focus();
  };
  return (
    <>
      <input ref={inputEl} type="text" />
      <button onClick={onButtonClick}>Focus the input</button>
    </>
  );
}
```

本質的に `useRef` とは、書き換え可能な値を `.current` プロパティ内に保持することができる「箱」のようなものです。

まずは `ref` のことを DOM にアクセスする手段として理解しているかもしれません。`<div ref={myRef} />` のようにして React に `ref` オブジェクトを渡した場合、React は DOM ノードに変更があるたびに `.current` プロパティをその DOM ノードに設定します。

しかしながら `useRef()` は `ref` 属性で使うだけではなく、より便利に使えます。これはクラスでインスタンス変数を使うのと同様にして、あらゆる書き換え可能な値を保持しておくのに便利です。

これは `useRef()` がプレーンな JavaScript オブジェクトを作成するからです。`useRef()` を使うことと自分で `{current: ...}` というオブジェクトを作成することとの唯一の違いとは、`useRef` は毎回のレンダーで同じ `ref` オブジェクトを返す、ということです。

`useRef` は中身が変更になってもそのことを通知しないということを覚えておいてください。`.current` プロパティを書き換えても再レンダーは発生しません。DOM ノードを `ref` に割り当てたり割り当てを解除したりする際に何らかのコードを走らせたいという場合は、`コールバック ref` を代わりに使用してください。

## useImperativeHandle {#useimperativehandle}

`useImperativeHandle(ref, createHandle, [deps])`

`useImperativeHandle` は `ref` が使われた時に親コンポーネントに渡されるインスタンス値をカスタマイズするのに使います。いつもの話ですが、`ref` を使った手続き的なコードはほとんどの場合に避けるべきです。`useImperativeHandle` は `forwardRef` と組み合わせて使います：

```
function FancyInput(props, ref) {
  const inputRef = useRef();
  useImperativeHandle(ref, () => ({
    focus: () => {
      inputRef.current.focus();
    }
  }));
  return <input ref={inputRef} ... />;
}
FancyInput = forwardRef(FancyInput);
```

この例では、`<FancyInput ref={inputRef} />` をレンダーする親コンポーネントは `inputRef.current.focus()` を呼べるようになります。

## useLayoutEffect {#uselayouтеffect}

この関数のシグネチャは `useEffect` と同一ですが、DOM の変更があった後で同期的に副作用が呼び出されます。これは DOM からレイアウトを読み出して同期的に再描画を行う場合に使ってください。`useLayoutEffect` の内部でスケジュールされた更新はブラウザによって描画される前のタイミングで同期的に処理されます。

可能な場合は画面の更新がブロックするのを避けるため、標準の `useEffect` を優先して使ってください。

### ヒント

クラスコンポーネントからコードを移行している場合、`useLayoutEffect` は `componentDidMount` や `componentDidUpdate` と同じフェーズで実行されるということに注意してください。しかしこれでも問題が発生する場合にのみ `useLayoutEffect` を試すことをお勧めします。

サーバレンダリングを使用している場合は、`useLayoutEffect` と `useEffect` のどちらも JavaScript がダウンロードされるまでは実行できないということを覚えておいてください。サーバでレンダーされるコンポーネントに `useLayoutEffect` が含まれている場合に React が警告を発生するのは、これが理由です。これを修正するには、そのロジックを `useEffect` に移動する（初回レンダーで必要がないロジックである場合）か、クライアントでレンダーされるまでコンポーネントの表示を遅らせる（`useLayoutEffect` が実行されるまで該当 HTML が正しく表示できない場合）ようにしてください。

サーバでレンダーされる HTML からレイアウト副作用を必要とするコンポーネントを除外したい場合は、それを `shouldChild && <Child />` のようにして条件付きでレンダーするようにして、表示を `useEffect(() => { setShowChild(true); }, [])` のようにして遅延させてください。これにより、JavaScript コードが注入される前に壊れた見た目の UI が表示されないようになります。

## useDebugValue {#usedebugvalue}

`useDebugValue(value)`

`useDebugValue` を使って React DevTools でカスタムフックのラベルを表示することができます。

例えば "独自フックの作成" で説明した `useFriendStatus` を例にします：

```
function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);

  // ...

  // Show a label in DevTools next to this Hook
  // e.g. "FriendStatus: Online"
  useDebugValue(isOnline ? 'Online' : 'Offline');

  return isOnline;
}
```

### ヒント

すべてのカスタムフックにデバッグ用の値を加えるのはお勧めしません。これが最も有用なのは共有ライブラリ内のカスタムフックです。

### デバッグ用の値のフォーマットを遅延させる

値を表示用にフォーマットすることが高価な処理である場合があります。また、フックが実際に DevTools でインスペクトされない場合はフォーマット自体が不要な処理です。

このため `useDebugValue` はオプションの第 2 引数としてフォーマット用関数を受け付けます。この関数はフックがインスペクトされている場合にのみ呼び出されます。この関数はデバッグ値を引数として受け取り、フォーマット済みの表示用の値を返します。

例えば `Date` 型の値を返すカスタムフックでは、以下のようなフォーマッタ関数を渡すことで、不必要に `toDateString` を呼び出すのを避けることができます。

```
useDebugValue(date, date => date.toDateString());
```

## useDeferredValue {#usedeferredvalue}

```
const deferredValue = useDeferredValue(value);
```

`useDeferredValue` は値を受け取りその値のコピーを返しますが、返り値はより緊急性の高い更新がある場合に遅延されるようになっています。現在のレンダーがユーザ入力のような緊急性の高い更新である場合には、React は前回と同じ値を返し、新しい値でのレンダーは緊急性の高いレンダーが完了した後に行うようにします。

このフックはデバウンス (debounce) やスロットル (throttle) を使って更新を遅延するためにユーザ側で作成されてきたフックと似ています。`useDeferredValue` を使う利点は、(常に何らかの固定の時間待つではなく) 他の作業が終わった時点ですぐに React が更新を処理できるという点と、`startTransition` と同様に値を遅延させることで既存のコンテンツがふいにフォールバックに隠されてしまわないよう待機できるという点です。

### 遅延可能な子コンポーネントのメモ化

`useDeferredValue` が遅延させるのは渡された値のみです。緊急性の高い更新の最終に子コンポーネントが再レンダーされるのを防ぎたい場合、そのコンポーネントで `React.memo` か `React.useMemo` を使ってメモ化を行う必要があります：

```
function Typeahead() {
  const query = useSearchQuery('');
  const deferredQuery = useDeferredValue(query);

  // Memoizing tells React to only re-render when deferredQuery changes,
  // not when query changes.
  const suggestions = useMemo(() =>
    <SearchSuggestions query={deferredQuery} />,
    [deferredQuery]
  );

  return (
    <>
      <SearchInput query={query} />
      <Suspense fallback="Loading results...">
        {suggestions}
      </Suspense>
    </>
  );
}
```

子をメモ化することで、`query` が変わった場合ではなく `deferredQuery` が変わった場合にのみ再レンダーすると React に伝えることができます。この注意事項は `useDeferredValue` に特有のものではなく、デバウンスやスロットルを使う似たようなフックで使われるものと同じパターンです。

## useTransition {#usetransition}

```
const [isPending, startTransition] = useTransition();
```

トランジションの実行中状態を表す状態値と、トランジションを開始するための関数を返します。

`startTransition` に渡されたコールバック内の更新はトランジションとしてマークされます：

```
startTransition(() => {
  setCount(count + 1);
})
```

`isPending` はトランジションがアクティブかどうかを表しており、ユーザに保留中状態を表示するのに使えます：

```
function App() {
  const [isPending, startTransition] = useTransition();
  const [count, setCount] = useState(0);

  function handleClick() {
    startTransition(() => {
      setCount(c => c + 1);
    })
  }

  return (
    <div>
      {isPending && <Spinner />}
      <button onClick={handleClick}>{count}</button>
    </div>
  );
}
```

補足：

トランジション内での更新はクリックのような緊急性の高い更新がある場合は遅延されることがあります。

トランジション内での更新によってコンテンツが再サスペンドした場合でもフォールバックは表示されません。これにより更新後のデータをレンダーしている最中に、ユーザが現在のコンテンツを操作しつづけられるようになります。

## useId {#useId}

```
const id = useId();
```

`useId` はハイドレーション時の不整合を防ぎつつサーバとクライアント間で安定な一意 ID を作成するためのフックです。

補足

`useId` はリスト内の `key` を作成するのに使うためのものではありません。`key` はあなたのデータから作成されるべきです。

基本的な例として、`id` をそれを必要としている要素に直接渡せます：

```
function Checkbox() {
  const id = useId();
  return (
    <>
    <label htmlFor={id}>Do you like React?</label>
    <input id={id} type="checkbox" name="react"/>
```

```

    </>
);
}

```

同じコンポーネントで複数の ID を使う場合は、同じ `id` に接尾辞を付けてください：

```

function NameFields() {
  const id = userId();
  return (
    <div>
      <label htmlFor={id + '-firstName'}>First Name</label>
      <div>
        <input id={id + '-firstName'} type="text" />
      </div>
      <label htmlFor={id + '-lastName'}>Last Name</label>
      <div>
        <input id={id + '-lastName'} type="text" />
      </div>
    </div>
  );
}

```

補足：

`userId` は : というトークンを含む文字列を生成します。これによりトークンが一意であることが保証しやすくなっていますが、CSS セレクタや `querySelectorAll` のような API では使用できません。

`userId` は複数のルートがあるアプリで衝突が起きないよう、`identifierPrefix` をサポートしています。設定する場合は `hydrateRoot` と `ReactDOMServer` のドキュメントを参照してください。

## ライブラリ用フック

以下のフックは React モデルと深く結合するライブラリの製作者向けであり、アプリケーションコードでは通常使われません。

### useSyncExternalStore {#usesyncexternalstore}

```

const state = useSyncExternalStore(subscribe, getSnapshot[, getServerSnapshot]);

```

`useSyncExternalStore` は、選択的ハイドレーションやタイムスライスなどの並行レンダリング機能と互換性を持つつ外部データソースから読み出しやデータの購読を行うために推奨されるフックです。

このメソッドは 3 つの引数を受け取り、ストアの値を返します：

- `subscribe`：ストアに変更があった場合に呼び出されるコールバックを登録するための関数。
- `getSnapshot`：現在のストアの値を返す関数。
- `getServerSnapshot`：サーバレンダリング時にスナップショットを返すための関数。

最も基本的な例では、ストア全体を単純に購読します：

```

const state = useSyncExternalStore(store.subscribe, store.getSnapshot);

```

ただし特定のフィールドを購読するようにもできます：

```
const selectedField = useSyncExternalStore(
  store.subscribe,
  () => store.getSnapshot().selectedField,
);
```

サーバレンダリングでは、サーバで使われるストアの値をシリアル化して `useSyncExternalStore` に渡す必要があります。React はハイドレーション中にこのスナップショットを利用してサーバとのミスマッチを防止します：

```
const selectedField = useSyncExternalStore(
  store.subscribe,
  () => store.getSnapshot().selectedField,
  () => INITIAL_SERVER_SNAPSHOT.selectedField,
);
```

補足：

`getSnapshot` はキャッシュされた値を返す必要があります。もし `getSnapshot` が連続して呼ばれる場合、その間にストアの更新がないのであれば、全く同一の値を返さなければなりません。

複数の React バージョンをサポートするための互換ライブラリが `use-sync-external-store/shim` として提供されています。このライブラリは `useSyncExternalStore` が存在する場合はそれを優先して利用し、ない場合はユーザースペースでの実装にフォールバックします。

便宜のため、`getSnapshot` の結果に対する自動的なメモ化をサポートしたバージョンの API を `use-sync-external-store/with-selector` として公開しています。

## useInsertionEffect {#useinsertioneffect}

```
useInsertionEffect(didUpdate);
```

シグネチャは `useEffect` と同一ですが、すべての DOM 更新の前に同期的に呼び出されます。`useLayoutEffect` でレイアウトを読み出す前に DOM にスタイルを注入するために利用してください。このフックの利用目的は限られているため、`ref` にアクセスしたり更新をスケジュールしたりすることはできません。

補足：

`useInsertionEffect` は `css-in-js` ライブラリの作者以外が使うべきではありません。`useEffect` や `useLayoutEffect` を通常は使ってください。

# フックのルール

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。

フックは JavaScript の関数ですが、それらを使う際には以下の 2 つのルールに従う必要があります。我々は自動的にこのルールを強制するための [linter プラグイン](#) を提供しています。

## フックを呼び出すのはトップレベルのみ

フックをループや条件分岐、あるいはネストされた関数内で呼び出してはいけません。代わりに、あなたの React の関数のトップレベルでのみ、あらゆる早期 return 文よりも前の場所で呼び出してください。これを守ることで、コンポーネントがレンダーされる際に毎回同じ順番で呼び出されるということが保証されます。これが、複数回 `useState` や `useEffect` が呼び出された場合でも React がフックの状態を正しく保持するための仕組みです（興味がある場合は[ページ下部](#)で詳しく説明しています）。

## フックを呼び出すのは React の関数内のみ

フックを通常の JavaScript 関数から呼び出さないでください。代わりに以下のようにします。

- React の関数コンポーネント内から呼び出す。
- カスタムフック内 ([次のページで説明します](#)) から呼び出す。

このルールを守ることで、コンポーネント内のすべての state を使うロジックがソースコードから間違いなく参照可能になります。

## ESLint プラグイン

これらの 2 つのルールを強制できる `eslint-plugin-react-hooks` という ESLint のプラグインをリリースしました。試したい場合はあなたのプロジェクトに以下のようにして加えることができます。

このプラグインは [Create React App](#) ではデフォルトで含まれています。

```
npm install eslint-plugin-react-hooks --save-dev
```

```
// Your ESLint configuration
{
  "plugins": [
    // ...
    "react-hooks"
  ],
  "rules": {
    // ...
    "react-hooks/rules-of-hooks": "error", // Checks rules of Hooks
    "react-hooks/exhaustive-deps": "warn" // Checks effect dependencies
  }
}
```

次のページまで飛ばして **独自のフック** を書く方法について学んでも構いません。このページの続きの部分ではこれらのルールの背後にある根拠について述べていきます。

## 解説

既に学んだ通り、ひとつのコンポーネント内で複数の state や副作用を使うことができます。

```
function Form() {
  // 1. Use the name state variable
  const [name, setName] = useState('Mary');

  // 2. Use an effect for persisting the form
  useEffect(function persistForm() {
    localStorage.setItem('formData', name);
  });

  // 3. Use the surname state variable
  const [surname, setSurname] = useState('Poppins');

  // 4. Use an effect for updating the title
  useEffect(function updateTitle() {
    document.title = name + ' ' + surname;
  });

  // ...
}
```

では React は、どの `useState` の呼び出しがどの state に対応するのか、どうやって知るのでしょうか？ その答えは **「React はフックが呼ばれる順番に依存している」** です。我々の例が動作するのは、フックの呼び出しの順序が毎回のレンダーごとに同じだからです。

```
// -----
// First render
// -----
useState('Mary')          // 1. Initialize the name state variable with 'Mary'
useEffect(persistForm)    // 2. Add an effect for persisting the form
useState('Poppins')       // 3. Initialize the surname state variable with
'Poppins'                 // 4. Add an effect for updating the title

// -----
// Second render
// -----
useState('Mary')           // 1. Read the name state variable (argument is
ignored)                  // ignored)
useEffect(persistForm)     // 2. Replace the effect for persisting the form
useState('Poppins')        // 3. Read the surname state variable (argument is
ignored)                  // ignored)
useEffect(updateTitle)      // 4. Replace the effect for updating the title

// ...
```

フックへの呼び出しの順番がレンダー間で変わらない限り、React はそれらのフックにローカル state を割り当てることができます。ですがフックの呼び出しを条件分岐内（例えば `persistForm` 副作用の内部で）で行ったらどうなるでしょうか？

```
// 🔴 We're breaking the first rule by using a Hook in a condition
if (name !== '') {
  useEffect(function persistForm() {
    localStorage.setItem('formData', name);
  });
}
```

`name !== ''` という条件は初回のレンダー時には `true` なので、フックは実行されます。しかし次回のレンダー時にはユーザーがフォームをクリアしているかもしれません、その場合にこの条件は `false` になります。するとレンダー途中でこのフックがスキップされるため、フックの呼ばれる順番が変わってしまいます。

```
useState('Mary')           // 1. Read the name state variable (argument is
                           // ignored)
// useEffect(persistForm) // 🔴 This Hook was skipped!
useState('Poppins')        // 🔴 2 (but was 3). Fail to read the surname state
                           // variable
useEffect(updateTitle)     // 🔴 3 (but was 4). Fail to replace the effect
```

React は 2 つ目の `useState` の呼び出しに対して何を返せばいいのか分からなくなります。React は 2 つめのフックの呼び出しは前回レンダー時と同様に `persistForm` に対応するものだと期待しているのですが、それが成り立たなくなっています。この部分より先では、スキップされたもの以降のすべてのフックがひとつずつずれているため、バグを引き起こします。

**これがフックを呼び出すのがトップレベルのみでなければならない理由です。** 条件付きで副作用を走らせたい場合は、その条件をフックの内部に入れることができます：

```
useEffect(function persistForm() {
  // 👍 We're not breaking the first rule anymore
  if (name !== '') {
    localStorage.setItem('formData', name);
  }
});
```

上記の [lint ルール](#)を使えばこの問題について心配する必要はない、ということに注意してください。しかしフックがなぜこのように動作するのか、このルールがどんな問題を防いでいるのかについて学びました。

## 次のステップ

ついに[自分独自のフックの書き方](#)について学ぶ準備ができました！カスタムフックを使えば React から提供されるフックを組み合わせて自分独自の抽象化を作り出し、複数の異なるコンポーネント間で `state` を使う共通のロジックを再利用することができます。

# ステートフックの利用法

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。

[フックの導入のページ](#)で以下の例を挙げてフックの紹介を行いました：

```
import React, { useState } from 'react';

function Example() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

このコードをクラスによる等価版と比較しながらフックについて学び始めましょう。

## クラスによる同等の例

以前 React でクラスを使っていたなら、このコードに馴染みがあるでしょう。

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

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

state は `{ count: 0 }` から始まり、ユーザがボタンをクリックした時に `this.setState()` を呼ぶことで `state.count` をインクリメントします。このページの全体にわたってこのクラスからの例が出てきます。

### 補足

もっと現実的な例ではなくカウンタを使っているのはなぜか気になるかもしれません。フックについての第一歩の説明ですので API にフォーカスするためです。

## フックと関数コンポーネント

念のため、React の関数コンポーネントとはこのようなものです：

```
const Example = (props) => {
  // You can use Hooks here!
  return <div />;
}
```

あるいはこのようなものです：

```
function Example(props) {
  // You can use Hooks here!
  return <div />;
}
```

これらのことを行なうことを「ステートレスコンポーネント (stateless component)」だと理解していたかもしれません。今からこれらの内部で React の state を利用できるようにしているので、「関数コンポーネント (function component)」という名前を使用しましょう。

フックはクラスの内部では動作しません。クラスを書く代わりに使うものです。

## フックとは何か？

この新しい例では、React から `useState` をインポートするところから始めましょう。

```
import React, { useState } from 'react';

function Example() {
  // ...
}
```

**フックとは何？** フックとは React の機能に「接続 (hook into)」するための特別な関数です。例えば `useState` によって React の state の機能を関数コンポーネントに追加できます。他のフックについても後で学びます。

**フックをいつ使うべき？** 関数コンポーネントを書いていて `state` が必要だと気付いた場合、これまでにはコンポーネントをクラスに変換する必要がありました。今後は、既に書いた関数コンポーネントの中でフックを使うことができます。早速やってみましょう！

### 補足：

コンポーネント内のどこでフックが使ってどこで使えないかに関する特別なルールがあります。これについては[フックのルール](#)で学びます。

## state 変数の宣言

クラスでは、コンストラクタ内で `this.state` を `{ count: 0 }` にセットするという方法で、`state` である `count` を `0` へと初期化します。

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

関数コンポーネントには `this` は存在しませんので、`this.state` を読み書きすることはできません。その代わりに、コンポーネントの内部で直接 `useState` フックを使いましょう。

```
import React, { useState } from 'react';

function Example() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);
```

`useState` を呼ぶと何が起きるの？ これにより『state 変数』が宣言されます。我々の例では変数を `count` と名付けましたが、`banana` でも何でも好きな名前にすることができます。これが関数コールの間で値を『保持』しておくための方法です — `useState` は、クラスにおいて `this.state` が提供するのと全く同じ機能を実現するための新しい方法です。通常、関数が終了すると変数は『消えて』しまいますが、`state` 変数は `React` によって保持されます。

引数として `useState` に何を渡すのか？ `useState()` フックの唯一の引数は `state` の初期値です。クラスの場合とは異なり、`state` はオブジェクトである必要はありません。数字や文字列が欲しいだけであればそれらを保持することができます。我々の例ではユーザがクリックした回数に対応する数字が欲しいだけですので、`0` を `state` 変数の初期値として渡します（もし違う値を保持したい場合は `useState()` を 2 回呼ぶことになります）。

`useState` は何を返すのか？ 現在の `state` と、それを更新するための関数とを、ペアにして返します。これが `const [count, setCount] = useState()` という書き方をした理由です。これはクラスにおける `this.state.count` と `this.setState` に似ていますが、それをペアとして手に入れる点が異なります。もしもここで使った構文になじみがない場合は、[このページの下部](#)で改めて説明します。

`useState` が何をやるのかが分かったので、最初の例の意味がより分かりやすくなりました：

```
import React, { useState } from 'react';

function Example() {
  // Declare a new state variable, which we'll call "count"
  const [count, setCount] = useState(0);
```

`count` という名前の state 変数を宣言しており、それを `0` にセットします。再レンダー間で React はその変数の現在値を記憶しており、最新の値を関数に渡します。現在の `count` の値を更新したい場合は、`setCount` を呼ぶことができます。

### 補足

どうして `createState` ではなく `useState` という名前なのか気になるでしょうか？

`state` が「作成」されるのはコンポーネントの初回レンダー時だけですので、`createState` という名前はあまり正確ではありません。次回以降のレンダー時には、`useState` からは既存の `state` の現在値を受け取ります。毎回作成していたのではそもそも「状態」になりませんね。また、フックの名前が常に `use` から始まることには理由があります。[フックのルール](#)で改めて説明します。

## state の読み出し

クラス内で現在のカウント値を表示したい場合、`this.state.count` を読み出します：

```
<p>You clicked {this.state.count} times</p>
```

関数内では、直接 `count` を使うことができます：

```
<p>You clicked {count} times</p>
```

## state の更新

クラスでは、`this.setState()` を呼ぶことで `count` ステートを更新します：

```
<button onClick={() => this.setState({ count: this.state.count + 1 })}>
  Click me
</button>
```

関数内では、すでに `setCount` と `count` を変数として受け取っていますので、`this` は必要ありません：

```
<button onClick={() => setCount(count + 1)}>
  Click me
</button>
```

## まとめ

ではこれまで学んだことを 1 行ずつまとめて、理解を確認しましょう。

```
1: import React, { useState } from 'react';
2:
3: function Example() {
4:   const [count, setCount] = useState(0);
5:
6:   return (
7:     <div>
8:       <p>You clicked {count} times</p>
```

```

9:         <button onClick={() => setCount(count + 1)}>
10:            Click me
11:        </button>
12:      </div>
13:    );
14:  }

```

- 1 行目 :** `useState` フックを React からインポートします。これにより関数コンポーネント内でローカル state が使えるようになります。
- 4 行目 :** `Example` コンポーネント内で `useState` フックを呼び出すことで新しい state 変数を宣言します。2 つの値のペアが返されるので、それらに名前を与えます。ボタンのクリック回数を保持するための変数ですので `count` と名付けましょう。`useState` 唯一の引数として `0` を渡すことで、変数をゼロへと初期化します。返り値の 2 つ目はそれ自体が関数です。これにより `count` を更新するので、`setCount` という名前にします。
- 9 行目 :** ユーザがクリックした時に、新しい値で `setCount` を呼びます。React は `Example` コンポーネントを再レンダーし、その際には新たな `count` の値を渡します。

最初は飲み込むのに時間がかかるかもしれません。急がないようにしましょう！途中で分からなくなったり場合は上記のコードを最初から最後まで読み直してください。一旦クラスで state がどう動くのかを「忘れて」新鮮な目でこのコードを見るようすれば、必ず理解できるようになるはずです。

## ヒント:この角カッコの意味は？

`state` 変数を宣言するときのこの角カッコに気付かれたでしょうか：

```
const [count, setCount] = useState(0);
```

左辺に書かれている名前は、React の API の一部ではありません。`state` 変数には好きな名前を付けることができます：

```
const [fruit, setFruit] = useState('banana');
```

この JavaScript の構文は "分割代入 (destructuring)" と呼ばれています。これが意味するところは、`fruit` と `setFruit` という名前の 2 つの変数を作り、`useState` から返される値のうち 1 つ目を `fruit` に、2 つ目を `setFruit` に代入する、ということです。これは以下のコードと等価です：

```

var fruitStateVariable = useState('banana'); // Returns a pair
var fruit = fruitStateVariable[0]; // First item in a pair
var setFruit = fruitStateVariable[1]; // Second item in a pair

```

`useState` で `state` 変数を宣言する際、つまり 2 つの要素を含んだ配列が返されます。1 つ目の要素は `state` の現在の値、2 つ目の要素はそれを更新するための関数です。これらには特定の意味があるので、アクセスするのに `[0]` や `[1]` を使うのではちょっと分かりづらくなります。だから代わりに分割代入を使うというわけです。

### 補足

React に対して `this` のようなものを一切渡していないので、どのようにコンポーネントと `useState` の呼び出しの対応を知るのか不思議に思うかもしれません。FAQ セクションで、[この質問](#) およびその他の疑問についてお答えしています。

## ヒント:複数の state 変数を使う

state 変数の宣言を `[something, setSomething]` のペアの形で行うのが便利であるもうひとつの理由は、state 変数を複数使いたくなった場合にそれらに異なる名前をつけることができるからです：

```
function ExampleWithManyStates() {
  // Declare multiple state variables!
  const [age, setAge] = useState(42);
  const [fruit, setFruit] = useState('banana');
  const [todos, setTodos] = useState([{ text: 'Learn Hooks' }]);
}
```

上記の例では、ローカル変数として `age`、`fruit`、`todos` があり、それぞれを個別に更新することができます。

```
function handleOrangeClick() {
  // Similar to this.setState({ fruit: 'orange' })
  setFruit('orange');
}
```

たくさんの state 変数を使う必要はありません。state 変数はオブジェクトや配列も何ら問題なく保持できますので、関連する値をいっしょにまとめておくこともできます。しかしクラスでの `this.setState` とは異なり、state 変数の更新は、マージではなく必ず古い値を置換します。

独立した state 変数を分割する際の推奨事項については [FAQ](#) で詳しく述べています。

## 次のステップ

このページでは React によって提供されるフックのうちのひとつである `useState` について学びました。これからは「ステートフック」という名前でも呼ぶことにします。ステートフックによって、React の歴史上はじめて、React の関数コンポーネントにローカル state を加えることができます！

またフックが一体何なのかについても少々学びました。フックとは関数コンポーネント内から React の機能に「接続する (hook into)」ための関数のことです。フックの名前は必ず `use` から始まり、他にもまだ説明していない様々なフックがあります。

**それでは次のフックである `useEffect` について学びましょう。** これらはクラスにおけるライフサイクルメソッドと似ており、コンポーネント内で副作用を実行することができるようになります。

# 貢献の方法

React は Facebook の最初のオープンソースプロジェクトの 1 つで現在も非常に活発に開発されており、facebook.com 上のあらゆる人々にコードを届けることにも使用されています。私たちはこのプロジェクトへの貢献ができるだけ簡単かつ透明性の高いものにするために努力していますが、まだ完全ではありません。このドキュメントがプロジェクトへの貢献の手順を明確にし、あなたの持つ疑問を解決できれば幸いです。

## 行動規範 {#code-of-conduct}

Facebook は行動規範として [Contributor Covenant](#) を採用しており、プロジェクト参加者が遵守することを期待しています。全文を読んでください、そうすれば参加者はどのような行動を取ればよいか、またどのような行動が許容されないのか理解できるでしょう。

## オープンな開発

React に関する開発作業はすべて GitHub 上で直接行われます。コアチームメンバーと外部のコントリビューターの両方が、同じレビュープロセスを経由するプルリクエストを送ります。

## セマンティック・バージョニング

React はセマンティック・バージョニングに従います。重要なバグ修正のためにパッチバージョンを、新機能や非本質的な変更のためにマイナーバージョンを、そして破壊的変更のためにメジャーバージョンをリリースします。私たちが破壊的変更を加える場合、ユーザが今後の変更について前もって知り、コードを移行できるようにするために、私たちはマイナーバージョンで非推奨警告を行います。我々の安定性および逐次的なマイグレーションに関する取り決めについては、[バージョニングポリシー](#)をご覧ください。

重要な変更はすべて [changelog file](#) に文書化されています。

## ブランチの構成

プルリクエストを送信する場合は、直接 main ブランチに対して行ってください。開発用、あるいは次期リリース用の別ブランチは使っていません。main をすべてのテストが通る良い状態に保つために努力しています。

main ブランチに取り込まれるコードは最新の安定リリースと互換性がなければなりません。追加の機能はあっても構いませんが、破壊的変更があってはなりません。main ブランチの先頭からは、新しいマイナーバージョンが常にリリースできる状態であるべきです。

## フィーチャー・フラグ (Feature Flags)

main ブランチをリリース可能な状態に保つため、破壊的な変更や実験的な機能はフィーチャー・フラグの背後で動作させます。

フィーチャー・フラグは packages/shared/ReactFeatureFlags.js で定義されています。React のいくつかのビルドでは別のフィーチャー・フラグを有効化していることがあります。例えば React Native のビルドは React DOM とは別に設定されているかもしれません。そのようなフラグは packages/shared/forks にあります。フィーチャー・フラグは Flow で静的に型付けされているため yarn flow を実行することで必要なフラグを更新したことを確認できます。

React のビルドシステムが、無効化されている機能を公開前に取り除きます。すべてのコミットについて継続的インテグレーションのジョブが実行され、バンドルサイズの変化を確認します。このサイズの変化の情報を使って、ある機能が正しくフラグ管理されていることを確認することが可能です。

## バグ

### 既知の問題を知るには

私たちは公開されるバグの管理に [GitHub Issues](#) を使用しています。私たちはこれを注意深く見守り、修正中の作業がある場合はそれを明確にするようにします。新しい Issue を提出する前に、既に同じものが存在しないか確かめてください。

### 新しい問題の報告

バグを修正するための最善の方法は、バグを再現する最小のテストケースを提供することです。この [JSFiddle テンプレート](#) は素晴らしい出発点です。

### セキュリティバグ

Facebook にはセキュリティバグの安全な開示のための [報奨金制度](#) が存在します。それを念頭において、セキュリティバグは公開の Issues に提出せず、上記のページの手順に従ってください。

## 連絡方法

- IRC : [# freenode の reactjs](#)
- [ディスカッションフォーラム](#)

また、React に関して助けが必要な場合は、[Discord 上の React コミュニティ](#) も存在します。

## 変更の提案

もしあなたがパブリック API に変更を加えたり、実装に些細とはいえない変更を加えたりしたい場合、Issue を提出することをお勧めします。これによって、あなたが大きな労力を割く前に提案について合意に達することができます。

バグを修正するだけの場合は、すぐにプルリクエストを送信しても問題ありませんが、修正したいバグの内容を詳細に記載した Issue を提出することをお勧めします。これは、あなたの修正自体は受け付けないがバグの追跡はしたいという場合に役立ちます。

## 初めてのプルリクエスト

はじめてのプルリクエストに取り組んでみますか？ この無料ビデオシリーズから手順を学ぶことができます：

### [GitHub でオープンソースプロジェクトに貢献する方法](#)

あなたが新しい試みをする上で、貢献プロセスに慣れるのを助けるために、私たちは比較的影響範囲の少ないバグを含む [good first issues](#) のリストを持っています。これはオープンソースプロジェクトへの貢献の入門に最適です。

Issue を解決することにした場合、誰かがすでに修正に取り組んでいる場合に備えて、コメントスレッドを必ず確認してください。現時点で誰も作業していない場合は、他の人が誤って重複して作業をしないように、作業する予定であることを示すコメントを残してください。

誰かが取り組むと宣言した Issue が 2 週間以上放置されている場合、それを引き継ぐことは問題ありませんが、その場合もコメントを残すべきです。

## プルリクエストを送信する

コアチームはプルリクエストを監視しています。プルリクエストをレビューしてマージするか、変更を要求するか、説明付きでクローズします。Facebook.com 内部で使用法の検討が必要な可能性のある API の変更については、対応が遅くなることがあります。プロセス全体を通して最新情報とフィードバックを提供するよう最善を尽くします。

**プルリクエストを送信する前に、以下のプロセスが行われているか確認してください：**

1. [リポジトリ](#) をフォークして `main` から新しいブランチを作成します。
2. `yarn` コマンドをリポジトリルートで実行します。
3. バグを修正したり、テストが必要なコードを追加した場合は、テストを追加します。
4. テストスイートが通ることを確認してください (`yarn test`)。ヒント：`yarn test --watch TestName` コマンドは開発時に役立ちます。
5. 本番環境でテストするために `yarn test --prod` コマンドを実行します。
6. デバッガが必要な場合は `yarn debug-test --watch TestName` を実行し `chrome://inspect` を開き "Inspect" を押してください。
7. [prettier](#) でコードをフォーマットします (`yarn prettier`)。
8. リントを行います (`yarn lint`)。ヒント：`yarn linc` は変更されたファイルのみに適用できます。
9. [Flow](#) による型チェックを行います (`yarn flow`)。
10. まだの場合は、先に CLA (Contributor License Agreement) の提出を済ませます。

## Contributor License Agreement (CLA)

あなたのプルリクエストを受け付けるために、Contributor License Agreement (CLA) の提出を行って頂く必要があります。これは一度だけ行えば良いので、あなたが他の Facebook オープンソースプロジェクトで既に完了させている場合は必要ありません。初めてプルリクエストを送信する場合は、CLA 提出を完了させたことをお知らせください。そうすれば私たちは GitHub のユーザ名と照らし合わせてチェックを行います。

[ここで CLA を完了させてください。](#)

## 貢献の前提条件

- Node の LTS と、Yarn v1.2.0+ がインストールされていること。
- JDK がインストールされていること。
- gcc がインストールされている、または必要に応じたコンパイラをインストールすることができる。依存関係の中にはコンパイルステップが必要なものもあります。OS X では Xcode のコマンドラインツールが役立つでしょう。Ubuntu では `apt-get install build-essential` コマンドで必要なパッケージをインストールできます。他の Linux ディストリビューションでも似たようなコマンドで実現できるでしょう。Windows では追加の手順が必要になり、詳しくは [node-gyp installation instructions](#) を参照してください。
- Git について精通していること。

## 開発ワークフロー

React リポジトリをクローンしたあと、`yarn` コマンドで依存関係のパッケージを取得してください。そうすれば、いくつかのコマンドが実行可能になります：

- `yarn lint` コードスタイルをチェックします。
- `yarn linc` `yarn lint` と似ていますが、変更されたファイルのみをチェックするのでこちらの方が速いです。
- `yarn test` 全てのテストスイートを実行します。
- `yarn test --watch` 対話式のテストウォッチャーを実行します。
- `yarn test <pattern>` 指定したパターンにマッチするファイルのみテストを実行します。
- `yarn test --prod` 本番環境でテストを実行します。
- `yarn debug-test` は `yarn test` に似ていますがデバッガ付きです。 `chrome://inspect` を開き "Inspect" を押してください。
- `yarn flow` `Flow` による型チェックを行います。
- `yarn build` 全てのパッケージを含む `build` フォルダを作成します。
- `yarn build react/index,react-dom/index --type=UMD` React と ReactDOM だけの UMD ビルドを作成します。

`yarn test` (またはそれに近い上記のコマンド) を実行し、あなたの行った変更によって何らかの異常を引き起こしていないか確認することをお勧めします。とはいえた実際のプロジェクトで自分の React のビルドを使ってみることも役に立つでしょう。

まず `yarn build` を実行します。これによってビルド済みのバンドルファイルが `build` フォルダ内に作られ、同時に `build/packages` 内に npm パッケージも用意されます。

変更を試す一番簡単な方法は `yarn build react/index,react-dom/index --type=UMD` を実行し、`fixtures/packaging/babel-standalone/dev.html` を開くことです。このファイルは `build` フォルダの `react.development.js` を既に使用しているので、変更が反映されます。

あなたの加えた変更を既存の React プロジェクトで試したい場合、`build/node_modules/react/umd/react.development.js`、`build/node_modules/react-dom/umd/react-dom.development.js`、もしくは他のビルドされたファイルをあなたのアプリケーションにコピーして安定版の代わりに使用することができます。

もし、npm 版の React を使用している場合は `react` と `react-dom` を依存関係から削除し、`yarn link` を使用してそれらがローカルの `build` フォルダを指すようにしてください。ビルド時には `--type=UMD` の代わりに `--type=NODE` を渡す必要があることに注意してください。また `scheduler` パッケージもビルドする必要があります：

```
cd ~/path_to_your_react_clone/
yarn build react/index,react/jsx,react-dom/index,scheduler --type=NODE

cd build/node_modules/react
yarn link
cd build/node_modules/react-dom
yarn link

cd ~/path/to/your/project
yarn link react react-dom
```

`yarn build` を React フォルダで実行するたびに、あなたのプロジェクトの `node_modules` フォルダに更新されたバージョンが現れるでしょう。その後、プロジェクトをビルドし直して変更を試すことができます。

もしまだいくつかのパッケージが不足している場合（例えばプロジェクトで `react-dom/server` を使っている場合）、常に `yarn build` でフルのビルドを行うことができます。ただしオプションなしの `yarn build` は時間がかかることに注意してください。

プルリクエストにあなたの新機能に応じたユニットテストを含めることは必須です。これによって将来あなたのコードを壊してしまわないことが担保されます。

## スタイルガイド

Prettier と呼ばれる自動コードフォーマッタを使います。 `yarn prettier` コマンドをコードを変更した後に実行してください。

そうすれば、後はリンターがあなたのコードに存在するほとんどの問題を捕らえるでしょう。自分の書いたコードのスタイルをチェックしたい場合は単に `yarn linc` を実行してください。

しかしながら、リンターでもチェックしきれないいくつかのスタイルがあります。何か分からなことがあります。Airbnb's Style Guide が正しい方向に導いてくれるでしょう。

## Request for Comments (RFC)

バグ修正やドキュメンテーションの改善を含む多くの変更は、通常の GitHub プルリクエストのワークフローを通して行われレビューされます。

ただし、いくつかの "大きめの" 変更は、多少の設計プロセスと React コアチームの合意を経ることをお願いします。

"RFC" (request for comments) プロセスは、新機能がプロジェクトに取り込まれるまでの一貫性があり整備された筋道を提供することを目的としています。rfcs リポジトリ を訪れれば貢献することができます。

## ライセンス

React に貢献するにあたって、あなたの貢献は MIT ライセンスの元にあることに同意したとみなします。

## 次のセクション

[次のセクション](#) を読んで、コードベースの構成方法について知ることができます。

# 実装に関するメモ

この章は [stack リコンサイラ \(reconciler\)](#) の実装に関するメモを集めたものです。

これは非常に技術的な内容であり、React の公開 API だけでなく、React がどのようにコア、レンダラ (renderer)、そしてリコンサイラに分割されているかについても、深く理解していることを前提としています。React のコードベースにあまり精通していないのであれば、まず [コードベースの概要](#) を読んでください。

また、これは [React のコンポーネント、インスタンスおよび要素の違い](#) についての理解を前提としています。

stack リコンサイラは、React 15 およびそれ以前のバージョンで使われていました。  
[src/renderers/shared/stack/reconciler](#) で見つけることができます。

## 動画: React をスクラッチで作成する

このドキュメントは、[Paul O'Shannessy](#) 氏の行った講演 [building React from scratch](#) に大いに啓発されています。

このドキュメントと彼の講演は、ともに実際のコードベースを簡素化したもので、両方に親しむことでより深く理解することができるでしょう。

## 概要

リコンサイラそのものは公開 API を持たません。リコンサイラは、React DOM や React Native のような [レンダラ](#) が、ユーザの記述した React コンポーネントに応じてユーザインターフェースを効率よく更新するために使用されます。

## 再帰的な処理としてマウントする

一番最初にコンポーネントをマウントするときのことを考えてみましょう：

```
const root = ReactDOM.createRoot(rootEl);
root.render(<App />);
```

`root.render` はリコンサイラに `<App />` を渡します。`<App />` が React 要素であること、つまり、何をレンダーするかの説明書きであることを思い出してください。これはプレーンなオブジェクトとして考えることができます：

```
console.log(<App />);
// { type: App, props: {} }
```

リコンサイラは `App` がクラスか関数かをチェックします。

もし `App` が関数なら、リコンサイラは `App(props)` を呼び出してレンダーされた要素を取得します。

もし `App` がクラスなら、リコンサイラは `new App(props)` で `App` をインスタンス化し、`componentWillMount()` ライフサイクルメソッドを呼び出し、それから `render()` メソッドを呼び出してレンダーされた要素を取得します。

どちらにせよ、リコンサイラは `App` が「レンダーされた」結果となる要素を手に入れます。

このプロセスは再帰的です。App は <Greeting /> へとレンダーされるかもしれませんし、Greeting は <Button /> にレンダーされるかもしれない、といったように続いていきます。リコンサイラはそれぞれのコンポーネントが何にレンダーされるかを学習しながら、ユーザ定義コンポーネントを再帰的に「掘り下げる」いきます。

この処理の流れは擬似コードで想像することができます：

```
function isClass(type) {
  // React.Component subclasses have this flag
  return (
    Boolean(type.prototype) &&
    Boolean(type.prototype.isReactComponent)
  );
}

// This function takes a React element (e.g. <App />)
// and returns a DOM or Native node representing the mounted tree.
function mount(element) {
  var type = element.type;
  var props = element.props;

  // We will determine the rendered element
  // by either running the type as function
  // or creating an instance and calling render().
  var renderedElement;
  if (isClass(type)) {
    // Component class
    var publicInstance = new type(props);
    // Set the props
    publicInstance.props = props;
    // Call the lifecycle if necessary
    if (publicInstance.componentWillMount) {
      publicInstance.componentWillMount();
    }
    // Get the rendered element by calling render()
    renderedElement = publicInstance.render();
  } else {
    // Component function
    renderedElement = type(props);
  }

  // This process is recursive because a component may
  // return an element with a type of another component.
  return mount(renderedElement);
}

// Note: this implementation is incomplete and recurses infinitely!
// It only handles elements like <App /> or <Button />.
// It doesn't handle elements like <div /> or <p /> yet.
}

var rootEl = document.getElementById('root');
var node = mount(<App />);
rootEl.appendChild(node);
```

#### 補足：

これは**全くの擬似コード**です。本物の実装に近いものではありません。また、いつ再帰を止めるか検討していないため、このコードはスタックオーバーフローを引き起こします。

上記の例でいくつかの鍵となるアイデアをおさらいしましょう：

- React 要素とはコンポーネントの型（例えば `App`）と `props` を表すプレーンなオブジェクトである。
- ユーザ定義コンポーネント（例えば `App`）はクラスであっても関数であってもよいが、それらは全て要素へと「レンダ一される」。
- 「マウント」とは、最上位の React 要素（例えば `<App />`）を受け取り、DOM もしくはネイティブなツリーを構築する再帰的な処理である。

## host要素のマウント

このようにして要素ができても、それを使って画面に何か表示しなければ意味がありません。

ユーザ定義 ("composite") コンポーネントに加え、React 要素はプラットフォームに固有な ("host") コンポーネントも表すことができます。例えば、`Button` は `render` メソッドから `<div />` を返すことが考えられます。

もし要素の `type` プロパティが文字列なら、私たちはいま host 要素を扱っていることになります：

```
console.log(<div />);
// { type: 'div', props: {} }
```

host 要素に関連付けられているユーザ定義のコードはありません。

リコンサイラは host 要素を見つけると、レンダラに host 要素のマウントを任せます。例えば、React DOM は DOM ノードを生成します。

host 要素に子要素がある場合、リコンサイラは前節で述べたものと同じアルゴリズムに従い、子要素を再帰的にマウントします。子要素が（`<div><hr /></div>` のような）host なのか、（`<div><Button /></div>` のような）composite なのか、もしくはその両方が含まれているかに関わらず、再帰的な処理が実行されます。

子コンポーネントにより生成された DOM ノードは親の DOM ノードに追加され、それが再帰的に行われることで、完全な DOM 構造が組み立てられます。

### 補足：

リコンサイラそのものは DOM と結合していません。マウントの結果自体（時にソースコードでは "mount image" とも呼ばれます）はレンダラに依存し、それは（React DOM なら）DOM ノードであったり、（React DOM Server なら）文字列であったり、（React Native なら）ネイティブのビューを表す数字であったりします。

前出のコードを host 要素も扱えるように拡張するとすれば、以下のようなものになるでしょう：

```
function isClass(type) {
  // React.Component subclasses have this flag
  return (
    Boolean(type.prototype) &&
    Boolean(type.prototype.isReactComponent)
  );
}

// This function only handles elements with a composite type.
// For example, it handles <App /> and <Button />, but not a <div />.
```

```

function mountComposite(element) {
  var type = element.type;
  var props = element.props;

  var renderedElement;
  if (isClass(type)) {
    // Component class
    var publicInstance = new type(props);
    // Set the props
    publicInstance.props = props;
    // Call the lifecycle if necessary
    if (publicInstance.componentWillMount) {
      publicInstance.componentWillMount();
    }
    renderedElement = publicInstance.render();
  } else if (typeof type === 'function') {
    // Component function
    renderedElement = type(props);
  }

  // This is recursive but we'll eventually reach the bottom of recursion when
  // the element is host (e.g. <div />) rather than composite (e.g. <App />):
  return mount(renderedElement);
}

// This function only handles elements with a host type.
// For example, it handles <div /> and <p /> but not an <App />.
function mountHost(element) {
  var type = element.type;
  var props = element.props;
  var children = props.children || [];
  if (!Array.isArray(children)) {
    children = [children];
  }
  children = children.filter(Boolean);

  // This block of code shouldn't be in the reconciler.
  // Different renderers might initialize nodes differently.
  // For example, React Native would create iOS or Android views.
  var node = document.createElement(type);
  Object.keys(props).forEach(propName => {
    if (propName !== 'children') {
      node.setAttribute(propName, props[propName]);
    }
  });

  // Mount the children
  children.forEach(childElement => {
    // Children may be host (e.g. <div />) or composite (e.g. <Button />).
    // We will also mount them recursively:
    var childNode = mount(childElement);

    // This line of code is also renderer-specific.
    // It would be different depending on the renderer:
    node.appendChild(childNode);
  });

  // Return the DOM node as mount result.
  // This is where the recursion ends.
  return node;
}

```

```

function mount(element) {
  var type = element.type;
  if (typeof type === 'function') {
    // User-defined components
    return mountComposite(element);
  } else if (typeof type === 'string') {
    // Platform-specific components
    return mountHost(element);
  }
}

var rootEl = document.getElementById('root');
var node = mount(<App />);
rootEl.appendChild(node);

```

このコードは動作しますが、それでもまだ現実のリコンサイラの実装方法からは隔たりがあります。ここにあるべき鍵となる要素は、更新に対応することです。

## 内部インスタンスの導入

React の鍵となる機能は、あらゆるものを再描画できることであり、その際に DOM を再生成したり、state をリセットしたりしないことです：

```

root.render(<App />);
// Should reuse the existing DOM:
root.render(<App />);

```

しかし、前節で実装したコードは最初のツリーをマウントする方法しか知りません。前節のコードは、全ての `publicInstance` や、どの DOM `node` がどのコンポーネントに対応しているかなど、必要な全情報を保有しているわけではないので、更新を実行することができません。

stack リコンサイラのコードベースでは、この問題を `mount()` 関数をメソッドとしてクラスに置くことで解決しています。しかしこのアプローチには欠点があるため、[進行中のリコンサイラの書き直し作業](#)では、反対の方向に進んでいます。それでも現時点では、この方式で動作しています。

別々の `mountHost` と `mountComposite` 関数の代わりに、2つのクラスを作成します：`DOMComponent` と `CompositeComponent` です。

両方のクラスが `element` を受け入れるコンストラクタと、マウントされたノードを返す `mount()` メソッドを持ちます。最上位の `mount()` 関数を、正しいクラスをインスタンス化するファクトリに置き換えます：

```

function instantiateComponent(element) {
  var type = element.type;
  if (typeof type === 'function') {
    // User-defined components
    return new CompositeComponent(element);
  } else if (typeof type === 'string') {
    // Platform-specific components
    return new DOMComponent(element);
  }
}

```

まず、`CompositeComponent` の実装から考えてみましょう：

```

class CompositeComponent {
  constructor(element) {
    this.currentElement = element;
    this.renderedComponent = null;
    this.publicInstance = null;
  }

  getPublicInstance() {
    // For composite components, expose the class instance.
    return this.publicInstance;
  }

  mount() {
    var element = this.currentElement;
    var type = element.type;
    var props = element.props;

    var publicInstance;
    var renderedElement;
    if (isClass(type)) {
      // Component class
      publicInstance = new type(props);
      // Set the props
      publicInstance.props = props;
      // Call the lifecycle if necessary
      if (publicInstance.componentWillMount) {
        publicInstance.componentWillMount();
      }
      renderedElement = publicInstance.render();
    } else if (typeof type === 'function') {
      // Component function
      publicInstance = null;
      renderedElement = type(props);
    }

    // Save the public instance
    this.publicInstance = publicInstance;

    // Instantiate the child internal instance according to the element.
    // It would be a DOMComponent for <div /> or <p />,
    // and a CompositeComponent for <App /> or <Button />:
    var renderedComponent = instantiateComponent(renderedElement);
    this.renderedComponent = renderedComponent;

    // Mount the rendered output
    return renderedComponent.mount();
  }
}

```

以前の `mountComposite()` の実装と大きな違いはありませんが、更新時に使用する `this.currentElement`、`this.renderedComponent` や、`this.publicInstance` のような情報を保存できるようになりました。

`CompositeComponent` のインスタンスは、ユーザが指定する `element.type` のインスタンスとは同一ではないことに注意してください。`CompositeComponent` はリコンサイラの実装の詳細であり、ユーザには決して公開されません。ユーザ定義クラスとは `element.type` から読み込むものであり、`CompositeComponent` がそのインスタンスを作成するのです。

混乱を避けるために、`CompositeComponent` と `DOMComponent` のインスタンスを「内部インスタンス」と呼ぶことにします。内部インスタンスは、長期間利用されるデータとそれらを関連付けられるようにするために存在します。それらの存在はレンダラとリコンサイラのみが認識しています。

一方、ユーザ定義クラスのインスタンスは「公開インスタンス」と呼ぶことにします。公開インスタンスは、独自コンポーネントの `render()` やその他のメソッド内で `this` として現れるものです。

`mountHost()` 関数は、`DOMComponent` クラスの `mount()` メソッドとしてリファクタリングされ、こちらも見慣れたものになります：

```
class DOMComponent {
  constructor(element) {
    this.currentElement = element;
    this.renderedChildren = [];
    this.node = null;
  }

  getPublicInstance() {
    // For DOM components, only expose the DOM node.
    return this.node;
  }

  mount() {
    var element = this.currentElement;
    var type = element.type;
    var props = element.props;
    var children = props.children || [];
    if (!Array.isArray(children)) {
      children = [children];
    }

    // Create and save the node
    var node = document.createElement(type);
    this.node = node;

    // Set the attributes
    Object.keys(props).forEach(propName => {
      if (propName !== 'children') {
        node.setAttribute(propName, props[propName]);
      }
    });

    // Create and save the contained children.
    // Each of them can be a DOMComponent or a CompositeComponent,
    // depending on whether the element type is a string or a function.
    var renderedChildren = children.map(instantiateComponent);
    this.renderedChildren = renderedChildren;

    // Collect DOM nodes they return on mount
    var childNodes = renderedChildren.map(child => child.mount());
    childNodes.forEach(childNode => node.appendChild(childNode));

    // Return the DOM node as mount result
    return node;
  }
}
```

`mountHost()` からリファクタリングした後の主な違いは、`this.node` と `this.renderedChildren` を内部の DOM コンポーネントインスタンスに関連付け続けていることです。これらは、将来的に非破壊的な更新を適用する際にも使用します。

結果として、それが composite であれ host であれ、内部インスタンスはそれぞれの子内部インスタンスを指すようになります。<App> 関数コンポーネントが <Button> コンポーネントをレンダーし、<Button> クラスが <div> をレンダーする場合、視覚的にわかりやすくすると、内部インスタンスのツリーはこのようになります：

```
[object CompositeComponent] {
  currentElement: <App />,
  publicInstance: null,
  renderedComponent: [object CompositeComponent] {
    currentElement: <Button />,
    publicInstance: [object Button],
    renderedComponent: [object DOMComponent] {
      currentElement: <div />,
      node: [object HTMLDivElement],
      renderedChildren: []
    }
  }
}
```

DOM の中では、<div> しか見えません。しかしながら、内部インスタンスのツリーは composite の内部インスタンスと host の内部インスタンスの両方を保有しています。

composite 内部インスタンスは以下のものを格納する必要があります：

- 現在の要素。
- 要素の型がクラスの場合、公開インスタンス。
- 単独の、レンダーされた内部インスタンス。これは DOMComponent か CompositeComponent のいずれかにあたります。

host 内部インスタンスは以下のものを格納する必要があります：

- 現在の要素。
- DOM ノード。
- すべての子内部インスタンス。各インスタンスは、DOMComponent または CompositeComponent のいずれかになります。

より複雑なアプリケーションにおいて、内部インスタンスのツリーがどのような構造になるのか想像しづらい場合は、React DevTools が host インスタンスを灰色に、composite インスタンスを紫色にハイライトしてくれるので、内部インスタンスのツリーにかなり近いものを得ることができます：

```

<TodoApp>
  ▼<div>
    <h3>TODO</h3>
    ▼<TodoList items=[{...}]>
      ▼<ul>
        <li key="1475260226151">Buy milk</li>
      </ul>
    </TodoList>
    ▼<form onSubmit=bound handleSubmit()>
      <input onChange=bound onChange() value="" />
      <button>Add #2</button>
    </form>
  </div>
</TodoApp>

```

このリファクタリングを完了するため、完全なツリーをコンテナノードとパブリックインスタンスにマウントする関数を導入することにします。

```

function mountTree(element, containerNode) {
  // Create the top-level internal instance
  var rootComponent = instantiateComponent(element);

  // Mount the top-level component into the container
  var node = rootComponent.mount();
  containerNode.appendChild(node);

  // Return the public instance it provides
  var publicInstance = rootComponent.getPublicInstance();
  return publicInstance;
}

var rootEl = document.getElementById('root');
mountTree(<App />, rootEl);

```

## アンマウント

これで、子内部インスタンスと DOM ノードを持った内部インスタンスができ、そこにアンマウントを実装できるようになります。composite 要素では、アンマウントはライフサイクルメソッドを呼び出し、再帰的な処理を行います。

```

class CompositeComponent {
  // ...

  unmount() {
    // Call the lifecycle method if necessary
    var publicInstance = this.publicInstance;
    if (publicInstance) {
      if (publicInstance.componentWillUnmount) {
        publicInstance.componentWillUnmount();
      }
    }
  }
}

```

```
// Unmount the single rendered component
var renderedComponent = this.renderedComponent;
renderedComponent.unmount();
}
```

`DOMComponent` では、アンマウントは子要素それぞれにアンマウントするように伝えます：

```
class DOMComponent {
  // ...
  unmount() {
    // Unmount all the children
    var renderedChildren = this.renderedChildren;
    renderedChildren.forEach(child => child.unmount());
  }
}
```

実際には、DOM コンポーネントをアンマウントすると、イベントリスナーの削除とキャッシュのクリアも行われますが、これらの詳細は省略します。

これで `ReactDOM.unmountComponentAtNode()` と同様の `unmountTree(containerNode)` という新規の最上位関数を追加することができます：

```
function unmountTree(containerNode) {
  // Read the internal instance from a DOM node:
  // (This doesn't work yet, we will need to change mountTree() to store it.)
  var node = containerNode.firstChild;
  var rootComponent = node._internalInstance;

  // Unmount the tree and clear the container
  rootComponent.unmount();
  containerNode.innerHTML = '';
}
```

これが動作するよう、DOM ノードから内部ルートインスタンスを読み込む必要があります。`mountTree()` を変更して、ルート DOM ノードに `_internalInstance` プロパティを追加します。`mountTree()` に既存の全てのツリーを破棄するようにも伝えて、複数回 `mountTree()` を呼び出せるようにします：

```
function mountTree(element, containerNode) {
  // Destroy any existing tree
  if (containerNode.firstChild) {
    unmountTree(containerNode);
  }

  // Create the top-level internal instance
  var rootComponent = instantiateComponent(element);

  // Mount the top-level component into the container
  var node = rootComponent.mount();
  containerNode.appendChild(node);

  // Save a reference to the internal instance
  node._internalInstance = rootComponent;

  // Return the public instance it provides
}
```

```

var publicInstance = rootComponent.getPublicInstance();
return publicInstance;
}

```

これで、`unmountTree()` を実行したり、`mountTree()` の実行を繰り返したりしても、古いツリーは破棄され、コンポーネント上で `componentWillUnmount()` ライフサイクルメソッドが実行されるようになりました。

## 更新

前節では、アンマウント機能を実装しました。しかし、各プロパティに変更があるたびにツリー全体をマウントしたりアンマウントしたりするようでは、React の使い勝手はあまり良いとは言えません。リコンサイラの目標は、DOM と state を保持できるように既存のインスタンスを再利用することです。

```

var rootEl = document.getElementById('root');

mountTree(<App />, rootEl);
// Should reuse the existing DOM:
mountTree(<App />, rootEl);

```

もう 1 つメソッドを追加して内部インスタンスを拡張しましょう。`mount()` と `unmount()` に加えて、`DOMComponent` と `CompositeComponent` の両方に `receive(nextElement)` と呼ばれる新しいメソッドを実装しましょう：

```

class CompositeComponent {
  // ...

  receive(nextElement) {
    // ...
  }
}

class DOMComponent {
  // ...

  receive(nextElement) {
    // ...
  }
}

```

このメソッドは、`nextElement` から受け取った指示に従って、コンポーネント（および全ての子要素）を最新の状態にするために必要なあらゆる作業を行う役割を担います。

実際に行われているのは、内部ツリーを再帰的に巡回して各内部インスタンスが更新を受け取れるようにすることなのですが、この処理こそ「仮想 DOM の差分処理 (diffing)」としてしばしば説明される部分です。

## composite コンポーネントの更新

composite コンポーネントが新たな要素を受け取るときに、`componentWillUpdate()` ライフサイクルメソッドを実行します。

それからコンポーネントを新たな `props` で再レンダーし、レンダーされた次の要素を取得します：

```

class CompositeComponent {
  // ...
}

```

```

receive(nextElement) {
  var prevProps = this.currentElement.props;
  var publicInstance = this.publicInstance;
  var prevRenderedComponent = this.renderedComponent;
  var prevRenderedElement = prevRenderedComponent.currentElement;

  // Update *own* element
  this.currentElement = nextElement;
  var type = nextElement.type;
  var nextProps = nextElement.props;

  // Figure out what the next render() output is
  var nextRenderedElement;
  if (isClass(type)) {
    // Component class
    // Call the lifecycle if necessary
    if (publicInstance.componentWillUpdate) {
      publicInstance.componentWillUpdate(nextProps);
    }
    // Update the props
    publicInstance.props = nextProps;
    // Re-render
    nextRenderedElement = publicInstance.render();
  } else if (typeof type === 'function') {
    // Component function
    nextRenderedElement = type(nextProps);
  }
}

// ...

```

続いて、レンダーされた要素の `type` を見てみましょう。もし最後のレンダー以降、`type` が変更されていなければ、これより下のコンポーネントもその場で更新されれば良いということになります。

例えばコンポーネントが最初に `<Button color="red" />` を返し、2回目に `<Button color="blue" />` を返したなら、対応する内部インスタンスに次の要素を `receive()` するよう伝えるだけよいのです：

```

// ...

// If the rendered element type has not changed,
// reuse the existing component instance and exit.
if (prevRenderedElement.type === nextRenderedElement.type) {
  prevRenderedComponent.receive(nextRenderedElement);
  return;
}

// ...

```

ただし、レンダーされた次の要素が前のものと異なる `type` である場合、内部インスタンスの更新はできません。`<button>` が `<input>` に「なる」ことはできないのです。

代わりに、既存の内部インスタンスをアンマウントし、レンダーされた要素の型に対応する新たな内部インスタンスをマウントします。例えば、前に `<button />` をレンダーしていたコンポーネントが `<input />` をレンダーした場合には、この処理が発生します：

```

// ...

// If we reached this point, we need to unmount the previously
// mounted component, mount the new one, and swap their nodes.

```

```

// Find the old node because it will need to be replaced
var prevNode = prevRenderedComponent.getHostNode();

// Unmount the old child and mount a new child
prevRenderedComponent.unmount();
var nextRenderedComponent = instantiateComponent(nextRenderedElement);
var nextNode = nextRenderedComponent.mount();

// Replace the reference to the child
this.renderedComponent = nextRenderedComponent;

// Replace the old node with the new one
// Note: this is renderer-specific code and
// ideally should live outside of CompositeComponent:
prevNode.parentNode.replaceChild(nextNode, prevNode);
}
}

```

まとめると、composite コンポーネントは新たな要素を受け取った際に、レンダーされた内部インスタンスに更新を委任するか、もしくは内部インスタンスをアンマウントしてそこに新しいものをマウントする、ということになります。

もう 1 つ、コンポーネントが要素を受け取らずに再マウントする状況があります。それは要素の `key` が変更された時です。既に複雑なチュートリアルがさらに複雑になってしまうので、このドキュメントでは `key` の取り扱いについては言及しません。

プラットフォーム固有のノードを配置して更新時に置換できるよう、`getHostNode()` と呼ばれるメソッドを内部インスタンスに追加する必要があったことに注意してください。その実装は両方のクラスで簡単にできます：

```

class CompositeComponent {
  // ...

  getHostNode() {
    // Ask the rendered component to provide it.
    // This will recursively drill down any composites.
    return this.renderedComponent.getHostNode();
  }
}

class DOMComponent {
  // ...

  getHostNode() {
    return this.node;
  }
}

```

## host コンポーネントの更新

`DOMComponent` のような host コンポーネントの実装では、異なった更新を行います。要素を受け取る際、背後のプラットフォーム固有のビューを更新する必要があります。React DOM の場合、これは DOM 属性の更新を意味します：

```

class DOMComponent {
  // ...

  receive(nextElement) {
    var node = this.node;
    var prevElement = this.currentElement;
    var prevProps = prevElement.props;
  }
}

```

```

var nextProps = nextElement.props;
this.currentElement = nextElement;

// Remove old attributes.
Object.keys(prevProps).forEach(propName => {
  if (propName !== 'children' && !nextProps.hasOwnProperty(propName)) {
    node.removeAttribute(propName);
  }
});
// Set next attributes.
Object.keys(nextProps).forEach(propName => {
  if (propName !== 'children') {
    node.setAttribute(propName, nextProps[propName]);
  }
});
// ...

```

そして、host コンポーネントは子コンポーネントを更新する必要があります。composite コンポーネントと異なり、host コンポーネントは 1 つ以上の子コンポーネントを保有している可能性があります。

この簡素化した例では内部インスタンスの配列を用い、受け取った `type` と以前の `type` が一致するかによって、内部インスタンスを更新もしくは置換しながら、配列をイテレートしています。実際のリコンサイラでは処理時に要素の `key` を考慮して、要素の挿入と削除に加えて移動を追跡しますが、そのロジックは省略しています。

リストの子要素への DOM 操作は、バッチで実行できるようまとめておきます：

```

// ...

// These are arrays of React elements:
var prevChildren = prevProps.children || [];
if (!Array.isArray(prevChildren)) {
  prevChildren = [prevChildren];
}
var nextChildren = nextProps.children || [];
if (!Array.isArray(nextChildren)) {
  nextChildren = [nextChildren];
}
// These are arrays of internal instances:
var prevRenderedChildren = this.renderedChildren;
var nextRenderedChildren = [];

// As we iterate over children, we will add operations to the array.
var operationQueue = [];

// Note: the section below is extremely simplified!
// It doesn't handle reorders, children with holes, or keys.
// It only exists to illustrate the overall flow, not the specifics.

for (var i = 0; i < nextChildren.length; i++) {
  // Try to get an existing internal instance for this child
  var prevChild = prevRenderedChildren[i];

  // If there is no internal instance under this index,
  // a child has been appended to the end. Create a new
  // internal instance, mount it, and use its node.
  if (!prevChild) {
    var nextChild = instantiateComponent(nextChildren[i]);
    var node = nextChild.mount();
  }
}

```

```

    // Record that we need to append a node
    operationQueue.push({type: 'ADD', node});
    nextRenderedChildren.push(nextChild);
    continue;
}

// We can only update the instance if its element's type matches.
// For example, <Button size="small" /> can be updated to
// <Button size="large" /> but not to an <App />.
var canUpdate = prevChildren[i].type === nextChildren[i].type;

// If we can't update an existing instance, we have to unmount it
// and mount a new one instead of it.
if (!canUpdate) {
  var prevNode = prevChild.getHostNode();
  prevChild.unmount();

  var nextChild = instantiateComponent(nextChildren[i]);
  var nextNode = nextChild.mount();

  // Record that we need to swap the nodes
  operationQueue.push({type: 'REPLACE', prevNode, nextNode});
  nextRenderedChildren.push(nextChild);
  continue;
}

// If we can update an existing internal instance,
// just let it receive the next element and handle its own update.
prevChild.receive(nextChildren[i]);
nextRenderedChildren.push(prevChild);
}

// Finally, unmount any children that don't exist:
for (var j = nextChildren.length; j < prevChildren.length; j++) {
  var prevChild = prevRenderedChildren[j];
  var node = prevChild.getHostNode();
  prevChild.unmount();

  // Record that we need to remove the node
  operationQueue.push({type: 'REMOVE', node});
}

// Point the list of rendered children to the updated version.
this.renderedChildren = nextRenderedChildren;

// ...

```

最後のステップとして、DOM 操作を実行します。ここでも、実際のリコンサイラのコードは要素の移動を扱わなければいけないので、より複雑になります：

```

// ...

// Process the operation queue.
while (operationQueue.length > 0) {
  var operation = operationQueue.shift();
  switch (operation.type) {
    case 'ADD':
      this.node.appendChild(operation.node);
      break;
    case 'REPLACE':
      this.node.replaceChild(operation.nextNode, operation.prevNode);
  }
}

```

```
        break;
    case 'REMOVE':
        this.node.removeChild(operation.node);
        break;
    }
}
```

host コンポーネントの更新については以上です。

### 最上位コンポーネントの更新

ここまで `CompositeComponent` と `DOMComponent` の両方ともが `receive(nextElement)` メソッドを実装しているので、要素の `type` が前回と同じだった場合は最上位の `mountTree()` 関数がそれを使えるよう、この関数を書き換えることができます：

```
function mountTree(element, containerNode) {
  // Check for an existing tree
  if (containerNode.firstChild) {
    var prevNode = containerNode.firstChild;
    var prevRootComponent = prevNode._internalInstance;
    var prevElement = prevRootComponent.currentElement;

    // If we can, reuse the existing root component
    if (prevElement.type === element.type) {
      prevRootComponent.receive(element);
      return;
    }

    // Otherwise, unmount the existing tree
    unmountTree(containerNode);
  }
}

// ...
}
```

これで、同じ型で `mountTree()` を 2 回呼び出しても、破壊的な変更にはなりません：

```
var rootEl = document.getElementById('root');

mountTree(<App />, rootEl);
// Reuses the existing DOM:
mountTree(<App />, rootEl);
```

以上の処理が React 内部での動作の仕組みの基本です。

このドキュメントで除外したもの

このドキュメントは、実際のコードベースよりもシンプルなものになっています。ここでは言及しなかった重要なポイントがいくつかあります：

- コンポーネントは `null` をレンダーでき、リコンサイラは配列やレンダーされた出力における「空スロット」部分を扱うことができます。

- リコンサイラは要素から `key` も読み取り、どの内部インスタンスが配列中のどの要素と対応するかを確認するために使用します。実際の React の実装における複雑さのかなりの部分が、この箇所に関わるものです。
- `composite` と `host` 型の内部インスタンスのクラスに加えて、"text" と "empty" コンポーネントのクラスもあります。それらはテキストノードと、`null` をレンダーすると得られる「空のスロット」を表します。
- レンダラは依存性注入を利用して `host` 内部クラスをリコンサイラに渡します。例えば、React DOM はリコンサイラに `ReactDOMComponent` を `host` 内部インスタンスの実装として使用するように指示します。
- 子要素のリストを更新するロジックは `ReactMultiChild` と呼ばれるミックスインに抽出され、そのミックスインが、React DOM および React Native 両方における `host` 内部インスタンスのクラスの実装に使用されます。
- リコンサイラは `composite` 要素における `setState()` のサポートも実装しています。イベントハンドラ内部での複数の更新は、単一の更新にバッチ処理されます。
- リコンサイラは、`composite` コンポーネントおよび `host` ノードへの `ref` の追加と削除についても対応しています。
- DOM の準備ができあがった後に呼び出される、`componentDidMount()` や `componentDidUpdate()` のようなライフサイクルメソッドは `callback queues` に集められ、単一のバッチの中で実行されます。
- React は現時点での更新についての情報を「トランザクション ("transaction")」と呼ばれる内部オブジェクトに格納します。トランザクションは、保留中のライフサイクルメソッドのキューや、警告の際に使用する現在の DOM のネスト構造、そしてある特定の更新に対して「グローバル」になっているその他あらゆるものとの経過を追うのに重宝します。トランザクションによって React が更新後に「全てをクリーンアップする」よう保証できます。例えば、React DOM が提供するトランザクションクラスは、入力フィールドの選択状態を更新後に復元します。

## コードに飛び込む

- `ReactMount` はこのチュートリアルにある `mountTree()` や `unmountTree()` のようなコードがある場所です。ここでは最上位コンポーネントのマウントやアンマウントが行われます。`ReactNativeMount` はその React Native 版です。
- `ReactDOMComponent` はこのチュートリアルでの `DOMComponent` にあたります。これは、React DOM レンダラ向けの `host` コンポーネントクラスを実装するものです。`ReactNativeBaseComponent` はその React Native 版です。
- `ReactCompositeComponent` はこのチュートリアルでの `CompositeComponent` にあたります。これは、ユーザ定義コンポーネントの呼び出しとその `state` の保持を扱います。
- `instantiateReactComponent` はある要素に対して構築すべき正しい内部インスタンスクラスを選ぶスイッチを持っています。これは、このチュートリアルにおける `instantiateComponent()` にあたります。
- `ReactReconciler` は `mountComponent()`、`receiveComponent()`、そして `unmountComponent()` メソッドのラッパーです。これは水面下で内部インスタンスの実装を呼び出しますが、それらに追加するコードも含んでおり、その追加コードは全ての内部インスタンスの実装で共有されます。
- `ReactChildReconciler` は子要素を要素の `key` に基づいてマウント、更新、そしてアンマウントするロジックを実装しています。
- `ReactMultiChild` は、子要素の挿入、削除、そして移動の操作に関するキューの処理を、レンダラとは独立して実装します。

- `mount()` と `receive()`、そして `unmount()` は、実際の React のコードベースでは歴史的な理由から、`mountComponent()`、`receiveComponent()`、そして `unmountComponent()` と呼ばれていますが、これらは要素を受け取っています。
- 内部インスタンス上のプロパティ名は、`_currentElement` のようにアンダースコアから始まります。これらはコードベース全体を通じて、読み取り専用の `public` なフィールドと見なされます。

## 今後の方向性

stack リコンサイラには、同期的処理であることや、作業を中断したりチャンクに分割したりできないといったことなど、固有の制限があります。現在、[全く異なるアーキテクチャ](#)による[新たな Fiber リコンサイラ](#)の開発が進行中です。将来的には、stack リコンサイラをこれに置き換える予定ですが、現時点では同等の機能を提供するには程遠い状態です。

## 次のステップ

React の開発時に私たちが使用するガイドラインについて学ぶには、[次の章](#)を読んでください。

# 他のライブラリとのインテグレーション

React はどのような Web アプリケーションにも適用できます。React は他のアプリケーションに組み込むことができ、また少しだけ気を付ければ、React に他のアプリケーションを組み込むこともできます。ここでは一般的なユースケースである [jQuery](#) と [Backbone](#) を使った例を紹介しますが、同じ考え方はどのような既存のコードにも適用可能です。

## DOM 操作プラグインとのインテグレーション

React は、React 以外のものが DOM に加えた変更を認識しません。React は自身の内部表現に基づいて更新内容を決定します。もし同じ DOM ノードが別のライブラリによって操作された場合、React は混乱してしまい、回復する方法がありません。

とはいっても、React と操作プラグインを組み合わせることが不可能、あるいは必ずしも難しいと言っているではありません。それぞれがやっていることを正しく認識する必要があるのです。

コンフリクトを回避する最も簡単な方法は、React コンポーネントが更新されないようにすることです。これは、空の `<div />` のように、React から更新する理由がない要素をレンダーすることで実現できます。

### この問題への取り組み方法

上記の方法を実証するために、一般的な jQuery プラグインのラッパーを作ってみましょう。

まず、ルート DOM 要素へ `ref` をアタッチします。`componentDidMount` を使い、`ref` へのリファレンスを取得し、jQuery プラグインに渡します。

マウントの後に React が DOM をいじってしまうことを防ぐため、`render()` メソッドからは空の `<div />` 要素を返すようにします。この空の `<div />` 要素はプロパティや子要素を持たないので、React はそれを更新する理由がなく、jQuery プラグインの側が DOM のその部分を自由に管理できるのです：

```
class SomePlugin extends React.Component {
  componentDidMount() {
    this.$el = $(this.el);
    this.$el.somePlugin();
  }

  componentWillUnmount() {
    this.$el.somePlugin('destroy');
  }

  render() {
    return <div ref={el => this.el = el} />;
  }
}
```

`componentDidMount` と `componentWillUnmount` の両方の [ライフサイクルメソッド](#)を定義したことに注意してください。多くの jQuery プラグインは DOM にイベントリスナーをアタッチするので、それらを `componentWillUnmount` でデタッチすることが重要です。もしプラグインがクリーンアップの方法を提供していない場合、あなた自身で提供する必要があります。メモリリークを防ぐためにプラグインが登録したイベントリスナーを削除することを忘れないでください。

## jQuery Chosen プラグインとのインテグレーション

このアイディアの具体的な例として、`<select>` 要素を拡張する [Chosen](#) プラグインの最小のラッパーを書いてみましょう。

### 補足:

これが可能だからといって、これが React アプリケーションに最適なアプローチであるという意味ではありません。可能な限り React コンポーネントを使用することをお勧めします。React コンポーネントは React アプリケーションで簡単に再利用でき、また多くの場合、それらの動作や外観をより細かくコントロールできます。

まず、Chosen が DOM に対してどういった操作をしているのか確認しましょう。

`<select>` DOM ノードで Chosen を呼ぶと元の DOM ノードから属性を読み取り、インライン CSS で隠してから、`<select>` の直後に独自の視覚表現を持つ別の DOM ノードを追加します。その後に jQuery イベントを発生させて変更を通知します。

我々の `<Chosen>` というラッパー React コンポーネントで作成したい API は以下のようなものであるとしましょう：

```
function Example() {
  return (
    <Chosen onChange={value => console.log(value)}>
      <option>vanilla</option>
      <option>chocolate</option>
      <option>strawberry</option>
    </Chosen>
  );
}
```

わかりやすくするために[非制御コンポーネント](#)として実装します。

まず、`<select>` を `<div>` で囲んで返す `render()` メソッドを持った、空のコンポーネントを作成します：

```
class Chosen extends React.Component {
  render() {
    return (
      <div>
        <select className="Chosen-select" ref={el => this.el = el}>
          {this.props.children}
        </select>
      </div>
    );
  }
}
```

余分な `<div>` で `<select>` をラップしていることに注目してください。これが必要なのは、Chosen は渡された `<select>` の直後に別の DOM 要素を追加するからです。しかし、React からみれば、この `<div>` は常に 1 つの子要素しか持っていない。これにより React による更新が Chosen によって追加された DOM ノードと確実に競合しないようにできるのです。重要なことは、React フローの外側で DOM を変更する場合は、React がその DOM ノードに触る理由を確実になくす必要がある、ということです。

次に、ライフサイクルメソッドを実装しましょう。`componentDidMount` の中で `ref` 経由で渡された `<select>` ノードで `Chosen` を初期化する必要があります。そして `componentWillUnmount` でそれを破棄します：

```
componentDidMount() {
  this.$el = $(this.el);
  this.$el.chosen();
}

componentWillUnmount() {
  this.$el.chosen('destroy');
}
```

#### Try it on CodePen

React からすると `this.el` フィールドに特別な意味はありません。以前に `render()` メソッドの中で `ref` からこれに代入したことによって成り立っています：

```
<select className="Chosen-select" ref={el => this.el = el}>
```

コンポーネントをレンダーするにはこれで十分ですが、値の変更について通知を受ける必要があります。この通知を実現するために、`Chosen` が管理する `<select>` で起こる jQuery の `change` イベントを受け取るようにします。

コンポーネントの `props` (およびその一部として渡されるイベントハンドラ) は時間の経過とともに変わってしまう可能性があるので、`Chosen` に直接 `this.props.onChange` を渡さないようにします。代わりに、`this.props.onChange` を呼び出す `handleChange()` メソッドを宣言し、そちらを jQuery の `change` イベントのコールバックに設定します：

```
componentDidMount() {
  this.$el = $(this.el);
  this.$el.chosen();

  this.handleChange = this.handleChange.bind(this);
  this.$el.on('change', this.handleChange);
}

componentWillUnmount() {
  this.$el.off('change', this.handleChange);
  this.$el.chosen('destroy');
}

handleChange(e) {
  this.props.onChange(e.target.value);
}
```

#### Try it on CodePen

最後にもう 1 つ作業が残っています。React では `props` は時間と共に変更される可能性があります。例えば、親コンポーネントの `state` が変更されると、`<Chosen>` コンポーネントは異なる子を受け取るようになるかもしれません。つまり、React に DOM の管理を任せることはもうできないので、インテグレーションを行う部分では `props` の変更に応じて DOM を手動で更新することが重要です。

`Chosen` のドキュメントによると元の DOM 要素への変更について通知するための API として jQuery `trigger()` API が使えます。`<select>` 要素の中の `this.props.children` の更新に関しては React にやってもらいましょう。しかしました、`componentDidUpdate()` のライフサイクルメソッドを追加し、`Chosen` に変更を通知する必要があります：

```

componentDidUpdate(prevProps) {
  if (prevProps.children !== this.props.children) {
    this.$el.trigger("chosen:updated");
  }
}

```

このようにして Chosen は、React が管理する `<select>` の子要素に変更があった場合に自分が管理する DOM 要素を変更すべき事が分かるようになります。

`Chosen` コンポーネントの完全な実装は以下のようになります：

```

class Chosen extends React.Component {
  componentDidMount() {
    this.$el = $(this.el);
    this.$el.chosen();

    this.handleChange = this.handleChange.bind(this);
    this.$el.on('change', this.handleChange);
  }

  componentDidUpdate(prevProps) {
    if (prevProps.children !== this.props.children) {
      this.$el.trigger("chosen:updated");
    }
  }

  componentWillUnmount() {
    this.$el.off('change', this.handleChange);
    this.$el.chosen('destroy');
  }

  handleChange(e) {
    this.props.onChange(e.target.value);
  }

  render() {
    return (
      <div>
        <select className="Chosen-select" ref={el => this.el = el}>
          {this.props.children}
        </select>
      </div>
    );
  }
}

```

[Try it on CodePen](#)

## 他のビューライブラリとのインテグレーション

React は `createRoot()` の柔軟性のおかげで、他のアプリケーションに組み込むことができます。

React は一般的に起動時に単一のルート React コンポーネントを DOM にロードして使用されるのですが、`createRoot()` はボタンのような小さなものからアプリケーション全体に至るまで、独立した UI のパートに対して複数回呼び出すこともできます。

実際、これはまさに React が Facebook で使用されている方法でもあります。これにより React でアプリケーションを少しずつ作成し、それらを既存のサーバ側テンプレートやその他のクライアントサイドコードと組み合わせることができます。

## React で文字列ベースのレンダーを置き換える

古いウェブアプリケーションによくあるパターンは、DOM のまとまりを文字列として記述して、`$el.html(htmlString)` のような形で DOM に挿入することです。このような箇所は React の導入にぴったりです。文字列ベースのレンダーを React コンポーネントに置き換えるだけで良いのです。

つまり、次のような jQuery による実装は

```
$('#container').html('<button id="btn">Say Hello</button>');
$('#btn').click(function() {
  alert('Hello!');
});
```

React コンポーネントを使用して次のように書き換えられます：

```
function Button() {
  return <button id="btn">Say Hello</button>;
}

$('#btn').click(function() {
  alert('Hello!');
});
```

ここから始めて、コンポーネントにロジック部分を更に移植していくことや、より一般的な React のプラクティスを採用していくことができます。例えば、コンポーネントでは同じコンポーネントが複数回レンダーされる可能性があるので、ID に依存しないことがベストプラクティスです。かわりに React の[イベントの仕組み](#)を使用してクリックハンドラを React の `<button>` 要素に直接登録します：

```
function Button(props) {
  return <button onClick={props.onClick}>Say Hello</button>;
}

function HelloButton() {
  function handleClick() {
    alert('Hello!');
  }
  return <Button onClick={handleClick} />;
}
```

### Try it on CodePen

このような分離されたコンポーネントを好きなだけ持つことができ、`ReactDOM.createRoot()` を使用して異なる DOM コンテナにそれらをレンダーすることができます。アプリケーションを少しずつ React に変換していくにつれて、より大きなコンポーネントへとインテグレーションできるようになり、`ReactDOM.createRoot()` の呼び出しを階層の上方へ移動させていく様になるでしょう。

## Backbone View に React を組み込む

`Backbone` view は通常、HTML 文字列、もしくは文字列を生成するテンプレート用関数を使って、DOM 要素の中身を作成します。この処理もまた React コンポーネントのレンダーに置き換えられます。

以下で、`ParagraphView` と呼ばれる Backbone view を作成します。Backbone の `render()` 関数をオーバーライドして、React の `<Paragraph>` コンポーネントを Backbone が提供する DOM 要素 (`this.el`) にレンダーします。ここでも `ReactDOM.createRoot()` を使用します：

```
function Paragraph(props) {
  return <p>{props.text}</p>;
}

const ParagraphView = Backbone.View.extend({
  initialize(options) {
    this.reactRoot = ReactDOM.createRoot(this.el);
  },
  render() {
    const text = this.model.get('text');
    this.reactRoot.render(<Paragraph text={text} />);
    return this;
  },
  remove() {
    this.reactRoot.unmount();
    Backbone.View.prototype.remove.call(this);
  }
});
```

[Try it on CodePen](#)

`remove` メソッド内で `root.unmount()` を呼び出して、コンポーネントツリーがデタッチされた際にイベントハンドラとコンポーネントツリーに関連付けられていたその他のリソースを React が解除することも重要です。

React ツリー内からコンポーネントが削除されるとクリーンアップは自動的に実行されますが、ツリー全体を手動で削除するため、このメソッドを呼び出す必要があります。

## Model 層とのインテグレーション

一般的には React の state、Flux、もしくは Redux のような一方向のデータフローの使用が推奨されますが、React コンポーネントは他のフレームワークやライブラリのモデル層を利用することができます。

### React コンポーネントで Backbone Model を使用する

React コンポーネントから Backbone のモデルとコレクションを利用する最もシンプルな方法は、様々な変更イベントを監視して手動で強制的に更新することです。

モデルのレンダーに責任を持つコンポーネントは `'change'` イベントを監視し、コレクションのレンダーに責任を持つコンポーネントは `'add'` および `'remove'` イベントを監視します。どちらの場合も、`this.forceUpdate()` を呼び出して新しいデータでコンポーネントを再レンダーします。

以下の例では、`List` コンポーネントは Backbone のコレクションをレンダーします。個別の要素のレンダーには `Item` コンポーネントを使用します。

```

class Item extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
  }

  handleChange() {
    this.forceUpdate();
  }

  componentDidMount() {
    this.props.model.on('change', this.handleChange);
  }

  componentWillUnmount() {
    this.props.model.off('change', this.handleChange);
  }

  render() {
    return <li>{this.props.model.get('text')}</li>;
  }
}

class List extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
  }

  handleChange() {
    this.forceUpdate();
  }

  componentDidMount() {
    this.props.collection.on('add', 'remove', this.handleChange);
  }

  componentWillUnmount() {
    this.props.collection.off('add', 'remove', this.handleChange);
  }

  render() {
    return (
      <ul>
        {this.props.collection.map(model => (
          <Item key={model.cid} model={model} />
        ))}
      </ul>
    );
  }
}

```

[Try it on CodePen](#)

## Backbone Model からデータを抽出する

上記のアプローチでは React コンポーネントが Backbone のモデルとコレクションを認識することが必要です。後で別のデータ管理ソリューションに移行する予定がある場合は、Backbone とのインテグレーション箇所はできるだけコードの一部に集中させることをお勧めします。

1つの解決法は、変更があるたびにモデルの属性をプレーンなデータとして抽出するようにし、そのロジックを一箇所だけで保持することです。以下のコードは Backbone モデルの全ての属性を state へと抽出し、そのデータをラップされるコンポーネントへと渡す高階コンポーネントです。

このようにすれば、高階コンポーネントだけが Backbone モデルの内部動作について知っておく必要があり、アプリケーション内のほとんどのコンポーネントは Backbone について知らないままでいられるのです。

以下の例では、モデルの属性のコピーを state の初期値に設定します。change イベントを監視して（アンマウント時には監視を解除し）、変更が発生した際には state をモデルの現在の属性で更新します。最後に、model プロパティそのものが変更となった場合には、必ず忘れずに古いモデルの監視を解除し、新しいモデルから情報を受け取るようにします。

この例は Backbone と協調して動作させるための網羅的なものとして書かれているわけではないことに注意すべきですが、この種の問題にどうアプローチすべきかの一般的な理解の助けになるはずです：

```
function connectToBackboneModel(WrappedComponent) {
  return class BackboneComponent extends React.Component {
    constructor(props) {
      super(props);
      this.state = Object.assign({}, props.model.attributes);
      this.handleChange = this.handleChange.bind(this);
    }

    componentDidMount() {
      this.props.model.on('change', this.handleChange);
    }

    componentWillReceiveProps(nextProps) {
      this.setState(Object.assign({}, nextProps.model.attributes));
      if (nextProps.model !== this.props.model) {
        this.props.model.off('change', this.handleChange);
        nextProps.model.on('change', this.handleChange);
      }
    }

    componentWillUnmount() {
      this.props.model.off('change', this.handleChange);
    }

    handleChange(model) {
      this.setState(model.changedAttributes());
    }

    render() {
      const propsExceptModel = Object.assign({}, this.props);
      delete propsExceptModel.model;
      return <WrappedComponent {...propsExceptModel} {...this.state} />;
    }
  }
}
```

このコードの使い方を例示するために、NameInput という React コンポーネントを Backbone モデルと接続して、入力が変更されるたびに firstName 属性を更新します：

```
function NameInput(props) {
  return (
    <p>
      <input value={props.firstName} onChange={props.handleChange} />
    </p>
  );
}
```

```
<br />
    My name is {props.firstName}.
</p>
);
}

const BackboneNameInput = connectToBackboneModel(NameInput);

function Example(props) {
  function handleChange(e) {
    props.model.set('firstName', e.target.value);
  }

  return (
    <BackboneNameInput
      model={props.model}
      handleChange={handleChange}
    />
  );
}

const model = new Backbone.Model({ firstName: 'Frodo' });
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Example model={model} />);
```

### Try it on [CodePen](#)

この手法は Backbone だけに限ったものではありません。ライフサイクルメソッドで変更を購読し必要に応じてデータをローカルの React 状態にコピーすることで、任意のモデルライブラリで React を使用できます。

# JSX の導入

以下の変数宣言を考えてみましょう：

```
const element = <h1>Hello, world!</h1>;
```

このおかしなタグ構文は文字列でも HTML でもありません。

これは JSX と呼ばれる JavaScript の構文の拡張です。UI がどのような見た目かを記述するために、Reactとともに JSX を使用することをお勧めしています。JSX はテンプレート言語を連想させるでしょうが、JavaScript の機能を全て備えたものです。

JSX は React "要素" を生成します。次の章で React 要素を DOM に変換する方法について見ていきます。以下では、JSX を使い始めるのに必要な基礎を学ぶことができます。

## JSXを使う理由

表示のためのロジックは、イベントへの応答や経時的な状態の変化、画面表示のためのデータを準備する方法といった、他の UI ロジックと本質的に結合したものであり、React はその事実を受け入れます。

マークアップとロジックを別々のファイルに書いて人為的に技術を分離するのではなく、React はマークアップとロジックを両方含む疎結合の「コンポーネント」という単位を用いて関心を分離します。後のセクションでコンポーネントについては改めて詳しく紹介しますが、現時点で JavaScript にマークアップを書くことが気にくわない場合、こちらの議論で考えが改まるかもしれません。

React で JSX を使うことは必須ではありませんが、ほとんどの人は JavaScript コードの中で UI を扱う際に、JSX を見た目に有用なものだと感じています。また、JSX があるために React は有用なエラーや警告をより多く表示することができます。

前置きはこのくらいにして、早速始めましょう。

## JSX に式を埋め込む

以下の例では、`name` という変数を宣言して、それを中括弧に囲んで JSX 内で使用しています。

```
const name = 'Josh Perez';
const element = <h1>Hello, {name}</h1>;
```

あらゆる有効な [JavaScript の式](#) を JSX 内で中括弧に囲んで使用できます。例えば、`2 + 2` や `user.firstName` や `formatName(user)` はすべて有効な JavaScript の式です。

以下の例では、`formatName(user)` という JavaScript 関数の結果を `<h1>` 要素内に埋め込んでいます。

```
function formatName(user) {
  return user.firstName + ' ' + user.lastName;
}

const user = {
  firstName: 'Harper',
  lastName: 'Perez'
```

```

};

const element = (
  <h1>
    Hello, {formatName(user)}!
  </h1>
);

```

### Try it on CodePen

読みやすさのため JSX を複数行に分けています。必須ではありませんが、複数行に分割する場合には、[自動セミコロン挿入](#)の落とし穴にはまらないように括弧で囲むことをおすすめします

## JSX もまた式である

コンパイルの後、JSX の式は普通の JavaScript の関数呼び出しに変換され、JavaScript オブジェクトへと評価されます。

これは JSX を `if` 文や `for` ループの中で使用したり、変数に代入したり、引数として受け取ったり、関数から返したりすることができるということです。

```

function getGreeting(user) {
  if (user) {
    return <h1>Hello, {formatName(user)}!</h1>;
  }
  return <h1>Hello, Stranger.</h1>;
}

```

## JSX で属性を指定する

文字列リテラルを属性として指定するために引用符を使用できます。

```
const element = <a href="https://www.reactjs.org"> link </a>;
```

属性に JavaScript 式を埋め込むために中括弧を使用することもできます。

```
const element = <img src={user.avatarUrl}></img>;
```

属性に JavaScript 式を埋め込む時に中括弧を囲む引用符を書かないでください。（文字列の場合は）引用符、もしくは（式の場合は）中括弧のどちらか一方を使用するようにし、同じ属性に対して両方を使用するべきではありません。

### 警告:

JSX は HTML よりも JavaScript に近いものなので、React DOM は HTML の属性ではなくキャメルケース (camelCase) のプロパティ命名規則を使用します。

JSX では例えば、`class` は `className` となり、`tabindex` は `tabIndex` となります。

## JSX で子要素を指定する

タグが空の場合、XML のように `/>` でタグを閉じる事ができます：

```
const element = <img src={user.avatarUrl} />;
```

JSX のタグは子要素を持つことができます：

```
const element = (
  <div>
    <h1>Hello!</h1>
    <h2>Good to see you here.</h2>
  </div>
);
```

## JSX はインジェクション攻撃を防ぐ

JSX にユーザの入力を埋め込むことは安全です：

```
const title = response.potentiallyMaliciousInput;
// This is safe:
const element = <h1>{title}</h1>;
```

デフォルトでは、React DOM は JSX に埋め込まれた値をレンダー前にエスケープします。このため、自分のアプリケーションで明示的に書かれたものではないあらゆるコードは、注入できないことが保証されます。レンダーの前に全てが文字列に変換されます。これは [XSS \(cross-site-scripting\)](#) 攻撃の防止に役立ちます。

## JSX はオブジェクトの表現である

Babel は JSX を `React.createElement()` の呼び出しへとコンパイルします。

以下の 2 つの例は等価です：

```
const element = (
  <h1 className="greeting">
    Hello, world!
  </h1>
);
```

```
const element = React.createElement(
  'h1',
  {className: 'greeting'},
  'Hello, world!'
);
```

`React.createElement()` はバグの混入を防止するためにいくつかのチェックも行いますが、本質的には以下のようなオブジェクトを生成します：

```
// Note: this structure is simplified
const element = {
  type: 'h1',
  props: {
    className: 'greeting',
  }
};
```

```
    children: 'Hello, world!'  
};  
};
```

このようなオブジェクトは "React 要素" と呼ばれます。これらは画面に表示したいものの説明書きとして考えることができます。React はこれらのオブジェクトを読み取り、DOM を構築して最新に保ちます。

[次の章](#)では React 要素を DOM に変換することについて見てきましょう。

**ヒント:**

利用したいエディタで "[Babel](#)" 言語定義を使用して、ES6 と JSX が適切にハイライトされるようにしておくことをお勧めします。

# JSX を深く理解する

JSX とは、つまるところ `React.createElement(component, props, ...children)` の糖衣構文にすぎません。例として、次の JSX コードを見てみましょう。

```
<MyButton color="blue" shadowSize={2}>
  Click Me
</MyButton>
```

これは以下のようにコンパイルされます。

```
React.createElement(
  MyButton,
  {color: 'blue', shadowSize: 2},
  'Click Me'
)
```

子要素を持たない場合には、自己クローズ (self-closing) タグを利用することもできます。次のコードを見てください。

```
<div className="sidebar" />
```

これは以下のようにコンパイルされます。

```
React.createElement(
  'div',
  {className: 'sidebar'}
)
```

具体的に JSX がどのように JavaScript へ変換されるのかをテストしたい場合は、[オンライン Babel コンパイラ](#)で試すことができます。

## React 要素の型を指定する

JSX タグの先頭の部分は、React 要素の型を決定しています。

大文字で始まる型は JSX タグが React コンポーネントを参照していることを示しています。このような JSX タグはコンパイルを経てその大文字で始まる変数を直接参照するようになります。つまり JSX の `<Foo />` 式を使用する場合、`Foo` がスコープになければなりません。

## React がスコープ内にあること

JSX は `React.createElement` の呼び出しへとコンパイルされるため、`React` ライブラリは常に JSX コードのスコープ内にある必要があります。

例えば以下のコードでは、`React` も `CustomButton` も JavaScript から直接は参照されていませんが、両方ともインポートされていることが必要です。

```
import React from 'react';
import CustomButton from './CustomButton';
```

```
function WarningButton() {
  // return React.createElement(CustomButton, {color: 'red'}, null);
  return <CustomButton color="red" />;
}
```

JavaScript のバンドルツールを使わずに `<script>` タグから React を読み込んでいる場合は、`React` はグローバル変数として既にスコープに入っています。

## JSX 型にドット記法を使用する

JSX の中においては、ドット記法を使うことによって React コンポーネントを参照することもできます。これは単一のモジュールがたくさんの React コンポーネントをエクスポートしているような場合に便利です。例えば、`MyComponents.DatePicker` というコンポーネントがあるのであれば、次のように JSX 内から直接利用することができます。

```
import React from 'react';

const MyComponents = {
  DatePicker: function DatePicker(props) {
    return <div>Imagine a {props.color} datepicker here.</div>;
  }
}

function BlueDatePicker() {
  return <MyComponents.DatePicker color="blue" />;
}
```

## ユーザ定義のコンポーネントの名前は大文字で始めること

ある要素の型が小文字から始まっているような場合、それは `<div>` や `<span>` のような組み込みのコンポーネントを参照しており、これらはそれぞれ `'div'` や `'span'` といった文字列に変換されて `React.createElement` に渡されます。一方で `<Foo />` のように大文字で始まる型は `React.createElement(Foo)` にコンパイルされ、JavaScript ファイルにおいて定義あるいはインポートされたコンポーネントを参照します。

コンポーネントを命名するときには、大文字から始めるようにしてください。もしすでに小文字から始まるコンポーネントを作ってしまっていたら、JSX 内で利用する前にいちど大文字から始まる変数に代入しておきましょう。

例えば、以下のコードは期待通りには動きません。

```
import React from 'react';

// 間違った例。これはコンポーネントなので、大文字ではじめなければいけません。
function hello(props) {
  // 正しい例。div は HTML タグなので、<div> と書くのは正解です。
  return <div>Hello {props.toWhat}</div>;
}

function HelloWorld() {
  // 間違った例。大文字ではじまついていないため、React は <hello /> を HTML タグと認識してしまいます。
  return <hello toWhat="World" />;
}
```

`hello` を `Hello` に書き換え、`<Hello />` を使って参照するようすれば、このコードはきちんと動作するようになります。

```
import React from 'react';

// 正しい例。コンポーネントなので大文字からはじまっています。
function Hello(props) {
  // 正しい例。div は HTML タグなので、<div> と書くのは正解です。
  return <div>Hello {props.toWhat}</div>;
}

function HelloWorld() {
  // 正しい例。大文字で始まっているため、React は <Hello /> がコンポーネントだと認識できます。
  return <Hello toWhat="World" />;
}
```

## 実行時に型を選択する

一般的な式を React の要素の型として使用することはできません。どうしても一般的な式を使って要素の型を示したいのであれば、まずその式を大文字から始まる変数に代入してから利用しましょう。これはプロパティの値に応じて異なるコンポーネントを表示し分けたいような場合によくあるケースです。

```
import React from 'react';
import { PhotoStory, VideoStory } from './stories';

const components = {
  photo: PhotoStory,
  video: VideoStory
};

function Story(props) {
  // 間違った例。式は JSX の型に指定できません。
  return <components[props.storyType] story={props.story} />;
}
```

大文字から始まる変数に型を代入することで、上のコードをきちんと動作するようにしてみましょう。

```
import React from 'react';
import { PhotoStory, VideoStory } from './stories';

const components = {
  photo: PhotoStory,
  video: VideoStory
};

function Story(props) {
  // 正しい例。大文字で始まる変数は JSX の型に指定できます。
  const SpecificStory = components[props.storyType];
  return <SpecificStory story={props.story} />;
}
```

## JSX における props

JSX で `props` を指定するやり方はいくつかあります。

## プロパティとしての JavaScript 式

任意の JavaScript 式は `{}` で囲むことによって `props` として渡すことができます。例として次の JSX を見てみましょう。

```
<MyComponent foo={1 + 2 + 3 + 4} />
```

この `MyComponent` について、`props.foo` の値は `10`、つまり `1 + 2 + 3 + 4` という式が評価されます。

`if` 文や `for` 文は JavaScriptにおいては式ではないため、JSX 内で直接利用することはできません。代わりに JSX の近くで間接的に利用してみてください。次がその例です。

```
function NumberDescriber(props) {
  let description;
  if (props.number % 2 == 0) {
    description = <strong>even</strong>;
  } else {
    description = <i>odd</i>;
  }
  return <div>{props.number} is an {description} number</div>;
}
```

これについては、[条件付きレンダーとループ](#)でさらに深く学ぶことができます。

## 文字列リテラル

文字列リテラルを `props` として渡すことができます。そのため以下のふたつの JSX の式はまったく等しいものとなります。

```
<MyComponent message="hello world" />
<MyComponent message={'hello world'} />
```

文字列リテラルを渡す際、その値における HTML エスケープは元の形に復元されます。そのため以下のふたつの JSX の式もまったく等しいものとなります。

```
<MyComponent message=&lt;3&gt; />
<MyComponent message={'<3'} />
```

この振る舞いは多くの場合それほど重要なものではありませんが、包括的な解説の一環としてここで触れておきます。

## プロパティのデフォルト値は `true`

プロパティに値を与えない場合、デフォルトの値は `true` となります。そのため以下のふたつの JSX の式はまったく等しいものとなります。

```
<MyTextBox autocomplete />
<MyTextBox autocomplete={true} />
```

特別な理由がある場合を除いて、このように値を省略することは推奨していません。ES6 におけるオブジェクトの簡略表記においては、`{foo}` は `{foo: true}` ではなく `{foo: foo}` を意味するため、HTML の動作に似せて作られたこの機能はかえって混乱をきたす可能性があります。

## 属性の展開

`props` オブジェクトがあらかじめ存在しており、それを JSX に渡したいような場合は「スプレッド」構文である `...` を使用することで、`props` オブジェクトそのものを渡すことができます。そのため以下のふたつの JSX の式はまったく等しいものとなります。

```
function App1() {
  return <Greeting firstName="Ben" lastName="Hector" />;
}

function App2() {
  const props = {firstName: 'Ben', lastName: 'Hector'};
  return <Greeting {...props} />;
}
```

また、コンポーネントが利用する適当なプロパティを取り出しつつ、残りのすべてのプロパティに対してスプレッド構文を利用することもできます。

```
const Button = props => {
  const { kind, ...other } = props;
  const className = kind === "primary" ? "PrimaryButton" : "SecondaryButton";
  return <button className={className} {...other} />;
};

const App = () => {
  return (
    <div>
      <Button kind="primary" onClick={() => console.log("clicked!")}>
        Hello World!
      </Button>
    </div>
  );
};
```

上の例では、`kind` プロパティは無事に取り出され、DOM 中の `<button>` 要素には渡されていません。残りのプロパティは `...other` オブジェクトにより渡され、このコンポーネントを柔軟性の高いものにしています。上記のコードは `onClick` や `children` プロパティを渡していることが見てとれるはずです。

スプレッド演算子は便利ではありますが、コンポーネント内で利用しないプロパティを不用意に渡してしまったり、意味をなさない HTML 属性を DOM に渡してしまうようなことが容易に起きえます。そのためこの構文は慎重に利用してください。

## JSX における子要素

開始タグと終了タグの両方を含む JSX 式においては、タグに囲まれた部分は、`props.children` という特別なプロパティとして渡されます。このような子要素を渡す方法はいくつかあります。

## 文字列リテラル

開始タグと終了タグの間に文字列を挟んでいる場合、その文字列が `props.children` となります。これは HTML 要素を JSX 内で利用するような場合よくあるケースです。次の例を見てください。

```
<MyComponent>Hello world!</MyComponent>
```

この JSX は正しく動作します。この場合 `props.children` は `MyComponent` において、単なる文字列 `"Hello world!"` となります。HTML エスケープは元の文字列に復元されるため、多くの場合は以下のように HTML を書くように JSX を書くことができます。

```
<div>This is valid HTML && JSX at the same time.</div>
```

JSX は行の先頭と末尾の空白文字を削除し、また空白行も削除します。タグに隣接する改行も削除され、文字列リテラル内での改行は 1 つの空白文字に置き換えられます。そのため以下の例はすべて同じものを表示します。

```
<div>Hello World</div>

<div>
  Hello World
</div>

<div>
  Hello
  World
</div>

<div>
  Hello World
</div>
```

## 子要素としての JSX 要素

JSX 要素を子要素として渡すこともできます。これはネストしたコンポーネントを表示したいときに活用することができます。

```
<MyContainer>
  <MyFirstComponent />
  <MySecondComponent />
</MyContainer>
```

異なる型の子要素を混在させることができるために、文字列リテラルを JSX 要素と一緒に子要素として渡すことができます。この点においても JSX と HTML は似ており、次のような例は JSX としても HTML としても正しく動作します。

```
<div>
  Here is a list:
  <ul>
    <li>Item 1</li>
    <li>Item 2</li>
  </ul>
</div>
```

また React コンポーネントは要素の配列を返すこともできます。

```
render() {
  // リスト化するだけのために要素を用意する必要はありません！
  return [
    // key 属性を書き忘れないようにしてください :)
    <li key="A">First item</li>,
    <li key="B">Second item</li>,
    <li key="C">Third item</li>,
  ];
}
```

## 子要素としての JavaScript 式

任意の JavaScript の式は `{ }` で囲むことによって子要素として渡すことができます。そのため以下の JSX の式はまったく等しいものとなります。

```
<MyComponent>foo</MyComponent>
<MyComponent>{'foo'}</MyComponent>
```

これは長さの決まっていない JSX 式のリストを表示したいような場合に特に便利に使うことができます。次の例は HTML の表を表示します。

```
function Item(props) {
  return <li>{props.message}</li>;
}

function TodoList() {
  const todos = ['finish doc', 'submit pr', 'nag dan to review'];
  return (
    <ul>
      {todos.map((message) => <Item key={message} message={message} />)}
    </ul>
  );
}
```

JavaScript の式は異なる型の子要素と併用することができるため、テンプレートリテラルの代わりに次のような書き方をすることもできます。

```
function Hello(props) {
  return <div>Hello {props.addressee}!</div>;
}
```

## 子要素としての関数

JSX タグに挟まれた JavaScript 式は、通常は文字列や React 要素、あるいはこれらのリストとして評価されます。しかし、他のあらゆる `props` と同様に `props.children` にも任意のデータを渡すことができ、そのデータとは必ずしも React がレンダーできるものに限りません。例えば独自コンポーネントに `props.children` を通してコールバックを定義することもできるのです。

```
// numTimes の数だけ子要素のコールバックを呼び出し、コンポーネントを繰り返し作成する
function Repeat(props) {
  let items = [];
  for (let i = 0; i < props.numTimes; i++) {
    items.push(props.children(i));
  }
}
```

```

    return <div>{items}</div>;
}

function ListOfTenThings() {
  return (
    <Repeat numTimes={10}>
      {({index}) => <div key={index}>This is item {index} in the list</div>}
    </Repeat>
  );
}

```

独自コンポーネントに渡される子要素は、レンダーが実行されるまでに React が理解できる要素に変換されている限りにおいては、どのようなものでも構いません。このようなやり方は一般的ではありませんが、JSX をさらに拡張したくなつた時には活用してみてください。

## 真偽値、null、undefined は無視される

`true` と `false`、`null`、そして `undefined` は子要素として渡すことができます。これらは何もレンダーしません。以下の JSX の式はすべて同じ結果となります。

```

<div />

<div></div>

<div>{false}</div>

<div>{null}</div>

<div>{undefined}</div>

<div>{true}</div>

```

これは条件に応じて React 要素を表示する際に活用できます。次の例は `showHeader` が `true` のときに限って `<Header />` が表示されます。

```

<div>
  {showHeader && <Header />}
  <Content />
</div>

```

1 つ注意点として、数値 `0` のように "falsy" な（偽と評価される）いくつかの値は、React によって表示されます。つまり次のコードは `props.messages` が空の配列のときには `0` が表示されてしまうため、期待通りには動作しません。

```

<div>
  {props.messages.length &&
    <MessageList messages={props.messages} />
  }
</div>

```

`&&` の前の式が必ず真偽値となるようにすれば、期待通りに動作します。

```

<div>
  {props.messages.length > 0 &&
    <MessageList messages={props.messages} />
  }
</div>

```

反対に、`false`、`true`、`null`、または`undefined`といった値を表示したいのであれば、まず文字列に変換する必要があります。

```
<div>
  My JavaScript variable is {String(myVariable)}.
</div>
```

---

[Go to TOC](#)

# Legacy Context

Note:

The legacy context API will be removed in a future major version. Use the [new context API](#) introduced with version 16.3. The legacy API will continue working for all 16.x releases.

## How To Use Context

This section documents a legacy API. See the [new API](#).

Suppose you have a structure like:

```
class Button extends React.Component {
  render() {
    return (
      <button style={{background: this.props.color}}>
        {this.props.children}
      </button>
    );
  }
}

class Message extends React.Component {
  render() {
    return (
      <div>
        {this.props.text} <Button color={this.props.color}>Delete</Button>
      </div>
    );
  }
}

class MessageList extends React.Component {
  render() {
    const color = "purple";
    const children = this.props.messages.map((message) =>
      <Message text={message.text} color={color} />
    );
    return <div>{children}</div>;
  }
}
```

In this example, we manually thread through a `color` prop in order to style the `Button` and `Message` components appropriately. Using context, we can pass this through the tree automatically:

```
import PropTypes from 'prop-types';

class Button extends React.Component {
```

```

render() {
  return (
    <button style={{background: this.context.color}}>
      {this.props.children}
    </button>
  );
}

Button.contextTypes = {
  color: PropTypes.string
};

class Message extends React.Component {
  render() {
    return (
      <div>
        {this.props.text} <Button>Delete</Button>
      </div>
    );
  }
}

class MessageList extends React.Component {
  getChildContext() {
    return {color: "purple"};
  }

  render() {
    const children = this.props.messages.map((message) =>
      <Message text={message.text} />
    );
    return <div>{children}</div>;
  }
}

MessageList.childContextTypes = {
  color: PropTypes.string
};

```

By adding `childContextTypes` and `getChildContext` to `MessageList` (the context provider), React passes the information down automatically and any component in the subtree (in this case, `Button`) can access it by defining `contextTypes`.

If `contextTypes` is not defined, then `context` will be an empty object.

#### Note:

`React.PropTypes` has moved into a different package since React v15.5. Please use [the prop-types library instead](#) to define `contextTypes`.

We provide a [codemod script](#) to automate the conversion.

## Parent-Child Coupling

This section documents a legacy API. See the [new API](#).

Context can also let you build an API where parents and children communicate. For example, one library that works this way is [React Router V4](#):

```
import { BrowserRouter as Router, Route, Link } from 'react-router-dom';

const BasicExample = () => (
  <Router>
    <div>
      <ul>
        <li><Link to="/">Home</Link></li>
        <li><Link to="/about">About</Link></li>
        <li><Link to="/topics">Topics</Link></li>
      </ul>

      <hr />

      <Route exact path="/" component={Home} />
      <Route path="/about" component={About} />
      <Route path="/topics" component={Topics} />
    </div>
  </Router>
);
```

By passing down some information from the `Router` component, each `Link` and `Route` can communicate back to the containing `Router`.

Before you build components with an API similar to this, consider if there are cleaner alternatives. For example, you can pass entire React components as props if you'd like to.

## Referencing Context in Lifecycle Methods

This section documents a legacy API. See the [new API](#).

If `contextTypes` is defined within a component, the following [lifecycle methods](#) will receive an additional parameter, the `context` object:

- `constructor(props, context)`
- `componentWillReceiveProps(nextProps, nextContext)`
- `shouldComponentUpdate(nextProps, nextState, nextContext)`
- `componentWillUpdate(nextProps, nextState, nextContext)`

Note:

As of React 16, `componentDidUpdate` no longer receives `prevContext`.

## Referencing Context in Function Components

This section documents a legacy API. See the [new API](#).

Function components are also able to reference `context` if `contextTypes` is defined as a property of the function. The following code shows a `Button` component written as a function component.

```
import PropTypes from 'prop-types';

const Button = ({children}, context) =>
  <button style={{background: context.color}}>
    {children}
  </button>

Button.contextTypes = {color: PropTypes.string};
```

## Updating Context

This section documents a legacy API. See the [new API](#).

Don't do it.

React has an API to update context, but it is fundamentally broken and you should not use it.

The `getChildContext` function will be called when the state or props changes. In order to update data in the context, trigger a local state update with `this.setState`. This will trigger a new context and changes will be received by the children.

```
import PropTypes from 'prop-types';

class MediaQuery extends React.Component {
  constructor(props) {
    super(props);
    this.state = {type: 'desktop'};
  }

  getChildContext() {
    return {type: this.state.type};
  }

  componentDidMount() {
    const checkMediaQuery = () => {
      const type = window.matchMedia("(min-width: 1025px)").matches ? 'desktop' :
        'mobile';
      if (type !== this.state.type) {
        this.setState({type});
      }
    };
    checkMediaQuery();
    window.addEventListener('resize', checkMediaQuery);
  }
}
```

```
};

    window.addEventListener('resize', checkMediaQuery);
    checkMediaQuery();
}

render() {
    return this.props.children;
}

MediaQuery.childContextTypes = {
    type: PropTypes.string
};
```

The problem is, if a context value provided by component changes, descendants that use that value won't update if an intermediate parent returns `false` from `shouldComponentUpdate`. This is totally out of control of the components using context, so there's basically no way to reliably update the context. [This blog post](#) has a good explanation of why this is a problem and how you might get around it.

# イベントブーリング

## 補足

このページの内容は React 16 以前および React Native にのみ関連します。

ウェブで使う React 17 ではイベントブーリングは**使用されていません**。

React 17 におけるこの変更についての[詳細はこちら](#)。

`SyntheticEvent` オブジェクトはプールされます。つまり `SyntheticEvent` はイベントハンドラが呼び出された後に再利用され、すべてのプロパティが `null` にセットされます。例えば、以下は動作しません：

```
function handleChange(e) {
  // This won't work because the event object gets reused.
  setTimeout(() => {
    console.log(e.target.value); // Too late!
  }, 100);
}
```

イベントハンドラが実行された後にオブジェクトのプロパティにアクセスする必要がある場合は、`e.persist()` を呼ぶ必要があります：

```
function handleChange(e) {
  // Prevents React from resetting its properties:
  e.persist();

  setTimeout(() => {
    console.log(e.target.value); // Works
  }, 100);
}
```

---

[Go to TOC](#)

# state のリフトアップ

しばしば、いくつかのコンポーネントが同一の変化するデータを反映する必要がある場合があります。そんなときは最も近い共通の祖先コンポーネントへ共有されている state をリフトアップすることを推奨します。これを、実際にはどのように行うかを見てみましょう。

この章では、与えられた温度で水が沸騰するかどうかを計算する温度計算ソフトを作成します。

`BoilingVerdict` というコンポーネントから始めましょう。これは温度を `celsius` という props として受け取り、水が沸騰するのに十分な温度かどうかを表示します。

```
function BoilingVerdict(props) {
  if (props.celsius >= 100) {
    return <p>The water would boil.</p>;
  }
  return <p>The water would not boil.</p>;
}
```

次に `Calculator` と呼ばれるコンポーネントを作成します。温度を入力するための `<input>` 要素をレンダーし、入力された値を `this.state.temperature` に保持します。

加えて、現在の入力値を判定する `BoilingVerdict` もレンダーします。

```
class Calculator extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {temperature: ''};
  }

  handleChange(e) {
    this.setState({temperature: e.target.value});
  }

  render() {
    const temperature = this.state.temperature;
    return (
      <fieldset>
        <legend>Enter temperature in Celsius:</legend>
        <input
          value={temperature}
          onChange={this.handleChange} />
        <BoilingVerdict
          celsius={parseFloat(temperature)} />
      </fieldset>
    );
  }
}
```

[Try it on CodePen](#)

## 2つ目の入力を追加する

新しい要件は、摂氏の入力に加えて、華氏の入力もできるようにして、それらを同期させておくことです。

`Calculator` から `TemperatureInput` コンポーネントを抽出するところから始めましょう。props として、`"c"` もしくは `"f"` の値をとる `scale` を新しく追加します：

```
const scaleNames = {
  c: 'Celsius',
  f: 'Fahrenheit'
};

class TemperatureInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {temperature: ''};
  }

  handleChange(e) {
    this.setState({temperature: e.target.value});
  }

  render() {
    const temperature = this.state.temperature;
    const scale = this.props.scale;
    return (
      <fieldset>
        <legend>Enter temperature in {scaleNames[scale]}:</legend>
        <input value={temperature}>
          onChange={this.handleChange}
        </input>
      </fieldset>
    );
  }
}
```

これで `Calculator` を 2つの別個の温度入力フィールドをレンダーするように変更することができます：

```
class Calculator extends React.Component {
  render() {
    return (
      <div>
        <TemperatureInput scale="c" />
        <TemperatureInput scale="f" />
      </div>
    );
  }
}
```

[Try it on CodePen](#)

2つの入力フィールドが用意できました。しかし、片方に温度を入力しても、もう片方は更新されません。これは要件を満たしていません：2つの入力フィールドを同期させたいのです。

`Calculator` から `BoilingVerdict` を表示することもできません。`Calculator` は `TemperatureInput` の中に隠されている現在の温度を知らないのです。

## 変換関数の作成

まず、摂氏から華氏に変換するものとその反対のものと、2つの関数を書きます。

```
function toCelsius(fahrenheit) {
  return (fahrenheit - 32) * 5 / 9;
}

function toFahrenheit(celsius) {
  return (celsius * 9 / 5) + 32;
}
```

これら2つの関数は数字を変換します。次に文字列で表現された `temperature` と変換関数を引数に取り文字列を返す、別の関数を作成します。この関数を一方の入力の値をもう一方の入力に基づいて計算するのに使用します。

常に値が小数第3位まで四捨五入されるようにし、無効な `temperature` には空の文字列を返します。

```
function tryConvert(temperature, convert) {
  const input = parseFloat(temperature);
  if (Number.isNaN(input)) {
    return '';
  }
  const output = convert(input);
  const rounded = Math.round(output * 1000) / 1000;
  return rounded.toString();
}
```

例えば、`tryConvert('abc', toCelsius)` は空の文字列を返し、`tryConvert('10.22', toFahrenheit)` は '50.396' を返します。

## state のリフトアップ

現時点では、両方の `TemperatureInput` コンポーネントは独立してローカルの state を保持しています：

```
class TemperatureInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {temperature: ''};
  }

  handleChange(e) {
    this.setState({temperature: e.target.value});
  }

  render() {
    const temperature = this.state.temperature;
    // ...
  }
}
```

しかし、2つの入力フィールドはお互いに同期されていて欲しいのです。摂氏の入力フィールドを更新したら、華氏の入力フィールドも華氏に変換された温度で反映されて欲しいですし、逆も同じです。

React での state の共有は、state を、それを必要とするコンポーネントすべての直近の共通祖先コンポーネントに移動することによって実現します。これを "state のリフトアップ (lifting state up)" と呼びます。`TemperatureInput` からローカルの state を削除して `Calculator` に移動しましょう。

`Calculator` が共有の state を保持すれば、それが両方の入力における現在の温度の "信頼できる情報源 (source of truth)" となります。それによって、両方に対して相互に一貫性のある値を持たせることができるようにになります。両方の `TemperatureInput` コンポーネントの props は同じ親コンポーネント `Calculator` から与えられるので、2 つの入力は常に同期されているようになります。

それでは、どのように動作するのかひとつ見ていきましょう。

まず、`TemperatureInput` コンポーネントの `this.state.temperature` を `this.props.temperature` に置き換えます。とりあえず、`this.props.temperature` は既にあるものだとしておきましょう。後でこれは `Calculator` から渡すようにします：

```
render() {
  // Before: const temperature = this.state.temperature;
  const temperature = this.props.temperature;
  // ...
```

`props` が読み取り専用であることは周知の通りです。`temperature` がローカルの state に格納されている間は、`TemperatureInput` は `this.setState()` を呼び出すだけでそれを変更することができました。しかし今や、`temperature` は親コンポーネントから与えられる props の一部ですから、`TemperatureInput` はそれを制御できません。

通常 React では、コンポーネントを "制御された (controlled)" ものとすることでこの問題を解決します。DOM である `<input>` が `value` と `onChange` プロパティの両方を受け取るように、カスタムコンポーネントの `TemperatureInput` は `temperature` と `onTemperatureChange` の両方を親コンポーネントの `Calculator` から受け取ることができます。

ここで、`TemperatureInput` が自身の温度を更新したい場合、`this.props.onTemperatureChange` を呼び出します：

```
handleChange(e) {
  // Before: this.setState({temperature: e.target.value});
  this.props.onTemperatureChange(e.target.value);
  // ...
```

補足：

カスタムコンポーネントの `temperature` や `onTemperatureChange` といった props の名前に特別な意味があるわけではありません。慣習に則り `value` や `onChange` など、他の任意の名前を使うこともできます。

`onTemperatureChange` プロパティは親コンポーネント `Calculator` から `temperature` プロパティと共に渡されます。親コンポーネントは入力の変化に応じて自身のローカル state を更新し、結果的に両方の入力フォームは新しい値で再レンダリングされます。`Calculator` をどう実装するかはこの後すぐに見ていきましょう。

`Calculator` の変更点を見ていく前に、`TemperatureInput` コンポーネントで行った変更をおさらいしましょう。ローカルの state を削除し、`this.state.temperature` の代わりに `this.props.temperature` を読み取るようにしました。また、変更を加えたい場合は `this.setState()` を呼び出す代わりに `Calculator` から与えられる `this.props.onTemperatureChange()` を呼び出すことにしました：

```

class TemperatureInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
  }

  handleChange(e) {
    this.props.onTemperatureChange(e.target.value);
  }

  render() {
    const temperature = this.props.temperature;
    const scale = this.props.scale;
    return (
      <fieldset>
        <legend>Enter temperature in {scaleNames[scale]}:</legend>
        <input value={temperature}
               onChange={this.handleChange} />
      </fieldset>
    );
  }
}

```

では `Calculator` コンポーネントの番です。

現時点での入力の `temperature` と `scale` を、このコンポーネントのローカルな state に保存することにします。これは入力コンポーネントから "リフトアップ" したものであり、両方にとての "信頼出来る情報源" として振る舞うことになります。これは、両方の入力コンポーネントをレンダーするために必要となる最小のデータの形です。

例えば、摂氏側の入力に 37 と打ち込むと、`Calculator` コンポーネントの state は以下のようになります：

```
{
  temperature: '37',
  scale: 'c'
}
```

その後に華氏の入力フィールドを 212 に変更すると、`Calculator` の state は以下のようになります：

```
{
  temperature: '212',
  scale: 'f'
}
```

両方の入力を保存することもできましたが、それは不必要だと分かります。最後に変更された値とそれが示す単位を保存すれば十分なのです。現時点での `temperature` と `scale` の 2 つさえあれば、もう一方の値は推測することができます。

同じ state から値が算出されるので、2 つの入力コンポーネントは常に同期します。

```

class Calculator extends React.Component {
  constructor(props) {
    super(props);
    this.handleCelsiusChange = this.handleCelsiusChange.bind(this);
    this.handleFahrenheitChange = this.handleFahrenheitChange.bind(this);
    this.state = {temperature: '', scale: 'c'};
  }

  handleCelsiusChange(temperature) {
    this.setState({scale: 'c', temperature});
  }
}

```

```

    }

    handleFahrenheitChange(temperature) {
      this.setState({scale: 'f', temperature});
    }

    render() {
      const scale = this.state.scale;
      const temperature = this.state.temperature;
      const celsius = scale === 'f' ? tryConvert(temperature, toCelsius) :
      temperature;
      const fahrenheit = scale === 'c' ? tryConvert(temperature, toFahrenheit) :
      temperature;

      return (
        <div>
          <TemperatureInput
            scale="c"
            temperature={celsius}
            onTemperatureChange={this.handleCelsiusChange} />
          <TemperatureInput
            scale="f"
            temperature={fahrenheit}
            onTemperatureChange={this.handleFahrenheitChange} />
          <BoilingVerdict
            celsius={parseFloat(celsius)} />
        </div>
      );
    }
  }
}

```

## Try it on CodePen

これで、どちらの入力コンポーネントを編集したかに関わらず、`Calculator` の `this.state.temperature` と `this.state.scale` が更新されます。片方の入力コンポーネントはあらゆるユーザからの入力が保持されるよう値をそのまま受け取り、もう片方の入力コンポーネントの値はそれに基づいて常に再計算されます。

入力値を変更した際に何が起こるのかをおさらいしましょう：

- React は DOM の `<input>` で `onChange` として指定された関数を呼び出します。この章の場合、`TemperatureInput` の `handleChange` メソッドが呼び出される関数になります。
- `TemperatureInput` の `handleChange` メソッドは `this.props.onTemperatureChange()` に新しい値を与えて呼び出します。`onTemperatureChange` を含む `props` は親コンポーネントである `Calculator` から与えられます。
- 前回のレンダー時に、`Calculator` は摂氏の `TemperatureInput` の `onTemperatureChange` には自身の `handleCelsiusChange` メソッドを指定し、華氏の `TemperatureInput` の `onTemperatureChange` には自身の `handleFahrenheitChange` を指定していたのでした。そのため、どちらの入力フィールドを編集したかによって、2つの `Calculator` メソッドのどちらが呼び出されるかが決まります。
- これらのメソッド内では、`Calculator` コンポーネントが新しい入力値と更新した方の入力値の単位を `this.setState()` に与えて呼び出して、React に `Calculator` コンポーネント自身を再レンダーさせます。
- React は `Calculator` コンポーネントの `render` メソッドを呼び出して、UI がどのような見た目になるべきかを学びます。両方の入力コンポーネントの値が、現在の温度とアクティブな単位に基づいて再計算されます。温度の変換処理はここで行われます。
- React は `Calculator` により与えられた新しい `props` で各 `TemperatureInput` の `render` メソッドを呼び出します。React はそれらの UI がどのような見た目になるかを学びます。

- React は props として摂氏温度を与えて、BoilingVerdict コンポーネントの render メソッドを呼び出します。
- React DOM は沸騰したかどうかの判定結果と入力コンポーネントの値によって、DOM を更新します。変更された入力コンポーネントは現在の値によって、もう一方の入力コンポーネントは変換された温度によって更新されます。

全ての更新は同じ手順で実行されるので、2つの入力コンポーネントは常に同期を保つことができます。

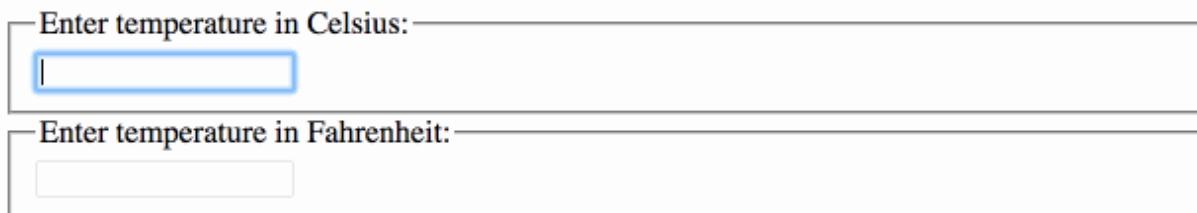
## この章で学んだこと

React アプリケーションで変化するどのようなデータも单一の "信頼出来る情報源" であるべきです。通常、state はレンダー時にそれを必要とするコンポーネントに最初に追加されます。それから、他のコンポーネントもその state を必要としているなら、直近の共通祖先コンポーネントにその state をリフトアップすることができます。異なるコンポーネント間で state を同期しようとする代わりに、[トップダウン型のデータフロー](#)の力を借りるべきです。

state のリフトアップは双方向のバインディング (two-way binding) を行う方法より多くの "ボイラープレート" コードを生み出しますが、その効果としてバグを発見して切り出す作業が少なく済むようになります。あらゆる state はいずれかのコンポーネント内に "存在" し、そのコンポーネントのみがその state を変更できるので、バグが潜む範囲は大幅に削減されます。加えて、ユーザ入力を拒否したり変換したりする任意の独自ロジックを実装することもできます。

props もしくは state から派生的に作りだす事のできるデータについては、おそらく state に保持すべきではないでしょう。例えば、今回は celsiusValue と fahrenheitValue の両方を保存する代わりに、最後に変更された temperature と、その値の scale のみを保存しています。もう一方の入力の値は常に render() メソッド内で計算することができます。これにより元のユーザ入力の精度を全く損なうことなくもう一方の入力フィールドに丸めを適用したり、もう一方の入力フィールドをクリアしたりできます。

UI で何かおかしな箇所があれば、[React Developer Tools](#) を使用して props を調査したり state の更新について責任を持っているコンポーネントに辿り着くまでツリーをさかのぼることができます。これによりバグをその原因まで追いかけることができます。



The screenshot shows the React Developer Tools component tree for the "Calculator" component. The tree includes "TemperatureInput" components for Celsius and Fahrenheit, and a "BoilingVerdict" component. The props for the "Calculator" component are shown as an empty object. The state for the "Calculator" component includes "scale: "c"" and "temperature: """.

react

---

[Go to TOC](#)

# リストと key

まず、JavaScript でリストを変換する方法についておさらいしましょう。

以下のコードでは、`map()` 関数を用い、`numbers` という配列を受け取って中身の値を 2 倍しています。`map()` 関数が返す新しい配列を変数 `doubled` に格納し、ログに出力します：

```
const numbers = [1, 2, 3, 4, 5];
const doubled = numbers.map((number) => number * 2);
console.log(doubled);
```

このコードはコンソールに `[2, 4, 6, 8, 10]` と出力します。

React では配列を要素のリストに変換することが、ほぼこれと同様のものです。

## 複数のコンポーネントをレンダーする

要素の集合を作成し中括弧 `{}` で囲むことで JSX に含めることができます。

以下では、JavaScript の `map()` 関数を利用して、`numbers` という配列に対して反復処理を行っています。それぞれの整数に対して `<li>` 要素を返しています。最後に、結果として得られる要素の配列を `listItems` に格納しています：

```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) =>
  <li>{number}</li>
);
```

そして、`listItems` という配列全体を `<ul>` 要素の内側に含めます：

```
<ul>{listItems}</ul>
```

[Try it on CodePen](#)

このコードは、1 から 5 までの数字の箇条書きのリストを表示します。

## 基本的なリストコンポーネント

通常、リストは何らかのコンポーネントの内部でレンダーしたいと思うでしょう。

前の例をリファクタリングして、`numbers` という配列を受け取って要素のリストを出力するコンポーネントを作ることができます。

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <li>{number}</li>
  );
  return (
    <ul>{listItems}</ul>
  );
}
```

```
const numbers = [1, 2, 3, 4, 5];
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<NumberList numbers={numbers} />);
```

このコードを実行すると、「リスト項目には key を与えるべきだ」という警告を受け取るでしょう。"key" とは特別な文字列の属性であり、要素のリストを作成する際に含めておく必要があるものです。なぜ key が重要なのか、次の節で説明します。

`numbers.map()` 内のリスト項目に `key` を割り当てて、`key` が見つからないという問題を修正しましょう。

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <li key={number.toString()}>
      {number}
    </li>
  );
  return (
    <ul>{listItems}</ul>
  );
}
```

[Try it on CodePen](#)

## Key

Key は、どの要素が変更、追加もしくは削除されたのかを React が識別するのに役立ちます。配列内の項目に安定した識別性を与えるため、それぞれの項目に key を与えるべきです。

```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) =>
  <li key={number.toString()}>
    {number}
  </li>
);
```

兄弟間でその項目を一意に特定できるような文字列を key として選ぶのが最良の方法です。多くの場合、あなたのデータ内にある ID を key として使うことになるでしょう：

```
const todoItems = todos.map((todo) =>
  <li key={todo.id}>
    {todo.text}
  </li>
);
```

レンダーされる要素に安定した ID がない場合、最終手段として項目のインデックスを使うことができます：

```
const todoItems = todos.map((todo, index) =>
  // Only do this if items have no stable IDs
  <li key={index}>
    {todo.text}
  </li>
);
```

要素の並び順が変更される可能性がある場合、インデックスを `key` として使用することはお勧めしません。パフォーマンスに悪い影響を与え、コンポーネントの状態に問題を起こす可能性があります。Robin Pokorny による、[key としてインデックスを用いる際の悪影響についての詳しい解説](#)をご覧ください。もし明示的に `key` を指定しない場合、React はインデックスを `key` のデフォルト値として使います。

より詳しく学びたい場合はこちらの [key が必要である詳細な理由](#)をご覧ください。

## key のあるコンポーネントの抽出

`key` が意味を持つのは、それをとり囲んでいる配列の側の文脈です。

例えば、`ListItem` コンポーネントを抽出する際には、`key` は `ListItem` 自体の `<li>` 要素に書くのではなく、配列内の `<ListItem />` 要素に残しておくべきです。

### 例：不適切な `key` の使用法

```
function ListItem(props) {
  const value = props.value;
  return (
    // Wrong! There is no need to specify the key here:
    <li key={value.toString()}>
      {value}
    </li>
  );
}

function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    // Wrong! The key should have been specified here:
    <ListItem value={number} />
  );
  return (
    <ul>
      {listItems}
    </ul>
  );
}
```

### 例：正しい `key` の使用法

```
function ListItem(props) {
  // Correct! There is no need to specify the key here:
  return <li>{props.value}</li>;
}

function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    // Correct! Key should be specified inside the array.
    <ListItem key={number.toString()} value={number} />
  );
  return (
    <ul>
      {listItems}
    </ul>
  );
}
```

[Try it on CodePen](#)

基本ルールとしては、`map()` 呼び出しの中に現れる要素に `key` が必要です。

## key は兄弟要素の中で一意であればよい

配列内で使われる `key` はその兄弟要素の中で一意である必要があります。しかしグローバルに一意である必要はありません。2つの異なる配列を作る場合は、同一の `key` が使われても構いません：

```
function Blog(props) {
  const sidebar = (
    <ul>
      {props.posts.map((post) =>
        <li key={post.id}>
          {post.title}
        </li>
      )}
    </ul>
  );
  const content = props.posts.map((post) =>
    <div key={post.id}>
      <h3>{post.title}</h3>
      <p>{post.content}</p>
    </div>
  );
  return (
    <div>
      {sidebar}
      <hr />
      {content}
    </div>
  );
}

const posts = [
];

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Blog posts={posts} />);
```

[Try it on CodePen](#)

`key` は React へのヒントとして使われますが、あなたが書くコンポーネントには渡されません。同じ値をコンポーネントの中でも必要としている場合は、別の名前の `prop` として明示的に渡してください：

```
const content = posts.map((post) =>
  <Post
    key={post.id}
    id={post.id}
    title={post.title} />
);
```

上記の例では、`Post` コンポーネントは `props.id` を読み取ることができますが、`props.key` は読み取れません。

## map() を JSX に埋め込む

上記の例では `listItems` 変数を別途宣言して、それを JSX に含めました：

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <ListItem key={number.toString()} value={number} />
  );
  return (
    <ul>
      {listItems}
    </ul>
  );
}
```

JSX では任意の式を埋め込むことができますので、`map()` の結果をインライン化することもできます。

```
function NumberList(props) {
  const numbers = props.numbers;
  return (
    <ul>
      {numbers.map((number) =>
        <ListItem key={number.toString()} value={number} />
      )}
    </ul>
  );
}
```

### Try it on CodePen

時としてこの結果はよりすっきりしたコードとなります。この記法は乱用されることもあります。普通の JavaScript でそうであるように、読みやすさのために変数を抽出する価値があるかどうか決めるのはあなたです。`map()` の中身がネストされすぎている場合は、[コンポーネントに抽出](#)する良いタイミングかもしれない、ということにも留意してください。

# パフォーマンス最適化

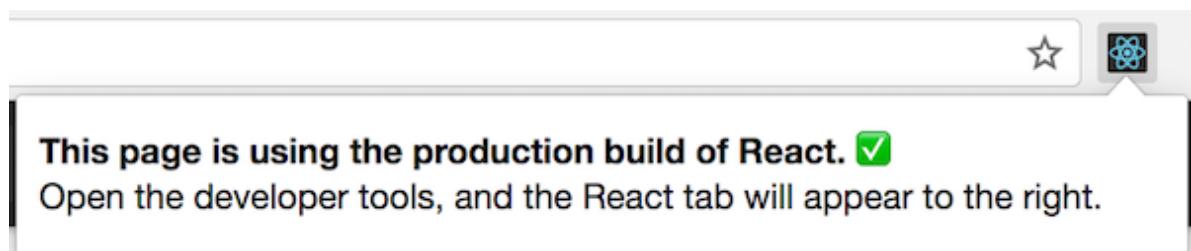
React は UI の更新時に必要となる高コストな DOM 操作の回数を最小化するために、内部的にいくつかの賢いテクニックを使用しています。多くのアプリケーションでは React を使用するだけで、パフォーマンス向上のための特別な最適化を苦労して行わなくとも、レスポンスの良いユーザインターフェースを実現できますが、それでもなお、React アプリケーションを高速化するための方法はいくつか存在します。

## 本番用ビルドを使用する

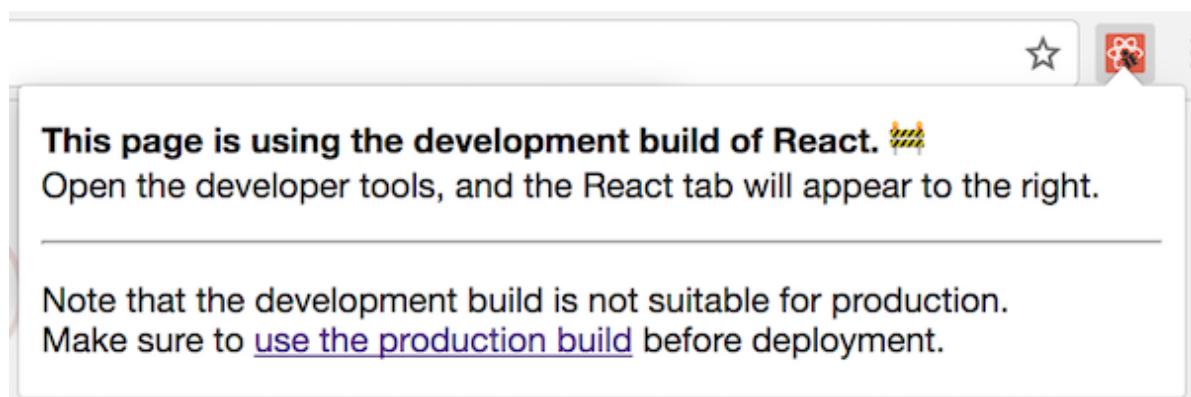
React アプリケーションでベンチマークを行う場合やパフォーマンスの問題が発生している場合には、ミニファイされた本番用ビルドでテストしていることを確認してください。

デフォルトで React は多くの有用な警告チェックを行い、開発時にはとても有用です。しかしそれによって React アプリケーションのサイズが肥大化し、速度が低下してしまうため、アプリケーションのデプロイ時には本番バージョンを使用していることを確認してください。

ビルドプロセスが正しく設定されているか分からない場合、[React Developer Tools for Chrome](#) をインストールして確認できます。本番用モードの React のサイトを訪れた場合、アイコンは暗い背景となっています。



開発モードの React のサイトを訪れた場合、アイコンは赤い背景となっています。



アプリケーションに対して作業をしているときは開発モードを使用し、利用者に配布する場合には本番用モードを使用することをお勧めします。

本番用にアプリを構築するためのそれぞれのツールにおける手順を以下に示します。

## Create React App

プロジェクトが [Create React App](#) で構築されているなら、以下のコードを実行してください。

```
npm run build
```

これでアプリケーションの本番用ビルドがプロジェクト内の `build/` フォルダに作成されます。

これが必要なのは本番用ビルドだけであることに留意してください。通常の開発作業では、`npm start` を使用してください。

## 單一ファイル版ビルド

React と ReactDOM をそれぞれ単一ファイル化した本番環境用のバージョンを提供しています。

```
<script src="https://unpkg.com/react@18/umd/react.production.min.js"></script>
<script src="https://unpkg.com/react-dom@18/umd/react-dom.production.min.js">
</script>
```

本番用に適しているのは、React ファイル名の末尾が `.production.min.js` であるもののみであることに留意ください。

## Brunch

Brunch で最も効率のよい本番用ビルドを行うには、`terser-brunch` をインストールしてください：

```
# If you use npm
npm install --save-dev terser-brunch

# If you use Yarn
yarn add --dev terser-brunch
```

そして、本番用ビルドを作成するために、`build` コマンドに `-p` オプションを指定して実行します。

```
brunch build -p
```

これが必要なのは本番用ビルドだけであることに留意してください。React の有用な警告表示が隠されたり、ビルド速度が大幅に遅くなったりしますので、開発用では `-p` フラグを指定したり、`uglify-js-brunch` プラグインを適用したりしないでください。

## Browserify

Browserify で最も効率の良い本番用ビルドを行うには、いくつかのプラグインをインストールしてください。

```
# If you use npm
npm install --save-dev envify terser uglifyify

# If you use Yarn
yarn add --dev envify terser uglifyify
```

本番用ビルドを作成するには、以下の変換 (transform) を追加してください（順番は重要です）。

- `envify` 変換は正しいビルド用の環境変数が確実に設定されるようにします。グローバルに設定してください (`-g`)。

- uglifyify 変換は開発用にインポートしたライブラリを削除します。これもグローバルに設定してください ( -g )。
- 最後に結果として出力されたものを、名前の圧縮のために terser にパイプします ([理由を読む](#))。

以下に例を示します。

```
browserify ./index.js \
-g [ envify --NODE_ENV production ] \
-g uglifyify \
| terser --compress --mangle > ./bundle.js
```

これが必要なのは本番用ビルトだけであることに留意してください。React の有用な警告文が隠されたり、ビルト速度が大幅に遅くなったりしますので、開発用ではこれらのプラグインを適用しないでください。

## Rollup

Rollup で最も効率のよい本番用ビルトを行うには、いくつかのプラグインを以下のようにインストールします。

```
# If you use npm
npm install --save-dev rollup-plugin-commonjs rollup-plugin-replace rollup-plugin-terser

# If you use Yarn
yarn add --dev rollup-plugin-commonjs rollup-plugin-replace rollup-plugin-terser
```

本番用ビルトを作成するには、以下のプラグインを追加してください (順番は重要です)。

- replace プラグインは正しいビルト用の環境変数が確実に設定されるようにします。
- commonjs プラグインは Rollup で CommonJS をサポートできるようにします。
- terser プラグインは出力された最終的なバンドルを圧縮し、mangle (訳注: 変数名や識別子を短縮) します。

```
plugins: [
  // ...
  require('rollup-plugin-replace')({
    'process.env.NODE_ENV': JSON.stringify('production')
  }),
  require('rollup-plugin-commonjs')(),
  require('rollup-plugin-terser')(),
  // ...
]
```

設定例の全体はこの [gist を参照](#)してください。

これらが必要なのは本番用ビルトだけであることに留意してください。React の有用な警告表示が隠されたり、ビルト速度が大幅に遅くなったりしますので、開発用では terser プラグインもしくは replace プラグインを 'production' という値で適用しないでください。

## webpack

**補足 :**

Create React App を利用している場合は、[Create React App についての前述の説明](#)に従ってください。  
このセクションは直接 webpack の設定を行いたい人向けです。

Webpack v4 以降では本番 (production) モードでコードの minify を自動で行います。

```
const TerserPlugin = require('terser-webpack-plugin');

module.exports = {
  mode: 'production',
  optimization: {
    minimizer: [new TerserPlugin({ /* additional options here */ })],
  },
};
```

より詳細な説明については [webpack のドキュメント](#)を参照ください。

これらが必要なのは本番用ビルドだけであることに留意してください。React の有用な警告文が隠されたり、ビルド速度が大幅に遅くなったりしますので、開発時には `TerserPlugin` を適用しないでください。

## DevTools プロファイラを使用したコンポーネントのプロファイリング

`react-dom` 16.5 以降と `react-native` 0.57 以降では、開発モードにおける強化されたプロファイリング機能を React DevTools プロファイラにて提供しています。このプロファイラの概要はブログ記事 "[Introducing the React Profiler](#)" で説明されています。チュートリアル動画も [YouTube](#) で閲覧できます。

React DevTools をまだインストールしていない場合は、以下で見つけることができます。

- [Chrome ブラウザ拡張](#)
- [Firefox ブラウザ拡張](#)
- [スタンドアロンの Node パッケージ](#)

### 補足

本番ビルド版 `react-dom` のプロファイリング可能なバンドルとして `react-dom/profiling` が利用可能です。このバンドルの使い方の詳細については、[fb.me/react-profiling](#) を参照してください。

### 補足

React 17 より前のバージョンでは、標準の [User Timing API](#) を用いて Chrome のパフォーマンスタブでコンポーネントのプロファイリングが行われていました。これについての概要是 [Ben Schwarz](#) によるこの記事を参照してください。

## 長いリストの仮想化

アプリケーションが長いデータのリスト（数百～数千行）をレンダーする場合は、「ウィンドウイング」として知られるテクニックを使うことをおすすめします。このテクニックでは、ある瞬間ごとにリストの小さな部分集合のみを描画することで、生成する DOM ノードの数およびコンポーネントの再描画にかかる時間を大幅に削減することができます。

[react-window](#) と [react-virtualized](#) は人気があるウィンドウイング処理のライブラリです。これらはリスト、グリッド、および表形式のデータを表示するための、いくつかの再利用可能コンポーネントを提供しています。アプリケーションの特定のユースケースに合わせた追加的な処理をする場合は、[Twitter](#) が行なっているように、独自のウィンドウイング処理のコンポーネントを作成することもできます。

## リコンシリエーション(差分検出処理)を避ける

React はレンダーされた UI の内部表現を構築し、維持します。その内部表現にはコンポーネントが返した React 要素も含まれています。React はこの内部表現を使うことによって、DOM ノードの不要な作成やアクセス（これらは JavaScript オブジェクト操作よりも低速です）を回避します。この内部表現はしばしば "仮想 DOM" と呼ばれますが、React Native 上でも同様に動くものです。

コンポーネントの `props` や `state` が変更された場合、React は新しく返された要素と以前にレンダーされたものとを比較することで、実際の DOM の更新が必要かを判断します。それらが等しくない場合、React は DOM を更新します。

React は変更された DOM ノードだけを更新するとはいえ、再レンダーには時間がかかります。多少の時間がかかるても多くの場合は問題にはなりませんが、遅延が目立つ場合、再レンダープロセスが開始される前にトリガーされるライフサイクル関数 `shouldComponentUpdate` をオーバーライド定義することで、スピードを抜本的に向上できます。この関数のデフォルトの実装は `true` を返し、React に更新処理をそのまま実行させます：

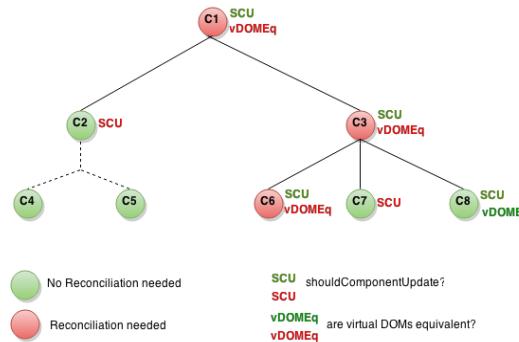
```
shouldComponentUpdate(nextProps, nextState) {
  return true;
}
```

ある状況においてコンポーネントを更新する必要がないと分かっているなら、`shouldComponentUpdate` から `false` を返すことにより、該当コンポーネントおよび配下への `render()` 呼び出しを含む、レンダー処理の全体をスキップすることができます。

ほとんどの場合には、手書きの `shouldComponentUpdate()` を定義する代わりに `React.PureComponent` を継承できます。これは現在と直前の `props` と `state` に対する浅い (shallow) 比較を行う `shouldComponentUpdate()` を実装することと同じです。

## shouldComponentUpdate の実際の動作

以下のようなコンポーネントのサブツリーがあるとします。それぞれ、`SCU` は `shouldComponentUpdate` が返した値（訳注：緑は `true`、赤は `false`）を示し、`vDOMEq` はレンダーされた React 要素が等しかったかどうか（訳注：緑は等しい、赤は等しくない）を示します。最後に、円の色はコンポーネントに対してツリーの差分を検出するリコンシリエーション処理を必要としたのかどうか（訳注：緑は不要、赤は必要）を示します。



C2 をルートとするサブツリーでは `shouldComponentUpdate` が `false` を返したので、React は C2 をレンダーしようとしました。したがって C4 と C5 については `shouldComponentUpdate` を実行する必要すらなかったわけです。

C1 と C3 では、`shouldComponentUpdate` が `true` を返したので、React は葉ノードにも移動してチェックする必要がありました。C6 では `shouldComponentUpdate` が `true` を返し、そしてレンダーされた React 要素も等しくなったので、React は DOM を更新する必要がありました。

最後の興味深いケースが C8 です。React はこのコンポーネントをレンダーする必要がありましたが、返された React 要素は前回レンダーされたときものと同じだったので、DOM の更新は必要ありませんでした。

React が実 DOM を更新しなければならなかったのは、C6 だけだったことに注目してください。C6 の更新は避けられないものでした。C8 では、レンダーされた React 要素の比較のおかげで実 DOM を修正せずに済みました。C2 のサブツリーと C7 のケースでは `shouldComponentUpdate` のおかげで、`render` メソッドの呼び出しや React 要素の比較処理すらスキップすることができました。

## 例

コンポーネントが変化するのが `props.color` または `state.count` 変数が変化した時だけだとしたら、`shouldComponentUpdate` では以下のようなチェックを行えます。

```
class CounterButton extends React.Component {
  constructor(props) {
    super(props);
    this.state = {count: 1};
  }

  shouldComponentUpdate(nextProps, nextState) {
    if (this.props.color !== nextProps.color) {
      return true;
    }
    if (this.state.count !== nextState.count) {
      return true;
    }
    return false;
  }

  render() {
    return (
      <button
        color={this.props.color}
        onClick={() => this.setState(state => ({count: state.count + 1}))}>
    
```

```

    Count: {this.state.count}
  </button>
);
}
}

```

このコードは、`shouldComponentUpdate` は `props.color` または `state.count` の変化の有無を単にチェックしているだけです。これらの値が変化していなければコンポーネントは更新されません。コンポーネントがもっと複雑な場合は、`props` と `state` のすべてのフィールドに対して「浅い比較」をするという同種のパターンでコンポーネント更新の必要性を決定できます。このパターンはとても一般的なので、React はこのロジックのためのヘルパーを用意しており、`React.PureComponent` から継承するだけで使用できます。なので以下のコードで前述のコードと同じことをよりシンプルに実装できます。

```

class CounterButton extends React.PureComponent {
  constructor(props) {
    super(props);
    this.state = {count: 1};
  }

  render() {
    return (
      <button
        color={this.props.color}
        onClick={() => this.setState(state => ({count: state.count + 1}))}>
        Count: {this.state.count}
      </button>
    );
  }
}

```

ほとんどの場合、自分で `shouldComponentUpdate` を記述する代わりに `React.PureComponent` を使うことができます。もっとも、浅い比較を行うだけですので、浅い比較では検出できない形で `props` や `state` がミューテート（mutate；書き換え）されている可能性がある場合には使えません。

この事はより複雑なデータ構造の場合には問題となります。例えば、カンマ区切りで単語をレンダーする `ListOfWords` コンポーネントと、ボタンをクリックしてリストに単語を追加できる親コンポーネント `WordAdder` が必要だとして、以下のコードは正しく動作しません。

```

class ListOfWords extends React.PureComponent {
  render() {
    return <div>{this.props.words.join(',')}</div>;
  }
}

class WordAdder extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      words: ['marklar']
    };
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    // This section is bad style and causes a bug
    const words = this.state.words;
  }
}

```

```

words.push('marklar');
this.setState({words: words});
}

render() {
  return (
    <div>
      <button onClick={this.handleClick} />
      <ListOfWords words={this.state.words} />
    </div>
  );
}
}

```

問題は `PureComponent` が `this.props.words` の古い値と新しい値を単純に比較していることにあります。上記のコードでは `WordAdder` の `handleClick` メソッド内で `words` 配列の内容をミューテートしてしまうので、`this.props.words` の新旧の値は、たとえ配列内の実際の単語が変更されていたとしても、比較の結果同じだとみなしてしまうのです。そのため `ListOfWords` はレンダーすべき新しい単語が追加されているにも関わらず、更新されません。

## データを変更しないことの効果

この問題を避ける最も単純な方法は、`props` や `state` として使用する値のミューテートを避けることです。例えば、上記の `handleClick` メソッドは `concat` を使って以下のように書き換えることができます：

```

handleClick() {
  this.setState(state => ({
    words: state.words.concat(['marklar'])
  }));
}

```

ES6 はこれをより簡潔に実装できる配列の[スプレッド構文](#)をサポートしています。Create React App を使用していれば、この構文はデフォルトで利用できます。

```

handleClick() {
  this.setState(state => ({
    words: [...state.words, 'marklar'],
  }));
}

```

同様に、オブジェクトについてもミューテートするコードをしないように書き換えることができます。例えば、`colormap` というオブジェクトがあり、`colormap.right` を `'blue'` に更新する関数が必要だとしましょう。以下のように書くことも可能ですが、

```

function updateColorMap(colormap) {
  colormap.right = 'blue';
}

```

この処理を、元オブジェクトをミューテートせずに実装するために、`Object.assign` を使用できます。

```

function updateColorMap(colormap) {
  return Object.assign({}, colormap, {right: 'blue'});
}

```

これで、`updateColorMap` は古いオブジェクトをミューテートするのではなく新しいオブジェクトを返すようになります。`Object.assign` は ES6 からの機能であり、ポリフィルが必要です（訳注：ブラウザや処理系が ES6 に未対応の場合）。

同様に、[オブジェクトのスプレッドプロパティ構文](#)を使うことで、ミューテートを避けてのオブジェクト更新が容易になります。

```
function updateColorMap(colormap) {
  return {...colormap, right: 'blue'};
}
```

この機能は ES2018 で JavaScript に追加されたものです。

Create React App を使用しているなら、`Object.assign` およびオブジェクトのスプレッド構文の両方がデフォルトで利用できます。

深くネストされたオブジェクトを扱っている場合、ミューテートを行わない形式で更新することが複雑に感じことがあります。このような問題がある場合は [Immer](#) や [immutability-helper](#) を試してみてください。これらのライブラリはミューテートを行わないことによる利点を損なわずに、読みやすいコードを書くのに役立ちます。

---

[Go to TOC](#)

# ポータル

ポータル (portal) は、親コンポーネントの DOM 階層外にある DOM ノードに対して子コンポーネントをレンダーするための公式の仕組みを提供します。

```
ReactDOM.createPortal(child, container)
```

第 1 引数 (`child`) は [React の子要素としてレンダー可能なもの](#)、例えば、要素、文字列、フラグメントなどです。第 2 引数 (`container`) は DOM 要素を指定します。

## 使い方

通常、コンポーネントの `render` メソッドから要素を返すと、最も近い親ノードの子として DOM にマウントされます。

```
render() {
  // React は新しい div 要素をマウントし、子をその中に描画します
  return (
    <div>
      {this.props.children}
    </div>
  );
}
```

しかし、時に子要素を DOM 上の異なる位置に挿入したほうが便利なことがあります。

```
render() {
  // React は新しい div をつくりません。子要素は `domNode` に対して描画されます。
  // `domNode` は DOM ノードであれば何でも良く、DOM 構造内のどこにあるかは問いません。
  return ReactDOM.createPortal(
    this.props.children,
    domNode
  );
}
```

ポータルの典型的なユースケースとは、親要素が `overflow: hidden` や `z-index` のスタイルを持っていても、子要素がコンテナを「飛び出して」見える必要があるものです。例えば、ダイアログ、ホバーカード、ツールチップがそれに当たります。

### 補足

ポータルを利用する際は、[キーボードのフォーカスの管理](#)を行うことが重要になるので、忘れずに行ってください。

モーダルダイアログについては [WAI-ARIA モーダルの推奨実装方法](#)に従い、誰もが利用できるという状態を確保してください。

[Try it on CodePen](#)

## ポータルを介したイベントのバーリング

ポータルは DOM ツリーのどこにでも存在できますが、他のあらゆる点では通常の React の子要素と変わらずに振る舞います。コンテクスト (context) のような機能は、たとえ子要素がポータルであろうと全く同じように動きます。というのも、DOM ツリー上の位置にかかわらず、ポータルは依然として *React* のツリー内にいるからです。

これにはイベントのバーリングも含まれます。ポータルの内部で発火したイベントは *React* のツリー内の祖先へと伝播します。たとえそれが DOM ツリー上では祖先でなくともです。次のような HTML 構造があったとして、

```
<html>
  <body>
    <div id="app-root"></div>
    <div id="modal-root"></div>
  </body>
</html>
```

`#app-root` 内にある `Parent` コンポーネントは、`#modal-root` 内のコンポーネントから伝播したイベントが捕捉されなかった場合に、それを捕捉できます。

```
// この 2 つのコンテナは DOM 上の兄弟要素とします
const appRoot = document.getElementById('app-root');
const modalRoot = document.getElementById('modal-root');

class Modal extends React.Component {
  constructor(props) {
    super(props);
    this.el = document.createElement('div');
  }

  componentDidMount() {
    // ポータルの要素が DOM ツリーに挿入されるのは、
    // Modal の子要素がマウントされた後になります。
    // つまり、子要素は一旦どこにも結びつかない
    // DOM ノードへとマウントされるということです。
    // もし子コンポーネントがマウント後すぐに DOM ツリーに結びついてほしい --
    // たとえば DOM ノードの大きさを測りたい、子孫要素で `autoFocus` を使いたいなど
    // -- 場合は、Modal に状態を持たせて Modal が
    // DOM ツリーに挿入されているときだけ子要素をレンダラーするようにします。
    modalRoot.appendChild(this.el);
  }

  componentWillUnmount() {
    modalRoot.removeChild(this.el);
  }

  render() {
    return ReactDOM.createPortal(
      this.props.children,
      this.el
    );
  }
}

class Parent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {clicks: 0};
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    console.log(`Clicked ${this.state.clicks} times`);
  }

  render() {
    return (
      <div>
        <p>Click me!</p>
        <button onClick={this.handleClick}>Click</button>
      </div>
    );
  }
}
```

```
}

handleClick() {
  // これは Child 内のボタンがクリックされた際に発火し、
  // Parent の state を更新します。
  // たとえそのボタンが DOM 上では直系の子孫でなかったとしてもです。
  this.setState(state => ({
    clicks: state.clicks + 1
  }));
}

render() {
  return (
    <div onClick={this.handleClick}>
      <p>Number of clicks: {this.state.clicks}</p>
      <p>
        Open up the browser DevTools
        to observe that the button
        is not a child of the div
        with the onClick handler.
      </p>
      <Modal>
        <Child />
      </Modal>
    </div>
  );
}

function Child() {
  // クリックするとイベントが親に伝播します。
  // なぜならここには `onClick` 属性が定義されていないからです。
  return (
    <div className="modal">
      <button>Click</button>
    </div>
  );
}

const root = ReactDOM.createRoot(appRoot);
root.render(<Parent />);
```

### Try it on CodePen

ポータルから伝播したイベントが親コンポーネントで捕捉できるということは、ポータルに本質的に依存することのない、より柔軟な抽象化が可能であるということを示しています。たとえば `<Modal />` の実装がポータルを使っているかに関係なく、`<Modal />` コンポーネントをレンダーしてそこから来るイベントを捕捉することができます。

---

[Go to TOC](#)

# ES6 なしで React を使う

通常、React コンポーネントはプレーンな JavaScript クラスとして定義されます。

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

もしあなたがまだ ES6 を使っていないのであれば、代わりに `create-react-class` モジュールを使うことができます。

```
var createReactClass = require('create-react-class');
var Greeting = createReactClass({
  render: function() {
    return <h1>Hello, {this.props.name}</h1>;
  }
});
```

ES6 のクラス API は `createReactClass()` とよく似ていますが、いくつかの例外があります。

## デフォルト props の宣言

関数や ES6 クラスでは、`defaultProps` はコンポーネント自体のプロパティとして定義されます。

```
class Greeting extends React.Component {
  // ...
}

Greeting.defaultProps = {
  name: 'Mary'
};
```

`createReactClass()` の場合、渡されるオブジェクト内の関数として `getDefaults()` を定義する必要があります。

```
var Greeting = createReactClass({
  getDefaults: function() {
    return {
      name: 'Mary'
    };
  },
  // ...
});
```

## state の初期値の設定

ES6 クラスでは、コンストラクタで `this.state` へ代入することで state の初期値を定義できます。

```
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {count: props.initialCount};
  }
  // ...
}
```

`createReactClass()` の場合、`state` の初期値を返す `getInitialState` メソッドを別途用意しなければなりません。

```
var Counter = createReactClass({
  getInitialState: function() {
    return {count: this.props.initialCount};
  },
  // ...
});
```

## 自動バインド

ES6 クラスとして宣言された React コンポーネントでは、メソッドは通常の ES6 クラスと同様のセマンティクスに従います。つまり、`this` がそのインスタンスへ自動的にバインドされることはありません。コンストラクタで明示的に `.bind(this)` を利用する必要があります。

```
class SayHello extends React.Component {
  constructor(props) {
    super(props);
    this.state = {message: 'Hello!'};
    // This line is important!
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    alert(this.state.message);
  }

  render() {
    // Because `this.handleClick` is bound, we can use it as an event handler.
    return (
      <button onClick={this.handleClick}>
        Say hello
      </button>
    );
  }
}
```

`createReactClass()` の場合は全てのメソッドがバインドされるため、明示的なバインドは必要ありません。

```
var SayHello = createReactClass({
  getInitialState: function() {
    return {message: 'Hello!'};
  },

  handleClick: function() {
    alert(this.state.message);
  },

  render: function() {
    return (

```

```

    <button onClick={this.handleClick}>
      Say hello
    </button>
  );
}
});

```

これはつまり、ES6 クラスで書くとイベントハンドラのための定型文が少し多くなってしまうということなのですが、一方では大きなアプリケーションの場合にわずかながらパフォーマンスが向上するという側面もあります。

この定型文的コードがあまりに醜く感じられる場合、[ES2022 のクラスプロパティ構文](#)を利用できます：

```

class SayHello extends React.Component {
  constructor(props) {
    super(props);
    this.state = {message: 'Hello!'};
  }

  // Using an arrow here binds the method:
  handleClick = () => {
    alert(this.state.message);
  };

  render() {
    return (
      <button onClick={this.handleClick}>
        Say hello
      </button>
    );
  }
}

```

他の選択肢もあります。

- コンストラクタでメソッドをバインドする。
- 例えば `onClick={(e) => this.handleClick(e)}` のような形でアロー関数を利用する。
- 引き続き `createClass` を利用する。

## ミックスイン

### 補足：

ES6 はミックスインのサポートを含んでいません。従って、React を ES6 クラスで使う場合にミックスインのサポートはありません。

加えて、ミックスインを用いたコードによる多くの問題が見つかっており、[新規コードで利用することは推奨されません](#)。

この節は参考のためだけに存在します。

時には同じ機能が全く異なるコンポーネント間で共有されることがあります。これは横断的関心事 (cross-cutting concerns) と呼ばれることができます。`createReactClass` であれば、横断的関心事のためにレガシーな `mixins` 機能を使うことができます。

よくある利用例のひとつは、一定時間ごとに自分自身を更新するコンポーネントです。`setInterval()` を使うのは簡単ですが、その場合はメモリ節約のため、コンポーネントが不要になった際にキャンセルすることが重要です。React はライフサイクルメソッドを提供しており、コンポーネントが生成、破棄されるときに知らせてくれます。次のようなシンプルなミックスインを作ってみましょう。このミックスインのメソッドは簡単な `setInterval()` 機能を提供し、コンポーネントが破棄されるタイミングで自動的にクリーンアップされます。

```
var SetIntervalMixin = {
  componentWillMount: function() {
    this.intervals = [];
  },
  setInterval: function() {
    this.intervals.push(setInterval.apply(null, arguments));
  },
  componentWillUnmount: function() {
    this.intervals.forEach(clearInterval);
  }
};

var createReactClass = require('create-react-class');

var TickTock = createReactClass({
  mixins: [SetIntervalMixin], // Use the mixin
  getInitialState: function() {
    return {seconds: 0};
  },
  componentDidMount: function() {
    this.setInterval(this.tick, 1000); // Call a method on the mixin
  },
  tick: function() {
    this.setState({seconds: this.state.seconds + 1});
  },
  render: function() {
    return (
      <p>
        React has been running for {this.state.seconds} seconds.
      </p>
    );
  }
});

const root = ReactDOM.createRoot(document.getElementById('example'));
root.render(<TickTock />);
```

コンポーネントが複数のミックスインを使っており、いくつかのミックスインが同じライフサイクルメソッドを定義している場合（例：コンポーネント破棄時に複数のミックスインがクリーンアップ処理をする）、全てのライフサイクルメソッドが呼ばれることが保証されています。ミックスインで定義されているメソッドはミックスインとして列挙された順に実行され、その後にコンポーネントのメソッドが呼ばれます。

# JSX なしで React を使う

JSX は React を使うための必須事項ではありません。JSX なしで React を使うことは、あなたのビルド環境で JSX のコンパイルの設定をしたくない時には便利です。

各 JSX 要素は、`React.createElement(component, props, ...children)` を呼び出すための単なるシンタックスシュガードです。つまり、JSX を使ってできることは、普通の JavaScript を使ってもできます。

例えば、JSX で書かれた以下のコードは：

```
class Hello extends React.Component {
  render() {
    return <div>Hello ${this.props.toWhat}</div>;
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Hello toWhat="World" />);
```

JSX を使わない以下のコードにコンパイルできます：

```
class Hello extends React.Component {
  render() {
    return React.createElement('div', null, `Hello ${this.props.toWhat}`);
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(React.createElement(Hello, {toWhat: 'World'}, null));
```

JSX から JavaScript への変換方法の例をもっと見たいなら、[オンラインの Babel コンパイラ](#)で試すことができます。

コンポーネントは文字列、`React.Component` のサブクラス、もしくはプレーンな関数のいずれかで指定されます。

たくさんの `React.createElement` をタイピングするのにうんざりした場合、一般的なパターンの 1 つは以下のショートハンドを割り当てることです。

```
const e = React.createElement;

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(e('div', null, 'Hello World'));
```

このショートハンドを `React.createElement` に使用すれば、JSX なしで React を使うのにとても便利です。

あるいは、簡潔な構文を提供する `react-hyperscript` や `hyperscript-helpers` のようなコミュニティプロジェクトも参照してみてください。

---

[Go to TOC](#)

# 差分検出処理

React は、各更新で実際に何が変更されるべきかを人間が心配する必要がないように、宣言型の API を提供しています。これによりアプリケーションの作成が大幅に容易になるわけですが、React の中でこの処理がどのように実装されているのかはよく分からぬかもしれません。この章では React の "差分" アルゴリズムについて、コンポーネントの更新を予測可能なものとしながら、ハイパフォーマンスなアプリケーションの要求を満たす速度を得られるように、私たちが行った選択について説明します。

## 何が問題なのか

React を使う際、`render()` 関数がある時点の React 要素のツリーを作成するものとして考えることができます。次の state や props の更新時には、`render()` 関数は React 要素の別のツリーを返します。React はそこから直近のツリーに合致させるように効率よく UI を更新する方法を見つける必要があります。

あるツリーを別のものに変換するための最小限の操作を求めるというアルゴリズム問題については、いくつかの一般的な解決方法が存在しています。しかし、[最新のアルゴリズム](#) でもツリーの要素数を  $n$  として  $O(n^3)$  ほどの計算量があります。

React でそのアルゴリズムを使った場合、1000 個の要素を表示するのに 10 億といったレベルの比較が必要となります。これではあまりに計算コストが高すぎます。代わりに、React は 2 つの仮定に基づくことで、ある程度近い結果を得ることができる  $O(n)$  ほどの計算量のアルゴリズムを実装しています。

- 異なる型の 2 つの要素は異なるツリーを生成する。
- 開発者は key プロパティを与えることで、異なるレンダー間でどの子要素が変化しない可能性があるのかについてヒントを出すことができる。

実際に、これらの仮定はほとんど全ての実践的なユースケースで有効です。

## 差分アルゴリズム

2 つのツリーが異なる場合、React は最初に 2 つのルート要素を比較します。そのふるまいはルート要素の型に応じて異なります。

### 異なる型の要素

ルート要素が異なる型を持つ場合は常に、React は古いツリーを破棄して新しいツリーをゼロから構築します。`<a>` から `<img>` へ、もしくは `<Article>` から `<Comment>` へ、もしくは `<Button>` から `<div>` へ — それらの全てがツリーをゼロから再構築させるのです。

ツリーを破棄する時点で、古い DOM ノードは破棄されます。コンポーネントのインスタンスは `componentWillUnmount()` を受け取ります。新しいツリーを構築する時点で、新しい DOM ノードが DOM に挿入されます。コンポーネントのインスタンスは `UNSAFE_componentWillMount()` とそれから `componentDidMount()` を受け取ります。古いツリーに関連付けられた全ての state は失われます。

ルート配下のコンポーネントはアンマウントされ、それらの state は破棄されます。例えば、以下のように異なる場合：

```
<div>
  <Counter />
</div>

<span>
  <Counter />
</span>
```

古い `Counter` は破棄され、新しいものが再マウントされます。

補足：

このコードはレガシーとみなされるため、新規コードでは[使用を避けるべきです](#)：

- `UNSAFE_componentWillMount()`

## 同じ型の DOM 要素

同じ型の 2 つの React DOM 要素を比較した場合、React はそれぞれの属性を調べ、対応する共通の DOM ノードを保持し、変更された属性のみを更新します。例えば：

```
<div className="before" title="stuff" />
<div className="after" title="stuff" />
```

これらの 2 つの要素を比べた場合、React は対応する DOM ノードの `className` のみを更新すればよいと分かります。

`style` を更新した場合は、React は同様に変更されたプロパティのみを更新すればよいと分かります。例えば：

```
<div style={{color: 'red', fontWeight: 'bold'}} />
<div style={{color: 'green', fontWeight: 'bold'}} />
```

2 つの要素を変換する場合、React は `fontWeight` ではなく `color` のみを変更すればよいことが分かります。

この DOM ノードを処理した後、React は子に対して再帰的に処理を行っていきます。

## 同じ型のコンポーネント要素

コンポーネントが更新される場合、インスタンスは同じままとなり、レンダー間で `state` は保持されます。React は対応するコンポーネントのインスタンスの `props` を新しい要素に合うように更新し、`UNSAFE_componentWillReceiveProps()`、`UNSAFE_componentWillUpdate()` および `componentDidUpdate()` を対応するインスタンスに対して呼び出します。

次に、`render()` メソッドが呼ばれ、差分アルゴリズムが再帰的に前の結果と新しい結果を処理します。

補足：

これらのコードはレガシーとみなされるため、新規コードでは[使用を避けるべきです](#)：

- `UNSAFE_componentWillUpdate()`

- UNSAFE\_componentWillReceiveProps()

## 子要素の再帰的な処理

デフォルトでは、DOM ノードの子に対して再帰的に処理を行う場合、React は単純に、両方の子要素リストのそれぞれ最初から同時に処理を行っていって、差分を見つけたところで毎回更新を発生させます。

例えば、子ノードの最後にひとつ要素を追加するような場合、以下の 2 つのツリー間の変換はうまく動作します：

```
<ul>
  <li>first</li>
  <li>second</li>
</ul>

<ul>
  <li>first</li>
  <li>second</li>
  <li>third</li>
</ul>
```

React は 2 つの `<li>first</li>` ツリーを一致させ、2 つの `<li>second</li>` ツリーを一致させ、最後に `<li>third</li>` ツリーを挿入します。

それを単純に実行した場合、先頭への要素の追加はパフォーマンスが悪くなります。

```
<ul>
  <li>Duke</li>
  <li>Villanova</li>
</ul>

<ul>
  <li>Connecticut</li>
  <li>Duke</li>
  <li>Villanova</li>
</ul>
```

React は `<li>Duke</li>` と `<li>Villanova</li>` サブツリーをそのまま保てるということに気づくことなく、すべての子要素を変更してしまいます。この非効率性は問題になることがあります。

## Keys

この問題を解決するため、React は `key` 属性をサポートします。子要素が `key` を持っている場合、React は `key` を利用して元のツリーの子要素と次のツリーの子要素を対応させます。例えば、`key` を前出の非効率な例に追加することで、ツリーの変換を効率的なものにすることができます。

```
<ul>
  <li key="2015">Duke</li>
  <li key="2016">Villanova</li>
</ul>

<ul>
  <li key="2014">Connecticut</li>
  <li key="2015">Duke</li>
  <li key="2016">Villanova</li>
</ul>
```

これで React は、'2014' の key を持つ要素が新規の要素であり、'2015' と '2016' の key をもつ要素は移動しただけだ、と理解するようになります。

実際に、key を探すのはたいてい難しくありません。表示しようとしている要素は既に固有の ID を持っているかもしれませんので、key をそのデータから設定するだけです。

```
<li key={item.id}>{item.name}</li>
```

そうではない場合には、新しい ID プロパティをモデルに追加するか、key を生成するためにコンテンツの一部をハッシュ化します。key は兄弟要素間で一意であればよく、グローバルに一意である必要はありません。

最後の手段として、配列の要素のインデックスを key として渡すことができます。項目が並び替えられることができればうまく動作しますが、並び替えられると遅くなります。

key として配列のインデックスが使用されている場合、並べ替えはコンポーネントの状態に関しても問題を起こすことがあります。コンポーネントのインスタンスは key に基づいて更新、再利用されます。インデックスが key の場合、要素の移動はインデックスの変更を伴います。結果として、非制御の入力などに対するコンポーネントの状態が混乱し、予期せぬ形で更新されてしまうことがあります。

CodePen で、[配列のインデックスを key として使うことで生じる問題](#)、および[配列のインデックスを使わないことで、ソートや並び替え、要素の先頭への追加にまつわる問題](#)がどのように解決されるのかを示した変更後のバージョンを見るることができます。

## トレードオフ

この差分検出処理アルゴリズムは内部の実装の詳細であることに気をつけておく事が重要です。React はアクション毎にアプリケーション全体を再レンダーし得るものです。最終結果はいずれにせよ同じでしょう。誤解のないように言っておくと、ここでの再レンダーとは全てのコンポーネントに対して `render` メソッドを呼び出すことであり、React がそれらをアンマウントして再びマウントすることではありません。前述のルールに従って変更を適用するだけです。

私達は一般的なユースケースで高速となるように、定期的にヒューリスティクスを改善しています。現時点の実装ではサブツリーが兄弟要素の間で移動したということは表現できますが、それ以外のどこか別の場所に移動したということは伝えることはできません。結果的にアルゴリズムはサブツリーを再レンダーします。

React はヒューリスティクスに依存するため、その背後にある仮定に合致しなければ、パフォーマンスが低下します。

1. 差分アルゴリズムは型が異なるコンポーネント間でサブツリーを照合しようとはしません。2つのとてもよく似た出力をするコンポーネントの型を入れ替えているケースがあれば、同じ型にした方がいいかもしれません。現実的には、これが問題となったことはありません。
2. key は予測可能で安定した、一意なものであるべきです。不安定な key (例えば `Math.random()` 関数で生成するような) は多くのコンポーネントのインスタンスと DOM ノードを不必要に再生成し、パフォーマンスの低下や子コンポーネントの state の喪失を引き起します。

[Go to TOC](#)

# DOM 要素

React はパフォーマンスとブラウザ間での互換性のために、ブラウザから独立した DOM システムを実装しています。このことを機に、ブラウザの DOM 実装にあるいくつかの粗削りな部分が取り払われました。

React では、DOM のプロパティと属性（イベントハンドラを含む）全てがキャメルケースで名前付けされる必要があります。例えば、HTML 属性 `tabindex` に React で対応する属性は `tabIndex` です。例外は `aria-*` と `data-*` 属性であり、これらは全て小文字に揃える必要があります。例えば、`aria-label` は `aria-label` のままにできます。

## 属性についての差異

React と HTML で挙動が異なる属性がいくつか存在します。

### `checked`

`checked` 属性はインプットタイプが `checkbox` または `radio` の `<input>` コンポーネントでサポートされています。コンポーネントがチェックされた状態かどうかの設定に、この属性を使うことができます。これは制御されたコンポーネント (controlled component) を構築する際に役立ちます。`defaultChecked` は非制御コンポーネント (uncontrolled component) において同様の働きをする属性で、そのコンポーネントが最初にマウントされた時に、チェックされた状態かどうかを設定します。

### `className`

CSS クラスを指定するには、`className` 属性を使用してください。このことは `<div>`、`<a>` など全ての標準 DOM 要素と SVG 要素に当てはまります。

React を（一般的ではありませんが）Web Components とともに使用する場合は、代わりに `class` 属性を使用してください。

### `dangerouslySetInnerHTML`

`dangerouslySetInnerHTML` は、ブラウザ DOM における `innerHTML` の React での代替です。一般に、コードから HTML を設定することは、誤ってあなたのユーザをクロスサイトスクリプティング (XSS) 攻撃に晒してしまいやさしいため、危険です。そのため、React では直接 HTML を設定することはできますが、それは危険であることを自覚するために `dangerouslySetInnerHTML` と入力し `__html` というキーを持つオブジェクトを渡す必要があります。例えば：

```
function createMarkup() {
  return {__html: 'First &nbsp; Second'};
}

function MyComponent() {
  return <div dangerouslySetInnerHTML={createMarkup()} />;
}
```

### `htmlFor`

`for` は JavaScript での予約語であるため、React 要素では代わりに `htmlFor` を使用します。

## onChange

`onChange` イベントはあなたが期待しているような挙動をします。フォームフィールドに変更があるたび、このイベントが発生します。React の `onChange` という名前は既存のブラウザの挙動に対しては適切な名前では無く、React はリアルタイムでのユーザ入力を扱うためにこのイベントに依存しているため、React では意図的に既存のブラウザの挙動は使用していません。

## selected

`<option>` 要素を選択済みとしてマークしたい場合、その `option` の `value` を対応する `<select>` の `value` として参照してください。 詳細は [select タグ](#) を確認してください。

## style

### 補足

このドキュメンテーションにあるいくつかの例では `style` を便宜上使用していますが、**style 属性を要素のスタイルリングの主要な手段として使うことは一般的に推奨されません。**多くの場合、`className` を使って外部の CSS スタイルシートに定義された CSS クラスを参照するべきです。React アプリケーションの中では、`style` は動的に計算されたスタイルをレンダー中に追加するために最もよく使われます。FAQ: Styling and CSS も参照してください。

`style` 属性は CSS 文字列ではなく、キャメルケースのプロパティを持った JavaScript オブジェクトを受け取ります。これは JavaScript での DOM の `style` プロパティとの一貫性があり、より効率的で、XSS 攻撃の対象となるセキュリティホールを防ぎます。例えば：

```
const divStyle = {
  color: 'blue',
  backgroundImage: 'url(' + imgUrl + ')',
};

function HelloWorldComponent() {
  return <div style={divStyle}>Hello World!</div>;
}
```

ベンダープレフィックスの自動追加は行われないことに注意してください。古いブラウザをサポートするには、対応するスタイルのプロパティを与える必要があります：

```
const divStyle = {
  WebkitTransition: 'all', // ここでは大文字の 'W' です
  msTransition: 'all' // 'ms' は小文字である必要がある唯一のベンダープレフィックスです
};

function ComponentWithTransition() {
  return <div style={divStyle}>This should work cross-browser</div>;
}
```

JavaScript から DOM ノードのプロパティにアクセスする場合（例えば `node.style.backgroundImage`）と一致させるために、スタイルのキー名はキャメルケースです。`ms` 以外のベンダープレフィックスは先頭を大文字にしてください。`WebkitTransition` に大文字 "W" があるのはこのためです。

React はインラインスタイルでの特定の数値プロパティに対して自動的に "px" サフィックスを付け加えます。"px" 以外の単位を使用したい場合は、その単位を付け加えた文字列で値を指定してください。例えば：

```
// 最終的なスタイルは '10px'  
<div style={{ height: 10 }}>  
  Hello World!  
</div>  
  
// 最終的なスタイルは '10%'  
<div style={{ height: '10%' }}>  
  Hello World!  
</div>
```

全てのスタイルプロパティがピクセル指定に変換されるわけではありません。特定のプロパティ（例えば `zoom`、`order`、`flex`）は単位が無いままとなります。単位の無いプロパティの完全なリストは[こちら](#)で確認できます。

## suppressContentEditableWarning

子要素を持つ要素に `contentEditable` 属性が付与されている場合、それは動作しないため通常は警告が出力されます。この属性は、その警告が出力されないようにします。`contentEditable` を自身で管理している [Draft.js](#) のようなライブラリを開発するときでもない限り、この属性は使用しないでください。

## suppressHydrationWarning

サーバサイドの React レンダリングを使用している場合、サーバとクライアントが違う内容をレンダーする時に通常は警告が出力されます。しかし、まれに両者の内容が完全に一致することの保証が非常に困難あるいは不可能な場合があります。例えば、サーバとクライアントでは、タイムスタンプは異なることが予想されます。

`suppressHydrationWarning` を `true` に設定した場合、その要素の属性と内容の、サーバとクライアントでの差異について React は警告しません。この機能は単一レベルの深さでのみ動作し、避難ハッチとして使われることが想定されています。そのため、むやみに使用しないでください。ハイドレーション機能の詳細については `ReactDOM.hydrateRoot()` のドキュメントで読むことができます。

## value

`value` 属性は `<input>`、`<select>` および `<textarea>` コンポーネントでサポートされています。コンポーネントの値を設定することに使用できます。これは制御されたコンポーネントを構築する際に役立ちます。`defaultValue` は非制御コンポーネントにおいて同様の働きをする属性で、コンポーネントが最初にマウントされた時の値を設定します。

## サポートされている全ての HTML 属性

React 16 では、標準あるいは独自の DOM 属性全てが完全にサポートされます。

React は DOM に対して JavaScript 中心に設計された API を常に提供してきました。React コンポーネントは、独自および DOM に関連した `props` を頻繁に受け取るため、React は DOM API と同様にキャメルケース (`camelCase`) の命名規則を属性の名前付けに使用します。

```
<div tabIndex={-1} /> // DOM API の node.tabIndex と同様に
<div className="Button" /> // DOM API の node.className と同様に
<input readOnly={true} /> // DOM API の node.readOnly と同様に
```

このような props は、これまでドキュメントで述べられてきた特殊な例外を除き、対応する HTML 属性と同様に機能します。

React でサポートされている DOM 属性には、以下が含まれます：

```
accept acceptCharset accessKey action allowFullScreen alt autoComplete
autoFocus autoPlay capture cellPadding cellSpacing challenge charSet checked
cite classID className colSpan cols content contentEditable contextMenu controls
controlsList coords crossOrigin data dateTime default defer dir disabled
download draggable encType form formAction formEncType formMethod formNoValidate
formTarget frameBorder headers height hidden high href hrefLang htmlFor
httpEquiv icon id inputMode integrity is keyParams keyType kind label lang list
loop low manifest marginHeight marginWidth max maxLength media mediaGroup method
min minLength multiple muted name noValidate nonce open optimum pattern
placeholder poster preload profile radioGroup readOnly rel required reversed
role rows rowSpan rows sandbox scope scoped scrolling seamless selected shape size
sizes span spellCheck src srcDoc srcLang srcSet start step style summary
tabIndex target title type useMap value width wmode wrap
```

同様に、全ての SVG 属性を完全にサポートしています：

```
accentHeight accumulate additive alignmentBaseline allowReorder alphabetic
amplitude arabicForm ascent attributeName attributeType autoReverse azimuth
baseFrequency baseProfile baselineShift bbox begin bias by calcMode capHeight
clip clipPath clipPathUnits clipRule colorInterpolation
colorInterpolationFilters colorProfile colorRendering contentScriptType
contentStyleType cursor cx cy d decelerate descent diffuseConstant direction
display divisor dominantBaseline dur dx dy edgeMode elevation enableBackground
end exponent externalResourcesRequired fill fillOpacity fillRule filter
filterRes filterUnits floodColor floodOpacity focusable fontFamily fontSize
fontSizeAdjust fontStretch fontStyle fontVariant fontWeight format from fx fy
g1 g2 glyphName glyphOrientationHorizontal glyphOrientationVertical glyphRef
gradientTransform gradientUnits hanging horizAdvX horizOriginX ideographic
imageRendering in in2 intercept k k1 k2 k3 k4 kernelMatrix kernelUnitLength
kerning keyPoints keySplines keyTimes lengthAdjust letterSpacing lightingColor
limitingConeAngle local markerEnd markerHeight markerMid markerStart
markerUnits markerWidth mask maskContentUnits maskUnits mathematical mode
numOctaves offset opacity operator order orient orientation origin overflow
overlinePosition overlineThickness paintOrder panose1 pathLength
patternContentUnits patternTransform patternUnits pointerEvents points
pointsAtX pointsAtY pointsAtZ preserveAlpha preserveAspectRatio primitiveUnits
r radius refX refY renderingIntent repeatCount repeatDur requiredExtensions
requiredFeatures restart result rotate rx ry scale seed shapeRendering slope
spacing specularConstant specularExponent speed spreadMethod startOffset
stdDeviation stemh stemv stitchTiles stopColor stopOpacity
strikethroughPosition strikethroughThickness string stroke strokeDasharray
strokeDashoffset strokeLinecap strokeLinejoin strokeMiterlimit strokeOpacity
strokeWidth surfaceScale systemLanguage tableValues targetX targetY textAnchor
textDecoration textLength textRendering to transform u1 u2 underlinePosition
underlineThickness unicode unicodeBidi unicodeRange unitsPerEm vAlphabetic
vHanging vIdeographic vMathematical values vectorEffect version vertAdvY
vertOriginX vertOriginY viewBox viewTarget visibility widths wordSpacing
writingMode x x1 x2 xChannelSelector xHeight xlinkActuate xlinkArcrole
xlinkHref xlinkRole xlinkShow xlinkTitle xlinkType xmlns xmlnsXlink xmlBase
xmlLang xmlSpace y y1 y2 yChannelSelector z zoomAndPan
```

独自の属性も、その名前が全て小文字であれば使用できます。

---

[Go to TOC](#)

# 合成イベント (SyntheticEvent)

このリファレンスガイドでは、React のイベントシステムの一部を構成する `SyntheticEvent` (合成イベント) ラッパについて説明します。詳細については、[イベント処理ガイド](#)を参照してください。

## 概要

イベントハンドラには、`SyntheticEvent` のインスタンスが渡されます。これはブラウザのネイティブイベントに対するクロスブラウザ版のラッパです。`stopPropagation()` と `preventDefault()` を含む、ブラウザのネイティブイベントと同じインターフェイスを持ちつつ、ブラウザ間で同じ挙動をするようになっています。

何らかの理由で実際のブラウザイベントが必要な場合は、単に `nativeEvent` 属性を使用するだけで取得できます。合成イベントはブラウザのネイティブイベントとは別物であり、直接の対応があるわけでもありません。例えば `onMouseLeave` イベントの場合、`event.nativeEvent` は `mouseout` イベントになっています。個々の対応については公開 API の範疇ではなく、常に変わる可能性があります。すべての `SyntheticEvent` オブジェクトは以下の属性を持っています。

```
boolean bubbles
boolean cancelable
DOMEventTarget currentTarget
boolean defaultPrevented
number eventPhase
boolean isTrusted
DOMEvent nativeEvent
void preventDefault()
boolean isDefaultPrevented()
void stopPropagation()
boolean isPropagationStopped()
void persist()
DOMEventTarget target
number timeStamp
string type
```

### 補足

v17 以降、`SyntheticEvent` は [プリング](#)されなくなったため、`e.persist()` は何も行わなくなります。

### 補足

v0.14 以降、イベントハンドラから `false` を返してもイベントの伝播は止まりません。代わりに、`e.stopPropagation()` または `e.preventDefault()` を手動で呼び出す必要があります。

## サポートするイベント

React は異なるブラウザ間でも一貫したプロパティを持つようにイベントを正規化します。

以下のイベントハンドラはイベント伝搬のバブリングフェーズで呼び出されます。キャプチャフェーズのイベントハンドラを登録するには、イベント名に `Capture` を追加します。たとえば、キャプチャフェーズでクリックイベントを処理するには `onClick` の代わりに `onClickCapture` を使用します。

- クリップボードイベント
  - コンポジションイベント
  - キーボードイベント
  - フォーカスイベント
  - フォームイベント
  - 汎用イベント
  - マウスイベント
  - ポイントイベント
  - 選択イベント
  - タッチイベント
  - UI イベント
  - ホイールイベント
  - メディアイベント
  - 画像イベント
  - アニメーションイベント
  - 遷移イベント
  - その他のイベント
- 

## リファレンス

### クリップボードイベント

イベント名：

`onCopy onCut onPaste`

プロパティ：

`DOMDataTransfer clipboardData`

---

### コンポジションイベント

イベント名：

`onCompositionEnd onCompositionStart onCompositionUpdate`

プロパティ：

`string data`

---

## キーボードイベント

イベント名：

`onKeyDown onKeyPress onKeyUp`

プロパティ：

```
boolean altKey
number charCode
boolean ctrlKey
boolean getModifierState(key)
string key
number keyCode
string locale
number location
boolean metaKey
boolean repeat
boolean shiftKey
number which
```

`key` プロパティは [DOM Level 3 Events spec](#) に記載されている任意の値を取ることができます。

## フォーカスイベント

イベント名：

`onFocus onBlur`

これらのフォーカスイベントは、フォーム要素だけでなくすべての React DOM 要素で動作します。

プロパティ：

`DOMEventTarget relatedTarget`

### onFocus

`onFocus` イベントは要素（あるいはその内部の別の要素）がフォーカスを受け取った時に呼び出されます。例えば、ユーザがテキスト入力をクリックした際に呼び出されます。

```
function Example() {
  return (
    <input
      onFocus={(e) => {
        console.log('Focused on input');
     }}
      placeholder="onFocus is triggered when you click this input."
    />
  )
}
```

## onBlur

`onBlur` イベントハンドラは要素（あるいはその内部の別の要素）からフォーカスが外れた場合に呼び出されます。例えば、ユーザが既にフォーカスされているテキスト入力の外側でクリックした場合に呼び出されます。

```
function Example() {
  return (
    <input
      onBlur={(e) => {
        console.log('Triggered because this input lost focus');
      }}
      placeholder="onBlur is triggered when you click this input and then you
      click outside of it."
    />
  )
}
```

## フォーカスが当たった・外れたことの検出

`currentTarget` と `relatedTarget` を用いることで、フォーカスが当たった・外れた際のイベントが親要素の外側で起きたかどうかを判定できます。以下のコピー・ペーストで使えるデモでは、子要素のどれかへのフォーカス、要素自身へのフォーカス、サブツリー全体から出入りするフォーカスを、それぞれどのように検出するかを示しています。

```
function Example() {
  return (
    <div
      tabIndex={1}
      onFocus={(e) => {
        if (e.currentTarget === e.target) {
          console.log('focused self');
        } else {
          console.log('focused child', e.target);
        }
      }}
      onBlur={(e) => {
        if (e.currentTarget === e.target) {
          console.log('unfocused self');
        } else {
          console.log('unfocused child', e.target);
        }
        if (!e.currentTarget.contains(e.relatedTarget)) {
          // Not triggered when swapping focus between children
          console.log('focus left self');
        }
      }}
    >
      <input id="1" />
      <input id="2" />
    </div>
  );
}
```

## フォームイベント

イベント名：

`onChange onInput onInvalid onReset onSubmit`

`onChange` イベントの詳細については、[Forms](#) を参照してください。

## 汎用イベント

イベント名：

`onError onLoad`

## マウスイベント

イベント名：

`onClick onContextMenu onDoubleClick onDrag onDragEnd onDragEnter onDragExit  
onDragLeave onDragOver onDragStart onDrop onMouseDown onMouseEnter onMouseLeave  
onMouseMove onMouseOut onMouseOver onMouseUp`

`onMouseEnter` と `onMouseLeave` イベントは通常のバーリングとは異なり、(ポインタが) 出て行った要素から入ってきた要素に伝播し、キャプチャフェーズを持ちません。

プロパティ：

```
boolean altKey
number button
number buttons
number clientX
number clientY
boolean ctrlKey
boolean getModifierState(key)
boolean metaKey
number pageX
number pageY
DOMEventTarget relatedTarget
number screenX
number screenY
boolean shiftKey
```

## ポインタイベント

イベント名：

`onPointerDown onPointerMove onPointerUp onPointerCancel onGotPointerCapture  
onLostPointerCapture onPointerEnter onPointerLeave onPointerOver onPointerOut`

`onPointerEnter` と `onPointerLeave` イベントは通常のバーリングとは異なり、(ポインタが) 出て行った要素から入ってきた要素に伝播し、キャプチャフェーズを持ちません。

プロパティ：

[W3 spec](#) に定義されている通り、ポイントイベントは下記のプロパティを持つマウスイベントの拡張です。

```
number pointerId
number width
number height
number pressure
number tangentialPressure
number tiltX
number tiltY
number twist
string pointerType
boolean isPrimary
```

クロスブラウザサポートについての補足：

すべてのブラウザでポイントイベントがサポートされているわけではありません（この記事の執筆時点でサポートされているブラウザは、Chrome、Firefox、Edge、および Internet Explorer です）。標準に準拠したポリフィルは [react-dom](#) のバンドルサイズを大幅に増加させるため、React は意図的にその他ブラウザのためのポリフィルを提供しません。

アプリケーションでポイントイベントが必要な場合は、サードパーティのポイントイベントポリフィルを追加することをお勧めします。

## 選択イベント

イベント名：

```
onSelect
```

## タッチイベント

イベント名：

```
onTouchCancel onTouchEnd onTouchMove onTouchStart
```

プロパティ：

```
boolean altKey
DOMTouchList changedTouches
boolean ctrlKey
boolean getModifierState(key)
boolean metaKey
boolean shiftKey
DOMTouchList targetTouches
DOMTouchList touches
```

## UI イベント

イベント名：

## onScroll

補足

React 17 以降、`onScroll` イベントはパブルしなくなりました。これはブラウザの挙動と合致しており、スクロール可能な要素がネストされている場合に離れた親要素に対してイベントが発火する場合に起きる混乱を回避できます。

プロパティ：

`number detail`  
`DOMAbstractView view`

## ホイールイベント

イベント名：

### onWheel

プロパティ：

`number deltaMode`  
`number deltaX`  
`number deltaY`  
`number deltaZ`

## メディアイベント

イベント名：

`onAbort` `onCanPlay` `onCanPlayThrough` `onDurationChange` `onEmptied` `onEncrypted`  
`onEnded` `onError` `onLoadedData` `onLoadedMetadata` `onLoadStart` `onPause` `onPlay`  
`onPlaying` `onProgress` `onRateChange` `onSeeked` `onSeeking` `onStalled` `onSuspend`  
`onTimeUpdate` `onVolumeChange` `onWaiting`

## 画像イベント

イベント名：

`onLoad` `onError`

## アニメーションイベント

イベント名：

`onAnimationStart` `onAnimationEnd` `onAnimationIteration`

プロパティ：

```
string animationName  
string pseudoElement  
float elapsedTime
```

## 遷移イベント

イベント名：

```
onTransitionEnd
```

プロパティ：

```
string propertyName  
string pseudoElement  
float elapsedTime
```

## その他のイベント

イベント名：

```
onToggle
```

---

[Go to TOC](#)

# React 用語集

## シングルページアプリケーション

シングルページアプリケーション (single-page application) は、単一の HTML ページでアプリケーションの実行に必要なすべてのアセット (JavaScript や CSS など) をロードするようなアプリケーションです。初回ページもしくはそれ以降のページでのユーザとのやりとりにおいて、サーバとの往復が不要、すなわちページのリロードが発生しません。

React でシングルページアプリケーションを構築することができますが、そうすることは必須ではありません。React は、既存のウェブサイトの小さな一部分を拡張してよりインタラクティブにするために使用することもできます。React で記述されたコードは、PHP などによりサーバでレンダリングされたマークアップや、他のクライアントサイドのライブラリと問題なく共存できます。実際に、Facebook では React はそのように利用されています。

## ES6、ES2015、ES2016 など

これらの頭字語は全て、ECMAScript 言語の標準仕様の最近のバージョンのことを指しており、JavaScript 言語とはこれらの仕様に対する実装です。ES6 バージョン (ES2015 とも呼ばれます) はそれ以前のバージョンから、多くのものが追加されています：アロー関数、クラス、テンプレートリテラル、`let` および `const` ステートメントなどです。特定のバージョンの詳細については[こちら](#)で学ぶことができます。

## コンパイラ

JavaScript コンパイラは JavaScript コードを受け取って変換し、別のフォーマットの JavaScript コードを返します。最も一般的なユースケースは ES6 構文を受け取り、古いブラウザが解釈できる構文に変換することです。Babel は React とともに利用されることが最も多いコンパイラです。

## バンドラ

バンドラは別々のモジュールとして記述された（しばしば数百個になる）JavaScript および CSS のコードを受け取り、ブラウザに最適化された数個のファイルに結合します。[Webpack](#) や [Browserify](#) を含む、いくつかのバンドラが React アプリケーションで一般的に利用されています。

## パッケージマネージャ

パッケージ マネージャーは、プロジェクト内の依存関係を管理するためのツールです。[npm](#) および [Yarn](#) の 2 つのパッケージマネージャが React アプリケーションで一般的に利用されています。どちらも同じ npm パッケージレジストリのクライアントです。

## CDN

CDN は Content Delivery Network の略です。CDN はキャッシュされた静的なコンテンツをネットワーク化されたサーバから世界中に配信します。

# JSX

JSX は JavaScript の拡張構文です。テンプレート言語に似ていますが、完全に JavaScript だけで動作します。JSX は、"React 要素" と呼ばれるプレーンな JavaScript オブジェクトを返す、`React.createElement()` のコールにコンパイルされます。JSX の基本的な導入部分を学ぶには[こちらのドキュメントを参照してください](#)。また、さらに JSX について詳細に学ぶには[こちら](#)を参照してください。

React DOM は HTML の属性名ではなく、キャメルケースの命名規則を使用します。例えば、`tabindex` は、JSX では`tabIndex`となります。`class` も `className` と記述されますが、これは `class` が JavaScript において予約語であるためです：

```
<h1 className="hello">My name is Clementine!</h1>
```

## 要素 {#elements}

React 要素は React アプリケーションを構成するブロックです。要素を、より広く知られている概念である "コンポーネント" と混同する人もいるかもしれません。要素はあなたが画面上に表示したいものの説明書きとなるものです。React 要素はイミュータブルです。

```
const element = <h1>Hello, world</h1>;
```

通常、要素は直接使用されるものではなく、コンポーネントから返されるものです。

## コンポーネント {#components}

React のコンポーネントとは、ページに表示される React 要素を返す、小さく再利用可能なコードのことです。もっともシンプルな形の React コンポーネントは、React 要素を返すプレーンな JavaScript 関数です：

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

コンポーネントは ES6 クラスであることもあります：

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

コンポーネントは機能的に異なる部品に分割でき、他のコンポーネントの中で使用することができます。コンポーネントは別のコンポーネント、配列、文字列および数値を返すことができます。役に立つ経験則として、UI の一部（Button、Panel、Avatar など）が複数回使われている場合、または UI の一部が独立できるほど複雑（App、FeedStory、Comment など）な場合、それらは再利用可能なコンポーネントの有力な候補であるといえます。コンポーネント名は常に大文字で始める必要があります（`<wrapper/>` ではなく `<Wrapper/>` とすること）。コンポーネントのレンダーについての詳細は[このドキュメント](#)を参照してください。

## props {#props}

`props` は、React コンポーネントへの入力のことです。親コンポーネントから子コンポーネントへと順番に渡されるようなデータです。

`props` は読み取り専用です。どのような形であれ変更されるべきではありません：

```
// 間違った例
props.number = 42;
```

ある値を、ユーザの入力やネットワークのレスポンスに応じて変更する必要がある場合は、代わりに `state` を使用してください。

## props.children {#propschildren}

`props.children` は全てのコンポーネントで使用可能です。これには、コンポーネントの開始タグと終了タグの間の全てのコンテンツが含まれています。例えば：

```
<Welcome>Hello world!</Welcome>
```

文字列 `Hello world!` は `Welcome` コンポーネントの `props.children` で利用できます：

```
function Welcome(props) {
  return <p>{props.children}</p>;
}
```

クラスとして定義されたコンポーネントでは、`this.props.children` を使用してください：

```
class Welcome extends React.Component {
  render() {
    return <p>{this.props.children}</p>;
  }
}
```

## state {#state}

あるコンポーネントが時間とともに変化するデータと関連付けられている場合は、`state` が必要です。例えば、`Checkbox` というコンポーネントはその `state` に `isChecked` が必要となるかもしれません。また `NewsFeed` というコンポーネントは `fetchedPosts` を `state` に入れて管理したいかもしれません。

`state` と `props` の最も重要な違いは、`props` は親コンポーネントから渡されますが、`state` はコンポーネント自身によって管理されるということです。コンポーネントは自身の `props` を変更できませんが、`state` を変更することはできます。

経時的に変化するそれぞれのデータについて、それを `state` として「所有する」コンポーネントは 1 つだけであるべきです。2 つの異なるコンポーネントの `state` を同期しようとしてください。代わりに、それらの直近の共通祖先コンポーネントに `state` をリフトアップして、両方に `props` として渡してください。

## ライフサイクルメソッド {#lifecycle-methods}

ライフサイクルメソッドは、コンポーネントの様々なフェーズにおいて実行される特別な関数です。コンポーネントが作成されて DOM に挿入（マウント）された時、コンポーネントが更新された時、コンポーネントが DOM からアンマウント（つまり削除）された時のそれぞれで、利用可能なメソッドがあります。

## 制御されたコンポーネント vs. 非制御コンポーネント

React では、フォームの入力を扱うのに 2 つの異なるアプローチがあります。

React によって値が制御される入力フォーム要素は制御されたコンポーネント (controlled component) と呼ばれます。ユーザが制御されたコンポーネントにデータを入力すると、変更イベントハンドラがトリガーされ、コードが入力が有効であるか（更新された値で再レンダーすることで）決定します。再レンダーしなければフォーム要素は変更されないままとなります。

非制御コンポーネント (uncontrolled component) は React の管理外にあるフォーム要素と同様に動作します。ユーザがフォームフィールド（入力ボックス、ドロップダウンなど）にデータを入力した場合、更新された情報が反映され、その際に React は何もする必要がありません。しかし、このことはフィールドに特定の値を設定できないということでもあります。

ほとんどの場合では、制御されたコンポーネントを使用するべきでしょう。

## key {#keys}

"key" は特別な文字列の属性で、要素の配列を作成する際に含めておく必要があります。key は React がどの要素が変更、追加もしくは削除されたかを識別するのに役立ちます。key は配列内の要素に安定した一意性を与えるよう設定されるべきです。

key は同じ配列内の兄弟要素間で一意としなければなりません。アプリケーション全体、単一のコンポーネントに渡ってすべて一意である必要はありません。

`Math.random()` のようなものを key として設定しないでください。key には再レンダリングをまたいだ「安定した一意性」を持たせることで、要素の追加、削除および並べ替えがあった時に React が識別できることが重要です。理想的には key は `post.id` のように、データから得られる一意で安定した識別子に対応するべきです。

## ref {#refs}

React は任意のコンポーネントに追加できる特別な属性をサポートしています。`ref` 属性は、`React.createRef()` 関数、コールバック関数、あるいは（古い API では）文字列によって生成されるオブジェクトです。`ref` 属性がコールバック関数の場合、その関数は引数として（要素の種類によって）DOM 要素またはクラスインスタンスを受け取ります。これによって、DOM 要素またはコンポーネントのインスタンスへと直接アクセスできます。

`ref` は消極的に利用してください。アプリケーション内で何かを実行するために `ref` を頻繁に使用している場合、[トップダウンのデータフロー](#)に慣れ親しむことを検討してください。

## イベント {#events}

React 要素でイベントを扱う場合には、構文上の違いがあります：

- React のイベントハンドラは小文字ではなく、キャメルケースで名前が付けられます。
- JSX ではイベントハンドラとして文字列ではなく関数を渡します。

## リコンシリエーション(更新検出処理) {#reconciliation}

コンポーネントの `props` か `state` が変更された場合、React は新しく返された要素を以前にレンダーされた要素と比較することで、本物の DOM の更新が必要かを判断します。それらが等しくない場合、React は DOM を更新します。このプロセスを "リコンシリエーション (reconciliation)" と呼びます。

# JavaScript 環境の要件

React 18 はすべてのモダンブラウザ（Edge, Firefox, Chrome, Safari など）をサポートします。

モダンなブラウザ機能についてネイティブ実装していないか非標準な実装をしている Internet Explorer のような古いブラウザをサポートする場合は、バンドルされたアプリにグローバルなポリフィルを含めることを検討してください。

以下が、React 18 で使われるモダンな機能の一覧です：

- `Promise`
- `Symbol`
- `Object.assign`

これらの機能を正しくポリフィルする方法はあなたの環境によって異なります。多くのユーザは、[Browserlist](#) を設定すれば大丈夫です。人によっては `core-js` のようなポリフィルを直接インポートする必要があるかもしれません。

# プロファイラ API

`Profiler` を使って、React アプリケーションのレンダーの頻度やレンダーの「コスト」を計測することができます。本機能の目的は、アプリケーション中の、低速で**モ化などの最適化**が有効な可能性のある部位を特定する手助けをすることです。

補足:

プロファイリングには追加のオーバーヘッドが生じますので、[本番ビルトでは無効化されます](#)。

本番環境でプロファイリングを利用するため、React はプロファイリングを有効化した特別な本番用ビルトを提供しています。このビルトの使用方法については [fb.me/react-profiling](https://fb.me/react-profiling) をご覧ください。

## 使用法

`Profiler` は React ツリー内の特定部位におけるレンダーのコストを計測するため、ツリー内のどこにでも追加できます。2 つの props が必要です。`id` (文字列) と、ツリー内のコンポーネントが更新を「コミット」した際に React が毎回呼び出す `onRender` コールバック (関数) です。

例えば、`Navigation` コンポーネントとその子孫のプロファイリングを行うには：

```
render(
  <App>
    <Profiler id="Navigation" onRender={callback}>
      <Navigation {...props} />
    </Profiler>
    <Main {...props} />
  </App>
);
```

アプリケーション内の複数部位の計測を行うために複数の `Profiler` コンポーネントを使うことができます：

```
render(
  <App>
    <Profiler id="Navigation" onRender={callback}>
      <Navigation {...props} />
    </Profiler>
    <Profiler id="Main" onRender={callback}>
      <Main {...props} />
    </Profiler>
  </App>
);
```

同一のサブツリー内の複数のコンポーネントで計測を行うために `Profiler` コンポーネントをネストすることもできます：

```
render(
  <App>
    <Profiler id="Panel" onRender={callback}>
```

```

<Panel {...props}>
  <Profiler id="Content" onRender={callback}>
    <Content {...props} />
  </Profiler>
  <Profiler id="PreviewPane" onRender={callback}>
    <PreviewPane {...props} />
  </Profiler>
</Panel>
</Profiler>
</App>
);

```

### 補足

`Profiler` は軽いコンポーネントですが、必要な時にのみ利用すべきです。使うごとにアプリケーションに多少の CPU およびメモリオーバーヘッドが生じます。

## onRender コールバック

`Profiler` には `props` として `onRender` 関数を渡す必要があります。プロファイリングされているツリー内のコンポーネントが更新を「コミット」した際に、React がこの関数を毎回呼び出します。この関数は、レンダー内容とかった時間に関する情報を引数として受け取ります。

```

function onRenderCallback(
  id, // the "id" prop of the Profiler tree that has just committed
  phase, // either "mount" (if the tree just mounted) or "update" (if it re-
  rendered)
  actualDuration, // time spent rendering the committed update
  baseDuration, // estimated time to render the entire subtree without memoization
  startTime, // when React began rendering this update
  commitTime, // when React committed this update
  interactions // the Set of interactions belonging to this update
) {
  // Aggregate or log render timings...
}

```

それを詳細に見てみましょう：

コミットが起きた `Profiler` の `id` プロパティ。複数のプロファイルを使用している場合にどのツリーにコミットが起きたのかを区別するのに使うことができます。

- `phase: "mount" | "update"` - ツリーが初回マウントされたのか、`props` や `state`、フックの変更によって再レンダーアップされたのかを区別します。
- `actualDuration: number` - 現在の更新で `Profiler` とその子孫のレンダーに要した時間。これが (`React.memo`、`useMemo`、`shouldComponentUpdate` などの) メモ化をどれだけうまく有効に使っているかの指標となります。理想的には、子孫要素は特定の `props` が変化した場合にのみ再レンダーアップされるため、初回マウント時以降にこの値は大幅に小さくなるはずです。
- `baseDuration: number` - `Profiler` ツリー内のそれぞれのコンポーネントの直近の `render` 時間。この値を使って最悪の場合のレンダーコスト（初回マウント時や、メモ化の一切ないツリーの場合）を見積もることができます。
- `startTime: number` - 現在の更新のレンダーを React が開始した時刻に対応するタイムスタンプ。

- `commitTime: number` - 現在の更新を React がコミットした時刻に対応するタイムスタンプ。必要に応じてグループ化できるよう、1 コミット内のすべてのプロファイラ間でこの値は共有されます。
- `interactions: Set` - 更新がスケジュールされた (`render` や `setState` の呼び出しなどにより) 際に `trace` された "interaction" の Set。

#### 補足

更新の原因を特定するために `interaction` を利用可能ですが、`trace` 用の API は依然実験的です。

[fb.me/react-interaction-tracing](https://fb.me/react-interaction-tracing) に詳細があります。

# PureRenderMixin

## Note

The `PureRenderMixin` mixin predates `React.PureComponent`. This reference doc is provided for legacy purposes, and you should consider using `React.PureComponent` instead.

If your React component's render function renders the same result given the same props and state, you can use this mixin for a performance boost in some cases.

Example:

```
var PureRenderMixin = require('react-addons-pure-render-mixin');
var createReactClass = require('create-react-class');
createReactClass({
  mixins: [PureRenderMixin],

  render: function() {
    return <div className={this.props.className}>foo</div>;
  }
});
```

Example using ES6 class syntax:

```
import PureRenderMixin from 'react-addons-pure-render-mixin';
class FooComponent extends React.Component {
  constructor(props) {
    super(props);
    this.shouldComponentUpdate = PureRenderMixin.shouldComponentUpdate.bind(this);
  }

  render() {
    return <div className={this.props.className}>foo</div>;
  }
}
```

Under the hood, the mixin implements `shouldComponentUpdate`, in which it compares the current props and state with the next ones and returns `false` if the equalities pass.

## Note:

This only shallowly compares the objects. If these contain complex data structures, it may produce false-negatives for deeper differences. Only mix into components which have simple props and state, or use `forceUpdate()` when you know deep data structures have changed. Or, consider using [immutable objects](#) to facilitate fast comparisons of nested data.

Furthermore, `shouldComponentUpdate` skips updates for the whole component subtree. Make sure all the children components are also "pure".

---

[Go to TOC](#)

# React.Component

このページには React コンポーネントクラス定義の詳細な API リファレンスがあります。また、あなたが [コンポーネント](#) や [props](#) などの基本的な React の概念、および [state](#) や [ライフサイクル](#) に精通していることを前提としています。そうでない場合は、まずそれらを読んでください。

## 概要

React では、コンポーネントをクラスまたは関数として定義できます。クラスとして定義されたコンポーネントは現在このページで詳細に説明されているより多くの機能を提供します。React コンポーネントクラスを定義するには、`React.Component` を継承する必要があります。

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

`React.Component` サブクラスで必ず定義しなければならない唯一のメソッドは `render()` です。このページで説明されている他のすべてのメソッドは任意です。

**独自の基底コンポーネントクラスを作成しないことを強くおすすめします。** React コンポーネントでは、[コードの再利用](#) は主に継承ではなく合成によって行われます。

補足:

React は ES6 クラスの構文を使うことを強制していません。回避したい場合は、代わりに `create-react-class` モジュールまたは、同様の独自の抽象化を使用できます。詳しくは、[Using React without ES6](#) をご覧ください。

## コンポーネントライフサイクル

各コンポーネントには、処理の過程の特定の時点でコードを実行するためにオーバーライドできるいくつかの「ライフサイクルメソッド」があります。この[ライフサイクル図](#)をチートシートとして使用できます。以下のリストでは、よく使われるライフサイクルメソッドは**太字**で表示されています。それらの残りは比較的まれなユースケースのために存在します。

### マウント

コンポーネントのインスタンスが作成されて DOM に挿入されるときに、これらのメソッドが次の順序で呼び出されます。

- `constructor()`
- `static getDerivedStateFromProps()`
- `render()`
- `componentDidMount()`

**補足:**

このメソッドはレガシーだと考えられているため、新しいコードでは[避けるべきです。](#)

- `UNSAFE_componentWillMount()`

**更新**

更新は `props` や `state` の変更によって発生する可能性があります。コンポーネントが再レンダーされるときに、これらのメソッドは次の順序で呼び出されます。

- `static getDerivedStateFromProps()`
- `shouldComponentUpdate()`
- `render()`
- `getSnapshotBeforeUpdate()`
- `componentDidUpdate()`

**補足:**

これらのメソッドはレガシーと見なされ、新しいコードでは避けるべきです。

- `UNSAFE_componentWillUpdate()`
- `UNSAFE_componentWillReceiveProps()`

**アンマウント**

このメソッドは、コンポーネントが DOM から削除されるときに呼び出されます。

- `componentWillUnmount()`

**エラーハンドリング**

これらのメソッドは任意の子コンポーネントのレンダー中、ライフサイクルメソッド内、またはコンストラクタ内でエラーが発生したときに呼び出されます。

- `static getDerivedStateFromError()`
- `componentDidCatch()`

**他の API**

各コンポーネントはその他にもいくつかの API を提供します。

- `setState()`
- `forceUpdate()`

**クラスプロパティ**

- `defaultProps`
- `displayName`

## インスタンスプロパティ

- `props`
- `state`

## リファレンス

### よく使われるライフサイクルメソッド

このセクションのメソッドは、React コンポーネントを作成する際に遭遇する大部分のユースケースを網羅しています。視覚的なリファレンスとして、この[ライフサイクル図](#)もチェックしてみてください。

#### `render()` {#render}

##### `render()`

`render()` メソッドは、クラスコンポーネントで必ず定義しなければならない唯一のメソッドです。

呼び出されると、`this.props` と `this.state` を調べて、次のいずれかの型を返します。

- **React 要素** 通常は [JSX](#) 経由で作成されます。例えば、`<div />` や `<MyComponent />` はそれぞれ React に DOM ノードやユーザが定義した他のコンポーネントをレンダーするように指示する React 要素です
- **配列とフラグメント** 複数の要素を `render()` から返します。詳しくは [フラグメント](#) を参照してください
- **ポータル** 子を異なる DOM サブツリーにレンダーします。詳しくは [ポータル](#) を参照してください
- **文字列と数値** これは DOM のテキストノードとしてレンダーされます
- **真偽値または null** 何もレンダーしません。(ほとんどの場合、`return test && <Child />` パターンをサポートするために存在しています。ここで、`test` は真偽値です)

`render()` 関数は「純粋」でなければなりません。つまり、コンポーネントの `state` を変更せず、呼び出されるたびに同じ結果を返し、ブラウザと直接対話しないということです。

ブラウザと対話する必要がある場合は、代わりに `componentDidMount()` や他のライフサイクルメソッドで行います。`render()` を純粋にしておくことで、コンポーネントについて考えやすくなります。

##### 補足

`shouldComponentUpdate()` が `false` を返した場合、`render()` は呼び出されません。

#### `constructor()` {#constructor}

##### `constructor(props)`

`state` の初期化もメソッドのバインドもしないのであれば、React コンポーネントのコンストラクタを実装する必要はありません。

React コンポーネントのコンストラクタは、マウントされる前に呼び出されます。`React.Component` サブクラスのコンストラクタを実装するときは、他の文の前に `super(props)` を呼び出す必要があります。そうでなければ、`this.props` はコンストラクタ内で未定義になり、バグの原因となる可能性があります。

通常、React では、コンストラクタは 2 つの目的にのみ使用されます。

- `this.state` にオブジェクトを代入して ローカル state を初期化すること
- イベントハンドラ をインスタンスにバインドすること

`constructor()` の中で `setState()` を呼び出さないでください。代わりに、コンポーネントがローカル state を使用する必要がある場合は、コンストラクタで直接 `this.state` に初期状態を割り当てます。

```
constructor(props) {
  super(props);
  // ここで this.setState() を呼び出さないでください
  this.state = { counter: 0 };
  this.handleClick = this.handleClick.bind(this);
}
```

コンストラクタは、`this.state` を直接代入する唯一の場所です。他のすべてのメソッドでは、代わりに `this.setState()` を使う必要があります。

コンストラクタに副作用や購読を導入しないでください。そのような場合は、代わりに `componentDidMount()` を使用してください。

## 補足

**props を state にコピーしないでください。これはよくある間違いです。**

```
constructor(props) {
  super(props);
  // してはいけません
  this.state = { color: props.color };
}
```

この問題はそれが不要(代わりに `this.props.color` を直接使用することができるため)であり、バグの作成につながる(`color` プロパティの更新は `state` に反映されないため)ことです。

**意図的にプロパティの更新を無視したい場合にのみ、このパターンを使用してください。** その場合は、プロパティの名前を `initialColor` または `defaultColor` に変更してください。その後、必要に応じてキーを変更することで、コンポーネントにその内部の `state` を「リセット」させることができます。

もしあなたが `props` に依存する何らかの `state` が必要だと思うなら、どうすればいいのか学ぶために私達の[派生 state を避けることについてのブログ記事](#)を読んでください。

## componentDidMount() {#componentdidmount}

`componentDidMount()`

`componentDidMount()` は、コンポーネントがマウントされた（ツリーに挿入された）直後に呼び出されます。DOM ノードを必要とする初期化はここで行われるべきです。リモートエンドポイントからデータをロードする必要がある場合、これはネットワークリクエストを送信するのに適した場所です。

このメソッドは、購読を設定するのに適した場所です。設定した場合は、`componentWillUnmount()` で購読を解除することを忘れないでください。

`componentDidMount()` の中で、あなたはすぐに `setState()` を呼び出すことができます。それは余分なレンダーを引き起こしますが、ブラウザが画面を更新する前に起ります。これにより、この場合 `render()` が 2 回呼び出されても、ユーザには中間状態が表示されません。このパターンはパフォーマンス上の問題を引き起こすことが多いので、慎重に使用してください。ほとんどの場合、代わりに `constructor()` で初期状態を `state` に代入できるはずです。ただし、モーダルやツールチップのような場合に、サイズや位置に応じて何かをレンダーする前に DOM ノードを測定することが必要になる場合があります。

## componentDidUpdate() {#componentdidupdate}

`componentDidUpdate(prevProps, prevState, snapshot)`

更新が行われた直後に `componentDidUpdate()` が呼び出されます。このメソッドは最初のレンダーでは呼び出されません。

コンポーネントが更新されたときに DOM を操作する機会にこれを使用してください。現在の `props` と前の `props` を比較している限り、これはネットワークリクエストを行うのにも適した場所です（たとえば、`props` が変更されていない場合、ネットワークリクエストは必要ないかもしれません）。

```
componentDidUpdate(prevProps) {
  // 典型的な使い方(props を比較することを忘れないでください)
  if (this.props.userID !== prevProps.userID) {
    this.fetchData(this.props.userID);
  }
}
```

`componentDidUpdate()` の中で、あなたはすぐに `setState()` を呼び出すことができますが、それは上記の例のような条件でラップされなければならないことに注意してください。そうしなければ、無限ループを引き起こすでしょう。また、余分な再レンダーが発生し、ユーザには見えないものの、コンポーネントのパフォーマンスに影響を与える可能性があります。親から来る `props` を何らかの `state` に「反映」しようとしている場合は、代わりに `props` を直接使用することを検討してください。[props を state にコピーするとバグが発生する理由](#)をよく読んでください。

コンポーネントが `getSnapshotBeforeUpdate()` ライフサイクルを実装している場合（これはまれです）、それが返す値は 3 番目の「スナップショット」パラメータとして `componentDidUpdate()` に渡されます。それ以外の場合、このパラメータは未定義になります。

### 補足

`shouldComponentUpdate()` が `false` を返した場合、`componentDidUpdate()` は呼び出されません。

## componentWillUnmount() {#componentwillunmount}

### componentWillUnmount()

`componentWillUnmount()` は、コンポーネントがアンマウントされて破棄される直前に呼び出されます。タイマーの無効化、ネットワークリクエストのキャンセル、`componentDidMount()` で作成された購読の解除など、このメソッドで必要なクリーンアップを実行します。

コンポーネントは再レンダーされないため、`componentWillUnmount()` で `setState()` を呼び出さないでください。コンポーネントインスタンスがアンマウントされると、再度マウントされることはありません。

## まれに使われるライフサイクルメソッド

このセクションのメソッドは、一般的でないユースケースに対応しています。これらは時々便利に使えますが、あなたのコンポーネントのほとんどはおそらくそれらのどれも必要としないでしょう。この[ライフサイクル図](#)の上部にある「あまり一般的ではないライフサイクルを表示する」チェックボックスをクリックすると、以下のほとんどの方法が表示されます。

## shouldComponentUpdate() {#shouldcomponentupdate}

### shouldComponentUpdate(nextProps, nextState)

コンポーネントの出力が現在の `state` の変化や `props` の影響を受けていないかどうかを React に知らせるには `shouldComponentUpdate()` を使用します。デフォルトの振る舞いはすべての状態変化を再レンダーすることです、そして大部分の場合、あなたはデフォルトの振る舞いに頼るべきです。

新しい `props` または `state` が受け取られると、レンダーする前に `shouldComponentUpdate()` が呼び出されます。デフォルトは `true` です。このメソッドは最初のレンダーや `forceUpdate()` の使用時には呼び出されません。

このメソッドはパフォーマンスの最適化としてのみ存在します。バグを引き起こす可能性があるので、レンダーを「抑止する」ためにそれを使用しないでください。`shouldComponentUpdate()` を書く代わりに、組み込みの [PureComponent](#) を使用することを検討してください。`PureComponent` は `props` と `state` を浅く比較し、必要なアップデートをスキップする可能性を減らします。

あなたが手でそれを書きたいと確信しているなら、あなたは `nextProps` と `this.props` または `nextState` と `this.state` を比較して、更新をスキップできることを React に伝えるために `false` を返すことができます。`false` を返しても、子コンポーネントの `state` が変化したときに子コンポーネントが再レンダーされるのを防ぐことはできません。

等価性を深く調べることや `shouldComponentUpdate()` で `JSON.stringify()` を使用することはおすすめしません。これは非常に非効率的であり、パフォーマンスに悪影響を及ぼします。

現在、`shouldComponentUpdate()` が `false` を返す場合、`UNSAFE_componentWillUpdate()`、`render()`、および `componentDidUpdate()` は呼び出されません。将来的には、React は `shouldComponentUpdate()` を厳密な命令ではなくヒントとして扱うようになり、`false` を返してもコンポーネントが再レンダーされる可能性があります。

## static getDerivedStateFromProps() {#static-getderivedstatefromprops}

### static getDerivedStateFromProps(props, state)

`getDerivedStateFromProps` は、初期マウント時とその後の更新時の両方で、`render()` メソッドを呼び出す直前に呼び出されます。`state` を更新するためにオブジェクトを返すか、何も更新しない場合は `null` を返すべきです。

このメソッドは、`state` が時間の経過とともに変化する `props` に依存するような [まれな使用例](#) のために存在します。たとえば、以前と以降の子を比較してどちらの子をアニメーションするかを決定する `<Transition>` コンポーネントを実装するときに便利です。

`state` を派生させると冗長なコードにつながり、コンポーネントを考えるのが難しくなります。 [より簡単な方法があるのでは](#) まずそちらに慣れるようにしてください。

- `props` の変更に応じて [副作用を実行する](#) 必要がある場合は（データのフェッチやアニメーションなど）、代わりに `componentDidUpdate` ライフサイクルを使用してください
- **プロパティが変更されたときにのみデータを再計算したい** 場合は、[代わりにメモ化ヘルパーを使用してください](#)
- **プロパティが変更されたときに何かの `state` を「リセット」したい** 場合は、代わりにコンポーネントを[完全に制御する](#)か、または `key` を使って[全く制御しない](#)かを検討してください

このメソッドはコンポーネントインスタンスにアクセスできません。必要に応じて、コンポーネントの `props` と `state` の純粋な関数を抽出し、クラス定義外に記述することで、`getDerivedStateFromProps()` と他のクラスメソッドの間でコードを再利用できます。

このメソッドは、原因に関係なく、すべてのレンダーで起動されることに注意してください。これは `UNSAFE_componentWillReceiveProps` とは対照的です。`UNSAFE_componentWillReceiveProps` は、ローカルの `setState()` に依らず、親が再レンダーを行ったときにのみ発生します。

## getSnapshotBeforeUpdate() {#getsnapshotbeforeupdate}

### getSnapshotBeforeUpdate(prevProps, prevState)

`getSnapshotBeforeUpdate()` は、最後にレンダーされた出力が DOM などにコミットされる直前に呼び出されます。これはコンポーネントが変更される可能性があるときに、変更される前に DOM から何らかの情報（たとえば、スクロール位置）を取得しておくことを可能にします。このライフサイクルメソッドによって返された値はすべて、`componentDidUpdate()` へのパラメータとして渡されます。

このユースケースは一般的ではありませんが、スクロール位置を特別な方法で処理する必要があるチャットのスレッドのような UI で発生する可能性があります。

スナップショット値（または `null`）が返されるべきです。

例：

```
embed:react-component-reference/get-snapshot-before-update.js
```

この例では、`getSnapshotBeforeUpdate` の中で `scrollHeight` プロパティを読み取ることが重要です。これは、（`render` のような）「描画」フェーズライフサイクルと（`getSnapshotBeforeUpdate` および `componentDidUpdate` のような）「コミット」フェーズライフサイクルの間に遅延が生じるためです。

## error boundary

[error boundary](#) は、子コンポーネントツリーのどこかで JavaScript エラーを捕捉し、それらのエラーを記録し、クラッシュしたコンポーネントツリーの代わりにフォールバック UI を表示する React コンポーネントです。error boundary は、その下のツリー全体のレンダー中、ライフサイクルメソッド内、およびコンストラクタ内で発生したエラーを捕捉します。

クラスコンポーネントは、ライフサイクルメソッド `static getDerivedStateFromError()` または `componentDidCatch()` のいずれか（または両方）を定義すると、error boundary になります。これらのライフサイクルから state を更新すると、下のツリーで発生した未処理の JavaScript エラーを捕捉してフォールバック UI を表示できます。

error boundary は予期しない例外からの回復のためだけに使用してください。**それらを制御フローに使用しないでください。**

詳細については、[React 16 のエラーハンドリング](#)を参照してください。

### 補足

error boundary は、ツリー内でその下にあるコンポーネント内のエラーのみを捕捉します。error boundary はそれ自体の中でエラーを捉えることはできません。

## static getDerivedStateFromError() {#static-getderivedstatefromerror}

### `static getDerivedStateFromError(error)`

このライフサイクルは、子孫コンポーネントによってエラーがスローされた後に呼び出されます。パラメータとしてスローされたエラーを受け取り、state を更新するための値を返すべきです。

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    // 次のレンダーでフォールバック UI が表示されるように state を更新する
    return { hasError: true };
  }

  render() {
    if (this.state.hasError) {
      // 任意のフォールバック UI をレンダーできます
      return <h1>Something went wrong.</h1>;
    }

    return this.props.children;
  }
}
```

### 補足

`getDerivedStateFromError()` は「描画」フェーズ中に呼び出されるので、副作用は許可されません。そのような場合は、代わりに `componentDidCatch()` を使用してください。

## componentDidCatch() {#componentdidcatch}

### componentDidCatch(error, info)

このライフサイクルは、子孫コンポーネントによってエラーがスローされた後に呼び出されます。以下の 2 つのパラメータを受け取ります。

1. `error` - スローされたエラー
2. `info` - どのコンポーネントがエラーをスローしたかについての情報を含む `componentStack` キーを持つオブジェクト

`componentDidCatch()` は「コミット」フェーズ中に呼び出されるため、副作用は許可されています。ロギング時のエラーなどのために使用されるべきです。

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    // 次のレンダーでフォールバック UI が表示されるように state を更新します
    return { hasError: true };
  }

  componentDidCatch(error, info) {
    // Example "componentStack":
    //   in ComponentThatThrows (created by App)
    //   in ErrorBoundary (created by App)
    //   in div (created by App)
    //   in App
    logComponentStackToMyService(info.componentStack);
  }

  render() {
    if (this.state.hasError) {
      // 任意のフォールバック UI をレンダーできます
      return <h1>Something went wrong.</h1>;
    }

    return this.props.children;
  }
}
```

React の本番用ビルドと開発用ビルドでは、`componentDidCatch()` がエラーを処理する方法がわずかに異なります。

開発用ビルドでは、エラーは `window` までバーピングするため、`componentDidCatch()` が捕捉したエラーを `window.onerror` や `window.addEventListener('error', callback)` でもインターceptすることができます。

一方で本番用ビルドではエラーはバーピングしないため、祖先要素のエラーハンドラは `componentDidCatch()` で明示的に捕捉されていないエラーのみを受け取ります。

### 補足

エラーが発生した場合は、`setState` を呼び出す `componentDidCatch()` を使用してフォールバック UI をレンダーできますが、これは将来のリリースでは推奨されなくなります。代わりにフォールバックの描画を扱うために、`static getDerivedStateFromError()` を使用してください。

## レガシーなライフサイクルメソッド

以下のライフサイクルメソッドは「レガシー」としてマークされています。動作はしますが、新しいコードで使用することはおすすめしません。この[ブログ記事](#)では、レガシーなライフサイクルメソッドからの移行についてさらに学ぶことができます。

### `UNSAFE_componentWillMount()` {#unsafe\_componentwillmount}

#### `UNSAFE_componentWillMount()`

### 補足

このライフサイクルは、以前は `componentWillMount` という名前でした。その名前はバージョン 17 まで機能し続けます。コンポーネントを自動的に更新するには、`rename-unsafe-lifecycles` [codemod](#) を使用してください。

マウントが行われる直前に `UNSAFE_componentWillMount()` が呼び出されます。これは `render()` の前に呼び出されるので、このメソッドで `setState()` を同期的に呼び出しても余分なレンダリングは行われません。一般に、`state` を初期化するためには代わりに `constructor()` を使うことをおすすめします。

このメソッドでは、副作用や読み取りを導入しないでください。そのような場合は、代わりに `componentDidMount()` を使用してください。

これは、サーバレンダリングで呼び出される唯一のライフサイクルメソッドです。

### `UNSAFE_componentWillReceiveProps()` {#unsafe\_componentwillreceiveprops}

#### `UNSAFE_componentWillReceiveProps(nextProps)`

### 補足

このライフサイクルは、以前は `componentWillReceiveProps` という名前でした。その名前はバージョン 17 まで機能し続けます。コンポーネントを自動的に更新するには、`rename-unsafe-lifecycles` [codemod](#) を使用してください。

**補足:**

このライフサイクルメソッドを使用すると、しばしばバグや矛盾が発生します。

- `props` の変更に応じて副作用を実行する必要がある場合は（データの取得やアニメーションなど）、代わりに `componentDidUpdate` ライフサイクルを使用してください
- `componentWillReceiveProps` を `props` が変更されたときにのみデータを再計算するために使う代わりに、[モ化ヘルパーを使用してください](#)
- `componentWillReceiveProps` を `props` が変更されたときに何かの `state` を「リセット」するために使う代わりに、コンポーネントを完全に制御するか、または `key` を使って全く制御しないかを検討してください

その他の使用例については、[派生 state に関するこのブログ投稿の推奨事項に従ってください](#)。

`UNSAFE_componentWillReceiveProps()` は、マウントされたコンポーネントが新しい `props` を受け取る前に呼び出されます。`prop` の変更に応じて `state` を更新する必要がある場合（たとえばリセットする必要がある場合）は、`this.props` と `nextProps` を比較し、このメソッドで `this.setState()` を使用して状態遷移を実行できます。

親コンポーネントによってコンポーネントが再レンダーされる場合、`props` が変更されていなくてもこのメソッドが呼び出されることに注意してください。変更だけを処理したい場合は、必ず現在の値と次の値を比較してください。

マウント時に、React は最初の `props` で `UNSAFE_componentWillReceiveProps()` を呼び出しません。一部のコンポーネントの `props` が更新される可能性がある場合にのみ、このメソッドを呼び出します。`this.setState()` を呼び出しても、通常 `UNSAFE_componentWillReceiveProps()` は呼び出されません。

## UNSAFE\_componentWillUpdate() {#unsafe\_componentwillupdate}

### UNSAFE\_componentWillUpdate(nextProps, nextState)

**補足**

このライフサイクルは、以前は `componentWillUpdate` と呼ばれていました。その名前はバージョン 17 まで機能し続けます。コンポーネントを自動的に更新するには、[rename-unsafe-lifecycles codemod](#) を使用してください。

`UNSAFE_componentWillUpdate()` は、新しい `props` または `state` を受け取ったときにレンダーの直前に呼び出されます。更新が発生する前の準備する機会としてこれを使用してください。このメソッドは最初のレンダーでは呼び出されません。

ここで `this.setState()` を呼び出すことはできません。また、`UNSAFE_componentWillUpdate()` が返る前に React コンポーネントへの更新を引き起こすような何か他のこと（たとえば、Redux アクションのディスパッチ）をするべきでもありません。

通常、このメソッドは `componentDidUpdate()` に置き換えることができます。このメソッドで DOM から読んでいる場合（スクロール位置を保存するなど）は、そのロジックを `getSnapshotBeforeUpdate()` に移動できます。

**補足**

`shouldComponentUpdate()` が `false` を返した場合、`UNSAFE_componentWillUpdate()` は呼び出されません。

## 他の API

上記のライフサイクルメソッド（React が呼び出すもの）とは異なり、以下のメソッドはあなたがコンポーネントから呼び出すことができるメソッドです。

それは `setState()` と `forceUpdate()` の 2 つだけです。

### setState() `{#setstate}`

`setState(updater[, callback])`

`setState()` はコンポーネントの `state` への変更をエンキューし、このコンポーネントとその子を更新された `state` で再レンダーする必要があることを React に伝えます。これは、イベントハンドラとサーバの応答に応じてユーザインターフェイスを更新するために使用する主な方法です。

`setState()` は、コンポーネントを更新するための即時のコマンドではなく、**要求**として考えてください。パフォーマンスをよくするために、React はそれを遅らせて、単一パスで複数のコンポーネントを更新することができます。DOM の更新が同期的に適用される必要がある稀なケースにおいては、更新を `flushSync` でラップすることができますが、これを使うとパフォーマンスが低下する可能性があります。

`setState()` は常にコンポーネントを直ちに更新するわけではありません。それはバッチ式に更新するか後で更新を延期するかもしれません。これは `setState()` を呼び出した直後に `this.state` を読み取ることが潜在的な危険になります。代わりに、`componentDidUpdate` または `setState` コールバック (`setState(updater, callback)`) を使用してください。どちらも更新が適用された後に起動することが保証されています。前の `state` に基づいて `state` を設定する必要がある場合は、下記の `updater` 引数についてお読みください。

`shouldComponentUpdate()` が `false` を返さない限り、`setState()` は常に再レンダーされます。ミュータブルなオブジェクトが使用されていて、条件付きでレンダーを行うためのロジックを `shouldComponentUpdate()` に実装できない場合、新しい `state` が前の `state` と異なるときにのみ `setState()` を呼び出すと、不要な再レンダーを回避できます。

最初の引数の `updater` 関数は次のようなシグネチャです。

`(state, props) => stateChange`

`state` は、変更が適用されているときのコンポーネントの `state` への参照です。直接変更するべきではありません。代わりに、`state` と `props` からの入力に基づいて新しいオブジェクトを構築することによって変更を表現する必要があります。たとえば、`props.step` によって `state` の値を増加したいとします。

```
this.setState((state, props) => {
  return {counter: state.counter + props.step};
});
```

`updater` 関数が受け取る `state` と `props` の両方が最新のものであることが保証されています。アップデータの出力は `state` と浅くマージされています。

`setState()` の 2 番目のパラメータは、`setState` が完了してコンポーネントが再レンダーアされると実行される省略可能なコールバック関数です。通常、そのようなロジックには代わりに `componentDidUpdate()` を使用することをおすすめします。

関数の代わりに、オブジェクトを `setState()` の最初の引数として渡すこともできます。

```
  setState(stateChange[, callback])
```

これは、たとえば、ショッピングカートの商品数を調整するために、`stateChange` の新しい state への浅いマージを実行します。

```
  this.setState({quantity: 2})
```

この形式の `setState()` も非同期であり、同じサイクル中の複数の呼び出しをまとめてバッチ処理することができます。たとえば、同じサイクルで品目数量を複数回増やそうとすると、次のようにになります。

```
  Object.assign(
  previousState,
  {quantity: state.quantity + 1},
  {quantity: state.quantity + 1},
  ...
)
```

後続の呼び出しは、同じサイクル内の前の呼び出しの値を上書きするため、数量は 1 回だけ増分されます。次の state が現在の state に依存する場合は、代わりに `updater` 関数の形式を使用することをおすすめします。

```
  this.setState((state) => {
  return {quantity: state.quantity + 1};
});
```

詳しくは以下を参照してください。

- `state` と ライフサイクル
- 詳細: `setState()` はいつ、なぜバッチ処理されるのですか？
- 詳細: `this.state` が直ちに更新されないのはなぜですか？

## forceUpdate() {#forceupdate}

```
  component.forceUpdate(callback)
```

デフォルトでは、コンポーネントの `state` や `props` が変わると、コンポーネントは再レンダーアれます。`render()` メソッドが他のデータに依存している場合は、`forceUpdate()` を呼び出してコンポーネントの再レンダーアが必要であることを React に伝えることができます。

`forceUpdate()` を呼び出すと、`shouldComponentUpdate()` をスキップして、コンポーネントに対して `render()` が呼び出されます。これにより、それぞれの子の `shouldComponentUpdate()` メソッドを含む、子コンポーネントの通常のライフサイクルメソッドがトリガーされます。マークアップが変更された場合にのみ React は DOM を更新します。

通常は全ての `forceUpdate()` の使用を避け、`render()` の `this.props` と `this.state` からのみ読み取るようにしてください。

---

## クラスプロパティ

### `defaultProps {#defaultprops}`

`defaultProps` は、コンポーネントクラス自体のプロパティとして定義して、そのクラスのデフォルトの `props` を設定できます。これは `undefined` であるプロパティに使用されますが、`null` であるプロパティには使用されません。例えば：

```
class CustomButton extends React.Component {
  // ...
}

CustomButton.defaultProps = {
  color: 'blue'
};
```

`props.color` が提供されていない場合は、デフォルトで `'blue'` に設定されます。

```
render() {
  return <CustomButton />; // props.color は blue にセットされます
}
```

`props.color` が `null` に設定されている場合、それは `null` のままになります。

```
render() {
  return <CustomButton color={null} />; // props.color は null のままになります
}
```

### `displayName {#displayname}`

`displayName` 文字列はデバッグメッセージに使用されます。通常、コンポーネントを定義する関数またはクラスの名前から推測されるため、明示的に設定する必要はありません。デバッグ目的で別の名前を表示する場合や、高階コンポーネントを作成する場合には、明示的に設定したくなるかもしれません。詳細については、[簡単なデバッグのために表示名をラップする](#)を参照してください。

---

## インスタンスプロパティ

### `props {#props}`

`this.props` には、このコンポーネントの呼び出し元によって定義された `props` が含まれています。`props` の紹介は [コンポーネントと props](#) を見てください。

特に、`this.props.children` は特別なプロパティで、通常はタグ自体ではなく JSX 式の子タグによって定義されます。

## state { #state }

state には、そのコンポーネント固有のデータが含まれており、これは時間の経過とともに変化する可能性があります。state はユーザ定義のものであり、プレーンな JavaScript オブジェクトでなければなりません。

レンダーやデータフローに値が使用されていない場合（たとえば、タイマー ID）は、値を state にする必要はありません。そのような値は、コンポーネントインスタンスのフィールドとして定義できます。

state の詳細については、[state とライフサイクル](#)を参照してください。

後で `setState()` を呼び出すと、行った変更が置き換えられる可能性があるため、`this.state` を直接変更しないでください。`this.state` がイミュータブルであるかのように扱ってください。

# ReactDOMClient

`react-dom/client` パッケージはクライアント側でアプリを初期化する際に使うクライアント専用メソッドを提供しています。ほとんどのコンポーネントではこのモジュールを使う必要はありません。

```
import * as ReactDOM from 'react-dom/client';
```

npm で ES5 を使っている場合、以下のように書けます：

```
var ReactDOM = require('react-dom/client');
```

## 概要

クライアント環境では以下のメソッドが利用可能です：

- `createRoot()`
- `hydrateRoot()`

## ブラウザサポート

React はすべてのモダンブラウザをサポートしていますが、古いバージョンを使う場合は[いくつかのポリフィルが必要](#)です。

### 補足

React は ES5 メソッドやマイクロタスクをサポートしていない Internet Explorer のような古いブラウザをサポートしていません。ページ内に `es5-shim` や `es5-sham` のようなポリフィルが含まれている場合、古いブラウザでもアプリケーションが動作することがありますが、この手段を選択するかどうかは自己責任で判断してください。

## リファレンス

### `createRoot() {#createroot}`

```
createRoot(container[, options]);
```

渡された `container` に対する React ルートを作成してそれを返します。root の `render` を使って DOM 内部に React 要素をレンダーできます：

```
const root = createRoot(container);
root.render(element);
```

`createRoot` は 2 つのオプションを受け取ります：

- `onRecoverableError` : React が自動的にエラーから復帰した際に呼ばれるオプションのコールバック。

- `identifierPrefix`: `React.useId` によって ID が生成される際に React が使うオプションのプリフィックス。同じページに複数のルートがある場合に競合を避けるために有用です。サーバ側で使うプリフィックスと同一である必要があります。

ルートは `unmount` でアンマウントすることができます :

```
root.unmount();
```

補足 :

`createRoot()` は渡されたコンテナノードの中身を管理します。コンテナ内部にある既存のあらゆる DOM 要素は `render` がコールされた時点で置き換えられます。後続のコールでは React の DOM 差分アルゴリズムを使って効率的な更新を行います。

`createRoot()` はコンテナノードを変更しません（コンテナの子要素のみ変更します）。既存の子要素を上書きせずにコンポーネントを既存の DOM ノードに挿入することが可能な場合があります。

`createRoot()` を使ってサーバでレンダーされたコンテナにハイドレーションを行うことはできません。代わりに `hydrateRoot()` を使ってください。

## hydrateRoot() {#hydrateroot}

```
hydrateRoot(container, element[, options])
```

`createRoot()` と同様ですが、`ReactDOMServer` により HTML コンテンツが描画されたコンテナをクライアントで再利用する（ハイドレーション）ために使用されます。React は既存のマークアップにイベントリスナーをアタッチしようとします。

`hydrateRoot` は 2 つのオプションを受け取ります :

- `onRecoverableError`: React が自動的にエラーから復帰した際に呼ばれるオプションのコールバック。
- `identifierPrefix`: `React.useId` によって ID が生成される際に React が使うオプションのプリフィックス。同じページに複数のルートがある場合に競合を避けるために有用です。サーバ側で使うプリフィックスと同一である必要があります。

補足

React はレンダーされる内容が、サーバ・クライアント間で同一であることを期待します。React はテキストコンテンツの差異を修復することは可能ですが、その不一致はバグとして扱い、修正すべきです。開発用モードでは、React はハイドレーション時に起きた両者のレンダーの不一致について警告します。不一致がある場合に属性の差異が修復されるという保証はありません。これはパフォーマンス上の理由から重要です。なぜなら、ほとんどのアプリケーションにおいて不一致が発生するということは稀であり、全てのマークアップを検証することは許容不可能なほど高コストになるためです。

[Go to TOC](#)

# ReactDOMServer

`ReactDOMServer` オブジェクトはコンポーネントを静的なマークアップとして変換できるようにします。これは、一般的に Node サーバで使われます。

```
// ES modules
import * as ReactDOMServer from 'react-dom/server';
// CommonJS
var ReactDOMServer = require('react-dom/server');
```

## 概要

以下のメソッドは **Node.js の Stream 環境**でのみ動作します：

- `renderToPipeableStream()`
- `renderToNodeStream()` (非推奨)
- `renderToStaticNodeStream()`

以下のメソッドは **Web Stream の環境**（ブラウザ、Deno やいくつかのモダンなエッジランタイム）でのみ動作します：

- `renderToReadableStream()`

以下のメソッドはストリームをサポートしない環境でも動作します：

- `renderToString()`
- `renderToStaticMarkup()`

## リファレンス

### `renderToPipeableStream() {#rendertopipeablestream}`

`ReactDOMServer.renderToPipeableStream(element, options)`

React 要素を初期状態の HTML に変換します。出力をパイプするための `pipe(res)` メソッドとリクエストを中止するための `abort()` メソッドを持ったストリームを返します。サスペンスや、HTML をストリーミングして「遅れてやってくる」コンテンツのブロックをインライン `<script>` タグで埋めるための機能を完全にサポートしています。[詳細はこちら](#)。

このようにサーバでレンダーされたマークアップを有するノードに対して `ReactDOM.hydrateRoot()` をコールすると、React はマークアップを保持してイベントハンドラだけをアタッチしますので、非常によい初回ロード体験が実現できます。

```
let didError = false;
const stream = renderToPipeableStream(
  <App />,
  {
    onShellReady() {
      // The content above all Suspense boundaries is ready.
      // If something errored before we started streaming, we set the error code
    }
  }
);
```

```

appropriately.
    res.statusCode = didError ? 500 : 200;
    res.setHeader('Content-type', 'text/html');
    stream.pipe(res);
},
onShellError(error) {
    // Something errored before we could complete the shell so we emit an
    // alternative shell.
    res.statusCode = 500;
    res.send(
        '<!doctype html><p>Loading...</p><script src="clientrender.js"></script>'
    );
},
onAllReady() {
    // If you don't want streaming, use this instead of onShellReady.
    // This will fire after the entire page content is ready.
    // You can use this for crawlers or static generation.

    // res.statusCode = didError ? 500 : 200;
    // res.setHeader('Content-type', 'text/html');
    // stream.pipe(res);
},
onError(err) {
    didError = true;
    console.error(err);
},
}
);
}

```

オプションの全リストを参照。

補足：

これは Node.js 専用の API です。Deno やモダンなエッジランタイムのような [Web Stream](#) の環境では、代わりに `renderToReadableStream` を使用してください。

## renderToReadableStream() {#rendertoreadablestream}

`ReactDOMServer.renderToReadableStream(element, options);`

React 要素を初期状態の HTML にストリーミングします。[Readable Stream](#) として resolve する Promise を返します。サスペンスと HTML ストリーミングを完全にサポートします。[詳細はこちら](#)。

このようにサーバでレンダーされたマークアップを有するノードに対して `ReactDOM.hydrateRoot()` をコールすると、React はマークアップを保持してイベントハンドラだけをアタッチしますので、非常によい初回ロード体験が実現できます。

```

let controller = new AbortController();
let didError = false;
try {
    let stream = await renderToReadableStream(
        <html>
            <body>Success</body>

```

```

        </html>,
    {
      signal: controller.signal,
      onError(error) {
        didError = true;
        console.error(error);
      }
    }
);

// This is to wait for all Suspense boundaries to be ready. You can uncomment
// this line if you want to buffer the entire HTML instead of streaming it.
// You can use this for crawlers or static generation:

// await stream.allReady;

return new Response(stream, {
  status: didError ? 500 : 200,
  headers: {'Content-Type': 'text/html'},
});
} catch (error) {
  return new Response(
    '<!doctype html><p>Loading...</p><script src="clientrender.js"></script>',
    {
      status: 500,
      headers: {'Content-Type': 'text/html'},
    }
);
}
}

```

オプションの全リストを参照。

補足：

この API は [Web Streams](#) に依存しています。Node.js では代わりに `renderToPipeableStream` を使用してください。

## ReactDOMServer.renderToString() (非推奨)

`ReactDOMServer.renderToString(element)`

React 要素を初期状態の HTML にストリーミングします。HTML 文字列を出力する [Node.js の 読み取りストリーム](#) を返します。このストリームからの HTML 出力は `ReactDOMServer.renderToString` が返すものと全く同じです。このメソッドを使ってサーバで HTML を生成し、初回リクエスト時にマークアップを送信することで、ページロードを高速化し、SEO 目的でサーチエンジンがクロールできるようになります。

このようにサーバでレンダーされたマークアップを有するノードに対して `ReactDOM.hydrateRoot()` をコールすると、React はマークアップを保持してイベントハンドラだけをアタッチしますので、非常によい初回ロード体験が実現できます。

補足：

この API はサーバ専用です。ブラウザでは利用できません。

このメソッドが返すストリームは UTF-8 でエンコードされたバイトストリームを返します。別的方式でエンコードされたストリームが必要な場合、テキストのトランスコーディングのために変換ストリームを提供している [iconv-lite](#) のようなプロジェクトを参照してください。

## renderToString() {#rendertostring}

ReactDOMServer.renderToString(element)

React が内部的に使用する `data-reactroot` のような追加の DOM 属性を作成しないことを除いて、`renderToNodeStream` と同様の動作をします。このメソッドは React を単純な静的サイトジェネレータとして使用したい場合に便利で、追加の属性を省略することでバイト数を削減できます。

このストリームによる HTML 出力は `ReactDOMServer.renderToString` が返すものと全く同じです。

マークアップをインタラクティブなものにするために、クライアントで React を導入しようとしている場合は、このメソッドを使用しないでください。代わりに、サーバで `renderToNodeStream` を、そしてクライアントで `ReactDOM.hydrateRoot()` を使用してください。

補足：

この API はサーバ専用です。ブラウザでは利用できません。

このメソッドが返すストリームは UTF-8 でエンコードされたバイトストリームを返します。別的方式でエンコードされたストリームが必要な場合、テキストのトランスコーディングのために変換ストリームを提供している [iconv-lite](#) のようなプロジェクトを参照してください。

## renderToStaticNodeStream() {#rendertostaticnodestream}

ReactDOMServer.renderToStaticNodeStream(element)

React 要素を初期状態の HTML に変換します。React は HTML 文字列を返します。このメソッドを使ってサーバで HTML を生成し、初回リクエスト時にマークアップを送信することで、ページロードを高速化し、SEO 目的でサーチエンジンがクロールできるようになります。

このようにサーバでレンダーされたマークアップを有するノードに対して `ReactDOM.hydrateRoot()` をコールすると、React はマークアップを保持してイベントハンドラだけをアタッチしますので、非常に早い初回ロード体験が実現できます。

補足

この API は Suspense を部分的にしかサポートしておらず、ストリーミングが行えません。

サーバサイドでは、`renderToPipeableStream`（Node.js の場合）または `renderToReadableStream`（Web Stream の場合）の利用をお勧めします。

## renderToString() {#rendertostaticmarkup}

### ReactDOMServer.renderToString(element)

React が内部的に使用する `data-reactroot` のような追加の DOM 属性を作成しないことを除いて、`renderToString` と同様の動作をします。このメソッドは React を単純な静的サイトジェネレータとして使用したい場合に便利で、追加の属性を省略することでバイト数を削減できます。

クライアントで React を使ってマークアップをインタラクティブにする目的では、このメソッドは使わないでください。代わりにサーバで `renderToString`、クライアントで `ReactDOM.hydrateRoot()` を使うようにします。

---

[Go to TOC](#)

# ReactDOM

`react-dom` パッケージではアプリのトップレベルで使うための DOM 特有のメソッドを提供しています。また必要に応じて React モデルから外に出るための避難ハッチ用メソッドも提供しています。

```
import * as ReactDOM from 'react-dom';
```

npm で ES5 を使っている場合は、以下のように書けます：

```
var ReactDOM = require('react-dom');
```

`react-dom` パッケージはクライアント専用モジュールとサーバ専用モジュールも提供しています。

- `react-dom/client`
- `react-dom/server`

## 概要

`react-dom` パッケージは以下のメソッドをエクスポートしています：

- `createPortal()`
- `flushSync()`

以下の `react-dom` メソッドもエクスポートされていますが、これらはレガシーです：

- `render()`
- `hydrate()`
- `findDOMNode()`
- `unmountComponentAtNode()`

補足：

`render` と `hydrate` は React 18 でいずれも新しい [クライアントメソッド](#) に置き換わりました。これらのメソッドを使った場合、React 17 と同様の動作となってしまう、という警告を表示します（詳細は[こちら](#)）。

## ブラウザサポート

React はすべてのモダンブラウザをサポートしていますが、古いバージョンを使う場合は[いくつかのポリフィルが必要です](#)。

補足

React は ES5 メソッドやマイクロタスクをサポートしていない Internet Explorer のような古いブラウザをサポートしていません。ページ内に `es5-shim` や `es5-sham` のようなポリフィルが含まれている場合、古いブラウザでもアプリケーションが動作することがありますが、この手段を選択するかどうかは自己責任で判断してください。

## リファレンス

### `createPortal()` {#createportal}

`createPortal(child, container)`

ポータルを作成します。ポータルは DOM コンポーネントの階層の外側に存在している DOM ノードに対して子要素をレンダーする方法を提供します。

### `flushSync()` {#flushsync}

`flushSync(callback)`

渡されたコールバック内の更新をすべて同期的に処理するよう強制します。これにより DOM が即座に更新されることが保証されます。

```
// Force this state update to be synchronous.
flushSync(() => {
  setCount(count + 1);
});
// By this point, DOM is updated.
```

補足：

`flushSync` はパフォーマンスを有意に低下させうるもので、慎重に使ってください。

`flushSync` によって保留中のサスペンスバウンダリは強制的に `fallback` 状態を表示するようになります。

`flushSync` は `return` する前に保留中の副作用も実行し、その内部にあるあらゆる更新を同期的に適用します。

`flushSync` はコールバック内の更新を適用するのに必要であれば渡されたコールバック外の更新も処理する可能性があります。例えば、クリックによって引き起こされた保留中の更新が既に存在する場合、React はコールバック内の更新を処理する前に先にそれらを処理します。

## レガシーメソッドのリファレンス

### `render()` {#render}

`render(element, container[, callback])`

補足：

`render` は React 18 で `createRoot` に置き換わりました。詳細は [createRoot](#) を参照してください。

渡された `container` の DOM に React 要素をレンダーし、コンポーネントへの参照（ステートレスコンポーネントの場合は `null`）を返します。

React 要素がすでに `container` にレンダーされている場合は更新を行い、最新の React 要素を反映するために必要な DOM のみを変更します。

オプションのコールバックが渡されている場合は、コンポーネントがレンダーまたは更新された後に実行されます。

補足:

`render()` は与えられたコンテナの内容を制御します。コンテナ内部のあらゆる既存の DOM 要素は、最初に呼び出された時に置き換えられます。後続する呼び出しでは効率的な更新のために React の DOM 差分アルゴリズムを使用します。

`render()` はコンテナノードを変更しません（コンテナの子要素のみ変更します）。既存の子要素を上書きせずにコンポーネントを既存の DOM ノードに挿入することが可能な場合があります。

`render()` は現時点ではルートの `ReactComponent` インスタンスへの参照を返します。しかし、この戻り値を使用する方法は古く、将来のバージョンの React では一部のケースで非同期にコンポーネントをレンダーするようになる可能性があるため、使用は避けるべきです。ルートの `ReactComponent` インスタンスへの参照が必要な場合は、ルート要素に [コールバック形式の ref](#) を追加することを推奨します。

サーバで描画されたコンテナをクライアントで再利用するために `ReactDOM.render()` を使用することは非推奨となっています。代わりに `hydrateRoot()` を使用してください。

## hydrate() {#hydrate}

`hydrate(element, container[, callback])`

Note:

`hydrate` は React 18 で `hydrateRoot` に置き換わりました。詳細は [hydrateRoot](#) を参照してください。

`render()` と同様ですが、`ReactDOMServer` により HTML コンテンツが描画されたコンテナをクライアントで再利用する（ハイドレーション）ために使用されます。React は既存のマークアップにイベントリスナーをアタッチしようとします。

React はレンダーされる内容が、サーバ・クライアント間で同一であることを期待します。React はテキストコンテンツの差異を修復することは可能ですが、その不一致はバグとして扱い、修正すべきです。開発用モードでは、React は両者のレンダーの不一致について警告します。不一致がある場合に属性の差異が修復されるという保証はありません。これはパフォーマンス上の理由から重要です。なぜなら、ほとんどのアプリケーションにおいて不一致が発生するということは稀であり、全てのマークアップを検証することは許容不可能なほど高コストになるためです。

单一要素の属性やテキストコンテンツがサーバ・クライアント間においてやむを得ず異なってしまう場合（例えばタイムスタンプなど）、要素に `suppressHydrationWarning={true}` を追加することで警告の発生を停止させることができます。それは 1 階層下の要素まで機能するものであり、また避難ハッシュとして使われるものです。そのため、多用しないでください。テキストコンテンツでない限り、React は修復を試行しようとはしないため、将来の更新まで不整合が残る可能性があります。

サーバとクライアントで異なるものをレンダーしたい場合は、2 パスレンダーを使用できます。クライアント側で異なるものをレンダーするコンポーネントでは、`this.state.isClient` のような state 変数を読み込み、`componentDidMount()` で `true` を設定することができます。こうすると、最初のレンダーパスではサーバ側と同一の内容を描画して不一致を回避しますが、追加のパスが初回レンダーの直後に同期的に発生します。このアプローチでは 2 回レンダーが発生することによりコンポーネントのパフォーマンスが低下しますので、注意して使用してください。

低速な接続下でのユーザ体験に留意することを忘れないでください。JavaScript のコードは初回の HTML の描画より大幅に遅れてロードされる可能性があるため、クライアントでのみ何か異なるものを描画した場合、その変化は不快感を与える可能性があります。しかしうまく実行されれば、サーバ上でアプリケーションの「外枠」を描画し、クライアント上でのみ追加のウィジェットを表示することは有益になるかもしれません。マークアップの不一致の問題を発生させずにこれを実行する方法については、前の段落の説明をご参照ください。

## unmountComponentAtNode() {#unmountcomponentatnode}

`unmountComponentAtNode(container)`

補足：

`unmountComponentAtNode` は React 18 で `root.unmount()` に置き換わりました。詳細は [createRoot](#) を参照してください。

DOM からマウントされた React コンポーネントを削除し、イベントハンドラや state をクリーンアップします。コンテナにコンポーネントがマウントされていない場合、このメソッドを呼び出しても何も行いません。コンポーネントがアンマウントされた場合は `true` を返し、アンマウントすべきコンポーネントが存在しなかった場合は `false` を返します。

## findDOMNode() {#finddomnode}

補足：

`findDOMNode` は内在する DOM ノードにアクセスするために使用される避難ハッシュです。ほとんどのケースにおいて、この避難ハッシュの使用はコンポーネントの抽象化に穴を開けてしまうためおすすめしません。[StrictMode](#) では非推奨になっています。

`findDOMNode(component)`

DOM にこのコンポーネントがマウントされている場合、このメソッドは対応するネイティブブラウザの DOM 要素を返します。このメソッドはフォームフィールドの値や DOM の大きさを計測するのに便利です。**ほとんどのケースにおいて、DOM ノードに ref をアタッチすることで `findDOMNode` の使用を避けることができます。**

コンポーネントが `null` や `false` をレンダーする場合、`findDOMNode` は `null` を返します。コンポーネントが文字列をレンダーする場合、`findDOMNode` はその値を含んだテキスト DOM ノードを返します。React 16 以降、コンポーネントは複数の子要素を含むフラグメントを返すことがあります、その場合 `findDOMNode` は最初の空でない子要素に対応する DOM ノードを返します。

補足:

`findDOMNode` はマウントされたコンポーネントに対してのみ機能します（つまり、DOM に配置されたコンポーネント）。まだマウントされていないコンポーネントにおいてこのメソッドを呼びうとする場合（まだ作成されていないコンポーネントにおける `render()` の中で `findDOMNode()` を呼びだす場合など）、例外がスローされます。

`findDOMNode` は関数コンポーネントでは使用できません。

# React の最上位 API

`React` は React ライブラリのエントリーポイントです。`<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`

## トランジション

トランジションは React 18 で導入された新しい並行レンダー機能です。これにより更新をトランジションとしてマークすることができ、既に表示されているコンテンツがサスペンスによるフォールバック状態に戻ってしまわないよう更新を中断して構わない、と React に伝えることができるようになります。

- `React.startTransition`
- `React.useTransition`

## 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`
  - `useDeferredValue`
  - `useTransition`
  - `useId`
- [ライブラリ製作者用フック](#)
  - `useSyncExternalStore`
  - `useInsertionEffect`

## リファレンス

### React.Component {#reactcomponent}

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 {#reactpurecomponent}

`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 {#reactmemo}

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

`React.memo` は高階コンポーネントです。

もしある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。

`React.memo` は `props` の変更のみをチェックします。`React.memo` でラップしているあなたのコンポーネントがその実装内で `useState`、`useReducer` や `useContext` フックを使っている場合、`state` やコンテクストの変化に応じた再レンダーは発生します。

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() [\[#createelement\]](#)

```
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() {#cloneelement}

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

`element` から新しい React 要素を複製して返します。`config` には新しく使う `props` や `key` や `ref` を指定します。結果の要素は元の要素の `props` に新しい `props` が浅くマージされたものになります。新しい子要素は既存の子要素を置き換えます。`config` 内で `key` や `ref` が指定されていない場合、元の要素の `key` や `ref` が保持されます。

`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。新しい `ref` や `key` が存在する場合古いものを置き換えます。

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() {#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() {#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 {#reactchildren}

`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 {#reactchildrenmap}

```
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 {#reactchildrenforeach}

`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 {#reactchildrencount}

`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 {#reactchildrenonly}

`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 {#reactchildrentoarray}

`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 {#reactfragment}

`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 {#reactcreateref}

`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。 [embed:16-3-release-blog-post/create-ref-example.js](#)

## React.forwardRef {#reactforwardref}

`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。

[embed:reference-react-forward-ref.js](#)

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 {#reactlazy}

`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。

## React.Suspense {#reactsuspense}

`React.Suspense` を使用することで、その配下のツリー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表示するローディングインジケータを指定できます。将来的には `Suspense` をデータフェッチングのようなより多くのシナリオで使えるようにする予定です。詳細は[コードマップ](#)を参照してください。

現時点ではコンポーネントの遅延ローディングが `<React.Suspense>` がサポートする唯一のユースケースです：

```
// 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。

補足：

既にユーザに表示されているコンテンツがある場合、それがローディングインジケータに戻ってしまうのは不親切です。新しい UI を準備している間「古い」UI を表示しておくことが望ましいことがあります。これを行うため、新たなトランジション API である `startTransition` と `useTransition` を用い、更新をトランジションとしてマークすることで意図しない場面でのフォールバックを避けることができます。

## サーバサイドレンダリングでの React.Suspense {#reactsuspense-in-server-side-rendering}

サーバサイドレンダリングにおいてもサスペンスバウンダリとサスPENDを用いることで、アプリを部分的に分割して表示していくことができます。コンポーネントがサスPENDした場合、直近のサスペンスバウンダリに指定されているフォールバックをレンダーするような低優先度のタスクがスケジュールされます。フォールバックを表示する前にコンポーネントのサスPENDが解除された場合は、フォールバックのコンテンツを捨てて実コンテンツを送信します。

## ハイドレーション中の `React.Suspense` `{#reactsuspense-during-hydration}`

サスペンスバウンダリがハイドレートされる前に親のバウンダリはハイドレートされていなければなりませんが、兄弟の関係にあるバウンダリとは独立してハイドレートすることができます。何らかのバウンダリでイベントが起こった場合、そのバウンダリは他のものより優先的にハイドレートされるようになります。[詳細](#)

## React.startTransition `{#starttransition}`

`React.startTransition(callback)`

`React.startTransition` は渡されたコールバック内で発生した更新をトランジションとしてマークします。これは `React.useTransition` が使えない場合でも使えるように設計されています。

補足：

トランジション内での更新はクリックのようなより緊急性の高い更新があった場合に遅延されます。

トランジション内で起こった更新は、コンテンツが再サスPENDした場合でもフォールバックを表示させないため、更新をレンダーしている最中でもユーザが操作できる状態が保たれます。

`React.startTransition` は `isPending` フラグを返しません。トランジションのペンドィング状態を知るには `React.useTransition` を参照してください。

---

[Go to TOC](#)

# Test Renderer

## インポート

```
import TestRenderer from 'react-test-renderer'; // ES6
const TestRenderer = require('react-test-renderer'); // ES5 with npm
```

## 概要

このパッケージは、DOM やネイティブのモバイル環境に依存せずに React コンポーネントをピュアな JavaScript オブジェクトにレンダーすることができる React レンダラを提供します。

本質的にこのパッケージは、ブラウザや `jsdom` を利用しなくても、React DOM や React Native コンポーネントがレンダーする（DOM ツリーに似た）ビューの階層構造のスナップショットを容易に取得できるようにするためのものです。

例：

```
import TestRenderer from 'react-test-renderer';

function Link(props) {
  return <a href={props.page}>{props.children}</a>;
}

const testRenderer = TestRenderer.create(
  <Link page="https://www.facebook.com/">Facebook</Link>
);

console.log(testRenderer.toJSON());
// { type: 'a',
//   props: { href: 'https://www.facebook.com/' },
//   children: [ 'Facebook' ] }
```

JSON ツリーを自動的にファイルに保存し、変更が起こったかをテストで確認するには、Jest のスナップショットテスト機能が利用できます。[詳細について知る](#)

出力を走査して特定のノードを検索し、それらに対してアサーションを行うこともできます。

```
import TestRenderer from 'react-test-renderer';

function MyComponent() {
  return (
    <div>
      <SubComponent foo="bar" />
      <p className="my">Hello</p>
    </div>
  )
}

function SubComponent() {
  return (
    <p className="sub">Sub</p>
  );
}
```

```
const testRenderer = TestRenderer.create(<MyComponent />);
const testInstance = testRenderer.root;

expect(testInstance.findByType(SubComponent).props.foo).toBe('bar');
expect(testInstance.findByProps({className: "sub"}).children).toEqual(['Sub']);
```

## TestRenderer

- `TestRenderer.create()`
- `TestRenderer.act()`

## TestRenderer instance

- `testRenderer.toJSON()`
- `testRenderer.toTree()`
- `testRenderer.update()`
- `testRenderer.unmount()`
- `testRenderer.getInstance()`
- `testRenderer.root`

## TestInstance

- `testInstance.find()`
- `testInstance.findByType()`
- `testInstance.findByProps()`
- `testInstance.findAll()`
- `testInstance.findAllByType()`
- `testInstance.findAllByProps()`
- `testInstance.instance`
- `testInstance.type`
- `testInstance.props`
- `testInstance.parent`
- `testInstance.children`

## リファレンス

### TestRenderer.create() {#testrenderercreate}

```
TestRenderer.create(element, options);
```

渡された React 要素から `TestRenderer` インスタンスを作成します。実際の DOM は使用しませんが、コンポーネントを完全な形でメモリにレンダーするので、アサーションを行うことができます。返り値は `TestRenderer` のインスタンスとなります。

### TestRenderer.act() {#testrendereract}

```
TestRenderer.act(callback);
```

`TestRenderer.act` は `react-dom/test-utils` の `act` ヘルパーと同じように、アサーション用のコンポーネントを準備します。このバージョンの `act()` を使って、`TestRenderer.create` と `testRenderer.update` の呼び出しをラップします。

```
import {create, act} from 'react-test-renderer';
import App from './app.js'; // The component being tested

// render the component
let root;
act(() => {
  root = create(<App value={1}>)
});

// make assertions on root
expect(root.toJSON()).toMatchSnapshot();

// update with some different props
act(() => {
  root.update(<App value={2}>);
})

// make assertions on root
expect(root.toJSON()).toMatchSnapshot();
```

## testRenderer.toJSON() {#testrenderertojson}

`testRenderer.toJSON()`

レンダーされたツリーを表すオブジェクトを返します。このツリーは `<div>` もしくは `<View>` のようなプラットフォーム固有のノードとそのプロパティを含みますが、ユーザ定義のコンポーネントは含まれません。スナップショットテストに便利です。

## testRenderer.toTree() {#testrenderertotree}

`testRenderer.toTree()`

レンダーされたツリーを表すオブジェクトを返します。`toJSON()` とは異なり、このツリーはより詳細なものであり、ユーザ定義のコンポーネントも含んでいます。テストレンダラを利用して自作のアサーションライブラリを作成している場合以外は、おそらくこのメソッドが必要となることはないでしょう。

## testRenderer.update() {#testrendererupdate}

`testRenderer.update(element)`

メモリ上のツリーを新規のルート要素で再レンダーします。ルートでの React の更新をシミュレートします。新しい要素が以前の要素と同じ型と `key` を持つ場合は、ツリーは更新されます。それ以外の場合は新しいツリーを再マウントします。

## testRenderer.unmount() {#testrendererunmount}

`testRenderer.unmount()`

メモリ上のツリーをアンマウントし、適切なライフサイクルイベントを発生させます。

## testRenderer.getInstance() {#testrenderergetinstance}

```
testRenderer.getInstance()
```

ルート要素に対応したインスタンスがある場合はそれを返します。関数コンポーネントはインスタンスを持たないため、ルート要素が関数コンポーネントの場合、このメソッドはうまく動作しません。

## testRenderer.root {#testrendererroot}

```
testRenderer.root
```

ツリー上の特定のノードに対してアサーションを行う際に役立つ、ルート「テストインスタンス」を返します。これは、配下の他の「テストインスタンス」を検索する際に使用することができます。

## testInstance.find() {#testinstancefind}

```
testInstance.find(test)
```

`test(testInstance)` が `true` を返す単一の子テストインスタンスを検索します。もし `test(testInstance)` に対して `true` を返すテストインスタンスの数がちょうど 1 でない場合は、エラーがスローされます。

## testInstance.findByType() {#testinstancefindbytype}

```
testInstance.findByType(type)
```

与えられた `type` を持つ単一の子テストインスタンスを検索します。もし与えられた `type` を持つテストインスタンスの数がちょうど 1 でない場合、エラーがスローされます。

## testInstance.findByProps() {#testinstancefindbyprops}

```
testInstance.findByProps(props)
```

与えられた `props` を持つ単一の子テストインスタンスを検索します。もし与えられた `props` を持つテストインスタンスの数がちょうど 1 でない場合、エラーがスローされます。

## testInstance.findAll() {#testinstancefindall}

```
testInstance.findAll(test)
```

`test(testInstance)` が `true` を返す全ての子テストインスタンスを検索します。

## testInstance.findAllByType() {#testinstancefindallbytype}

```
testInstance.findAllByType(type)
```

与えられた `type` を持つ全ての子テストインスタンスを検索します。

## testInstance.findAllByProps() {#testinstancefindallbyprops}

```
testInstance.findAllByProps(props)
```

与えられた `props` を持つ全ての子テストインスタンスを検索します。

### `testInstance.instance {#testinstanceinstance}`

`testInstance.instance`

当該テストインスタンスに対応するコンポーネントのインスタンスです。関数コンポーネントはインスタンスを持たないため、クラスコンポーネントでのみ使用することができます。与えられたコンポーネント内での `this` の値と一致します。

### `testInstance.type {#testinstancetype}`

`testInstance.type`

当該テストインスタンスに対応するコンポーネントの型です。例えば、`<Button />` コンポーネントは `Button` 型を持っています。

### `testInstance.props {#testinstanceprops}`

`testInstance.props`

当該テストインスタンスに対応するコンポーネントの `props` です。例えば、`<Button size="small" />` コンポーネントは `{size: 'small'}` を `props` として持っています。

### `testInstance.parent {#testinstanceparent}`

`testInstance.parent`

当該テストインスタンスの親テストインスタンスです。

### `testInstance.children {#testinstancechildren}`

`testInstance.children`

当該テストインスタンスの子テストインスタンスです。

## 使い方の一例

オプションとして `createNodeMock` 関数を `TestRenderer.create` に渡すことで、独自のモック `refs` を作成することができます。`createNodeMock` は現在の要素を受け取り、モックの `ref` オブジェクトを返す必要があります。`refs` に依存したコンポーネントのテストに便利です。

```
import TestRenderer from 'react-test-renderer';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.input = null;
  }
  componentDidMount() {
    this.input.focus();
  }
  render() {
```

```
        return <input type="text" ref={el => this.input = el} />
    }

let focused = false;
TestRenderer.create(
  <MyComponent />,
{
  createNodeMock: (element) => {
    if (element.type === 'input') {
      // mock a focus function
      return {
        focus: () => {
          focused = true;
        }
      };
    }
    return null;
  }
);
expect(focused).toBe(true);
```

# Ref と DOM

Ref は render メソッドで作成された DOM ノードもしくは React の要素にアクセスする方法を提供します。

一般的な React のデータフローでは、`props` が、親コンポーネントがその子要素とやりとりする唯一の方法です。子要素を変更するには、新しい `props` でそれを再レンダーします。ただし、この一般的なデータフロー以外で、子要素を命令型のコードを使って変更する必要がある場合もあります。変更したい子要素が React コンポーネントのインスタンスのことや、DOM 要素のこともあるでしょう。どちらの場合でも、React は避難ハッチを提供します。

## いつ Ref を使うか

Ref に適した使用例は以下の通りです。

- フォーカス、テキストの選択およびメディアの再生の管理
- アニメーションの発火
- サードパーティの DOM ライブラリとの統合

宣言的に行えるものには `ref` を使用しないでください。

例えば、`Dialog` コンポーネントに `open()` と `close()` メソッドを実装するかわりに、`isOpen` プロパティを渡してください。

## Ref を使いすぎない

最初はアプリ内で「何かを起こす」ために `ref` を使いがちかもしれません。そんなときは、少し時間をかけて、コンポーネントの階層のどこで状態を保持すべきかについて、よりしっかりと見てみてください。多くの場合、その状態を「保持する」ための適切な場所は階層のより上位にあることが明らかになるでしょう。具体例については [state のリフトアップガイド](#) を参照してください。

### 補足

以下の例は React 16.3 で導入された `React.createRef()` API を使うように更新されました。以前のリリースの React を使用している場合は、代わりに `callback ref` を使用することをお勧めします。

## Ref を作成する

Ref は `React.createRef()` を使用して作成され、`ref` 属性を用いて React 要素に紐付けられます。Ref は通常、コンポーネントの構築時にインスタンスプロパティに割り当てられるため、コンポーネントを通して参照が可能です。

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.myRef = React.createRef();
  }
  render() {
```

```

        return <div ref={this.myRef} />;
    }
}
```

## Ref へのアクセス

`ref` が `render` メソッドの要素に渡されると、そのノードへの参照は `ref` の `current` 属性でアクセスできるようになります。

```
const node = this.myRef.current;
```

`ref` の値はノードの種類によって異なります。

- HTML 要素に対して `ref` 属性が使用されている場合、`React.createRef()` を使ってコンストラクタ内で作成された `ref` は、その `current` プロパティとして根底にある DOM 要素を受け取ります
- `ref` 属性がカスタムクラスコンポーネントで使用されるとき、`ref` オブジェクトはコンポーネントのマウントされたインスタンスを `current` として受け取ります
- 関数コンポーネント (function components)** には `ref` 属性を使用してはいけません。なぜなら、関数コンポーネントはインスタンスを持たないからです

以下の例ではその違いを示しています。

### DOM 要素への Ref の追加

このコードでは DOM ノードへの参照を保持するために `ref` を使います。

```

class CustomTextInput extends React.Component {
  constructor(props) {
    super(props);
    // textInput DOM 要素を保持するための ref を作成します。
    this.textInput = React.createRef();
    this.focusTextInput = this.focusTextInput.bind(this);
  }

  focusTextInput() {
    // 生の DOM API を使用して明示的にテキストの入力にフォーカスします。
    // 補足 : DOM ノードを取得するために "current" にアクセスしています。
    this.textInput.current.focus();
  }

  render() {
    // コンストラクタで作成した `textInput` に <input> ref を関連付けることを
    // React に伝えます。
    return (
      <div>
        <input
          type="text"
          ref={this.textInput} />
        <input
          type="button"
          value="Focus the text input"
          onClick={this.focusTextInput}
        />
      </div>
    );
  }
}
```

コンポーネントがマウントされると React は `current` プロパティに DOM 要素を割り当て、マウントが解除されると `null` に戻します。`ref` の更新は `componentDidMount` または `componentDidUpdate` ライフサイクルメソッドの前に行われます。

### クラスコンポーネントへの Ref の追加

マウント直後にクリックされることをシミュレーションするために上記の `CustomTextInput` をラップしたい場合は、`ref` を使用してカスタムインプットにアクセスし、その `focusTextInput` メソッドを手動で呼び出せます。

```
class AutoFocusTextInput extends React.Component {
  constructor(props) {
    super(props);
    this.textInput = React.createRef();
  }

  componentDidMount() {
    this.textInput.current.focusTextInput();
  }

  render() {
    return (
      <CustomTextInput ref={this.textInput} />
    );
  }
}
```

これは `CustomTextInput` がクラスとして宣言されている場合にのみ機能することに注意してください。

```
class CustomTextInput extends React.Component {
  // ...
}
```

### Ref と関数コンポーネント

関数コンポーネントにはインスタンスがないため、デフォルトでは関数コンポーネントに `ref` 属性を使用することはできません。

```
function MyFunctionComponent() {
  return <input />;
}

class Parent extends React.Component {
  constructor(props) {
    super(props);
    this.textInput = React.createRef();
  }

  render() {
    // これは動き「ません」！
    return (
      <MyFunctionComponent ref={this.textInput} />
    );
  }
}
```

関数コンポーネントに対して `ref` が使用できるようにしたい場合は、`forwardRef` を（必要に応じて `useImperativeHandle` と組み合わせて）利用するか、コンポーネントをクラスに書き換えます。

ただし、DOM 要素またはクラスコンポーネントを参照している限り、**関数コンポーネント内で ref 属性を使用することはできます。**

```
function CustomTextInput(props) {
  // ref が参照できるように、textInput をここで宣言する必要があります。
  const textInput = useRef(null);

  function handleClick() {
    textInput.current.focus();
  }

  return (
    <div>
      <input
        type="text"
        ref={textInput} />
      <input
        type="button"
        value="Focus the text input"
        onClick={handleClick}
      />
    </div>
  );
}
```

## DOM の Ref を親コンポーネントに公開する

まれに、親コンポーネントから子コンポーネントの DOM ノードにアクセスしたい場合があります。これは、コンポーネントのカプセル化を壊すため、一般的にはおすすめできませんが、フォーカスを発火させたり、子の DOM ノードのサイズや位置を計測するのに役立つことがあります。

[子コンポーネントに ref を追加することはできますが、DOM ノードではなくコンポーネントインスタンスしか取得できないため、これは理想的な解決策ではありません。また、これは関数コンポーネントでは機能しません。](#)

React 16.3 以降を使用している場合、これらの場合には [ref のフォワーディング](#) を使用することをお勧めします。**Ref のフォワーディングを使うと、コンポーネントは任意の子コンポーネントの ref を自分自身の ref として公開できるようになります。** [ref のフォワーディングのドキュメント](#) に、子の DOM ノードを親コンポーネントに公開する方法の詳細な例があります。

React 16.2 以下を使用している場合、または ref フォワーディングで提供される以上の柔軟性が必要な場合は、[この代替手法](#)を使用して ref を異なる名前の props として明示的に渡すことができます。

可能であれば DOM ノードを公開しないことをおすすめしますが、これは便利な避難ハッチになることもあります。留意すべき点として、この方法では子コンポーネントにコードを追加する必要があります。子コンポーネントの実装にまったく手を加えられない場合、最後の選択肢は `findDOMNode()` を使用することですが、おすすめできない上に、`StrictMode` では非推奨です。

## コールバック Ref

React はまた「コールバック Ref」と呼ばれる、より細かい制御が可能な ref を設定するための別の方法をサポートします。

`createRef()` によって作成された `ref` 属性を渡す代わりに、関数を渡します。この関数は、引数として React コンポーネントのインスタンスまたは HTML DOM 要素を受け取ります。これを保持することで、他の場所からアクセスできます。

以下は、`ref` コールバックを用いて DOM ノードへの参照をインスタンスプロパティに格納する一般的な実装例です。

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

    this.textInput = null;

    this.setTextInputRef = element => {
      this.textInput = element;
    };

    this.focusTextInput = () => {
      // 生の DOM API を使用して明示的にテキストの入力にフォーカスします。
      if (this.textInput) this.textInput.focus();
    };
  }

  componentDidMount() {
    // マウント時に入力をオートフォーカスします。
    this.focusTextInput();
  }

  render() {
    // インスタンスフィールド（例えば this.textInput）にテキスト入力の DOM 要素への
    // 参照を保存するために `ref` コールバックを使用してください。
    return (
      <div>
        <input
          type="text"
          ref={this.setTextInputRef}
        />
        <input
          type="button"
          value="Focus the text input"
          onClick={this.focusTextInput}
        />
      </div>
    );
  }
}
```

コンポーネントがマウントされると React は DOM 要素とともに `ref` コールバックを呼び出し、マウントが解除されると `null` とともにコールバックを呼び出します。Ref は `componentDidMount` または `componentDidUpdate` が発火する前に最新のものであることが保証されています。

`React.createRef()` で作成されたオブジェクトの `ref` と同様に、コンポーネント間でコールバック `ref` を渡すことができます。

```
function CustomTextInput(props) {
  return (
    <div>
      <input ref={props.inputRef} />
    </div>
  );
}
```

```

class Parent extends React.Component {
  render() {
    return (
      <CustomTextInput
        inputRef={el => this.inputElement = el}
      />
    );
  }
}

```

上記の例では、`Parent` は `ref` コールバックを `inputRef` プロパティとして `CustomTextInput` に渡し、`CustomTextInput` は同じ関数を特別な `ref` 属性として `<input>` に渡します。その結果、`Parent` の `this.inputElement` は、`CustomTextInput` の `<input>` 要素に対応する DOM ノードに設定されます。

## レガシー API: 文字列形式の Ref

以前に React を使用したことがある場合は、`ref` 属性が `"textInput"` のような文字列になっており DOM ノードが `this.refs.textInput` としてアクセスされる古い API に慣れているかもしれません。文字列形式の `ref` には [いくつかの問題](#)があり、レガシーと見なされ、[将来のリリースのいずれかで削除される可能性が高い](#)ため、使用することをおすすめしません。

### 補足

`Ref` にアクセスするために `this.refs.textInput` を現在使用している場合は、代わりに [コールバックパターン](#) もしくは [createRef API](#) を使用することをおすすめします。

## コールバック Ref の注意事項

`ref` コールバックがインライン関数として定義されている場合、更新中に 2 回呼び出されます。最初は `null`、次に DOM 要素で呼び出されます。これは、それぞれのレンダーで関数の新しいインスタンスが作成されるため、React は古い `ref` を削除し、新しい `ref` を設定する必要があるためです。`ref` コールバックをクラス内のバインドされたメソッドとして定義することでこれを回避できますが、ほとんどの場合は問題にならないはずです。

---

[Go to TOC](#)

# リリースチャンネル

React はバグを報告し、プルリクエストをオープンし、[RFC を投稿](#)してくださっている活発なオープンソースコミュニティを頼りにしています。フィードバックを促すため、私たちはリリースされていない機能を含んだ React の特別なビルドを共有することができます。

このドキュメントは、フレームワーク、ライブラリ、開発者向けツールに取り組んでいる開発者に最も関連します。主にユーザ向けアプリケーションを構築するために React を使用している開発者は、プレリリースチャンネルについて心配する必要はありません。

React の各リリースチャンネルは、明確なユースケースのために設計されています。

- **Latest** は安定版であり、セマンティック・バージョニングに基づいた React のリリースです。それは npm から React をインストールすると得られるものです。これは、既にあなたが使用しているチャンネルです。**全てのユーザ向け React アプリケーションのためにこれを使用します。**
- **Next** は React ソースコードのリポジトリの main ブランチを追跡します。これを、次のマイナーバージョンのリリース候補と考えてください。React とサードパーティプロジェクト間のインテグレーションテストにこれを使用します。
- **Experimental** には、安定版のリリースでは使用できない実験的な API や機能が含まれます。これも main ブランチを追跡しますが、追加のフィーチャー・フラグが有効になっています。近々予定されているリリース前の機能を試すためにこれを使用してください。

全てのリリースが npm に公開されていますが、Latest のみが [セマンティック・バージョニング](#) を使用します。プレリリース（Next チャンネルと Experimental チャンネル）は、Next の場合は `0.0.0-68053d940-20210623`、Experimental の場合は `0.0.0-experimental-68053d940-20210623` のように、コンテンツのハッシュとコミット日時から生成されたバージョンを持ちます。

**Latest** が、ユーザ向けアプリケーションのために公式にサポートされた唯一のリリースチャンネルです。Next と Experimental はテスト目的のためにのみ提供されており、また、リリース間で動作が変わらないことも保証されません。これらは、Latest リリースで使用しているセマンティック・バージョニングに準拠していません。

プレリリースを安定版のリリースと同じレジストリに公開することで、[unpkg](#) や [CodeSandbox](#) のような、npm のワークフローをサポートする多くのツールを活用できます。

## Latest チャンネル

Latest チャンネルは、安定版の React をリリースするために使用するチャンネルです。npm の `latest` タグに対応します。実際のユーザに提供される全ての React アプリケーションに対して、このチャンネルが推奨されます。

**どのチャンネルを使用すべきか分からない場合、Latest チャンネルを使用してください。** あなたが React 開発者なら、Latest チャンネルを既に利用しています。

Latest のアップデートは非常に安定していると期待できます。バージョンは、セマンティック・バージョニングのスキームに従います。私たちの安定性と段階的移行への取り組みについては、[バージョニングポリシー](#)をご覧ください。

## Next チャンネル

Next チャンネルは、React リポジトリの main ブランチを追跡する、プレリリースチャンネルです。私たちは Next チャンネルのプレリリースを、Latest チャンネルのリリース候補として使用します。Next を、より頻繁に更新が行われる Latest のスーパーセットだと考えることができます。

最新の Next リリースと Latest リリース間の差分の程度は、2 つのマイナーリリース間とほぼ同じです。しかし **Next チャンネルは、セマンティック・バージョニングに準拠していません。** 連続したリリース間でしばしば破壊的変更が発生するということを、Next チャンネルでは予期しなければなりません。

**プレリリースをユーザ向けアプリケーションに使用しないでください。**

Next リリースは、npm に `next` タグ付きで公開されています。`0.0.0-68053d940-20210623` のように、ビルトされたコンテンツのハッシュおよびコミット日時から、バージョンが生成されます。

### インテグレーションテストに Next チャンネルを使用する

Next チャンネルは、React と他のプロジェクト間のインテグレーションテストをサポートするように設計されています。

React に対する全ての変更は、一般向けリリース前に広範囲の内部テストを受けます。しかし、React のエコシステム全体では無数の環境と構成が存在しており、その全てに対してテストを行うことは不可能です。

あなたが React サードパーティのフレームワーク、ライブラリ、開発者向けツール、あるいは同様のインフラストラクチャ寄りのプロジェクトの作者である場合、最新の React の変更に対して定期的にテストスイートを実行することで、あなたのユーザや React コミュニティ全体のために React の安定化を助けることができます。興味がある場合、次の手順に従ってください。

- お好みの継続的インテグレーションプラットフォームを利用して、クーロンジョブをセットアップします。クーロンジョブは、[CircleCI](#) と [Travis CI](#) の両方でサポートされています。
- クーロンジョブで npm の `next` タグを使い、React パッケージを Next チャンネルの最新のリリースにアップデートします。npm cli を使って：

```
npm update react@next react-dom@next
```

もしくは yarn で：

```
yarn upgrade react@next react-dom@next
```

- アップデートされたパッケージに対してテストスイートを実行します。
- 全てのテストがパスしたのであれば、素晴らしいです！ あなたのプロジェクトは次の React のマイナーリリースで動作することが期待できます。
- 意図せず何かが壊れた場合は、[Issue を提出して](#)お知らせください。

このワークフローを使用しているプロジェクトとして、Next.js があります（シャレではありません！ 真面目です！）。例として彼らの [CircleCI の設定](#)を参照できます。

## Experimental チャンネル

Next チャンネルと同じく、Experimental チャンネルは React リポジトリの main ブランチを追跡するプレリリースチャンネルです。しかし Next とは異なり、Experimental リリースには、より広範にリリースするにはまだ準備ができない追加の機能と API が含まれます。

通常、Next への更新には、対応した Experimental への更新が伴います。これらは同じリビジョンのソースをベースにしていますが、異なるフィーチャー・フラグの設定を使用してビルドしています。

Experimental リリースは、Next や Latest のリリースとは大きく異なる場合があります。**Experimental リリースを、ユーザ向けアプリケーションに使用しないでください。** リリース間で頻繁に破壊的変更が発生しうるということを、Experimental チャンネルでは予期しなければなりません。

Experimental リリースは、npm に `experimental` タグ付きで公開されています。`0.0.0-experimental-68053d940-20210623` のように、ビルドされたコンテンツのハッシュおよびコミット日時から、バージョンが生成されます。

### Experimental リリースには何が含まれていますか？

実験的機能は、一般向けに広く公開するための準備ができていないものであり、最終決定までに大幅に変化する可能性があります。いくつかの試みは、最終的に採用されない可能性があります。このような実験を行うのは、提案された変更内容の実行可能性をテストするためです。

例えば、フックを発表したときに Experimental が存在していた場合、Latest で利用可能になる数週間前に Experimental チャンネルでフックをリリースしていくでしょう。

Experimental に対してインテグレーションテストを実行することには、価値があるかもしれません。これはあなた次第です。しかし、Experimental は Next よりもさらに不安定であるとアドバイスしておきます。**Experimental のリリース間では安定性は何も保証されません。**

### 実験的機能の詳細をどのように知ればよいですか？

実験的機能については、ドキュメント化されることもありますが、されないこともあります。通常、Next もしくは Latest のリリースが近づくまで、ドキュメント化されません。

ドキュメント化されていない機能については、RFC が付随することがあります。

新しい試みを発表する準備が整ったときに [React blog](#) にそのことを投稿しますが、全ての試みについて公開するとは限りません。

GitHub で公開されている React のリポジトリの [history](#) で、変更の包括的なリストをいつでも参照できます。

# レンダープロップ

"レンダープロップ (render prop)" という用語は、値が関数である `props` を使って、コンポーネント間でコードを共有するためのテクニックを指します。

レンダープロップを持つコンポーネントは、自身のレンダーロジックを実装する代わりに、React 要素を返す関数を受け取ってそれを呼び出します。

```
<DataProvider render={data =>
  <h1>Hello {data.target}</h1>
}>
```

レンダープロップを用いたライブラリとしては、[React Router](#) や [Downshift](#)、[Formik](#) などがあります。

このドキュメントでは、レンダープロップが役立つ理由と、その実装手順について解説します。

## 横断的関心事にレンダープロップを使う

コンポーネントは、React でコードを再利用するための主な要素ですが、あるコンポーネントがカプセル化した `state` や振る舞いを、同じ `state` を必要とする別のコンポーネントと共有する方法については、あまり自明ではありません。

たとえば、以下のコンポーネントは、ウェブアプリケーション内でのマウスの位置を追跡します。

```
class MouseTracker extends React.Component {
  constructor(props) {
    super(props);
    this.handleMouseMove = this.handleMouseMove.bind(this);
    this.state = { x: 0, y: 0 };
  }

  handleMouseMove(event) {
    this.setState({
      x: event.clientX,
      y: event.clientY
    });
  }

  render() {
    return (
      <div style={{ height: '100vh' }} onMouseMove={this.handleMouseMove}>
        <h1>Move the mouse around!</h1>
        <p>The current mouse position is ({this.state.x}, {this.state.y})</p>
      </div>
    );
  }
}
```

画面上でカーソルが移動すると、コンポーネントはその `(x, y)` 座標を `<p>` 内に表示します。

ここで疑問となるのは、この振る舞いを他のコンポーネントで再利用する方法です。つまり、他のコンポーネントもカーソルの位置を知る必要がある時、この振る舞いだけをカプセル化し、そのコンポーネントと簡単に共有することは可能でしょうか？

コンポーネントは React でコードを再利用するための基本要素ですので、コードを少しリファクタリングし、他の場所で再利用したい振る舞いをカプセル化した `<Mouse>` というコンポーネントを作つて、それを使うようにしてみましょう。

```
// The <Mouse> component encapsulates the behavior we need...
class Mouse extends React.Component {
  constructor(props) {
    super(props);
    this.handleMouseMove = this.handleMouseMove.bind(this);
    this.state = { x: 0, y: 0 };
  }

  handleMouseMove(event) {
    this.setState({
      x: event.clientX,
      y: event.clientY
    });
  }

  render() {
    return (
      <div style={{ height: '100vh' }} onMouseMove={this.handleMouseMove}>
        {/* ...but how do we render something other than a <p>? */}
        <p>The current mouse position is ({this.state.x}, {this.state.y})</p>
      </div>
    );
  }
}

class MouseTracker extends React.Component {
  render() {
    return (
      <>
        <h1>Move the mouse around!</h1>
        <Mouse />
      </>
    );
  }
}
```

これで `<Mouse>` コンポーネントは、`mousemove` イベントとカーソルの (x, y) 座標に紐付けられた全ての振る舞いをカプセル化していますが、まだ再利用可能と言うには不十分です。

たとえば、画面の中でマウスを追いかける猫の画像をレンダーする `<Cat>` コンポーネントがあるとしましょう。`<Cat mouse={{ x, y }}>` props を使って、このコンポーネントにマウスの座標を受け渡し、画面上のどこに猫の画像を配置すれば良いかを知らせたいでしょう。

手始めに、`<Mouse>` の `render` メソッド内で、以下のように `<Cat>` をレンダーしてみましょう。

```
class Cat extends React.Component {
  render() {
    const mouse = this.props.mouse;
    return (
      
    );
  }
}
```

```

class MouseWithCat extends React.Component {
  constructor(props) {
    super(props);
    this.handleMouseMove = this.handleMouseMove.bind(this);
    this.state = { x: 0, y: 0 };
  }

  handleMouseMove(event) {
    this.setState({
      x: event.clientX,
      y: event.clientY
    });
  }

  render() {
    return (
      <div style={{ height: '100vh' }} onMouseMove={this.handleMouseMove}>
        /*
          We could just swap out the <p> for a <Cat> here ... but then
          we would need to create a separate <MouseWithSomethingElse>
          component every time we need to use it, so <MouseWithCat>
          isn't really reusable yet.
        */
        <Cat mouse={this.state} />
      </div>
    );
  }
}

class MouseTracker extends React.Component {
  render() {
    return (
      <div>
        <h1>Move the mouse around!</h1>
        <MouseWithCat />
      </div>
    );
  }
}

```

これだけが目的であれば正しく動作しますが、再利用可能な方法でこの振る舞いをカプセル化するという目的はまだ果たせていません。他の異なるユースケースでもマウスの位置を知りたい場合、毎回そのユースケースに沿ったものをレンダーする新しいコンポーネント（つまり、本質的に別の `<MouseWithCat>`）を作成する必要があります。

ここでレンダープロップの出番となります。`<Mouse>` コンポーネント内でハードコードされた `<Cat>` でレンダーの出力を変更する代わりに、`<Mouse>` コンポーネントに関数型の `props` を渡し、コンポーネントはその関数を使って何をレンダすべきか動的に知るのです。これがレンダープロップの役割です。

```

class Cat extends React.Component {
  render() {
    const mouse = this.props.mouse;
    return (
      
    );
  }
}

```

```

class Mouse extends React.Component {
  constructor(props) {
    super(props);
    this.handleMouseMove = this.handleMouseMove.bind(this);
    this.state = { x: 0, y: 0 };
  }

  handleMouseMove(event) {
    this.setState({
      x: event.clientX,
      y: event.clientY
    });
  }

  render() {
    return (
      <div style={{ height: '100vh' }} onMouseMove={this.handleMouseMove}>
        /*
          Instead of providing a static representation of what <Mouse> renders,
          use the `render` prop to dynamically determine what to render.
        */
        {this.props.render(this.state)}
      </div>
    );
  }
}

class MouseTracker extends React.Component {
  render() {
    return (
      <div>
        <h1>Move the mouse around!</h1>
        <Mouse render={mouse => (
          <Cat mouse={mouse} />
        )}/>
      </div>
    );
  }
}

```

ここでは、特定のユースケースを解決するために `<Mouse>` コンポーネントを複製して `render` メソッドに他のものをハードコードする代わりに、`<Mouse>` に `render` プロパティを渡して、何をレンダーするか動的に決定できるようにしています。

より具体的に述べると、**レンダープロップとは、あるコンポーネントが何をレンダーすべきかを知るために使う関数型の props です。**

このテクニックによって、再利用する必要がある振る舞いの移植性が極めて高くなります。その振る舞いをさせるためには、現在のカーソルの  $(x, y)$  座標にレンダーするものを示す `render` プロパティを使って `<Mouse>` をレンダーすれば良いのです。

レンダープロップの興味深い点として、多くの**高階コンポーネント** (HOC) がレンダープロップを使った通常のコンポーネントによって実装可能ということが挙げられます。たとえば、`<Mouse>` コンポーネントよりも `withMouse` HOC が好みであれば、レンダープロップを持つ `<Mouse>` を使って簡単に作成可能です。

```
// If you really want a HOC for some reason, you can easily
// create one using a regular component with a render prop!
function withMouse(Component) {
  return class extends React.Component {
    render() {
      return (
        <Mouse render={mouse => (
          <Component {...this.props} mouse={mouse} />
        )}>
      );
    }
  }
}
```

つまり、レンダープロップによってどちらのパターンも可能になります。

## render 以外の props を使う

このパターンが「レンダープロップ」という名前だからといって、必ずしも `render` という名前の `props` を使う必要はないということを念頭に置いてください。実際、コンポーネントがレンダーするものを知るために使う関数の `props` は、その名前が何であれ、技術的には「レンダープロップ」と呼ぶことができます。

上記の例では `render` を用いていますが、`children` プロパティを使っても同じくらい簡単です！

```
<Mouse children={mouse => (
  <p>The mouse position is {mouse.x}, {mouse.y}</p>
)}>
```

さらに、`children` プロパティは実際には JSX 要素の「属性」の一覧内で名前を付ける必要がないことも忘れないでください。代わりに、要素内部に直接設定可能です！

```
<Mouse>
  {mouse => (
    <p>The mouse position is {mouse.x}, {mouse.y}</p>
  )}
</Mouse>
```

このテクニックは、`react-motion` の API などで使用されています。

このテクニックは若干珍しいため、このような API 設計時には、`children` が関数であることを `propTypes` で明示した方が良いでしょう。

```
Mouse.propTypes = {
  children: PropTypes.func.isRequired
};
```

## 注意事項

### レンダープロップを React.PureComponent で使うときの注意点

レンダープロップを使う際、`render` メソッド内で関数を作成していると、`React.PureComponent` を使う利点が相殺されます。これは `props` の浅い (shallow) 比較は新しい `props` の値に対して常に `false` を返し、そして `render` は毎回レンダープロップとして新しい値を生成するためです。

たとえば、上記の `<Mouse>` コンポーネントの場合、`Mouse` が `React.Component` ではなく `React.PureComponent` を継承していたとすると、次のようにになります。

```
class Mouse extends React.PureComponent {
  // Same implementation as above...
}

class MouseTracker extends React.Component {
  render() {
    return (
      <div>
        <h1>Move the mouse around!</h1>

        {/* 
          This is bad! The value of the `render` prop will
          be different on each render.
        */}
        <Mouse render={mouse => (
          <Cat mouse={mouse} />
        )}>
      </div>
    );
  }
}
```

この例では、`<MouseTracker>` がレンダーされるたびに `<Mouse render>` プロパティの値として新しい関数が生成されるので、`<Mouse>` が `React.PureComponent` を継承している効果がそもそもなくなってしまいます！

この問題を回避するため、レンダープロップをインスタンスマソッドとして次のように定義することもできます。

```
class MouseTracker extends React.Component {
  // Defined as an instance method, `this.renderTheCat` always
  // refers to *same* function when we use it in render
  renderTheCat(mouse) {
    return <Cat mouse={mouse} />;
  }

  render() {
    return (
      <div>
        <h1>Move the mouse around!</h1>
        <Mouse render={this.renderTheCat} />
      </div>
    );
  }
}
```

`props` を静的に定義できない場合（たとえば、コンポーネントの `props` や `state` をクロージャで囲む場合など）、`<Mouse>` は代わりに `React.Component` を継承すべきです。

# 要素のレンダー

要素とは React アプリケーションの最小単位の構成ブロックです。

要素は画面上に表示したいものの説明書きです：

```
const element = <h1>Hello, world</h1>;
```

ブラウザの DOM 要素とは異なり、React 要素はプレーンなオブジェクトであり、安価に作成できます。React DOM が React 要素に合致するように DOM を更新する作業を担当します。

## 補足:

要素のことを、より広く知られている概念である "コンポーネント" と混同する人もいるかもしれません。コンポーネントについては[次の章](#)で説明します。要素とはコンポーネントを "構成する" ものです。次に進む前にこの章を読んでいくことをお勧めします。

## 要素を DOM として描画する

HTML ファイルの中に `<div>` 要素があったとしましょう：

```
<div id="root"></div>
```

この中にあるもの全てが React DOM によって管理されることになるので、"ルート" DOM ノードと呼ぶことにしましょう。

React だけで構築されたアプリケーションは、通常ルート DOM ノードをひとつだけ持ります。既存のアプリに React を統合しようとしている場合は、独立したルート DOM ノードを好きなだけ持つことができます。

React 要素をルート DOM ノードにレンダーするには、まず `ReactDOM.createRoot()` に DOM 要素を渡し、`root.render()` に React 要素を渡します：

`embed:rendering-elements/render-an-element.js`

### Try it on CodePen

このコードはページに "Hello, world" を表示します。

## レンダーされた要素の更新

React 要素は*イミュータブル*です。一度要素を作成すると、その子要素もしくは属性を変更することはできません。要素は映画の中のひとつのフレームのようなものであり、それは特定のある時点の UI を表します。

今のところ我々が学んだ UI を更新する唯一の方法は、新しい要素を作成して `root.render()` に渡すことだけです。

以下の秒刻みで動く時計の例について考えます：

```
embed:rendering-elements/update-rendered-element.js
```

## Try it on CodePen

上記のコードでは `setInterval()` のコールバックから `root.render()` を毎秒呼び出しています。

### 補足:

実際には大抵の React アプリケーションは `root.render()` を一度しか呼び出しません。次の章では上記のようなコードをどのように `state` 付きコンポーネントへとカプセル化するのかを学びます。

トピックはお互いを基礎として構成されているため、読み飛ばさないことをお勧めします。

## React は必要な箇所のみを更新する

React DOM は要素とその子要素を以前のものと比較し、DOM を望ましい状態へと変えるのに必要なだけの DOM の更新を行います。

このことは、[最後の例](#)をブラウザツールで調査すれば確認できます：



毎秒ごとに UI ツリー全体を表す要素を作成しているにも関わらず、内容が変更されたテキストノードのみが React DOM により更新されます。

私達の経験上、時間の経過によりどのように UI が変更されるかを考えるよりも、任意の時点において UI がどのように見えるべきかを考えることで、あらゆる類のバグを排除することができます。

---

[Go to TOC](#)

# state とライフサイクル

このページでは React コンポーネントにおける state とライフサイクルについての導入を行います。[詳細なコンポーネントの API リファレンスはこちら](#)にあります。

以前の章のひとつにあった秒刻みの時計の例を考えてみましょう。要素のレンダーの章にて、UI を更新するための方法をひとつだけ学びました。それはレンダーされた出力を更新するために `root.render()` を呼び出す、というものでした。

```
const root = ReactDOM.createRoot(document.getElementById('root'));

function tick() {
  const element = (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {new Date().toLocaleTimeString()}.</h2>
    </div>
  );
  root.render(element);
}

setInterval(tick, 1000);
```

[Try it on CodePen](#)

このセクションでは、この `Clock` コンポーネントを真に再利用可能かつカプセル化されたものにする方法を学びます。コンポーネントが自身でタイマーをセットアップし、自身を毎秒更新するようにします。

時計の見た目をカプセル化するところから始めてみましょう：

```
const root = ReactDOM.createRoot(document.getElementById('root'));

function Clock(props) {
  return (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {props.date.toLocaleTimeString()}.</h2>
    </div>
  );
}

function tick() {
  root.render(<Clock date={new Date()} />);
}

setInterval(tick, 1000);
```

[Try it on CodePen](#)

しかし上記のコードは重要な要件を満たしていません：`Clock` がタイマーを設定して UI を毎秒ごとに更新するという処理は、`Clock` の内部実装の詳細 (implementation detail) であるべきだということです。

理想的には以下のコードを一度だけ記述して、`Clock` に自身を更新させたいのです：

```
root.render(<Clock />);
```

これを実装するには、`Clock` コンポーネントに "ステート (state)" を追加する必要があります。

`state` は `props` に似ていますが、コンポーネントによって完全に管理されるプライベートなものです。

## 関数をクラスに変換する

以下の 5 ステップで、`Clock` のような関数コンポーネントをクラスに変換することができます。

1. `React.Component` を継承する同名の `ES6 クラス`を作成する。
2. `render()` と呼ばれる空のメソッドを 1 つ追加する。
3. 関数の中身を `render()` メソッドに移動する。
4. `render()` 内の `props` を `this.props` に書き換える。
5. 空になった関数の宣言部分を削除する。

```
class Clock extends React.Component {
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.props.date.toLocaleTimeString()}</h2>
      </div>
    );
  }
}
```

### Try it on CodePen

これで、`Clock` は関数ではなくクラスとして定義されるようになりました。

`render` メソッドは更新が発生した際に毎回呼ばれます、同一の DOM ノード内で `<Clock />` をレンダーしている限り、`Clock` クラスのインスタンスは 1 つだけ使われます。このことにより、ローカル `state` やライフサイクルメソッドといった追加の機能が利用できるようになります。

## クラスにローカルな `state` を追加する

以下の 3 ステップで `date` を `props` から `state` に移します：

1. `render()` メソッド内の `this.props.date` を `this.state.date` に書き換える：

```
class Clock extends React.Component {
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}</h2>
      </div>
    );
  }
}
```

2. `this.state` の初期状態を設定するクラスコンストラクタを追加する：

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

親クラスのコンストラクタへの `props` の渡し方に注目してください：

```
constructor(props) {
  super(props);
  this.state = {date: new Date()};
}
```

クラスのコンポーネントは常に `props` を引数として親クラスのコンストラクタを呼び出す必要があります。

3. `<Clock />` 要素から `date` プロパティを削除する：

```
root.render(<Clock />);
```

タイマーのコードはコンポーネント自身に後で追加しなおします。

結果は以下のようになります：

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Clock />);
```

[Try it on CodePen](#)

次に、`Clock` が自分でタイマーを設定し、毎秒ごとに自分を更新するようにします。

## クラスにライフサイクルメソッドを追加する

多くのコンポーネントを有するアプリケーションでは、コンポーネントが破棄された場合にそのコンポーネントが占有していたリソースを解放することがとても重要です。

[タイマーを設定](#)したいのは、最初に `Clock` が DOM として描画されるときです。このことを React では "マウント (mounting)" と呼びます。

また [タイマーをクリア](#)したいのは、`Clock` が生成した DOM が削除されるときです。このことを React では "アンマウント (unmounting)" と呼びます。

コンポーネントクラスで特別なメソッドを宣言することで、コンポーネントがマウントしたりアンマウントしたりした際にコードを実行することができます：

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  componentDidMount() {}

  componentWillUnmount() {}

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}</h2>
      </div>
    );
  }
}
```

これらのメソッドは "ライフサイクルメソッド (lifecycle method)" と呼ばれます。

`componentDidMount()` メソッドは、出力が DOM にレンダーされた後に実行されます。ここがタイマーをセットアップするのよい場所です：

```
componentDidMount() {
  this.timerID = setInterval(
    () => this.tick(),
    1000
  );
}
```

タイマー ID を直接 `this` 上に (`this.timerID` として) 格納したことに注目してください。

`this.props` は React 自体によって設定され、また `this.state` は特別な意味を持っていますが、何かデータフローに影響しないデータ（タイマー ID のようなもの）を保存したい場合に、追加のフィールドを手動でクラスに追加することは自由です。

タイマーの後片付けは `componentWillUnmount()` というライフサイクルメソッドで行います：

```
componentWillUnmount() {
  clearInterval(this.timerID);
}
```

最後に、`Clock` コンポーネントが毎秒ごとに実行する `tick()` メソッドを実装します。

コンポーネントのローカル `state` の更新をスケジュールするために `this.setState()` を使用します：

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  componentDidMount() {
    this.timerID = setInterval(
      () => this.tick(),
      1000
    );
  }

  componentWillUnmount() {
    clearInterval(this.timerID);
  }

  tick() {
    this.setState({
      date: new Date()
    });
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}</h2>
      </div>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Clock />);
```

### Try it on CodePen

これで、この時計は毎秒ごとに時間を刻みます。

何が起こったのかをメソッドが呼び出される順序にそって簡単に振り返ってみましょう：

- `<Clock />` が `root.render()` に渡されると、React は `Clock` コンポーネントのコンストラクタを呼び出します。`Clock` は現在時刻を表示する必要があるので、現在時刻を含んだオブジェクトで `this.state` を初期化します。あとでこの `state` を更新していきます。
- 次に React は `Clock` コンポーネントの `render()` メソッドを呼び出します。これにより React は画面に何を表示すべきか知ります。その後に、React は DOM を `Clock` のレンダー出力と一致するように更新します。

3. `Clock` の出力が DOM に挿入されると、React は `componentDidMount()` ライフサイクルメソッドを呼び出します。その中で、`Clock` コンポーネントは毎秒ごとにコンポーネントの `tick()` メソッドを呼び出すためにタイマーを設定するようブラウザに要求します。
4. ブラウザは、毎秒ごとに `tick()` メソッドを呼び出します。その中で `Clock` コンポーネントは、現在時刻を含んだオブジェクトを引数として `setState()` を呼び出すことで、UI の更新をスケジュールします。`setState()` が呼び出されたおかげで、React は state が変わったということが分かるので、`render()` メソッドを再度呼び出して、画面上に何を表示すべきかを知ります。今回は、`render()` メソッド内の `this.state.date` が異なっているので、レンダーされる出力には新しく更新された時間が含まれています。それに従って React は DOM を更新します。
5. この後に `Clock` コンポーネントが DOM から削除されがあれば、React は `componentWillUnmount()` ライフサイクルメソッドを呼び出し、これによりタイマーが停止します。

## state を正しく使用する

`setState()` について知っておくべきことが 3 つあります。

### state を直接変更しないこと

例えば、以下のコードではコンポーネントは再レンダーされません：

```
// Wrong
this.state.comment = 'Hello';
```

代わりに `setState()` を使用してください：

```
// Correct
this.setState({comment: 'Hello'});
```

`this.state` に直接代入してよい唯一の場所はコンストラクタです。

### state の更新は非同期に行われる可能性がある

React はパフォーマンスのために、複数の `setState()` 呼び出しを 1 度の更新にまとめて処理することができます。

`this.props` と `this.state` は非同期に更新されるため、次の state を求める際に、それらの値に依存するべきではありません。

例えば、以下のコードはカウンターの更新に失敗することがあります：

```
// Wrong
this.setState({
  counter: this.state.counter + this.props.increment,
});
```

これを修正するために、オブジェクトではなく関数を受け取る `setState()` の 2 つ目の形を使用します。その関数は前の state を最初の引数として受け取り、更新が適用される時点での props を第 2 引数として受け取ります：

```
// Correct
this.setState((state, props) => ({
  counter: state.counter + props.increment
}));
```

上記のコードではアロー関数を使いましたが、通常の関数でも動作します：

```
// Correct
this.setState(function(state, props) {
  return {
    counter: state.counter + props.increment
  };
});
```

## state の更新はマージされる

`setState()` を呼び出した場合、React は与えられたオブジェクトを現在の `state` にマージします。

例えば、あなたの `state` はいくつかの独立した変数を含んでいるかもしれません：

```
constructor(props) {
  super(props);
  this.state = {
    posts: [],
    comments: []
  };
}
```

その場合、別々の `setState()` 呼び出しで、それらの変数を独立して更新することができます：

```
componentDidMount() {
  fetchPosts().then(response => {
    this.setState({
      posts: response.posts
    });
  });

  fetchComments().then(response => {
    this.setState({
      comments: response.comments
    });
  });
}
```

マージは浅く (shallow) 行われる所以、`this.setState({comments})` は `this.state.posts` をそのまま残しますが、`this.state.comments` を完全に置き換えます。

## データは下方向に伝わる

親コンポーネントであれ子コンポーネントであれ、特定の他のコンポーネントがステートフルかステートレスかを知ることはできませんし、特定のコンポーネントの定義が関数型かクラス型かを気にするべきではありません。

これが、`state` はローカルのものである、ないしはカプセル化されている、と言われる理由です。`state` を所有してセットするコンポーネント自身以外からはその `state` にアクセスすることができません。

コンポーネントはその子コンポーネントに `props` として自身の `state` を渡してもかまいません。

```
<FormattedDate date={this.state.date} />
```

`FormattedDate` コンポーネントは `props` 経由で `date` を受け取りますが、それが `Clock` の `state` から来たのか、`Clock` の `props` から来たのか、もしくは手書きされたものなのは分かりません：

```
function FormattedDate(props) {
  return <h2>It is {props.date.toLocaleTimeString()}</h2>;
}
```

### Try it on CodePen

このデータフローは一般的には "トップダウン" もしくは "単一方向" データフローと呼ばれます。いかなる `state` も必ず特定のコンポーネントが所有し、`state` から生ずる全てのデータまたは UI は、ツリーでそれらの "下" にいるコンポーネントにのみ影響します。

コンポーネントツリーとは `props` が流れ落ちる滝なのだと想像すると、各コンポーネントの `state` とは任意の場所で合流してくる追加の水源であり、それらもまた下に流れ落ちていくものなのです。

全てのコンポーネントが本当に独立していることを示すのに、3 つの `<Clock>` をレンダーする `App` コンポーネントを作成します：

```
function App() {
  return (
    <div>
      <Clock />
      <Clock />
      <Clock />
    </div>
  );
}
```

### Try it on CodePen

各 `Clock` は独立してタイマーをセットし、独立して更新されます。

React アプリケーションでは、コンポーネントがステートフルかステートレスかは、コンポーネントにおける内部実装の詳細 (implementation detail) とみなされ、それは時間と共に変化しうるものです。ステートレスなコンポーネントをステートフルなコンポーネントの中で使うことが可能であり、その逆も同様です。

# 静的型チェック

Flow や TypeScript のような静的型チェッカーを使用することでコードを実行する前の早期段階に型不整合等の問題を検知することができます。他にも型解析言語を使用することで補完などの機能が追加され、開発体験を上げることができます。これらの理由から、大きいコードベースでは `PropTypes` の代わりに Flow や TypeScript を使うことをおすすめします。

## Flow

Flow は JavaScript 用の静的型チェック機能です。Facebook で開発されており、React と一緒に使われることが多いです。変数、関数や React コンポーネントに型を定すことができ、型不整合を早期に発見できるようになります。[はじめての Flow](#) に基本情報が記載されているので、それを読むと良いでしょう。

Flow を使用するには以下の手順を踏みます：

- Flow をプロジェクトの dependency に入れる。
- Flow の型定義をコンパイル時にコードから剥がすように設定する。
- 型定義を追加し、Flow を起動して確認する。

順を追って説明します。

### Flow をプロジェクトに追加する

まず、ターミナル上であなたのプロジェクトが含まれているディレクトリに入り、以下のコマンドを入力してください。

Yarn を使っている場合：

```
yarn add --dev flow-bin
```

npm を使っている場合：

```
npm install --save-dev flow-bin
```

このコマンドによって Flow の最新版をあなたのプロジェクトに入れることができます。

次に、`package.json` の `"scripts"` に `flow` を追加しましょう。これで、ターミナルで Flow を実行できるようになります。

```
{
  // ...
  "scripts": {
    "flow": "flow",
    // ...
  },
  // ...
}
```

最後に、以下のコマンドを実行します：

[Yarn](#) を使っている場合：

```
yarn run flow init
```

[npm](#) を使っている場合：

```
npm run flow init
```

このコマンドで Flow の設定ファイルを生成することができます。この設定ファイルはバージョン管理しましょう。

## Flow の型定義をコンパイル時にコードから剥がす

Flow は JavaScript 言語を拡張し、型定義のための特殊な記法を使えるようにします。しかし、ブラウザはその記法について知らない為、ブラウザに送信されるコンパイル後の JavaScript バンドルに含まれないようにしなければいけません。

剥がす方法は使用している JavaScript のコンパイルツールによって変わってきます。

### Create React App

もし、あなたのプロジェクトが [Create React App](#) によって生成されている場合、おめでとうございます！ 既にビルド時に Flow の型定義は剥がされる設定になっているため、このステップで何もする必要はありません。

### Babel

補足:

以下の手順は Create React App で React をセットアップしている方向けではありません。Create React App も内部で Babel を使用していますが、既に Flow を使えるように設定されています。以下の手順は Create React App を使っていない場合のみ実行してください。

もし、あなたが手動で Babel の設定を行っていた場合、以下のコマンドで Flow 用の Babel プリセットをインストールする必要があります。

yarn を使っている場合：

```
yarn add --dev @babel/preset-flow
```

npm を使っている場合：

```
npm install --save-dev @babel/preset-flow
```

そして、`flow` のプリセットを [Babel の設定ファイル](#) に追加してください。例えば、Babel の設定を `.babelrc` ファイルで行っている場合は以下のようになります。

```
{
  "presets": [
    "@babel/preset-flow",
```

```
"react"
]
}
```

これであなたのコードで Flow の型定義が使えるようになります。

補足:

Flow を使うためには必ず `react` のプリセットが必要というわけではありません。ただ、よく組み合わせて使われています。Flow はそのままでも JSX のシンタックスを認識できます。

## 他のビルド設定

もし、Create React App も Babel も使用していない場合、[flow-remove-types](#) を使って Flow の型定義を剥がすことができます。

## Flow の実行

もし、今までの手順を踏んでいる場合、以下のコマンドで Flow が実行されます。

```
yarn flow
```

`npm` を使っている場合 :

```
npm run flow
```

以下のようなメッセージが表示されます。

```
No errors!
✨ Done in 0.17s.
```

## Flow の型定義の追記

Flow の初期設定では、以下のコメントが含まれているファイルのみ型チェックを行います。

```
// @flow
```

上記コメントは基本的にファイルの頭に記載します。あなたのプロジェクトのいくつかのファイルに足してみて、`yarn flow` や `npm run flow` を実行して Flow が何か問題を検知するかどうかを確認してみましょう。

Flow の型チェックを、コメントが含まれているかにかかわらず全てのファイルに対してかける[設定](#)も存在します。これは既存のプロジェクトに対してかけると確認箇所が大量に出てしまいますが、新規プロジェクトの立ち上げで全てのファイルに型を入れたい場合は合理的な選択でしょう。

これで準備は整いました！ Flow についてより深く知りたい場合には以下の資料が役立つでしょう。

- [Flow Documentation: Type Annotations](#)
- [Flow Documentation: Editors](#)
- [Flow Documentation: React](#)

- [Linting in Flow](#)

## TypeScript

TypeScript は Microsoft によって開発されたプログラミング言語です。型を所有した JavaScript のスーパー セットで、独自のコンパイラを所持しています。TypeScript は型言語であるため、実行前のビルド時にエラーやバグを検知することができます。React と TypeScript を組み合わせて使うことに関しては、[ここ](#)にも詳しく記載されています。

TypeScript を使うためには以下のことを行います。

- TypeScript をプロジェクトの dependency に入れる。
- TypeScript のコンパイラの設定を行う。
- TypeScript 用のファイル拡張子を使用する。
- 使用するライブラリの型定義をインストールする。

順を追って説明します。

### Create React App で TypeScript を使用する

Create React App には最初から TypeScript のサポートが含まれています。

TypeScript サポートが含まれている新規プロジェクトを作成するには、以下のコマンドを実行します。

```
npx create-react-app my-app --template typescript
```

この資料に記載されているように、既存の Create React App のプロジェクトにも TypeScript のサポートを追加することができます。

補足:

もし、Create React App を使っている場合、以下の記述は全て飛ばして大丈夫です。Create React App を使用していない場合のセットアップ手順について記載しています。

### TypeScript をプロジェクトに追加する

まずは TypeScript をインストールするところから始まります。プロジェクトが入っているフォルダで以下のコマンドを入力してください。

[Yarn](#) を使っている場合 :

```
yarn add --dev typescript
```

[npm](#) を使っている場合 :

```
npm install --save-dev typescript
```

おめでとうございます！ これで最新の TypeScript があなたのプロジェクトにインストールされました。これによって、`tsc` コマンドが使えるようになります。設定をする前に、まずは `tsc` を `package.json` の `"scripts"` に追加しましょう。

```
{
  // ...
  "scripts": {
    "build": "tsc",
    // ...
  },
  // ...
}
```

## TypeScript コンパイラの設定

コンパイラはこちらから設定しないと無価値です。TypeScript ではコンパイラの設定は `tsconfig.json` というファイルで定義します。このファイルを生成するには以下のコマンドを実行してください。

`Yarn` を使っている場合：

```
yarn run tsc --init
```

`npm` を使っている場合：

```
npx tsc --init
```

新しく生成された `tsconfig.json` を見てみると、コンパイラ設定用の様々なオプションがあることがわかります。これら全てのオプションの詳細に関しては[こちら](#)を参照してください。

たくさんの設定の中から `rootDir` と `outDir` に注目します。本来、コンパイラは TypeScript のファイルから JavaScript のファイルを生成します。しかし、既存のソースコードと生成されたコードが混合してしまうことは避けるべきです。

以上の問題を 2 つの手順によって対応します。

- まず、プロジェクトの構成を以下のように変更しましょう。全てのソースコードを `src` ディレクトリに移動します。

```

└── package.json
└── src
  └── index.ts
└── tsconfig.json
```

- 次に、`tsconfig.json` を編集してコンパイラにソースコードの場所と生成する場所を設定します。

```
// tsconfig.json
{
  "compilerOptions": {
    // ...
    "rootDir": "src",
    "outDir": "build"
    // ...
  },
}
```

素晴らしい！これで `build` のスクリプトを実行した際、コンパイラが生成したコードは `build` ディレクトリに格納されます。他の設定に関しては、[TypeScript React Starter](#) に React プロジェクト始めるのにおすすめな `tsconfig.json` を提供しているので、そちらを参照してください。

基本的に、生成された JavaScript のコードはバージョン管理するべきではありません。`build` フォルダは `.gitignore` に追記しましょう。

## ファイル拡張子

React ではおそらくコンポーネントを `.js` ファイルに書いていたと思います。TypeScript には 2 種類のファイル拡張子が存在します。

`.ts` がデフォルトの拡張子で、`.JSX` が含まれているファイルは `.tsx` を使います。

## TypeScript の実行

上記の手順を踏んでいれば、以下のコマンドで TypeScript を実行することができます。

```
yarn build
```

`npm` を使っている場合：

```
npm run build
```

実行結果に何も表示されない場合は、正常に完了したと言うことです。

## 型宣言

他のライブラリを使用している時の型エラーやヒントを表示させるために、コンパイラは型宣言ファイルを参照します。型宣言ファイルにはそのライブラリが使用する全ての型の情報が含まれています。これを使うことによって、`npm` 等から取得した JavaScript のライブラリをそのまま使うことができます。

ライブラリの型宣言ファイルを取得するには主に以下の 2 つの方法があります。

**Bundled** - これはライブラリ自体が型宣言ファイルを所有している場合です。この場合、ライブラリをインストールするだけでそのまま使用することができます。ライブラリが型宣言ファイルを所有しているかどうか確認するには、プロジェクトに `index.d.ts` ファイルがあるかどうかを見ます。一部のライブラリは `package.json` の `typings`、または `types` の下に型宣言ファイルのパスを指定しています。

**DefinitelyTyped** - DefinitelyTyped には型宣言ファイルがバンドルされていない様々なライブラリのための型定義が用意されています。これらの型定義はクラウドソースにより Microsoft とオープンソースのコントリビュータが管理しています。例えば、React には型宣言ファイルが含まれておませんが、DefinitelyTyped から取得することができます。取得するにはターミナルに以下のコマンドを入力してください。

```
# yarn
yarn add --dev @types/react
```

```
# npm
npm i --save-dev @types/react
```

**ローカル型定義** もし、使用しているライブラリに型宣言ファイルが含まれておらず、DefinitelyTyped にも該当する型宣言ファイルがない場合、自前で型宣言ファイルを作成することができます。それを行うには `declarations.d.ts` をソースディレクトリのルートに作成します。型宣言は以下のように行なえます。

```
declare module 'querystring' {
  export function stringify(val: object): string
  export function parse(val: string): object
}
```

これでコードを書く準備は整いました！ TypeScript についてより深く知りたい場合には以下の資料が役立つでしょう。

- [TypeScript Documentation: Everyday Types](#)
- [TypeScript Documentation: Migrating from JavaScript](#)
- [TypeScript Documentation: React and Webpack](#)

## ReScript

ReScript は JavaScript にコンパイルされる型付き言語です。コアの機能として、保証付きの 100% 型カバレッジ、高度な JSX サポート、専用の React 対応コードが含まれており、既存の JS / TS を使った React コードベースと結合することができます。

既存の JS / React コードベースに ReScript を統合するための情報は [こちら](#) で参照可能です。

## Kotlin

Kotlin は JetBrains により開発された静的型言語です。JVM、Android、LLVM、や [JavaScript](#) といったプラットフォームを対象としています。

JetBrains は [React bindings](#) や [Create React Kotlin App](#) のような様々な React コミュニティの為のツールの開発、運営を行っています。後者は React アプリをビルド設定いらずで Kotlin で書き始めることができます。

## 他の言語

JavaScript にコンパイルされ、React にも活用できる静的型言語は他にも存在します。例えば、[F#/Fable](#) と [elmish-react](#) の組み合わせです。詳細はそれぞれのサイトを確認してください。他にも React を活用できる静的型言語があれば、ぜひこのページに追記してください！

---

[Go to TOC](#)

# strict モード

`StrictMode` はアプリケーションの潜在的な問題点を洗い出すためのツールです。 `Fragment` と同様に、`StrictMode` は目に見える UI を描画しません。`StrictMode` の子孫要素に対しては、付加的な検査および警告が動くようになります。

補足：

strict モードでの検査は開発モードでのみ動きます。本番ビルドには影響を与えません。

strict モードはアプリケーションの任意の箇所で有効にできます。下はその一例です。 [embed:strict-mode/enabling-strict-mode.js](#)

上のコード例において、`Header` と `Footer` に対しては strict モードの検査はされません。しかし `ComponentOne`、`ComponentTwo` およびそのすべての子孫要素に対しては検査が働きます。

現在、`StrictMode` は以下のことに役立ちます。

- 安全でないライフサイクルの特定
- レガシーな文字列 `ref API` の使用に対する警告
- 非推奨な `findDOMNode` の使用に対する警告
- 意図しない副作用の検出
- レガシーなコンテクスト `API` の検出
- `state` の再利用性を保証する

将来の React のリリースではこの他にも機能が追加される予定です。

## 安全でないライフサイクルの特定

このブログ記事で書かれているように、いくつかのライフサイクルメソッドは非同期な React アプリケーションで使用するにあたって安全ではありません。しかしながら、アプリケーションがサードパーティのライブラリを用いているなら、そのような安全でないライフサイクルが使用されていないと保証することは難しくなります。strict モードは、幸運にもこのような場合に役立ちます！

strict モードが有効のとき、React は安全でないライフサイクルを使用した全てのクラス型コンポーネントのリストをまとめあげ、それらのコンポーネントの情報を含む下のような警告のログを出力します。

```
▶ Warning: Unsafe lifecycle methods were found within a strict-mode tree:
  in div (created by ExampleApplication)
  in ExampleApplication

componentWillMount: Please update the following components to use componentDidMount instead: ThirdPartyComponent

Learn more about this warning here:
https://fb.me/react-strict-mode-warnings
```

今 strict モードによって特定された問題に対処しておくことで、将来の React のリリース時に、並行レンダリングを活用しやすくなります。

## レガシーな文字列 ref API の使用に対する警告

以前は、React は ref を管理するためにレガシーな文字列 ref API とコールバック API の 2 つの手法を提供していました。文字列 ref API はより便利なものでしたが、いくつか不都合な点があり、公式にコールバック形式を代わりに用いることを推奨しました。

React 16.3 ではこれらの不都合なく文字列 ref の利点を活かせるような次の第 3 の選択を追加しました。 [embed:16-3-release-blog-post/create-ref-example.js](#)

オブジェクトによる ref は文字列 ref を置きかえるため主に追加されたため、現在 strict モードでは文字列 ref の使用に対して警告します。

### 補足：

コールバックによる ref は新しい `createRef` API に加えて継続してサポートされます。

コンポーネント内のコールバックによる ref を置きかえる必要はありません。コールバック ref は少しだけ柔軟に使えるため、発展的な機能として残り続けます。

新しい `createRef` API についてはこちらを参照してください。

## 非推奨な `findDOMNode` の使用に対する警告

React ではかつてクラスのインスタンスを元にツリー内の DOM ノードを見つける `findDOMNode` がサポートされていました。通常、DOM ノードに ref を付与することができるため、このような操作は必要ありません。

`findDOMNode` はクラスコンポーネントでも使用可能でしたが、これによって親要素が特定の子要素がレンダーされるのを要求する状況が許されてしまい、抽象レベルを破壊してしまってきました。このことにより、親要素が子の DOM ノードにまで踏み込んでしまう可能性があるためにコンポーネントの詳細な実装を変更できない、というようなリファクタリングの危険要因を生み出していました。`findDOMNode` は 1 番目の子要素しか返しませんが、フラグメントを使うことによりコンポーネントは複数の DOM ノードをレンダーできます。更に `findDOMNode` はその場限りの読み込み API であり、問い合わせたときにしか結果を返しません。もし子コンポーネントが別のノードをレンダーした場合に、この変化を捕捉することはできません。これらのため、`findDOMNode` はコンポーネントが絶対に変化することのない単一の DOM ノードのみを返す場合のみ有効なものでした。

代わりに `ref のフォワーディング` を使うことで、カスタムコンポーネントに ref を渡し、DOM にまで引き継ぐことでこれを明示的にすることができます。

コンポーネントのラッパーの DOM ノードを追加し、そこに直接 ref を付与することもできます。

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.wrapper = React.createRef();
  }
  render() {
    return <div ref={this.wrapper}>{this.props.children}</div>;
  }
}
```

補足：

CSS では、特定のノードをレイアウトの一部にしたくない場合 `display: contents` 属性が利用できます。

## 意図しない副作用の検出

概念的に、React は次の 2 つのフェーズで動作します。

- レンダーフェーズでは、変更対象（例えば DOM）にどのような変更が必要か決めます。このフェーズにおいて、React は `render` を呼び出し、1 つ前のレンダー結果と比較します。
- コミットフェーズで React は変更を反映します（React DOM の場合ではここで React は DOM ノードの挿入、更新、削除を行います）。React はこのフェーズで `componentDidMount` や `componentDidUpdate` などのライフサイクルの呼び出しも行います。

コミットフェーズは大体の場合非常に高速ですが、レンダーは低速になることがあります。このため、今後追加される並行モード（現状ではまだデフォルトでは無効です）ではレンダー処理を細分化し、ブラウザをブロックしてしまうことを避けるために処理を中断、再開するようになります。これは、React がコミットの前にレンダーフェーズのライフサイクルを複数回呼び出しうるということであり、（エラーや優先度の高い割り込みによって）コミットを行わずに呼び出しうるということを意味します。

レンダーフェーズのライフサイクルには次のクラス型コンポーネントのメソッドが含まれます。

- `constructor`
- `componentWillMount` (or `UNSAFE_componentWillMount`)
- `componentWillReceiveProps` (or `UNSAFE_componentWillReceiveProps`)
- `componentWillUpdate` (or `UNSAFE_componentWillUpdate`)
- `getDerivedStateFromProps`
- `shouldComponentUpdate`
- `render`
- `setState` 更新関数（第 1 引数）

上記のメソッドは複数回呼ばれることがあるため、副作用を持たないようにすることが大切です。このルールを破ると、メモリリークやアプリケーションの無効な状態など、多くの問題を引き起こします。不幸にも、これらの問題はしばしば非決定的なため、検出が難しくなります。

strict モードでは自動的には副作用を見つけてはくれませんが、それらの副作用をほんの少し決定的にして特定できる助けになります。これは、以下の関数を意図的に 2 回呼び出すことによって行われます。

- クラスコンポーネントの `constructor`, `render`, `shouldComponentUpdate` メソッド
- クラスコンポーネントの `getDerivedStateFromProps` 静的メソッド
- 関数コンポーネントの本体
- `state` 更新用関数（`setState` の第 1 引数として渡されるもの）
- `useState`, `useMemo`, `useReducer` に渡される関数

補足：

この機能は開発モードのみで適用されます。ライフサイクルは本番モードでは 2 回呼び出されることはありません。

例えば、次のようなコードを考えてみましょう。 [embed:strict-mode/side-effects-in-constructor.js](#)

はじめ見たとき、このコードには問題があるようには見えないかもしれません。しかし、`SharedApplicationState.recordEvent` が幕等ではないとすると、このコンポーネントを複数回インスタンス化するとアプリケーションの無効な状態を引き起こします。このような分かりづらいバグは開発中には現れないかもしれませんし、バグが一貫性のない挙動をして見逃してしまうかもしれません。

コンポーネントのコンストラクタなどのメソッドを意図的に 2 度呼び出すことによって、strict モードではこのようなことが起きた場合に気付きやすくしています。

#### 補足

React 17 では、2 回目にコールされたライフサイクル関数からはログが表示されないようにするため、`console.log()` のようなコンソールメソッドを自動的に書き換えていました。しかしこれは特定のケースで望ましくない挙動を引き起こしており、そのための回避策も存在していました。

React 18 以降では、React はログを一切抑止しなくなります。ただし React DevTools をインストールしている場合、2 回目以降のコールからのログはわずかに薄く表示されるようになります。また React DevTools にはそのようなログを完全に抑制するオプションも存在します（デフォルトでは off です）。

## レガシーなコンテクスト API の検出

レガシーなコンテクスト API はエラーを起こしがちで、将来のメジャーバージョンで削除予定です。16.x の全てのバージョンでは依然として動きますが、strict モードでは下のような警告文が表示されます。

```
✖ ▶Warning: Legacy context API has been detected within a strict-mode tree:  
  in div (at App.js:32)  
  in App (at index.js:7)  
  
Please update the following components: LegacyContextConsumer, LegacyContextProvider  
  
Learn more about this warning here:  
https://fb.me/react-strict-mode-warnings
```

Read the [new context API documentation](#) to help migrate to the new version.

## state 再利用可能性の保証

将来的に、React が state を保ったまま UI の一部分を追加・削除できるような機能を導入したいと考えています。例えば、ユーザがタブを切り替えて画面を離れてから戻ってきた場合に、React が以前の画面をすぐに表示できるようにしたいのです。これを可能にするため、React はアンマウントする前にコンポーネントが使用していたものと同じ state を使用してツリーを再マウントする機能をサポートする予定です。

この機能により、React の標準状態でのパフォーマンスが向上しますが、コンポーネントは副作用が何度も登録されたり破棄されたりすることに対して耐性を持つことが必要になります。ほとんどの副作用は何の変更もなく動作しますが、一部の副作用は破棄用コールバックで購読を適切にクリーンアップしていかなかったり、暗黙のうちに一度だけマウントまたは破棄されるものと想定しています。

これらの問題に気付きやすくするために、React 18 は strict モードに新しい開発時専用のチェックを導入します。この新しいチェックは、コンポーネントが初めてマウントされるたびに、すべてのコンポーネントを自動的にアンマウント・再マウントし、かつ 2 回目のマウントで以前の state を復元します。

strict モードでこの機能を有効にした場合の開発時動作を示すために、React が新しいコンポーネントをマウントするときに起こることを考えてみましょう。この機能がない状態でコンポーネントがマウントされる際、React は以下のように副作用を作成します。

- \* React がコンポーネントをマウント
- \* レイアウト副作用 (layout effect) を作成
- \* (通常の) 副作用を作成

React 18 以降の strict モードでは、開発時にコンポーネントがマウントされた場合、React はコンポーネントの即時アンマウント・再マウントをシミュレーションします：

- \* React がコンポーネントをマウント
  - \* レイアウト副作用を作成
  - \* 副作用を作成
- \* マウントされたコンポーネント内で副作用の破棄をシミュレート
  - \* レイアウト副作用を破棄
  - \* 副作用を破棄
- \* マウントされたコンポーネント内で副作用の再生成をシミュレート
  - \* レイアウト副作用を作成
  - \* 副作用の作成用コードの実行

2 度目のマウントにおいて React は初回マウント時の state を復元します。この機能はタブを操作して画面を離れてから戻ってくる、といったユーザの挙動をシミュレートしたものであり、コードが state の復元を正しく処理できることを保証できます。

コンポーネントがアンマウントされる場合は副作用は通常通り破棄されます：

- \* React がコンポーネントをアンマウント
- \* レイアウト副作用を破棄
- \* 副作用を破棄

アンマウント・再マウントでは以下が発生します：

- `componentDidMount`
- `componentWillUnmount`
- `useEffect`
- `useLayoutEffect`
- `useInsertionEffect`

補足

この挙動は開発モードの場合にのみ適用されます。本番用の挙動は変わりません。

よくある問題についてのサポートは以下を参照してください：

- [How to support Reusable State in Effects](#)

---

[Go to TOC](#)

# テスト環境

このドキュメントではあなたの環境に影響する要素や、いくつかのシナリオにおける推奨事項について概説します。

## テストランナー

[Jest](#)、[mocha](#)、[ava](#) のようなテストランナーを使えば、標準的な JavaScript を使ってテストスイートを書き、開発プロセスの一環として実行できるようにできます。加えて、テストスイートは継続的インテグレーションの一部としても実行されます。

- Jest は React プロジェクトとの広範な互換性を有しており、[モジュールやタイマー](#) のモック、[jsdom](#) のサポートを有しています。**Create React App を使っている場合、Jest は有用なデフォルト値とともに追加設定なしでインストールされます。**
- mocha のようなライブラリは実際のブラウザ環境でうまく動作するため、それが必要なテストでは有用でしょう。
- End-to-end テストは複数のページにまたがる長いフローのテストに使用され、[異なったセットアップ](#)が必要です。

## 描画画面のモック

しばしばテストは、ブラウザのような実際の描画用の画面にアクセスできない環境で実行されます。このような環境では、Node.js 内で実行される軽量のブラウザ実装である [jsdom](#) を使ってブラウザ環境をシミュレートすることをお勧めします。

多くの場合、jsdom は普通のブラウザの挙動と同様に動作しますが、[レイアウトやナビゲーション](#) のような機能は有していません。それでも、個々のテストでブラウザを立ち上げるよりも高速に動作するので、ほとんどのウェブベースのコンポーネントのテストでは有用です。また、jsdom はあなたのテストと同一のプロセスで動作するため、描画された DOM に対して内容を調べたりアサーションを行うコードを書くことができます。

本物のブラウザと全く同様に、jsdom ではユーザ操作をモデルすることができます。テストは DOM ノードに対してイベントをディスパッチして、そのアクションに対する副作用を観察したり検証したりすることができます ([例](#))。

UI テストの大部分は上記のようなセットアップを行って書くことができます。すなわちテストランナーとして Jest を使い、jsdom にレンダーし、ブラウザイベントの羅列としてユーザ操作を定義し、[act\(\)](#) ヘルプを活用します ([例](#))。例えば、React 自体のテストの多くもこの組み合わせで書かれています。

主にブラウザ特有の動作をテストする必要があるライブラリを書いており、レイアウトや本物のユーザ入力などネイティブなブラウザの挙動が必要な場合は、[mocha](#) のようなフレームワークを利用できます。

DOM をシミュレートできない環境（例えば Node.js で React Native のコンポーネントをテストする場合など）では、[イベントシミュレーションヘルプ](#) を使って要素とのインタラクションをシミュレーションできます。あるいは、[@testing-library/react-native](#) の [fireEvent](#) ヘルプを利用することもできます。

[Cypress](#)、[puppeteer](#)、[webdriver](#) のようなフレームワークは [end-to-end テスト](#) を実行するのに有用です。

## 関数のモック

テストを書く際は、テスト環境に同等物がないコード（例えば `navigator.onLine` の状態を Node.js 内で確認する、など）のモック化をしたくなります。テストではいくつかの関数を監視し、テストの他の部位がその関数とどう作用するのかを観察することもできます。そのような場合に、これらの関数をテストしやすいバージョンで選択的にモック化できれば有用です。

これはデータを取得する場面では特に有用です。本物の API エンドポイントからデータを取得することによって遅くなったり不安定になったりするのを避けるため、通常はフェイクデータを利用する方が望まれます（[例](#)）。これによりテストが予想可能になります。[Jest](#) や [sinon](#) などのライブラリはモック関数をサポートしています。End-to-end テストの場合、ネットワークのモックはより困難ですが、いずれにせよそのようなテストでは本物の API エンドポイントを使ってテストをしたいでしょう。

## モジュールのモック

コンポーネントによってはテスト環境でうまく動作しないか、テストにとって本質的ではないモジュールに依存していることがあります。このようなモジュールを選択的にモック化して適切な代替物で置き換えることは有用です（[例](#)）。

Node.js では、Jest のようなテストランナーはモジュールのモックをサポートしています。[mock-require](#) のようなライブラリを使用することもできます。

## タイマーのモック

コンポーネントは `setTimeout`、`setInterval` や `Date.now` のような時間に依存する関数を使っているかもしれません。テスト環境では、これらの関数をモック版で置き換えて、手動で時間を「進められる」ようにすることが有用です。これはテストを高速に実行させるために役立ちます！ タイマーに依存しているテストは順番通りに処理されますが、より高速になるのです（[例](#)）。[Jest](#)、[sinon](#) や [lolex](#) を含むほとんどのフレームワークにおいて、テストでモックタイマーを利用できます。

時には、タイマーのモックをやりたくない場合があります。例えばアニメーションや、タイミングに依存するエンドポイント（API レートリミッタなど）をテストしているのかもしれません。タイマーのモックが利用できるライブラリでは、テストあるいはスイートごとにモックを有効化・無効化できるようになっているため、どのようにテストを実行するかを明示的に選択できます。

## End-to-end テスト

End-to-end テストは長いワークフロー、特にあなたの業務にとってとても重要なワークフロー（例えば支払いやサインアップ）をテストするのに有用です。これらのテストをする際は、本物のブラウザがアプリケーション全体をいかに描画し、本物の API エンドポイントからいかにデータを取得し、セッションやクッキーをいかに使い、さまざまなリンク間でいかにナビゲーションするかをすべて試験したいでしょう。また、おそらく DOM の状態だけではなく、バックエンドのデータに対する検証（例えばデータベースに更新が正しく永続化されているかの確認）も行いたいかもしれません。

このようなシナリオの場合は、[Cypress](#)、[Playwright](#) のようなフレームワークや [Puppeteer](#) のようなライブラリを使うことで、複数のルート間をナビゲートし、ブラウザのみならず、必要に応じてバックエンド側の副作用についても検証を行うことができるでしょう。

---

[Go to TOC](#)

# テストのレシピ集

React コンポーネントのための一般的なテストのパターン集です。

補足：

このページではテストランナーとして [Jest](#) を使用することを前提としています。もし別のテストランナーを使う場合は API を修正する必要があるかもしれません、やり方の全体的な見た目についてはおそらく同じようなものになるはずです。テスト環境のセットアップについては [テスト環境](#) のページをご覧ください。

このページでは主に関数コンポーネントを利用します。しかし以下に述べるテスト手法は実装の詳細には依存しませんので、クラスコンポーネントでも全く同様に動作します。

- [セットアップ/後始末](#)
- [act\(\)](#)
- [レンダー](#)
- [データの取得](#)
- [モジュールのモック化](#)
- [イベント](#)
- [タイマー](#)
- [スナップショットテスト](#)
- [複数のレンダラ](#)
- [何かが足りない？](#)

## セットアップ/後始末

それぞれのテストにおいて、通常われわれは React ツリーを `document` に結びついた DOM 要素として描画することになります。これは DOM イベントを受け取れるようにするために重要です。テストが終了した際に「クリーンアップ」を行い、`document` からツリーをアンマウントします。

このためによく行うのは `beforeEach` と `afterEach` ブロックのペアを使い、それらを常に実行することで、各テストの副作用がそれ自身にとどまるようにすることです。

```
import { unmountComponentAtNode } from "react-dom";

let container = null;
beforeEach(() => {
  // setup a DOM element as a render target
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // cleanup on exiting
  unmountComponentAtNode(container);
```

```
    container.remove();
    container = null;
});
```

他のパターンを利用しても構いませんが、仮にテストが失敗した場合でもクリーンアップコードを実行するようとするべき、ということを覚えておいてください。さもなくば、テストは「穴の開いた」ものとなってしまい、あるテストが他のテストの挙動に影響するようになってしまいます。これはデバッグを困難にします。

## act() {#act}

UI テストを記述する際、レンダー、ユーザイベント、データの取得といったタスクはユーザインターフェースへのインタラクションの「ユニット (unit)」であると考えることができます。`react-dom/test-utils` が提供する `act()` というヘルパーは、あなたが何らかのアサーションを行う前に、これらの「ユニット」に関連する更新がすべて処理され、DOM に反映されていることを保証します。

```
act(() => {
  // render components
});
// make assertions
```

これによりあなたのテストは、実際のユーザがアプリケーションを使う時に体験するのと近い状況で実行されるようになります。以下の例ではこのような保証を得るために `act()` を利用しています。

直接 `act()` を使うのはちょっと冗長だと感じるかもしれません。ボイラープレートの記述を軽減するために、[React Testing Library](#) のようなライブラリを使うこともできます。このライブラリのヘルパーは `act()` でラップされています。

補足：

`act` という名称は [Arrange-Act-Assert](#) パターンから来ています。

## レンダー

与えられた `props` に対してコンポーネントが正しくレンダーされているかをテストしたいことがよくあります。`props` の内容に応じてメッセージをレンダーする以下のシンプルなコンポーネントを考えてください。

```
// hello.js

import React from "react";

export default function Hello(props) {
  if (props.name) {
    return <h1>Hello, {props.name}!</h1>;
  } else {
    return <span>Hey, stranger</span>;
  }
}
```

このコンポーネントのテストは以下のように書くことができます：

```
// hello.test.js

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";

import Hello from "./hello";

let container = null;
beforeEach(() => {
  // setup a DOM element as a render target
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // cleanup on exiting
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});

it("renders with or without a name", () => {
  act(() => {
    render(<Hello />, container);
  });
  expect(container.textContent).toBe("Hey, stranger");

  act(() => {
    render(<Hello name="Jenny" />, container);
  });
  expect(container.textContent).toBe("Hello, Jenny!");

  act(() => {
    render(<Hello name="Margaret" />, container);
  });
  expect(container.textContent).toBe("Hello, Margaret!");
});
```

## データの取得

あなたのテスト内で本物の API を呼ぶかわりに、リクエストをモック化してダミーのデータを返すようにできます。データ取得をモック化して「フェイク」のデータを使うことで、バックエンドが利用できないせいでテストが不確実にならずに済み、テストの動作が高速になります。とはいえ、テストの一部については、アプリケーション全体が協調して動作しているかを確認できる "end-to-end" のフレームワークを利用して実行したいかもしれません。

```
// user.js

import React, { useState, useEffect } from "react";

export default function User(props) {
  const [user, setUser] = useState(null);

  async function fetchUserData(id) {
    const response = await fetch(`/${id}`);
    setUser(await response.json());
  }

  useEffect(() => {
    fetchUserData(props.id);
  });
}
```

```

}, [props.id]);

if (!user) {
  return "loading...";
}

return (
<details>
  <summary>{user.name}</summary>
  <strong>{user.age}</strong> years old
  <br />
  lives in {user.address}
</details>
);
}

```

このコンポーネントに対するテストは以下のように書けます：

```

// user.test.js

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";
import User from "./user";

let container = null;
beforeEach(() => {
  // setup a DOM element as a render target
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // cleanup on exiting
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});

it("renders user data", async () => {
  const fakeUser = {
    name: "Joni Baez",
    age: "32",
    address: "123, Charming Avenue"
  };

  jest.spyOn(global, "fetch").mockImplementation(() =>
    Promise.resolve({
      json: () => Promise.resolve(fakeUser)
    })
  );

  // Use the asynchronous version of act to apply resolved promises
  await act(async () => {
    render(<User id="123" />, container);
  });

  expect(container.querySelector("summary").textContent).toBe(fakeUser.name);
  expect(container.querySelector("strong").textContent).toBe(fakeUser.age);
  expect(container.textContent).toContain(fakeUser.address);
})

```

```
// remove the mock to ensure tests are completely isolated
global.fetch.mockRestore();
});
```

## モジュールのモック化

モジュールによってはテスト環境でうまく動かないものや、テスト自体にとって本質的でないものがあります。これらのモジュールをモック化してダミーに置き換えることで、あなた自身のコードに対するテストが記述しやすくなることがあります。

サードパーティの `GoogleMap` を埋め込んでいる `Contact` というコンポーネントがあるとします：

```
// map.js

import React from "react";

import { LoadScript, GoogleMap } from "react-google-maps";
export default function Map(props) {
  return (
    <LoadScript id="script-loader" googleMapsApiKey="YOUR_API_KEY">
      <GoogleMap id="example-map" center={props.center} />
    </LoadScript>
  );
}

// contact.js

import React from "react";
import Map from "./map";

export default function Contact(props) {
  return (
    <div>
      <address>
        Contact {props.name} via{" "}
        <a data-testid="email" href={"mailto:" + props.email}>
          email
        </a>
        or on their <a data-testid="site" href={props.site}>
          website
        </a>
      </address>
      <Map center={props.center} />
    </div>
  );
}
```

我々のテスト中でこのコンポーネントをロードしたくない場合、この依存モジュール自体をダミーのコンポーネントにモック化した上でテストを実行することができます：

```
// contact.test.js

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";

import Contact from "./contact";
import MockedMap from "./map";
```

```

jest.mock("./map", () => {
  return function DummyMap(props) {
    return (
      <div data-testid="map">
        {props.center.lat}:{props.center.long}
      </div>
    );
  };
});

let container = null;
beforeEach(() => {
  // setup a DOM element as a render target
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // cleanup on exiting
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});

it("should render contact information", () => {
  const center = { lat: 0, long: 0 };
  act(() => {
    render(
      <Contact
        name="Joni Baez"
        email="test@example.com"
        site="http://test.com"
        center={center}
      />,
      container
    );
  });
  expect(
    container.querySelector("[data-testid='email']").getAttribute("href")
  ).toEqual("mailto:test@example.com");

  expect(
    container.querySelector('[data-testid="site"]').getAttribute("href")
  ).toEqual("http://test.com");

  expect(container.querySelector('[data-testid="map"]').textContent).toEqual(
    "0:0"
  );
});
});

```

## イベント

DOM 要素に対して本物の DOM イベントをディスパッチし、その結果に対してアサーションを行うことをお勧めします。以下の `Toggle` コンポーネントを考えてみましょう：

```

// toggle.js

import React, { useState } from "react";
export default function Toggle(props) {

```

```

const [state, setState] = useState(false);
return (
  <button
    onClick={() => {
      setState(previousState => !previousState);
      props.onChange(!state);
    }}
    data-testid="toggle"
  >
    {state === true ? "Turn off" : "Turn on"}
  </button>
);
}

```

これに対するテストは以下のように書けます：

```

// toggle.test.js

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";

import Toggle from "./toggle";

let container = null;
beforeEach(() => {
  // setup a DOM element as a render target
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // cleanup on exiting
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});

it("changes value when clicked", () => {
  const onChange = jest.fn();
  act(() => {
    render(<Toggle onChange={onChange} />, container);
  });

  // get a hold of the button element, and trigger some clicks on it
  const button = document.querySelector("[data-testid=toggle]");
  expect(button.innerHTML).toBe("Turn on");

  act(() => {
    button.dispatchEvent(new MouseEvent("click", { bubbles: true }));
  });

  expect(onChange).toHaveBeenCalledTimes(1);
  expect(button.innerHTML).toBe("Turn off");

  act(() => {
    for (let i = 0; i < 5; i++) {
      button.dispatchEvent(new MouseEvent("click", { bubbles: true }));
    }
  });
})

```

```
expect(onChange).toHaveBeenCalledTimes(6);
expect(button.innerHTML).toBe("Turn on");
});
```

他の DOM イベントやそれらのプロパティは [MDN](#) で解説されています。React のリスナにイベントが到達するように、作成するそれぞれのイベントに対して `{ bubbles: true }` を指定する必要があることに気を付けてください。React ではイベントは自動的にルート要素にデリゲートする形で処理されるためです。

補足：

React Testing Library にはイベントを発火するための[より簡便なヘルパー](#)があります。

## タイマー

あなたのコードでは、未来に行う作業をスケジュールするために `setTimeout` のようなタイマーベースの関数を使っているかもしれません。以下の例では、選択肢のパネルを表示してどれかが選択されるまで待ちますが、5 秒以内に選択が起きなければタイムアウトするようになっています。

```
// card.js

import React, { useEffect } from "react";

export default function Card(props) {
  useEffect(() => {
    const timeoutID = setTimeout(() => {
      props.onSelect(null);
    }, 5000);
    return () => {
      clearTimeout(timeoutID);
    };
  }, [props.onSelect]);

  return [1, 2, 3, 4].map(choice => (
    <button
      key={choice}
      data-testid={choice}
      onClick={() => props.onSelect(choice)}
    >
      {choice}
    </button>
  ));
}
```

このコンポーネントに対するテストは、[Jest のタイマーモック](#)を活用し、可能性のある状態のそれぞれをテストする、というやりかたで書くことができます。

```
// card.test.js

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";

import Card from "./card";
```

```
let container = null;
beforeEach(() => {
  // setup a DOM element as a render target
  container = document.createElement("div");
  document.body.appendChild(container);
  jest.useFakeTimers();
});

afterEach(() => {
  // cleanup on exiting
  unmountComponentAtNode(container);
  container.remove();
  container = null;
  jest.useRealTimers();
});

it("should select null after timing out", () => {
  const onSelect = jest.fn();
  act(() => {
    render(<Card onSelect={onSelect} />, container);
  });

  // move ahead in time by 100ms
  act(() => {
    jest.advanceTimersByTime(100);
  });
  expect(onSelect).not.toHaveBeenCalled();

  // and then move ahead by 5 seconds
  act(() => {
    jest.advanceTimersByTime(5000);
  });
  expect(onSelect).toHaveBeenCalledWith(null);
});

it("should cleanup on being removed", () => {
  const onSelect = jest.fn();
  act(() => {
    render(<Card onSelect={onSelect} />, container);
  });

  act(() => {
    jest.advanceTimersByTime(100);
  });
  expect(onSelect).not.toHaveBeenCalled();

  // unmount the app
  act(() => {
    render(null, container);
  });

  act(() => {
    jest.advanceTimersByTime(5000);
  });
  expect(onSelect).not.toHaveBeenCalled();
});

it("should accept selections", () => {
  const onSelect = jest.fn();
  act(() => {
    render(<Card onSelect={onSelect} />, container);
  });
});
```

```

act(() => {
  container
    .querySelector("[data-testid='2']")
    .dispatchEvent(new MouseEvent("click", { bubbles: true }));
});

expect(onSelect).toHaveBeenCalledWith(2);
});

```

フェイクタイマーは一部のテストでのみ使うようにすることができます。上記の例では、`jest.useFakeTimers()` を呼ぶことでフェイクタイマーを有効化しています。これによる主な利点は、実行されるまで実際に 5 秒待つ必要がないということと、テストのためだけにコンポーネントのコードをより複雑にする必要がない、ということです。

## スナップショットテスト

Jest のようなフレームワークでは、`toMatchSnapshot` / `toMatchInlineSnapshot` を使ってデータの「スナップショット」を保存することができます。これを使うことで、レンダーされたコンポーネントの出力を「セーブ」しておき、変更がスナップショットへの変更として明示的にコミットされるよう保証できます。

以下の例では、コンポーネントをレンダーし、作成された HTML を `pretty` パッケージで整形し、インラインスナップショットとして保存します：

```

// hello.test.js, again

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";
import pretty from "pretty";

import Hello from "./hello";

let container = null;
beforeEach(() => {
  // setup a DOM element as a render target
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // cleanup on exiting
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});

it("should render a greeting", () => {
  act(() => {
    render(<Hello />, container);
  });

  expect(
    pretty(container.innerHTML))
    .toMatchInlineSnapshot(); /* ... gets filled automatically by jest ... */

  act(() => {
    render(<Hello name="Jenny" />, container);
  });
});

```

```

expect(
  pretty(container.innerHTML)
).toMatchSnapshot(); /* ... gets filled automatically by jest ... */

act(() => {
  render(<Hello name="Margaret" />, container);
});

expect(
  pretty(container.innerHTML)
).toMatchSnapshot(); /* ... gets filled automatically by jest ... */
);

```

一般的には、スナップショットを使うよりもより個別的なアサーションを行う方がベターです。このようなテストは実装の詳細を含んでいるために壊れやすく、チームはスナップショットが壊れても気にならないようになってしまふかもしれません。選択的に一部の子コンポーネントをモックすることで、スナップショットのサイズを減らし、コードレビューで読みやすく保つことができるようになります。

## 複数のレンダラ

稀なケースにおいて、複数のレンダラを使うコンポーネントに対するテストを実行することがあるかもしれません。例えば、`react-test-renderer` を使ってコンポーネントのスナップショットテストを走らせているが、その内部で子コンポーネントが何かを表示するのに `react-dom` の `render` が使われている、ということがあるかもしれません。このようなシナリオでは、それらのレンダラーに対応する複数の `act()` で更新をラップすることができます。

```

import { act as domAct } from "react-dom/test-utils";
import { act as testAct, create } from "react-test-renderer";
// ...
let root;
domAct(() => {
  testAct(() => {
    root = create(<App />);
  });
});
expect(root).toMatchSnapshot();

```

## 何かが足りない？

もしよくあるシナリオがカバーされていない場合は、ドキュメント用ウェブサイトの[イシュートラッカ](#)でお知らせください。

# テスト概要

- "community/testing.html"

React コンポーネントは他の JavaScript のコードと同じようにテストできます。

React コンポーネントをテストするのにはいくつか方法があります、大きく 2 つのカテゴリに分けられます。

- **コンポーネントツリーのレンダー** をシンプルなテスト環境で行い、その結果を検証する
- **アプリケーション全体の動作** をブラウザ同等の環境で検証する (end-to-end テストとして知られる)

このセクションでは、最初のケースにおけるテスト戦略にフォーカスします。end-to-end テストが重要な機能のリグレーションを防ぐのに有効である一方で、そのようなテストは React コンポーネントとは特に関係なく、このセクションのスコープ外です。

## トレードオフ

テストツールを選定する時、いくつかのトレードオフを考慮することは価値があります。

- **実行速度 vs 環境の現実度**： 変更を加えてから結果を見るまでのフィードバックが早いツールは、ブラウザでの動作を正確に模倣しません。一方実際のブラウザ環境を使うようなツールは、イテレーションの速度が落ちる上 CI サーバでは壊れやすいです。
- **モックの粒度** コンポーネントのテストでは、ユニットテストとインテグレーションテストの区別は曖昧です。フォームをテストする時、そのテストはフォーム内のボタンもテストすべきでしょうか。それともボタンコンポーネント自体が自身のテストを持つべきでしょうか。ボタンのリファクタリングはフォームのテストを壊さないべきでしょうか。

チームやプロダクトに応じて、答えは違ってきます。

## 推薦ツール

**Jest** は `jsdom` を通じて DOM にアクセスできる JavaScript のテストランナーです。`jsdom` はブラウザの模倣環境にすぎませんが、React コンポーネントをテストするのには十分なことが多いです。Jest は **モジュール** や **タイマー** のモックのような機能を組み合わせて、高速にイテレーションを回すことができ、コードをどう実行するかをよりコントロールできます。

**React Testing Library** は実装の詳細に依存せずに React コンポーネントをテストすることができるツールセットです。このアプローチはリファクタリングを容易にし、さらにアクセシビリティのベスト・プラクティスへと手向けてくれます。コンポーネントを `children` 抜きに「浅く」レンダーする方法は提供していませんが、Jest のようなテストランナーで **モック** することで可能です。

## より詳しく

このセクションは 2 つのページに分かれます：

- **レシピ集**： React コンポーネントのテストを書く際の一般的なパターン集
- **環境**： React コンポーネントのためのテスト環境をセットアップする際に考えること

---

[Go to TOC](#)

# React の流儀

- 'blog/2013/11/05/thinking-in-react.html'
- 'docs/thinking-in-react-zh-CN.html'

巨大で軽快な Web アプリを開発する場合に、React は最高の手段であると、私たちは考えています。Facebook や Instagram といった私たちのサービスにおいても、とてもよくスケールしています。

React のすばらしい特長がいくつもありますが、あなたがどんなアプリを作ろうかと考えたことが、そのままアプリの作り方になる、というのはそのひとつです。本ドキュメントでは、検索可能な商品データ表を React で作っていく様子をお見せしましょう。

## モックから始めよう

すでに、JSON API が実装済みで、デザイナーからもデザインモックがもらえているとしましょう。モックは次のような見た目だったとします。

<input type="text" value="Search..."/>
<input type="checkbox"/> Only show products in stock
Name      Price
Sporting Goods
Football    \$49.99
Baseball    \$9.99
Basketball    \$29.99
Electronics
iPod Touch \$99.99
iPhone 5    \$399.99
Nexus 7    \$199.99

デザインモック

また、JSON API は次のようなデータを返してくるとしましょう。

```
[ ];
```

## Step 1: UI をコンポーネントの階層構造に落とし込む

まず最初に行うのは、モックを形作っている各コンポーネント（構成要素）を四角で囲んで、それぞれに名前をつけていくことです。もしあなたがデザイナーと一緒に仕事をしている場合は、彼らがすでにこれに相当する作業を終えている可能性がありますので、話をしに行きましょう。彼らが Photoshop でレイヤ名にしていた名前が、最終的にはあなたの React コンポーネントの名前になりうるのです！

しかし、どうやって単一のコンポーネントに括るべき範囲を見つけられるのでしょうか。ちょうど、新しい関数やオブジェクトをいつ作るのかを決めるときと、同じ手法が使えます。このような手法のひとつに、[単一責任の原則 \(single responsibility principle\)](#) があり、これはすなわち、ひとつのコンポーネントは理想的にはひとつのことだけをするべきだということです。将来、コンポーネントが肥大化してしまった場合には、小さなコンポーネントに分割するべきです。

JSON のデータモデルをユーザに向けて表示することはよくありますので、モデルを正しく構築されていれば、UI（つまりコンポーネントの構造）にもうまくマッピングされるということが分かるでしょう。これは、UI とデータモデルが同じ **情報の構造** を持つ傾向があるためです。UI を分割して、それぞれのコンポーネントがデータモデルの厳密に一部分だけを表現するよう、落とし込みましょう。



コンポーネントのネストを表した図

5 種類のコンポーネントがこのアプリの中にあることが見て取れます。それぞれの解説の中で、データを表すものについては太字にしました。図中の番号は以下の番号と対応しています。

1. **FilterableProductTable** (オレンジ色)：このサンプル全体を含む
2. **SearchBar** (青色)：すべてのユーザ入力を受け付ける
3. **ProductTable** (緑色)：ユーザ入力に基づくデータの集合を表示・フィルタする
4. **ProductCategoryRow** (水色)：カテゴリを見出しとして表示する
5. **ProductRow** (赤色)：各商品を 1 行で表示する

**ProductTable** を見てみると、表のヘッダ（「Name」や「Price」のラベルを含む）が単独のコンポーネントになっていないことがわかります。これは好みの問題で、コンポーネントにするかしないかは両論あります。今回の例でいえば、ヘッダを **ProductTable** の一部にしたのは、データの集合を描画するという **ProductTable** の責務の一環として適切だったからです。しかしながら、将来ヘッダーが肥大化して複雑になった場合（例えばソート機能を追加した場合など）は、**ProductTableHeader** のようなコンポーネントにするのが適切になるでしょう。

さて、モック内にコンポーネントを特定できましたので、階層構造に並べてみましょう。モックで他のコンポーネントの中にあるコンポーネントを、階層構造でも子要素として配置すればいいのです。次のようになります。

- **FilterableProductTable**
  - **SearchBar**
  - **ProductTable**
    - **ProductCategoryRow**
    - **ProductRow**

さて、コンポーネントの階層構造が決まつたので、アプリの実装に取り掛かりましょう。最初は、データモデルを受け取って UI の描画だけを行い、ユーザからの操作はできないというバージョンを作るのが、もっとも簡単でしょう。表示の実装とユーザ操作の実装を切り離しておくことは重要です。静的な（操作できない）バージョンを作る際には、タイプ量が多い代わりに考えることが少なく、ユーザ操作を実装するときには、考えることが多い代わりにタイプ量は少ないからです。なぜそうなのかは後で説明します。

データモデルを描画するだけの機能を持った静的なバージョンのアプリを作る際には、他のコンポーネントを再利用しつつそれらに `props` を通じてデータを渡す形で、自分のコンポーネントを組み上げていきましょう。`props` は親から子へとデータを渡すための手段です。もし、あなたが `state` に慣れ親しんでいる場合でも、今回の静的なバージョンを作る上では一切 `state` を使わないでください。`state` はユーザ操作や時間経過などで動的に変化するデータを扱うために確保されている機能です。今回のアプリは静的なバージョンなので、`state` は必要ありません。

コンポーネントはトップダウンで作っても、ボトムアップで作っても問題ありません。つまり、高い階層にあるコンポーネント（例えば `FilterableProductTable`）から作り始めて、低い階層にあるコンポーネント（`ProductRow` など）から作り始めて、どちらでもいいです。シンプルなアプリでは通常トップダウンで作った方が楽ですが、大きなプロジェクトでは開発をしながらテストを書き、ボトムアップで進める方がより簡単です。

ここまでステップを終えると、データモデルを描画する再利用可能なコンポーネントの一式が手に入ります。このアプリは静的なバージョンなので、コンポーネントは `render()` メソッドだけを持つことになります。階層構造の中で最上位のコンポーネント（`FilterableProductTable`）が、データモデルを `props` として受け取ることになるでしょう。元となるデータモデルを更新して再度 `root.render()` を呼び出すと、UI が更新されることになります。このやり方なら、複雑なことをしていないので、UI がどのように更新されて、どこを変更すればよいか、理解できることでしょう。React の**単方向データフロー**（あるいは**単方向バインディング**）により、すべてがモジュール化された高速な状態で保たれます。

このステップを実施する上で助けが必要な場合は、[React ドキュメント](#)を参照してください。

## 幕間:Props vs State

React には 2 種類の「モデル」データが存在します。`props` と `state` です。このふたつの相違を理解するのは重要なことです。違いについて自信がない場合は、[公式の React ドキュメント](#)に目を通すとよいでしょう。FAQ: `state` と `props` の違いは何ですか？もご覧ください。

## Step 3: UI 状態を表現する必要かつ十分な state を決定する

UI をインタラクティブなものにするためには元となっているデータモデルを更新できる必要があります。これは React なら `state` を使うことで実現できます。

適切に開発を進めていくにあたり、そのアプリに求められている更新可能な状態の最小構成を、最初に考えておいたほうがよいでしょう。ここで重要なのは、DRY (don't repeat yourself) の原則です。アプリケーションが必要としている最小限の状態を把握しておき、他に必要なものが出てきたら、そのとき計算すればよいのです。例えば、TODO リストを作る場合、TODO の各項目を配列で保持するだけにし、個数のカウント用に別の `state` 変数を持たないようにします。その代わりに、TODO の項目数を表示したいのであれば、配列の `length` を使えばよいのです。

今回のサンプルアプリを形作るすべてのデータについて考えてみましょう。次のようなデータがあります。

- 元となる商品のリスト
- ユーザが入力した検索文字列

- チェックボックスの値
- フィルタ済みの商品のリスト

それについて見ていき、どれが state になりうるのか考えてみます。各データについて、3 つの質問をしてみましょう。

1. 親から props を通じて与えられたデータでしょうか？ もしそうなら、それは state ではありません
2. 時間経過で変化しないままいるデータでしょうか？ もしそうなら、それは state ではありません
3. コンポーネント内にある他の props や state を使って算出可能なデータでしょうか？ もしそうなら、それは state ではありません

元となる商品のリストは props から渡されるので、これは state ではありません。検索文字列とチェックボックスは時間の経過の中で変化し、また、算出することもできないため、state だと思われます。最後に、フィルタ済みの商品のリストは state ではありません。何故ならば、元となる商品のリストと検索文字列とチェックボックスの値を組み合わせることで、フィルタ済みの商品のリストを算出することが可能だからです。

というわけで、state と呼べるのは次の 2 つです。

- ユーザが入力した検索文字列
- チェックボックスの値

## Step 4: state をどこに配置するべきなのかを明確にする

いいですね。`state` を `FilterableProductTable` の中に配置することが決まりました。では早速、インスタンス変数として `this.state = {filterText: '', inStockOnly: false}` を `FilterableProductTable` の `constructor` に追加して、初期状態をアプリに反映しましょう。その次は、`filterText` と `inStockOnly` を `ProductTable` と `SearchBar` に `props` として渡します。最後に、これらの `props` を使って `ProductTable` のフィルタ処理を行い、`SearchBar` のフォームにも値を埋めます。

これで、このアプリがどんな振る舞いをするのか見られるようになってきました。`filterText` に `"ball"` と入力した状態でアプリを更新すると、データの表が正しく更新されたことが確認できるはずです。

## Step 5: 逆方向のデータフローを追加する

ここまでで、`props` と `state` が階層構造を流れ落ちていく関数として、正しく描画を行うアプリを作ることができました。

---

[Go to TOC](#)

それでは、別の方向のデータフローもサポートしてみましょう。階層構造の奥深くにあるフォームのコンポーネントが、`FilterableProductTable` にある `state` を更新できるようにするのです。

React ではデータフローが明示的になりプログラムの動作が理解しやすくなりますが、従来の双方向データバインディングよりも少しタイプ量が増えてはしまいます。

# PropTypes を用いた型チェック

注意:

React.PropTypes は React v15.5において別パッケージに移動しました。代わりに [prop-types ライブリ](#) を利用するようにしてください。

コードを自動で変換するための [codemod スクリプト](#) を提供しています。

アプリケーションが成長するにつれて、型チェックによって多くの不具合を見つけられるようになります。アプリケーションによっては、[Flow](#) もしくは [TypeScript](#) のような JavaScript 拡張を使ってアプリケーション全体の型チェックを行うことができるでしょう。しかしそれらを使用せずとも、React は組み込みの型チェック機能を備えています。コンポーネントの props に型チェックを行うために、特別な `propTypes` プロパティを割当てることができます。

```
import PropTypes from 'prop-types';

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

Greeting.propTypes = {
  name: PropTypes.string
};
```

この例ではクラスコンポーネントを使っていますが、同様の機能は関数コンポーネントや `React.memo` および `React.forwardRef` で作ったコンポーネントでも使用できます。

`PropTypes` は受け取ったデータが有効かどうかを確認するために使用できる種々のバリデーターをエクスポートしています。上記の例では、`PropTypes.string` を使用しています。無効な値がプロパティに与えられた場合、JavaScript のコンソールに警告文が出力されます。パフォーマンス上の理由から、`propTypes` のチェックは開発モードでのみ行われます。

## PropTypes

PropTypes によって提供されている様々なバリデーターの実例を紹介します。

```
import PropTypes from 'prop-types';

MyComponent.propTypes = {
  // You can declare that a prop is a specific JS type. By default, these
  // are all optional.
  optionalArray: PropTypes.array,
  optionalBool: PropTypes.bool,
  optionalFunc: PropTypes.func,
  optionalNumber: PropTypes.number,
  optionalObject: PropTypes.object,
  optionalString: PropTypes.string,
```

```

optionalSymbol: PropTypes.symbol,
// Anything that can be rendered: numbers, strings, elements or an array
// (or fragment) containing these types.
optionalNode: PropTypes.node,
// A React element.
optionalElement: PropTypes.element,
// A React element type (ie. MyComponent).
optionalElementType: PropTypes.elementType,
// You can also declare that a prop is an instance of a class. This uses
// JS's instanceof operator.
optionalMessage: PropTypes.instanceOf(Message),
// You can ensure that your prop is limited to specific values by treating
// it as an enum.
optionalEnum: PropTypes.oneOf(['News', 'Photos']),
// An object that could be one of many types
optionalUnion: PropTypes.oneOfType([
  PropTypes.string,
  PropTypes.number,
  PropTypes.instanceOf(Message)
]),
// An array of a certain type
optionalArrayOf: PropTypes.arrayOf(PropTypes.number),
// An object with property values of a certain type
optionalObjectOf: PropTypes.objectOf(PropTypes.number),
// An object taking on a particular shape
optionalObjectWithShape: PropTypes.shape({
  color: PropTypes.string,
  fontSize: PropTypes.number
}),
// An object with warnings on extra properties
optionalObjectWithStrictShape: PropTypes.exact({
  name: PropTypes.string,
  quantity: PropTypes.number
}),
// You can chain any of the above with `isRequired` to make sure a warning
// is shown if the prop isn't provided.
requiredFunc: PropTypes.func.isRequired,
// A required value of any data type
requiredAny: PropTypes.any.isRequired,
// You can also specify a custom validator. It should return an Error
// object if the validation fails. Don't `console.warn` or throw, as this
// won't work inside `oneOfType`.
customProp: function(props, propName, componentName) {
  if (!/matchme/.test(props[propName])) {
    return new Error(
      'Invalid prop `' + propName + '` supplied to' +
      ' `' + componentName + '`. Validation failed.'
    );
}

```

```

    },
    // You can also supply a custom validator to `arrayOf` and `objectOf`.
    // It should return an Error object if the validation fails. The validator
    // will be called for each key in the array or object. The first two
    // arguments of the validator are the array or object itself, and the
    // current item's key.
    customArrayProp: PropTypes.arrayOf(function(propValue, key, componentName,
location, propFullName) {
  if (!/matchme/.test(propValue[key])) {
    return new Error(
      'Invalid prop `' + propFullName + '` supplied to' +
      ' `' + componentName + '`. Validation failed.'
    );
  }
});

```

## 単一の子要素を要求する

`PropTypes.element` を使うことで、コンポーネントに単一の子要素しか渡せないことを指定することができます。

```

import PropTypes from 'prop-types';

class MyComponent extends React.Component {
  render() {
    // This must be exactly one element or it will warn.
    const children = this.props.children;
    return (
      <div>
        {children}
      </div>
    );
  }
}

MyComponent.propTypes = {
  children: PropTypes.element.isRequired
};

```

## props のデフォルト値

`defaultProps` というプロパティを割り当てることで、`props` に値が渡されなかった際のデフォルト値を定義することができます。

```

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

// Specifies the default values for props:
Greeting.defaultProps = {
  name: 'Stranger'
};

```

```
// Renders "Hello, Stranger":
const root = ReactDOM.createRoot(document.getElementById('example'));
root.render(<Greeting />);
```

ES2022 以降は、React のコンポーネントクラス内で、`defaultProps` を静的なプロパティとして宣言することができます。詳細は[クラスのパブリック静的フィールド](#)を参照してください。これはモダンな構文であり古いブラウザではコンパイル手順が必要です。

```
class Greeting extends React.Component {
  static defaultProps = {
    name: 'stranger'
  }

  render() {
    return (
      <div>Hello, {this.props.name}</div>
    )
  }
}
```

`defaultProps` を使えば、`this.props.name` が親コンポーネントから値が指定されなかった場合でも値が代入されていることを保証できます。`propTypes` による型チェックは `defaultProps` が解決した後に行われるため、`defaultProps` にも型チェックが適用されます。

## 関数コンポーネント

普段の開発で関数コンポーネントを利用している場合、正しく PropTypes が適用されるように小さな変更を加える必要があるかもしれません。

以下のようなコンポーネントがあるとしましょう：

```
export default function HelloWorldComponent({ name }) {
  return (
    <div>Hello, {name}</div>
  )
}
```

PropTypes を加えるためには、エクスポートする前にコンポーネントを以下のように別の関数として宣言する必要があります：

```
function HelloWorldComponent({ name }) {
  return (
    <div>Hello, {name}</div>
  )
}

export default HelloWorldComponent
```

これで `HelloWorldComponent` に PropTypes を加えることができるようになります：

```
import PropTypes from 'prop-types'

function HelloWorldComponent({ name }) {
  return (
    <div>Hello, {name}</div>
```

```
)  
}  
  
HelloWorldComponent.propTypes = {  
  name: PropTypes.string  
}  
  
export default HelloWorldComponent
```

---

[Go to TOC](#)

# 非制御コンポーネント

ほとんどの場合では、フォームの実装には制御されたコンポーネント (controlled component) を使用することをお勧めしています。制御されたコンポーネントでは、フォームのデータは React コンポーネントが扱います。非制御コンポーネント (uncontrolled component) はその代替となるものであり、フォームデータを DOM 自身が扱います。

非制御コンポーネントを記述するには、各 state の更新に対してイベントハンドラを書く代わりに、[ref を使用](#)して DOM からフォームの値を取得します。

例えば、以下のコードは非制御コンポーネントで 1 つの名前を受け取ります：

```
class NameForm extends React.Component {
  constructor(props) {
    super(props);
    this.handleSubmit = this.handleSubmit.bind(this);
    this.input = React.createRef();
  }

  handleSubmit(event) {
    alert('A name was submitted: ' + this.input.current.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name:
          <input type="text" ref={this.input} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

[CodePen で試す](#)

非制御コンポーネントでは DOM に信頼できる情報源 (source of truth) を保持するため、使用すれば React と非 React のコードの統合が簡単になることがあります。汚くても構わないので速く記述したいと思うなら少しだけコード量も減らせます。そうでなければ、通常の制御されたコンポーネントを使用するべきです。

特定の状況に対してどちらのタイプのコンポーネントを使用すれば良いかまだはっきりしない場合は、[制御された入力 vs 非制御入力の記事](#)が参考になるでしょう。

## デフォルト値

React のレンダーのライフサイクルでは、フォーム要素の `value` 属性は DOM の値を上書きします。非制御コンポーネントでは、React に初期値を指定させるが後続の更新処理には関与しないようにしたいことがよくあるでしょう。このケースを扱うために、`value` の代わりに `defaultValue` 属性を指定できます。コンポーネントのマウント後に `defaultValue` 属性の値を変更しても DOM 内の値の更新は引き起こされません。

```

render() {
  return (
    <form onSubmit={this.handleSubmit}>
      <label>
        Name:
        <input
          defaultValue="Bob"
          type="text"
          ref={this.input} />
      </label>
      <input type="submit" value="Submit" />
    </form>
  );
}

```

同様に、`<input type="checkbox">` と `<input type="radio">` が `defaultChecked` を、そして `<select>` と `<textarea>` が `defaultValue` をサポートしています。

## ファイル input タグ

HTML では、`<input type="file">` を利用してユーザに 1 つ以上のファイルをデバイストレージから選択させ、サーバにアップロードしたり File API を通じて JavaScript で操作したりします。

```
<input type="file" />
```

React では、`<input type="file" />` は値がユーザだけが設定できるものでありプログラムでは操作できないため、常に非制御コンポーネントです。

ファイルを取り扱うのに File API を使用してください。以下の例では DOM ノードへの `ref` を作成し submit ハンドラでファイルにアクセスしています。

[embed:uncontrolled-components/input-type-file.js](#)

[CodePen で試す](#)

---

[Go to TOC](#)

# Web Components

React と Web Components は異なる課題を解決する為に構築されました。Web Components はコンポーネントをパッケージ化して、高い再利用性を与えます。一方で React は DOM とデータを同期させる為の宣言型のライブラリを提供しています。この 2 つの目標は互いを補完しあっています。あなたは開発者として、Web Components 内で React を使用することも、React 内で Web Components を使用することも、あるいはその両方を行うこともできます。

React を使用するほとんどの人は Web Components を使用しませんが、Web Components を用いたサードパーティ製の UI コンポーネントを使用したい時などには活用できるかもしれません。

## React で Web Components を使用する

```
class HelloMessage extends React.Component {
  render() {
    return <div>Hello <x-search>{this.props.name}</x-search>!</div>;
  }
}
```

補足:

Web Components はよく命令型の API を公開しています。例えば、`video` という Web Component が `play()` や `pause()` といった関数を公開しているかもしれません。Web Component が使う命令型の API にアクセスするには DOM ノードと直接やり取りするために `ref` を使う必要があります。サードパーティ製の Web Components を使用している場合は、Web Component のラッパとして機能する React のコンポーネントを作成するのがベストな選択でしょう。

Web Component から発された Event は React のレンダーツリーを正しく伝わってこない可能性があります。React コンポーネント内でイベントに適切に対応するにはそのためのイベントハンドラを与える必要があります。

よくある混乱のひとつとして、Web Components が "className" の代わりに "class" を使用しているケースがあります。

```
function BrickFlipbox() {
  return (
    <brick-flipbox class="demo">
      <div>front</div>
      <div>back</div>
    </brick-flipbox>
  );
}
```

## Web Components で React を使用する

```
class XSearch extends HTMLElement {
  connectedCallback() {
    const mountPoint = document.createElement('span');
    this.attachShadow({ mode: 'open' }).appendChild(mountPoint);
```

```
const name = this.getAttribute('name');
const url = 'https://www.google.com/search?q=' + encodeURIComponent(name);
const root = ReactDOM.createRoot(mountPoint);
root.render(<a href={url}>{name}</a>);
}
customElements.define('x-search', XSearch);
```

補足:

Babel を使ってクラス変換を行うと上記のコードは**機能しません**。詳細や議論は[この issue](#) を参照してください。この問題を解決するには [custom-elements-es5-adapter](#) をあなたの web component の前に読み込む必要があります。

# Colophon

This book is created by using the following sources:

- React - 日本語
- GitHub source: [reactjs/ja.reactjs.org/content](https://reactjs/ja.reactjs.org/content)
- Created: 2022-11-28
- Bash v5.2.2
- Vivliostyle, <https://vivliostyle.org/>
- By: @shinokada
- Viewer: <https://read-html-download-pdf.vercel.app/>
- GitHub repo: <https://github.com/shinokada/markdown-docs-as-pdf>
- Viewer repo: <https://github.com/shinokada/read-html-download-pdf>