他のライブラリとのインテグレーション

React はどのような Web アプリケーションにも適用できます。React は他のアプリケーションに組み込むことができ、また少しだけ気を付ければ、React に他のアプリケーションを組み込むこともできます。ここでは一般的なユースケースである jQuery と Backbone を使った例を紹介しますが、同じ考え方はどのような既存のコードにも適用可能です。

DOM 操作プラグインとのインテグレーション

React は、React の外で DOM に加えられた変更を認識しません。React は自身の内部表現に基づいて更新を決定します。もし同じ DOM ノードが別のライブラリによって操作された場合、React は混乱してしまい、回復する方法がありません。
とはいえ、React と操作プラグインを組み合わせることが不可能、あるいは必ずしも難しいと言っているのではありません。それぞれがやっていることを正しく認識する必要があるのです。
コンフリクトを回避する最も簡単な方法は、React コンポーネントが更新されないようにすることです。これは、空の `<div />` のように、React から更新する理由がない要素をレンダーすることで実現できます。

この問題への取り組み方法

上記の方法を実証するために、一般的な jQuery プラグインのラッパーを作ってみましょう。
まず、ルート DOM 要素へ ref をアタッチします。`componentDidMount` を使い、ref へのリファレンスを取得し、jQuery プラグインに渡します。
マウントの後に React が DOM をいじってしまうことを防ぐため、`render()` メソッドからは空の `<div />` 要素を返すようにします。この空の `<div />` 要素はプロパティや子要素を持たないので、React はそれを更新する理由がなく、jQuery プラグインの側が DOM のその部分を自由に管理できるのです：

```
class SomePlugin extends React.Component {
  componentDidMount() {
    this.$el = $(this.el);
    this.$el.somePlugin();
  }

  componentWillUnmount() {
    this.$el.somePlugin('destroy');
  }

  render() {
    return <div ref={el => this.el = el} />;
  }
}
```

`componentDidMount` と `componentWillUnmount` の両方の ライフサイクルメソッド を定義したことに注意してください。多くの jQuery プラグインは DOM にイベントリスナーをアタッチするので、それらを `componentWillUnmount` でデタッチすることが重要です。もしプラグインがクリーンアップの方法を提供していない場合、あなた自身で提供する必要があります。メモリリークを防ぐためにプラグインが登録したイベントリスナを削除することを忘れないでください。

jQuery Chosen プラグインとのインテグレーション

このアイディアの具体的な例として、`<select>` 要素を拡張する Chosen プラグインの最小のラッパーを書いてみましょう。

> **Note:**
> それが可能だからといって、それが React アプリケーションに最適なアプローチであるという意味ではありません。可能であれば React コンポーネントを使用することをお勧めします。React コンポーネントは React アプリケーションで簡単に再利用でき、また多くの場合、それらの動作や外観をより細かくコントロールできます。

まず、Chosen が DOM に対してどういった操作をしているのか確認しましょう。
`<select>` DOM ノードで Chosen を呼ぶと元の DOM ノードから属性を読み取り、インライン CSS で隠してから、`<select>` の直後に独自の視覚表現を持つ別の DOM ノードを追加します。その後に jQuery イベントを発生させて変更を通知します。
我々の `<Chosen>` というラッパー React コンポーネントで作成したい API は以下のようなものであるとしましょう：

```
function Example() {
  return (
    <Chosen onChange={value => console.log(value)}>
      <option>vanilla</option>
      <option>chocolate</option>
      <option>strawberry</option>
    </Chosen>
  );
}
```

わかりやすくするために非制御コンポーネントとして実装します。
まず、`<select>` を `<div>` で囲んで返す `render()` メソッドを持った、空のコンポーネントを作成します：

```
class Chosen extends React.Component {
  render() {
    return (
      <div>
        <select className="Chosen-select" ref={el => this.el = el}>
          {this.props.children}
        </select>
      </div>
    );
  }
}
```

余分な で <select> をラップしていることに注目してください。これが必要なのは、Chosen は渡された <select> の直後に別の DOM 要素を追加するからです。しかし、React からみれば、この は常に 1 つの子要素しか持っていません。これにより React による更新が Chosen によって追加された DOM ノードと確実に競合しないようにできるのです。重要なことは、React フローの外側で DOM を変更する場合は、React がその DOM ノードに触る理由を確実になくす必要がある、ということです。
次に、ライフサイクルメソッドを実装しましょう。`componentDidMount` の中で ref 経由で渡された <select> ノードで Chosen を初期化する必要があります。そして `componentWillUnmount` でそれを破棄します：

```
componentDidMount() {
  this.$el = $(this.el);
  this.$el.chosen();
}

componentWillUnmount() {
  this.$el.chosen('destroy');
}
```

Try it on CodePen

React からすると `this.el` フィールドに特別な意味はありません。以前に `render()` メソッドの中で `ref` からこれに代入したことによって成り立っています：

```
<select className="Chosen-select" ref={el => this.el = el}>
```

コンポーネントをレンダーするにはこれで十分ですが、値の変更について通知を受ける必要もあります。この通知を実現するために、Chosen が管理する <select> で起こる jQuery の `change` イベントを受け取るようにします。
コンポーネントの props（およびその一部として渡されるイベントハンドラ）は時間の経過とともに変わってしまう可能性があるので、Chosen に直接 `this.props.onChange` を渡さないようにします。代わりに、`this.props.onChange` を呼び出す `handleChange()` メソッドを宣言し、そちらを jQuery の `change` イベントのコールバックに設定します：

```
componentDidMount() {
  this.$el = $(this.el);
  this.$el.chosen();

  this.handleChange = this.handleChange.bind(this);
  this.$el.on('change', this.handleChange);
}

componentWillUnmount() {
  this.$el.off('change', this.handleChange);
  this.$el.chosen('destroy');
}

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

Try it on CodePen

最後にもう 1 つ作業が残っています。React では props は時間と共に変更される可能性があります。例えば、親コンポーネントの state が変更されると、<Chosen> コンポーネントは異なる子を受け取るようになるかもしれません。つまり、React に DOM の管理を任せることはもうできないので、インテグレーションを行う部分では props の変更に応じて DOM を手動で更新することが重要です。
Chosen のドキュメントによると元の DOM 要素への変更について通知するための API として jQuery `trigger()` API が使えます。<select> 要素の中の `this.props.children` の更新に関しては React にやってもらいましょう。しかしまた、`componentDidUpdate()` のライフサイクルメソッドを追加し、Chosen に変更を通知する必要があります：

```
componentDidUpdate(prevProps) {
  if (prevProps.children !== this.props.children) {
    this.$el.trigger("chosen:updated");
  }
}
```

のようにして Chosen は、React が管理する <select> の子要素に変更があった場合に自分が管理する DOM 要素を変更すべき事が分かるようになります。
`Chosen` コンポーネントの完全な実装は以下のようになります：

```
class Chosen extends React.Component {
  componentDidMount() {
    this.$el = $(this.el);
    this.$el.chosen();

    this.handleChange = this.handleChange.bind(this);
    this.$el.on('change', this.handleChange);
  }
```

```
  componentDidUpdate(prevProps) {
    if (prevProps.children !== this.props.children) {
      this.$el.trigger("chosen:updated");
    }
  }

  componentWillUnmount() {
    this.$el.off('change', this.handleChange);
    this.$el.chosen('destroy');
  }

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

  render() {
    return (
      <div>
        <select className="Chosen-select" ref={el => this.el = el}>
          {this.props.children}
        </select>
      </div>
    );
  }
}
```

Try it on CodePen

他のビューライブラリとのインテグレーション

React は `ReactDOM.render()` の柔軟性のおかげで、他のアプリケーションに組み込むことができます。
React は一般的に起動時に単一のルート React コンポーネントを DOM にロードして使用されるものですが、`ReactDOM.render()` はボタンのような小さなものからアプリケーション全体に至るまで、独立した UI のパーツに対して複数回呼び出すこともできます。
実際、これはまさに React が Facebook で使用されている方法でもあります。これにより React でアプリケーションを少しずつ作成し、それらを既存のサーバー生成テンプレートやその他のクライアントサイドコードと組み合わせることができます。

React で文字列ベースのレンダーを置き換える

古いウェブアプリケーションによくあるパターンは、DOM のまとまりを文字列として記述して、`$el.html(htmlString)` のような形で DOM に挿入することです。このような箇所は React の導入にぴったりです。文字列ベースのレンダーを React コンポーネントに置き換えるだけで良いのです。
つまり、次のような jQuery による実装は

```
$('#container').html('<button id="btn">Say Hello</button>');
$('#btn').click(function() {
  alert('Hello!');
});
```

React コンポーネントを使用して次のように書き換えられます：

```
function Button() {
  return <button id="btn">Say Hello</button>;
}

ReactDOM.render(
  <Button />,
  document.getElementById('container'),
  function() {
    $('#btn').click(function() {
      alert('Hello!');
    });
  }
);
```

ここから始めて、コンポーネントにロジック部分を更に移植していくことや、より一般的な React のプラクティスを採用していくことができます。例えば、コンポーネントでは同じコンポーネントが複数回レンダーされる可能性があるので、ID に依存しないことがベストプラクティスです。かわりに React の React event system を使用してクリックハンドラを React の `<button>` 要素に直接登録します：

```
function Button(props) {
  return <button onClick={props.onClick}>Say Hello</button>;
}

function HelloButton() {
  function handleClick() {
    alert('Hello!');
  }
  return <Button onClick={handleClick} />;
}

ReactDOM.render(
  <HelloButton />,
```

```
  document.getElementById('container')
);
```

Try it on CodePen

このような分離されたコンポーネントを好きなだけ持つことができ、`ReactDOM.render()` を使用して異なる DOM コンテナにそれらをレンダーすることができます。アプリケーションを少しずつ React に変換していくにつれて、より大きなコンポーネントへとインテグレーションできるようになり、`ReactDOM.render()` の呼び出しを階層の上の方へ移動させていけるようになるでしょう。

Backbone View に React を組み込む

Backbone view は通常、HTML 文字列、もしくは文字列を生成するテンプレート用関数を使って、DOM 要素の中身を作成します。この処理もまた React コンポーネントのレンダリングに置き換えられます。

以下で、`ParagraphView` と呼ばれる Backbone view を作成します。Backbone の `render()` 関数をオーバーライドして、React の `<Paragraph>` コンポーネントを Backbone が提供する DOM 要素 (this.el) にレンダリングします。ここでも `ReactDOM.render()` を使用します：

```
function Paragraph(props) {
  return <p>{props.text}</p>;
}

const ParagraphView = Backbone.View.extend({
  render() {
    const text = this.model.get('text');
    ReactDOM.render(<Paragraph text={text} />, this.el);
    return this;
  },
  remove() {
    ReactDOM.unmountComponentAtNode(this.el);
    Backbone.View.prototype.remove.call(this);
  }
});
```

Try it on CodePen

`remove` メソッドで `ReactDOM.unmountComponentAtNode()` を呼び出して、コンポーネントツリーがデタッチされた際にイベントハンドラとコンポーネントツリーに関連付けられていたその他のリソースを React が解除することも重要です。

React **ツリー内**からコンポーネントが削除されるとクリーンアップは自動的に実行されますが、ツリー全体を手動で削除するため、このメソッドを呼び出す必要があります。

Model 層とのインテグレーション

一般的には React の state、Flux、もしくは Redux のような一方向のデータフローの使用が推奨されますが、React コンポーネントは他のフレームワークやライブラリのモデル層を利用することができます。

React コンポーネントで Backbone Model を使用する

React コンポーネントから Backbone のモデルとコレクションを利用する最もシンプルな方法は、様々な変更イベントを監視して手動で強制的に更新することです。

モデルのレンダーに責任をもつコンポーネントは `'change'` イベントを監視し、コレクションのレンダーに責任をもつコンポーネントは `'add'` および `'remove'` イベントを監視します。どちらの場合も、`this.forceUpdate()` を呼び出して新しいデータでコンポーネントを再レンダリングします。

以下の例では、`List` コンポーネントは Backbone のコレクションをレンダーします。個別の要素のレンダーには `Item` コンポーネントを使用します。

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

  handleChange() {
    this.forceUpdate();
  }

  componentDidMount() {
    this.props.model.on('change', this.handleChange);
  }

  componentWillUnmount() {
    this.props.model.off('change', this.handleChange);
  }

  render() {
    return <li>{this.props.model.get('text')}</li>;
  }
}

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

  handleChange() {
    this.forceUpdate();
  }
```

```
  componentDidMount() {
    this.props.collection.on('add', 'remove', this.handleChange);
  }

  componentWillUnmount() {
    this.props.collection.off('add', 'remove', this.handleChange);
  }

  render() {
    return (
      <ul>
        {this.props.collection.map(model => (
          <Item key={model.cid} model={model} />
        ))}
      </ul>
    );
  }
}
```

Try it on CodePen

Backbone Model からデータを抽出する

上記のアプローチでは React コンポーネントが Backbone のモデルとコレクションを認識することが必要です。後で別のデータ管理ソリューションに移行する予定がある場合は、Backbone とのインテグレーション箇所はできるだけコードの一部に集中させることをお勧めします。

1 つの解決法は、変更があるたびにモデルの属性をプレーンなデータとして抽出するようにし、そのロジックを一箇所だけで保持することです。以下のコードは Backbone モデルの全ての属性を state へと抽出し、そのデータをラップされるコンポーネントへと渡す高階コンポーネントです。

このようにすれば、高階コンポーネントだけが Backbone モデルの内部動作について知っておく必要があり、アプリケーション内のほとんどのコンポーネントは Backbone について知らないままでいられるのです。

以下の例では、モデルの属性のコピーを state の初期値に設定します。`change` イベントを監視して（アンマウント時には監視を解除し）、変更が発生した際には state をモデルの現在の属性で更新します。最後に、`model` プロパティそのものが変更となった場合には、必ず忘れずに古いモデルの監視を解除し、新しいモデルから情報を受け取るようにします。

この例は Backbone と協調して動作させるための網羅的なものとして書かれているわけではないことに注意すべきですが、この種の問題にどうアプローチすべきかの一般的な理解の助けになるはずです：

```
function connectToBackboneModel(WrappedComponent) {
  return class BackboneComponent extends React.Component {
    constructor(props) {
      super(props);
      this.state = Object.assign({}, props.model.attributes);
      this.handleChange = this.handleChange.bind(this);
    }

    componentDidMount() {
      this.props.model.on('change', this.handleChange);
    }

    componentWillReceiveProps(nextProps) {
      this.setState(Object.assign({}, nextProps.model.attributes));
      if (nextProps.model !== this.props.model) {
        this.props.model.off('change', this.handleChange);
        nextProps.model.on('change', this.handleChange);
      }
    }

    componentWillUnmount() {
      this.props.model.off('change', this.handleChange);
    }

    handleChange(model) {
      this.setState(model.changedAttributes());
    }

    render() {
      const propsExceptModel = Object.assign({}, this.props);
      delete propsExceptModel.model;
      return <WrappedComponent {...propsExceptModel} {...this.state} />;
    }
  }
}
```

このコードの使い方を例示するために、`NameInput` という React コンポーネントを Backbone モデルと接続して、入力が変更されるたびに `firstName` 属性を更新します：

```
function NameInput(props) {
  return (
    <p>
      <input value={props.firstName} onChange={props.handleChange} />
      <br />
      My name is {props.firstName}.
    </p>
  );
}

const BackboneNameInput = connectToBackboneModel(NameInput);

function Example(props) {
  function handleChange(e) {
```

```
    props.model.set('firstName', e.target.value);
  }

  return (
    <BackboneNameInput
      model={props.model}
      handleChange={handleChange}
    />

  );
}

const model = new Backbone.Model({ firstName: 'Frodo' });
ReactDOM.render(
  <Example model={model} />,
  document.getElementById('root')
);
```

Try it on CodePen

この手法は Backbone だけに限ったものではありません。ライフサイクルメソッドで変更を購読し必要に応じてデータをローカルの React 状態にコピーすることで、任意のモデルライブラリで React を使用できます。

このページを編集する