state とライフサイクル

このページでは React コンポーネントにおける state とライフサイクルについての導入を行います。詳細なコンポーネントの API リファレンスはこちらにあります。
以前の章のひとつにあった秒刻みの時計の例を考えてみましょう。要素のレンダーの章にて、UI を更新するための方法をひとつだけ学びました。それはレンダーされた出力を更新するために `ReactDOM.render()` を呼び出す、というものでした。

```
function tick() {
  const element = (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {new Date().toLocaleTimeString()}.</h2>
    </div>
  );
  ReactDOM.render(
    element,
    document.getElementById('root')
  );
}

setInterval(tick, 1000);
```

Try it on CodePen
このセクションでは、この `Clock` コンポーネントを真に再利用可能かつカプセル化されたものにする方法を学びます。コンポーネントが自分でタイマーをセットアップし、自身を毎秒更新するようにします。
時計の見た目をカプセル化するところから始めてみましょう：

```
function Clock(props) {
  return (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {props.date.toLocaleTimeString()}.</h2>
    </div>
  );
}

function tick() {
  ReactDOM.render(
    <Clock date={new Date()} />,
    document.getElementById('root')
  );
}

setInterval(tick, 1000);
```

Try it on CodePen
しかし上記のコードは重要な要件を満たしていません：`Clock` がタイマーを設定して UI を毎秒ごとに更新するという処理は、`Clock` の内部実装の詳細 (implementation detail) であるべきだということです。
理想的には以下のコードを一度だけ記述して、`Clock` に自身を更新させたいのです：

```
ReactDOM.render(
  <Clock />,
  document.getElementById('root')
);
```

これを実装するには、`Clock` コンポーネントに “ステート (state)” を追加する必要があります。
state は props に似ていますが、コンポーネントによって完全に管理されるプライベートなものです。
以前に述べたように、クラスとして定義されたコンポーネントにはいくつか追加の機能があります。ローカルな state がまさにそれです：クラスでのみ使用できる機能です。

関数をクラスに変換する

以下の 5 ステップで、`Clock` のような関数コンポーネントをクラスに変換することができます。

1. `React.Component` を継承する同名の ES6 クラスを作成する。
2. `render()` と呼ばれる空のメソッドを 1 つ追加する。
3. 関数の中身を `render()` メソッドに移動する。
4. `render()` 内の `props` を `this.props` に書き換える。

5. 空になった関数の宣言部分を削除する。

```
class Clock extends React.Component {
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.props.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

Try it on CodePen

これでもう、`Clock` は関数ではなくクラスとして定義されています。

`render` メソッドは更新が発生した際に毎回呼ばれますが、同一の DOM ノード内で `<Clock />` をレンダーしている限り、`Clock` クラスのインスタンスは 1 つだけ使われます。このことにより、ローカル state やライフサイクルメソッドといった追加の機能が利用できるようになります。

クラスにローカルな state を追加する

以下の 3 ステップで `date` を props から state に移します：

1. `render()` メソッド内の `this.props.date` を `this.state.date` に書き換える：

```
class Clock extends React.Component {
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

2. `this.state` の初期状態を設定するクラスコンストラクタを追加する：

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

親クラスのコンストラクタへの `props` の渡し方に注目してください：

```
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }
```

クラスのコンポーネントは常に `props` を引数として親クラスのコンストラクタを呼び出す必要があります。

3. `<Clock />` 要素から `date` プロパティを削除する：

```
ReactDOM.render(
  <Clock />,
  document.getElementById('root')
);
```

タイマーのコードはコンポーネント自身に後で追加しなおします。

結果は以下のようになります：

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
```

```
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

ReactDOM.render(
  <Clock />,
  document.getElementById('root')
);
```

Try it on CodePen

次に、`Clock` が自分でタイマーを設定し、毎秒ごとに自分を更新するようにします。

クラスにライフサイクルメソッドを追加する

多くのコンポーネントを有するアプリケーションでは、コンポーネントが破棄された場合にそのコンポーネントが占有していたリソースを開放することがとても重要です。

タイマーを設定したいのは、最初に `Clock` が DOM として描画されるときです。このことを React では “マウント (mounting)” と呼びます。

またタイマーをクリアしたいのは、`Clock` が生成した DOM が削除されるときです。このことを React では “アンマウント (unmounting)” と呼びます。

コンポーネントクラスで特別なメソッドを宣言することで、コンポーネントがマウントしたりアンマウントしたりした際にコードを実行することができます：

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  componentDidMount() {

  }

  componentWillUnmount() {

  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

これらのメソッドは “ライフサイクルメソッド (lifecycle method)” と呼ばれます。

`componentDidMount()` メソッドは、出力が DOM にレンダーされた後に実行されます。ここがタイマーをセットアップするのによい場所です：

```
  componentDidMount() {
    this.timerID = setInterval(
      () => this.tick(),
      1000
    );
  }
```

タイマー ID を直接 `this` 上に格納したことに注目してください。

`this.props` は React 自体によって設定され、また `this.state` は特別な意味を持っていますが、何かデータフローに影響しないデータ（タイマー ID のようなもの）を保存したい場合に、追加のフィールドを手動でクラスに追加することは自由です。

タイマーの後片付けは `componentWillUnmount()` というライフサイクルメソッドで行います：

```
  componentWillUnmount() {
    clearInterval(this.timerID);
  }
```

最後に、`Clock` コンポーネントが毎秒ごとに実行する `tick()` メソッドを実装します。

コンポーネントのローカル state の更新をスケジュールするために `this.setState()` を使用します：

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }
```

```
  componentDidMount() {
    this.timerID = setInterval(
      () => this.tick(),
      1000
    );
  }

  componentWillUnmount() {
    clearInterval(this.timerID);
  }

  tick() {
    this.setState({
      date: new Date()
    });
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

ReactDOM.render(
  <Clock />,
  document.getElementById('root')
);
```

Try it on CodePen

これで、この時計は毎秒ごとに時間を刻みます。

何が起こったのかをメソッドが呼び出される順序にそって簡単に振り返ってみましょう：

1. `<Clock />` が `ReactDOM.render()` に渡されると、React は `Clock` コンポーネントのコンストラクタを呼び出します。`Clock` は現在時刻を表示する必要があるので、現在時刻を含んだオブジェクトで `this.state` を初期化します。あとでこの state を更新していきます。
2. 次に React は `Clock` コンポーネントの `render()` メソッドを呼び出します。これにより React は画面に何を表示すべきか知ります。そののちに、React は DOM を `Clock` のレンダー出力と一致するように更新します。
3. `Clock` の出力が DOM に挿入されると、React は `componentDidMount()` ライフサイクルメソッドを呼び出します。その中で、`Clock` コンポーネントは毎秒ごとにコンポーネントの `tick()` メソッドを呼び出すためにタイマーを設定するようブラウザに要求します。
4. ブラウザは、毎秒ごとに `tick()` メソッドを呼び出します。その中で `Clock` コンポーネントは、現在時刻を含んだオブジェクトを引数として `setState()` を呼び出すことで、UI の更新をスケジュールします。`setState()` が呼び出されたおかげで、React は state が変わったということが分かるので、`render()` メソッドを再度呼び出して、画面上に何を表示すべきかを知ります。今回は、`render()` メソッド内の `this.state.date` が異なっているので、レンダリングされる出力には新しく更新された時間が含まれています。それに従って React は DOM を更新します。
5. この後に `Clock` コンポーネントが DOM から削除されることがあれば、React は `componentWillUnmount()` ライフサイクルメソッドを呼び出し、これによりタイマーが停止します。

---

state を正しく使用する

`setState()` について知っておくべきことが 3 つあります。

state を直接変更しないこと

例えば、以下のコードではコンポーネントは再レンダーされません：

```
// Wrong
this.state.comment = 'Hello';
```

代わりに `setState()` を使用してください：

```
// Correct
this.setState({comment: 'Hello'});
```

`this.state` に直接代入してよい唯一の場所はコンストラクタです。

state の更新は非同期に行われる可能性がある

React はパフォーマンスのために、複数の `setState()` 呼び出しを 1 度の更新にまとめて処理することがあります。

`this.props` と `this.state` は非同期に更新されるため、次の state を求める際に、それらの値に依存するべきではありません。

例えば、以下のコードはカウンターの更新に失敗することがあります：

```
// Wrong
this.setState({
  counter: this.state.counter + this.props.increment,
});
```

これを修正するために、オブジェクトではなく関数を受け取る `setState()` の 2 つ目の形を使用します。その関数は前の state を最初の引数として受け取り、更新が適用される時点での props を第 2 引数として受け取ります：

```
// Correct
this.setState((state, props) => ({
  counter: state.counter + props.increment
}));
```

上記のコードではアロー関数を使いましたが、通常の関数でも動作します：

```
// Correct
this.setState(function(state, props) {
  return {
    counter: state.counter + props.increment
  };
});
```

state の更新はマージされる
`setState()` を呼び出した場合、React は与えられたオブジェクトを現在の state にマージします。
例えば、あなたの state はいくつかの独立した変数を含んでいるかもしれません：

```
  constructor(props) {
    super(props);
    this.state = {
      posts: [],
      comments: []
    };
  }
```

その場合、別々の `setState()` 呼び出しで、それらの変数を独立して更新することができます：

```
  componentDidMount() {
    fetchPosts().then(response => {
      this.setState({
        posts: response.posts
      });
    });

    fetchComments().then(response => {
      this.setState({
        comments: response.comments
      });
    });
  }
```

マージは浅く (shallow) 行われるので、`this.setState({comments})` は `this.state.posts` をそのまま残しますが、`this.state.comments` は完全に置き換えます。

データは下方向に伝わる

親コンポーネントであれ子コンポーネントであれ、特定の他のコンポーネントがステートフルかステートレスかを知ることはできませんし、特定のコンポーネントの定義が関数型かクラス型かを気にするべきではありません。
これが、state はローカルのものである、ないしはカプセル化されている、と言われる理由です。state を所有してセットするコンポーネント自身以外からはその state にアクセスすることができません。コンポーネントはその子コンポーネントに props として自身の state を渡してもかまいません。

```
<h2>It is {this.state.date.toLocaleTimeString()}.</h2>
```

ユーザ定義のコンポーネントでも動作します：

```
<FormattedDate date={this.state.date} />
```

`FormattedDate` コンポーネントは props 経由で `date` を受け取りますが、それが `Clock` の state から来たのか、`Clock` の props から来たのか、もしくは手書きされたものなのかは分かりません：

```
function FormattedDate(props) {
  return <h2>It is {props.date.toLocaleTimeString()}.</h2>;
}
```

Try it on CodePen

このデータフローは一般的には “トップダウン” もしくは “単一方向” データフローと呼ばれます。いかなる state も必ず特定のコンポーネントが所有し、state から生ずる全てのデータまたは UI は、ツリーでそれらの “下” にいるコンポーネントにのみ影響します。
コンポーネントツリーとは props が流れ落ちる滝なのだと想像すると、各コンポーネントの state とは任意の場所で合流してくる追加の水源であり、それらもまた下に流れ落ちていくものなのです。
全てのコンポーネントが本当に独立していることを示すのに、3 つの `<Clock>` をレンダリングする `App` コンポーネントを作成します：

```
function App() {
  return (
    <div>
      <Clock />
      <Clock />
      <Clock />
    </div>
  );
}

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

Try it on CodePen
各 `Clock` は独立してタイマーをセットし、独立して更新します。
React アプリケーションでは、コンポーネントがステートフルかステートレスかは、コンポーネントにおける内部実装の詳細 (implementation detail) とみなされ、それは時間と共に変化しうるものです。ステートレスなコンポーネントをステートフルなコンポーネントの中で使うことが可能であり、その逆も同様です。

このページを編集する