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も構わないからです。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ひとつ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、次回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。いくつかの副作用ではクリーンアップが必要な可能性があり、クリーンアップが必要な副作用は関数を返します：

```
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` に分離してしまっ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、複数の副作用を利用することができます。このため、無関係な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的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します：

```
// Mount with { friend: { id: 100 } } props
ChatAPI.subscribeToFriendStatus(100, handleStatusChange);     // Run first effect

// Update with { friend: { id: 200 } } props
ChatAPI.unsubscribeFromFriendStatus(100, handleStatusChange); // Clean up previous effect
ChatAPI.subscribeToFriendStatus(200, handleStatusChange);     // Run next effect

// Update with { friend: { id: 300 } } props
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るのです。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 Hooks 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、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ということもお忘れなく。
`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し、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 はどのようにし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 はどのように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。**

このページを編集する