

# 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。

```
npm install eslint-plugin-react-hooks
```

```
// Your ESLint configuration
{
  "plugins": [
    // ...
    "react-hooks"
  ],
  "rules": {
    // ...
    "react-hooks/rules-of-hooks": "error", // Checks rules of Hooks
    "react-hooks/exhaustive-deps": "warning" // Checks effect dependencies
  }
}
```

将来的にはこのプラグインを Create React App や類似のツールキットにデフォルトで含めるつもりです。

**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と同じだからです。

```
// -----
// 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'
useEffect(updateTitle) // 4. Add an effect for updating the title

// -----
// Second render
// -----
useState('Mary')      // 1. Read the name state variable (argument is ignored)
useEffect(persistForm) // 2. Replace the effect for persisting the form
useState('Poppins')    // 3. Read the surname state variable (argument is 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ければ、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。しかし次のレンダー時にはユーザがフォームをクリアしているかもしれず、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わからなくなります。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して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で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。

このページを編集する