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黄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ー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か改めて説明します。

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、あるいは手動での DOM 更新を行ったことがおありでしょう。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時される時に `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(() => {
    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 カ所だけあります — 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が、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。これまでは、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 FriendListItem(props) {
  const isOnline = useFriendStatus(props.friend.id);

  return (
    <li style={{ color: isOnline ? 'green' : 'black' }}>
      {props.friend.name}
    </li>
  );
}
```

これら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、あるいは恐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 Hooks 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 のコンテク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 Hooks 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！ まだよく分からないことがあった場合や、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。
Hooks 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。

このページを編集する