コンテクスト

コンテクストは各階層で手動でプロパティを下に渡すことなく、コンポーネントツリー内でデータを渡す方法を提供します。
典型的な React アプリケーションでは、データは props を通してトップダウン（親から子）で渡されますが、アプリケーション内の多くのコンポーネントから必要とされる特定のタイプのプロパティ（例： ロケール設定、UI テーマ）にとっては面倒です。コンテクストはツリーの各階層で明示的にプロパティを渡すことなく、コンポーネント間でこれらの様な値を共有する方法を提供します。

- コンテクストをいつ使用すべきか
- コンテクストを使用する前に
- API
  - React.createContext
  - Context.Provider
  - Class.contextType
  - Context.Consumer
- 例
  - 動的なコンテクスト
  - ネストしたコンポーネントからコンテクストを更新する
  - 複数のコンテクストを使用する
- 注意事項
- レガシーな API

コンテクストをいつ使用すべきか

コンテクストは、ある React コンポーネントのツリーに対して「グローバル」とみなすことができる、現在の認証済みユーザー・テーマ・優先言語といったデータを共有するために設計されています。例えば、以下のコードでは Button コンポーネントをスタイルする為に、手動で “theme” プロパティを通しています。

```
class App extends React.Component {
  render() {
    return <Toolbar theme="dark" />;
  }
}

function Toolbar(props) {
  // Toolbar コンポーネントは外部から "theme" プロパティを受け取り、
  // プロパティを ThemedButton へ渡します。
  // アプリ内の各ボタンがテーマを知る必要がある場合、
  // プロパティは全てのコンポーネントを通して渡される為、面倒になります。
  return (
    <div>
      <ThemedButton theme={props.theme} />
    </div>
  );
}

class ThemedButton extends React.Component {
  render() {
    return <Button theme={this.props.theme} />;
  }
}
```

コンテクストを使用することで、中間の要素群を経由してプロパティを渡すことを避けることができます。

```
// コンテクストを使用すると、全てのコンポーネントを明示的に通すことなしに
// コンポーネントツリーの深くまで値を渡すことができます。
// 現在のテーマ（デフォルトは "light"）の為のコンテクストを作成します。
const ThemeContext = React.createContext('light');

class App extends React.Component {
  render() {
    // 以下のツリーへ現在のテーマを渡すためにプロバイダを使用します。
    // どんな深さでも、どのようなコンポーネントでも読み取ることができます。
    // このサンプルでは、"dark" を現在の値として渡しています。
    return (
      <ThemeContext.Provider value="dark">
        <Toolbar />
      </ThemeContext.Provider>
    );
  }
```

```
}

// 間のコンポーネントはもう明示的にテーマを
// 下に渡す必要はありません。
function Toolbar(props) {
  return (
    <div>
      <ThemedButton />
    </div>
  );
}

class ThemedButton extends React.Component {
  // 現在のテーマのコンテクストを読むために、contextType に指定します。
  // React は上位の最も近いテーマプロバイダを見つけ、その値を使用します。
  // この例では、現在のテーマは "dark" です。
  static contextType = ThemeContext;
  render() {
    return <Button theme={this.context} />;
  }
}
```

コンテクストを使用する前に

コンテクストは主に、何らかのデータが、ネストレベルの異なる**多く**のコンポーネントからアクセスできる必要がある時に使用されます。コンテクストはコンポーネントの再利用をより難しくする為、慎重に利用してください。

もし多くの階層を経由していくつかの props を渡すことを避けたいだけであれば、コンポーネントコンポジションは多くの場合、コンテクストよりシンプルな解決策です。

例えば、深くネストされた `Link` と `Avatar` コンポーネントがプロパティを読み取ることが出来るように、`user` と `avatarSize` プロパティをいくつかの階層下へ渡す `Page` コンポーネントを考えてみましょう。

```
<Page user={user} avatarSize={avatarSize} />
// ... Page コンポーネントは以下をレンダー ...
<PageLayout user={user} avatarSize={avatarSize} />
// ... PageLayout コンポーネントは以下をレンダー ...
<NavigationBar user={user} avatarSize={avatarSize} />
// ... NavigationBar コンポーネントは以下をレンダー ...
<Link href={user.permalink}>
  <Avatar user={user} size={avatarSize} />
</Link>
```

最終的に `Avatar` コンポーネントだけがプロパティを必要としているのであれば、多くの階層を通して `user` と `avatarSize` プロパティを下に渡すことは冗長に感じるかもしれません。また、`Avatar` コンポーネントが上のコンポーネントから追加のプロパティを必要とする時はいつでも、全ての間の階層にも追加しないといけないことも厄介です。

コンテクストを使用せずにこの問題を解決する１つの手法は、`Avatar` コンポーネント自身を渡すようにするというもので、そうすれば間のコンポーネントは `user` や `avatarSize` プロパティを知る必要はありません。

```
function Page(props) {
  const user = props.user;
  const userLink = (
    <Link href={user.permalink}>
      <Avatar user={user} size={props.avatarSize} />
    </Link>
  );
  return <PageLayout userLink={userLink} />;
}

// これで以下のようになります。
<Page user={user} avatarSize={avatarSize} />
// ... Page コンポーネントは以下をレンダー ...
<PageLayout userLink={...} />
// ... PageLayout コンポーネントは以下をレンダー ...
<NavigationBar userLink={...} />
// ... NavigationBar コンポーネントは以下をレンダー ...
{props.userLink}
```

この変更により、一番上の Page コンポーネントだけが、`Link` と `Avatar` コンポーネントの `user` と `avatarSize` の使い道について知る必要があります。

この**制御の反転**はアプリケーション内で取り回す必要のあるプロパティの量を減らし、ルートコンポーネントにより多くの制御を与えることにより、多くのケースでコードを綺麗にすることができます。

しかし、この方法は全てのケースで正しい選択とはなりません：ツリー内の上層に複雑性が移ることは、それら高い階層のコンポーネントをより複雑にして、低い階層のコンポーネントに必要以上の柔軟性を強制します。

コンポーネントに対して１つの子までという制限はありません。複数の子を渡したり、子のために複数の別々の「スロット」を持つことさえできます。ドキュメントはここにあります。

```
function Page(props) {
  const user = props.user;
  const content = <Feed user={user} />;
  const topBar = (
    <NavigationBar>
      <Link href={user.permalink}>
        <Avatar user={user} size={props.avatarSize} />
      </Link>
    </NavigationBar>
  );
  return (
```

```
    <PageLayout
      topBar={topBar}
      content={content}
    />
  );
}
```

このパターンは、そのすぐ上の親から子を切り離す必要がある多くのケースにとって十分です。レンダリングの前に子が親とやり取りする必要がある場合、さらにレンダープロップと合わせて使うことができます。
しかし、たまに同じデータがツリー内の異なるネスト階層にある多くのコンポーネントからアクセス可能であることが必要となります。コンテクストはそのようなデータとその変更を以下の全てのコンポーネントへ「ブロードキャスト」できます。コンテクストを使うことが他の手法よりシンプルである一般的な例としては、現在のロケール、テーマ、またはデータキャッシュの管理が挙げられます。

API

`React.createContext`

```
const MyContext = React.createContext(defaultValue);
```

コンテクストオブジェクトを作成します。React がこのコンテクストオブジェクトが登録されているコンポーネントをレンダーする場合、ツリー内の最も近い上位の一致する `Provider` から現在のコンテクストの値を読み取ります。
`defaultValue` 引数は、コンポーネントがツリー内の上位に一致するプロバイダを持っていない場合のみ使用されます。これは、ラップしない単独でのコンポーネントのテストにて役に立ちます。補足： `undefined` をプロバイダの値として渡しても、コンシューマコンポーネントが `defaultValue` を使用することはありません。

`Context.Provider`

```
<MyContext.Provider value={/* 何らかの値 */}>
```

全てのコンテクストオブジェクトにはプロバイダ (Provider) コンポーネントが付属しており、これによりコンシューマコンポーネントはコンテクストの変更を購読できます。
プロバイダは `value` プロパティを受け取り、これが子孫であるコンシューマコンポーネントに渡されます。1 つのプロバイダは多くのコンシューマと接続することが出来ます。プロバイダは値を上書きするために、ツリー内のより深い位置でネストできます。
プロバイダの子孫の全てのコンシューマは、プロバイダの `value` プロパティが変更されるたびに再レンダーされます。プロバイダからその子孫コンシューマへの伝播は `shouldComponentUpdate` メソッドの影響を受けないため、コンシューマは祖先のコンポーネントで更新処理が止められた場合でも更新されます。
変更は、`Object.is` と同じアルゴリズムを使用し、新しい値と古い値の比較によって判断されます。

> **補足**
> この方法で変更の有無を判断するため、オブジェクトを `value` として渡した場合にいくつかの問題が発生する可能性があります。詳細は注意事項を参照。

`Class.contextType`

```
class MyClass extends React.Component {
  componentDidMount() {
    let value = this.context;
    /* MyContextの値を使用し、マウント時に副作用を実行します */
  }
  componentDidUpdate() {
    let value = this.context;
    /* ... */
  }
  componentWillUnmount() {
    let value = this.context;
    /* ... */
  }
  render() {
    let value = this.context;
    /* MyContextの値に基づいて何かをレンダーします */
  }
}
MyClass.contextType = MyContext;
```

クラスの `contextType` プロパティには `React.createContext()` により作成されたコンテキストオブジェクトを指定することができます。これにより、`this.context` を使って、そのコンテクストタイプの最も近い現在値を利用できます。レンダー関数を含むあらゆるライフサイクルメソッドで参照できます。

> **補足:**
> この API では、1 つのコンテクストだけ登録することができます。もし 2 つ以上を読み取る必要がある場合、複数のコンテクストを使用する を参照してください。
> 実験的な public class fields syntax を使用している場合、**static** クラスフィールドを使用することで `contextType` を初期化することができます。

```
class MyClass extends React.Component {
  static contextType = MyContext;
  render() {
    let value = this.context;
    /* 値に基づいて何かをレンダーします */
  }
}
```

`Context.Consumer`

```
<MyContext.Consumer>
  {value => /* コンテクストの値に基づいて何かをレンダーします */}
</MyContext.Consumer>
```

コンテクストの変更を購読する React コンポーネントです。関数コンポーネント 内でコンテクストを購読することができます。
function as a child が必要です。この関数は現在のコンテクストの値を受け取り、React ノードを返します。この関数に渡される引数 `value` は、ツリー内の上位で一番近いこのコンテクスト用のプロバイダの `value` プロパティと等しくなります。このコンテクスト用のプロバイダが上位に存在しない場合、引数の `value` は `createContext()` から渡された `defaultValue` と等しくなります。

> **補足**
> “function as a child” パターンについてさらに情報が必要な場合は レンダープロップ を参照してください。

例

動的なコンテクスト
テーマに動的な値を使用したより複雑な例：
theme-context.js

```
export const themes = {
  light: {
    foreground: '#000000',
    background: '#eeeeee',
  },
  dark: {
    foreground: '#ffffff',
    background: '#222222',
  },
};

export const ThemeContext = React.createContext(
  themes.dark // デフォルトの値
);
```

themed-button.js

```
import {ThemeContext} from './theme-context';

class ThemedButton extends React.Component {
  render() {
    let props = this.props;
    let theme = this.context;
    return (
      <button
        {...props}
        style={{backgroundColor: theme.background}}
      />
    );
  }
}
ThemedButton.contextType = ThemeContext;

export default ThemedButton;
```

app.js

```
import {ThemeContext, themes} from './theme-context';
import ThemedButton from './themed-button';

// ThemedButtonを使用する中間のコンポーネント
function Toolbar(props) {
  return (
    <ThemedButton onClick={props.changeTheme}>
      Change Theme
    </ThemedButton>
  );
```

```
}

class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      theme: themes.light,
    };

    this.toggleTheme = () => {
      this.setState(state => ({
        theme:
          state.theme === themes.dark
            ? themes.light
            : themes.dark,
      }));
    };
  }

  render() {
    // ThemeProvider 内の ThemedButton ボタンは
    // state からのテーマを使用し、外側では
    // デフォルトの dark テーマを使用します
    return (
      <Page>
        <ThemeContext.Provider value={this.state.theme}>
          <Toolbar changeTheme={this.toggleTheme} />
        </ThemeContext.Provider>
        <Section>
          <ThemedButton />
        </Section>
      </Page>
    );
  }
}

ReactDOM.render(<App />, document.root);
```

ネストしたコンポーネントからコンテクストを更新する

コンポーネントツリーのどこか深くネストされたコンポーネントからコンテクストを更新することはよく必要になります。このケースでは、コンテクストを通して下に関数を渡すことで、コンシューマがコンテクストを更新可能にすることができます。

theme-context.js

```
// createContextに渡されるデフォルトの値の形が、
// コンシューマが期待する形と一致することを確認してください！
export const ThemeContext = React.createContext({
  theme: themes.dark,
  toggleTheme: () => {},
});
```

theme-toggler-button.js

```
import {ThemeContext} from './theme-context';

function ThemeTogglerButton() {
  // ThemeTogglerButton は theme だけでなく、
  // toggleTheme 関数もコンテクストから受け取ります
  return (
    <ThemeContext.Consumer>
      {({theme, toggleTheme}) => (
        <button
          onClick={toggleTheme}
          style={{backgroundColor: theme.background}}>
          Toggle Theme
        </button>
      )}
    </ThemeContext.Consumer>
  );
}

export default ThemeTogglerButton;
```

app.js

```
import {ThemeContext, themes} from './theme-context';
import ThemeTogglerButton from './theme-toggler-button';

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

    this.toggleTheme = () => {
      this.setState(state => ({
        theme:
```

```
          state.theme === themes.dark
            ? themes.light
            : themes.dark,
      }));
    };

    // state には更新する関数も含まれているため、
    // コンテクストプロバイダにも渡されます。
    this.state = {
      theme: themes.light,
      toggleTheme: this.toggleTheme,
    };
  }

  render() {
    // state は全部プロバイダへ渡されます
    return (
      <ThemeContext.Provider value={this.state}>
        <Content />
      </ThemeContext.Provider>
    );
  }
}

function Content() {
  return (
    <div>
      <ThemeTogglerButton />
    </div>
  );
}

ReactDOM.render(<App />, document.root);
```

複数のコンテクストを使用する
コンテクストの再レンダーを高速に保つために、React は各コンテクストのコンシューマをツリー内の別々のノードにする必要があります。

```
// テーマのコンテクスト、デフォルトのテーマは light
const ThemeContext = React.createContext('light');

// サインイン済みのユーザのコンテクスト
const UserContext = React.createContext({
  name: 'Guest',
});

class App extends React.Component {
  render() {
    const {signedInUser, theme} = this.props;

    // コンテクストの初期値を与える App コンポーネント
    return (
      <ThemeContext.Provider value={theme}>
        <UserContext.Provider value={signedInUser}>
          <Layout />
        </UserContext.Provider>
      </ThemeContext.Provider>
    );
  }
}

function Layout() {
  return (
    <div>
      <Sidebar />
      <Content />
    </div>
  );
}

// コンポーネントは複数のコンテクストを使用する可能性があります
function Content() {
  return (
    <ThemeContext.Consumer>
      {theme => (
        <UserContext.Consumer>
          {user => (
            <ProfilePage user={user} theme={theme} />
          )}
        </UserContext.Consumer>
      )}
    </ThemeContext.Consumer>
  );
}
```

2 つ以上のコンテクストの値が一緒に使用されることが多い場合、両方を提供する独自のレンダープロップコンポーネントの作成を検討した方が良いかもしれません。

注意事項

コンテクストは参照の同一性を使用していつ再レンダーするかを決定するため、プロバイダの親が再レンダーするときにコンシューマで意図しないレンダーを引き起こす可能性があるいくつかの問題があります。例えば以下のコードでは、新しいオブジェクトが `value` に対して常に作成されるため、プロバイダが再レンダーするたびにすべてのコンシューマを再レンダーしてしまいます。

```
class App extends React.Component {
  render() {
    return (
      <Provider value={{something: 'something'}}>
        <Toolbar />
      </Provider>
    );
  }
}
```

この問題を回避するためには、親の state に値をリフトアップします。

```
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: {something: 'something'},
    };
  }

  render() {
    return (
      <Provider value={this.state.value}>
        <Toolbar />
      </Provider>
    );
  }
}
```

レガシーな API

> **補足**
>
> React は以前に実験的なコンテクスト API を公開していました。その古い API は全ての 16.x 系のリリースでサポートされる予定ですが、アプリケーションで使用しているのであれば、新しいバージョンにマイグレーションすべきです。レガシーな API は将来の React メジャーバージョンで削除されます。レガシーなコンテクストのドキュメントはここにあります。

このページを編集する