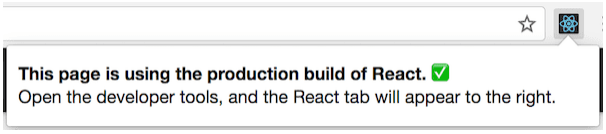
パフォーマンス最適化

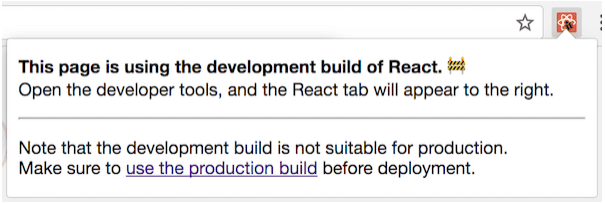
React は UI の更新時に必要となる高コストな DOM 操作の回数を最小化するために、内部的にいくつかの賢いテクニックを使用しています。多くのアプリケーションでは React を使用するだけで、パフォーマンス向上のための特別な最適化を苦労して行わなくても、レスポンスの良いユーザーインターフェースを実現できますが、それでもなお、React アプリケーションを高速化するための方法はいくつか存在します。

本番用ビルドを使用する

React アプリケーションでベンチマークを行う場合やパフォーマンスの問題が発生している場合には、ミニファイされた本番用ビルドでテストしていることを確認して下さい。
デフォルトで React は多くの有用な警告チェックを行い、開発時にはとても有用なのですが、それによって React アプリケーションのサイズは肥大化し、速度が低下してしまうので、アプリケーションのデプロイ時には本番バージョンを使用していることを確認してください。
ビルドプロセスが正しく設定されているか分からない場合、React Developer Tools for Chrome をインストールして確認できます。本番用モードの React のサイトを訪れた場合、アイコンは暗い背景となっています。



開発モードの React のサイトを訪れた場合、アイコンは赤い背景となっています。



アプリケーションに対して作業をしているときは開発モードを使用し、利用者に配布する場合には本番用モードを使用することをお勧めします。
本番用にアプリを構築するためのそれぞれのツールにおける手順を以下に示します。

Create React App

プロジェクトが Create React App で構築されているなら、以下のコードを実行して下さい。

```
npm run build
```

これでアプリケーションの本番用ビルドがプロジェクト内の `build/` フォルダに作成されます。
これが必要なのは本番用ビルドだけであることに留意してください。通常の開発作業では、`npm start` を使用してください。

単一ファイル版ビルド

React と ReactDOM をそれぞれ単一ファイル化した本番環境用のバージョンを提供しています。

```
<script src="https://unpkg.com/react@16/umd/react.production.min.js"></script>
<script src="https://unpkg.com/react-dom@16/umd/react-dom.production.min.js"></script>
```

本番用に適しているのは、React ファイル名の末尾が `.production.min.js` であるもののみであることに留意ください。

Brunch

Brunch で最も効率のよい本番用ビルドを行うには、`uglify-js-brunch` をインストールしてください：

```
# If you use npm
npm install --save-dev uglify-js-brunch

# If you use Yarn
yarn add --dev uglify-js-brunch
```

そして、本番用ビルドを作成するために、`build` コマンドに `-p` オプションを指定して実行します。

```
brunch build -p
```

これが必要なのは本番用ビルドだけであることに留意してください。React の有用な警告表示が隠されたり、ビルド速度が大幅に遅くなったりしますので、開発用では `-p` フラグを指定したり、`uglify-js-brunch` プラグインを適用したりしないでください。

Browserify

Browserify で最も効率の良い本番用ビルドを行うには、いくつかのプラグインをインストールしてください。

```
# If you use npm
npm install --save-dev envify uglify-js uglifyify

# If you use Yarn
yarn add --dev envify uglify-js uglifyify
```

本番用ビルドを作成するには、以下の変換 (transform) を追加してください（**順番は重要です**）。

- `envify` 変換は正しいビルド用の環境変数が確実に設定されるようにします。グローバルに設定してください (`-g`)。
- `uglifyify` 変換は開発用にインポートしたライブラリを削除します。これもグローバルに設定してください (`-g`)。
- 最後に結果として出力されたものを、名前の圧縮のために `uglify-js` にパイプします（理由を読む）。

以下に例を示します。

```
browserify ./index.js \
  -g [ envify --NODE_ENV production ] \
  -g uglifyify \
  | uglifyjs --compress --mangle > ./bundle.js
```

> **補足：**
> パッケージ名は `uglify-js` ですが、パッケージが提供するバイナリ名は `uglifyjs` です。
> タイプミスではありません。

これが必要なのは本番用ビルドだけであることに留意してください。React の有用な警告文が隠されたり、ビルド速度が大幅に遅くなったりしますので、開発用ではこれらのプラグインを適用しないで下さい。

Rollup

Rollup で最も効率のよい本番用ビルドを行うには、いくつかのプラグインを以下のようにインストールします。

```
# If you use npm
npm install --save-dev rollup-plugin-commonjs rollup-plugin-replace rollup-plugin-uglify

# If you use Yarn
yarn add --dev rollup-plugin-commonjs rollup-plugin-replace rollup-plugin-uglify
```

本番用ビルドを作成するには、以下のプラグインを追加してください（**順番は重要**です）。

- `replace` プラグインは正しいビルド用の環境変数が確実に設定されるようにします。
- `commonjs` プラグインは Rollup で CommonJS をサポートできるようにします。
- `uglify` プラグインは出力された最終的なバンドルを圧縮し、mangle（訳注： 変数名や識別子を短縮）します。

```
plugins: [
  // ...
  require('rollup-plugin-replace')({
    'process.env.NODE_ENV': JSON.stringify('production')
  }),
  require('rollup-plugin-commonjs')(),
  require('rollup-plugin-uglify')(),
  // ...
]
```

設定例の全体はこの gist を参照してください。
これらが必要なのは本番用ビルドだけであることに留意してください。React の有用な警告表示が隠されたり、ビルド速度が大幅に遅くなったりしますので、開発用では `uglify` プラグインもしくは `replace` プラグインを `'production'` という値で適用しないでください。

webpack

> **補足：**
> Create React App を利用している場合は、Create React App についての前述の説明に従ってください。
> このセクションは直接 webpack の設定を行いたい人向けです。

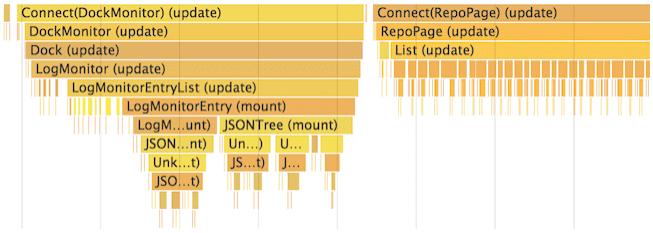
webpack で最も効率のよい本番用ビルドを行うには、本番ビルドの設定中に必ず以下のプラグインを含めるようにしてください。

```
new webpack.DefinePlugin({
  'process.env.NODE_ENV': JSON.stringify('production')
}),
new webpack.optimize.UglifyJsPlugin()
```

より詳細な説明については webpack のドキュメントを参照ください。
これらが必要なのは本番用ビルドだけであることに留意してください。React の有用な警告文が隠されたり、ビルド速度が大幅に遅くなったりしますので、開発用では `UglifyJsPlugin` もしくは `DefinePlugin` を `'production'` という値で適用しないでください。

Chrome のパフォーマンスタブでコンポーネントをプロファイルする

開発モードでは、対応するブラウザのパフォーマンス分析ツールで、コンポーネントのマウント・更新・アンマウントの様子を以下のように視覚化することができます。



Chrome での操作は以下の通り。

1. 一時的に **React DevTools を含むすべての Chrome 拡張機能を無効にする**。無効にしないと、結果が正確でなくなる可能性があります。
2. アプリケーションが開発モードで動作していることを確認する。
3. Chrome DevTools の**パフォーマンス**タブを開いて **Record（記録）** ボタンを押す。
4. プロファイル対象のアクションを実行する。なお、20 秒以上は記録しないでください。さもないと Chrome がハングアップすることがあります。
5. 記録を停止する。
6. React イベントが **User Timing** ラベルの下にグループ化される。

さらなる詳細については、Ben Schwarz によるこの記事を参照ください。
プロファイル結果の数値は相対的なものであり、コンポーネントは本番環境ではより速くレンダーされることに注意してください。それでも、無関係な UI 部分が誤って更新されているのを見つけたり、どの程度の頻度と深さで UI の更新が発生するのかを知る手助けになるはずです。
現時点では、Chrome、Edge、そして IE のみがこの機能をサポートするブラウザですが、私達は標準の User Timing API を採用しているので、より多くのブラウザがサポートしてくれることを期待しています。

DevToolsプロファイラを使用したコンポーネントのプロファイリング

`react-dom` 16.5 以降と `react-native` 0.57 以降では、開発モードにおける強化されたプロファイリング機能を React DevTools プロファイラにて提供しています。 このプロファイラの概要はブログ記事 “Introducing the React Profiler” で説明されています。 チュートリアル動画も YouTube で閲覧できます。
React DevTools をまだインストールしていない場合は、以下で見つけることができます。

- Chrome ブラウザ拡張
- Firefox ブラウザ拡張
- スタンドアロンの Node パッケージ

> **補足**
> 本番ビルド版 `react-dom` のプロファイリング可能なバンドルとして `react-dom/profiling` が利用可能です。 このバンドルの使い方の詳細については、fb.me/react-profiling を参照してください。

長いリストの仮想化

アプリケーションが長いデータのリスト（数百〜数千行）をレンダーする場合は、「ウィンドウイング」として知られるテクニックを使うことをおすすめします。このテクニックでは、ある瞬間ごとにはリストの小さな部分集合のみを描画することで、生成する DOM ノードの数およびコンポーネントの再描画にかかる時間を大幅に削減することができます。
react-window と react-virtualized は人気があるウィンドウイング処理のライブラリです。これらはリスト、グリッド、および表形式のデータを表示するための、いくつかの再利用可能コンポーネントを提供しています。アプリケーションの特定のユースケースに合わせた追加的な処理をする場合は、Twitter が行なっているように、独自のウィンドウイング処理のコンポーネントを作成することもできます。

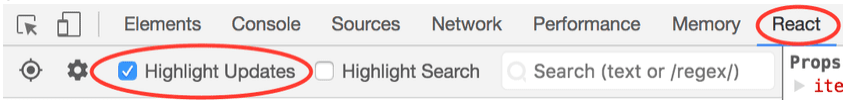
リコンシリエーション（差分検出処理）を避ける

React はレンダーされた UI の内部表現を構築し、維持します。その内部表現にはコンポーネントが返した React 要素も含まれています。React はこの内部表現を使うことによって、JavaScript オブジェクトの操作よりも操作が遅くなるかもしれない DOM ノードの不要な作成やアクセスを回避します。この内部表現はしばしば “仮想 DOM” と呼ばれますが、React Native 上でも同様に動くものです。

コンポーネントの props や state が変更された場合、React は新しく返された要素と以前にレンダーされたものとを比較することで、実際の DOM の更新が必要かを判断します。それらが等しくない場合、React は DOM を更新します。
以下の React DevTools を使用することで、仮想 DOM のこれらの再レンダーを視覚化できるようになりました。

- Chrome ブラウザ拡張
- Firefox ブラウザ拡張
- スタンドアロン Node パッケージ

開発者コンソールの **React** タブで **Highlight Updates** オプションを選択します：



ページを操作すると、再レンダーされたコンポーネントの周囲に色付きの枠線が一定時間表示されます。これにより、不要な再レンダーを見つけることができます。React DevTools のこの機能の詳細については、Ben Edelstein によるブログ投稿から学ぶことができます。
以下の例を考えてみましょう。

2 つ目の TODO 項目を入力しているとき、1 つ目の TODO 項目もキーストロークの度に画面上で点滅することに注意してください。これは、入力によって React が一緒に再レンダーしていることを意味します。これは「無駄な」レンダーと呼ばれることがあります。最初の TODO 項目の内容は変更されていないので、再レンダーの必要がないことを我々は知っていますが、React はそれを知りません。
React は変更された DOM ノードだけを更新するとはいえ、再レンダーには時間がかかります。多少の時間がかかっても多くの場合は問題にはなりませんが、遅延が目立つ場合、再レンダープロセスが開始される前にトリガーされるライフサイクル関数 `shouldComponentUpdate` をオーバーライド定義することで、スピードを抜本的に向上できます。この関数のデフォルトの実装は `true` を返し、React に更新処理をそのまま実行させます：

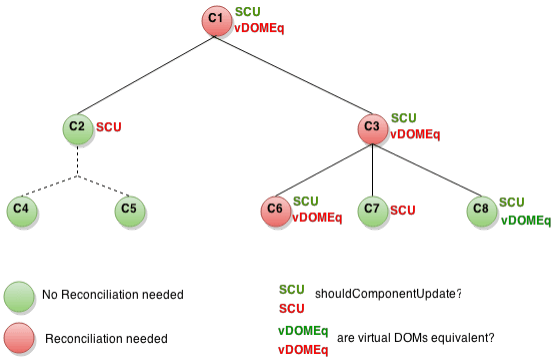
```
shouldComponentUpdate(nextProps, nextState) {
  return true;
}
```

ある状況においてコンポーネントを更新する必要がないと分かっているなら、`shouldComponentUpdate` から `false` を返すことにより、該当コンポーネントおよび配下への `render()` 呼び出しを含む、レンダー処理の全体をスキップすることができます。
ほとんどの場合には、手書きの `shouldComponentUpdate()` を定義する代わりに `React.PureComponent` を継承できます。これは現在と直前の props と state に対する浅い (shallow) 比較を行う `shouldComponentUpdate()` を実装することと同じです。

---

shouldComponentUpdate の実際の動作

以下のようなコンポーネントのサブツリーがあるとします。それぞれ、`SCU` は `shouldComponentUpdate` が返した値（訳注：緑は true、赤は false）を示し、`vDOMEq` はレンダーされた React 要素が等しかったかどうか（訳注：緑は等しい、赤は等しくない）を示します。最後に、円の色はコンポーネントに対してツリーの差分を検出するリコンシリエーション処理を必要としたのかどうか（訳注：緑は不要、赤は必要）を示します。



C2 をルートとするサブツリーでは `shouldComponentUpdate` が `false` を返したので、React は C2 をレンダーしようとしませんでした。したがって C4 と C5 については `shouldComponentUpdate` を実行する必要すらなかったわけです。
C1 と C3 では、`shouldComponentUpdate` が `true` を返したので、React は葉ノードにも移動してチェックする必要がありました。C6 では `shouldComponentUpdate` が `true` を返し、そしてレンダーされた React 要素も等しくなかったので、React は DOM を更新する必要がありました。
最後の興味深いケースが C8 です。React はこのコンポーネントをレンダーする必要がありましたが、返された React 要素は前回レンダーされたときものと同じだったので、DOM の更新は必要ありませんでした。

React が実 DOM を更新しなければならなかったのは、C6 だけだったことに注目してください。C6 の更新は避けられないものでした。C8 では、レンダーされた React 要素の比較のおかげで実 DOM を修正せずに済みました。C2 のサブツリーと C7 のケースでは `shouldComponentUpdate` のおかげで、 `render` メソッドの呼び出しや React 要素の比較処理すらスキップすることができました。

例

コンポーネントが変化するのが `props.color` または `state.count` 変数が変化した時だけだとしたら、 `shouldComponentUpdate` では以下のようなチェックを行えます。

```
class CounterButton extends React.Component {
  constructor(props) {
    super(props);
    this.state = {count: 1};
  }

  shouldComponentUpdate(nextProps, nextState) {
    if (this.props.color !== nextProps.color) {
      return true;
    }
    if (this.state.count !== nextState.count) {
      return true;
    }
    return false;
  }

  render() {
    return (
      <button
        color={this.props.color}
        onClick={() => this.setState(state => ({count: state.count + 1}))}>
        Count: {this.state.count}
      </button>
    );
  }
}
```

このコードは、 `shouldComponentUpdate` は `props.color` または `state.count` の変化の有無を単にチェックしているだけです。これらの値が変化していなければコンポーネントは更新されません。コンポーネントがもっと複雑な場合は、 `props` と `state` のすべてのフィールドに対して「浅い比較」をするという同種のパターンでコンポーネント更新の必要性を決定できます。このパターンはとても一般的なので、React はこのロジックのためのヘルパーを用意しており、 `React.PureComponent` から継承するだけで使用できます。なので以下のコードで前述のコードと同じことをよりシンプルに実装できます。

```
class CounterButton extends React.PureComponent {
  constructor(props) {
    super(props);
    this.state = {count: 1};
  }

  render() {
    return (
      <button
        color={this.props.color}
        onClick={() => this.setState(state => ({count: state.count + 1}))}>
        Count: {this.state.count}
      </button>
    );
  }
}
```

ほとんどの場合、自分で `shouldComponentUpdate` を記述する代わりに `React.PureComponent` を使うことができます。もっとも、浅い比較を行うだけですので、浅い比較では検出できない形で props や state がミューテート（mutate; 書き換え）されている可能性がある場合には使えません。
この事はより複雑なデータ構造の場合には問題となります。例えば、カンマ区切りで単語をレンダーする `ListOfWords` コンポーネントと、ボタンをクリックしてリストに単語を追加できる親コンポーネント `WordAdder` が必要だとして、以下のコードは正しく動作しません。

```
class ListOfWords extends React.PureComponent {
  render() {
    return <div>{this.props.words.join(',')}</div>;
  }
}

class WordAdder extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      words: ['marklar']
    };
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    // This section is bad style and causes a bug
    const words = this.state.words;
    words.push('marklar');
    this.setState({words: words});
  }
```

```
  render() {
    return (
      <div>
        <button onClick={this.handleClick} />
        <ListOfWords words={this.state.words} />
      </div>
    );
  }
}
```

問題は `PureComponent` が `this.props.words` の古い値と新しい値を単純に比較していることにあります。上記のコードでは `WordAdder` の handleClick メソッド内で `words` 配列の内容をミューテートしてしまうので、`this.props.words` の新旧の値は、たとえ配列内の実際の単語が変更されていたとしても、比較の結果同じだとみなしてしまうのです。そのため `ListOfWords` はレンダーすべき新しい単語が追加されているにも関わらず、更新されません。

データを変更しないことの効果

この問題を避ける最も単純な方法は、props や state として使用する値のミューテートを避けることです。例えば、上記の `handleClick` メソッドは `concat` を使って以下のように書き換えることができます：

```
handleClick() {
  this.setState(state => ({
    words: state.words.concat(['marklar'])
  }));
}
```

ES6 はこれをより簡潔に実装できる配列のスプレッド構文をサポートしています。Create React App を使用していれば、この構文はデフォルトで利用できます。

```
handleClick() {
  this.setState(state => ({
    words: [...state.words, 'marklar'],
  }));
};
```

同様に、オブジェクトについてもミューテートするコードをしないように書き換えることができます。例えば、`colormap` というオブジェクトがあり、`colormap.right` を `'blue'` に更新する関数が必要だとしましょう。以下のように書くことも可能ですが、

```
function updateColorMap(colormap) {
  colormap.right = 'blue';
}
```

この処理を、元オブジェクトをミューテートせずに実装するために、Object.assign を使用できます。

```
function updateColorMap(colormap) {
  return Object.assign({}, colormap, {right: 'blue'});
}
```

これで、`updateColorMap` は古いオブジェクトをミューテートするのではなく新しいオブジェクトを返すようになります。`Object.assign` は ES6 からの機能であり、ポリフィルが必要です（訳注：ブラウザや処理系が ES6 に未対応の場合）。
同様に、オブジェクトに対してもミューテートをしない更新を容易に記述できるようにするオブジェクトのスプレッドプロパティ構文を JavaScript に追加することが提案されています（訳注：ECMAScript 2018 で正式採用されました）：

```
function updateColorMap(colormap) {
  return {...colormap, right: 'blue'};
}
```

Create React App を使用しているなら、`Object.assign` およびオブジェクトのスプレッド構文の両方がデフォルトで利用できます。

不変（イミュータブル）なデータ構造の使用

Immutable.js はこの問題を解決する別の方法であり、構造の共有を元にした、不変で永続的なデータのコレクションを提供します。

- **不変性**: 一度作成されたら、データのコレクションはその後で変更されることはない。
- **永続性**: 既存のコレクションから、あるいはそれに set などの変更操作を行うことで新しいデータのコレクションを作成することができる。元のコレクションは新しいデータのコレクションが作成された後も有効である。
- **構造の共有**: 新しいデータのコレクションは、元のコレクションが含む同じ構造を可能な限り共有して作られるので、データのコピー量が減りパフォーマンスが向上する。

不変性により、変化を検出するためのコストが下がります。変化したデータは常に新しいオブジェクトになるので、オブジェクトの参照が違うかをどうかをチェックすればよくなるのです。例えば、以下の通常の JavaScript コードにおいて、

```
const x = { foo: 'bar' };
const y = x;
y.foo = 'baz';
x === y; // true
```

ここで `y` は編集されたにも関わらず、`x` と同じオブジェクトを参照しているため、上記の比較は `true` を返します。これと似たコードを immutable.js で書くとこうなります：

```
const SomeRecord = Immutable.Record({ foo: null });
const x = new SomeRecord({ foo: 'bar' });
const y = x.set('foo', 'baz');
const z = x.set('foo', 'bar');
x === y; // false
x === z; // true
```

この場合、`x` を変更すると新しい参照が返されるので、参照の比較（`x === y`）をするだけで、`y` に保存されている新しい値は `x` に保存されていた値とは違うことが確認できます。
不変データの使用を助けてくる他のライブラリとして seamless-immutable や immutability-helper の 2 つが挙がります。
不変データ構造はオブジェクトの変化の検出を容易にします。まさにそれが `shouldComponentUpdate` の実装に必要なことのすべてです。これによってパフォーマンスを大幅に向上できる場合があります。

このページを編集する