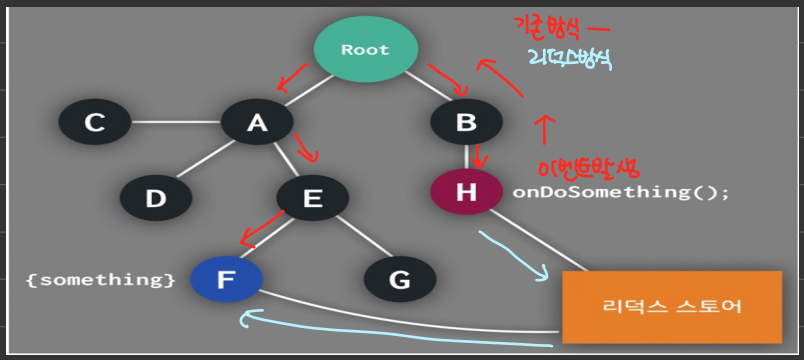
Redux (Library)

(Yarn add redux react-redux)

컴포넌트 상태관련 로직들을 다른파일로 분리하여 더욱 효율적으로 관리할수 있다.

컴포넌트끼리 상태 공유에도 여러 컴포넌트를 거치지 않고도 손쉽게 상태 값을 전달할 수 있다.



리덕스 스토어 (상태변화 3요) 통해서 우리가 원하는 상태 값과

함수를 직접 전달 할 수 있다.

① 액션

어떠한 변화가 필요할때 액션은 발생

```
{
  type : "      "
  data : {
      id : 0
      text : `리덕스배우기`
      }
```

② 액션 생성함수

파라미터를 받아와서 액션 객체 생성

```
const ChangeInput = text => ({
    type: "Change_Input"
    text
})
```

① ② 파라미터

③ 리듀서

변화를 일으키는 함수 (state , action)

```
const reducer = (S, A ) => {
        return alteredstate }
```

④ 스토어

리덕스에는 한 어플 당 하나의 스토어 만듦

스토어 안에는 현재 앱 state, reducer 가 들어있다

+ 몇가지 내장함수 (dispatch) 가 들어있다

⑤ 디스패치

액션을 발생시키는것, 디스패치 함수에는 액션을

파라미터로 전달 → 호출 → 스토어는 리듀서 함수 실행

→ 액션을 처리

⑥ 구독 (Subscribe)

파라미터로 함수형태 값을 넣어줌

특정함수전달 → 액션이 디스패치 될때마다

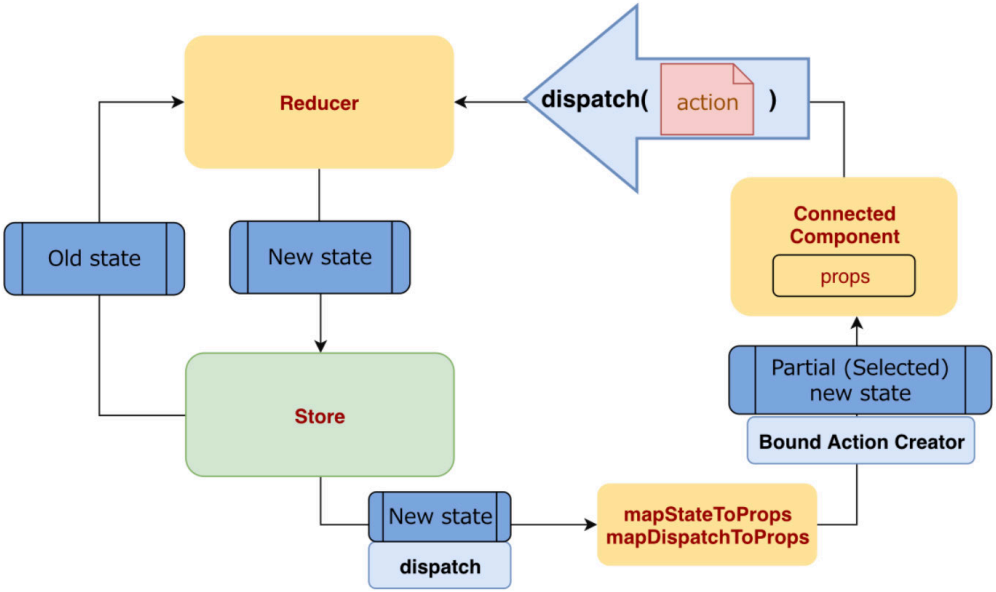
전달해준 함수가 호출

React랑 React+Redux의 결정적 차이

- React는 React 컴포넌트 자신이 개별적으로 상태관리를 한다.
- React+Redux는 **상태관리를 하는 전용 장소(store)**에서 상태를 관리하고, React 컴포넌트는 그걸 보여주기만 하는 용도로 쓰인다.

Redux 플로우의 이해

플로우 전체도



Store

상태는 기본적으로 전부 여기서 집중관리 됩니다. **커다란 JSON의 결정체** 정도의 이미지입니다.

```
{
    value: 0,
}
```

규모가 클 경우에는 상태를 카테고리별로 분류하는 경우가 일반적입니다.

```
{
    // 세션과 관련된 것
    session: {
        loggedIn: true,
        user: {
            id: "114514",
            screenName: "@mpyw",
        },
    },

    // 표시중인 타임라인에 관련된 것
    timeline: {
        type: "home",
        statuses: [
            {id: 1, screenName: "@mpyw", text: "hello"},
            {id: 2, screenName: "@mpyw", text: "bye"},
        ],
    },

    // 알림과 관련된 것
    notification: [],
}
```

Action 및 Action Creator

Store 및 Store에 존재하는 State는 아주 신성한 것이라고 할 수 있습니다. React 컴포넌트같은 하등한 것이 직접 접근하려고 하면 안 되는 것이죠. 직접 접근하기 위해 Action이라는 의식을 거쳐야 합니다. 이벤트 드리븐과 같은 개념입니다.

1. Store에 대해 뭔가 하고 싶은 경우엔 Action 을 발행한다.
2. Store의 문지기가 Action의 발생을 감지하면, State가 경신된다.

Reducer

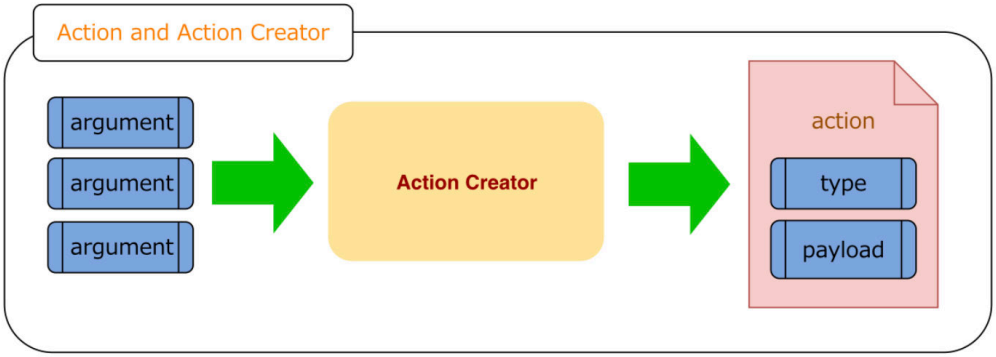
Action은 기본적으로 아래와 같은 포맷을 갖고 있는 오브젝트가 됩니다.

```
{
    type: "액션의 종류를 한번에 식별할 수 있는 문자열 혹은 심볼",
    payload: "액션의 실행에 필요한 임의의 데이터",
}
```

예를 들어 카운터의 값을 2배 늘리는 경우, 아래와 같은 오브젝트가 될 것입니다. 머릿부분에 `@@myapp/` 이라고 Prefix을 붙인건 다른 사람이 쓴 코드와의 충돌을 피하기 위함입니다.

```
{
    type: "@@myapp/ADD_VALUE",
    payload: 2,
}
```

그런데 하나하나 이런 오브젝트를 만드는 걸 수작업으로 하는 것도 정말 괴로운 일이겠죠. 또 `"@@myapp/ADD_VALUE"` 같이 매번 Action명을 문자열로 쓰는 것도 정말 싫습니다. 그래서, 이걸 조금 편하게 하기 위해 상수와 함수를 준비하는게 일반적입니다. 외부 파일이 참고할수도 있으니 제대로 export 해놓는게 좋겠습니다.

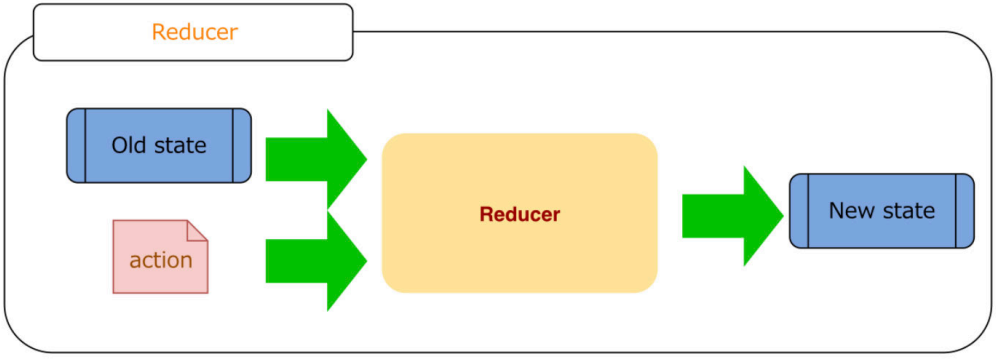



```
export const ADD_VALUE = '@@myapp/ADD_VALUE';
export const addValue = amount => ({type: ADD_VALUE, payload:
amount});
```

Reducer

앞에 'Store의 문지기'라고 쓴 적이 있습니다만, 그 개념과 비슷한 역할을 하는 것이 Reducer입니다.

함수형 프로그래밍에서 Reducer라는 용어는 합성곱을 의미합니다만, Redux에 한해서는 아래와 같이 **이전 상태와 Action을 합쳐, 새로운 state를 만드는** 조작을 말합니다.



```
import { ADD_VALUE } from './actions';

export default (state = {value: 0}, action) => {
    switch (action.type) {
        case ADD_VALUE:
            return { ...state, value: state.value + action.payload
};
        default:
            return state;
    }
}
```

Spread

불변성 유지

주의해서 봐야할 것은 바로 두가지 입니다.

- **초기상태는 Reducer의 디폴트 인수에서 정의된다**
- 상태가 변할 때 전해진 `state`는 그 자체의 값으로 대체 되는 것이 아니라, 새로운 것이 합성되는 것처럼 쓰여진다.

반환된 `state`는 store에 바로 반영 되어 아래와 같이 변화합니다.

```
{
    value: 2,
}
```

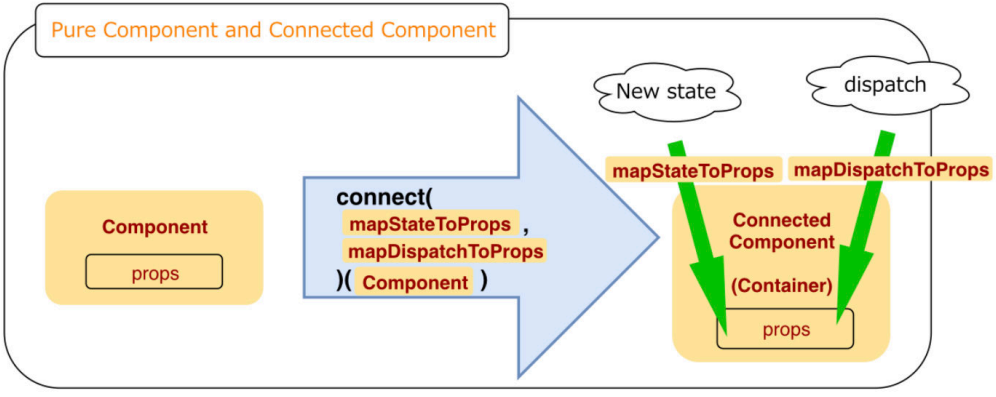
트위터에서처럼 대규모 개발에 Reducer를 미세하게 분할하는 경우 Redux에서 제공하는 `combineReducers`함수를 이용하여 아래와 같이 씁니다.

```
import { combineReducers } from 'redux';

const sessionReducer = (state = {loggedIn: false, user: null},
payload) => {
    /* 省略 */
};
const timelineReducer = (state = {type: "home", statuses: []},
payload) => {
    /* 省略 */
};
const notificationReducer = (state = [], payload) => {
    /* 省略 */
};

export default combineReducers({
    session: sessionReducer,
    timeline: timelineReducer,
    notification: notificationReducer,
})
```

이렇게 하면, **Reducer분할에 쓰인 Key가 그대로 State분할에도 쓰입니다**. 또한 실제로 각각의 reducer의 정의 자체도 다른 파일로 나누는 것이 일반적입니다.

순수한 Component와 연결된 Component

React의 Component자체는 Redux의 흐름에 타는 것이 불가능 합니다. 흐름에 타기 위해서는 ReactRedux에 의해 제공 되는 `connect` 라고 불리는 함수를 이용하여 아래와 같이 씁니다. 함수판과 클래스판 각각 씁니다.



```
>>>>>>>> 함수판
import React, { Component } from 'react';
import { connect } from 'react-redux';
import { addValue } from './actions';

const Counter = ({ value, dispatchAddValue }) => (
    <div>
        Value: {value}
        <a href="#" onClick={e => dispatchAddValue(1)}>+1</a>
        <a href="#" onClick={e => dispatchAddValue(2)}>+2</a>
    </div>
);

export default connect(
    state => ({ value: state.value }),
    dispatch => ({ dispatchAddValue: amount =>
dispatch(addValue(amount)) })
)(Counter)


>>>>>>>>> 클래스판
import React, { Component } from 'react';
import { connect } from 'react-redux';
import { addValue } from './actions';

class Counter extends Component {
    render() {
        const { value, dispatchAddValue } = this.props;
        return (
            <div>
                Value: {value}
                <a href="#" onClick={e =>
dispatchAddValue(1)}>+1</a>
                <a href="#" onClick={e =>
dispatchAddValue(2)}>+2</a>
            </div>
        );
    }
}

export default connect(
    state => ({ value: state.value }),
    dispatch => ({ dispatchAddValue: amount =>
dispatch(addValue(amount)) })
)(Counter)
```

자 여기서 드디어 도식도가 복잡해졌음을 느끼셨을겁니다. 제가 Redux를 공부하면서도 제일 이해에 시간이 많이 걸린 부분도 이 부분입니다. 이해하고 나면 별 것 아니기 때문에 초조해하지 말고 냉정하게 도식도를 바라봅시다.

먼저, Component가 Store로부터 무언가 정보를 받는 경우, 그걸 `props`를 통해 받습니다. `props`는 immutable합니다. 다시말해, **상태가 변경될 때마다 새로운 Component가 다시 만들어진다**는 의미입니다. 이것을 염두에 둔 후에, `connect`를 실행하고 있는 주변 코드를 봅시다.

1. Store가 가진 `state`를 어떻게 `props`에 엮을 지 정한다**(이 동작을 정의하는 함수는 `mapStateToProps`라고 불립니다)**
2. Reducer에 action을 알리는 함수 `dispatch`를 어떻게 `props`에 엮을 지 정한다 **(이 동작을 정의하는 함수는 `mapDispatchToProps`라고 불립니다)**
3. 위에 두가지가 적용된 `props`를 받을 Component를 정한다
4. Store와 Reducer를 연결시킬 수 있도록 만들어진 Component가 반환값이 된다

`connect(mapStateToProps, mapDispatchToProps)(Component)`라고 쓰인걸 보면 좀 독특하다고 생각할 수 있겠지만, 결국 최종적인 반환값은 4번과 같습니다. 아래선 `mapStateToProps`와 `mapDispatchToProps`에 대해 상세히 설명하도록 하겠습니다.

mapStateToProps

인수로 전달된 `state`는 **전체를 의미한다**는 것에 주의해야 합니다. 카운터의 예를 다시 보면,

```
{
    value: 2,
}
```

가

```
<Counter value={2} />
```

로 들어가길 바라며 `state => ({ value: state.value})`라고 썼습니다. 이번 경우에는 다른 프로퍼티가 없기 때문에 `state => state`라고 써도 동작에는 무리가 없겠지만, 기본적으로 **필요한 것만 선별하여 `props`로 엮는다**가 원칙이라고 생각합니다.

mapDispatchToProps

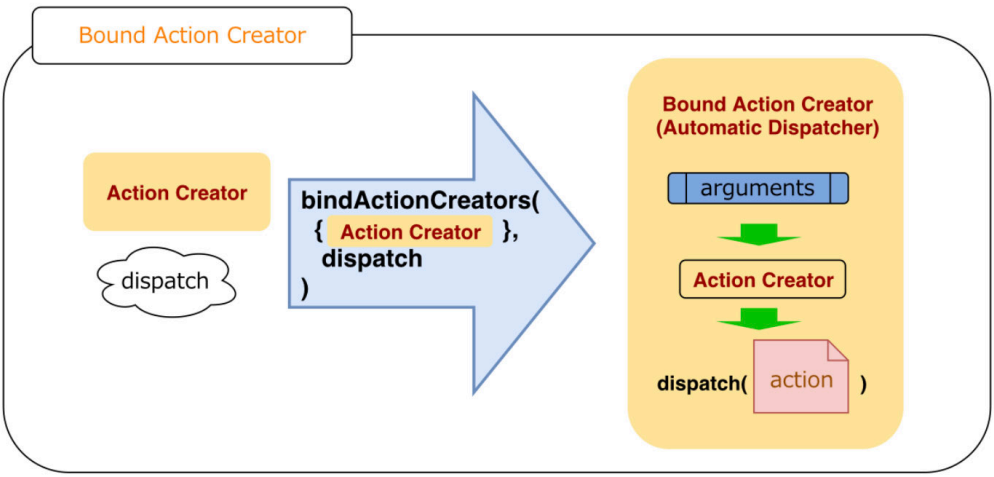
Action Creator에서 action을 만든다고 해도, 그것으론 아무 일도 일어나지 않습니다. Reducer를 향해 통지를 할 수 있게 만들기 위해서는 만든 action을 `dispatch`라는 함수에 넘겨줘야만 합니다.

이렇게 하면 **모든 Reducer가 실행 됩니다.** Reducer에 switch문으로 분기를 나눈 것은 바로 이 때문입니다. Reducer는 관계없는 action을 무시하고, 자기에게 주어진 action만을 처리하도록 되어있어야만 합니다.

또 **Component 쪽에 하나하나 수동으로 `dispatch`하는 처리를 하지 않아도 되도록, 여기서 action의 생성부터 `dispatch`의 실행까지 한번에 이뤄질 수 있도록 함수를 정의하여 `props`에 넘겨주도록 한다**는 멋진 존재 의의도 엿볼 수 있습니다.

bindActionCreators

하지만 무려 `mapDispatchToProps`를 이용하여 위와 같은 코드를 짜는 것에서도 도망칠 수 있습니다. `bindActionCreators`라는 함수를 제공하기 때문입니다. 이걸 쓰면 아래와 같은 생략이 가능합니다.



```
import React, { Component } from 'react';
import { bindActionCreators } from 'redux';
import { connect } from 'react-redux';
import { addValue } from './actions';


const Counter = ({ value, addValue }) => (
    <div>
        Value: {value}
        <a href="#" onClick={e => addValue(1)}>+1</a>
        <a href="#" onClick={e => addValue(2)}>+2</a>
    </div>
);

export default connect(
    state => ({ value: state.value }),
    dispatch => bindActionCreators({ addValue }, dispatch)
)(Counter)
```

현재는 `bindActionCreators`의 실행도 생략할 수 있게끔 되었습니다.

```
import React, { Component } from 'react';
import { connect } from 'react-redux';
import { addValue } from './actions';


const Counter = ({ value, addValue }) => (
    <div>
        Value: {value}
        <a href="#" onClick={e => addValue(1)}>+1</a>
        <a href="#" onClick={e => addValue(2)}>+2</a>
    </div>
);

export default connect(
    state => ({ value: state.value }),
    { addValue }
)(Counter)
```

Container

이 항에서는 지금까진 '연결된 Component'라고 불렀습니다만, 상황에 따라서 'Container'라고 불려야하는 Component도 나옵니다. 아래와 같은 것들이 해당됩니다.

- 수많은 Component가 리스트 형식으로 모여있는데 가운데 각 요소의 Component를 각각 연결하면 수습이 안 되므로, 대표적인 자식요소를 안고 있는 하나의 부모Component가 connect 되는 경우

```
<UsersList>
    <User />
    <User />
    <User />
    <User />
</UsersList>
```

이 **대표로서 connect될 부모 Component**를 **Container**라고 부릅니다.
Container는 가독성을 높이기 위해, Component와는 디렉토리를 따로 나누는 경우가 많습니다.

이해도 체크