レンダープロップ

“レンダープロップ (render prop)”という用語は、値が関数である props を使って、コンポーネント間でコードを共有するためのテクニックを指します。
レンダープロップを持つコンポーネントは、自身のレンダーロジックを実装する代わりに、React 要素を返す関数を受け取ってそれを呼び出します。

```
<DataProvider render={data => (
  <h1>Hello {data.target}</h1>
)}/>
```

レンダープロップを用いたライブラリとしては、React Router や Downshift などがあります。
このドキュメントでは、レンダープロップが役立つ理由と、その実装手順について解説します。

横断的関心事にレンダープロップを使う

コンポーネントは、React でコードを再利用するための主な要素ですが、あるコンポーネントがカプセル化した state や振る舞いを、同じ state を必要とする別のコンポーネントと共有する方法については、あまり自明ではありません。
たとえば、以下のコンポーネントは、ウェブアプリケーション内でのマウスの位置を追跡します。

```
class MouseTracker extends React.Component {
  constructor(props) {
    super(props);
    this.handleMouseMove = this.handleMouseMove.bind(this);
    this.state = { x: 0, y: 0 };
  }

  handleMouseMove(event) {
    this.setState({
      x: event.clientX,
      y: event.clientY
    });
  }

  render() {
    return (
      <div style={{ height: '100%' }} onMouseMove={this.handleMouseMove}>
        <h1>Move the mouse around!</h1>
        <p>The current mouse position is ({this.state.x}, {this.state.y})</p>
      </div>
    );
  }
}
```

画面上でカーソルが移動すると、コンポーネントはその (x, y) 座標を <p> 内に表示します。
ここで疑問となるのは、この振る舞いを他のコンポーネントで再利用する方法です。つまり、他のコンポーネントもカーソルの位置を知る必要がある時、この振る舞いだけをカプセル化し、そのコンポーネントと簡単に共有することは可能でしょうか？
コンポーネントは React でコードを再利用するための基本要素ですので、コードを少しリファクタリングし、他の場所で再利用したい振る舞いをカプセル化した <Mouse> というコンポーネントを作って、それを使うようにしてみましょう。

```
// The <Mouse> component encapsulates the behavior we need...
class Mouse extends React.Component {
  constructor(props) {
    super(props);
    this.handleMouseMove = this.handleMouseMove.bind(this);
    this.state = { x: 0, y: 0 };
  }

  handleMouseMove(event) {
    this.setState({
      x: event.clientX,
      y: event.clientY
    });
  }

  render() {
    return (
      <div style={{ height: '100%' }} onMouseMove={this.handleMouseMove}>

        {/* ...but how do we render something other than a <p>? */}
        <p>The current mouse position is ({this.state.x}, {this.state.y})</p>
      </div>
    );
  }
```

```
}

class MouseTracker extends React.Component {
  render() {
    return (
      <div>
        <h1>Move the mouse around!</h1>
        <Mouse />
      </div>
    );
  }
}
```

これで <Mouse> コンポーネントは、 mousemove イベントとカーソルの (x, y) 座標に紐付けられた全ての振る舞いをカプセル化していますが、まだ再利用可能と言うには不十分です。
たとえば、画面の中でマウスを追いかける猫の画像をレンダーする <Cat> コンポーネントがあるとしましょう。<Cat mouse={{ x, y }}> props を使って、このコンポーネントにマウスの座標を受け渡し、画面上のどこに猫の画像を配置すれば良いかを知らせたいでしょう。
手始めに、**<Mouse> の render メソッド内** で、以下のように <Cat> をレンダーしてみましょう。

```
class Cat extends React.Component {
  render() {
    const mouse = this.props.mouse;
    return (
      <img src="/cat.jpg" style={{ position: 'absolute', left: mouse.x, top: mouse.y }} />
    );
  }
}

class MouseWithCat extends React.Component {
  constructor(props) {
    super(props);
    this.handleMouseMove = this.handleMouseMove.bind(this);
    this.state = { x: 0, y: 0 };
  }

  handleMouseMove(event) {
    this.setState({
      x: event.clientX,
      y: event.clientY
    });
  }

  render() {
    return (
      <div style={{ height: '100%' }} onMouseMove={this.handleMouseMove}>

        {/*
          We could just swap out the <p> for a <Cat> here ... but then
          we would need to create a separate <MouseWithSomethingElse>
          component every time we need to use it, so <MouseWithCat>
          isn't really reusable yet.
        */}
        <Cat mouse={this.state} />
      </div>
    );
  }
}

class MouseTracker extends React.Component {
  render() {
    return (
      <div>
        <h1>Move the mouse around!</h1>
        <MouseWithCat />
      </div>
    );
  }
}
```

これだけが目的であれば正しく動作しますが、再利用可能な方法でこの振る舞いをカプセル化するという目的はまだ果たせていません。他の異なるユースケースでもマウスの位置を知りたい場合、毎回そのユースケースに沿ったものをレンダーする新しいコンポーネント（つまり、本質的に別の <MouseWithCat>）を作成する必要があります。
ここでレンダープロップの出番となります。<Mouse> コンポーネント内でハードコードされた <Cat> でレンダーの出力を変更する代わりに、 <Mouse> コンポーネントに関数の props を渡し、コンポーネントはその関数を使って何をレンダーすべきか動的に知るのです。これがレンダープロップの役割です。

```
class Cat extends React.Component {
  render() {
    const mouse = this.props.mouse;
    return (
      <img src="/cat.jpg" style={{ position: 'absolute', left: mouse.x, top: mouse.y }} />
    );
  }
}

class Mouse extends React.Component {
  constructor(props) {
    super(props);
    this.handleMouseMove = this.handleMouseMove.bind(this);
    this.state = { x: 0, y: 0 };
  }
```

```
  handleMouseMove(event) {
    this.setState({
      x: event.clientX,
      y: event.clientY
    });
  }

  render() {
    return (
      <div style={{ height: '100%' }} onMouseMove={this.handleMouseMove}>

        {/*
          Instead of providing a static representation of what <Mouse> renders,
          use the `render` prop to dynamically determine what to render.
        */}
        {this.props.render(this.state)}
      </div>
    );
  }
}

class MouseTracker extends React.Component {
  render() {
    return (
      <div>
        <h1>Move the mouse around!</h1>
        <Mouse render={mouse => (
          <Cat mouse={mouse} />
        )}/>
      </div>
    );
  }
}
```

ここでは、特定のユースケースを解決するために `<Mouse>` コンポーネントを複製して `render` メソッドに他のものをハードコードする代わりに、`<Mouse>` に `render` プロパティを渡して、何をレンダーするか動的に決定できるようにしています。
より具体的に述べると、**レンダープロップとは、あるコンポーネントが何をレンダーすべきかを知るために使う関数の props です。**
このテクニックによって、再利用する必要がある振る舞いの移植性が極めて高くなります。その振る舞いをさせるためには、現在のカーソルの (x, y) 座標にレンダーするものを示す `render` プロパティを使って `<Mouse>` をレンダーすれば良いのです。
レンダープロップの興味深い点として、多くの高階コンポーネント (HOC) がレンダープロップを使った通常のコンポーネントによって実装可能ということが挙げられます。たとえば、`<Mouse>` コンポーネントよりも `withMouse` HOC が好みであれば、レンダープロップを持つ `<Mouse>` を使って簡単に作成可能です。

```
// If you really want a HOC for some reason, you can easily
// create one using a regular component with a render prop!
function withMouse(Component) {
  return class extends React.Component {
    render() {
      return (
        <Mouse render={mouse => (
          <Component {...this.props} mouse={mouse} />
        )}/>
      );
    }
  }
}
```

つまり、レンダープロップによってどちらのパターンも可能になります。

---

render 以外の props を使う

このパターンが「レンダープロップ」という名前だからといって、必ずしも **render という名前の props を使う必要はない**ということを念頭に置いてください。実際、コンポーネントがレンダーするものを知るために使う関数の props は、**その名前が何であれ**、技術的には「レンダープロップ」と呼ぶことができます。
上記の例では `rendrer` を用いていますが、`children` プロパティを使っても同じくらい簡単です！

```
<Mouse children={mouse => (
  <p>The mouse position is {mouse.x}, {mouse.y}</p>
)}/>
```

さらに、`children` プロパティは実際には JSX 要素の「属性」の一覧内で名前を付ける必要がないことも忘れないでください。代わりに、要素**内部に**直接設定可能です！

```
<Mouse>
  {mouse => (
    <p>The mouse position is {mouse.x}, {mouse.y}</p>
  )}
</Mouse>
```

このテクニックは、react-motion の API などで使用されています。
このテクニックは若干珍しいため、このような API 設計時には、`children` が関数であることを `propTypes` で明示した方が良いでしょう。

```
Mouse.propTypes = {
  children: PropTypes.func.isRequired
};
```

注意事項

レンダープロップを React.PureComponent で使うときの注意点

レンダープロップを使う際、`render` メソッド内で関数を作成していると、`React.PureComponent` を使う利点が相殺されます。これは props の浅い (shallow) 比較は新しい props の値に対して常に `false` を返し、そして `render` は毎回レンダープロップとして新しい値を生成するためです。
たとえば、上記の `<Mouse>` コンポーネントの場合、`Mouse` が `React.Component` ではなく `React.PureComponent` を継承していたとすると、次のようになります。

```
class Mouse extends React.PureComponent {
  // Same implementation as above...
}

class MouseTracker extends React.Component {
  render() {
    return (
      <div>
        <h1>Move the mouse around!</h1>

        {/*
          This is bad! The value of the `render` prop will
          be different on each render.
        */}
        <Mouse render={mouse => (
          <Cat mouse={mouse} />
        )}/>
      </div>
    );
  }
}
```

この例では、`<MouseTracker>` がレンダーされるたびに `<Mouse render>` プロパティの値として新しい関数が生成されるので、`<Mouse>` が `React.PureComonent` を継承している効果がそもそもなくなってしまいます！
この問題を回避するため、レンダープロップをインスタンスメソッドとして次のように定義することもできます。

```
class MouseTracker extends React.Component {
  // Defined as an instance method, `this.renderTheCat` always
  // refers to *same* function when we use it in render
  renderTheCat(mouse) {
    return <Cat mouse={mouse} />;
  }

  render() {
    return (
      <div>
        <h1>Move the mouse around!</h1>
        <Mouse render={this.renderTheCat} />
      </div>
    );
  }
}
```

props を静的に定義できない場合（たとえば、コンポーネントの props や state をクロージャで囲む場合など）、`<Mouse>` は代わりに `React.Component` を継承すべきです。

このページを編集する