strict モード

`StrictMode` はアプリケーションの潜在的な問題点を洗い出すためのツールです。`Fragment` と同様に、`StrictMode` は目に見える UI を描画しません。`StrictMode` の子孫要素に対しては、付加的な検査および警告が動くようになります。

> **補足：**
> strict モードでの検査は開発モードでのみ動きます。**本番ビルドには影響を与えません。**

strict モードはアプリケーションの任意の箇所で有効にできます。下はその一例です。

```
import React from 'react';

function ExampleApplication() {
  return (
    <div>
      <Header />
      <React.StrictMode>
        <div>
          <ComponentOne />
          <ComponentTwo />
        </div>
      </React.StrictMode>
      <Footer />
    </div>
  );
}
```

上のコード例において、`Header` と `Footer` に対しては strict モードの検査は**されません。**しかし `ComponentOne`、`ComponentTwo` およびそのすべての子孫要素に対しては検査が働きます。
現在、`StrictMode` は以下のことに役立ちます。

- 安全でないライフサイクルの特定
- レガシーな文字列 ref API の使用に対する警告
- 非推奨な findDOMNode の使用に対する警告
- 意図しない副作用の検出
- レガシーなコンテクスト API の検出

将来の React のリリースではこの他にも機能が追加される予定です。

安全でないライフサイクルの特定

このブログ記事で書かれているように、いくつかのライフサイクルメソッドは非同期な React アプリケーションで使用するにあたって安全ではありません。しかしながら、アプリケーションがサードパーティーのライブラリを用いているなら、そのような安全でないライフサイクルが使用されていないと保証することは難しくなります。strict モードは、幸運にもこのような場合に役立ちます！
strict モードが有効のとき、React は安全でないライフサイクルを使用した全てのクラス型コンポーネントのリストをまとめあげ、それらのコンポーネントの情報を含む下のような警告のログを出力します。

```
▶Warning: Unsafe lifecycle methods were found within a strict-mode tree:
    in div (created by ExampleApplication)
    in ExampleApplication

componentWillMount: Please update the following components to use componentDidMount instead: ThirdPartyComponent

Learn more about this warning here:
https://fb.me/react-strict-mode-warnings
```

今 strict モードによって特定された問題に対処しておくことで、将来の React のリリース時に、非同期レンダリングを活用しやすくなります。

レガシーな文字列 ref API の使用に対する警告

以前は、React は ref を管理するためにレガシーな文字列 ref API とコールバック API の 2 つの手法を提供していました。文字列 ref API はより便利なものでしたが、いくつか不都合な点があり、公式にコールバック形式を代わりに用いることを推奨しました。
React 16.3 ではこれらの不都合なく文字列 ref の利点を活かせるような次の第 3 の選択を追加しました。

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);

    this.inputRef = React.createRef();
  }

  render() {
    return <input type="text" ref={this.inputRef} />;
  }

  componentDidMount() {
```

```
    this.inputRef.current.focus();
  }
}
```

オブジェクトによる ref は文字列 ref を置きかえるため主に追加されたため、現在 strict モードでは文字列 ref の使用に対して警告します。

> **補足：**
> コールバックによる ref は新しい `createRef` API に加えて継続してサポートされます。
> コンポーネント内のコールバックによる ref を置きかえる必要はありません。コールバック ref は少しだけ柔軟に使えるため、発展的な機能として残り続けます。

新しい `createRef` API についてはこちらを参照してください。

非推奨な findDOMNode の使用に対する警告
React ではかつてクラスのインスタンスを元にツリー内の DOM ノードを見つける `findDOMNode` がサポートされていました。通常、DOM ノードに ref を付与することができるため、このような操作は必要ありません。
`findDOMNode` はクラスコンポーネントでも使用可能でしたが、これによって親要素が特定の子要素がレンダーされるのを要求する状況が許されてしまい、抽象レベルを破壊してしまっていました。このことにより、親要素が子の DOM ノードにまで踏み込んでしまう可能性があるためにコンポーネントの詳細な実装を変更できない、というようなリファクタリングの危険要因を生み出してしまっていました。`findDOMNode` は 1 番目の子要素しか返しませんが、フラグメントを使うことによりコンポーネントはコンポーネントは複数の DOM ノードをレンダーできます。`findDOMNode` は 1 回限りの読みこみ API で、問い合わせたときの解答しか返しません。もし子コンポーネントが別のノードをレンダーしていても、この変化を管理することはできません。このため、`findDOMNode` はコンポーネントが絶対に変化することのない単一の DOM ノードのみを返す場合のみ有効といえます。
代わりに ref のフォワーディングを使うことで、カスタムコンポーネントに ref を渡し、DOM にまで引き継ぐことでこれを明示的にすることができます。
コンポーネントのラッパーの DOM ノードを追加し、そこに直接 ref を付与することもできます。

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.wrapper = React.createRef();
  }
  render() {
    return <div ref={this.wrapper}>{this.props.children}</div>;
  }
}
```

> **補足：**
> CSS では、特定のノードをレイアウトの一部にしたくない場合 `display: contents` 属性が利用できます。

意図しない副作用の検出
概念的に、React は次の 2 つのフェーズで動作します。

- **レンダー**フェーズでは、変更対象（例えば DOM）にどのような変更が必要か決めます。このフェーズにおいて、React は `render` を呼び出し、1 つ前のレンダー結果と比較します。
- **コミット**フェーズで React は変更を反映します（React DOM の場合ではここで React は DOM ノードの挿入、更新、削除を行います）。React はこのフェーズで `componentDidMount` や `componentDidUpdate` などのライフサイクルの呼び出しも行います。

コミットフェーズは大体の場合非常に高速ですが、レンダーは低速になることがあります。このため、今後追加される非同期モード（現状ではまだデフォルトでは無効です）ではレンダリングの処理を細分化し、ブラウザをブロックしてしまうことを避けるために処理を中断、再開するようになります。これは、React がコミットの前にレンダーフェーズのライフサイクルを複数回呼び出しうるということであり、（エラーや優先度の高い割り込みによって）コミットを行わずに呼び出しうるということを意味します。
レンダーフェーズのライフサイクルには次のクラス型コンポーネントのメソッドが含まれます。

- `constructor`
- `componentWillMount`
- `componentWillReceiveProps`
- `componentWillUpdate`
- `getDerivedStateFromProps`
- `shouldComponentUpdate`
- `render`
- `setState` 更新関数（第 1 引数）

上記のメソッドは複数回呼ばれることがあるため、副作用を持たないようにすることが大切です。このルールを破ると、メモリリークやアプリケーションの無効な状態など、多くの問題を引き起こしえます。不幸にも、これらの問題はしばしば非決定的なため、検出が難しくなります。
strict モードでは自動的には副作用を見つけてはくれませんが、それらの副作用をほんの少し決定的にすることによって特定できる助けになります。これは、以下のメソッドを意図的に 2 回呼び出すことによって行われます。

- クラス型コンポーネントの `constructor` メソッド
- `render` メソッド
- `setState` 更新関数（第 1 引数）

- スタティックなライフサイクル `getDerivedStateFromProps`

> **補足：**
> この機能は開発モードのみで適用されます。**ライフサイクルは本番モードでは 2 回呼び出されることはありません。**

例えば、次のようなコードを考えてみましょう。

```
class TopLevelRoute extends React.Component {
  constructor(props) {
    super(props);

    SharedApplicationState.recordEvent('ExampleComponent');
  }
}
```

はじめ見たとき、このコードには問題があるようには見えないかもしれません。しかし、`SharedApplicationState.recordEvent` が冪等ではないとすると、このコンポーネントを複数回インスタンス化するとアプリケーションの無効な状態を引き起こしえます。このような分かりづらいバグは開発中には現れないかもしれませんし、バグが一貫性のない挙動をして見逃してしまうかもしれません。
コンポーネントのコンストラクタなどのメソッドを意図的に 2 度呼び出すことによって、strict モードではこのようなことが起きた場合に気付きやすくしています。

レガシーなコンテクスト API の検出
レガシーなコンテクスト API はエラーを起こしがちで、将来のメジャーバージョンで削除予定です。16.x の全てのバージョンでは依然として動きますが、strict モードでは下のような警告文が表示されます。

```
▶Warning: Legacy context API has been detected within a strict-mode tree:
    in div (at App.js:32)
    in App (at index.js:7)

Please update the following components: LegacyContextConsumer, LegacyContextProvider

Learn more about this warning here:
https://fb.me/react-strict-mode-warnings
```

新バージョンへの移行にあたっては新コンテクスト API のドキュメントを参考にしてください。

このページを編集する