高階 (Higher-Order) コンポーネント

高階コンポーネント (higher-order component; HOC) はコンポーネントのロジックを再利用するための React における応用テクニックです。HOC それ自体は React の API の一部ではありません。HOC は、React のコンポジションの性質から生まれる設計パターンです。
具体的には、**高階コンポーネントとは、あるコンポーネントを受け取って新規のコンポーネントを返すような関数です。**

```
const EnhancedComponent = higherOrderComponent(WrappedComponent);
```

コンポーネントが props を UI に変換するのに対して、高階コンポーネントはコンポーネントを別のコンポーネントに変換します。
HOC は Redux における `connect` や Relay における `createFragmentContainer` のように、サードパーティ製の React ライブラリでは一般的なものです。
このドキュメントでは、なぜ高階コンポーネントが便利で、自身でどのように記述するのかを説明します。

横断的関心事に HOC を適用する

> **補足**
> 以前に横断的関心事を処理する方法としてミックスインをお勧めしました。私たちはその後にミックスインはそれが持つ価値以上の問題を引き起こすことに気づきました。ミックスインから離れる理由と、既存のコンポーネントを移行する方法についてはこちらの詳細な記事を読んでください。

コンポーネントは React のコード再利用における基本単位です。しかし、いくつかのパターンの中には、これまでのコンポーネントが素直に当てはまらないことがあることに気づいたかもしれません。
例えば、コメントのリストを描画するのに外部のデータソースの購読を行う `CommentList` コンポーネントがあるとしましょう：

```
class CommentList extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {
      // "DataSource" is some global data source
      comments: DataSource.getComments()
    };
  }

  componentDidMount() {
    // Subscribe to changes
    DataSource.addChangeListener(this.handleChange);
  }

  componentWillUnmount() {
    // Clean up listener
    DataSource.removeChangeListener(this.handleChange);
  }

  handleChange() {
    // Update component state whenever the data source changes
    this.setState({
      comments: DataSource.getComments()
    });
  }

  render() {
    return (
      <div>
        {this.state.comments.map((comment) => (
          <Comment comment={comment} key={comment.id} />
        ))}
      </div>
    );
  }
}
```

後になって、前述のパターンと似たような形で、1 件のブログ記事に関する情報を購読するコンポーネントを書くとしましょう：

```
class BlogPost extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {
      blogPost: DataSource.getBlogPost(props.id)
    };
  }
```

```
  componentDidMount() {
    DataSource.addChangeListener(this.handleChange);
  }

  componentWillUnmount() {
    DataSource.removeChangeListener(this.handleChange);
  }

  handleChange() {
    this.setState({
      blogPost: DataSource.getBlogPost(this.props.id)
    });
  }

  render() {
    return <TextBlock text={this.state.blogPost} />;
  }
}
```

`CommentList` と `BlogPost` は同一ではありません。`DataSource` に対して異なるメソッドを呼び出し、異なる出力を描画します。しかし、それらの実装の大部分は同じです：

- コンポーネントのマウント時に、`DataSource` にイベントリスナを登録する。
- リスナの内部で、`setState` をデータソースが変更されるたびに呼び出す。
- コンポーネントのアンマウント時には、イベントリスナを削除する。

大規模なアプリケーションにおいては、`DataSource` を購読して `setState` を呼び出すという同様のパターンが何度も発生することが想像できるでしょう。1 つの場所にロジックを定義し、多数のコンポーネントを横断してロジックを共有可能にするような抽象化が欲しいところです。このような場合には高階コンポーネントが有効です。
コンポーネントを作成するような関数を書いて、`DataSource` からデータを受け取る、`CommentList` や `BlogPost` のようなコンポーネントを作り出せます。その関数は引数の 1 つとして子コンポーネントを受け取り、その子コンポーネントは購読したデータを props の一部として受け取ります。この関数を `withSubscription` と呼ぶことにしましょう。

```
const CommentListWithSubscription = withSubscription(
  CommentList,
  (DataSource) => DataSource.getComments()
);

const BlogPostWithSubscription = withSubscription(
  BlogPost,
  (DataSource, props) => DataSource.getBlogPost(props.id)
);
```

1 つ目の引数はラップされるコンポーネントです。2 つ目の引数は、与えられた `DataSource` と現在の props をもとに、関心のあるデータを取り出します。
`CommentListWithSubscription` と `BlogPostWithSubscription` が描画されると、`CommentList` と `BlogPost` は `DataSource` から取得した最新データを `data` プロパティとして受け取ります：

```
// This function takes a component...
function withSubscription(WrappedComponent, selectData) {
  // ...and returns another component...
  return class extends React.Component {
    constructor(props) {
      super(props);
      this.handleChange = this.handleChange.bind(this);
      this.state = {
        data: selectData(DataSource, props)
      };
    }

    componentDidMount() {
      // ... that takes care of the subscription...
      DataSource.addChangeListener(this.handleChange);
    }

    componentWillUnmount() {
      DataSource.removeChangeListener(this.handleChange);
    }

    handleChange() {
      this.setState({
        data: selectData(DataSource, this.props)
      });
    }

    render() {
      // ... and renders the wrapped component with the fresh data!
      // Notice that we pass through any additional props
      return <WrappedComponent data={this.state.data} {...this.props} />;
    }
  };
}
```

HOC は入力のコンポーネントを改変したり、振る舞いをコピーするのに継承を利用したりしません。むしろ HOC は元のコンポーネントをコンテナコンポーネント内に**ラップする**ことで**組み合わせる**のです。HOC は副作用のない純関数です。
それだけです！ ラップされたコンポーネントはコンテナの props のすべてに加えて新規のプロパティである `data` を受け取り、出力の描画に使用します。外側にある HOC は渡すデータが使われる方法や理由には関心がありませんし、ラップされたコンポーネントの側はデータがどこからやって来たのかには関心を持ちません。

`withSubscription` は通常の関数なので、引数を好きなだけ増やしたり減らしたりできます。例えば、`data` プロパティの名前を変更可能にして、HOC をラップされるコンポーネントから更に分離させることもできるでしょう。もしくは `shouldComponentUpdate` を設定する引数を受け取ったり、データソースを設定する引数を受け取りたいこともあるかもしれません。HOC ではコンポーネントがどのように定義されるかを完全に管理できるため、上述のことは全て実現できます。
コンポーネントのように、`withSubscription` とラップされるコンポーネントの間の契約は完全に props に基づいています。これにより同じ props をラップされるコンポーネントに与える限りは、ある HOC を他の HOC と簡単に交換できます。このことは例えばデータ取得ライブラリを変更する場合に便利でしょう。

元のコンポーネントを変更するのではなく、コンポジションを使うこと

HOC の中でコンポーネントのプロトタイプを変更したり、あるいは何にせよコンポーネントに変更を加えたりしたくなる誘惑に負けてはいけません。

```
function logProps(InputComponent) {
  InputComponent.prototype.componentWillReceiveProps = function(nextProps) {
    console.log('Current props: ', this.props);
    console.log('Next props: ', nextProps);
  };
  // The fact that we're returning the original input is a hint that it has
  // been mutated.
  return InputComponent;
}

// EnhancedComponent will log whenever props are received
const EnhancedComponent = logProps(InputComponent);
```

このコードにはいくつかの問題があります。1 つは入力のコンポーネントを改変されたコンポーネントとは別に再利用できなくなってしまうことです。さらに悪いことに、もしこの `EnhancedComponent` に別の HOC を適用し、それが**同様に** `componentWillReceiveProps` に変更を加えるものであった場合、最初の HOC が加えた機能は上書きされてしまいます！ またこの HOC はライフサイクルメソッドを持たない関数型コンポーネントには機能しません。
コンポーネントの改変を行うような HOC は不完全な抽象化です。つまり、利用する側は他の HOC との競合を避けるため、どのように実装されているかを知っておく必要があるのです。
改変を行う代わりに、HOC はコンテナコンポーネントで入力されたコンポーネントをラップすることによるコンポジションを使用するべきです：

```
function logProps(WrappedComponent) {
  return class extends React.Component {
    componentWillReceiveProps(nextProps) {
      console.log('Current props: ', this.props);
      console.log('Next props: ', nextProps);
    }
    render() {
      // Wraps the input component in a container, without mutating it. Good!
      return <WrappedComponent {...this.props} />;
    }
  }
}
```

この HOC は改変を行うバージョンと同等の機能を持ちつつ、衝突の可能性を回避しています。クラス型と関数コンポーネントのどちらでも同様にうまく動作します。そして純関数なので、自分自身を含めた他の HOC と組み合わせることができます。
おそらく HOC と**コンテナコンポーネント**と呼ばれるパターンの類似性に気づいたでしょう。コンテナコンポーネントは高レベルと低レベルの関心事の責任を分離する戦略の一部です。コンテナはデータ購読や state を管理してコンポーネントに props を渡し、渡された側のコンポーネントは UI の描画などの事柄を取り扱います。HOC はコンテナをその実装の一部として使用します。HOC をパラメータ化されたコンテナコンポーネントの定義であると考えることができます。

規則：自身に関係のない props はラップされるコンポーネントにそのまま渡すこと

HOC はコンポーネントに機能を追加するものです。その props にもとづく契約は大きく変更すべきではありません。HOC の返り値のコンポーネントはラップされたコンポーネントと似たようなインターフェースを持つことが期待されます。
HOC はその特定の関心とは関係のない props はラップされる関数に渡すべきです。大抵の HOC はこのような描画メソッドを持ちます：

```
render() {
  // Filter out extra props that are specific to this HOC and shouldn't be
  // passed through
  const { extraProp, ...passThroughProps } = this.props;

  // Inject props into the wrapped component. These are usually state values or
  // instance methods.
  const injectedProp = someStateOrInstanceMethod;

  // Pass props to wrapped component
  return (
    <WrappedComponent
      injectedProp={injectedProp}
      {...passThroughProps}
    />
  );
}
```

この決まり事により、HOC が可能な限り柔軟で再利用しやすいものになります。

規則：組み立てやすさを最大限保つこと

すべての HOC が同じ見た目になるわけではありません。引数としてラップされるコンポーネント 1 つだけを受け取ることがあります。

```
const NavbarWithRouter = withRouter(Navbar);
```

通常、HOC は追加の引数を受け取ります。この Relay からの例では、config オブジェクトがコンポーネントのデータ依存を指定するために使われています：

```
const CommentWithRelay = Relay.createContainer(Comment, config);
```

もっとも一般的な HOC の型シグネチャはこのようなものです：

```
// React Redux's `connect`
const ConnectedComment = connect(commentSelector, commentActions)(CommentList);
```

これは何なのでしょう?! バラバラにしてみると、何が起こっているのかを理解しやすくなります。

```
// connect is a function that returns another function
const enhance = connect(commentListSelector, commentListActions);
// The returned function is a HOC, which returns a component that is connected
// to the Redux store
const ConnectedComment = enhance(CommentList);
```

言い換えれば、`connect` は高階コンポーネントを返す高階関数なのです！
この形式は分かりにくかったり不要なものに思えるかもしれませんが、便利な性質を持っています。`connect` 関数によって返されるもののような単一引数の HOC は、`Component => Component` という型シグネチャを持ちます。入力の型と出力の型が同じ関数は一緒に組み合わせるのが大変簡単なのです。

```
// Instead of doing this...
const EnhancedComponent = withRouter(connect(commentSelector)(WrappedComponent))

// ... you can use a function composition utility
// compose(f, g, h) is the same as (...args) => f(g(h(...args)))
const enhance = compose(
  // These are both single-argument HOCs
  withRouter,
  connect(commentSelector)
)
const EnhancedComponent = enhance(WrappedComponent)
```

（この性質を使えば、`connect` や他の機能追加方式の HOC をデコレータ（提唱中の JavaScript の実験的機能）で使用することも可能になります）
`compose` ユーティリティ関数は lodash（`lodash.flowRight` として）、Redux、そして Ramda といった多くのサードパーティ製ライブラリから提供されています。

規則：デバッグしやすくするため表示名をラップすること

HOC により作成されたコンテナコンポーネントは他のあらゆるコンポーネントと同様、React Developer Tools に表示されます。デバッグを容易にするため、HOC の結果だと分かるよう表示名を選んでください。
最も一般的な手法は、ラップされるコンポーネントの表示名をラップすることです。つまり高階コンポーネントが `withSubscription` と名付けられ、ラップされるコンポーネントの表示名が `CommentList` である場合、`WithSubscription(CommentList)` という表示名を使用しましょう：

```
function withSubscription(WrappedComponent) {
  class WithSubscription extends React.Component {/* ... */}
  WithSubscription.displayName = `WithSubscription(${getDisplayName(WrappedComponent)})`;
  return WithSubscription;
}

function getDisplayName(WrappedComponent) {
  return WrappedComponent.displayName || WrappedComponent.name || 'Component';
}
```

注意事項

高階コンポーネントには、あなたが React を始めて間もないならすぐには分からないような、いくつかの注意事項があります。

render メソッド内部で HOC を使用しないこと

React の差分アルゴリズム（“reconciliation” と呼ばれる）は、既存のサブツリーを更新すべきかそれを破棄して新しいものをマウントすべきかを決定する際に、コンポーネントの型が同一かどうかの情報を利用します。`render` メソッドから返されるコンポーネントが以前の描画から返されたコンポーネントと（`===` で検証して）同一だった場合、React はサブツリーを新しいツリーとの差分を取りながら再帰的に更新します。コンポーネントが同一でなければ、以前のサブツリーは完全にアンマウントされます。

通常このことを考慮する必要はありません。ですが HOC に関しては考えるべきことです。このことが、render メソッド中でコンポーネントに HOC を適用してはいけないということを意味しているからです：

```
render() {
  // A new version of EnhancedComponent is created on every render
  // EnhancedComponent1 !== EnhancedComponent2
  const EnhancedComponent = enhance(MyComponent);
  // That causes the entire subtree to unmount/remount each time!
  return <EnhancedComponent />;
}
```

ここでの問題はパフォーマンスだけではありません。コンポーネントの再マウントによりコンポーネントとその子要素全ての state が失われるのです。
こうするのではなく、結果としてのコンポーネントが 1 回だけつくられるようにするため、コンポーネント定義の外で HOC を適用してください。そうすれば、レンダー間でその同一性が保たれるようになるでしょう。何にせよ、通常の場合これが望ましい実装になります。
HOC を動的に適用する必要があるような稀なケースでも、コンポーネントのライフサイクルメソッドやコンストラクタの中で行うようにしましょう。

静的メソッドは必ずコピーすること
React のコンポーネントで静的メソッドを定義することは便利であることがあります。例えば、Relay のコンテナは GraphQL fragment のコンポジションを容易に実現するため、`getFragment` という静的メソッドを公開しています。
しかし、HOC をコンポーネントに適用すると、元のコンポーネントはコンテナコンポーネントにラップされます。つまり新しいコンポーネントは元のコンポーネントの静的メソッドを 1 つも持っていないということになってしまいます。

```
// Define a static method
WrappedComponent.staticMethod = function() {/*...*/}
// Now apply a HOC
const EnhancedComponent = enhance(WrappedComponent);

// The enhanced component has no static method
typeof EnhancedComponent.staticMethod === 'undefined' // true
```

この問題を解決するために、コンテナコンポーネントを返す前にメソッドをコピーすることができます。

```
function enhance(WrappedComponent) {
  class Enhance extends React.Component {/*...*/}
  // Must know exactly which method(s) to copy :(
  Enhance.staticMethod = WrappedComponent.staticMethod;
  return Enhance;
}
```

しかし、この方法ではどのメソッドがコピーされる必要があるのか正確に知っておく必要があります。hoist-non-react-statics を使用することで、全ての非 React の静的メソッドを自動的にコピーできます：

```
import hoistNonReactStatic from 'hoist-non-react-statics';
function enhance(WrappedComponent) {
  class Enhance extends React.Component {/*...*/}
  hoistNonReactStatic(Enhance, WrappedComponent);
  return Enhance;
}
```

もう 1 つの解決策となりうる方法はコンポーネント自身とは分離して静的メソッドをエクスポートすることです。

```
// Instead of...
MyComponent.someFunction = someFunction;
export default MyComponent;

// ...export the method separately...
export { someFunction };

// ...and in the consuming module, import both
import MyComponent, { someFunction } from './MyComponent.js';
```

ref 属性は渡されない
高階コンポーネントの通例としては、すべての props はラップされたコンポーネントに渡されますが、ref に関してはそうではありません。これは `ref` 属性が（`key` と同様）実際のプロパティではなく、React によって特別に処理されているものだからです。HOC から出力されたコンポーネントの要素に ref 属性を追加する場合、ref 属性はラップされた内側のコンポーネントではなく、最も外側のコンテナコンポーネントを参照します。
この問題の解決方法は（React 16.3 で導入された）`React.forwardRef` API を使うことです。詳しくは ref のフォワーディングの章をご覧ください。

このページを編集する