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？

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 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 は次の安定版リリースでフックを全面的にサポートします。

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、フックで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、コンテクストや 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ような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ことを見つけたら Issue を報告していただければ、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**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型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` と `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。
まだフックはできたばかりですので、幾つかのサードパーティ製のライブラリは現時点でフックとの互換性がないかもしれません。

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）。とはいえ大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。
将来的には、これらのライブラリの新バージョンが、例えば `useRedux()` や `useRouter()` のようなカスタムフックをエクスポートし、ラッパコンポーネントなしで同様の機能が使えるようになるかもしれません。

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。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に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。
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';
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.render(<Counter />, container);
  });
  const button = container.querySelector('button');
  const label = container.querySelector('p');
  expect(label.textContent).toBe('You clicked 0 times');
  expect(document.title).toBe('You clicked 0 times');

  // Test second render and 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っ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。

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）。
- `componentDidCatch` と `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？

フックを使ってデータの取得をする方法について、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すか？

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` プロパティの値は変更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**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、マージを行わなくても `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 が単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時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？

現時点では、これは ref を使って手動で行うことができます：

```
function Counter() {
  const [count, setCount] = useState(0);

  const prevCountRef = useRef();
  useEffect(() => {
    prevCountRef.current = count;
  });
  const prevCount = prevCountRef.current;

  return <h1>Now: {count}, before: {prevCount}</h1>;
}
```

上記はちょっと複雑かもしれませんが、これをカスタムフックに抽出することができます。

```
function Counter() {
  const [count, setCount] = useState(0);
  const prevCount = usePrevious(count);
  return <h1>Now: {count}, before: {prevCount}</h1>;
}

function usePrevious(value) {
  const ref = useRef();
  useEffect(() => {
    ref.current = value;
  });
  return ref.current;
}
```

これは props でも state でも、その他計算されたどのような値に対しても動作します。

```
function Counter() {
  const [count, setCount] = useState(0);

  const calculation = count * 100;
  const prevCalculation = usePrevious(calculation);
  // ...
```

これは比較的よくあるユースケースですので、将来的に `usePrevious` というフックを React が最初から提供するようにする可能性があります。
派生 state における推奨されるパターンについても参照してください。

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、それを書き換えたり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下さい。

> **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}) {
  let [isScrollingDown, setIsScrollingDown] = useState(false);
  let [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が、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まりないはずですが、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。

`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() {
  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() {
  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`、`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、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 productId prop
    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);
    }
    return () => { ignore = true };
  }, [productId]);
```

副作用内に関数を移動したことで、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ず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**ことが可能です。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、副作用がとても頻繁に変化する state からの読み出しを行う場合があります。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が、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 interval がリセットされることになります。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、`setState` 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保証されているので、省略して構いません）
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.
  let 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);
  }, []); // This effect never re-runs
}
```

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可能です。

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。しかし入力である `[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**再計算されては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` を使えば入力が同じ場合のために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、**稀に `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 に設定することが可能です。

```
function Image(props) {
  const ref = useRef(null);

  // ✅ IntersectionObserver is created lazily once
  function getObserver() {
    let observer = ref.current;
    if (observer !== null) {
      return observer;
    }
    let newObserver = new IntersectionObserver(onIntersect);
    ref.current = newObserver;
    return newObserver;
  }

  // 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 がやるべき仕事も減ります。

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って渡すと子コンポーネントでの `souldComponentUpdate` 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が、面倒な『配管工事』をしている気分になることがあります。
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` 関数を作って、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、コンテキスト経由で `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チの目的で掲載しているものです。
> また、concurrent mode においてこのパターンは問題を起こす可能性があることにも注意してください。将来的にはより使いやすい代替手段を提供することを計画していますが、現時点での最も安全な解決法は、コールバックが依存している何らかの値が変わった場合はコールバックを無効化して作り直すことです。

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ーなのであれば、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();

  useLayoutEffect(() => {
    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チ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.');
  });

  useLayoutEffect(() => {
    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（あるいはそれら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った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。

このページを編集する