フック API リファレンス

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 React 組み込みのフックについて説明します。
フックが初めての方は、先に概要ページを確認してください。よくある質問にも有用な情報が掲載されています。

基本のフック

useState

```
const [state, setState] = useState(initialState);
```

ステートフルな値と、それを更新するための関数を返します。
初回のレンダー時に返される `state` は第 1 引数として渡された値 (`initialState`) と等しくなります。
`setState` 関数は state を更新するために使用します。新しい state の値を受け取り、コンポーネントの再レンダーをスケジューリングします。

```
setState(newState);
```

後続の再レンダー時には、`useState` から返される 1 番目の値は常に、更新を適用した後の最新版の state になります。

> **補足**
> React は再レンダー間で `setState` 関数の同一性が保たれ、変化しないことを保証します。従って `useEffect` や `useCallback` の依存リストにはこの関数を含めないでも構いません。

関数型の更新
新しい state が前の state に基づいて計算される場合は、`setState` に関数を渡すことができます。この関数は前回の state の値を受け取り、更新された値を返します。以下は、`setState` の両方の形式を用いたカウンタコンポーネントの例です。

```
function Counter({initialCount}) {
  const [count, setCount] = useState(initialCount);
  return (
    <>
      Count: {count}
      <button onClick={() => setCount(initialCount)}>Reset</button>
      <button onClick={() => setCount(prevCount => prevCount + 1)}>+</button>
      <button onClick={() => setCount(prevCount => prevCount - 1)}>-</button>
    </>
  );
}
```

"+" と "-" のボタンは、更新後の値が更新前の値に基づいて計算されるため、関数形式を使っています。"Reset" ボタンは常にカウントを 0 に戻すので通常の形式を使っています。

> **補足**

クラスコンポーネントの `setState` メソッドとは異なり、`useState` は自動的な更新オブジェクトのマージを行いません。この動作は関数型の更新形式をスプレッド構文と併用することで再現可能です：

```
setState(prevState => {
  // Object.assign would also work
  return {...prevState, ...updatedValues};
});
```

別の選択肢としては `useReducer` があり、これは複数階層の値を含んだ state オブジェクトを管理する場合にはより適しています。

state の遅延初期化

`initialState` 引数は初回レンダー時に使われる state 値です。後続のレンダー時にはその値は無視されます。もし初期 state が高価な計算をして求める値である場合は、代わりに関数を渡すことができます。この関数は初回のレンダー時にのみ実行されます。

```
const [state, setState] = useState(() => {
  const initialState = someExpensiveComputation(props);
  return initialState;
});
```

state 更新の回避

現在値と同じ値で更新を行った場合、React は子のレンダーや副作用の実行を回避して処理を終了します。（React は `Object.is` による比較アルゴリズム を使用します）

useEffect

```
useEffect(didUpdate);
```

副作用を有する可能性のある命令型のコードを受け付けます。
DOM の書き換え、データの購読、タイマー、ロギング、あるいはその他の副作用を、関数コンポーネントの本体（React の**レンダーフェーズ**）で書くことはできません。それを行うと UI にまつわるややこしいバグや非整合性を引き起こします。
代わりに `useEffect` を使ってください。`useEffect` に渡された関数はレンダーの結果が画面に反映された後に動作します。副作用とは React の純粋に関数的な世界から命令型の世界への避難ハッチであると考えてください。
デフォルトでは副作用関数はレンダーが終了した後に毎回動作しますが、特定の値が変化した時のみ動作させるようにすることもできます。

エフェクトのクリーンアップ

副作用はしばしば、コンポーネントが画面から消える場合にクリーンアップする必要があるようなリソース（例えば購読やタイマー ID など）を作成します。これを実現するために、`useEffect` に渡す関数はクリーンアップ用関数を返すことができます。例えば、データ購読を作成する場合は以下のようになります。

```
useEffect(() => {
  const subscription = props.source.subscribe();
  return () => {
    // Clean up the subscription
    subscription.unsubscribe();
  };
});
```

メモリリークを防止するため、コンポーネントが UI から削除される前にクリーンアップ関数が呼び出されます。それに加えて、コンポーネントが複数回レンダーされる場合（大抵はそうですが）、**新しい副作用を実行する前に前回の副作用はクリーンアップされます**。この例では、更新が発生する度に新しい購読が作成される、ということです。毎回の更新で副作用が実行されるのを抑制するためには、後の節をご覧ください。

副作用のタイミング

`componentDidMount` や `componentDidUpdate` と異なり、`useEffect` に渡された関数はレイアウトと描画の**後で**遅延されたイベントとして実行されます。ほとんどの作業はブラウザによる画面への描画をブロックするべきではないため、購読やイベントハンドラのセットアップといったよくある副作用のほとんどにとって、この動作は適切です。
しかしすべての副作用が遅延できるわけではありません。例えばユーザに見えるような DOM の改変は、ユーザが見た目の不整合性を感じずに済むよう、次回の描画が発生する前に同期的に発生する必要があります（この違いは概念的には受動的なイベントリスナと能動的なイベントリスナの違いに似ています）。このようなタイプの副作用のため、React は `useLayoutEffect` という別のフックを提供しています。これは `useEffect` と同じシグネチャを持っており、実行されるタイミングのみが異なります。
`useEffect` はブラウザが描画を終えるまで遅延されますが、次回のレンダーが起こるより前に実行されることは保証されています。React は新しい更新を始める前に常にひとつ前のレンダーの副作用をクリーンアップします。

条件付きで副作用を実行する

デフォルトの動作では、副作用関数はレンダーの完了時に毎回実行されます。これにより、コンポーネントの依存配列のうちのひとつが変化した場合に毎回副作用が再作成されます。
しかし、上述のデータ購読の例でもそうですが、これは幾つかのケースではやりすぎです。新しい購読を設定する必要があるのは毎回の更新ごとではなく、`source` プロパティが変化した場合のみです。
これを実装するためには、`useEffect` の第 2 引数として、この副作用が依存している値の配列を渡します。変更後の例は以下のようになります。

```
useEffect(
  () => {
    const subscription = props.source.subscribe();
    return () => {
      subscription.unsubscribe();
    };
  },
  [props.source],
);
```

これで、データの購読は `props.source` が変更された場合にのみ再作成されるようになります。
空の配列 `[]` を渡すと、この副作用がコンポーネント内のどの値にも依存していないということを React に伝えることになります。つまり副作用はマウント時に実行されアンマウント時にクリーンアップされますが、更新時には実行されないようになります。

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。関数の扱い方とこの配列の値が頻繁に変わる場合の対処法も参照してください。
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。

依存の配列は副作用関数に引数として渡されるわけではありません。しかし概念としては、この記法は副作用関数の引数が何なのかを表現しています。副作用関数の内部で参照されているすべての値は入力の配列内にも現れるべきです。将来的には、コンパイラが発達すればこの配列を自動で作成することも可能であるはずです。

`useContext`

```
const context = useContext(Context);
```

コンテキストオブジェクト（`React.createContext` からの戻り値）を受け取り、該当コンテキストに対応する最も近いコンテキストプロバイダから得られる、コンテキストの現在値を返します。
プロバイダが更新された場合、このフックは最新のコンテキストの値を使って再レンダーを発生させます。

追加のフック
以下のフックは前節で説明した基本のフックの変種であったり、特定の稀なケースでのみ必要となったりするものです。最初から無理に学ぼうとしなくて構いません。

`useReducer`

```
const [state, dispatch] = useReducer(reducer, initialArg, init);
```

`useState` の代替品です。`(state, action) => newState` という型のリデューサ (reducer) を受け取り、現在の state を `dispatch` メソッドとペアにして返します。（もし Redux に馴染みがあれば、これがどう動作するのかはご存じでしょう）
通常、`useReducer` が `useState` より好ましいのは、複数の値にまたがる複雑な state ロジックがある場合や、前の state に基づいて次の state を決める必要がある場合です。また、`useReducer` を使えばコールバックの代わりに `dispatch` を下位コンポーネントに渡せるようになるため、複数階層にまたがって更新を発生させるようなコンポーネントではパフォーマンスの最適化にもなります。
以下は `useState` の部分で挙げたカウンタの例を、リデューサを使うよう書き換えたものです。

```
const initialState = {count: 0};

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return {count: state.count + 1};
    case 'decrement':
      return {count: state.count - 1};
    default:
      throw new Error();
  }
}

function Counter({initialState}) {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <>
      Count: {state.count}
      <button onClick={() => dispatch({type: 'increment'})}>+</button>
      <button onClick={() => dispatch({type: 'decrement'})}>-</button>
    </>
  );
}
```

補足
React は再レンダー間で `dispatch` 関数の同一性が保たれ、変化しないことを保証します。従って `useEffect` や `useCallback` の依存リストにはこの関数を含めないでも構いません。

初期 state の指定

`useReducer` の初期化の方法には 2 種類あります。ユースケースによりどちらかを選択してください。最も単純な方法は第 2 引数として初期 state を渡すものです。

```
  const [state, dispatch] = useReducer(
    reducer,
    {count: initialCount}
  );
```

補足
React では、リデューサの引数で `state = initialState` のようにして初期値を示すという、Redux で普及した慣習を使用しません。初期値は props に依存している可能性があるため、フックの呼び出し部分で指定します。強いこだわりがある場合は `useReducer(reducer, undefined, reducer)` という呼び出し方で Redux の振る舞いを再現できますが、お勧めはしません。

遅延初期化

初期 state の作成を遅延させることもできます。そのためには `init` 関数を第 3 引数として渡してください。初期 state が `init(initialArg)` に設定されます。
これにより初期 state の計算をリデューサの外部に抽出することができます。これはアクションに応じて state をリセットしたい場合にも便利です。

```
function init(initialCount) {
  return {count: initialCount};
}

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return {count: state.count + 1};
    case 'decrement':
      return {count: state.count - 1};
    case 'reset':
      return init(action.payload);
    default:
      throw new Error();
  }
}

function Counter({initialCount}) {
  const [state, dispatch] = useReducer(reducer, initialCount, init);
  return (
    <>
      Count: {state.count}
      <button
        onClick={() => dispatch({type: 'reset', payload: initialCount})}>

        Reset
      </button>
      <button onClick={() => dispatch({type: 'increment'})}>+</button>
      <button onClick={() => dispatch({type: 'decrement'})}>-</button>
    </>
  );
}
```

dispatch による更新の回避

state の現在値として同じ値を返した場合、React は子のレンダーや副作用の実行を回避して処理を終了します。（React は `Object.is` による比較アルゴリズム を使用します）

`useCallback`

```
const memoizedCallback = useCallback(
  () => {
    doSomething(a, b);
  },
  [a, b],
);
```

メモ化されたコールバックを返します。
インラインのコールバックとそれが依存している値の配列を渡してください。`useCallback` はそのコールバックをメモ化したものを返し、その関数は依存配列の要素のひとつが変化した場合にのみ変化します。これは、不必要なレンダーを避けるために（例えば `shouldComponentUpdate` などを使って）参照の同一性を見るよう最適化されたコンポーネントにコールバックを渡す場合に便利です。
`useCallback(fn, deps)` は `useMemo(() => fn, deps)` と等価です。

補足
依存する値の配列はコールバックに引数として渡されるわけではありません。しかし概念としては、この記法はコールバックの引数が何なのかを表現しています。コールバックの内部で参照されているすべての値は依存の配列内にも現れるべきです。将来的には、コンパイラが発達すればこの配列を自動で作成することも可能であるはずです。
`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。

useMemo

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

メモ化された値を返します。
“作成用” 関数とそれが依存する値の配列を渡してください。`useMemo` は依存配列の要素のひとつが変化した場合にのみメモ化された値を再計算します。この最適化によりレンダー毎に高価な計算が実行されるのを避けることができます。
`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この関数内で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。
配列が渡されなかった場合は、第 1 引数に新しい関数が渡された場合に常に新しい値が計算されます。（第 1 引数がインラインの関数だった場合、毎レンダー毎に値が計算されます）
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ため、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。

補足
依存する値の配列は第 1 引数の関数に引数として渡されるわけではありません。しかし概念としては、この記法は関数の引数が何なのかを表現しています。関数の内部で参照されているすべての値は依存の配列内にも現れるべきです。将来的には、コンパイラが発達すればこの配列を自動で作成することも可能であるはずです。
`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。

useRef

```
const refContainer = useRef(initialValue);
```

`useRef` はミュータブルな ref オブジェクトを返し、`.current` プロパティは渡された引数 (`initialValue`) に初期化されています。返されるオブジェクトはコンポーネントの存在期間全体にわたって存在し続けます。
よくあるユースケースは、子コンポーネントに命令型でアクセスするというものです：

```
function TextInputWithFocusButton() {
  const inputEl = useRef(null);
  const onButtonClick = () => {
    // `current` points to the mounted text input element
    inputEl.current.focus();
  };
  return (
    <>
      <input ref={inputEl} type="text" />
      <button onClick={onButtonClick}>Focus the input</button>
    </>
  );
}
```

`useRef()` は `ref` 属性で使うだけではなく、より便利に使えるということに注意してください。これはクラスでインスタンス変数を使うのと同様にして、あらゆるミュータブルな値を保持しておくのに便利です。

useImperativeHandle

```
useImperativeHandle(ref, createHandle, [deps])
```

`useImperativeHandle` は `ref` が使われた時に親コンポーネントに渡されるインスタンス値をカスタマイズするのに使います。いつもの話ですが、ref を使った手続き的なコードはほとんどの場合に避けるべきです。`useImperativeHandle` は `forwardRef` と組み合わせて使います：

```
function FancyInput(props, ref) {
  const inputRef = useRef();
  useImperativeHandle(ref, () => ({
    focus: () => {
      inputRef.current.focus();
    }
  }));
  return <input ref={inputRef} ... />;
}
FancyInput = forwardRef(FancyInput);
```

この例では、`<FancyInput ref={fancyInputRef} />` をレンダーする親コンポーネントは `fancyInputRef.current.focus()` を呼べるようになります。

useLayoutEffect

この関数のシグネチャは `useEffect` と同一ですが、DOM の変更があった後で同期的に副作用が呼び出されます。これは DOM からレイアウトを読み出して同期的に再描画を行う場合に使ってください。`useLayoutEffect` の内部でスケジュールされた更新はブラウザによって描画される前のタイミングで同期的に処理されます。
可能な場合は画面の更新がブロックするのを避けるため、標準の `useEffect` を優先して使ってください。

> **ヒント**
> `useLayoutEffect` は `componentDidMount` や `componentDidUpdate` と同じフェーズで実行されますので、クラスコンポーネントから移行しておりどちらのフックを使えばいいか自信がない場合は、恐らくリスクが最も低くなります。

useDebugValue

```
useDebugValue(value)
```

`useDebugValue` を使って React DevTools でカスタムフックのラベルを表示することができます。
例えば “独自フックの作成” で説明した `useFriendStatus` を例にします：

```
function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);

  // ...

  // Show a label in DevTools next to this Hook
  // e.g. "FriendStatus: Online"
  useDebugValue(isOnline ? 'Online' : 'Offline');

  return isOnline;
}
```

> **ヒント**
> すべてのカスタムフックにデバッグ用の値を加えるのはお勧めしません。これが最も有用なのは共有ライブラリ内のカスタムフックです。

デバッグ用の値のフォーマットを遅延させる

値を表示用にフォーマットすることが高価な処理である場合があります。また、フックが実際に DevTools でインスペクトされない場合はフォーマット自体が不要な処理です。
このため `useDebugValue` はオプションの第 2 引数としてフォーマット用関数を受け付けます。この関数はフックがインスペクトされている場合にのみ呼び出されます。この関数はデバッグ値を引数として受け取り、フォーマット済みの表示用の値を返します。
例えば `Date` 型の値を返すカスタムフックでは、以下のようなフォーマッタ関数を渡すことで、不必要に `toDateString` を呼び出すのを避けることができます。

```
useDebugValue(date, date => date.toDateString());
```

このページを編集する