

明日からできない React.js, Redux入門

落合隆行

じゃ、説明始めまーす

今日伝えること

- ・ React.jsの概要・メリット
- ・ FluxとReduxの概要・メリット
- ・ 今後興味を持って自習したりするときに必要な全体の俯瞰図

伝えないこと

- ・ React.jsやReduxで一ふえくとぷりちーなアプリを作る方法

まず、今日のゴールについて確認します。

今日はReact.jsとReduxについて、その概要とどういう時に役立つのか、あとちょっぴりどういう実装や処理の流れになるのかを説明します。

今後もしReactやReduxといったものに興味を持っていただけた時に、
どういう考え方で作られていて、自分がどういう知識が足りていないかを理解してもらうための全体像を掴んでもらうことを目的としています。

逆にReact.jsやReduxを使って具体的にどうやってアプリケーションを作っていくのかというところをガツガツやる感じではありません。

さすがに30分かそこら解説して書けるようになる程学習コストが低いものではないので、諦めてください。

質問はReact.jsについて喋ったあとととか、キリのいいタイミングで受け付けますが、途中で止めてくださっても構いません。

Reactのことなんでも知ってるぜ！ってわけじゃないので、答えられない質問だったらわかりません！って言います！すいません！

React



じゃ、まずReactについて説明します。

4行でわかるReact

- ・ Facebookがメインで開発を進めているOSS
- ・ 画面の描画に関わるJavaScriptのViewライブラリ（フレームワークではない！）
- ・ Google先生が担っているAngularと双璧をなす完成度と資料の豊富さで、採用実績も◎
- ・ ライブラリであって、フレームワークではない（大事なことでry

まず初めに、技術的なこと以外について、Reactの説明をします。

React.jsはFacebookが主導して開発を進めているOSSのViewを描画するためのライブラリです。SPAを作ったりリッチなUIを作ったり、UIのコンポーネント化を促進するためのライブラリとして急成長しています。React.jsはFacebookが開発を進めていて、Facebookはもちろん国内外の大きなプロダクトで導入された実績があるライブラリなので、品質の高さ、資料の豊富さ、実績の蓄積が抜きん出ています。

WebアプリケーションでリッチなUIUXを作りたい。でも保守性も高く品質も維持して開発のスピードも落としたくない、という状況でのクライアント側の技術選定を行う場合、Googleが主導で開発を進めているAngularと共に使用技術の第一候補にあがって来ます。

ポイントは、ライブラリであってフレームワークではない。ということです。

あとで説明しますがReactの良い点を生かしたフレームワークの実装がReduxなどになっていきます。

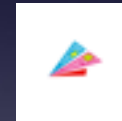
採用事例

海外



NETFLIX

国内



参考) <https://github.com/facebook/react/wiki/Sites-Using-React>

国外だと

Facebookはもちろん、UberとかNetflixとかが採用してます。

国内だと

Gunosy

ランサーズ

wantedly

pixiv

freee

qiita

サイボウズ

自社でサービス持ってる事業会社さんで最近ちらほら採用事例が出てきています。

全部React、ってのは少なく、ある機能の部分だけReactで作りましたとかリニューアルしました。というのは

最近のフロントエンド

- ・ スマホの普及
- ・ クライアント端末の性能向上
- ・ ネイティブアプリの普及

ユーザーの要求は大絶賛増速中

では、本格的な紹介に入る前に、現在の画面側の実装はどんな状況か自分の認識を共有させてください。

スマートフォンの普及や技術の進歩・クライアント端末の性能向上に伴い、ユーザーの要求はどんどん高レベルなものになってます。

具体的には、みんなスマホからインターネットを使うようになって、PCだけでインターネットを使っていた頃と使われ方が変わってきました。

特にtoC向けの製品ではサクサク動いてアニメーションとかのインタラクションでユーザーの体験を途切れさせないネイティブアプリの普及に伴い、Webアプリケーションでもデスクトップアプリやネイティブアプリによく似た操作性や俊敏な動作が求められるケースが増えています。

それに引っ張られて、PCでもリッチなユーザー体験をさせたい、っていう要求も出てきてますね。

当然Webアプリの画面側への要求も増える

- ・もっとリアルタイムに動かしたい
- ・ネイティブアプリよりしくページ遷移でアニメーションとかけたい！
- ・もっとレスポンス早く！
- ・というかりロードするたびに画面が白くなるのはユーザー体験途切れるからヤダ

これらの要求はAjax + JavaScript(JQuery)でも解決できる
解決できるが、保守が難しくて実装は困難を極める

そんなわけで、Webアプリにも似たような要求が降りてくるよう似になりました。

たとえば、タップしたら画面が一瞬白くなる、なんてことなくスライドアニメーションで次のページに移ったり、画面のリロードリアルタイムにグラフ・チャートを描画していったりといったことです。

こうした高い操作性や俊敏な動作をWebアプリケーションで実現するためWebの世界でもシングルページアプリケーションと呼ばれる方法が開発されたりしています。

スライドに書いたような要求はAjaxと素のJSやjQueryでも実装はできます。

ですが、技術選定を誤って無計画に作っていくと、保守が難しくなって開発がどんどん困難になっていきます。

参考)

SPAは名前の通り、単一のページで構成されるWebアプリケーションのことです。初回のアクセスでは、まずページ全体をHTMLでサーバーから取得しますが、以降のページの更新・画面遷移などをブラウザで動くJavaScriptで賄います。JSだけで賄いきれないサーバー側のデータの更新やデータの取得はAjaxなどを利用します

フロントエンドをリッチにするための課題

- ・ JavaScriptからの従来のDOM操作は難しい
- ・ 見た目とデータが分離できていない
- ・ データが変更されたときの画面の変更は自分で書かないといけない
- ・ いつどこで誰が持ってるデータが変わって、画面のどこが変わるのか管理しきれてない

保守が難しくなって開発がどんどん困難になっていく原因とは何でしょうか。

例えば、次のような課題です。

AjaxとJQueryとかでゴリゴリコードを書いてみないと実感としてわからないかと思いますが、JavaScriptでDOMを操作するのは結構難しいものです。

例えばDOMはどのJSからでも変更可能なので、どこから書き換えが飛んでくるか予想がつかないとか、逆にあるボタンをクリックした時の処理をどこでしているかわからないとか（あ、うちのFWはon~イベントに関数を直接バインドさせるコードを自動生成してくれますけど、あれは例外だと思ってます）、そういった辛さを抱えています。

見た目とデータが分離できていないとかデータが変更された時の画面の変更は自分で書く、っていうのは

たとえばサーバーからデータを再取得した時は、画面を再描画する必要がありますよね。エンジニアはそれを実現するためにゴリゴリHTMLを操作して取得したデータをHTMLに埋め込むJQueryを書かなければなりません。

画面で現在展開しているデータを取得してPOSTしたいって時もHTML構造に合わせたJQueryを書かないといけません。

自分でもらってきたデータをもとにDOMを操作するようなコードを書くとミスがどうしても出ますし、要求が変わってHTML構造が変わったりしたら、そのJQueryが動かなくなることも頻発します。

Reactが目指す世界

- ・データ（JSONで保存）と画面を表示する部品（コンポーネント）を明確に切り離す
- ・画面を描画するときは、今のデータの状態からHTMLをクライアント側で作成して表示する
- ・保存されたデータが書き換えられたときは、まず保存しているデータを書き換えて、書き換え結果をもとにもう一回新しいHTMLを全部作り直す

これらの問題を解決するために作られたのがReact.jsです。

これらの問題を解決するために、Reactはこんな世界を作ればいいんじゃない？

と言っています。

まず、データ（JSONで保存）と画面を表示する部品（コンポーネント）を明確に切り離しましょう。そうすればデータを取得する時もいちいちDOMをいじってパースしてデータを取りに行く必要はなくなりますよね。

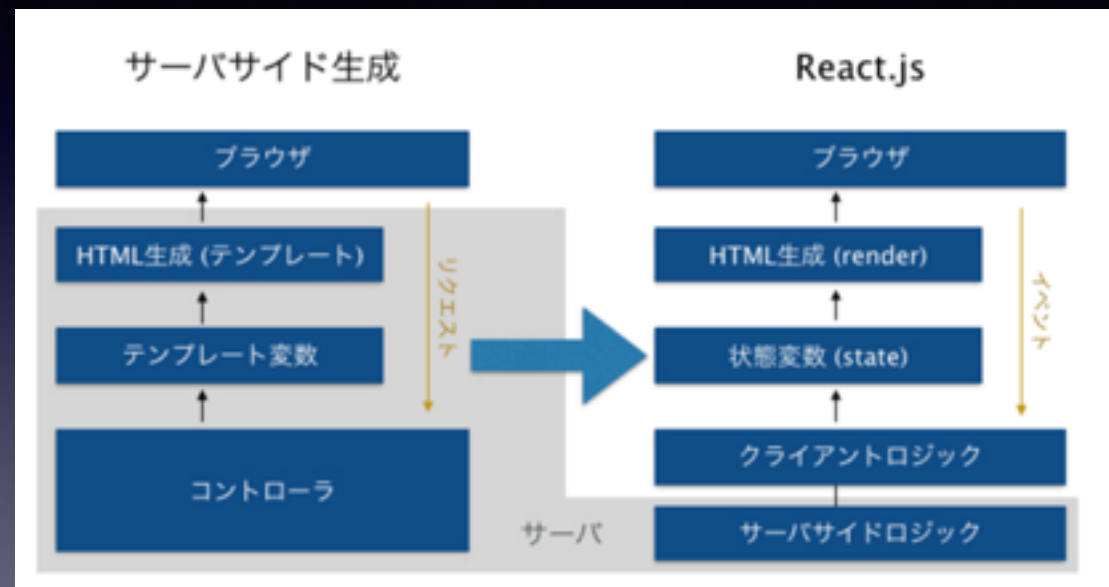
つづいて当然ですが、切り離したデータをもとに画面をクライアント側（要するにブラウザ側ですね）で、HTMLを作ってくれるレンダリング機能も持ちましょう。

保存されたデータが書き換えられたときは、まず保存しているデータを書き換えて、書き換え結果をもとにもう一回新しいHTMLを全部作り直す



3 番目、これだけ見ても何言ってんだってことになるかと思います。
でも、ここがReact一番のキモになります。

サーバーサイド生成と同じ仕組みをクライアントで



引用) <http://blog.masuidrive.jp/2015/03/03/react/>

これは、一昔前のWebアプリケーションをイメージしてもらうとわかりやすいと思います。

従来のAjaxも何もない時代のWebアプリケーションって、サーバー側に投げたリクエストをもとにDBを更新したり参照したりして、HTMLを生成して返すってことをやってました。そのHTMLを一箇所だけでも変えたければ、サーバーにリクエストを投げて画面全部のHTMLを書き換えてもらう必要がありました。

React.jsでは、このサーバサイド生成と同じ流れをクライアント側で行います。

クライアント側のロジックでは、ユーザーの操作とかのイベントが起こるとクライアント側で持っているデータを更新したり参照します。データが更新されると自動でrenderメソッドというHTMLを生成するメソッドが呼ばれて、そこでHTMLをまるっと全部再生成して表示してくれます。

しかもそこはFacebookの中の人がいろいろ考えてまるっとHTMLを作るっぽいことをするんですが、すごく低コストで素早く生成してくれるような仕組みを持っています！（ここの仕組みは今回は喋らないのですが、興味ある人は仮想DOMとかバーチャルDOMって言葉でググってください）

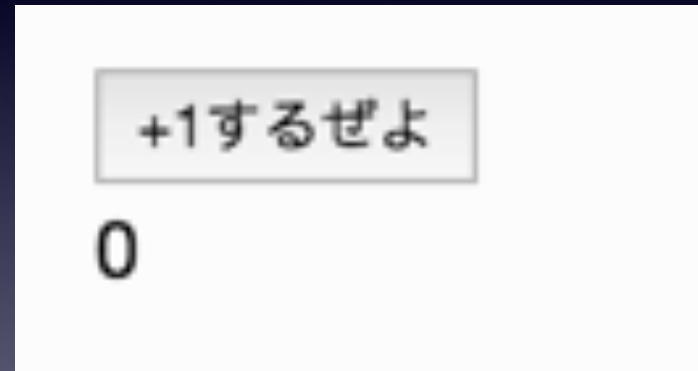
このようにReact.jsでは、いまの画面の状態を考える事無く、常に画面全体を生成していく仕組みになってます。これによりDOMの事を考える必要がなくAjaxやユーザーが起こすイベントをハンドリングする工数を劇的に減らすことができます。

具体的にはどうするのか？

では、具体的にどんな感じで動くのか？

触ってみましょう

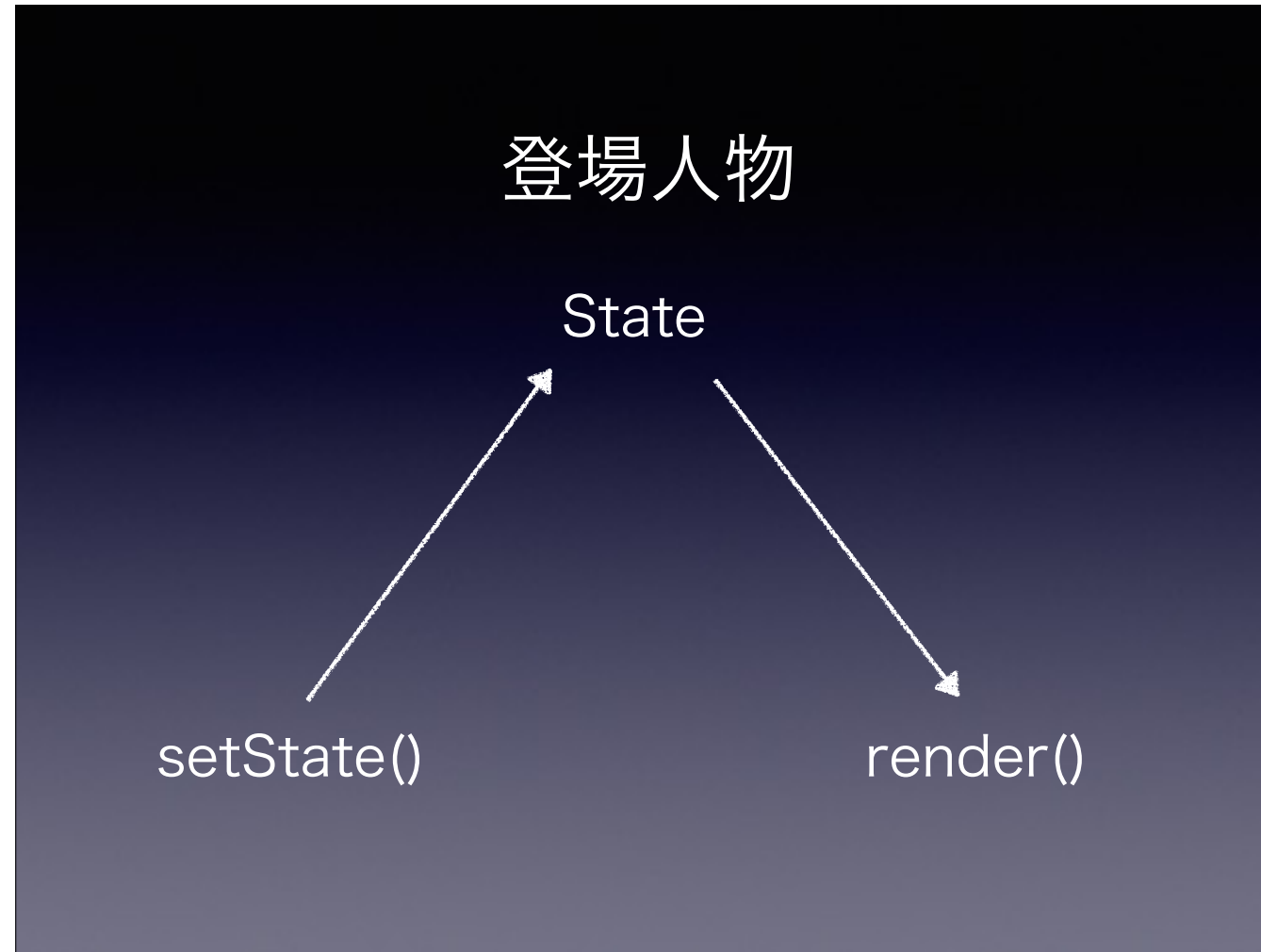
カウントアップ！



では、サンプルアプリを触ってみましょう。

このアプリは、ボタンをクリックすると数字をカウントするだけの簡単なアプリです。

デモします。



このカウントアップの処理には3人の登場人物がでてきます。

setState, State, renderの三つです。

まずState。これは画面を表示するための元となるデータの保管場所です。countってプロパティを持っているオブジェクトです。

次にsetState。これはStateを更新するための関数で、データの更新と画面の再描画の起点となる関数です。

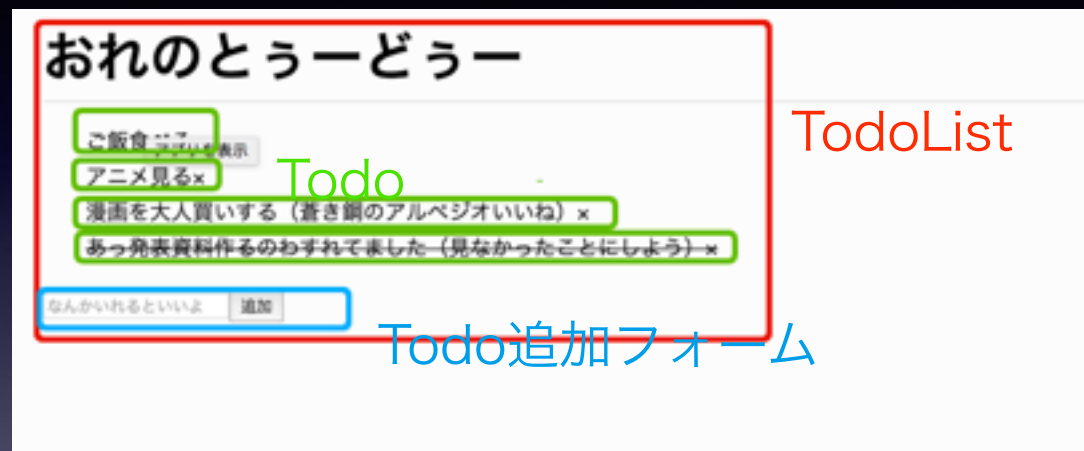
最後にrender。HTMLを生成して描画するための関数で、これは初期表示やデータが更新されるとReactの方から勝手に呼ばれます。私たちはrenderの中身にどういうHTMLを表示したいか書いていきます。

countupボタンが押されると、ボタンクリック時に呼ばれる関数の中でsetState関数が呼ばれます。この関数はStateのcountというプロパティに対して、現在の値+1したものをセットします。

setStateが無事終わると、React側でStateの変更を検知してrender関数が再び呼ばれます。render関数はStateの中のcountというプロパティを読み取って、HTMLをもう一度作り直します。

エンジニアが画面の値を変えたい時は、setStateで保存しているデータを書き換えるだけでOKです。あとはReactが描画をよしなにやってくれます。

応用のサンプル（おれのとうーどうー）



- ・ Todoを表示できる
- ・ 完了したTodoを取り消し線で表現できる
- ・ 新しくTodoを追加できる

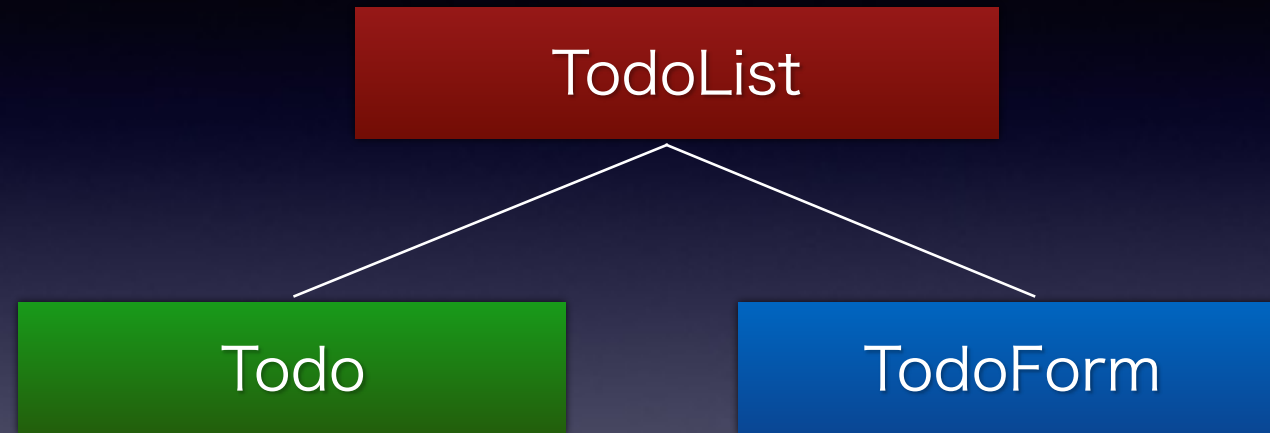
今回用意したサンプルについて説明していきます。

今回用意したサンプルはTodoアプリです。

TOdoの表示、完了したTodoの取り消し、新しいTodoの追加ができるようになっています。

色のついた四角は、このアプリのHTML構造をざっくり模式にしたものです。

HTMLの構成イメージ



わかりやすくするとこんな感じです。

TodoListっていう全体をTodoとTodoFormで構成しているイメージになります。

データと描画する部品を分離

表示に使っている元データ

```
// コンポーネント内でStateの初期値を定義する関数
getInitialState: function() {
  return {
    todos: [
      { value: 'ご飯食べる', done: false },
      { value: 'アニメ見る', done: false },
      { value: '漫画を大人買いする (書き真のアルペジオいいね)', done: false },
      { value: 'あっ発表資料作るのわすれてました (見なかったことにしよう)', done: true }
    ]
  }
},
```

Reactではデータは`state`か`prop`のどちらかの状態で管理される

じゃあ実装を少しだけみてみます。

まずデータと描画する部品を分離した部分。

表示に使っているデータはこんな感じでただのJSONとして宣言されています。

Reactではデータは`state`か`prop`という状態で管理されています。

`state`は変更可能な大元のデータ。画面側のDBみたいなものですね。

`prop`は画面を描画する部分に渡す、変更不可能なデータです。基本的には`prop`を使います。

データと描画する部品を分離

描画に使っているテンプレート

Todo

```
// ToDoListの子コンポーネント。親からもらったデータを、変化させることができないpropsという状態でうけるとる~  
var ToDo = React.createClass({  
  render: function() {  
    var defaultClass = 'callout';  
    ~  
    ~ defaultClass += this.props.done ? ' callout-success' : ' callout-info';  
    ~ return (~  
      ~~~~~<div className={defaultClass}>  
      ~~~~~<i className='ficon ficon-checkmark mark-done' onClick={this.props.onClickDone}></i>  
      ~~~~~<span>{this.props.value}</span>  
      ~~~~~<i className='close' onClick={this.props.onClickClose}>&times;</i>  
      ~~~~~</div>  
    ~~~~);  
  }  
});
```

Reactではこうしたテンプレートを**コンポーネント**という
複数のコンポーネントを作って別のコンポーネントを作る
ことも可能

HTMLの構成イメージ



ここまでの内容を整理してみます。

TodoListというすべての部品の集合体は、Todoを表示するためのデータを書き換え可能な親データ、**state**として持っています。

TodoListは自分の配下のTodoコンポーネントにTodoを表示してもらうために、Todoの情報を、書き換え不能な**prop**という形式で渡します。

TodoFormは親からはデータもらってないですね。そもそも分離してないっすね。

TodoList

```
// ...
var todos = this.state.todos.map(function(todo, index) {
  return (
    <Todo
      key={index}
      value={todo.value}
      done={todo.done}
      onClickClose={this.removeTodo.bind(this, index)}
      onClickDone={this.markTodoDone.bind(this, index)}
    />
  );
}).bind(this);

class TodoList extends React.Component {
  render() {
    return (
      <div className="container">
        <div className="col-xs-6 col-xs-offset-3">
          <h1>TodoList</h1>
          <div>
            <form>
              <div className="form-inline todo-form col-xs-8 col-xs-offset-2">
                <div className="input-group">
                  <input type="text" value={this.state.inputValue} />
                  <button type="button" value="追加" />
                </div>
              </div>
            </form>
            <div>
              {this.props.todos}
            </div>
          </div>
        </div>
      </div>
    );
  }
}
```

・ Todoを4個分作る

・ TodoListはTodoに対して自分のstate
をpropとして渡す

・ 作ったTodo4つ分を埋め込み

・ TodoForm

・ Todo4つ分+TodoForm=TodoList

これがTodoListコンポーネントのレンダリング部分です。

まず上の部分でさっき宣言したTodoコンポーネントを呼び出して、自分のstateの情報をpropとして渡しています。

続いて下半分の赤枠の部分で、TodoList本体をレンダリングしています。

データの流れ



つづいてTDoを完了させる。とかTodoを新しく入力させるときのデータの流れを見ていきます。

すごく大事なことです。画面を書き換えるときは、コンポーネントは直接画面の内容を書き換えることはできないようになっています。

かならず自分や親が持っている、書き換え可能なstateのデータを一回書き換えます。stateを書き換えるとReactがstateが変わったことを自動的に検知して自分や子供のコンポーネントにもう一度データを渡します。それからもう一度HTMLの生成処理が走ります。

ポイント

- ・ データはstateかpropとして保管する
- ・ コンポーネントは自分の持つstateかpropを使ってHTMLを表示する
- ・ 親コンポーネントは子コンポーネントに自分のstateかpropを渡すことができる
- ・ 画面を書き換えたいときは、一回stateを書き換える必要がある。**propは直接書き換えることができない！**

リストへの追加とか、指定した子要素のbool値の変更とかの処理をしています。

あとで興味のある人は見てください。

ポイントはstate以外にもpropっていう値の持ち方をしているところです。

表で比べるフロントエンド

	手軽さ	大規模・複雑	枯れているか
JQuery	◎	×	○
Angular	○	○	△
React + Flux	△	◎	△

以上を踏まえて、よく使われているJSのライブラリとかフレームワークの比較表を作ってみました。

やっぱり手軽さ、検証され尽くした感があるのはjQueryで今後ごく普通のWebアプリケーションを作る場合は全然現役で使われていくと思います。というかそういうAjaxがおまけみたいなアプリにはjQuery使ってください。でもフロントエンドが複雑なアプリケーションでまかりまちがってもjQueryとか使わないでください。保守する人が死にたくなくなります。

React + Fluxっていうとやっぱり大きくて複雑なアプリケーションが向いています。ユーザーイベントがバリバリきて、それで画面とかデータとかが変更されたり、メンテもすごい頻繁にしてPDCA回さないといけないようなアプリケーションに向いています。

Angularはその間で、jQueryほど簡単じゃないけどReact + Fluxよりは1つのフレームワークで完結する分手軽です。その代わりReact + Fluxほど大規模で複雑なアプリケーションに対応するのは得意じゃなさそう、という印象です。今年Angularは2系が正式リリースされてそこからまた地殻変動が起きると思いますが。

Reactおわり

ここまでがReactのお話です。

何か質問ありますか？

たとえばAngularとどう違うねん！みたいなところとかはあんまり喋ってなかったですけど、大丈夫ですか？

AngularもReactもデータと画面の状態を常に同期させておこう、という発想は同じです。同じですが、その実現方法に違いがあります。

Angular.js 1系は2way-bindingといって、Viewの変更はデータに通知されて、データの変更はViewに通知されてという仕組みを自動化することで実現しました。

でもReactは、Viewのデータを直接書き換えることができないようにしました。

これは2way-bindingが次のような批判を受けたからです。

- ・ 値がどこで変えられたのか分からなくてつらい！
- ・ 2way-bindingを実現するには、objectの値変更を監視しなきゃならん！コスト高い！

今話したような批判を回避するために

Flux



Redux



Fluxは設計思想・Reduxはその実装

オブジェクト指向→Java, Ruby, Perl

MVC→Spring, Ruby on Rails, Angular

Flux→Redux

ReduxはFluxから派生した独立の考えだ、っていう見方もあるのですが、今回はReduxも広い意味でFluxという考えの実装である。というスタンスをとります。
ここらへん突っ込んで話していると最強のテキストエディタはSublime かVimかEMacsかみたいなレベルの話になるので脇に置いておきます。

4行でわかるFlux

- データの流れを1方向にするのが目的
- プログラムをView, Action, Dispatcher, Storeの4つの役割で分割する
- この4つをObserverパターンでつなげる*1
- View → Action → Dispatcher → Store → View



*1あるプログラムの部品の状態が変わったとき、状態が変わったことを知らせるためのテクニック。
オブジェクト指向プログラミングのデザインパターンの一種

つまり、FluxとはObserver
パターンだったんだよ！！！！



Facebookがやったのは、Observerパターンを連鎖させる設計に
Fluxと名付けて枠組みをつくったこと

参考) <http://inside.pixiv.net/entry/2015/04/27/170944>

おさらい

フロントエンドをリッチにするための課題

- ・ JavaScriptからの従来のDOM操作は難しい
- ・ 見た目とデータが分離できていない
- ・ データが変更されたときの画面の変更は自分で書かないといけない
- ・ いつどこで誰が持ってるデータが変わって、画面のどこが変わるのか管理しきれてない

さっき、自分は上3つについて簡単に説明しました。

そして、さいごの内容についてはあまり説明しませんでした。

Fluxはこの4つ目を解決するようために作られた仕組みだからです。

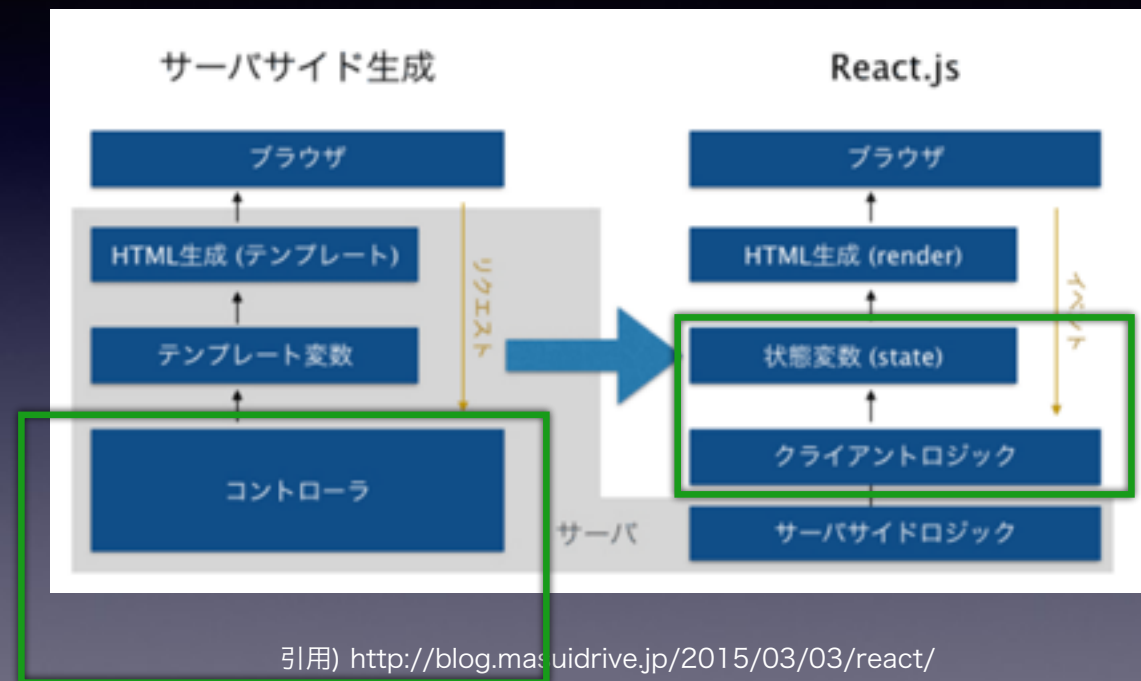
おさらい

フロントエンドをリッチにするための課題

- JavaScriptからの従来のDOM操作は難しい
 - DOM操作の部分はReact.jsがしてくれる
- 見た目とデータが分離できていない
 - データはJSONとして保持・供給される
- データが変更されたときの画面の変更は自分で書かないといけない
 - データが変わるとReactが勝手にrenderメソッドを呼んで再描画してくれる
- いつどこで誰が持ってるデータが変わって、画面のどこが変わるのか管理しきれてない
 - そもそもReactはビューライブラリなので管轄外

一応stateという状態を保持できる部分は持っていますが、本格的な管理には向いてません。

サーバーサイド生成と同じ仕組みをクライアントで 状態管理もクライアントで



ここでもう一回この図に戻りますが、サーバーサイド生成と同じ仕組みをクライアントでやろうとすると、昔はサーバーで管理していたデータ管理をクライアントでやる必要が出てきます

具体例

今回SPAで作ったTodoアプリが管理しなければならない画面側の状態

- Todoの内容や数、完了状態
- Todoの表示条件（全部、未完了、完了）
- どのページを見ているか（表示しているページとURL）
- Todo入力コンポーネントが開いているか閉じているか
- Todo入力フォームに入っているデータの内容
- どのアニメーションを使って遷移するか

単純なアプリでも、管理しなければならない状態はたくさん。

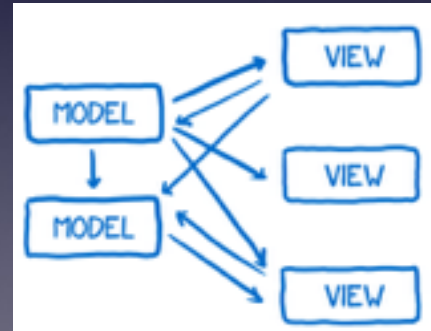
今回ハイブリッドアプリのサンプルとして作ったTodoアプリを例に説明してみます。

スライドには、今回のTodoアプリが管理しなければならない画面側の状態を全部書いています。

細かい説明はしませんが、まあサーバーサイドで管理するより増えて複雑になった、ぐらいに思っていてください。

なぜFluxが必要だったのか？

- ・ Facebookはコードがでかくなりすぎた
- ・ 1行直すと3つバグるハウルの動く城ができあがっていた(らしい)
- ・ しかもどのコードがどういう順序で状態や画面を書き換えているのかわからないことが原因



参考) <https://medium.com/@sotayamashita/%E6%BC%AB%E7%94%BB%E3%81%A7%E8%AA%AC%E6%98%8E%E3%81%99%E3%82%8B-flux-1a219e50232b#.mlqdh9tkg>

こうしたデータをブラウザなどのJSで管理しなければならない、という前提を含めて、FacebookがFluxを作った理由を簡単に説明します。

たとえば、あるバグを直そうとしたとします。変更の結果、ユーザー操作で起きたイベントであるデータ1個を書き換える。それだけを意図したはずが、そのデータは今度は別の画面やデータを書き換えて、書き換えられた画面やデータがそのまた次の・・・といった玉突き連鎖でエンジニアが想定していないところでバグが発生し続ける、といった悪循環が起きていたらしいです。

Facebookはこれを抜本的に改善したかった。

双方向にデータのやり取りが あるからわかりにくい

Facebookのちょう頭いい人たちは、これらの問題の諸悪の根源を次のように結論付けました。

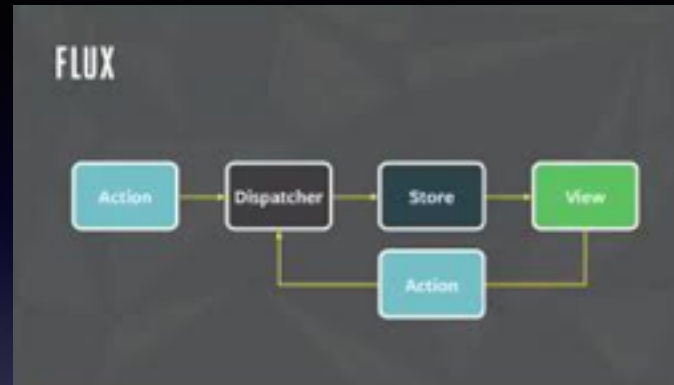
双方向にデータのやり取りがあるから処理が追いにくいんだ。

データの流を一方向にしちやえばどこ
で何を書き換えたかわかりやすいしトレー
スしやすいよね！

じゃあ、データの流を一方向にしちやえばどこで何を書き換えたかわかりやすいしトレースしやすいよね

そんなわけでデータや処理の流を強制的に一方こうにするFluxという考え方が生まれました。

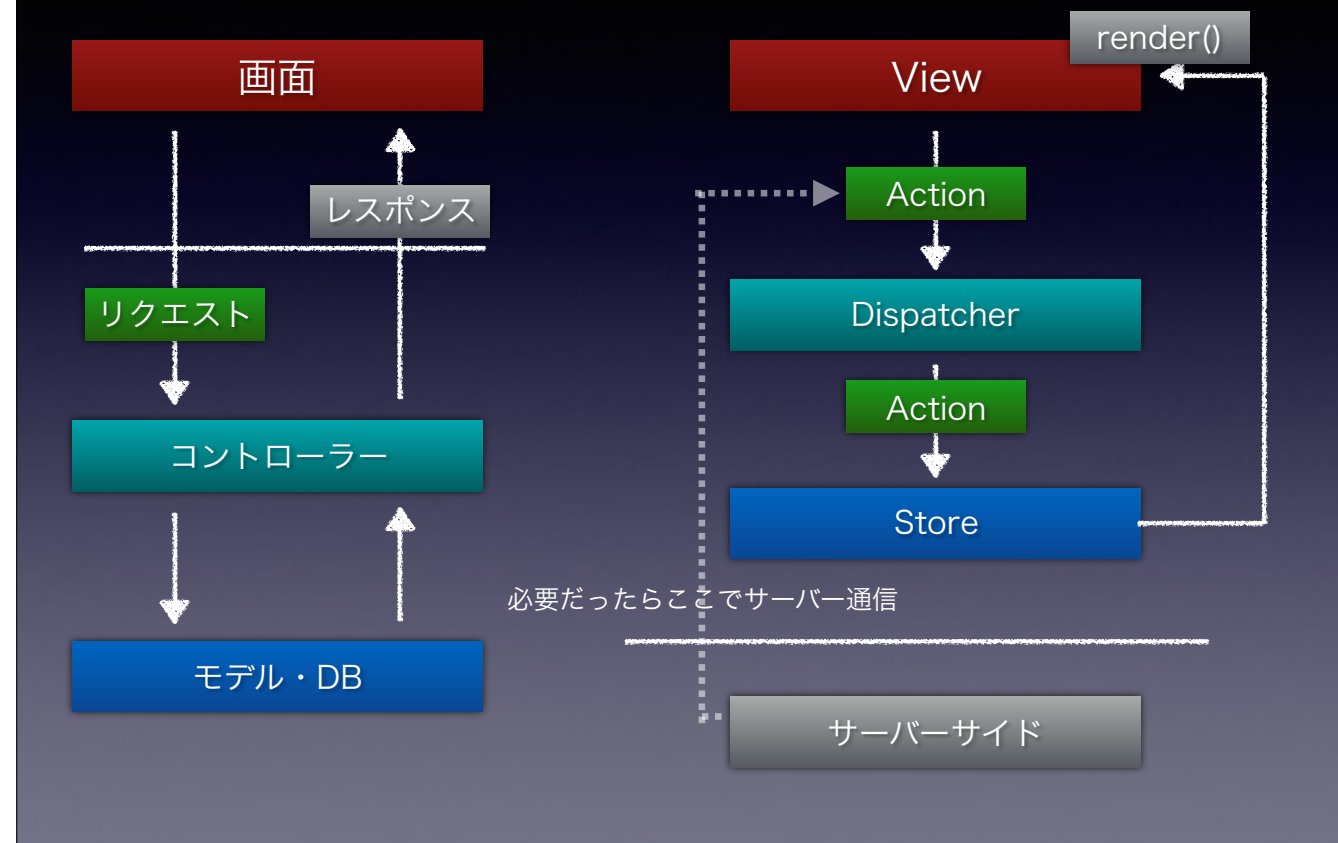
各部品の役割分担



- View → 渡されるデータに基づいて画面を描画(React.jsが担当)
Storeの変化をlistenしている。
- Action → アプリケーションの状態の変更や画面の変更時に発火するイベント
とどんな変更が起きるかの内容
- Dispatcher → Actionが発行されたかをStoreに伝える。
どのStoreに伝えるべきかはDispatcherが知っている。
- Store → データや状態の保管場所。Dispatcherからデータを受け取ると、
自分のデータを書き換えて、Viewに「データが変わったから
再取得して再描画オナシャス」する

書いたんですけどわかりにくいで飛ばします。

MVCとFluxのざっくり比較



サーバーサイドで作るアプリケーションと似通っているとも多いので比較しながら説明していきます。
ECサイトでカートに商品を追加するケースを想定してみます。

お客さんがカートに追加、って押すとカートのアイコンのところに数字が出てくるようなイメージです。
ふつうのWebアプリケーションだと、カートに追加、ってぽちっと押すとリクエストがサーバーに投げられて、それをコントローラーが受け取ってモデル・DBの部分で永続化やら何やらして、結果を一度コントローラーに戻してからレスポンスをカートのアイコンの部分に数字が現れているHTMLで返すと思います。

Fluxの場合はカートに追加、ってぽちっと押すとActionというものが作られます。これはリクエストに相当するものですね。ViewはDispatcherにActionを渡します。Dispatcherはコントローラーに近い役割を果たしています。

Dispatcherはアクションの中身を判断して、モデルやDBと同じようにデータの管理を担当しているStoreという部品にActionがきましたよ、ということを知らせます。

StoreはActionを受け取って中身を判別して、自分が持っているデータを更新します。
更新が終わると、Viewに対して更新したからデータを取りに来てください、ということを知らせます。

ViewはReact.jsが担当しているのですが、Storeから通知を受け取ると、データを再取得してrenderメソッドを呼び出してHTMLを生成します。

もっと簡単をお願いします

Viewさん：「カート追加したいんでAction発行します」

Dispatcherさん：「カート追加したいってAction来たんで、追加内容を新しくカートデータの保管庫に追加してください」

カート担当のStoreさん：「言われた通りカートデータを追加したから、Viewさんはデータを取って再描画してください」

Reduxとは

- Fluxの考え方を実装したフレームワークです
- Flux実装の中では本家Facebookのものを抜いて現在一番人気
- 登場人物はAction, Reducer, Store, View
- Dispatcherはオブジェクトからただのメソッドに格下げ
- Actionは全てReducerが受け取る。受け取った内容を解釈してStoreを更新するのがReducerの仕事

参考) https://developers.eure.jp/tech/redux_feature/

以上を踏まえて、Reduxってどういう実装かを説明します。

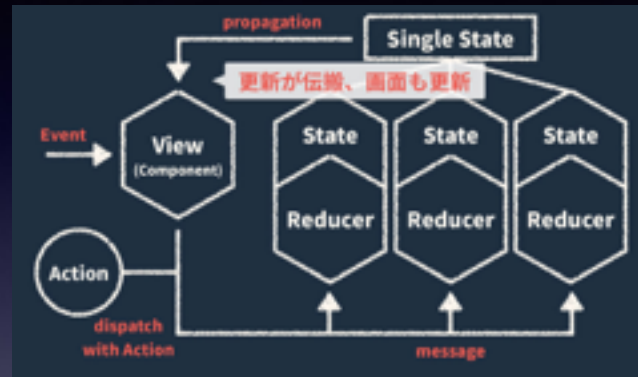
ReduxというのはFluxの考え方を実装に落とし込んだフレームワークです。

Flux実装の中では本家Facebookのものを抜いて現在一番人気です。

本家Fluxと異なるのは登場人物です。登場人物はAction, Reducer, Store, Viewの4つ。Dispatcherはオブジェクトからただのメソッドに格下げされています。

ReducerはdispatchされてきたActionを受け取って、その内容を解釈してStoreを更新する役割を持っています。

Reduxもデータの流れは一方方向




- Viewがイベントに応じてActionを作るよ
- **dispatch**メソッドでReducerにActionの発行を知らせるよ
- **Reducer**は受け取ったActionをもとにStoreの状態を変えるよ
- Storeは変更があったらViewに変更を伝えるよ

引用) <https://speakerdeck.com/axross/introduction-to-redux>

Fluxと違うのは赤字の部分です。

データの流れがView Reducer Stateの書き換え renderってなることだけ覚えて帰ってください。



実際に動かしてみます

実際に動かしてみますが、たぶんふーんで終わります。

今言った順番で処理が動く、ってことだけ感じてもらえればOKです。

実際使ってみて感じたメリット

- ・ どこで入力が起きたら次はどこにデータが流れるか決まっているので、問題の認識や切り分けが楽
- ・ Fluxという枠組みの中で、どこにどんな処理を書くべきかが整理されているため、あんま悩まなくて済む

実際使ってみて感じたデメリット

- ・書いているとやっぱり冗長になるケースがある（コンポーネントのある部分の表示非表示のトグル処理を書くためにFluxを一周書くのはキツイ）
- ・2way-bindingのほうが書くコードは少なくて済む
- ・JSの荒野から必要でメンテされててメジャーなライブラリを自分で選定しないといけない

React + Fluxが向いているもの

- ・ 大規模なSPAの作成
- ・ 画面側のあらゆるところでイベントが発生するアプリ
- ・ リアルタイムにサーバーとやりとりしてその結果を表示するようなアプリ

要するに大きくて複雑なフロントエンド機能を持ったアプリを破綻なく作ることに向いている

質問タイム

Material DesignとかCSS Moduleとかデザイン実装関連の話もどうぞ

最後に

フロントエンドは戦国時代

技術の変化が凄く早いし 1 年
前に流行った技術が廃れてる
なんてこともある

でも、その歴史は繋がっているし、考え方がただただ捨てられているわけではないし、
現在に影響を与えている

その技術がなぜ出来たのか、何を解決するのか正しく理解した上で、今必要な技術を選択できるといいですね！