

`const a = () => {` 가능 + 속성의 접근  
 Lexical Environment  $\Rightarrow$  함수형 프로그래밍에서 클로저는 "객체"이다.  
`const b = () => { }`  
 3  $\rightarrow$  closure

\* React는 함수가 최소 2개  $\rightarrow$  함수형 프로그래밍이 아니고 객체이다.

\*  $y=f(x) \Rightarrow$  함수 : 1차원 ( $\uparrow$   $f(x)$ ) vs 2차원 ( $\frac{f(x)}{x}$ ) = 2차원 (why? 함수형)  
 $\frac{f(x)}{x}$

\* 함수형 P : 상태 + 가변 데이터를 **지양** vs 객체지향 P : 상태 / 가변데이터 가능

$\Rightarrow$  2가지의 딜레마 / Paradox ... > 해결하기 위해 나온 것이 Redux!

$\hookrightarrow$  Why? 2가지로는 클로저 구조  $\Rightarrow$  객체 (무상태가 힘들다)  $\hookrightarrow$  함수형

$\rightarrow$  순수함수 개념의 등장. (무상태 Programming)

$\downarrow$  무상태 / 비동기 처리 필요.

for 마이크로 서비스  $\rightarrow$  무한 확장 가능함 (계층화, 모듈화)

* 함수형	vs	객체형
클로저 (리액트)		클래스
순수함수 (리덕스)		인터페이스

* 배열 함수. (map(), forEach())			
기준	Index	length	
for	0	0	
map()	X	X	
forEach()			

## 원하는 만큼 함수적이 되어라.

→ 무상태

함수형 프로그래밍(functional programming)은 본질적으로 프로그래밍을 수학으로 간주하는 것이다. 함수형 프로그래밍의 많은 개념들은 알론조 처치(Alonzo Church)의 람다 대수(Lambda Calculus)가 기초가 되었는데 이는 현대 컴퓨터의 개념과 조금이라도 흡사했던 그 어떤 것보다 앞섰다. 하지만 컴퓨터의 실제 역사는 다르게 흘러간다. 컴퓨터가 처음 발명될 무렵, 존 폰 노이만(John von Neumann)의 아이디어는 처치(Church)보다 더욱 중요하게 여겨졌고 그에 따라 초기 컴퓨터들의 설계에서부터 현재까지 많은 영향을 미쳤다. 폰 노이만의 생각은 “프로그램이란 명령들을 실행하기 위해 설계된 기계장치에서 실행되는 명령들의 목록”이라고 확고했다.

그렇다면 함수형 프로그래밍을 “수학으로 간주되는 프로그래밍”이라고 여기는 것에 대한 의미는 무엇일까? 폰 노이만은 수학자였고, 모든 프로그래밍의 근원지는 수학에서 찾을 수 있다. 함수형 프로그래밍이 수학적이라는 것은 무슨 의미이고 어떠한 수학과 관련되어 있다는 것일까?

사실 특정한 수학 분야와 관련되어 있다는 의미는 아니다. 람다 대수가 집합론(set theory)이나 논리수학, 범주론 등의 특정한 분야의 수학과 밀접한 관련이 있는 것은 사실이다. 하지만 프로그래밍의 기초가 되는 초등학교 수학의 수준부터 접근해보자. 다음은 자주 봤던 코드일 것이다.

Streams는 종종 함수형 언어와 연관되어 있다. 기본적으로 느릿느릿 평가되는 긴(거의 무한한) 리스트들이다. 즉, 문자열의 요소가 필요한 경우에만 평가된다는 의미다. Maps는 리스트의 모든 요소에 함수를 적용하여 (이러한 목적을 위해) 전문화된 리스트인 스트림을 포함한 새로운 리스트를 리턴한다. 이는 엄청나게 유용한 기능이다. 반복문(loop)을 작성할 필요 없이, 그리고 심지어 데이터를 얼마나 가지고 있는지조차 모르는 상태에서 반복문을 작성하는 최고의 방법이다. 또한 스트림 요소를 아웃풋에 패스할지 여부를 선택하는 filter를 만들고 maps와 filter를 함께 연결할 수도 있다. Unix pipeline이 생각난다면 맞다. Streams, maps, filters와 이 모든 걸 함께 연결하는 것은 함수형 언어와 Unix shell이 연관되어 있듯이 마찬가지로 연관이 있다.

for문

반복문을 피하는 또 다른 방법은 파이썬의 기능인 “comprehensions(컴프리헨션)”을 사용하는 것이다. List comprehension(리스트 컴프리헨션)과 익숙해지기 쉽다. 컴팩트하고 하나하나의 오류를 제거하며 탄력적이다. 컴프리헨션이 전통적인 반복문에서는 컴팩트한 표기법처럼 보일 수 있지만, 사실 집합론에서 비롯되었으며 가장 가까운 컴퓨터적 “친척”은 함수형 프로그래밍이 아닌 관계형 데이터베이스(relational database)다. 다음은 리스트의 모든 요소에 함수를 적용하는 컴프리헨션이다.

object  
 key  
 ↗  
 배열상수  
 ↗  
 차원별  
 ↗  
 반대면  
 ↗  
 구조화 / cf) 구조화  
 ↗  
 Getter/Setter

```

    []map((j, i) => {
      return (
        <tr key={ i }>
          <td>{ j, i } </td>
        </tr>
      )
    })
  
```

\* map()은 데이터를 분해하는 역할

객체와 배열은 자바스크립트에서 가장 많이 쓰이는 자료 구조입니다.

키를 가진 데이터 여러 개를 하나의 엔티티에 저장할 땐 객체를, 컬렉션에 데이터를 순서대로 저장할 땐 배열을 사용하죠.

개발을 하다 보면 함수에 객체나 배열을 전달해야 하는 경우가 생기곤 합니다. 가끔은 객체나 배열에 저장된 데이터 전체가 아닌 일부만 필요한 경우가 생기기도 하죠.

이럴 때 객체나 배열을 변수로 '분해'할 수 있게 해주는 특별한 문법인 구조 분해 할당(destructuring assignment)을 사용할 수 있습니다. 이 외에도 함수의 매개변수가 많거나 매개변수 기본값이 필요한 경우 등에서 구조 분해(destructuring)는 그 진가를 발휘합니다.

Input	Output
○	×

\* 객체에서 속성(리액트에서는 state)은 Getter/Setter를 통해 관리  
 Setter: Public void setName (String name) { this.name = name } Consumer  
 Getter: Public String getName () { return this.name } Supplier X ○  
 ⇒ this에 들어있는 Field를 자유자재로 (가비지 컬렉션) 나를 대신 자유로운 속성을 넘겨줄 수.  
 관리.

→ 고차함수 (숫자상수 X, 무상태 X)

\* 그렇다면 리액트에서 데이터/상태는 어떻게 다뤄야 할까?

- ① props : 복수의 파일에서 이동 파일 <sup>Props</sup>
- ② State : 단수의 파일에서 이동

\* props는 번역이 불가능 ⇒ State에 담아서 번역을 해주어야 한다.

State는 파일간 이동이 X ⇒ props에 담아서 전달해 주야 한다.

\* axios

function 풀기  
Why? 리턴이 없으면 피아프 구조가 가능.  
(then or catch)

```
axios.get(``,{})  
  .then(res=>{  
    // 피아프 구조  
    //  
    //  
  })  
  .catch(err=>{  
    //  
    //  
  })  
})
```

```

const SeoulCCTV = () => {
    const [items, setItems] = useState([]) // function
    const list = () => {
        axios.get('/data/SeoulFloatingPopulation.json')
            .then(res=>{
                setItems(res.data.DATA)
            })
            .catch(err=>{
                alert(err)
            })
    }
    useEffect(() => {
        fetchList();
    }, []);
}



상수를 뜯기 전략이 있다.



함수를 파악하기 어렵다 ×



전략이 있다,



함수 안에 함수



클리어


```

- 함수 사용하는 순간

호출이 되어야  
사용한다!

return하고 / const [ ] = useState 사용

그리고 대기 상태

는

내부가 막으면 허용 / 막으면 허용

⇒ 중복 예제 모아놓기 => useEffect

(실행시 추가해야 한다)

함수

B) useEffect() → 허용  
useEffect()은 } → 허용

→

"호출"

↓

전술이

Redux Toolkit  
 JSON  
 initial state  
 reducer  
 key  
 name  
 string

Creating a slice requires a string name to identify the slice, an initial state value, and one or more reducer functions to define how the state can be updated. Once a slice is created, we can export the generated Redux action creators and the reducer function for the whole slice.

Redux requires that we write all state updates immutably, by making copies of data and updating the copies. However, Redux Toolkit's createSlice and createReducer APIs use Immer inside to allow us to write "mutating" update logic that becomes correct immutable updates.

localStorage.setItem("k", "j")  
 sessionStorage

단위 테스트 가능  
 Static 테스트 가능

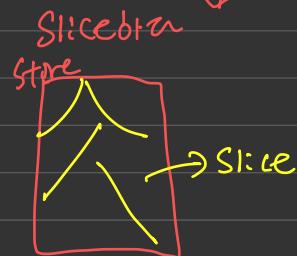
⇒ 저장 공간 ↓ → State 를 만들어야 한다  
 → Only One!

각

```
import { createSlice } from '@reduxjs/toolkit';
```

```
const CounterSlice = createSlice({})
```

```
export const {} = CounterSlice.actions;
export default CounterSlice.reducer;
```



**Redux Toolkit** is our official recommended approach for writing Redux logic. It wraps around the Redux core, and contains packages and functions that we think are essential for building a Redux app. Redux Toolkit builds in our suggested best practices, simplifies most Redux tasks, prevents common mistakes, and makes it easier to write Redux applications.

RTK includes utilities that help simplify many common use cases, including store setup, creating reducers and writing immutable update logic, and even creating entire "slices" of state at once.

Whether you're a brand new Redux user setting up your first project, or an experienced user who wants to simplify an existing application,

**Redux Toolkit** can help you make your Redux code better.

인자값

→ 각 액션에서 state는 들어온 것처.

The whole global state of your app is stored in an object tree inside a single store. The only way to change the state tree is to create an action, an object describing what happened, and dispatch it to the store. To specify how state gets updated in response to an action, you write pure reducer functions that calculate a new state based on the old state and the action.

Local에서 처리하는 때

다시 같은 데 useState 사용해야 한다.

```
function counterReducer(state) =  
{ value: 0 }, action) {  
switch (action.type) {  
  case 'counter/incremented':  
    return { value:  
      state.value + 1 }  
  case 'counter/decremented':  
    return { value:  
      state.value - 1 }  
  default:  
    return state  
}  
}
```

newState

It's important that you should

작성방법 X \* not mutate the state object, but return a new object if the state changes.

상태

Instead of mutating the state directly, you specify the mutations you want to happen with plain objects called *actions*. Then you write a special function called a *reducer* to decide how every action transforms the entire application's state.

In a typical Redux app, there is just a single store with a single root reducing function. As your app grows, you split the root reducer into smaller reducers independently operating on the different parts of the state tree. This is exactly like how there is just one root component in a React app, but it is composed out of many small components.

This architecture might seem like a lot for a counter app, but the beauty of this pattern is how well it scales to large and complex apps. It also enables very powerful developer tools, because it is possible to trace every mutation to the action that caused it. You can record user sessions and reproduce them just by replaying every action.

## CORE CONCEPTS

트리구조  
상태(액션) 가치의 구조  
action 이터링

```
{ todos: [{ text: 'Eat food', completed: true }, { text: 'Exercise', completed: false }], visibilityFilter: 'SHOW_COMPLETED' }
```

This object is like a “model” except that there are no setters. This is so that different parts of the code can't change the state arbitrarily, causing hard-to-reproduce bugs.

To change something in the state, you need to dispatch an action. An action is a plain JavaScript object (notice how we don't introduce any magic?) that describes what happened. Here are a few example actions:

→JSON

프로세스의 상태전이 = 프로그램 실행  
준비 = 메모리에 로드

하나의 프로그램이 실행되면 그 프로그램에 대응되는 프로세스가 생성되어 준비 리스트의 끝에 들어간다. 준비 리스트 상의 다른 프로세스들이 CPU를 할당받아 준비 리스트를 떠나면, 그 프로세스는 점차 준비 리스트의 앞으로 나가게 되고 언젠가 CPU를 사용할 수 있게 된다.

• 디스패치(dispatch) = 핸들러(쓰레드)

준비 리스트의 맨 앞에 있던 프로세스가 CPU를 점유하게 되는 것, 즉 준비 상태에서 실행 상태로 바뀌는 것을 디스패치라고 하며 다음과 같이 표시한다.

dispatch (processname) : ready  $\rightarrow$  running = (state, action) => newState

• 보류(block)

실행 상태의 프로세스가 허가된 시간을 다 쓰기 전에 입출력 동작을 필요로 하는 경우 프로세스는 CPU를 스스로 반납하고 보류 상태로 넘어간다. 이것을 보류라고 하며 다음과 같이 표시한다.

block (processname) : running  $\rightarrow$  blocked

• 깨움(wakeup)

입출력 작업 종료 등 기다리던 사건이 일어났을 때 보류 상태에서 준비 상태로 넘어가는 과정을 깨움이라고 하며 다음과 같이 표시한다.

wakeup (processname) : blocked  $\rightarrow$  ready

• 시간제한(timeout)

운영체제는 프로세스가 프로세서를 계속 독점해서 사용하지 못하게 하기 위해 clock interrupt를 두어서 프로세스가 일정 시간동안만 (시분할 시스템의 time slice) 프로세서를 점유할 수 있게 한다

timeout (processname) : running  $\rightarrow$  ready

트리구조란 다음과 같다. 그리고 반드시 Root를 가진다. 리액트와 리덕스는 모두 루트를 가지므로 트리구조이다. 리액트는 컴포넌트 트리구조이고, 리덕스는 리듀서 트리구조이다.

트리 구조(tree 構造, 문화어: 나무구조)란 그래프의 일종으로, 여러 노드가 한 노드를 가리킬 수 없는 구조이다. 간단하게는 회로가 없고, 서로 다른 두 노드를 잇는 길이 하나뿐인 그래프를 트리라고 부른다.

트리에서 최상위 노드를 **루트 노드**(root node 뿌리 노드[·])라고 한다. 또한 노드 A가 노드 B를 가리킬 때 A를 B의 **부모 노드**(parent node), B를 A의 **자식 노드**(child node)라고 한다. 자식 노드가 없는 노드를 **잎 노드**(leaf node 리프 노드[·])라고 한다. 잎 노드가 아닌 노드를 **내부 노드**(internal node)라고 한다.

action 각자.

```
{ type: 'ADD_TODO', text: 'Go to swimming pool' }  
{ type: 'TOGGLE_TODO', index: 1 }  
{ type: 'SET_VISIBILITY_FILTER', filter: 'SHOW_ALL' }
```

Enforcing that every change is described as an action lets us have a clear understanding of what's going on in the app. If something changed, we know why it changed. Actions are like breadcrumbs of what has happened. Finally, to tie state and actions together, we write a function called a reducer. Again, nothing magical about it—it's just a function that takes state and action as arguments, and returns the next state of the app. It would be hard to write such a function for a big app, so we write smaller functions managing parts of the state:

만화의 State

And we write another reducer that manages the complete state of our app by calling those two reducers for the corresponding state keys:

```
function todoApp(state = {}, action) {  
  return {  
    todos: todos(state.todos, action),  
    visibilityFilter:  
      visibilityFilter(state.visibilityFilter,  
        action)  
  }  
}
```

수정할!

부모의 순수함수      핵심적인(여기서는) 개별 함수 (양수작용)

Member m = new Member

Member m = get + Name()      → 2번

각자

각자

각자

각자

## 함수 객체(function object)

STL 알고리즘에 데이터를 전달하기 위해서는 다음과 같은 방법을 사용할 수 있습니다.

1. 함수 포인터
2. 함수 객체
3. 람다 표현식

많은 STL 알고리즘이 데이터를 처리하기 위해 매개변수로 함수 객체(function object)를 받아들입니다.

펑크터(functor)라고도 불리는 함수 객체는 호출 연산자(())와 함께 사용할 수 있는 객체를 의미합니다.

이러한 함수 객체는 우선 타입을 선언하고, 해당 클래스에서 호출 연산자(())를 오버로딩하여 구현하게 됩니다.

### 함수 객체의 장점

직접적인 함수 호출과 비교하여 함수 객체를 사용하면 다음과 같은 장점을 가집니다.

1. 함수 객체는 상태(state)를 포함할 수 있습니다.
2. 함수 객체는 타입이므로, 템플릿 인수로 사용할 수 있습니다.

↓  
기본구조       $\text{reducer: function}([\ ], \{ \}) \{$   
                   $\text{return} [\ ] \}$   
                  ↓  
                   $\text{new state}$

Action → type이라는 Key값  
 대체로 어떤 것인지

```
{ type: 'ADD_TODO', text: 'Go to swimming pool' }
{ type: 'TOGGLE_TODO', index: 1 }
{ type: 'SET_VISIBILITY_FILTER', filter: 'SHOW_ALL' }
```

Enforcing that every change is described as an action lets us have a clear understanding of what's going on in the app. If something changed, we know why it changed. Actions are like breadcrumbs of what has happened. Finally, to tie state and actions together, we write a function called a reducer. Again, nothing magical about it—it's just a function that takes state and action as arguments, and returns the next state of the app. It would be hard to write such a function for a big app, so we write smaller functions managing parts of the state:

상태의 일부에 따라  
 ↗ 리듀서는 원래 단계 → slice ⌈ N 개의 조각으로 나눔

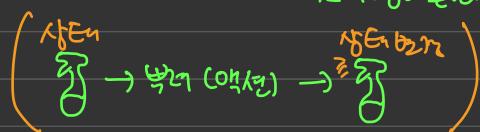
And we write another reducer that manages the complete state of our app by calling those two reducers for the corresponding state keys:

```
function todos(state = [], action) {
  switch (action.type) {
    case 'ADD_TODO':
      return state.concat([{ text: action.text, completed: false }])
    case 'TOGGLE_TODO':
      return state.map((todo, index)
        =>
          action.index === index
            ? { text: todo.text, completed: !todo.completed }
            : todo
      )
    default:
      return state
  }
}
```

상태의 Key

This is basically the whole idea of Redux. Note that we haven't used any Redux APIs. It comes with a few utilities to facilitate this pattern, but the main idea is that you describe how your state is updated over time in response to action objects, and 90% of the code you write is just plain JavaScript, with no use of Redux itself, its APIs, or any magic. 순수함수.

순수 함수는 외부의 상태를 변경하지 않으면서 동일한 인자에 대해 항상 똑같은 값을 리턴하는 함수다  $\hookrightarrow$  액션의 상태에는 영향이 없음



yarn add redux-actions

이렇게 불러와 사용 할 수 있다.

```
import {createAction, handleActions } from 'redux-actions';
```

- switch 문 대신 handleActions 사용하기

리듀서에 switch문을 사용하여 액션 type에 따라 다른 작업을 하도록 했다.

오류가 발생

이 방식에는 중요한 결점이 있는데

바로 scope를 리듀서 함수로 설정했다는 것이다.

그렇기 때문에 서로 다른 case에서 let이나 const를 사용하여 변수를 선언하려고 할 때

같은 이름이 중첩되어 있으면 문법 검사를 하면서 오류가 발생한다.

const reducer = handleActions({  
 INCREMENT: (state, action) => {  
 counter: state.counter + action.payload  
 },  
 DECREMENT: (state, action) => {  
 counter: state.counter - action.payload  
 }}, { counter: 0});  
cs

JSON  
reducer (함수)  
함수를 실행해라라는 뜻  
() => {}  
() => ({})  
JSON을 나타낸다는 표시.  
이렇게 표시하는 이유.  
[ {}, {}, {} ] → Store는 객체  
스토어는 액션 리듀서 가능  
속성 간에 객체 함수 가능

첫 번째 파라미터로는 액션에 따라 실행할 함수들을 가진 객체를 넣어주고,

두 번째 파라미터로는 상태의 기본 값( initialState )

## 자바 String의 불변성

- String: 문자열(String) 타입은 참조타입임에도 불구하고 직접 new 연산자를 통해 객체를 생성하여 사용하는 것이 아닌 문자열 리터럴 형태로 사용하는 것이 허용 된다.

```
String str1 = new String("Hello"); //가능  
String str2 = "Hello"; //가능
```

- String(문자열 객체)은 최초에 한 번 생성되면 절대로 그 값이 변하지 않는다. *readonly*

```
String str1 = "문자열1";  
str1 = "문자열2";
```

- 위와 같이 String 객체가 생성된 이후 "문자열1"을 "문자열2"로 바꾼다고 해서 실제 내부적으로 최초에 생성된 String 객체의 값이 변경된 것이 아닌 새로운 String 객체가 생성되어 그 참조가 str 변수에 할당된 것이다. 즉 이 상태에서는 최초에 생성된 "문자열1"과 "문자열2" 두 개의 객체가 heap에 생성되어 있는 상태이다.

스팅 값으로 기값으로 참으면 상태를 변경할 수 없다.

⇒ const increase 주소 값을 생성 → reducer는 상태를 변경해야 존재.

스토어 객체

기능	속성
리듀서	스테이트
변경	액션
수동형수 → JSON 리턴	{ "key" } 제작하는

Store: 상태 객체의 컨테이너 (저장소)

State: 스토어 안에 생성된 객체

Action: 리듀서를 생성(호출)하는 객체

Reducer: 스테이트를 변경하는 함수

useSelector: 스토어의 스테이트를 리턴하는 리스너.

useEffect: 기능을 (여기서는 리듀서 향상) 리턴하는 리스너?

```

import {createAction, handleActions} from 'redux-actions'

const INCREASE = "COUNTER_INCREASE";
const DECREASE = "COUNTER_DECREASE";

//액션객체 -> 리듀서의 인자로 전달하기 위해 만듬
const increase = createAction("INCREASE");
const decrease = createAction("DECREASE");

//상태
const initialState = {
    number: 0
};

//reducer
const CounterReducer = handleActions({
    [INCREASE] : (state, action)=>{
        number : state.number + 1
    },
    [DECREASE] : (state, action)=>{
        number : state.number - 1
    }
}, initialState);

```

↑ 초기(INCREASE) 액션  
 ↑ 치즈(DECREASE) 액션  
 ↑ 초기값이 저장  
 ↑ 초기값  
 ↑ 초기상태

### 디자인패턴 - 데코레이터 패턴(Decorator Pattern)

decoration은 '장식(포장)'이란 뜻이다. 빵집에서 케이크를 만들 때 먼저 둥근 모양의 빵을 만든다. 이 위에 초콜릿을 바르면 초콜릿 케이크가 되고, 치즈를 바르면 치즈 케이크가 된다. 또 생크림을 바르고 과일을 많이 올려놓으면 과일 생크림 케이크가 된다.

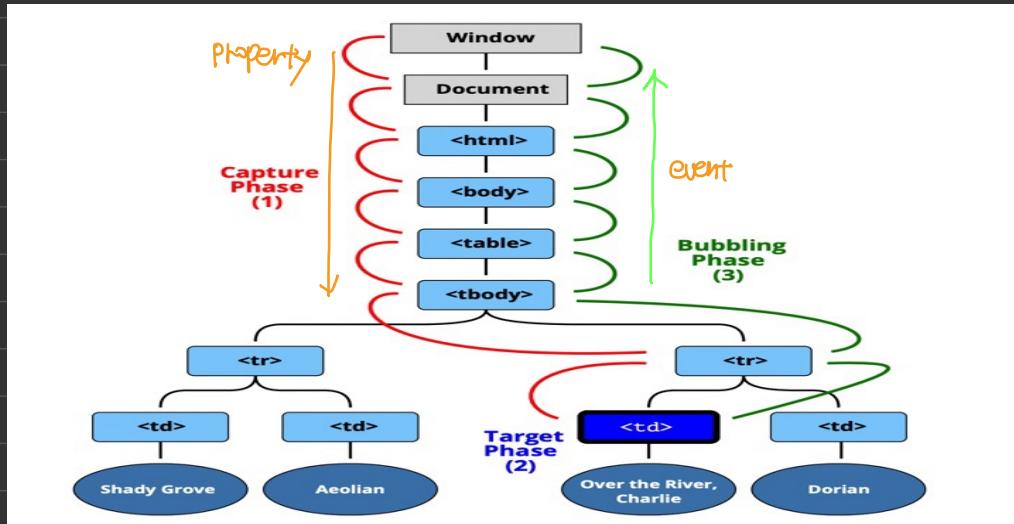
버블링(bubbling)의 원리는 간단합니다.

↑ 콜백

한 요소에 이벤트가 발생하면, 이 요소에 할당된 핸들러가 동작하고, 이어서 부모 요소의 핸들러가 동작합니다. 가장 최상단의 조상 요소를 만날 때까지 이 과정이 반복되면서 요소 각각에 할당된 핸들러가 동작합니다.

이벤트는 자식에서 부모로 전달 : 버블링

캡처링 .



스코어  
액션

중간 봇은 맥선을 보내줄 것을 만들자

↑  
이벤트 맥선

컨테이너  
의 역할

useSelector()

Store의 State를 관리하는 메소드.  
→ State를 참조할 수 있다.

```
import React, { useCallback } from 'react';
import { useSelector, useDispatch } from 'react-redux';
import ReduxCounter from '../components/ReduxCounter'
import { decrease, increase } from '../reducer/Counter.reducer';

const CounterContainer = () => {
  const number = useSelector(state => (state.counterReducer.number));
  const dispatch = useDispatch();
  const onIncrease = useCallback(()=> dispatch(increase()),[dispatch]);
  const onDecrease = useCallback(()=> dispatch(decrease()),[dispatch]);
  return (<>
    <ReduxCounter number={number} onIncrease={onIncrease}
    onDecrease={onDecrease}/>
  </>
)
}

export default CounterContainer;
```

부모 컴포넌트  
CounterReducer 내부의 State  
참조도 가능 why? 디스패치  
구독 가능하고  
동작하기 때문.

그럼 이 버튼은  
Store의 Dispatch를  
(Bubbling) 트리킹

props 전달

# 메모이제이션

<https://ko.wikipedia.org/wiki/%EB%A9%94%EB%AA%A8%EC%9D%B4%EC%A0%9C%EC%9D%B4%EC%85%98>

%EB%A9%94%EB%AA%A8%EC%9D%B4%EC%A0%9C%EC%9D%B4%EC%85%98

메모이제이션(*memoization*)은 컴퓨터 프로그램이 동일한 계산을 반복해야 할 때, 이전에 계산한 값을 메모리에 저장함으로써 동일한 계산의 반복 수행을 제거하여 프로그램 실행 속도를 빠르게 하는 기술이다. 동적 계획법의 핵심이 되는 기술이다.

함수형 프로그래밍의 기본원칙을 잘 지켰다면, (어떠한 외부 부수 효과에 영향을 받지 않는다면) 어떤 input이 들어가도 그 input에 대한 output은 동일 할 것이고, 따라서 동일한 input이 들어온다면 미리 전에 계산해두었던 output을 그대로 돌려줘도 될 것이다.

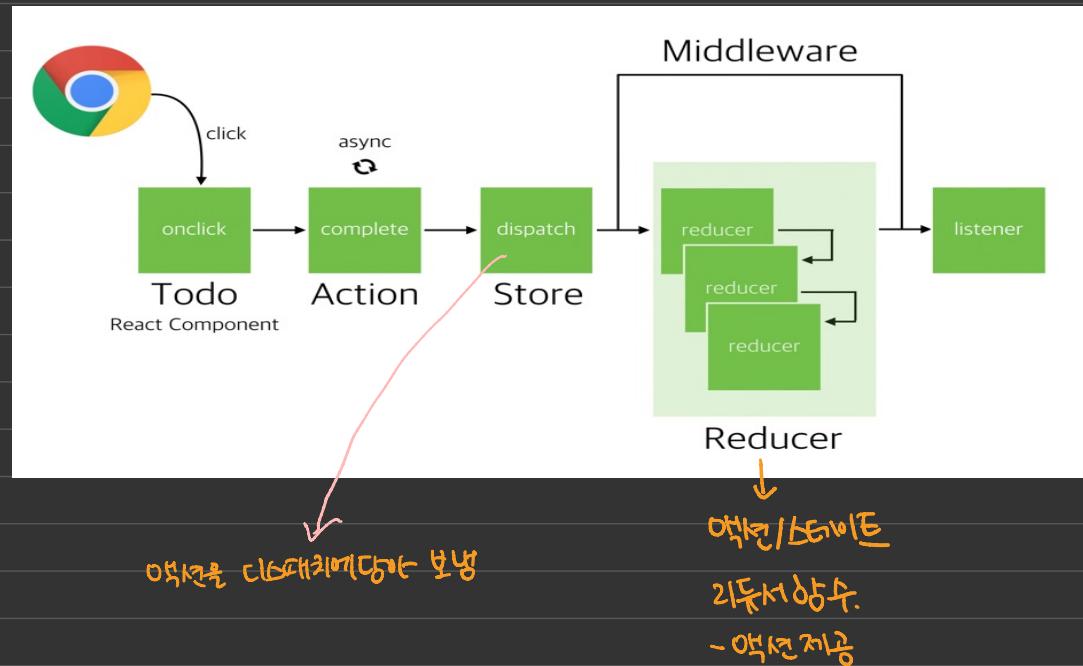
메모이제이션은 사전에 저장 혹은 정의된 속성이나 기능을 재 호출하는 것이다.

그중 속성 재호출은 useMemo이고, 기능 재호출은 useCallback이다. 예제에서는 dispatch 형태 () => () 기능 중에서 increase, decrease 를 호출하는 것이다. 그런데 그것이 리듀서이므로 useCallback()을 사용한다.

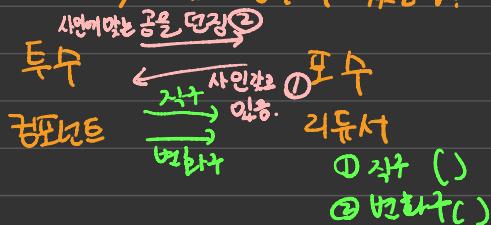
4/21

## Redux 전 흐름 과정

without Redux → Vanilla Redux → Tool Kit



액션이 why 리듀서 함수에 있는가?



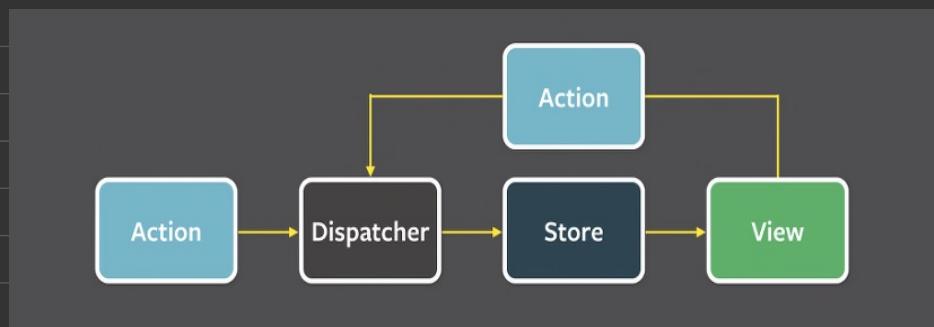
React.JS의 Redux 관련 글을 포스팅하면서 그리고 MVC, MVP, MVVM 패턴에 관한 포스팅을 하면서 다른 디자인 패턴에 관심을 가지게 되었습니다. Vuex는 Flux 패턴에 영감을 받아 만들어졌다고도 하고, Flux는 React.JS의 Redux의 디자인 패턴이기도 합니다. Flux와 MVC를 비교하기 전에 [디자인 패턴] MVC, MVP, MVVM 비교를 참고하시면 좀 더 이해하는데 도움이 될 것 같습니다. Flux는 MVC 모델의 단점을 보완하기 위해 페이스북에서 발표한 아키텍처입니다.

페이스북에서 이야기 하는 MVC의 가장 큰 단점은 양방향 데이터 흐름이었습니다.

### 기초 역사

복잡하지 않은 어플리케이션에서는 양방향 데이터 흐름이 문제가 크지 않을 수 있습니다. 하지만 어플리케이션이 복잡해 진다면 이런 양방향 데이터 흐름은 새로운 기능이 추가될 때에 시스템의 복잡도를 기하급수적으로 증가시키고, 예측 불가능한 코드를 만들게 됩니다. 개발자가 만든 어플리케이션이 개발자도 예측 못할 버그들을 쏟아 내게 됩니다.

페이스북에서 이야기하는 MVC의 양방향 데이터 흐름이 만들어 낸 예측하기 어려운 버그 중 하나는 알림 버그입니다. 페이스북에 로그인 했을 때, 화면 위 메시지 아이콘에 알림이 떠 있지만, 그 메시지 아이콘을 클릭하면 아무런 메시지가 없는 버그를 보신 적이 있으실 겁니다. 이 버그를 수정하고 얼마동안은 괜찮았지만 어플리케이션을 업데이트 하다 보면, 곧 다시 알림 버그가 나타나고 다시 버그를 수정하는 일이 반복되었습니다. 페이스북에서 이 문제의 해결 방법을 단방향 데이터 흐름으로 어플리케이션을 예측 가능하도록 만드는 방법에서 찾았습니다.



Flux의 가장 큰 특징은 단방향 데이터 흐름입니다. 데이터 흐름은 항상 Dispatcher에서 Store로, Store에서 View로, View는 Action을 통해 다시 Dispatcher로 데이터가 흐르게 됩니다. 이런 단방향 데이터 흐름은 데이터 변화를 훨씬 예측하기 쉽게 만듭니다. Flux를 크게 Dispatcher, Store, View 세 부분으로 구성됩니다.

# Vanilla → RTK Query

## ① Index.js

```
import React from "react";
import { render } from "react-dom";
- import { createStore } from "redux";
+ import { configureStore } from "@reduxjs/toolkit";
import { Provider } from "react-redux";
import App from "./components/App";
import rootReducer from "./reducers";

- const store = createStore(rootReducer);
+ const store = configureStore({
+   reducer: rootReducer,
+});
```

리듀서는 어떤가

## ② 슬라이스

'slice'가 무엇인지 궁금할 것입니다. 일반적인 Redux 앱은 상태트리의 최상단에 JS 객체를 가지고 있으며, 이 객체는 combineReducers 함수에서 여러 reducer들을 하나로 결합한 "root reducer"입니다. 이 객체에서 key/value로 구분되는 object를 "slice"라고 하며, slice의 상태를 업데이트 하는 리듀서를 "slice reducer"라고 합니다. 이 앱에서 루트 리듀서는 다음과 같습니다.:

JSON



## ③ CreateSlice (createReducer + createAction)

A function that accepts an initial state (an object full of reducer functions), and a "slice name", and automatically generates action creators and action types that correspond to the reducers and state.

This API is the standard approach for writing Redux logic.

Internally, it uses createAction and createReducer, so you may also use Immer to write "mutating" immutable updates:

State는 객체 → {key : Value}, or 배열? [ ] ? { } ?

`createSlice` accepts a single configuration object parameter, with the following options:

```
function createSlice({
  // A name, used in action types
  name: string,
  // The initial state for the reducer
  initialState: any,
  // An object of "case reducers". Key names will be used to generate actions.
  reducers: Object<string, ReducerFunction | ReducerAndPrepareObject>
  // A "builder callback" function used to add more reducers, or
  // an additional object of "case reducers", where the keys should be other
  // action types
  extraReducers?: | Object<string, ReducerFunction>
  | ((builder: ActionReducerMapBuilder<State>) => void)
})
```

## \* Payload

If provided, all arguments from the action creator will be passed to the prepare callback, and it should return an object with the payload field (otherwise the payload of created actions will be undefined). Additionally, the object can have a meta and/or an error field that will also be added to created actions. meta may contain extra information about the action, error may contain details about the action failure. These three fields (payload, meta and error) adhere to the specification of Flux Standard Actions.

**Note:** The type field will be added automatically. 단방향 패턴

```
{ type: 'counter/INCREASE',
  Payload: {amount: 1} }
```

## toolKit



import { createAction, handleActions } from 'redux-actions'  
사과기  
const INCREASE = 'counter/INCREASE'  
const DECREASE = 'counter/DECREASE'

1개로 풀기  
export const increase = createAction(INCREASE)  
//const increase = () => ({type: INCREASE})  
export const decrease = createAction(DECREASE)

const initialState = { number: 1 }

const counterReducer = handleActions({  
 [INCREASE] : (state, action) => ({...state, number :  
 state.number + 1}),  
 [DECREASE] : (state, action) => ({...state, number :  
 state.number - 1})  
}, initialState)

export default counterReducer

import { createSlice } from '@reduxjs/toolkit';  
1개로 풀기  
const counterSlice = createSlice()  
name: 'counterSlice',  
initialState: {number: 0},  
reducers: {  
 increase(state, action) {  
 return({  
 ...state, number: state.number + action.payload  
 })  
 },  
 decrease(state,action) {  
 return({  
 ...state, number: state.number - action.payload  
 })  
 }  
}  
export const {increase, decrease} = counterSlice.actions;  
이전은 생성 경포인트에서  
내가 쓴 것  
→ export default counterSlice.reducer;

## Container

```
import React, { useCallback } from 'react';
import { useSelector, useDispatch } from 'react-redux';
import ReduxCounter from '../components/ReduxCounter'
import { decrease, increase } from '../reducer/Counter.reducer';

const CounterContainer = () => {
  const number = useSelector(state =>
    state.counterReducer.number);
  const dispatch = useDispatch();
  const onIncrease = useCallback(() => dispatch(increase()), [dispatch]);
  const onDecrease = useCallback(() => dispatch(decrease()), [dispatch]);
  return (<>
    <ReduxCounter number={number}
      onIncrease={onIncrease} onDecrease={onDecrease}>
    </>,
  );
}

export default CounterContainer;
```

```
import { useDispatch, useSelector } from 'react-redux'
import CounterSlice from '../components/CounterSlice'
import { increase, decrease } from '../reducer/counter.slice'
```

```
const CounterSliceContainer = () => {
  const number = useSelector(state => (state.counterSlice.number));
  const dispatch = useDispatch()

  return (<>
    <CounterSlice number={number}
      onIncrease={() => dispatch(increase(1))}
      onDecrease={() => dispatch(decrease(1))}>
    </>
  )
}
```

```
export default CounterSliceContainer
```

## createSlice 옵션

어떤 옵션들이 있는지 살펴봅니다.:

createSlice는 다음 옵션과 함께 option 객체를 인자로 사용합니다. takes an options object as its argument, with these options:

name: 생성 된 action types를 생성하기 위해 사용되는 prefix

initialState: reducer의 초기 상태

reducers: key는 action type 문자열이 되고 할수록 해당 액션이 dispatch 될 때 실행될 reducer입니다. (switch-case 문과 비슷해서 "case reducers"라고도 합니다.)

따라서, "todos/addTodo" 액션이 dispatch될 때 addTodoreducer가 수행 됩니다.

default 핸들러는 없습니다. createSlice에 의해 생성된 리듀서는 현재 dispatch된 액션이 아닌 다른 액션들에 대해 자동으로 현재 상태를 반환하도록 처리되어 있기 때문에, 직접 핸들링해주지 않아도 됩니다.

## # "Mutable" 업데이트 로직

addTodo리듀서는 state.push()를 호출합니다. 일반적으로 이런 방식은 array.push() 함수가 기존 배열을 변형하기 때문에 좋은 방법이 아니고, Redux에서는 reducers에서 절대 state를 직접 변경하지 않아야 합니다!

그러나, createSlice와 createReducer는 Immer library의 produce로 래핑합니다. 이것은 이 함수를 사용하는 개발자는 리듀서 내부의 상태를 "변형하는" 코드를 작성할 수 있으며, Immer는 상태를 안전하게 불변하게 다룰 수 있도록 처리해줍니다.

마찬가지로, toggleTodo는 배열을 순회하거나 일치하는 todo 객체를 복사하지 않습니다. 대신, 일치하는 todo 객체를 찾은 다음 todo.completed = !todo.completed 코드로 변경합니다. Immer는 이 객체가 업데이트 된 것을 감지하고 todo 객체와 이를 포함하는 배열을 모두 복사합니다.

일반적인 불변성 관리는 추가 복사가 모두 발생하여 실제로 수행하려는 작업을 모호하게 하는 경향도 있습니다. 의도가 조금 더 명확히 드러나야 합니다: 배열 끝에 항목을 추가하고 todo 항목의 필드를 수정하는 것.

## Action Payloads와 함께 사용하기

잠시 reducer 로직을 다시 살펴보겠습니다.

기본적으로 RTK "createAction" 함수는 "payload"라는 하나의 인수만 허용합니다.

action.payload는 그 자체로 특별한 것이 없습니다. Redux는 그 이름을 모르거나 신경 쓰지 않습니다. 그러나 "payload"라는 이름은 "Flux Standard Actions"라는 또 다른 Redux 커뮤니티 컨벤션에서 유래되었습니다.

Action에는 일반적으로 "type" 필드와 함께 일부 추가 데이터가 포함되어야 합니다.

addTodo의 원래 Redux 코드에는 {type, id, text}처럼 보이는 action 객체가 있습니다. **FSA 규약에 따르면 임의의 이름을 가진 데이터 필드를 action 객체에**

**직접 포함하기보다는 항상 'payload'라는 필드 안에 데이터를 넣어야 합니다.**

각 액션 유형에 대해 'payload'를 무엇으로 하던, 그리고 어떤 코드를 dispatch하던 그 기대에 부합하는 값을 전달해야 한다고 생각하는지를 결정하는 것은 reducer에 달려 있습니다. 하나의 값만 필요한 경우 이 값을 전체 "payload" 값으로 직접 사용할 수 있습니다. 일반적으로 여러 값을 전달해야 합니다. 이 경우 'payload'는 해당 값을 포함하는 객체여야 합니다.

우리의 todos 슬라이스에서 addtodo는 id와 text라는 두 개의 필드가 필요하기 때문에 우리는 그것들을 payload로 객체 안에 넣었습니다. toggleTodo의 경우, 우리가 필요한 유일한 값은 변경되는 할일의 id입니다. 우리는 '페이로드'를 만들 수는 있지만 항상 '페이로드'를 객체로 사용하는 것을 선호하므로 대신 'action.payload.id'로 만들었습니다.

(잠시 살펴보면, action 객체 payload가 생성되는 방식을 사용자 정의하는 방법이 있습니다. 이 튜토리얼의 뒷부분에서 살펴보거나 createAction API 문서에서 설명을 볼 수 있습니다.)

4/22

## Hook vs Reducer

리액트의 함수

리액스의 함수

함수이다

Hook이 React 버전 16.8에 새로 추가되었습니다. Hook을 이용하여 Class를 작성할 필요 없이 상태 값과 여러 React의 기능을 사용할 수 있습니다.

리액트의 스테이트

함수

Hook은 without class와 react의 function

Reducer는 reduced function, state 변경 함수이다.

리스너

데이터가 “져오기” 구독(subscription) 설정하기, 수동으로 리액트 컴포넌트의 DOM을 수정하는 것까지 이 모든 것이 side effects입니다. 이런 기능들(operations)을 side effect(혹은 effect)라 부르는 것이 익숙하지 않을 수도 있지만, 아마도 이전에 만들었던 컴포넌트에서 위의 기능들을 구현해보았을 것입니다.

순수 함수 (Pure Function), 부수 효과 (Side Effect)

~~TODAY'S SPECIALTY~~

development

컴퓨터 프로그래밍 분야에서, 순수 함수(Pure Function)은 아래의 2가지 속성을 가진 함수이다.

- 1 같은 인자에 대해 같은 반환 값을 가진다.

- 비-로컬 변수, 로컬 정적 변수, 참조에 의해 전달된 인수, I/O에 의한 변화가 없어야 함

- 2 부수 효과(Side Effect)가 없다.

- 비-로컬 변수 수정, 로컬 정적 변수 수정, 참조에 의해 전달된 인수 수정, I/O 수행을 하지 않아야 함

말 그대로 외부 환경에 영향을 주지도 받지도 않는 함수이다.

## 부수 효과 (Side Effect)

프로그래밍 분야에서는 처음보는 단어!

컴퓨터 과학 분야에서, 함수가 Local 환경 밖의 상태를 변경할 때 그 함수는 “부수 효과”가 있다고 표현한다. (값을 바꾸는 것 외에 외부 세계와 상호작용을 할 때)

위에서 언급한 대로, 비-로컬 변수 수정, 로컬 정적 변수 수정, 참조에 의해 전달된 인수 수정, I/O 속성을 하지 않아야 한다.

부수 효과가 있는 함수는 작동 순서가 중요하며, 디버깅하기 위해서는 Context와 실행 기록에 대학 지식이 필요하다.

 원서 타입 vs 강조타입.

useState

Member<sub>1</sub> = new Member()

↓ ↓ ↓ ↓

type 창조 메모리 사용자

↳ 타입이란? 개별적 가르침.  $\Rightarrow$  DOM : HTML 개별적.

정리(Clean-up)를 이용하지 않는 Effects

리액트가 DOM을 업데이트한 뒤 추가로 코드를 실행해야 하는 경우가 있습니다.  
네트워크 리퀘스트, DOM 수동 조작, 로깅 등은 정리(clean-up)가 필요 없는 경 우들입니다. 이러한 예들은 실행 이후 신경 쓸 것이 없기 때문입니다.

리액트의 class 생명주기 메서드에 친숙하다면, useEffect Hook을  
componentDidMount와 componentDidUpdate,  
componentWillUnmount가 합쳐진 것으로 생각해도 좋습니다.

Hook을 이용하여 이 문제를 어떻게 해결할 수 있을까요? State Hook을 여러 번 사용할 수 있는 것처럼 effect 또한 여러 번 사용할 수 있습니다. Effect를 이용하여 서로 관련이 없는 로직들을 갈라놓을 수 있습니다.

```
function FriendStatusWithCounter(props) {  
  const [count, setCount] = useState(0);  
  useEffect(() => {  
    document.title = `You clicked ${count} times`;  
  });
```

```
const [isOnline, setIsOnline] = useState(null);  
useEffect(() => {  
  function handleStatusChange(status) {  
    setIsOnline(status.isOnline);  
  }
```

```
  ChatAPI.subscribeToFriendStatus(props.friend.id,  
  handleStatusChange);  
  return () => {  
    ChatAPI.unsubscribeFromFriendStatus(props.friend.id,  
    handleStatusChange);  
  };  
};
```

```
useEffect(() => {
  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
  return () => {
    ChatAPI.unsubscribeFromFriendStatus(props.friend.id,
    handleStatusChange);
  };
}, [props.friend.id]); // props.friend.id가 바뀔 때만 재구독합니다.
```

두 번째 인자는 빌드 시 변환에 의해 자동으로 추가될 수도 있습니다.

## 주의

이 최적화 방법을 사용한다면 배열이 컴포넌트 범위 내에서 바뀌는 값들과 effect에 의해 사용되는 값을 모두 포함하는 것을 기억해주세요. 그렇지 않으면 현재 값이 아닌 이전의 렌더링 때의 값을 참고하게 됩니다. 이에 대해서는 함수를 다루는 방법과 의존성 배열이 자주 바뀔 때는 어떻게 해야 하는가에서 더 자세히 알아볼 수 있습니다.

effect를 실행하고 이를 정리(clean-up)하는 과정을 (마운트와 마운트 해제 시에) 딱 한 번씩만 실행하고 싶다면, 빈 배열([])을 두 번째 인수로 넘기면 됩니다. 이렇게 함으로써 리액트로 하여금 여러분의 effect가 prop이나 state의 그 어떤 값에도 의존하지 않으며 따라서 재실행되어야 할 필요가 없음을 알게 하는 것입니다. 이는 의존성 배열의 작동 방법을 그대로 따라서 사용하는 것일 뿐이며 특별한 방법인 것은 아닙니다.

빈 배열([])을 넘기게 되면, effect 안의 prop과 state는 초기값을 유지하게 됩니다. 빈 배열([])을 두 번째 인수로 넘기는 것이 기존에 사용하던 componentDidMount와 componentWillUnmount 모델에 더 가깝지만, effect의 잦은 재실행을 피할 수 있는 더 나은 해결방법이 있습니다. 또한 리액트는 브라우저가 다 그려질 때까지 useEffect의 실행을 지연하기 때문에 추가적인 작업을 더하는 것이 큰 문제가 되지는 않습니다.

exhaustive-deps 규칙을 eslint-plugin-react-hooks 패키지에 포함하는 것을 추천합니다. 이 패키지는 의존성이 바르지 않게 지정되었을 때 경고하고 수정하도록 알려줍니다.

Hook이 생소하다면 Hook 개요를 먼저 읽어 보기 바랍니다. 혹은 frequently asked questions에서 유용한 정보를 찾을 수도 있습니다.

## 기본 Hook

useState

useEffect

useContext

## 추가 Hooks

useReducer

→ reducer에서 사용

useCallback

useMemo

useRef

useImperativeHandle

useLayoutEffect → 동기식에서 사용

useDebugValue

## \* 메모이제이션 패턴

이전에 계산된 값에 대한 재사용.

메모이제이션(memoization)은 컴퓨터 프로그램이 동일한 계산을 반복해야 할 때, 이전에 계산한 값을 메모리에 저장함으로써 동일한 계산의 반복 수행을 제거하여 프로그램 실행 속도를 빠르게 하는 기술이다. 동적 계획법의 핵심이 되는 기술이다.  
메모아이제이션이라고도 한다.

Member m = new Member("홍") 하는 것.

메모이제이션은 글자 그대로 해석하면 ‘메모리에 넣기’라는 의미이며 ‘기억되어야 할 것’이라는 뜻의 라틴어 memorandum에서 파생되었다. 흔히 메모라이제이션(memorization)과 비슷하지만 구분되는 용어이다.

메모이제이션이라는 용어를 처음 만든 사람은 도널드 미치(Donald Michie)이다. 1968년 네이처에 실린 논문 <Memor functions and machine learning>에서 처음으로 메모이제이션이라는 말이 나왔다.

## useCallback

```
const memoizedCallback = useCallback(  
() => {  
  doSomething(a, b);  
},  
[a, b],  
);
```

메모이제이션된 콜백을 반환합니다. → 이벤트가 발생한다음에 향수가  
콜백되는 순간

인라인 콜백과 그것의 의존성 값의 배열을 전달하세요. useCallback은 콜백의 메모  
이제이션된 버전을 반환할 것입니다. 그 메모이제이션된 버전은 콜백의 의존성이 변  
경되었을 때에만 변경됩니다. 이것은, 불필요한 렌더링을 방지하기 위해 (예로  
shouldComponentUpdate를 사용하여) 참조의 동일성에 의존적인 최적화된 자식  
컴포넌트에 콜백을 전달할 때 유용합니다.

useCallback(fn, deps)은 useMemo(() => fn, deps)와 같습니다.

useMemo  
const memoizedValue = useMemo(() => computeExpensiveValue(a,  
b), [a, b]);  
메모이제이션된 값을 반환합니다.

Member m = fn(b)

메모리에 온게 있다는 것을  
객체안에 있다는 것과 같은 것)

“생성(create)” 함수와 그것의 의존성 값의 배열을 전달하세요. useMemo는 의  
존성이 변경되었을 때에만 메모이제이션된 값만 다시 계산 할 것입니다. 이 최적  
화는 모든 렌더링 시의 고비용 계산을 방지하게 해 줍니다.

useMemo로 전달된 함수는 렌더링 중에 실행된다는 것을 기억하세요. 통상적  
으로 렌더링 중에는 하지 않는 것을 이 함수 내에서 하지 마세요. 예를 들어, 사이  
드 이펙트(side effects)는 useEffect에서 하는 일이지 useMemo에서 하는 일  
이 아닙니다.

배열이 없는 경우 매 렌더링 때마다 새 값을 계산하게 될 것입니다.

`useRef`는 `.current` 프로퍼티로 전달된 인자(`initialValue`)로 초기화된 변경 가능한 `ref` 객체를 반환합니다. 반환된 객체는 컴포넌트의 전 생애주기를 통해 유지될 것입니다.

일반적인 유스케이스는 자식에게 명령적으로 접근하는 경우입니다.

```
function TextInputWithFocusButton() {  
  const inputEl = useRef(null);  
  const onButtonClick = () => {  
    // `current` points to the mounted text input element  
    inputEl.current.focus();  
  };  
  return (  
    <>  
    <input ref={inputEl} type="text" />  
    <button onClick={onButtonClick}>Focus the input</button>  
    </>  
  );  
}
```

본질적으로 `useRef`는 `.current` 프로퍼티에 변경 가능한 값을 담고 있는 “상자”와 같습니다.

→ DOM을 향유정 한다.

아마도 여러분은 DOM에 접근하는 방법으로 refs에 친숙할지도 모르겠습니다.  
만약 `<div ref={myRef} />`를 사용하여 React로 ref 객체를 전달한다면,  
React는 모드가 변경될 때마다 변경된 DOM 노드에 그것의 .current 프로퍼티  
를 설정할 것입니다.

- ES6의 여러가지 문법들 중에서도 Vue.js 사용을 편하게 해주는  
-**const & let, Arrow Function, Enhanced Object Literals, Modules** 사용해보기.
- Enhanced Object Literals - 향상된 객체 리터럴  
- 객체의 속성을 메소드로 사용할 때 function 예약어를 생략하고 생성 한다.

```
var dic = {  
  words: 100,  
  //ES5  
  lookup: function() {  
    console.log("find words");  
  },  
  //ES6  
  lookup() {  
    console.log("find words");  
  }  
};
```

4/23

## Flux 란?

제거식 (내보내는 역할) vs 풀오버 표준역할

정의는 험

단방향

페이지로드가 많다

페이지로드가 많다.

=> 영으면 양방향

=> 있으면 단방향

순수함수: 향후에 변수 외에 외부의 값은 참조, 의존하지 않거나 변경하지 않음

```
let obj = {val: 20};
```

```
const notPureFn = (obj, b) => {
  obj.val += b;
}
```

```
const pureFn = (obj, b) => {
  return {val: obj.val + b} → JSON을 확인.
}
```

```
140
141  var hasChanged = false  hasChanged = false
142  var nextState = {}  nextState = Object {todos: Array[1]}
143  for (var i = 0; i < finalReducerKeys.length; i++) {  i = 0
144    var key = finalReducerKeys[i]  key = "todos"
145    var reducer = finalReducers[key]  reducer = function todos()
146    var previousStateForKey = state[key]  previousStateForKey = [Object], state = Obj
147    var nextStateForKey = reducer(previousStateForKey, action)  nextStateForKey = [Ob
148    if (typeof nextStateForKey === 'undefined') {
149      var errorMessage = getUndefinedStateErrorMessage(key, action)  errorMessage = u
150      throw new Error(errorMessage)
151    }
152    nextState[key] = nextStateForKey  nextState = Object {todos: Array[1]}, key = "todo
153    hasChanged = hasChanged || nextStateForKey !== previousStateForKey
154  }
155  return hasChanged ? nextState : state
156}
157}  Redux simply checks if the old object is the same as the new object.
```

Pass the old state to Reducer and get NEW state

Redux simply checks if the old object is the same as the new object.

리덕스는 주어진 상태(객체)를 가져와서 loop의 각 리듀서로 전달한다.  
그리고 변경사항이 있는 경우 리듀서의 새로운 객체를 리턴하고, 변경사항이 없으면 이전 객체를 리턴한다.

### 주소비교.

리덕스는 두 객체(prevState,newState)의 메모리 위치를 비교하여 이전 객체가 새 객체와

동일한지 여부를 단순 체크한다. 만약 리듀서 내부에서 이전 객체의 속성을 변경하면 새 상태 와 이전 상태가 모두 동일한 객체를 가리킨다. 그렇게 되면 리덕스는 아무것도 변경되지 않았다고 판단하여 동작하지 않는다.

### ↓ 속도상의 문제 해결

```
function counter(state = initialState, action)  
{ switch(action.type) { case types.INCREMENT:  
    return { ...state, number: state.number + 1 };  
  case types.DECREMENT: return { ...state,  
    number: state.number - 1 }; default: return  
    state; } }
```

본번상태, 새로운주소에  
→ Value 수정후  
하고 . → 상태변화

출처: <https://boxfoxs.tistory.com/406> [박스여우 - BoxFox]

createSlice 에는 객체를 넣을 때도 된다? why?

```
const toDoSlice = createSlice({  
    name: 'todo',  
    initialState,  
    reducers: {  
        addTodo(state, action){  
            state.push({id:uuid(), text: action.payload, done: false })  
        },  
  
        delTodo(state, {payload}){  
            return state.filter((todo) => todo.id !== payload)  
            // return state.findIndex(todo => todo.id !== payload.id)  
            // state.splice(state.findIndex(j => j.id === payload))  
        },  
        toggleTodo(state, {payload}) {  
            const todo = state.find((todo) => todo.id === payload)  
            todo ? (todo.done = !todo.done) : (todo.done = todo.done)  
        }  
    }  
})
```

## Inner ~~state~~의

```
const nextState = {
  ...state,
  posts: state.posts.map(post
=>
  post.id === 1
  ?{
    ...post,
    comments:
      post.comments.concat({
        id: 3,
        text: '새로운 댓글'
      })
  }
  :(post)
);
};
```

```
const nextState =
produce(state, draft => {
  const post =
    draft.posts.find(post => post.id
    === 1);
  post.comments.push({
    id: 3,
    text: '와 정말 쉽다!'
  });
});
```