스벨트 정리 #2

## 계산된 스토어

계산된 스토어(derived) 는 다른 스토어와 달리 초기 설정 값으로, 스토어 객체의 값을 전달 받는다. 전달 받은 스토어 객체가 업데이트 될 때, 2번째 인수로 전달한 콜백으로 이 계산된 스토어의 값을 set한다.

계산된 스토어 : 기본

./store.js

import {writable,derived } from 'svelte/store'

export const countStore = writable(0);

export const doubleStore = derived(countStore,(c) => c \* 2)

derived stroe의 첫번째 인자는 store객체를 전달하고, 두번째 인자는 전달 받은 스토어가 업데이트가 발생 될 때 마다, 실행할 콜백을 전달한다. 이 콜백의 return 값은 derivedStore의 새로운 값이 된다.

./App.svelte

<script>

import { countStore, doubleStore } from "./store";

</script>

<h1 on:click={() => ($countStore += 1)}>

count : {$countStore}

</h1>

<h1>

double : {$doubleStore}

</h1>

계산된 스토어 : 2개이상의 스토어 반응

2개 이상의 스토어를 계산된 스토어에 적용 시키고 싶을 때는, 첫번째 인자로 스토어 객체들을 담은 배열들을 전달한다. 이 경우 2번째 인수의 콜백은 기존과는 다르게 다음과 같이 변경된다.

type callBack = ([] : Store[], set : (value : set) : void ) => () => void

./store

import {writable,derived } from 'svelte/store'

export const countStore = writable(0);

export const doubleStore = derived(countStore,(c) => c \* 2);

export const tripleStore = derived([countStore,doubleStore],([c,d],set)=>{

set(c \* d)

return () => {

console.log("구독자가 0명인 경우")

}

})

## Store 값 얻기

Store의 값을 .svelte 파일 이외에서 받기 위해서는 지금까지 배운 subscribe를 이용해 똥꼬쇼를 해야한다. 이를 위해 get이라는 함수가 svelt 에 존재하는데, 이 함수는 첫번째 인자로 스토어 객체를 받아, 이 store객체의 값을 리턴 받을 수 있다.

import {writable,derived,get } from 'svelte/store'

export const countStore = writable(0);

export const doubleStore = derived(countStore,(c) => c \* 2);

export const tripleStore = derived([countStore,doubleStore],([c,d],set)=>{

set(c \* d)

return () => {

console.log("구독자가 0명인 경우")

}

})

console.log(get(countStore))

console.log(get(doubleStore))

console.log(get(tripleStore))

## 커스텀 스토어 개념

스토어 객체의 메서드는 단 3개 뿐이다. 이 3개 메서드를 포함하고 있는 객체를 만들어, 좀 더 사용하기 좋은 객체를 만드는 것이 커스텀 객체이다.

./store.js

import {writable } from 'svelte/store'

const { set, update, subscribe } = writable(0);

export const countStore = {

set,

update,

subscribe,

increase : () => update( n => n+=1),

decrease : () => update( n => n-=1),

reset : () => set(0)

}

./App.svelte

<script>

import { countStore } from "./store";

</script>

<button on:click={countStore.increase}> 증가 </button>

<button on:click={countStore.decrease}> 감소 </button>

<button on:click={countStore.reset}> 리셋 </button>

<h1>{$countStore}</h1>

## Action 사용 패턴 정리

스벨트에서 component에 use옵션을 붙여 해당 노드 요소를 가져 오는 것을 액션이라고 한다 ? . 이 액션을 통해서 노드 요소를 동적으로 변경 할 수 있고, 노드 요소가 변경되거나, 삭제 되었을 때 연결할 콜백을 작성 할 수 있다.

기본 동작 예제

<script>

function hello(node){

console.log(node);

node.style.width = "100px";

node.style.height = "100px";

node.style.backgroundColor = "tomato";

}

</script>

<div use:hello ></div>

Option 데이터 연결 하기.

<script>

function hello(node,option = {}){

const { width , height, background} = option;

node.style.width = width;

node.style.height = height;

node.style.backgroundColor = background;

}

</script>

<div use:hello={{

width : "100px",

height : "100px",

background : "royalblue"

}} ></div>

<div use:hello={{

width : "100px",

height : "100px",

background : "tomato"

}} ></div>

노드 요소를 반응성을 가지게 끔 변경하기

<script>

let width = 100;

function hello(node,option = {}){

const { width , height, background} = option;

node.style.width = width;

node.style.height = height;

node.style.backgroundColor = background;

return {

update : (option) => {

node.style.width = option.width

}

}

}

$ : console.log(width)

</script>

<button on:click={() => (width += 100)}> Increase </button>

<div

use:hello={{

width: "100px",

height: "100px",

background: "royalblue",

}}

/>

<div

use:hello={{

width: `${width}px`,

height: "100px",

background: "tomato",

}}

/>

Hello에 옵션으로 전달한 객체는 리터럴로 작성되었기 때문에, width가 증가하여도 업데이트가 발생하지 않는다. 이 경우, 정의한 hello함수에서 update를 가지는 객체를 return 하여야 한다. 이 업데이트 메서드는 hello로 전달되는 값이 변경 될 때 마다, 호출되며 첫번째 인자로 새롭게 전달되는 option의 데이터를 전달 받는다.

엘리먼트가 destroy될 때 발생하는 콜백 만들기

<script>

let toggle = false;

let width = 100;

function hello(node, option = {}) {

const { width = "199px", height = "199px", background = "red" } = option;

node.style.width = width;

node.style.height = height;

node.style.backgroundColor = background;

return {

update: (option) => {

node.style.width = option.width;

},

destroy: () => {

console.log("박살 ");

},

};

}

$: console.log(width);

</script>

<button on:click={() => (width += 100)}> Increase </button>

<button on:click={() => (toggle = !toggle)}>토글</button>

<div

use:hello={{

width: "100px",

height: "100px",

background: "royalblue",

}}

/>

{#if toggle}

<div

use:hello={{

width: `${width}px`,

height: "100px",

background: "tomato",

}}

/>

{/if}

Action 함수가 리턴하는 객체의 메서드로 destroy도 작성 할 수 있다. 이 메서드는 컴포넌트가 연결 해제되었을 경우, 발생하는 메서드 이다.

## 컴포넌트 재귀 호출

React와 달리, 스벨트 컴포넌트는 파일명으로 컴포넌트를 인식하기 때문에, 컴포넌트에서 재귀 호출을 발생 시킬 수 없다. 그래서 스벨트에서는 특별한 컴포넌트인 svelte:self 태그를 이용하여, 자자신의 컴포넌트를 호출 할 수 있다.

./App.svelte

<script>

import Self from "./Self.svelte";

const address = {

label: "대한민국",

child: [

{

label: "경기도",

child: [

{

label: "오산",

},

],

},

{

label: "경상북도",

child: [

{

label: "대구",

},

],

},

],

};

</script>

<Self label={address.label} child={address.child}></Self>

./Self.svelte

<script>

export let label;

export let child;

</script>

<ul>

<li>

<h1>{label}</h1>

</li>

{#if child}

{#each child as { label, child }}

<svelte:self {label} {child} />

{/each}

{/if}

</ul>

재귀 호출이 그러하듯, 컴포넌트가 무한정으로 재귀하여 오류를 일으키지 않기 위해서는 적절한 조건으로 컴포넌트 호출을 막아야 한다.

## 동적 컴포넌트 렌더링

조건에 따라 컴포넌트를 렌더링 하기 위해서는 {#if}문으로 컴포넌트를 떡칠 해야 한다. 만약 이렇게 된 경우, 컴포넌트가 늘어날수록 관리가 매우 힘들며, 코드가 if문으로 매우 오염된다. 그래서 보통 리액트에서는 switch 구문을 통해서, 동적으로 컴포넌트 렌더링을 진행하는데 스벨트에서는 이보다 더 편리한 svelte:component 태그를 지원한다.

<script>

import Neo from './Neo.svelte'

import Heropy from './Heropy.svelte';

const components = [

{name : "Neo", comp : Neo},

{name : "Heropy", comp : Heropy},

]

let index = 1;

let selected = components[0].comp;

</script>

{#each components as {name,comp},i}

<input type="radio"

value={comp}

bind:group={selected}

on:change={() => index = i + 1}

>{name}

{/each}

<svelte:component this={selected} {index}/>

./Neo.svelte

<script>

export let index;

</script>

<h2>{index} Neo</h2>

./Heropy.svelte

<script>

export let index;

</script>

<h2>{index}heropy</h2>

## Svelte:window

윈도우 객체를 다루는 가장 기본적인 방법은 window객체를 사용하는 것이다. 그러나, window객체에서 연결할 이벤트가 많아지면 코드는 오염된다. 이를 방지하기 위해서 스벨트에는 window 태그가 존재하는데, 이 태그는 window객체와 연결되며, 다양한 윈도우 객체의 프로퍼티를 변수와 바인딩 할 수 있게끔 도와준다.

<script>

let key = ""

let innerWidth;

let innerHeight;

let scrollX;

let scrollY;

let outerWidth;

let outerHeight;

let online; //연결이 끊기면 false로 변경된다.

</script>

<svelte:window

on:keydown={e => key = e.key}

bind:innerHeight

bind:innerWidth

bind:scrollX

bind:scrollY

bind:online

bind:outerHeight

bind:outerWidth

/>

<h1>{key}</h1>

바인딩 가능한 프로퍼티는 위의 7가지 이다.

## Svelte head, body

Svelte:Window처럼 스벨트에는 head와 body를 지정할 수 있는 컴포넌트가 존재한다.

./App.svelte

<script>

import Neo from './Neo.svelte'

import Heropy from './Heropy.svelte'

let toggle = false;

</script>

{#if toggle}

<Neo></Neo>

<Heropy></Heropy>

{/if}

<button on:click={() => toggle = !toggle}>Toggle</button>

<h1>

기본 색상은 블랙

</h1>

./dummy.css

h1{

color : red

}

./Neo.svelte

<svelte:head>

<link rel="stylesheet" href="./dummy.css">

</svelte:head>

이 head태그 안에 있는 요소들은 index.html의 재구성 하는 것이 아니라, 기존 head에 추가만 된다.

./Heropy.svelte

<script>

let key;

</script>

<svelte:body

on:keydown={e => key = e.key }

/>

<h1>

{key}

</h1>

## Svelte : option 불변성 선언

자바스크립트에는 가변성 데이터와 불변성 데이터가 존재한다. 예를 들어 스트링 타입 “hokeys” 와 “hokeys”는 일치 한다. 이처럼 새롭게 생성한 값이 기존 값과 동일하다면 일치하는 것으로 보는 타입을 불변성 데이터라고 한다. 그러나, {name : “hokeys”} 와 {name : “hokeys”}는 불일치 한다. 왜 그런가 ? 그 이유는 객체 데이터는 메모리 주소를 리턴하기 때문에, 새로 만들어진 객체의 메모리주소와 기존 객체의 메모리 주소를 비교 했을 때 불일치 하기 때문이다. 이를 가변성 데이터라고 한다.

스벨트도 불변성 데이터가 아닌 가변성 데이터를 프롭스로 전달한 경우, 기존 전달 받은 객체와 새롭게 전달 받은 객체가 동일하여도, 컴포넌트를 업데이트 한다. 이는 애초에 가변성 데이터로 전달되었기 때문에 새로운 프롭스 또한 새로운 데이터로 판별 하기 떄문이다. 이를 막기위해 svelte:option 태그에는 immutable이라는 속성이 존재하는데, 이 속성을 true로 변경하게 되면, props로 전달되는 가변성 데이터가 다른 데이터일 때만 컴포넌트를 렌더링 한다.

./App.svelte

<script>

import Neo from './Neo.svelte'

let fruits = [

{id : 0, name : "사과"},

{id : 1, name : "배"},

{id : 2, name : "딸기"},

{id : 3, name : "망고"},

{id : 4, name : "오렌지"},

]

</script>

<button on:click={() => {

fruits[0] = {id : 0, name : "사과"};

fruits = fruits;

}}>

클릭

</button>

{#each fruits as fruit (fruit.id)}

<Neo {fruit} immutable={true} ></Neo>

{/each}

./Neo.svelte

<script>

import { afterUpdate } from "svelte";

export let fruit;

let count = 0; //update횟수를 알기 위한 count

afterUpdate(()=>{

count += 1

})

</script>

<svelte:options

immutable={true}

/>

<h1>{fruit.name}({count})</h1>

막약 이를 방지 하고 싶다면, 객체를 전개하여, 불변성 데이터로 전달하면된다.

## Svelte:option accessors

리액트에서는 상위 컴포넌트에서 하위 컴포넌트로 특정 패턴을 사용하지 않으면, 접근이 불가능하다. 스벨트에서는 svelte:option태그의 accessors 옵션을 true로 지정해줌에 따라 하위 컴포넌트에서 export 한 데이터들에 상위 컴포넌트가 접근 할 수 있다 .

./App.svelte

<script>

import Neo from './Neo.svelte'

let neo;

function handler(){

console.log(neo.name)

neo.sayHello()

}

</script>

<button on:click={handler}></button>

<Neo bind:this={neo}></Neo>

./Neo.svelte

<script>

export let name = "hokeys";

export let sayHello = () => name = "hello world";

</script>

<svelte:options accessors={true}/>

<h1>

{name}

</h1>