

TypeScriptで Redux + React チュートリアル

TypeScript + Redux + Reactでミニマムなアプリを作っていきます。題材は例の如く、**Todo アプリ**です。**Create React App**を使い、Reactアプリの土台を作っていきます。

もしReduxが初めての場合、**React + Reduxアプリ**の最初の設計を整えるのはちょっと**労力が必要**ですが、メンテナンスしていく中で壊れにくいアプリを作れるなどそれに見合うだけの価値はあると思います。私も一番始めに、Reduxに触れた時には、表面上の複雑さに困惑しましたが、とりあえず自分で作ったコードを改変して、色々試してみるのが一番早いのかな、と思っています。

では、始めましょう。

Node.js 8系以上と yarnを使いますので、持っていない場合は必要なツールを揃えます。また、現時点で、TypeScriptを書くための個人的なおすすめのエディタは[Visual Studio Code](#)です。無料で入手できますので、エディタに拘りが無い場合には使ってみても良いと思います。

- [Node.js のインストール](#)
- [Yarn のインストール](#)
- [Chrome ウェブストアで拡張機能 Redux Dev Toolsのインストール & Chromeの再起動](#)
- [Visual Studio Codeのインストール \(任意\)](#)

このチュートリアルは下記の環境で動作させています。

- * OS: Mac OS X 10.13.4
- * Node.js: 8.11.1
- * Yarn: 1.5.1

また、今回作ったミニマムなサンプルは下記からダウンロードできます。

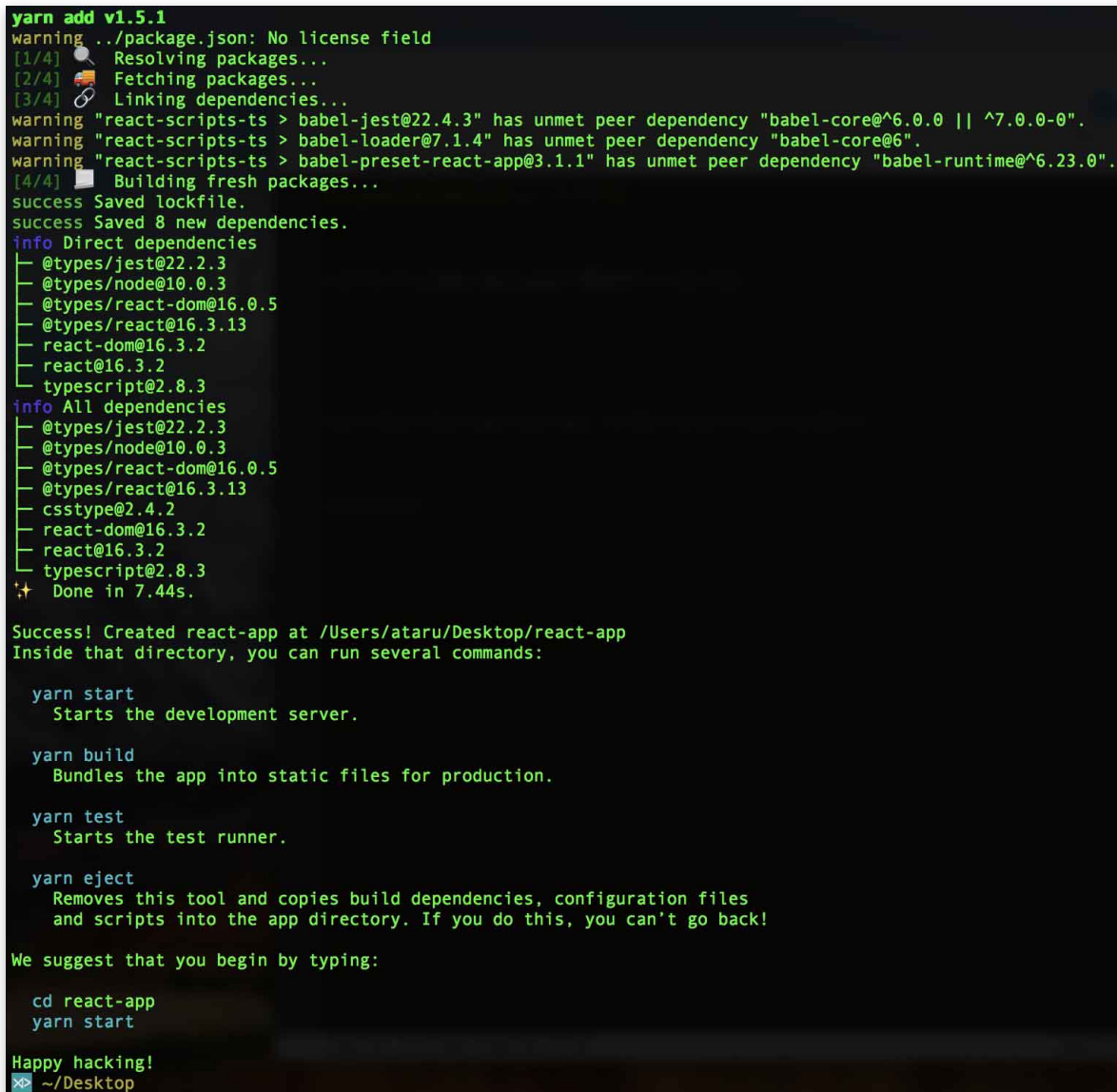
- [1段階目 \(Taskの追加機能のみ\)](#)
- [2段階目 \(Taskの追加・削除・完了更新機能有\)](#)

👉 [Create React AppsでReactアプリの雛形をつくる](#) 🍷

下記のコマンドをTerminalで叩いて、TypeScriptでのReactアプリの雛形を作ります。

TypeScriptで最小限なReactアプリを作成

```
npx create-react-app react-app --scripts-version=react-scripts-ts
```



```
yarn add v1.5.1
warning ../package.json: No license field
[1/4] Resolving packages...
[2/4] Fetching packages...
[3/4] Linking dependencies...
warning "react-scripts-ts > babel-jest@22.4.3" has unmet peer dependency "babel-core@^6.0.0 || ^7.0.0-0".
warning "react-scripts-ts > babel-loader@7.1.4" has unmet peer dependency "babel-core@6".
warning "react-scripts-ts > babel-preset-react-app@3.1.1" has unmet peer dependency "babel-runtime@^6.23.0".
[4/4] Building fresh packages...
success Saved lockfile.
success Saved 8 new dependencies.
info Direct dependencies
├─ @types/jest@22.2.3
├─ @types/node@10.0.3
├─ @types/react-dom@16.0.5
├─ @types/react@16.3.13
├─ react-dom@16.3.2
├─ react@16.3.2
└─ typescript@2.8.3
info All dependencies
├─ @types/jest@22.2.3
├─ @types/node@10.0.3
├─ @types/react-dom@16.0.5
├─ @types/react@16.3.13
├─ csstype@2.4.2
├─ react-dom@16.3.2
├─ react@16.3.2
└─ typescript@2.8.3
✨ Done in 7.44s.

Success! Created react-app at /Users/ataru/Desktop/react-app
Inside that directory, you can run several commands:

  yarn start
    Starts the development server.

  yarn build
    Bundles the app into static files for production.

  yarn test
    Starts the test runner.

  yarn eject
    Removes this tool and copies build dependencies, configuration files
    and scripts into the app directory. If you do this, you can't go back!

We suggest that you begin by typing:

  cd react-app
  yarn start

Happy hacking!
~/Desktop
```

ディレクトリの中に移動して、コマンドを入力します。

```
cd react-app
```

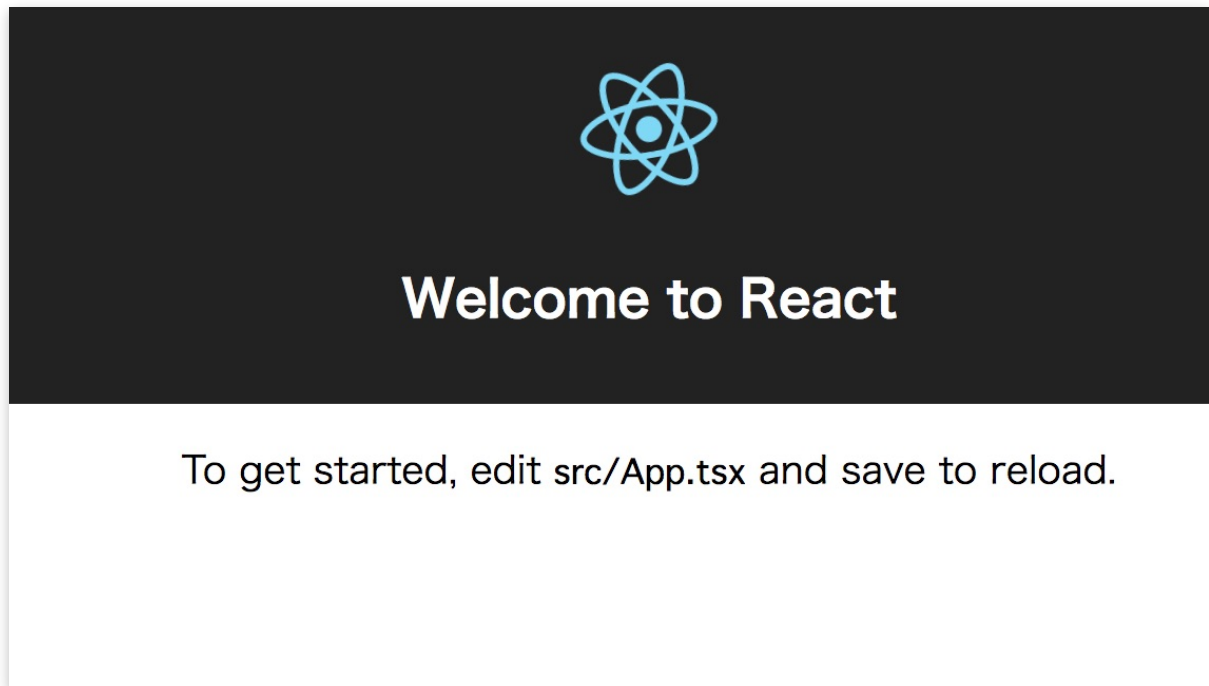
Reactアプリをejectする

```
yarn run eject
```

Yes or Noと聞かれるので、Yを入力します。そして、次に下記コマンドを入力します。

```
yarn start
```

これでReactアプリが起動しますので、念のため、動いているところを見ておきましょう。



必要ライブラリのインストール 🍰

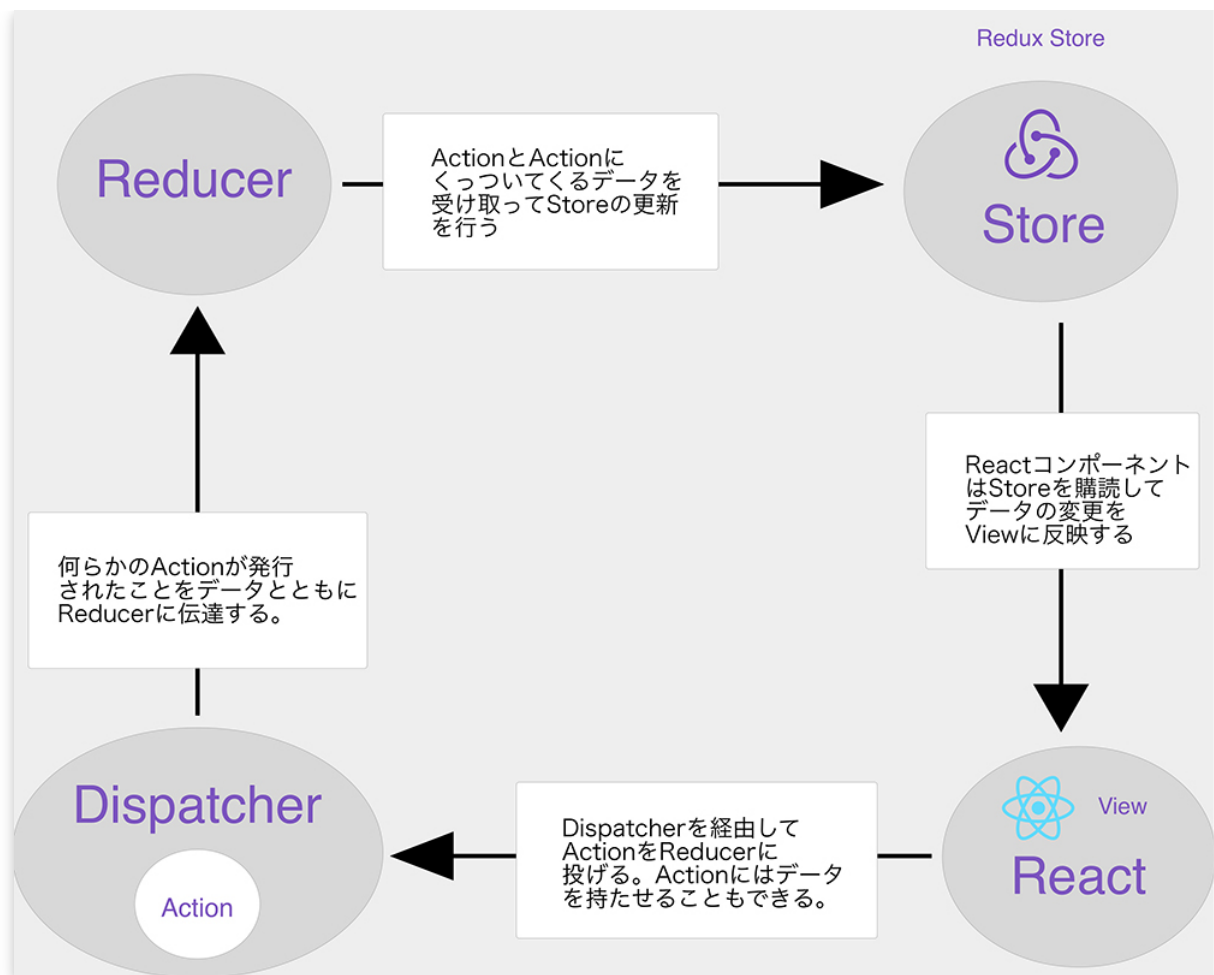
Reduxなど必要なライブラリーを入れる。

```
yarn add  redux react-redux redux-devtools-extension typescript-fsa typescript-fsa-red  
yarn add -D @types/redux @types/react-redux
```

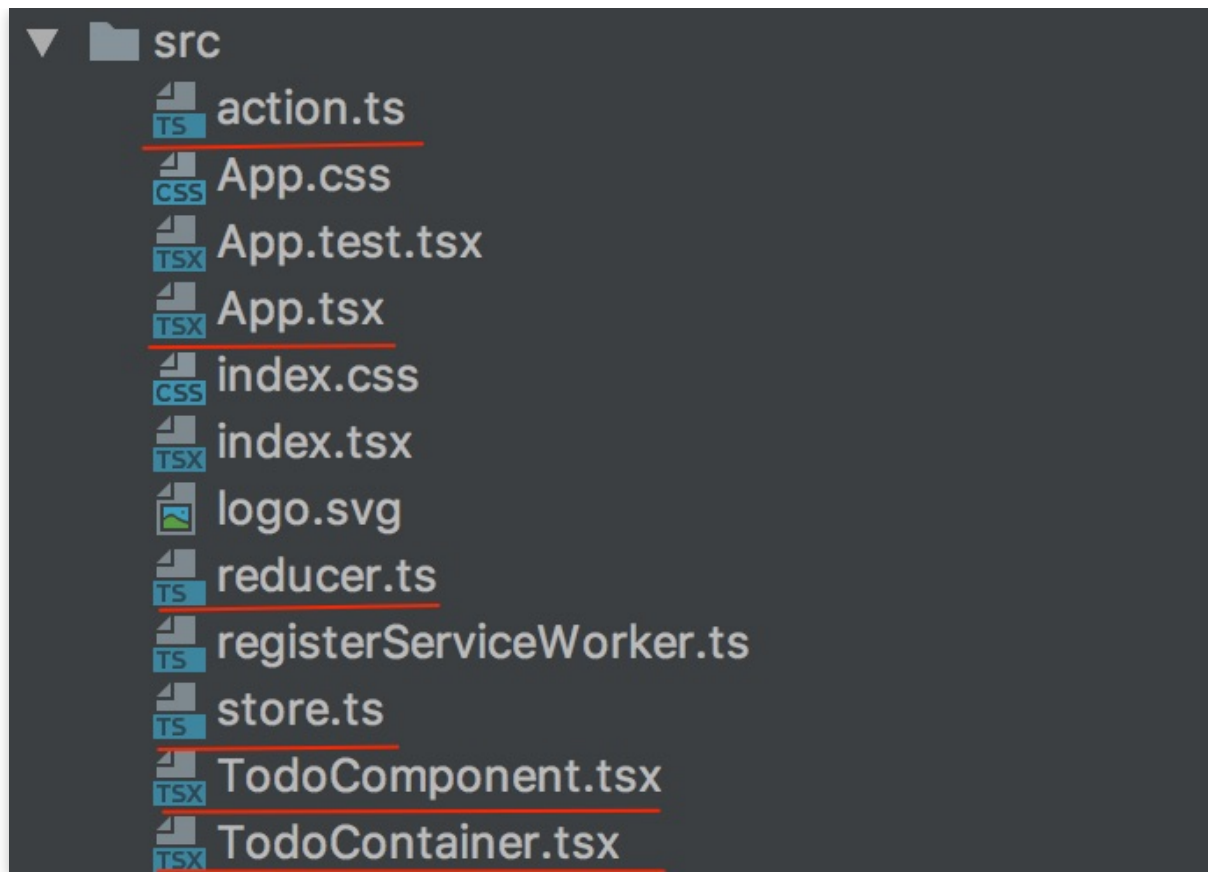
少しだけ、ライブラリの説明をします。

- **redux**: Reactのメジャーなステートマネージャーライブラリです。
- **redux-devtools-extension**: ReduxのStateの状態やアクションの履歴をGUIでわかりやすく把握することのできるブラウザ拡張機能のためのライブラリです。
- **typescript-fsa, typescript-fsa-reducers** : TypeScript用に型を指定してアクションを発行したり、そのアクションをReducer側で受け取ったりできます。

まず全体像を把握しておきます。他のところで解説されている図式と若干違うかもしれませんが、こちらの方が解りやすいと思い、下記のように図式化してます。ちょっと歪んでいる汚い図ですが、ご寛恕ください。



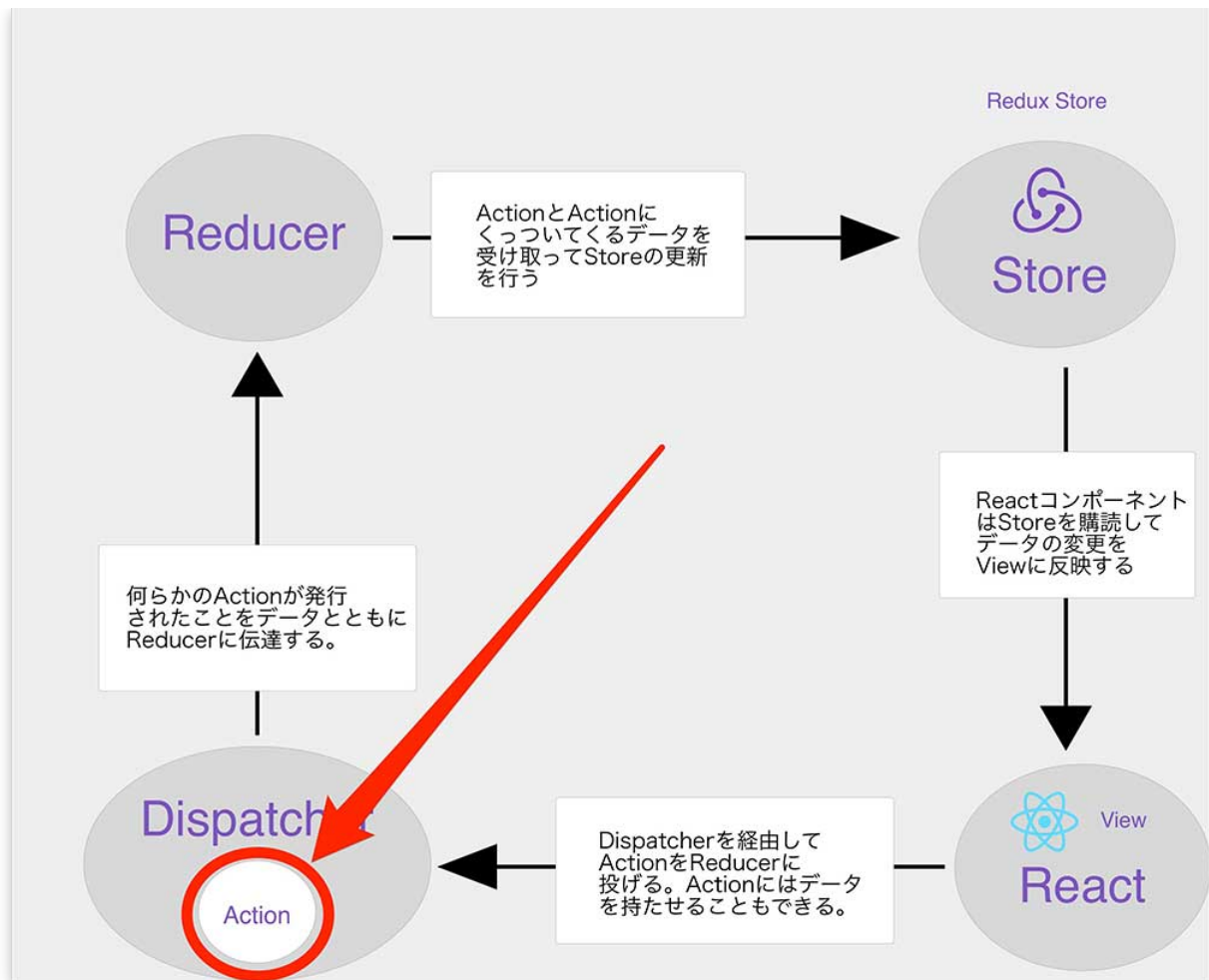
ではこれから、最小限のReduxアプリを作っていきます。まずは追加の機能しか持たない **Todoアプリ** です。これから、**srcディレクトリ** 内部の下記赤線を引いたファイル群を作成もしくは編集していきます。



最初に **Action** の作成から取り掛かっていきましょう。

② Actionをつくる 🏠

Actionをつくりましょう。作成するのは全体図の中でのこの部分です。



ではファイル **src/actions.ts** を作成します。

```
/* src/actions.ts を作成 */
```

```
import actionCreatorFactory from 'typescript-fsa';
const actionCreator = actionCreatorFactory();
```

```
/* actionCreator()で型指定しながらActionをつくる。 string型のpayload (データ) を伴
export const addTodo = actionCreator<string>(
  'ADD_TODO',
);
```

Action は **Dispatcher** (あとから出てきます) を経由して呼び出される、ただのJavaScriptオブジェクトです。下記のような形式を通常は持ちます。

```
{
  type: 'ADD_LANGUAGE',
```

```
    payload: 'Java'
  }
```

Action の主流の形式に関しては **Flux Standard Actions** が詳しいので、興味のある方は覗いてみてください。 **src/actions.ts** ではライブラリの **typescript-fsa** から提供される **actionCreator()** という関数を使って、下記の **Action** を作っています。 **actionCreator** は手軽に **Action** を定義するための関数です。

- * typeとして "ADD_TODO" を持ちます。
- * string型の Payload (Reducerに渡されるデータ) を持ちます。

Dispatcher を経由して例えば、上の **Action** を呼び出すことで **Action** が **Reducer** に向かって飛んでいきます。 **Reducer** はその **Action** を受け取って、アプリ全体の **Store** を更新するというしくみです。

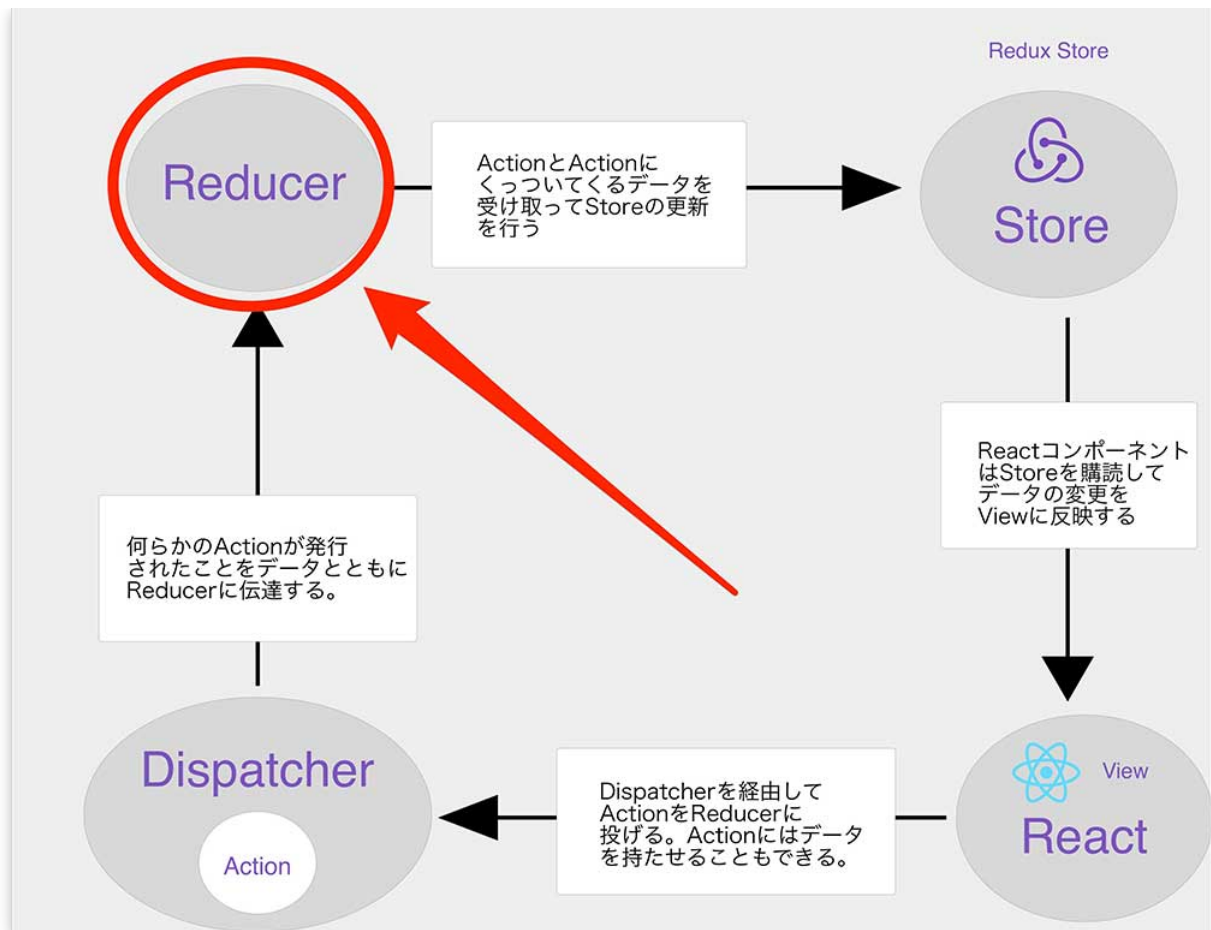


とりあえずは **Action** をひとつ作りました。これで **Action** は完了です。次はこの **Action** が飛んでいく先、すなわち **Reducer** を作りましょう。

Reducerをつくる 📁

Reducer を作っていきましょう。 **Reducer** の役割は飛んできた **Action** を受け取って **Redux** の **Store** (データ) のことを更新することです。

作成するのは全体図の中でのこの部分です。



Reducerとして、下記のファイルをつくります。

```
/* src/reducer.ts を作成 */

import { reducerWithInitialState } from 'typescript-fsa-reducers';

/* すべてのActionをインポート */
import * as actions from './action';

/* tasks[] 配列に格納するオブジェクトの型を定義する */
interface ITask {
  id: number;
  text: string;
  done: boolean;
}

/* Storeの型を定義する。 */
export interface IToDoState {
  tasks: ITask[];
}
```



```
/* 初期状態のStoreのデータを定義する */
export const initialReduce_TODOState: I_TODOState = {
  tasks: [
    {
      done: false,
      id: 1,
      text: 'initial task',
    }
  ],
};

let idCounter: number = 1;

/* Taskを作成する。ITaskという指定された型を返す。 */
const buildTask = (text: string): I_Task => ({
  done: false,
  id: ++idCounter,
  text,
})

/*
  addToDoというActionを待ち受けるとともに初期状態のStoreをセットする。
  addToDoが飛んできた場合には、新しいTaskをStoreに格納して、Storeを更新する。
*/
export default reducerWithInitialState(initialReduce_TODOState)
  .case(actions.addToDo, (state: I_TODOState, payload) => ({
    ...state,
    tasks: state.tasks.concat(
      buildTask(payload)
    )
  })))
  .build();
```

粗っぽく誤解を恐れずに言えば、関数である **Reducer()** の実行結果 (戻り値) が **Redux Store** のデータとなります。ここでは **typescript-fsa-reducers** というライブラリを使って、**reducer** を定義しています。**typescript-fsa-reducers** は型情報を伴ってラクに **Reducer** を定義するためのライブラリです。

case() という関数で対応する **Action** を受け取ります。第一引数に **対応するAction** 第二引数のコールバックに **現在のState** と **Actionから渡されてきたデータ** が入ります。それらをもとに、新しい **State** を返すことによって、**Redux Store** を更新します。

中身を見ていきましょう。 **addTodo** という **Action** を受け取った時に 既存の **State** が持つ **tasks** という配列に対して、 **buildTask()** で生成した新しい **task** を追加する仕組みになっています。

```
({
  ...state,
  tasks: state.tasks.concat(
    buildTask(payload)
  )
})
```

上記の式が実行されると、下記のようなオブジェクトが **Reducer** から戻り値として返されます。それがReduxの **Store** となります。

```
{
  tasks: [
    {
      done: false,
      id: 1,
      text: 'initial task',
    },
    {
      done: false,
      id: 2,
      text: 'Do Rails Tutorial!',
    }
  ]
}
```

addTodo という **Action** は先程、 **Action** のところで定義したように **stringのデータ(payload)**を **伴って飛んできます** ので 飛んできた string型のデータを **buildTask()** に渡して **Task** を生成します。このようにしてReduxの **Store** に新しい **Task** が追加され、更新されます。 **id** は新しい **Task** を作るたびに自動的に増えていくようにします。

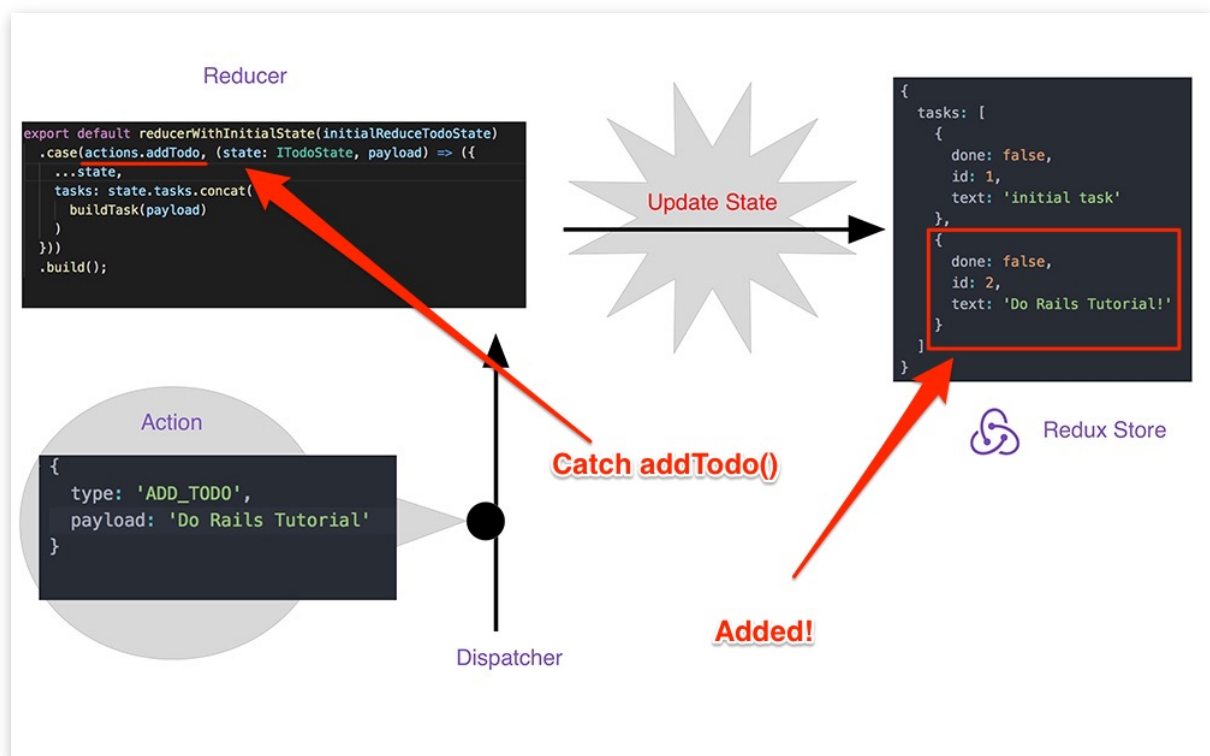
```
let idCounter: number = 1;

const buildTask = (text: string): ITasks => ({
  done: false,
  id: ++idCounter,
  text,
```

```
  })
```

```
export default reducerWithInitialState(initialReduce_TODO_State)
  .case(actions.addToDo, (state: I_TODO_State, payload) => ({
    ...state,
    tasks: state.tasks.concat(
      buildTask(payload)
    )
  }))
  .build();
```

下記のようなイメージです。



さらに、この **Reducer** で **Redux Store** の初期状態のデータも定義してしまいます。

```
export const initialReduce_TODO_State: I_TODO_State = {
  tasks: [
    {
      done: false,
      id: 1,
      text: 'initial task',
    },
  ],
};
```

```
export default reducerWithInitialState(initialReduce_TODO_State)
...

```

initialReduce_TODO_State で初期状態のデータを定義します。そして、***initialReduce_TODO_State*** を ***reducerWithInitialState()*** に渡すことで、Reduxの ***Store*** に初期状態のデータがセットされます。初期状態の ***Store*** が持つデータは下記のようになります。

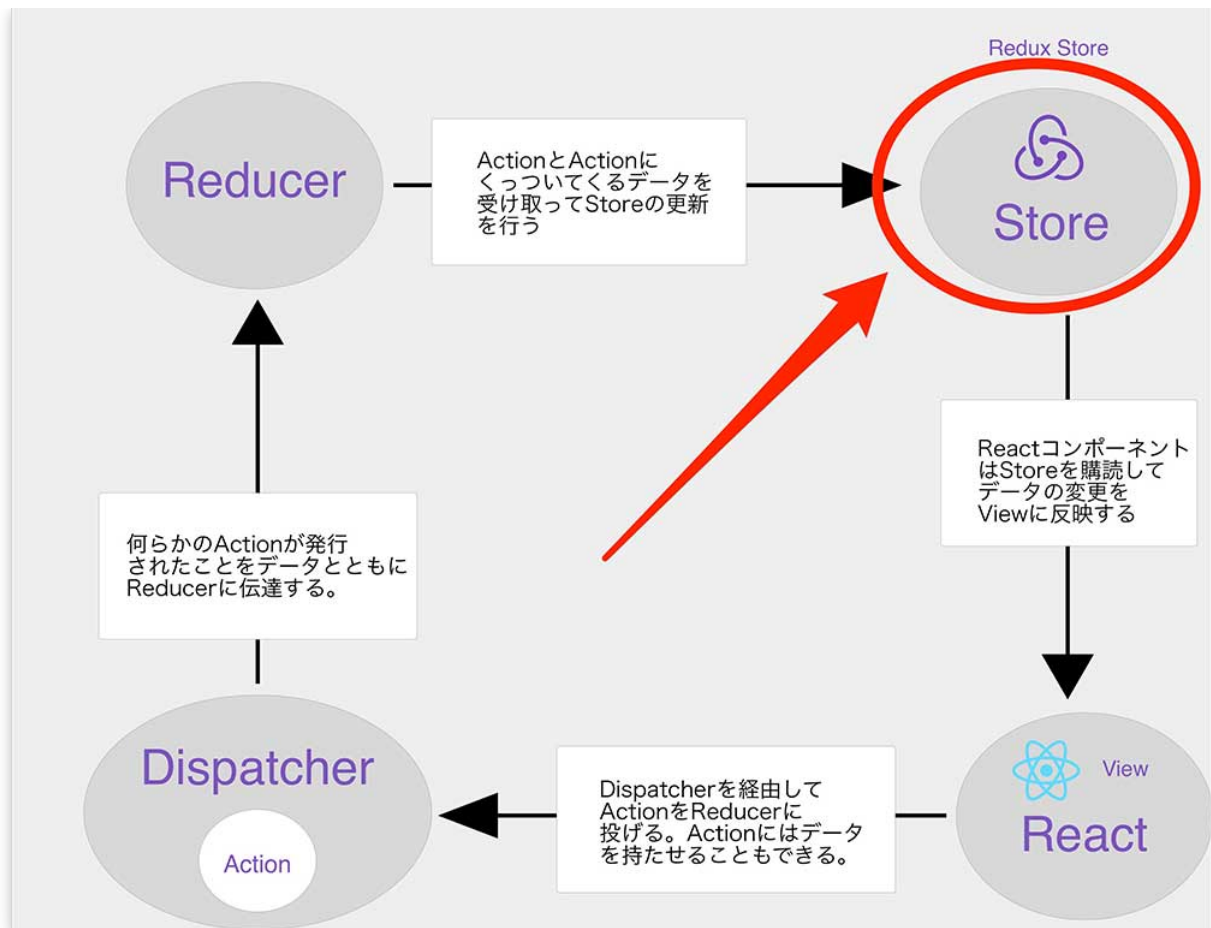
```
/* 初期状態の Redux Store */

{
  tasks: [
    {
      done: false,
      id: 1,
      text: 'initial task',
    }],
}
```

Storeをつくる

上記でつくった ***Reducer*** は ***Store*** と密接に結びついています。

Storeをつくりましょう。作成するのはこの部分です。



それでは **src/store.ts** を作りましょう。

```
/* src/store.ts を作る */

import { createStore } from 'redux';
import { composeWithDevTools } from 'redux-devtools-extension';
import reducer from './reducer';

const composeEnhancers = composeWithDevTools({});

/* production環境でない場合にはRedux Dev Toolsを有効化する */
export const buildTodoStore = () => (
  process.env.NODE_ENV === 'production' ?
    createStore(
      reducer,
    )
    :
    createStore(
      reducer,
      composeEnhancers()
    )
);
```

```
)  
)
```

Redux が提供する **createStore()** という関数で Reduxの **Store** をつくります。

```
createStore(reducer)
```

createStore() の引数には、初期状態のデータや、Reduxの **middleware** といった、ライブラリ群をとります。ここでは先程、定義した **Reducer** を引数にかませることによって、**Reducer**が返す初期状態のデータ、すなわち下記が Reduxの **Store** にセットされます。**Action** が飛んできて、**Reducer** が発動し、Redux **Store** が更新されるたびにデータは更新されます。

```
{  
  tasks: [  
    {  
      done: false,  
      id: 1,  
      text: 'initial task',  
    }],  
}
```

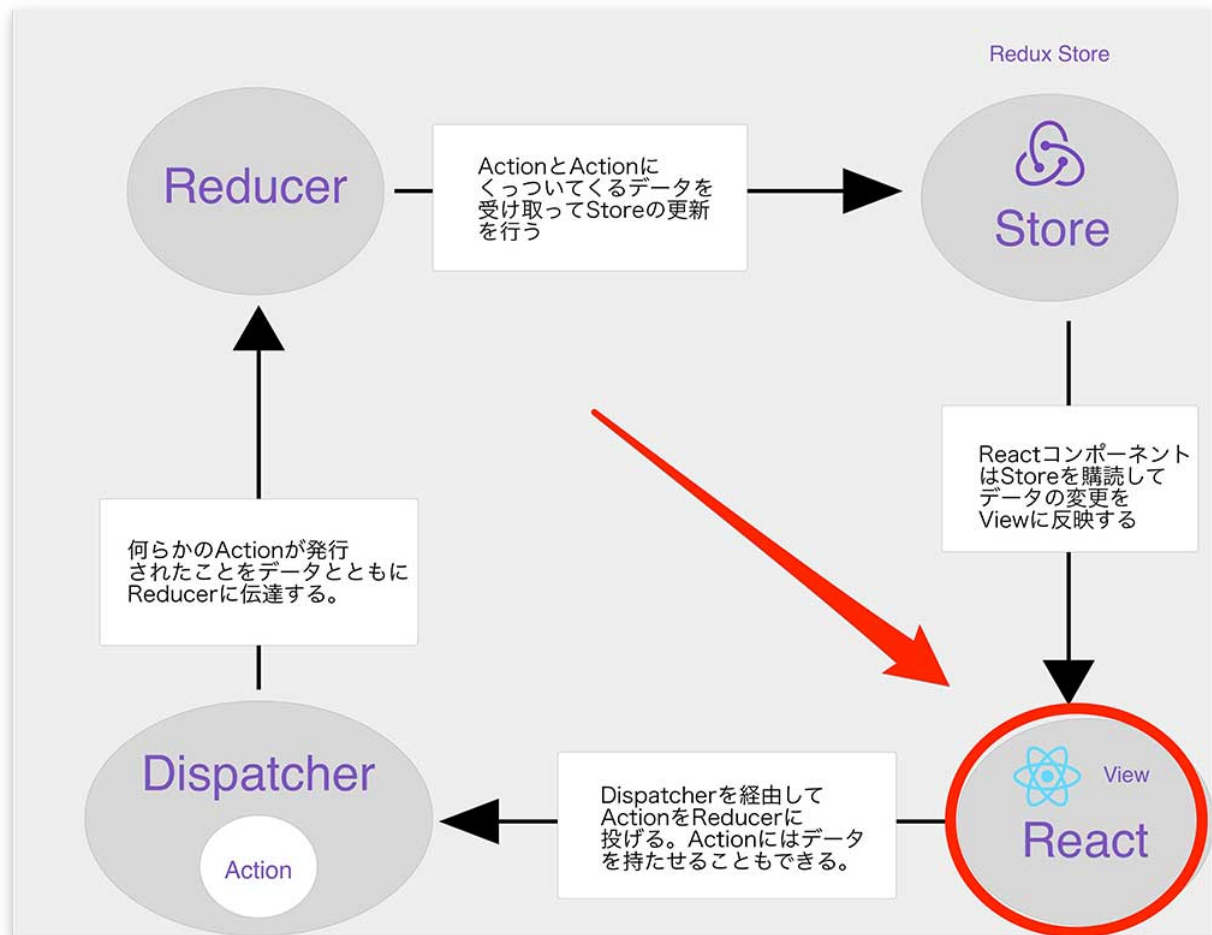
下記のコードで **production** 環境でない時にはReduxデバッグ用の **Redux Dev Tools** を適用します。**redux-devtools-extension** が提供する**composeWithDevTools()** で初期化をしています。**Google Chromeの拡張機能** (他のブラウザでもアドオンとして提供されています) で Redux アプリの **State** や発行された **Actionの履歴**などを俯瞰できる便利なデバッグツールです。

```
const composeEnhancers = composeWithDevTools({});  
  
...  
  
createStore(  
  reducer,  
  composeEnhancers()  
)
```

これで **Store** は完成です。これから、この **Store** を利用する Reactコンポーネントをつくっていきます。

View (React コンポーネント) をつくる

Redux Store を利用するViewを作成していきます。 作成するのはこの部分です。



では **src/App.tsx** を編集しましょう。下記のように編集します。

```
/* src/App.tsxを編集する */

import * as React from 'react';
import { Provider } from 'react-redux';

import './App.css';
import { buildTodoStore } from './store';
import TodoContainer from './TodoContainer';

/* react-reduxのProviderでラッピングしたコンポーネントをレンダリング */
class App extends React.Component {
  public render() {
    return (
      <Provider store={buildTodoStore()}>
        <TodoContainer/>
      </Provider>
    );
  }
}
```



```
    </Provider>
  );
}
}
```

```
export default App;
```

react-redux というライブラリが提供する **Provider** でReactコンポーネントをラッピングします。この **react-redux** というライブラリで **React** と **Redux**をつなぎます。そして、**Provider** の **Props** に **buildTodoStore()** することで生成した **Store** を渡してあげます。この **Provider**でラッピングしたReactコンポーネント配下では、**react-redux** が提供する **connect()** 関数を使えるようになります。

この **connect()** 関数で **Redux Store** が持つデータをコンポーネントに流し込めるようになります。Reactコンポーネントは **connect()** によって流し込まれた **Store** が持つデータを見ながら 自分見た目を変えたり、振る舞いを変えたりする流れです。

ここでは **TodoContainer** というReact コンポーネントを **Provider** でラッピングしています。

```
<Provider store={buildTodoStore()}>
  <TodoContainer/>
</Provider>
```

さて、このラッピングしている **TodoContainer** をつくりましょう。

connect() でReduxとReactをつなぐ 😊

src/TodoContainer をつくります。

```
/* src/TodoContainer.tsxをつくる */

import { connect, Dispatch } from 'react-redux';
import { ITodoState } from './reducer';
import TodoComponent from './TodoComponent'

/* 操作を加えることもできるが、今回は何も操作を加えない stateをそのままPropsに渡す */
const mapStateToProps = (state: ITodoState) => state
const mapDispatchToProps = (dispatch: Dispatch<any>) => ({ dispatch });

/* ReduxのStore由来のデータとDispatcherをPropsに格納して、TodoComponentに渡す。 */
```

```
export default connect(mapStateToProps, mapDispatchToProps)(TodoComponent);
```

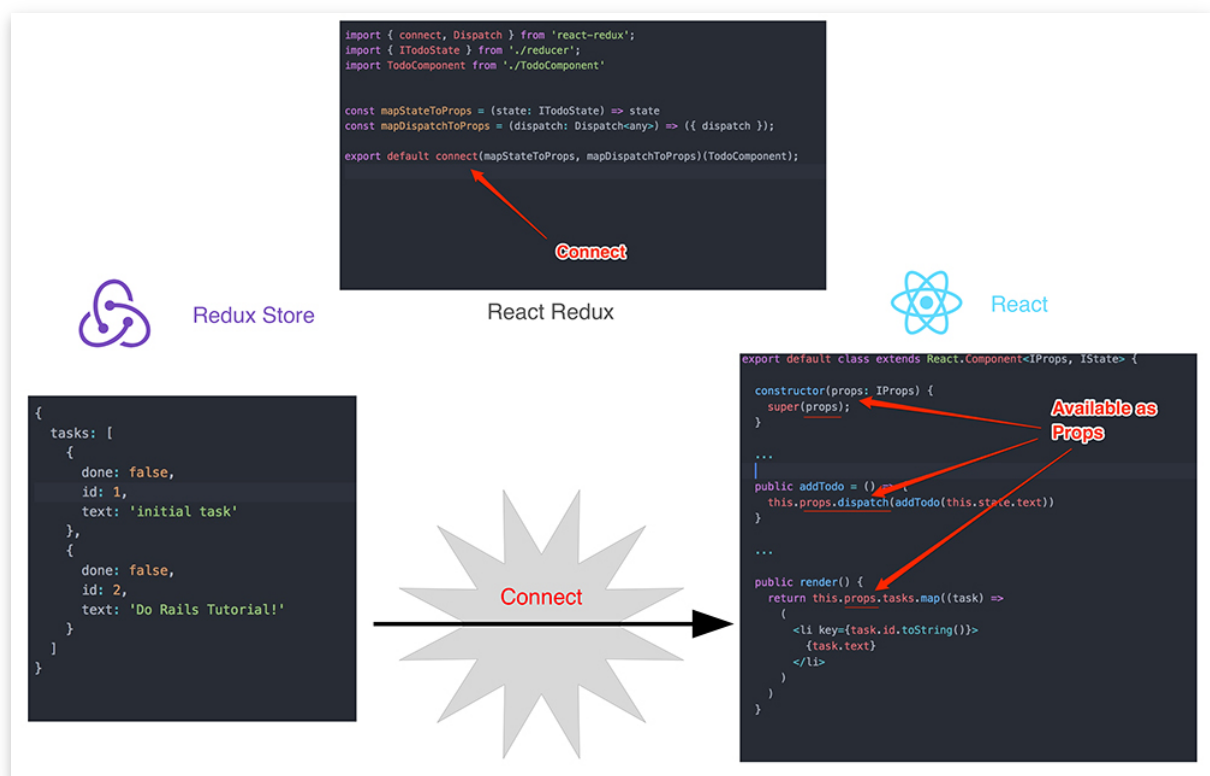
react-redux から **connect()** 関数をインポートして **TodoComponent** に対して、

- mapStateToProps で **Store** の持つデータをPropsに格納して、**TodoComponent** に渡します。
- mapDispatchToProps で **Dispatcher** をPropsに格納して**TodoComponent** に渡します。

上記をまとめて、Redux Storeのデータが下記のPropsとしてTodoComponentに渡ってきて、コンポーネント内部で自由に使えるようになります。

```
{
  tasks: [{
    done: false,
    id: 1,
    text: 'initial task',
  }],
  dispatch: 'Dispatcher (これを用いて、ReducerにActionを飛ばします)。'
}
```

イメージにすると下記ようになります。



Reactのレンダリングおよび、Actionの発行 ✨

次はconnect()する対象である **TodoComponent** をつくりましょう。ファイル **src/TodoComponent.ts** をつくります。

```
import * as React from 'react';
import { Dispatch } from 'redux';
import { addTodo } from './action';
import { ITodoState } from './reducer';

/* コンポーネントのProps ITodoStateを継承することで、ITodoStateの持つプロパティに加
interface IProps extends ITodoState {
  dispatch: Dispatch<any>;
}

/* コンポーネントが持つ内部State。今回は新しく追加するtodoのテキストを ReduxのStoreに
interface IState {
  text: string;
}

/* tslint:disable:jsx-no-lambda */
export default class extends React.Component<IProps, IState> {

  constructor(props: IProps) {
    super(props);

    this.state = {
      text: ''
    }
  }

  /* Propsとして渡ってきたDispatcherを経由して、ReducerにActionを投げる */
  public addTodo = () => {
    this.props.dispatch(addTodo(this.state.text))
  }

  /* Propsとして渡ってきた Redux Storeのデータをもとに自身のTodoリストをレンダリング
  public renderTodoList = () => (
    this.props.tasks.map((task) => (<li key={task.id.toString()}>{task.text}</li>))
  )
```

```
public render() {
  return(
    <section style={{width: '500px', margin: '0 auto'}}>
      <h1>MY TODO LIST</h1>
      <input
        type="text"
        value={this.state.text}
        onChange={(e: React.ChangeEvent<HTMLInputElement>) => {
          this.setState({text: e.currentTarget.value})
        }}
      />
      <button onClick={() => { this.addToDo() }}>Add Todo</button>
      <ul>
        {this.renderToDoList()}
      </ul>
    </section>
  )
}
```

基本的な設計の完成 🎉

ひとまずブラウザで確認する

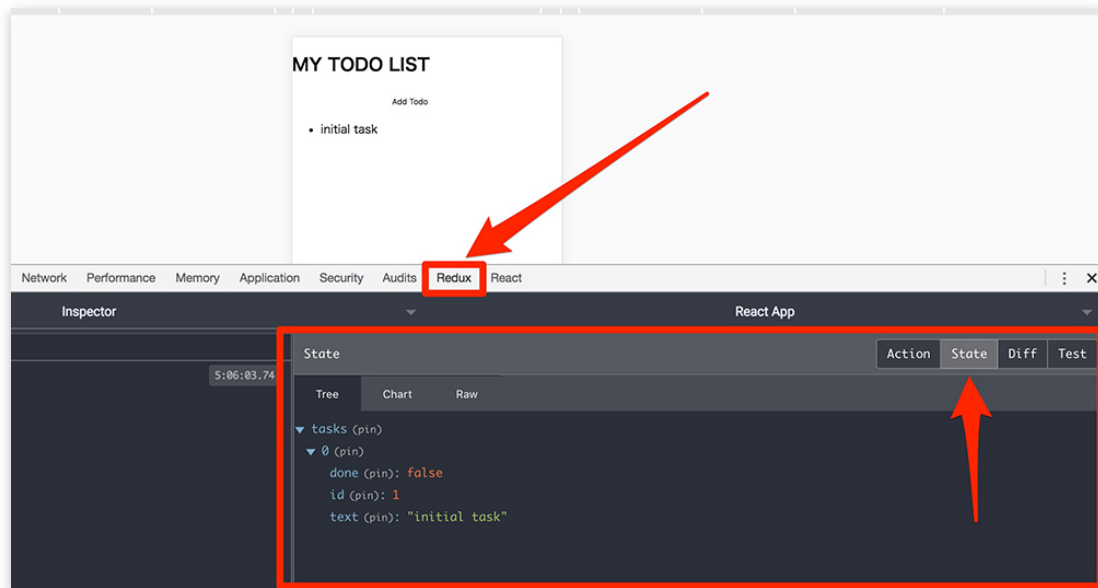
これで、Reduxアプリの "最小限の基本的な" 屋台骨の設計はいったん完成です。あとは、必要に応じて、ActionやReducerを増やしていったり、Reactコンポーネントを増やしていったりします。一旦、ここまでの結果をブラウザで確認してみましょう。

適当なテキストを入力して、「Add Todo」 ボタンをクリックします。 **Dispatcher** を経由して **Action** が **Reducer** に飛んでいき、その結果 **Store** が更新され、 **View** にテキストが追加されていくと思います。

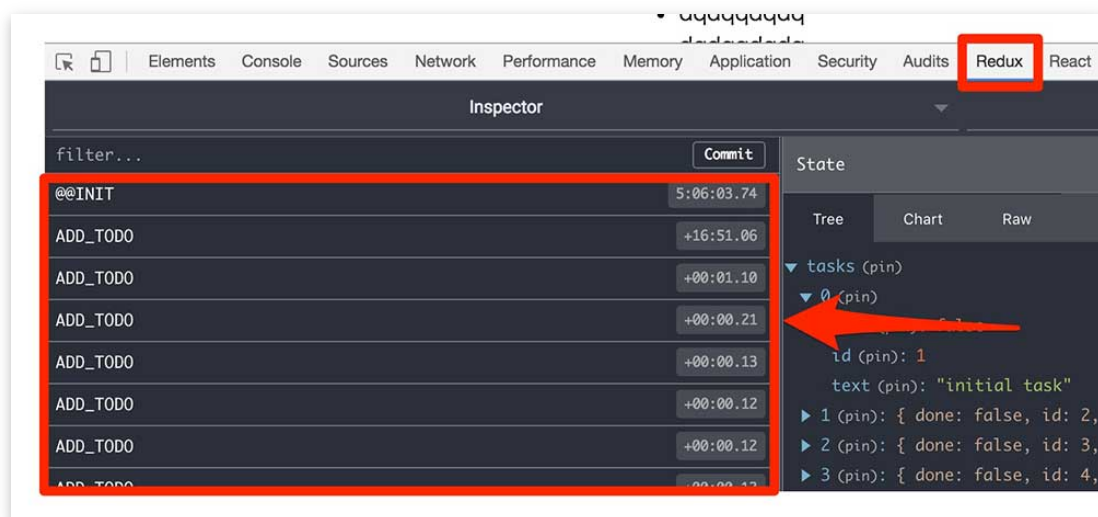
Redux Dev Toolsで確認する

また、Google Chromeにインストールした拡張機能の **Redux Dev Tools** を開いてみてみましょう。現在の **State** はもちろん、 **Action** が発行された履歴や **Action** が発行された時点へJumpしてその時点の **State** を確認できたりします。Stateをゴリゴリ変更することで、その時点のアプリの状態を自由に再現できたりします。

Redux Dev ToolsでStateを把握



Redux Dev ToolsでActionの履歴を把握



ここまでのミニマムなサンプルは下記からダウンロードできます。

1段階目 (Taskの追加機能のみ)

2 Actionの追加機能編集 (タスクの削除とタスク完了機能) 🍷

タスクの削除とタスク完了の更新機能のActionを追加していきましょう。

src/action.ts を編集します。新たにdeleteTodoとupdateDoneTodoを追加します。それぞれnumber型のpayloadを伴って発行されるようにします。

```
/* src/action.tsを編集*/
```

```
import actionCreatorFactory from 'typescript-fsa';
const actionCreator = actionCreatorFactory();

export const addTodo = actionCreator<string>(
  'ADD_TODO',
);

export const deleteTodo = actionCreator<number>(
  'DELETE_TODO',
);

export const updateDoneTodo = actionCreator<number>(
  'UPDATE_DONE_TODO',
);
```

Reducerの追加編集

上記で追加したActionの *deleteTodo* と *updateDoneTodo* に対応する処理を *Reducer* に追記しましょう。 *src/reducer.ts* を下記のように編集します。

```
/* src/reducer.tsを編集 */

import { reducerWithInitialState } from 'typescript-fsa-reducers';
import * as actions from './action';

export interface ITask {
  id: number;
  text: string;
  done: boolean;
}

export interface IToDoState {
  tasks: ITask[];
}

export const initialReduceTodoState: IToDoState = {
  tasks: [
```

```
{
  done: false,
  id: 1,
  text: 'initial task',
}],
};

let idCounter: number = 1;

const buildTask = (text: string): ITask => ({
  done: false,
  id: ++idCounter,
  text,
})

const updateDone = (tasks: ITask[], taskId: number): ITask[] => (
  tasks.map((task) => {
    if(task.id === taskId) {
      task.done = true
    }
    return task;
  })
)

export default reducerWithInitialState(initialReduceTodoState)
  .case(actions.addTodo, (state: ITodoState, payload) => ({
    ...state,
    tasks: state.tasks.concat(
      buildTask(payload)
    )
  }))
  .case(actions.deleteTodo, (state: ITodoState, payload) => ({
    ...state,
    tasks: state.tasks.filter((task) => ( task.id !== payload ))
  }))
  .case(actions.updateDoneTodo, (state: ITodoState, payload) => ({
    ...state,
    tasks: updateDone(state.tasks, payload)
  }))
  .build();
```

追加したのは下記の処理です。 **deleteTodoが来た場合** には指定されたidのタスクをtasks配列から取り除いた新しいStateを返します。 **updateDoneTodoが来た場合** には、指定されたidのタスク

のdoneをtrueに更新したStateを返します。

生のJavaScriptで処理を書いています、不慣れな場合には [lodash](#) などを使うのもありだと思います。

```
const updateDone = (tasks: ITask[], taskId: number): ITask[] => (
  tasks.map((task) => {
    if (task.id === taskId) {
      task.done = true
    }
    return task;
  })
)

...

.case(actions.deleteTodo, (state: IToDoState, payload) => ({
  ...state,
  tasks: state.tasks.filter((task) => ( task.id !== payload ))
}))
.case(actions.updateDoneTodo, (state: IToDoState, payload) => ({
  ...state,
  tasks: updateDone(state.tasks, payload)
}))
```

Viewの追加編集

上記で追加したActionを発行できるように、そして画面にStateの状態を反映できるようにVlewを編集しましょう。 **src/ToDoComponent.tsx** を編集します。

```
/* src/ToDoComponent.tsxを編集 */

import * as React from 'react';
import { Dispatch } from 'redux';
import { addTodo, deleteTodo, updateDoneTodo } from './action';
import { IToDoState } from './reducer';

interface IProps extends IToDoState {
  dispatch: Dispatch<any>;
```

```
}

interface IState {
  text: string;
}

/* tslint:disable:jsx-no-lambda */
export default class extends React.Component<IProps, IState> {

  constructor(props: IProps) {
    super(props);

    this.state = {
      text: ''
    }
  }

  public addTodo = () => {
    this.props.dispatch(addTodo(this.state.text))
  }

  public renderDoneBtn = (taskId: number) => (
    <button
      onClick={() => {
        this.props.dispatch(updateDoneTodo(taskId))
      }}
    >
      DONE
    </button>
  )

  public renderDeleteBtn = (taskId: number) => (
    <button
      onClick={() => {
        this.props.dispatch(deleteTodo(taskId))
      }}
    >
      Delete
    </button>
  )

  public renderDone = (done: boolean) => (
    done ? <span>done!</span> : null
  )
}
```

```

    )

    public renderTodoList = () => (
      this.props.tasks.map((task) => (
        <li key={task.id.toString()}>
          <span>{task.id}</span>
          <span>{task.text}</span>
          {this.renderDeleteBtn(task.id)}
          {this.renderDoneBtn(task.id)}
          {this.renderDone(task.done)}
        </li>
      )
    )

    public render() {
      return(
        <section style={{width: '500px', margin: '0 auto'}}>
          <h1>MY TODO LIST</h1>
          <input
            type="text"
            value={this.state.text}
            onChange={(e: React.ChangeEvent<HTMLInputElement>) => {
              this.setState({text: e.currentTarget.value})
            }}
          />
          <button onClick={() => { this.addToDo() }}>Add Todo</button>
          <ul>
            {this.renderTodoList()}
          </ul>
        </section>
      )
    }
  }
}

```

ここまで更新をして、一旦画面を見みましょう。「DELETE」をクリックすると追加したタスクが削除され「DONE」をクリックすると追加されたタスクが完了状態になることを確認できると思います。

ここまでのサンプルはこちらからダウンロードできます。

[2段階目 \(Taskの追加・削除・完了更新機能有\)](#)