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 は次の安定リリース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。
Hooks 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こと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がなく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きます。

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。例えばとあるコンポーネントは `componentDidMount` と `componentDidUpdae` 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、ステート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。イベントハンドラーを `bind` するよう覚えておく必要があります。仕様が不確定な提案中の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リ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ないと思います。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。**

このページを編集する