React

[Node.js]

スケルトン作成

1.React 開発の全体像を把握しつつ開発環境を整える
2.React の単純なサンプルに React Router を適用する
3.React の単純なサンプルに React Router を組み込んだものに Redux-Saga を適用する
4.React の単純なサンプルに React Router を組み込んだものに Redux-Saga を適用したものから Ajax 通信を組み込む
5.React 環境 ->React Router->Redux-Saga->SuperAgent に Bootstrap を適用する

導入

・https://facebook.github.io/react/docs/installation.html
・https://github.com/facebookincubator/create-react-app/blob/master/packages/react-scripts/template/README.md#tab
・React の SPA を作成するのによい方法

```
npm install -g create-react-app
```

・React はバックエンドロジックやデータベースを持たないが、使いたいものを使えばよい。
・Babel や webpack のようなビルドツールも設定なしに利用できる。

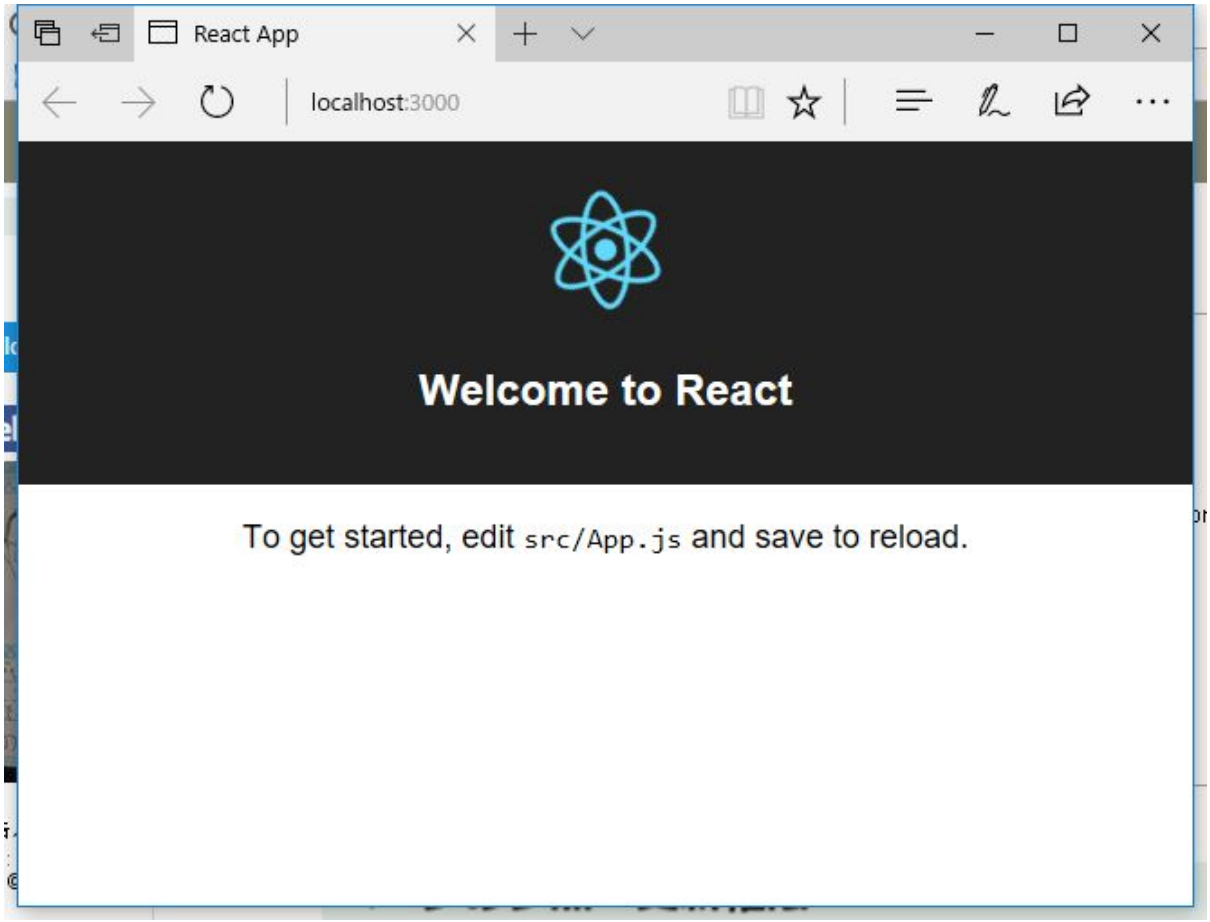
```
PS C:\workspaces\vscode\reactlesson> create-react-app react-lesson
Creating a new React app in C:\workspaces\vscode\reactlesson\react-lesson.

Installing packages. This might take a couple minutes.
Installing react, react-dom, and react-scripts...
              :
```

・アプリケーションが作成されたら実行

```
PS C:\workspaces\vscode\reactlesson> cd react-lesson
PS C:\workspaces\vscode\reactlesson> npm start
```

・実行された



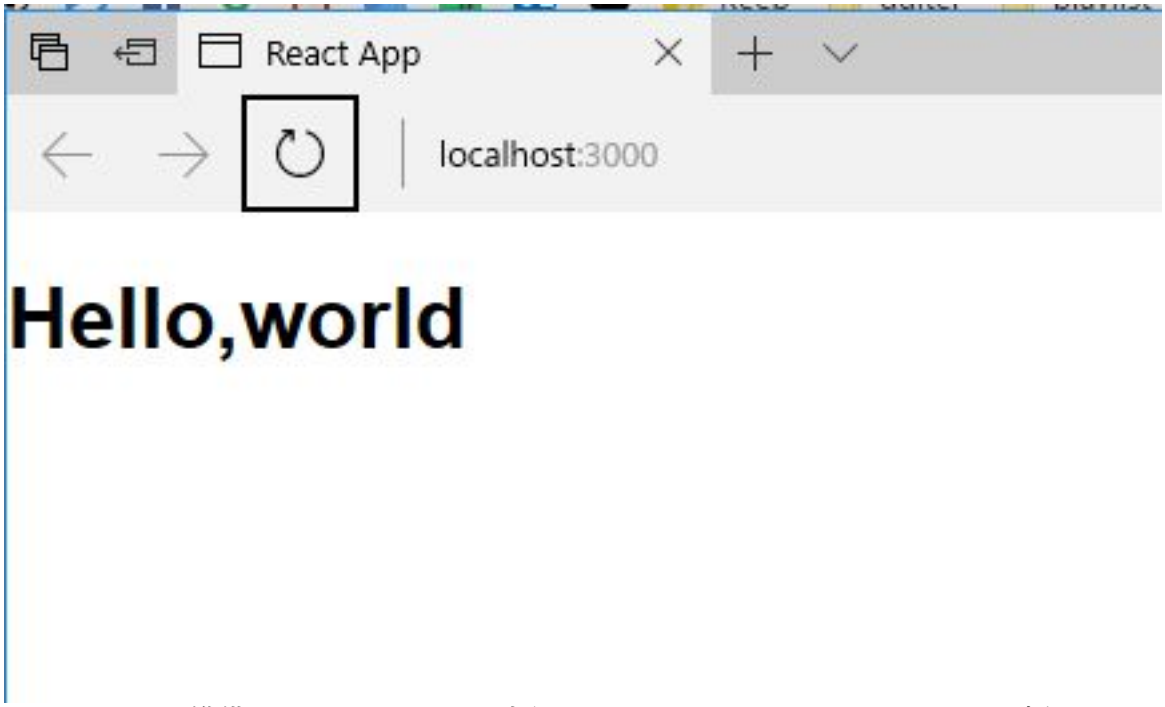
・/src/App.js を書き換えてみる

```
import React, { Component } from 'react';
import logo from './logo.svg';
import './App.css';

const element = (<h1>Hello,world</h1>);
class App extends Component {
  render() {
    return (
      element
    );
  }
}

export default App;
```

・反映された



・リリース準備ができたら、以下を実行することで、build フォルダ以下に最適化されたアプリケーションを作成する

```
npm run build
```

クイックスタート

JSX

```
const element = <h1>Hello,world!</h1>;
```

・文字列でも、HTML でもなく、JSX
・JavaScript の拡張文法
・テンプレートと思われるかもしれないが、完全な JavaScript
・JSX は React の要素を生成する

JSX 表現

・どのような JavaScript の表現も、中括弧で囲むことで JSX に埋め込むことができる
・const element = (<h1>Hello, {formatName(user)}!</h1>);

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Hello World</title>
    <script src="https://unpkg.com/react@latest/dist/react.js"></script>
    <script src="https://unpkg.com/react-dom@latest/dist/react-dom.js"></script>
    <script src="https://unpkg.com/babel-standalone@6.15.0/babel.min.js"></script>
  </head>
  <body>
    <div id="root"></div>
    <script type="text/babel">
    function formatName(user) {
        return user.firstName + ' ' + user.lastName;
    }
    const user = {firstName:'Hiroto', lastName:'Yagi'};
    const element = (<h1>Hello, {formatName(user)}!</h1>);
```

```
    ReactDOM.render(
        element,
        document.getElementById('root')
    );
    </script>
  </body>
</html>
```

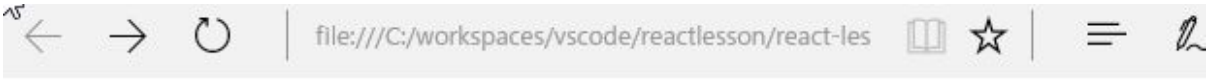
・コンパイルされた JSX は通常の JavaScript オブジェクト
・JSX を if 文や for ループ、変数への割り当て、引数や戻り値に利用できる

```
function greeting(user) {
   var now = new Date();
   if (5 <= now.getHours() && now.getHours() <= 12) {
       return <h1> Good morning {user.firstName + ' ' + user.lastName}.</h1>;
   } else {
       return <h1> Hello {user.firstName + ' ' + user.lastName}.</h1>;
   }
}
```

■属性に利用

・中括弧を利用して、二重引用符なし (使用すると文字列リテラルとして扱われる) で直接利用できる。

```
<div id="root"></div>
<script type="text/babel">
function greeting(user) {
    var now = new Date();
    if (5 <= now.getHours() && now.getHours() <= 12) {
            return <h1> Good morning <a href={user.webpageUrl}>{user.firstName + ' ' +
user.lastName}</a>.</h1>;
    } else {
        return <h1> Hello <a href={user.webpageUrl}>{user.firstName + ' ' + user.lastName}</a>.</h1
>;
    }
}
const user = {firstName:'Hiroto', lastName:'Yagi', webpageUrl:'http://typea.info'};
const element = (<h1>{greeting(user)}</h1>);
ReactDOM.render(
    element,
    document.getElementById('root')
);
```



Hello Hiroto Yagi.

■子要素

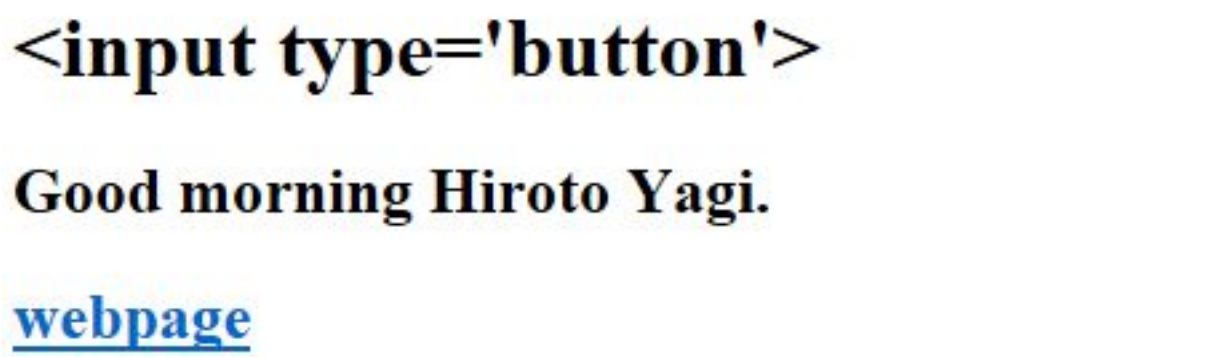
・子要素がない場合、XML 同様　/> で閉じる
・子要素を含む
・JSX は HTML より JavaScript により近い。ReactDOM はキャメルケースプロパティ (HTML では、class が className、HTML では tabindex が tabIndex など) を持つ

```
function greeting(user) {
    return <div>
            <h1> Good morning {user.firstName + ' ' + user.lastName}.</h1>
            <h2><a href={user.webpageUrl}>webpage</a></h2>
           </div>
           ;
}
const user = {firstName:'Hiroto', lastName:'Yagi', webpageUrl:'http://typea.info'};
const element = (greeting(user));
ReactDOM.render(
    element,
    document.getElementById('root')
);
```

■インジェクション攻撃の予防

・デフォルトで React DOM は、JSX に埋め込まれた値をレンダリング前にエスケープする
・明示的にアプリケーションに記述しなくてもインジェクションされないことを保証する
・XSS 攻撃の予防を助ける

```
function greeting(user, title) {
    return <div>
             <h1>{title}</h1>
             <h2> Good morning {user.firstName + ' ' + user.lastName}.</h2>
             <h2><a href={user.webpageUrl}>webpage</a></h2>
           </div>
           ;
}
const title = "<input type='button'>";
const user = {firstName:'Hiroto', lastName:'Yagi', webpageUrl:'http://typea.info'};
const element = (greeting(user,title));
ReactDOM.render(
    element,
    document.getElementById('root')
);
```



■ JSX Represent オブジェクト

・Babel は、React.createElement() を呼び出しコンパイルを行う
・以下の 2 つは同じ意味

```
const element1 = (<h1 className='greeting'>hello</h1>);
const element2 = React.createElement('h1',{className:'greeting'},'hello');
```

・React.createElement() は、バグを防ぐ手助けをするが、本質的には以下のようなオブジェクトを生成する

```
// 簡易表現
const element = {type:'h1',props:{className:'greeting',children:'hello'}};
```

・これらの要素を "React elements" と呼ぶ
・画面に表示させたいものの記述と考えることができる
・React はこれらのオブジェクトを読み、DOM の構築と最新化に利用する

Elements のレンダリング

・Elements は React アプリケーションの最小のビルディングブロック
・1 つの要素は、画面上に表示するものを表現する
・ブラウザの DOM 要素と異なり、React Elements はプレーンなオブジェクトで生成のコストは小さい
・React DOM は、DOM を React Elements に一致するように取り扱う

■ 1 つの要素を DOM の中にレンダリング

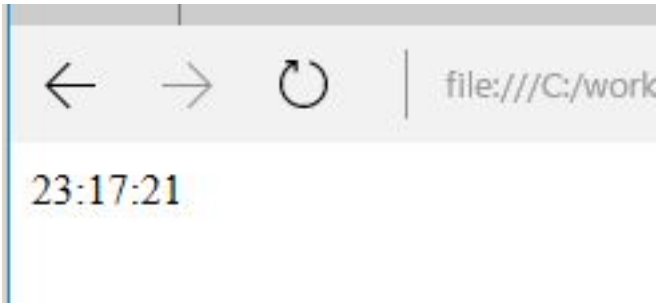
・HTML どこかに記述する、以下の DIV 要素を root DOM ノードと呼ぶ
・React DOM が管理するすべてが、この要素の中にある

```
<div id="root"></div>
```

■ React 要素の更新

・React 要素は不変。一旦生成したら、子要素、属性などは変更できない。
・UI を更新するには、新しい要素を作成し、ReactDOM.rendar() に引き渡す。

```
function tick() {
  const element = (<div>{(new Date()).toLocaleTimeString()}</div>);
  ReactDOM.render(
      element,
      document.getElementById('root')
  );
}
setInterval(tick,1000);
```



通常の React アプリケーションでは、ReactDOM.rendar() は 一度しか呼びださない。

■ React は必要なもののみ更新する

・React DOM は、要素および子要素を前の状態と比較し、DOM の更新が必要な個所にのみ適用する。
・ブラウザツールで、上記のソースコードを確認する

毎 tick() の呼び出しで、すべての UI ツリーを生成するよう記述してるが、変更が発生したテキストノードのみ React DOM により更新されている。

コンポーネントと Props

・コンポーネントは UI を独立し再利用可能な部分に分割する。
・概念的にコンポーネントは JavaScript の関数のようなもの。
・コンポーネントは props と呼ばれる任意の入力を受け付け、React Element を返す。

■コンポーネントの機能とクラス

・JavaScript の関数としてコンポーネントを定義する
・この関数は有効な React コンポーネント、なぜなら、単一の props 引数を引数として取り、React Element を返す。
・このようなコンポーネントを "functional" と呼ぶ。

```
function Welcome(props) {
    return <h1>Hello,{props.name}</h1>;
}
```

・ES6 のクラスをコンポーネントの定義として利用できる

```
class Welcome extends React.Component {
    render() {
        return <h1>Hello,{props.name}</h1>;
    }
}
```

・上記 2 つのコンポーネントは React の視点からは同じ

■コンポーネントのレンダリング

・Element は、ユーザー定義コンポーネントも表すことができる

```
function Welcome(props) {
 return <h1>Hello,{props.name}</h1>;
}
const element = <Welcome name='Hiroto' />
ReactDOM.render(
    element,
    document.getElementById('root')
);
```

・React がユーザー定義コンポーネントを表示するときに、JSX の属性からコンポーネントへ "props" としてオブジェクトが渡される。

コンポーネント名はいつも大文字から始める。DOM タグは、 だが、<Welcome /> はコンポーネントを表現する。Welcome がスコープに存在すること。

■コンポーネントの構成

・コンポーネントはその出力において、他のコンポーネントに影響を与えることができる
・同じコンポーネントをどんなレベルの詳細にも抽象的に利用できる、ボタン、フォーム、ダイアログ、スクリーン
・React でこれらは、一般にコンポーネントで表現される

```
function Welcome(props) {
  return <h1>Hello,{props.name}</h1>;
}
function App() {
  return ( <div>
      <Welcome name="Yagi"/>
      <Welcome name="Kaela"/>
      <Welcome name="Hiroto"/>
    </div> );
}
ReactDOM.render(
    <App />,
    document.getElementById('root')
);
```

・一般的に新しい React アプリケーションは、一つの App コンポーネントを最上位に持つ。既存のアプリケーションに React を統合する場合は、小さなコンポーネント、例えば Button など、ボトムアップから開始し、徐々に View 階層の最上位にいたる。

コンポーネントは、一つの root 要素を返さなければならない。上記で、Welcome 要素を div に含めたのはこのため

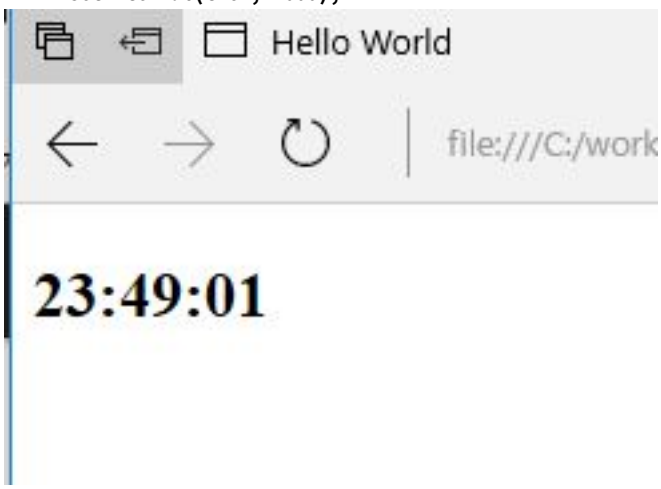
■ Props は読み取り専用

・React はかなりフレキシブルだが、1 つ厳格なルールがある。function だろうと class だろうとコンポーネントは props を編集できない。

状態とライフサイクル

・次の Clock コンポーネントを再利用可能にカプセル化する

```
function Clock(props) {
  return (
    <div>
      <h2>{props.date.toLocaleTimeString()}</h2>
    </div>
  );
}
function tick() {
  ReactDOM.render(
      <Clock date={new Date()} />,
      document.getElementById('root')
  );
}
setInterval(tick, 1000);
```



・実装のために、Clock コンポーネントに state を追加する。
・状態は、props と同様だが、private かつ、コンポーネントから完全に制御される
・クラス宣言したコンポーネントにはいくつかの追加的な機能があるが、local state はこれにあたる

■関数をクラスに変更する

1.React.Component を継承して同名の ES6 クラスを作成する
2.render() メソッドを作成し、処理を移動、function を削除
3.props を this.props に変更

```
class Clock extends React.Component {
  render() {
    return (
      <div>
        <h2>{this.props.date.toLocaleTimeString()}</h2>
      </div>
    );
  }
}
```

■クラスにローカル状態を追加する

・date を props から state へ移動する
1.render() メソッドの中の、this.props.date を this.state.date に変更
2. クラスにコンストラクタを追加し、this.state の初期状態を記述する
3.<Clock date={new Date()} /> から date を削除

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date : new Date()};
  }
  render() {
    return (
      <div>
        <h2>{this.state.date.toLocaleTimeString()}</h2>
      </div>
    );
  }
}
```

■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使用するアプリケーションでは、破棄されたコンポーネントのリソースの解放が重要
・Clock が DOM に最初にレンダリングされるときには、必ずタイマーをセットアップしたい。これを React では、"mounting" という
・また、Clock が DOM から取り除かれるときには、タイマーをクリアしたい。これを React では、"unmounting" という
・コンポーネントが、mount/unmount されるときに実行される特別なメソッドをクラスに宣言できる
・componentDidMount()、componentWillUnmount() これらのメソッドはライフサイクルフックと呼ばれる
・どのように timer ID を this に保存するか
  ・this.props が React によりセットアップされる間、this.state は特別な意味を持つ
  ・表示に使用しないならば、クラスに自由にフィールドを追加できる
  ・render() の中では、それらは利用できないし、ステートも持たない
・最後に、毎秒実行される tick() メソッドを実装する。
・this.setState() でローカルコンポーネントの状態を更新する

```
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>
        <h2>{this.state.date.toLocaleTimeString()}</h2>
      </div>
    );
  }
}
ReactDOM.render(
  <Clock />,
  document.getElementById('root')
);
```

■状態を正しく使う

state を直接操作しない

```
// まちがい
this.state.comment = 'Hello';
```

・上記の代わりに、setState() を利用する

```
// 正しい
this.setSatate({comment:'Hello'});
```

this.state を割り当てることができるのは、コンストラクタのみ

state は、非同期に更新される

・React は、複数の setState() をパフォーマンスのためにまとめて一度に処理する。
・このため、this.props と this.state は非同期に更新される
・これらの値を信頼して、次の状態への計算を行うべきではない

```
// まちがい
this.setState({
  counter: this.state.counter + this.props.increament,
});
```

・修正するには、オブジェクトではなく関数を受け取る、setState() を使用する
・関数は、ひとつ前の状態 (state) を最初の引数として、更新時の props を 2 つ目の引数として取る

```
// 正しい
this.setState((prevState, props) => ({
  counter: prevState.counter + prpos.increament
}));
```

・通常の関数でもよい

```
// 正しい
this.setState(function(prevState, props) {
  return { counter: prevState.counter + props.increament };
});
```

状態の更新はマージされる

・setState() を呼び出すと、React は、現在の state に提供したオブジェクトをマージする
・例えば、いくつかの独立した変数を含む場合

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

・これらを別の setState() で更新

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

・マージはシャローなので、setState({comment}) は、this.state.posts を損なわずに、this.state.comments を置き換える

■データは下へ流れる

・親だけでなく子コンポーネントも確実にステートフルかステートレスか知ることはできない。
・状態はしばしばローカルから呼び出されるかカプセル化されていて、他からアクセスできない。
・コンポーネントは、自身の状態を props を通して子コンポーネントに渡す

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

・これは、ユーザー定義コンポーネントでも動作する

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

・FormattedDate コンポーネントは、props に date を受け取り、Clock の状態が何かは知らない。

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

・これは一般的に、トップダウン、もしくは、ユニディレクショナル データフローという
・状態は常に特定のコンポーネントに属する。
・データや UI は状態に由来し、ツリーのより下のコンポーネントにのみ影響を与えることができる

イベントの処理

・React 要素のイベント処理は、DOM での処理ととても似ているが、いくつかシンタックスの差異がある。
  ・React イベントは、ロウアーケースではなく、キャメルケース
  ・JSX では、string ではなく、関数をイベントハンドラーとして渡す。

HTML

```
<button onclick="test()" />
```

React

```
<button onClick={test} />
```

・もう一つの違いは、React では、false を返しデフォルトのふるまいを防止することができない。
・明示的に、preventDefault を呼び出す必要がある。

HTML でデフォルトの新しいページを開くリンクのふるまいを防止する

```
<a href="#" onclick="console.log('hoge'); return false" >Click</a>
```

React

```
function ActionLink() {
  function handleClick(e) {
    e.preventDefault();
    console.log('hoge');
  }
  return (
    <a href="#" onClick={handleClick} >Click</a>
  )
}
```

・e は、本物ではないイベント。React では、W3C に合わせるために定義している。
・なので、ブラウザの互換性を心配する必要はない。リファレンスを参照
・React では、addEventListener を通常呼び出す必要はない。要素が最初にレンダリングされるときに提供される
・ES6 のクラスを使用するときの一般的なパターン、例えば、Toggle ボタン

```
class Toggle extends React.Component {
  constructor(props) {
    super(props);
    this.state = {isToggleOn: true};
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() {
    this.setState(prevStae => ({
      isToggleOn: !prevStae.isToggleOn
    }));
  }
  render() {
    return (
      <button onClick={this.handleClick} >
        {this.state.isToggleOn?"ON":"OFF"}
      </button>
    );
   }
}
ReactDOM.render(
  <Toggle />,
  document.getElementById('root')
);
```

・この文法の問題点は、LoggingButton がレンダリングされるたびに、異なったコールバックが生成されること
・多くのケースで、これは問題ないが、このコールバックが下位のコンポーネントに prop として渡される場合、コンポーネントは余分にレンダリングすることになる。
・コンストラクタでバインドするか、プロパティイニシャライザでバインドすることを推奨する。

条件付きのレンダリング

・React では、必要なら、振る舞いがカプセルかされた他と異なったコンポーネントを作成できる。
・React の条件付きレンダリングは、JavaScript の条件が動作するのと同じ方法で動作する。
・JavaScript の if のような演算子もしくは、条件演算子は、現在の状態を表した要素を作成するための、React が UI をそれに一致させることを許す。

以下の 2 つのコンポーネントについて考える

・ユーザーがログインしているか否かによって使い分けたい。
・isLoggedIn により、異なるレンダリングがなされる。

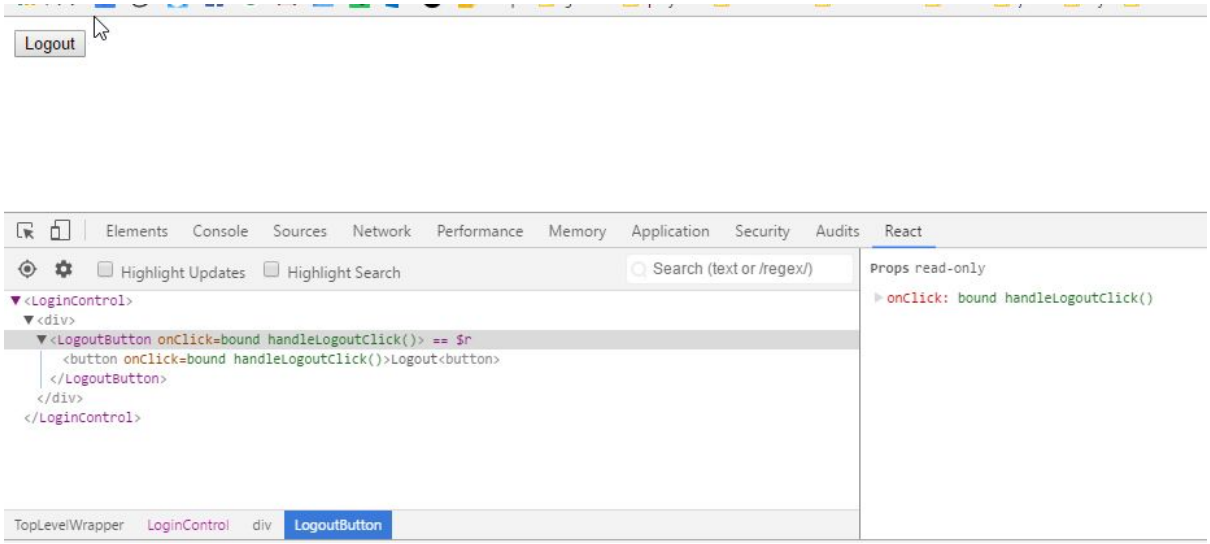
```
function UserGreeting(props){
   return <h1>Welcome to back!</h1>
}
function GuestGreeting(props) {
   return <h1>Please Sign up!</h1>
}
function Greeting(props) {
   const isLoggedIn = props.isLoggedIn;
   if (isLoggedIn) {
     return <UserGreeting />
   } else {
     return <GuestGreeting />
   }
}
ReactDOM.render(
  <Greeting isLoggedIn={true} />,
  document.getElementById('root')
);
```

■要素変数

・要素を保持するのに変数を利用できる。
・これは、出力結果が変わらない間に、条件によりコンポーネントの一部をレンダリングするのを助ける。

```
function LoginButton(props) {
   return (
     <button onClick={props.onClick} >
       Login
     </button>
   );
}
function LogoutButton(props) {
   return (
     <button onClick={props.onClick} >
       Logout
     </button>
   );
}
class LoginControl extends React.Component {
   constructor(props) {
     super(props);
     this.handleLoginClick = this.handleLoginClick.bind(this);
     this.handleLogoutClick = this.handleLogoutClick.bind(this);
     this.state = {isLoggedIn: false};
   }
   handleLoginClick() {
     this.setState({isLoggedIn: true});
   }
   handleLogoutClick(){
     this.setState({isLoggedIn: false});
   }
   render() {
     const isLoggedIn = this.state.isLoggedIn;
     let button = null;
     if (isLoggedIn) {
       button = <LogoutButton onClick={this.handleLogoutClick} />
     } else {
       button = <LoginButton onClick={this.handleLoginClick} />
     }
     return (<div>{button}</div>);
   }
}


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



■インライン if と 論理 && 演算子

・true && 演算子の場合評価され、false && の場合、false となり React は無視する

```
function MailBox(props) {
   const unreadMessages = props.unreadMessages;
   return  (
     <div>
       <h1>Hello!</h1>
       {unreadMessages.length > 0 &&
         <h2>
           You have {unreadMessages.length} unread messages.
         </h2>
       }
     </div>
   );
}
const messages = ['message1','message2'];
ReactDOM.render(
  <MailBox unreadMessages={messages} />,
  document.getElementById('root')
);
```

■インライン if-else とコンディション演算子

・条件 ? true : false

```
function UserGreeting(props){
    return <h1>Welcome to back!</h1>
}
function GuestGreeting(props) {
    return <h1>Please Sign up!</h1>
}
function Greeting(props) {
    const isLoggedIn = props.isLoggedIn;
    return (
      <div>
      {isLoggedIn ? (
          <UserGreeting />
        ) : (
          <GuestGreeting />
        )
      }
      </div>
    );
}
ReactDOM.render(
  <Greeting isLoggedIn={false} />,
  document.getElementById('root')
);
```

■コンポーネントのレンダリングさせない

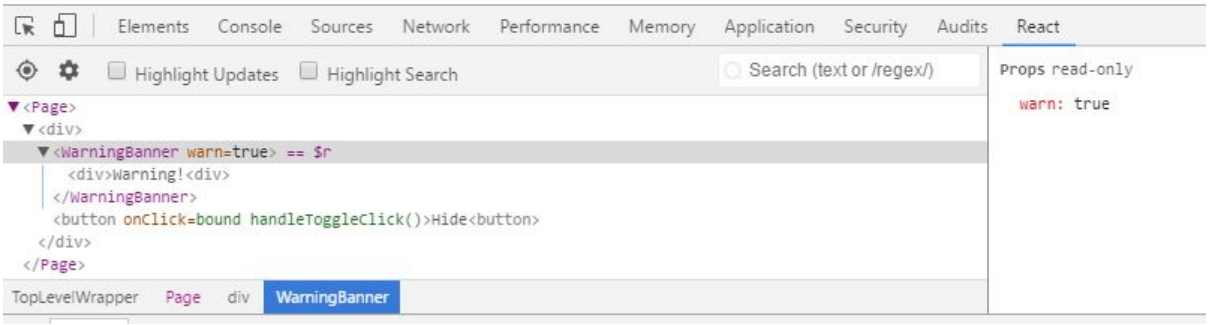
・他のコンポーネントからレンダリングされたとしてもコンポーネントを隠したい場合、render 出力の代わりに null を返す

```
function WarningBanner(props){
  if(!props.warn) {
    return null;
  }
  return (
    <div>Warning!</div>
  );
}
class Page extends React.Component {
  constructor(props) {
    super(props);
    this.state = {showWarning : true}
    this.handleToggleClick = this.handleToggleClick.bind(this);
  }
  handleToggleClick() {
    this.setState(prevState =>({showWarning : !prevState.showWarning}));
  }
  render() {
    return (
      <div>
        <WarningBanner warn={this.state.showWarning} />
        <button onClick={this.handleToggleClick}>
          {this.state.showWarning ? 'Hide' : 'Show'}
        </button>
      </div>
    );
  }
}

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

Warning!

Hide



リストとキー

JavaScript でリストの変換には、map() を利用する

```
const numbers = [1,2,3,4,5];
const doubled = numbers.map((number)=> number * 2);
console.log(doubled)
> [2, 4, 6, 8, 10]
```

・React では同様に配列をリストに変換する

■複数のコンポーネントをレンダリングする

```
const fruits = ['apple','orange','grape','banana'];
const listItems = fruits.map((fruit) => <li>{fruit}</li>);
ReactDOM.render(
  <ol>{listItems}</ol>,
  document.getElementById('root')
);
```

■基本的なリストコンポーネント

・通常リストはコンポーネントに含まれる

Keys

・key を指定しないと警告がでる
・key は、React が、どのアイテムが登録、変更、削除されたのか特定するのに役立つ
・文字列で兄弟間でユニークな値を設定するのが望ましい

```
function FruitList(props) {
  const fruits = props.fruits;
  const listItems = fruits.map((fruit) => <li key={fruit}>{fruit}</li>);
  return (
    <ol>{listItems}</ol>
  );
}
const fruits = ['apple','orange','grape','banana'];
ReactDOM.render(
  <FruitList fruits={fruits}/>,
  document.getElementById('root')
);
```

■コンポーネントを Key と取り出す

・Key が意味を持つのは、配列を囲むコンテキストにおいて
・兄弟間で同じ Key を持つとエラーとなる

```
function ListItem(props) {
  // この <li> に key を設定するのは、ここがルートになるため間違い
  return <li>{props.value}</li>
}
function FruitList(props) {
  const fruits = props.fruits;
  // <ListItem> の配列となるため、ここで key を指定するのが正しい
  const listItems = fruits.map((fruit) => <ListItem key={fruit} value={fruit} />);
  return (
    <ol>{listItems}</ol>
  );
}
const fruits = ['apple','orange','grape','banana'];
```

```
ReactDOM.render(
  <FruitList fruits={fruits}/>,
  document.getElementById('root')
);
```

■ JSX で map() を埋め込む

・中括弧のインラインに map() を置くことができる

```
function ListItem(props) {
  return <li>{props.value}</li>
}
function FruitList(props) {
  const fruits = props.fruits;
  return (
    <ol>{fruits.map((fruit) => <ListItem key={fruit} value={fruit} />)}</ol>
  );


}
const fruits = ['apple','orange','grape','banana'];
ReactDOM.render(
  <FruitList fruits={fruits}/>,
  document.getElementById('root')
);
```

Form

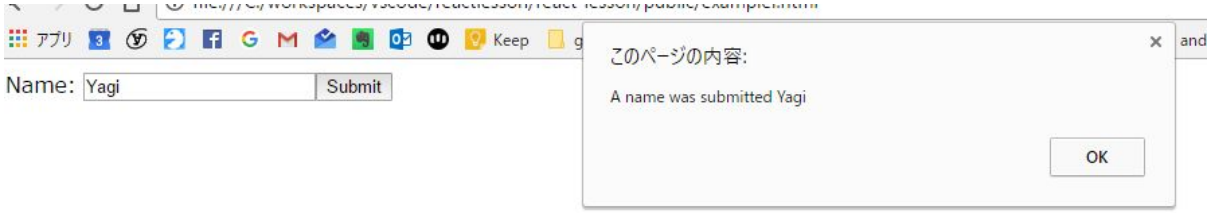
・HTML Form は、React では、少し他の DOM とは動作が異なる。
・なぜなら From は通常内部状態をもつから。

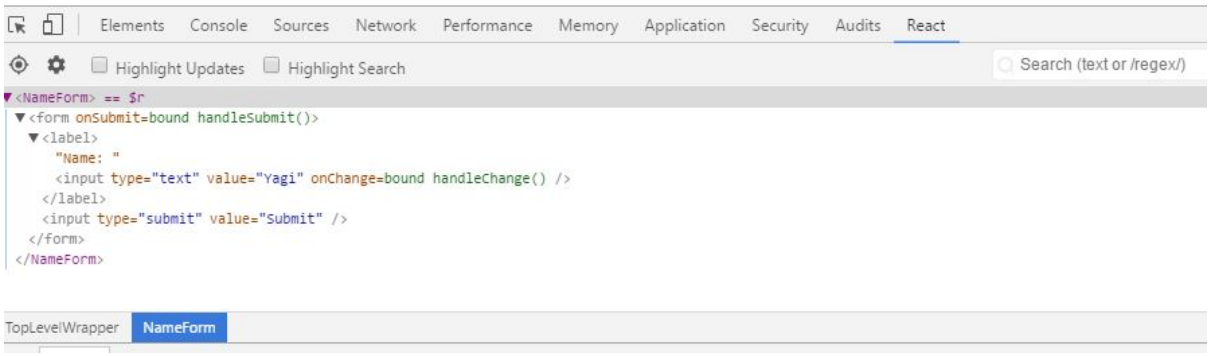
■コントロールされたコンポーネント

・HTML で Form は、自身の状態を維持しユーザー入力に基づいて状態を更新する。
・React では、変化可能な状態はコンポーネントの state プロパティに保持し、setState() からのみ更新される。
・"single source of truth" として React state を作成することにより 2 つを結びつけることができる。
・React コンポーネントは Form またコントロールでユーザー入力により、何が起きたかを描画する。
・input Form 要素の値は、React によりこの方法で制御される。
・これは、「コントロールされたコンポーネント」と呼ばれる。

```
class NameForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value : ''};
    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }
  handleChange(event) {
    this.setState({value : event.target.value });
  }
  handleSubmit(event) {
    alert('A name was submitted ' + this.state.value);
    event.preventDefault();
  }
  render() {
    return(
      <form onSubmit={this.handleSubmit}>
            <label>Name:  <input  type="text"  value={this.state.value}  onChange={this.handleChange}
/></label>
        <input type="submit" value="Submit"/>
      </form>
    );
  }
}
ReactDOM.render(
```

```
    <NameForm />,
    document.getElementById('root')
  );
```





・value が、Form 要素に設定されると、表示される value は、常に this.state.value となる
・すべてのキーストロークで、handleChange が起動し、React の state を更新する
・表示される value も更新される

```
handleChange(event) {
  this.setState({value : event.target.value.toUpperCase() });
}
```

■ textarea/select タグ

```
class SelectParts extends React.Component {
  constructor(props){
    super(props);
    this.state = {value: 'coconuts'};
    this.handleChange = this.handleChange.bind(this);
  }
  handleChange(event) {
    this.setState({value: event.target.value});
  }
  render(){
    return(
      <label>Select:
        <select value={this.state.value} onChange={this.handleChange}>
          <option value="grapefruit">Grape Fruit</option>
          <option value="lime">Lime</option>
          <option value="coconuts">Coconuts</option>
          <option value="mango">Mango</option>
        </select>
      </label>
    );
  }
}
class TextareaParts extends React.Component {
  constructor(props){
    super(props);
    this.state = {
      value : ' なにか書いて '
    };
    this.handleChange = this.handleChange.bind(this);
  }
  handleChange(event) {
    this.setState({value : event.target.value});
  }
  render(){
    return (
```

```
        <label>Textarea:<textarea value={this.state.value} onChange={this.handleChange} /></label>
    );
  }
}
ReactDOM.render(
  <div>
    <SelectParts />
    <TextareaParts />
  </div>,
  document.getElementById('root')
);
```

■複数入力項目の処理

・複数のコントロールされた入力要素を処理したい場合、name 属性を各要素に追加する
・ハンドラー関数で、event.target.name の値に基づいて、どの要素か判定
・ES6 計算されたプロパティ名構文を key 状態を更新するのに使用

```
class Reservation extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      isGoing: true,
      numberOfGuests: 2
    };
    this.handleInputChange = this.handleInputChange.bind(this);
  }
  handleInputChange(event) {
    const target = event.target;
    const value = (target.type == 'checkbox') ? target.checked : target.value;
    const name = target.name;
    this.setState({[name]: value});
  }
  render(){
    return (
      <form>
            <label>Is going:<input name="isGoing" type="checkbox" checked={this.state.isGoing}
onChange={this.handleInputChange}/></label>
        <br />
                    <label>Number of guests:<input name="numberOfGuests" type="number"
value={this.state.numberOfGuests} onChange={this.handleInputChange}/></label>
      </form>
    );
  }
}
ReactDOM.render(
  <div>
    <Reservation />
  </div>,
  document.getElementById('root')
);
```

■コントロールされたコンポーネント以外の方法

・コントロールされたコンポーネントを利用するのが、冗長な場合がある。
・入力フォームを実装するための、コントロールされないコンポーネントを参照。

状態を持ち上げる

・いくつかのコンポーネントは同じ変更をデータに反映させることを必要とする。
・共有状態をそれらの、最も近い共通の祖先に持ち上げることを推奨する。

水が沸騰しているかどうかの計算を行う。

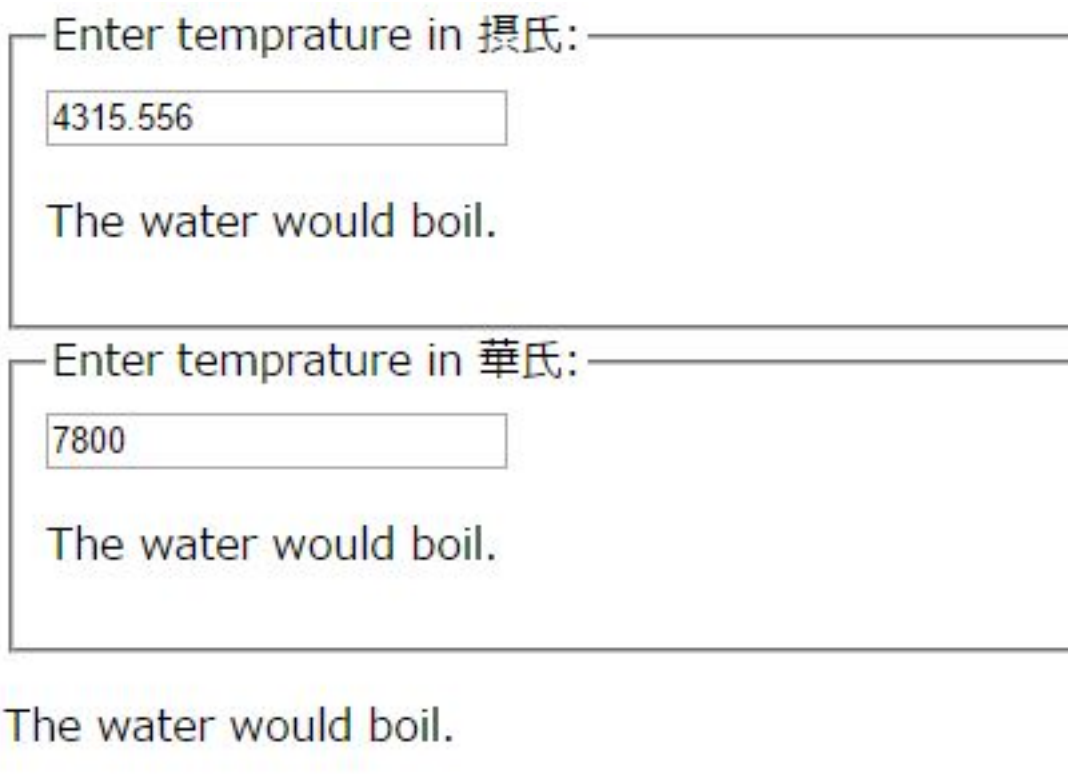
```
function BoilingVerdict(props) {
  if (props.celsius >= 100) {
    return <p>The water would boil.</p>;
  }
  return <p>The water would not boil.</p>
}
class Calculator extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {temprature: ''};
  }
  handleChange(e){
    this.setState({temprature: e.target.value});
  }
  render(){
    const temprature = this.state.temprature;
    return (
      <fieldset>
        <legend>Enter temprature in Celsius:</legend>
        <input value={temprature} onChange={this.handleChange} />
        <BoilingVerdict celsius={parseFloat(temprature)} />
      </fieldset>
    );
  }
}
ReactDOM.render(
  <div>
    <Calculator />
  </div>,
  document.getElementById('root')
);
```

■ 2 つ目の入力を追加する

・華氏の入力を追加し同期させる
・React で state を共有するには、共通の祖先に移動させることで実現でき、「状態の持ち上げ」という
・TempratureInput のローカル state を Calcurator へ移動させる
・Calucrator の共有 state は、「真の情報源」となる
・TempratureInput にとって、tempurature がローカルの場合、props は読み取り専用となるので、this.setState() とするしかないが、temprature が、親の props なので、管理する必要はなくなる。

```
const scaleName = {
  c: '摂氏',
  f: '華氏'
};
function BoilingVerdict(props) {
  if (props.celsius >= 100) {
    return <p>The water would boil.</p>;
  }
  return <p>The water would not boil.</p>
```

```
}
function toCelsius(fahrenheit) {
  return (fahrenheit - 32) * 5/9;
}
function toFahrenheit(celsius) {
  return (celsius * 9/5) + 32;
}
function tryConvert(temprature, convert) {
  const input = parseFloat(temprature);
  if (Number.isNaN(input)) {
    return '';
  }
  const output = convert(input);
  const rounded = Math.round(output * 1000) / 1000;
  return rounded.toString();
}
class TempratureInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {temprature: ''};
  }
  handleChange(e){
    // before:this.setState({temprature: e.target.value});
    this.props.onTempratureChange(e.target.value);
  }
  render(){
    // before:const temprature = this.state.temprature;
    const temprature = this.props.temprature;
    const scale = this.props.scale;
    return (
      <fieldset>
        <legend>Enter temprature in {scaleName[scale]}:</legend>
        <input value={temprature} onChange={this.handleChange} />
        <BoilingVerdict celsius={parseFloat(temprature)} />
      </fieldset>
    );
  }
}
class Calculator extends React.Component {
  constructor(props) {
    super(props);
    this.handleCelsiumChange = this.handleCelsiumChange.bind(this);
    this.handlFahrenheitChange = this.handlFahrenheitChange.bind(this);
    this.state = {scale:'c', temprature:''};
  }
  handleCelsiumChange(temprature) {
    this.setState({scale:'c', temprature});
  }
  handlFahrenheitChange(temprature) {
    this.setState({scale:'f', temprature});
  }
  render() {
    const scale = this.state.scale;
    const temprature = this.state.temprature;
    const celsius = scale == 'f' ? tryConvert(temprature, toCelsius) : temprature;
    const fahrenheit = scale == 'c' ? tryConvert(temprature, toFahrenheit) : temprature;
    return (
      <div>
                                        <TempratureInput        scale="c"        temprature={celsius}
onTempratureChange={this.handleCelsiumChange} />
                                      <TempratureInput       scale="f"       temprature={fahrenheit}
onTempratureChange={this.handlFahrenheitChange}/>
        <BoilingVerdict celsius={parseFloat(celsius)} />
      </div>
    );
  }
}
ReactDOM.render(
  <div>
    <Calculator />
  </div>,
  document.getElementById('root')
);
```



コンポジションと継承

・React は強力なコンポジションモデルをもつ。コンポーネントの再利用には継承よりコンポジションを推奨する

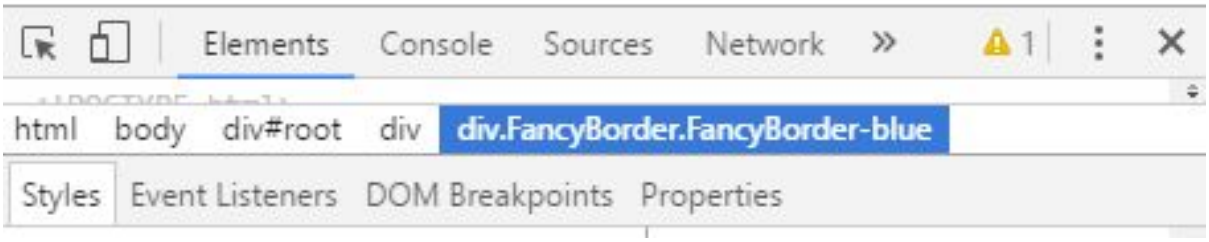
■封じ込める

・一部のコンポーネントは自身の子供を事前に知ることができない
・これは、特にサイドバーやダイアログなどボックスを表現するコンポーネント共通
・このようなコンポーネントには特別な子供 prop を使用し直接その要素に出力する
・<FancyBorder> の内側に何があっても、JSX タグは FancyBorder コンポーネントを props.children として経由する

```
<style type="text/css">
  div.FancyBorder {
    border : 2px dashed;
  }
  div.FancyBorder-blue {
    border-color: blue;
  }
</style>
```

```
function FancyBorder(props) {
  return (
    <div className={'FancyBorder FancyBorder-' + props.color}>
      {props.children}
    </div>
  );
}
function WelcomDialog() {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">Welcome</h1>
    </FancyBorder>
  );
}
```

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



■特殊化

・コンポーネントを他のコンポーネントの特殊なものとして考えることがある、例えば、WelcomDialog は Dialog の特殊な例として。
・React では、コンポジションを使って実現する
・クラスを用いても同様に記述できる

```
function Dialog(props) {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">{props.title}</h1>
    </FancyBorder>
  );
}
function FancyBorder(props) {
  return (
    <div className={'FancyBorder FancyBorder-' + props.color}>
      {props.children}
    </div>
  );
}
function WelcomDialog() {
  return (
    <Dialog title="Welcome" />
  );
}
```

React Router

・React Router

Redux

・Redux

Tips

・React のシンタックスエラーをデバッグ

■ビルド

・https://mae.chab.in/archives/2765
・https://mae.chab.in/archives/2891