

コンセプトから学び始める

# Redux入門

そもそもRedux

って、なんだっけ？

# Reduxの役割

- ✓ モダンな開発では必須のフレームワーク
- ✓ SPA<sup>※</sup>などのリッチなUIを作る技術の登場に伴い、フロント側の状態管理が必要に
- ✓ グローバルなState管理を担当

# ReduxがStateを管理？

ReactのStateが肥大化・複雑化

→Webアプリケーションにおける、

グローバルな状態管理の構造や設計が必要に

リッチなUIの実現

→ 多くのStateを

フロント側で持ち、

素早く出したい

例)チャットアプリ

ログイン情報の管理

メンバーの一覧

メンバーの詳細情報

etc...

# 企業での使われ方

- 会社規模の複雑性を持つアプリであればグローバルのStateを管理する仕組みが必要
- Reactを採用している企業の多くで採用されている。

→Reactを使うフロントエンドエンジニアであれば知っておきたい

つまり

＜アーキテクチャ＞

**Reduxとは設計思想**

# アーキテクチャとは？

## 原型 : Fluxアーキテクチャ

Facebookが提唱。データの流を一方向に限定し  
データ状態の複雑さを軽減させようとする設計思想



**React** → **Redux**

**Vue** → **Vuex**

Fluxの概念から派生。  
Reactで管理する状態を  
ひとまとめにしてRedux  
で管理することに

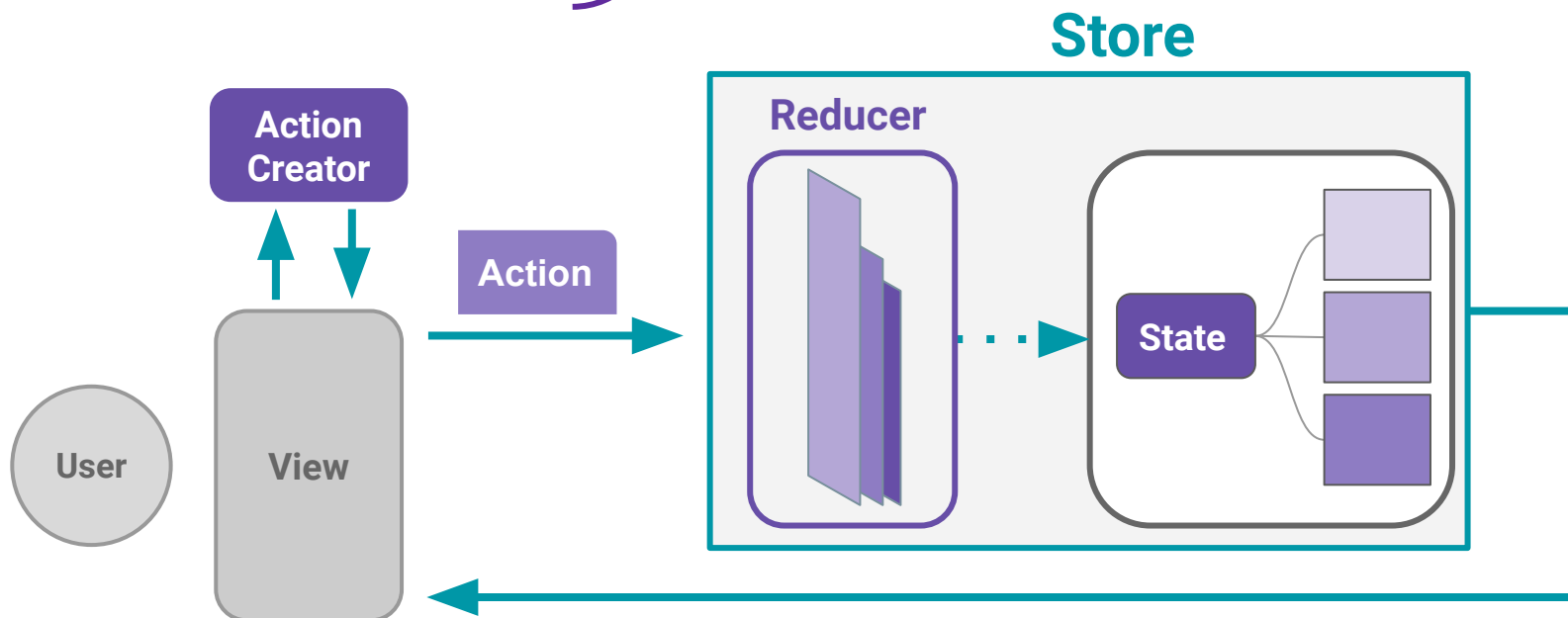
# Reduxの構造

✓ Action

✓ Store

✓ Reducer

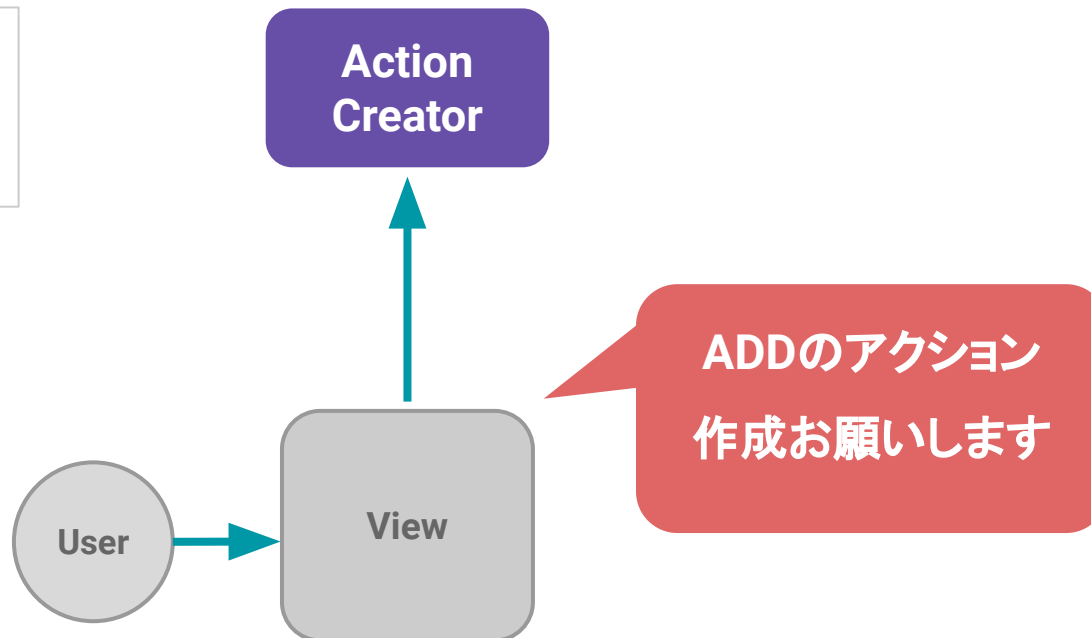
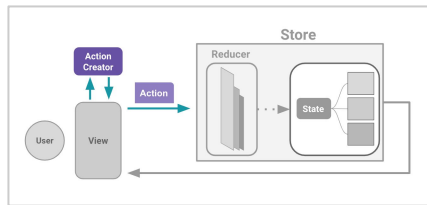
これらの3つを使い、  
データフローの  
一方向性を実現させる





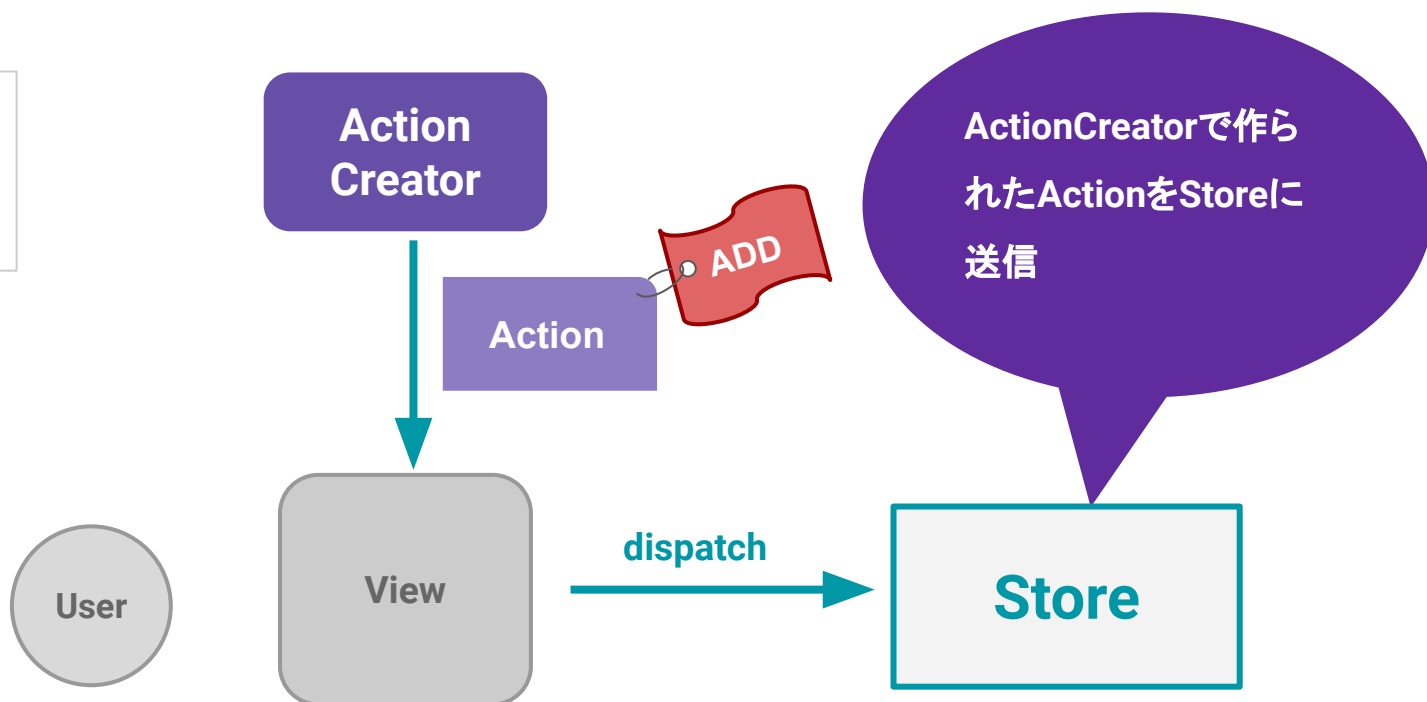
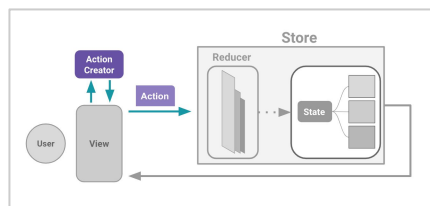
# Action

Stateの変更が必要なユーザーの入力が発生した場合に、ActionCreatorメソッドに入力内容が渡されActionオブジェクトを生成



# Action

Stateの変更が必要なユーザーの入力が発生した場合に、ActionCreatorメソッドに入力内容が渡されActionオブジェクトを生成



# Action

何のためのActionなのか、ラベル(Type)をつけて返す

```
// Action Type:
const ADD_TODO = 'ADD_TODO'
const REMOVE_TODO = 'REMOVE_TODO'

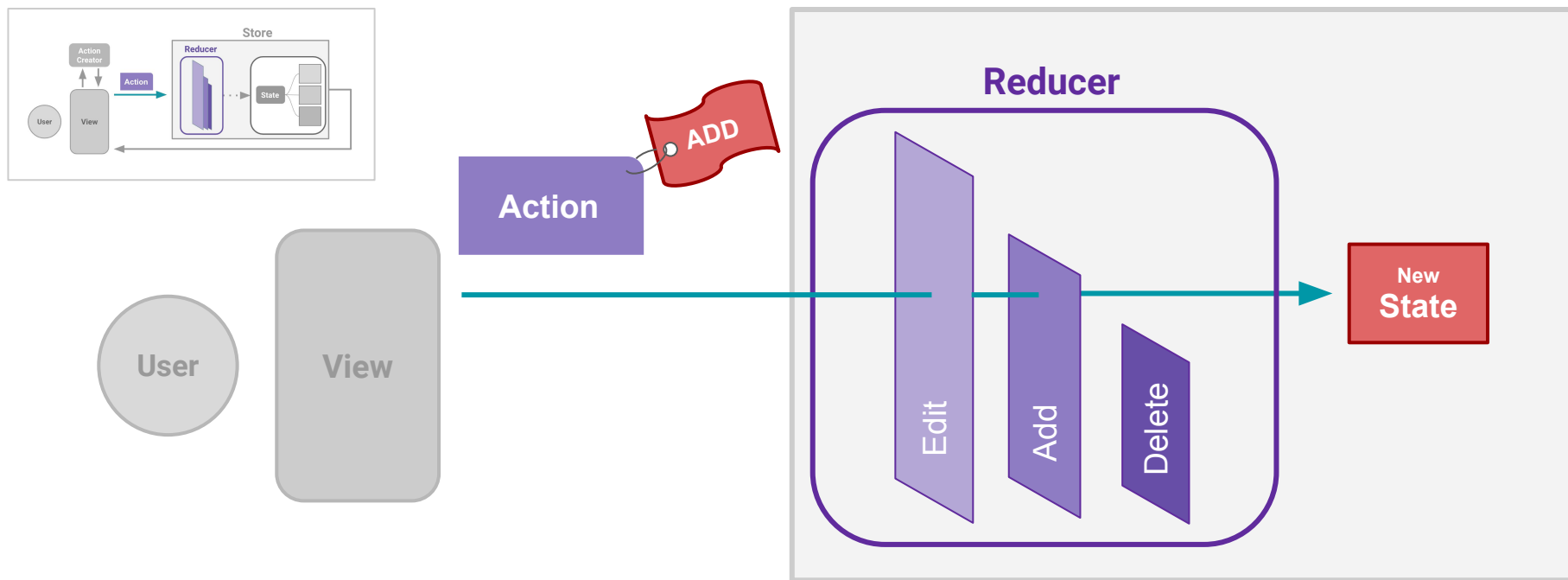
// Action Creators:
function addTodo (todo) {
  return {type: 'ADD_TODO', payload: todo };
}

function removeTodo (todo) {
  return {type: 'REMOVE_TODO', payload: todo };
}
```

生成されるActionオブジェクト

# Reducer

発行(Dispatch)されたActionを識別し、  
Reducer内の対応するメソッドを使って新しい  
Stateを生成



# Reducerの例

```
const initialData = {
  1: { id: 1, text: "brush a tooth"},
  2: { id: 2, text: "wash a face"},
  3: { id: 3, text: "drink a coffee"},
}

const initialState = {
  index: Object.values(initialData).length,
  data: initialData
}
```

```
const reducer = (state = initialState, action) => {
  let newState = {...state}
  const { id, text } = action.payload || {}
  switch (action.type) {
    case ADD_TODO:
      newState.index++
      newState.data[newState.index] = { id: newState.index, text };
      return newState
    case REMOVE_TODO:
      let newData = {...newState.data}
      delete newData[id]
      return {
        ...newState,
        data: newData
      }
    default:
      return state;
  }
}
```

**Actionのtypeを元に、  
必要なStateを新規作成**

## 【ルール】

- ・必ずイミュータブルに保つ
- ・副作用を起こさない pureな関数に保つ
- ・オブジェクト自体の書き換えはしない

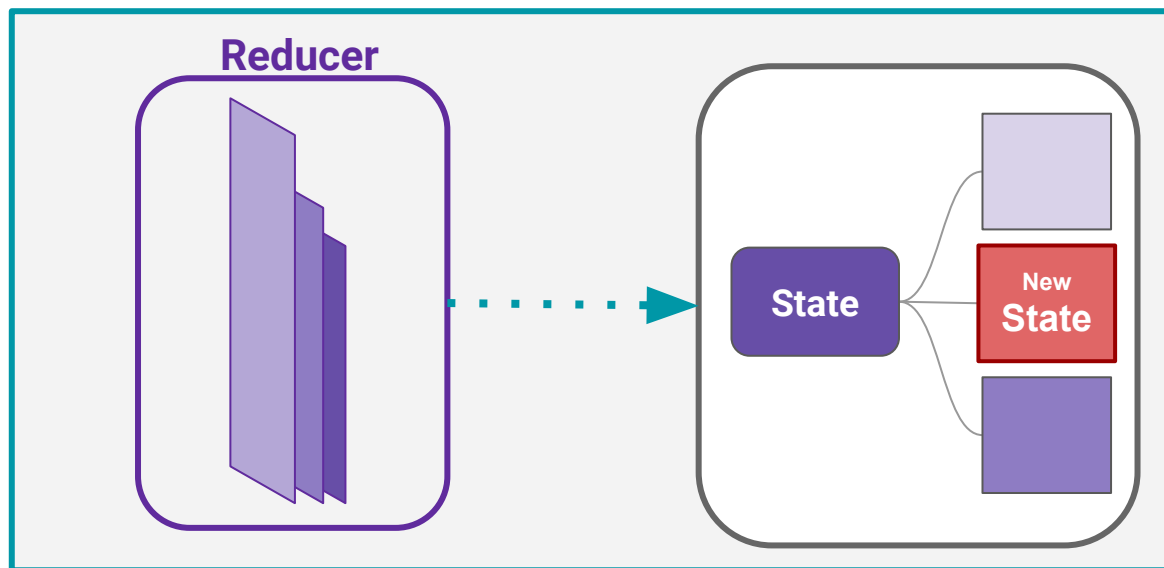
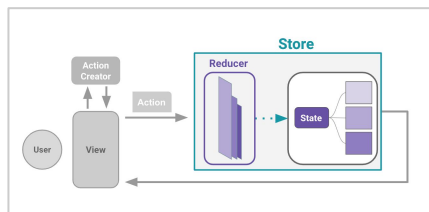
# Store

※グローバルに保持する必要のないState、簡単なStateはReactをそのまま使う場合も

アプリケーション内で1つ存在し、グローバルで管理が必要なStateを保持

Reducerから生成されたStateに置き換えられる

## Store



# Storeの作成方法

作成したreducerを元に、  
createStoreメソッドで作成する

```
var store = Redux.createStore(reducer);
```

```
const store = Redux.createStore(  
  combineReducers({  
    user: userReducer,  
    content: contentReducer  
  })  
);
```

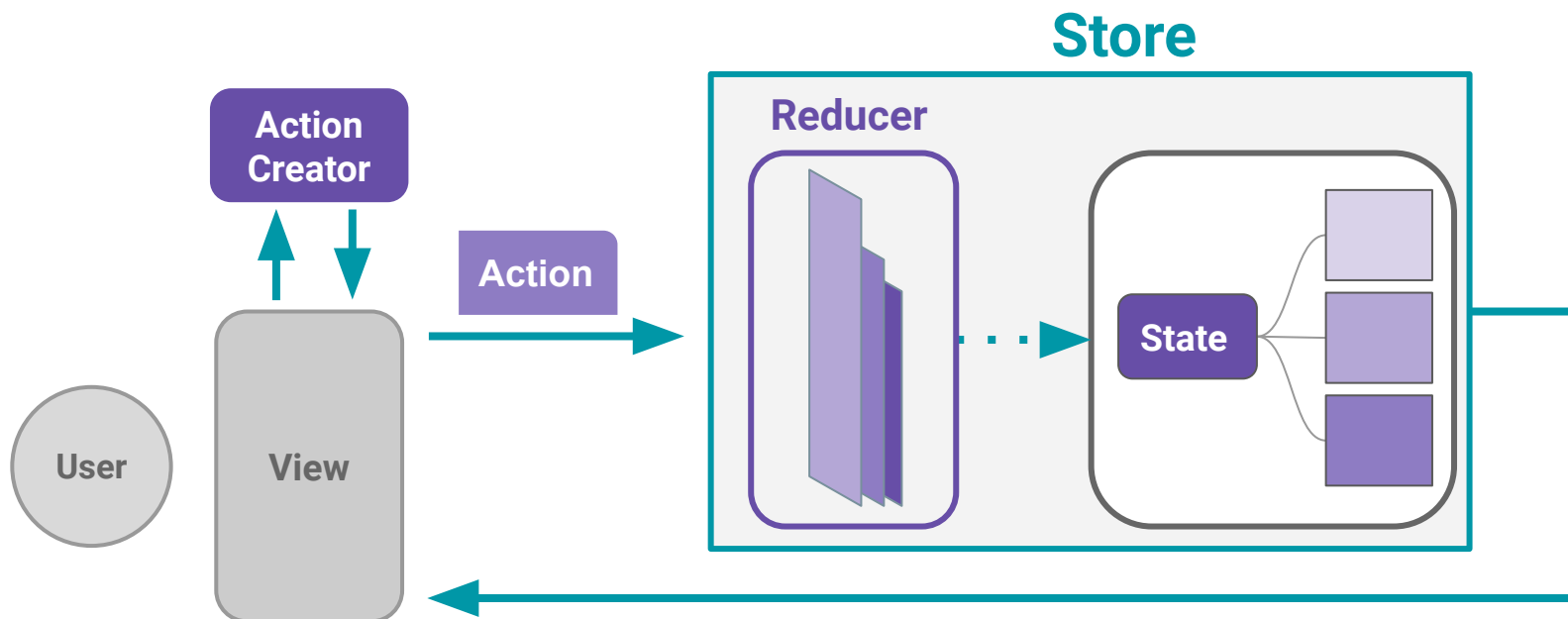
アプリケーション／Storeを構成する  
オブジェクトの構造が複雑になった  
場合、Reducerを複数の関数に分  
割、Stateを分けて処理することも可  
能

# Redux導入のコンセプト

データフローを一方向にし、単純化。Reactとの役割を分割

React : View担当

Redux : Stateの管理・変更担当





でもけっきょく

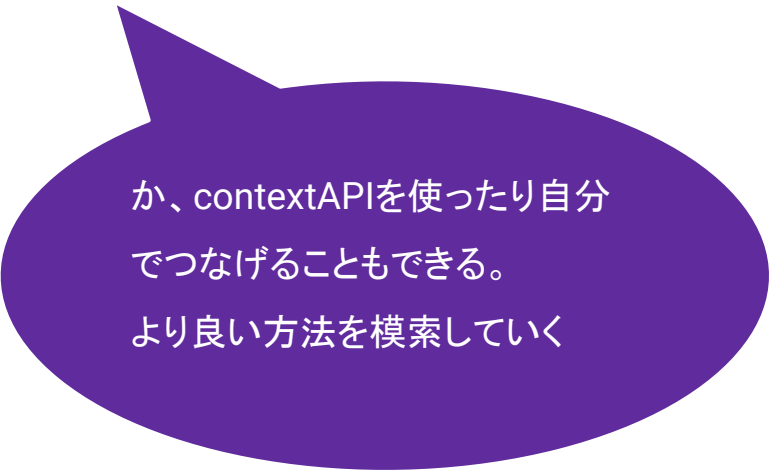
どうやってReactと繋げるの？

**Reduxは独立しているので  
専用のモジュールを使う**

# react-reduxライブラリの導入

# react-reduxライブラリ

ReactとReduxを結びつける定番パターンのライブラリ。コンテナーを使ってReduxのStoreとconnectしてあげる



か、contextAPIを使ったり自分でつなげることもできる。  
より良い方法を模索していく

# コンテナー(container)

- ✓ ReduxのStoreが管理するState
- ✓ Stateを変更するためのAction(Dispatch)

上記2つを、Reactコンポーネントにプロパティ  
(props)として流し込めるようにするための箱

# Reduxと連携① コンテナを作る

container/AppContainer.js

```
import React from 'react'
import { connect } from 'react-redux'

import App from '../component/app'
import { add } from '../action/app'
```

```
function mapStateToProps(state) {
  return {
    data: state.data
  };
}

function mapDispatchToProps(dispatch) {
  return {
    handleClick: () => { dispatch(add()) }
  }
}
```

```
export default connect(mapStateToProps, mapDispatchToProps)(App)
```

(1) ReactコンポーネントAppとReduxを繋ぎたい

(2) AppにPropsとして流したい、  
必要なStateとActionを名前をつけて指定(準備)  
mapStateToProps: StoreのどのStateが必要か  
mapDispatchToProps: どのActionを使うか  
(dispatchするか)

(3) connectでAppと繋げ、Propsとして使えるように

Connectメソッドはコンテナを作成する  
→今回はファイル名「AppContainer」  
というコンテナが生成された

# Reduxと連携② Storeの作成と連携

index.js

```
import React from 'react'
import { render } from 'react-dom'
import { createStore } from 'redux'
import { Provider } from 'react-redux'
```

```
import AppContainer from './container/AppContainer'
import reducer from './reducer'
```

(0) コンテナをインポート

```
const store = createStore(reducer)
```

(1) 作成・インポートしたReducerを元に、  
Storeを作成(createStoreメソッドを使用)

```
render(
  <Provider store={store}>
    <AppContainer />
  </Provider>,
  document.getElementById('root')
)
```

(2) react-reduxのProviderにstoreをPropsとして渡す

(3) 作成したContainerを設置し、StoreとAppを繋ぐ

# Reduxと連携② Storeの作成と連携

index.js

```
import React from 'react'
import { render } from 'react-dom'
import { createStore } from 'redux'
import { Provider } from 'react-redux'
```

```
import AppContainer from './container/AppContainer'
import reducer from './reducer'
```

(0) コンテナをインポート

```
const store = createStore(reducer)
```

(1) 作成・インポートしたReducerを元に、  
Storeを作成(createStoreメソッドを使用)

```
render(
  <Provider store={store}>
    <AppContainer />
  </Provider>,
  document.getElementById('root')
)
```

(2) react-reduxのProviderにstoreをPropsとして渡す

(3) 作成したContainer

連携完了!



# ※コンポーネント内での使い方

App.js

```
import React from 'react'

export default class App extends React.Component {
  render() {
    return (
      <div>
        <span>{this.props.data}</span>
        <button onClick={ () => this.props.handleClick() }>Add</button>
      </div>
    )
  }
}
```

Containerで指定・命名したstateとdispatchをpropsとして使う

StateもStateの書き換えも  
Redux側での動作になるため、  
ReactはViewに特化できるように

ね、簡単でしょ？

Reduxって  
ファイルを分けたり  
結構複雑

なので代替手段が  
生まれつつあります

# Reduxの今後のな

✓ recoil

✓ reactn

などもう少し手軽な状態管理ライブラリがあるので  
使わなくなる可能性も

ReactのHooksにも  
Reduxと同様の機能あり

まとめ

# Reduxはアーキテクチャ

データフローの一方方向性

グローバルな状態の管理

フロントエンドを目指す人、  
自作アプリを作る人、  
ポートフォリオを作る人、



**モダンな開発をする人なら  
知っておこう**

ありがとうございました