state のリフトアップ

しばしば、いくつかのコンポーネントが同一の変化するデータを反映する必要がある場合があります。そんなときは直近の共通祖先コンポーネントに共有されている状態をリフトアップすることを推奨します。これを、実際にはどのように行うかを見てみましょう。

この章では、与えられた温度で水が沸騰するかどうかを計算する温度計算ソフトを作成します。

`BoilingVerdict` というコンポーネントから始めましょう。これは温度を `celsius` という props として受け取り、水が沸騰するのに十分な温度かどうかを表示します。

```
function BoilingVerdict(props) {
  if (props.celsius >= 100) {
    return <p>The water would boil.</p>;
  }
  return <p>The water would not boil.</p>;
}
```

次に `Calculator` と呼ばれるコンポーネントを作成します。温度を入力するための `<input>` 要素をレンダーし、入力された値を `this.state.temperature` に保持します。

加えて、現在の入力値を判定する `BoilingVerdict` もレンダーします。

```
class Calculator extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {temperature: ''};
  }

  handleChange(e) {
    this.setState({temperature: e.target.value});
  }

  render() {
    const temperature = this.state.temperature;
    return (
      <fieldset>
        <legend>Enter temperature in Celsius:</legend>
        <input
          value={temperature}
          onChange={this.handleChange} />

        <BoilingVerdict
          celsius={parseFloat(temperature)} />

      </fieldset>
    );
  }
}
```

Try it on CodePen

---

2 つ目の入力を追加する

新しい要件は、摂氏の入力に加えて、華氏の入力もできるようにして、それらを同期させておくことです。

`Calculator` から `TemperatureInput` コンポーネントを抽出するところから始めましょう。props として、`"c"` もしくは `"f"` の値をとる `scale` を新しく追加します：

```
const scaleNames = {
  c: 'Celsius',
  f: 'Fahrenheit'
};

class TemperatureInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {temperature: ''};
  }

  handleChange(e) {
    this.setState({temperature: e.target.value});
  }

  render() {
    const temperature = this.state.temperature;
    const scale = this.props.scale;
    return (
      <fieldset>
```

```
        <legend>Enter temperature in {scaleNames[scale]}:</legend>
        <input value={temperature}
               onChange={this.handleChange} />
      </fieldset>
    );
  }
}
```

これで Calculator を 2 つの別個の温度入力フィールドをレンダーするように変更することができます：

```
class Calculator extends React.Component {
  render() {
    return (
      <div>
        <TemperatureInput scale="c" />
        <TemperatureInput scale="f" />
      </div>
    );
  }
}
```

Try it on CodePen

2 つの入力フィールドが用意できました。しかし、片方に温度を入力しても、もう片方は更新されません。これは要件を満たしていません： 2 つの入力フィールドを同期させたいのです。
Calculator から BoilingVerdict を表示することもできません。Calculator は TemperatureInput の中に隠されている現在の温度を知らないのです。

変換関数の作成

まず、摂氏から華氏に変換するものとその反対のものと、2 つの関数を書きます。

```
function toCelsius(fahrenheit) {
  return (fahrenheit - 32) * 5 / 9;
}

function toFahrenheit(celsius) {
  return (celsius * 9 / 5) + 32;
}
```

これら 2 つの関数は数字を変換します。次に文字列で表現された temperature と変換関数を引数に取り文字列を返す、別の関数を作成します。この関数を一方の入力の値をもう一方の入力に基づいて計算するのに使用します。
常に値が小数第 3 位までで四捨五入されるようにし、無効な temperature には空の文字列を返します。

```
function tryConvert(temperature, convert) {
  const input = parseFloat(temperature);
  if (Number.isNaN(input)) {
    return '';
  }
  const output = convert(input);
  const rounded = Math.round(output * 1000) / 1000;
  return rounded.toString();
}
```

例えば、tryConvert('abc', toCelsius) は空の文字列を返し、tryConvert('10.22', toFahrenheit) は '50.396' を返します。

---

state のリフトアップ

現時点では、両方の TemperatureInput コンポーネントは独立してローカルの state を保持しています：

```
class TemperatureInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {temperature: ''};
  }

  handleChange(e) {
    this.setState({temperature: e.target.value});
  }

  render() {
    const temperature = this.state.temperature;
    // ...
```

しかし、2 つの入力フィールドはお互いに同期されていて欲しいです。摂氏の入力フィールドを更新したら、華氏の入力フィールドも華氏に変換された温度で反映されて欲しいですし、逆も同じです。
React での state の共有は、state を、それを必要とするコンポーネントすべての直近の共通祖先コンポーネントに移動することによって実現します。これを “state のリフトアップ (lifting state up)” と呼びます。TemperatureInput からローカルの state を削除して Calculator に移動しましょう。

`Calculator` が共有の state を保持すれば、それが両方の入力における現在の温度の “信頼できる情報源 (source of truth)” となります。それによって、両方に対して相互に一貫性のある値を持たせることができるようになります。両方の `TemperatureInput` コンポーネントの props は同じ親コンポーネント `Calculator` から与えられるので、2 つの入力は常に同期されているようになります。
それでは、どのように動作するのかひとつずつ見ていきましょう。
まず、`TemperatureInput` コンポーネントの `this.state.temperature` を `this.props.temperature` に置き換えます。とりあえず、`this.props.temperature` は既にあるものだとしておきましょう。後でこれは `Calculator` から渡すようにします：

```
  render() {
    // Before: const temperature = this.state.temperature;
    const temperature = this.props.temperature;
    // ...
```

props が読み取り専用であることは周知の通りです。`temperature` がローカルの state に格納されている間は、`TemperatureInput` は `this.setState()` を呼び出すだけでそれを変更することができました。しかし今や、`temperature` は親コンポーネントから与えられる props の一部ですから、`TemperatureInput` はそれを制御できません。
通常 React では、コンポーネントを “制御された (controlled)” ものとすることでこの問題を解決します。DOM である `<input>` が `value` と `onChange` プロパティの両方を受け取るように、カスタムコンポーネントの `TemperatureInput` は `temperature` と `onTemperatureChange` の両方を親コンポーネントの `Calculator` から受け取ることができます。
ここで、`TemperatureInput` が自身の温度を更新したい場合、`this.props.onTemperatureChange` を呼び出します：

```
  handleChange(e) {
    // Before: this.setState({temperature: e.target.value});
    this.props.onTemperatureChange(e.target.value);
    // ...
```

補足：
カスタムコンポーネントの `temperature` や `onTemperatureChange` といった props の名前に特別な意味があるわけではありません。慣習に則り `value` や `onChange` など、他の任意の名前を使うこともできます。

`onTemperatureChange` プロパティは親コンポーネント `Calculator` から `temperature` プロパティと共に渡されます。親コンポーネントは入力の変化に応じて自身のローカル state を更新し、結果的に両方の入力フォームは新しい値で再レンダーされます。`Calculator` をどう実装するかはこの後すぐに見ていきましょう。
`Calculator` の変更点を見ていく前に、`TemperatureInput` コンポーネントで行った変更をおさらいしましょう。ローカルの state を削除し、`this.state.temperature` の代わりに `this.props.temperature` を読み取るようにしました。また、変更を加えたい場合は `this.setState()` を呼び出す代わりに `Calculator` から与えられる `this.props.onTemperatureChange()` を呼び出すことにしました：

```
class TemperatureInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
  }

  handleChange(e) {
    this.props.onTemperatureChange(e.target.value);
  }

  render() {
    const temperature = this.props.temperature;
    const scale = this.props.scale;
    return (
      <fieldset>
        <legend>Enter temperature in {scaleNames[scale]}:</legend>
        <input value={temperature}
               onChange={this.handleChange} />
      </fieldset>
    );
  }
}
```

では `Calculator` コンポーネントの番です。
現時点での入力の `temperature` と `scale` を、このコンポーネントのローカルな state に保存することにします。これは入力コンポーネントから “リフトアップ” したものであり、両方にとっての “信頼出来る情報源” として振る舞うことになります。これは、両方の入力コンポーネントをレンダリングするために必要となる最小のデータの形です。
例えば、摂氏側の入力に 37 と打ち込むと、`Calculator` コンポーネントの state は以下のようになります：

```
{
  temperature: '37',
  scale: 'c'
}
```

その後に華氏の入力フィールドを 212 に変更すると、`Calculator` の state は以下のようになります：

```
{
  temperature: '212',
  scale: 'f'
}
```

両方の入力を保存することもできましたが、それは不必要だと分かります。最後に変更された値とそれが示す単位を保存すれば十分なのです。現時点での `temperature` と `scale` の 2 つさえあれば、もう一方の値は推測することができます。
同じ state から値が算出されるので、2 つの入力コンポーネントは常に同期します。

```
class Calculator extends React.Component {
  constructor(props) {
    super(props);
    this.handleCelsiusChange = this.handleCelsiusChange.bind(this);
    this.handleFahrenheitChange = this.handleFahrenheitChange.bind(this);
    this.state = {temperature: '', scale: 'c'};
  }

  handleCelsiusChange(temperature) {
    this.setState({scale: 'c', temperature});
  }

  handleFahrenheitChange(temperature) {
    this.setState({scale: 'f', temperature});
  }

  render() {
    const scale = this.state.scale;
    const temperature = this.state.temperature;
    const celsius = scale === 'f' ? tryConvert(temperature, toCelsius) : temperature;
    const fahrenheit = scale === 'c' ? tryConvert(temperature, toFahrenheit) : temperature;

    return (
      <div>
        <TemperatureInput
          scale="c"
          temperature={celsius}
          onTemperatureChange={this.handleCelsiusChange} />

        <TemperatureInput
          scale="f"
          temperature={fahrenheit}
          onTemperatureChange={this.handleFahrenheitChange} />

        <BoilingVerdict
          celsius={parseFloat(celsius)} />

      </div>
    );
  }
}
```

Try it on CodePen
これで、どちらの入力コンポーネントを編集したかに関わらず、`Calculator` の `this.state.temperature` と `this.state.scale` が更新されます。片方の入力コンポーネントはあらゆるユーザからの入力が保持されるよう値をそのまま受け取り、もう片方の入力コンポーネントの値はそれに基づいて常に再計算されます。
入力値を変更した際に何が起こるのかをおさらいしましょう：

- React は DOM の `<input>` で `onChange` として指定された関数を呼び出します。この章の場合、`TemperatureInput` の `handleChange` メソッドが呼び出される関数になります。
- `TemperatureInput` の `handleChange` メソッドは `this.props.onTemperatureChange()` に新しい値を与えて呼び出します。`onTemperatureChange` を含む props は親コンポーネントである `Calculator` から与えられます。
- 前回のレンダリング時に、`Calculator` は摂氏の `TemperatureInput` の `onTemperatureChange` には自身の `handleCelsiusChange` メソッドを指定し、華氏の `TemperatureInput` の `onTemperatureChange` には自身の `handleFahrenheitChange` を指定していたのでした。そのため、どちらの入力フィールドを編集したかによって、2 つの `Calculator` メソッドのどちらが呼び出されるかが決まります。
- これらのメソッド内では、`Calculator` コンポーネントが新しい入力値と更新した方の入力値の単位を `this.setState()` に与えて呼び出して、React に `Calculator` コンポーネント自身を再レンダリングさせます。
- React は `Calculator` コンポーネントの `render` メソッドを呼び出して、UI がどのような見た目になるべきかを学びます。両方の入力コンポーネントの値が、現在の温度とアクティブな単位に基づいて再計算されます。温度の変換処理はここで行われます。
- React は `Calculator` により与えられた新しい props で各 `TemperatureInput` の `render` メソッドを呼び出します。React はそれらの UI がどのような見た目になるかを学びます。
- React は props として摂氏温度を与えて、`BoilingVerdict` コンポーネントの `render` メソッドを呼び出します。
- React DOM は沸騰したかどうかの判定結果と入力コンポーネントの値によって、DOM を更新します。変更された入力コンポーネントは現在の値によって、もう一方の入力コンポーネントは変換された温度によって更新されます。

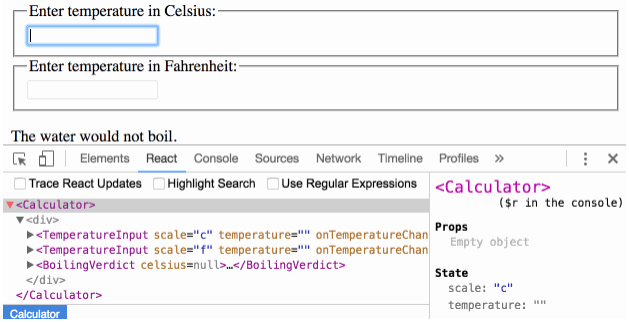
全ての更新は同じ手順で実行されるので、2 つの入力コンポーネントは常に同期を保つことができます。

この章で学んだこと

React アプリケーションで変化するどのようなデータも単一の “信頼出来る情報源” であるべきです。通常、state はレンダリング時にそれを必要とするコンポーネントに最初に追加されます。それから、他のコンポーネントもその state を必要としているなら、直近の共通祖先コンポーネントにその state をリフトアップすることができます。異なるコンポーネント間で state を同期しようとする代わりに、トップダウン型のデータフローの力を借りるべきです。
state のリフトアップは双方向のバインディング (two-way binding) を行う方法より多くの “ボイラープレート” コードを生み出しますが、その効果としてバグを発見して切り出す作業が少なく済むようになります。あらゆる state はいずれかのコンポーネント内に存在し、そのコンポーネントのみがその state を変更できるので、バグが潜む範囲は大幅に削減されます。加えて、ユーザー入力を拒否したり変換したりする任意の独自ロジックを実装することもできます。

props もしくは state から作りだす事のできるデータについては、おそらく state に保持すべきではないでしょう。例えば、今回は `celsiusValue` と `fahrenheitValue` の両方を保存する代わりに、最後に変更された `temperature` と、その値の `scale` のみを保存しています。もう一方の入力の値は常に `render()` メソッド内で計算することができます。これにより元のユーザ入力の精度を全く損なうことなくもう一方の入力フィールドに丸めを適用したり、もう一方の入力フィールドをクリアしたりできます。

UI で何かおかしな箇所があれば、React Developer Tools を使用して props を調査したり state の更新について責任を持っているコンポーネントに辿り着くまでツリーをさかのぼることができます。これによりバグをその原因まで追いかけることができます。



このページを編集する