Day7 Pinia

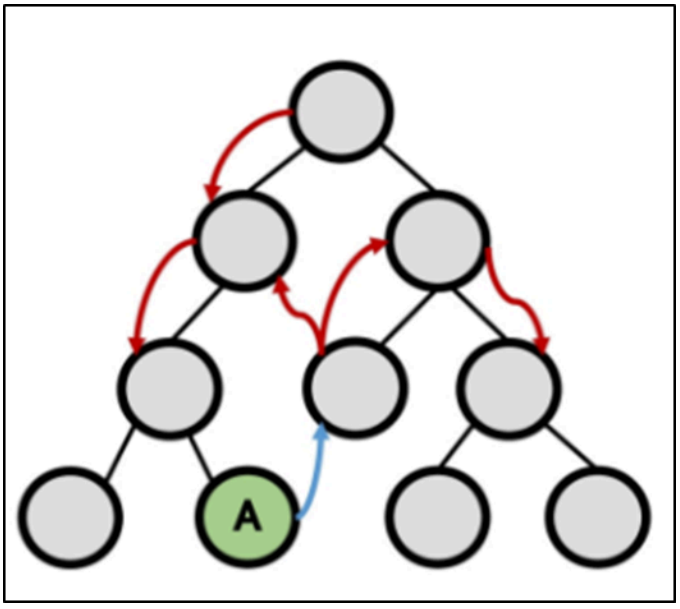
자료	Vue
구분	Vue
과목	

pinia

`props`와 `emit`을 사용하여 컴포넌트 간 데이터를 주고받는 방식은 Vue.js에서 흔하게 사용되는 패턴이다, 하지만 컴포넌트 간 데이터 전달이 확장되거나 복잡한 구조로 바뀔 경우 몇 가지 단점이 발생할 수 있다.

- 컴포넌트 구조가 복잡해질 경우
- 데이터를 주고받을 컴포넌트가 부모 자식 관계가 아닐 경우

그림으로 보면 다음과 같다.



이렇게, 컴포넌트 트리구조의 깊이가 길어질수록 `props` / `emit` 만으로 데이터를 다루기가 매우 어려워진다.

그래서 Vue 에서는 이러한 문제점을 극복하기 위한 다양한 방법들이 있다. 이벤트 method 들을 사용하기도 하거나 혹은 Props / emit과 비슷한 Provide / Inject 를 이용해서 '직계'

부모-자식 관계가 아니더라도 컴포넌트 간에 데이터를 주고 받는 방법이 있다.

하지만, Vue.js 공식 가이드 문서에서는 Provide / Inject 사용을 추천하지 않는다.

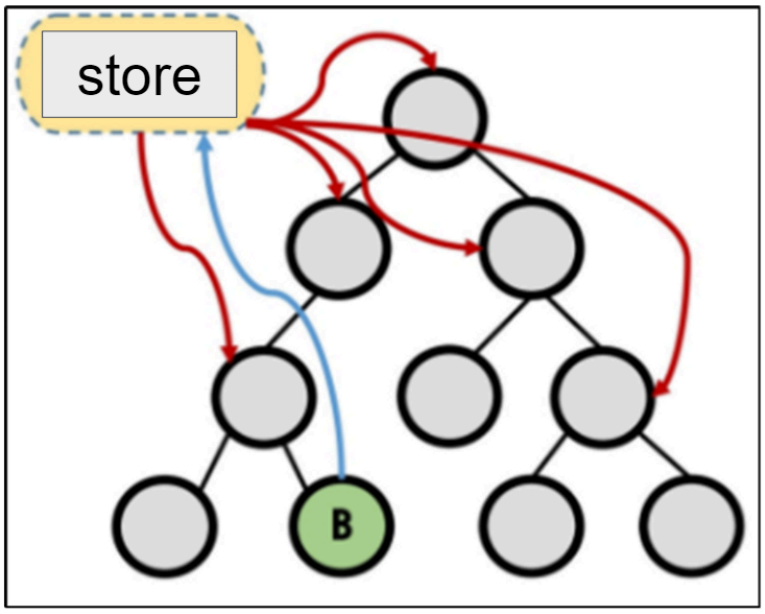
- **상세:**

> ! `provide`와 `inject`는 주로 고급 플러그인/컴포넌트 라이브러리를 위해 제공됩니다. 일반 애플리케이션 코드에서는 사용하지 않는 것이 좋습니다.
>
> 이 옵션 쌍은 함께 사용하여 상위 컴포넌트가 컴포넌트 계층 구조의 깊이에 관계없이 모든 하위 항목에 대한 종속성을 주입하는 역할을 하도록 허용합니다. React에 익숙하다면 이것은 React의 컨텍스트 기능과 매우 유사합니다.

프로젝트의 규모가 클 수록 Vue에서 기본으로 제공하는 기능 이외의 서드파티 패키지를 적극적으로 고려할 필요가 있다.

Vue3 공식문서에서는 Pinia 라는 상태 관리 패키지 (state management package) 를 사용하는 것을 강력 추천하고 있다.

Pinia 의 기본 아이디어는 다음과 같다.



`store` 라는 곳에 state를 저장해서, 각 컴포넌트에서 state 를 가져오거나, 변경한다.

`store` 를 사용 할 경우, 다음과 같이 두 가지 문제가 해결 될 것이다.

- 컴포넌트 구조가 복잡해지더라도, store 에 접근하면 원하는 데이터에 쉽게 접근가능해진다.
- 부모 자식 관계를 신경쓰지 않고, store 에 접근하면 된다.

프로젝트 진행 시 `props/emit` 그리고 `Pinia`를 많이 사용한다. 하지만 모든 state 관리를 Pinia 로만 작업하지는 않는다. 상황에 따라서 `props/emit` 그리고 `Pinia` 를 적절하게 섞어서 사용하면 된다. 컴포넌트가 복잡하지 않은 부모-자식 관계에서는 `props/emit` 을 사용하는 것이 비용이 적게 들고 성능상 유리하기 때문이다.

실습을 위해 프로젝트를 하나 만들고 초기 셋팅을 하자

```
$ npm create vue@latest

// 그리고 아래 옵션을 Yes로 체크하자 (라우터 그리고 pinia 옵션에 Yes 하기)

✓ Add Vue Router for Single Page Application development? Yes
✓ Add Pinia for state management? Yes
```

```
$npm install
```

```
// 그리고 필요없는 파일들을 삭제하자.

1. assets 폴더 안에 파일들
2. components 폴더안을 비우자
3. main.js 에서 import './assets/main.css' 지우기
```

```
4. App.vue 코드 지우고 아래 코드 남겨놓기

<template>
  <header>
    <nav>
      <RouterLink to="/">Home</RouterLink> |
      <RouterLink to="/about">About</RouterLink>
    </nav>
```

```
  </header>
  <RouterView />
</template>

<script setup>
import { RouterLink, RouterView } from 'vue-router'
</script>
```

`HomeView` `AboutView`컴포넌트 수정

`HomeView`

```
<template>
  <h1>Home</h1>
  <HomeChild />
</template>

<script setup>
import HomeChild from "@/components/HomeChild.vue";
</script>
```

`AboutView`

```
<template>
  <h1>About</h1>
</template>
```

그리고 components 폴더 안에

`HomeChild.vue` 그리고 `GrandChild.vue` 도 생성 해보자.

`HomeChild`

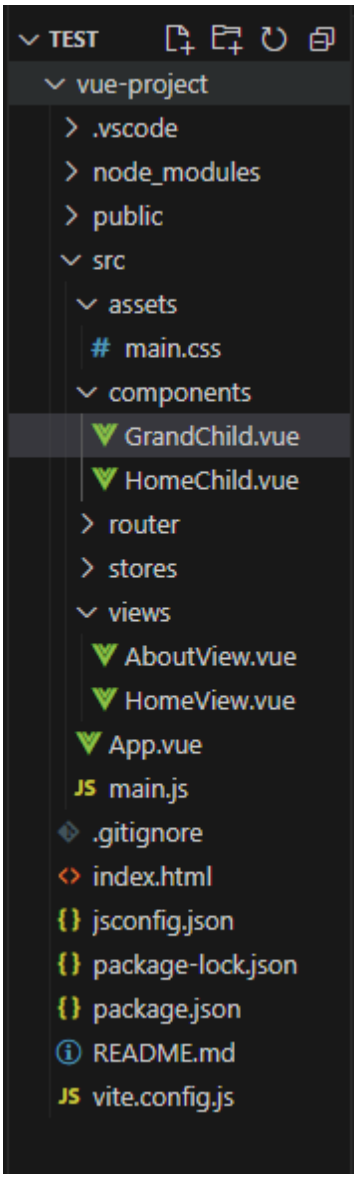
```
<template>
  <h2>Child</h2>
  <GrandChild />
</template>

<script setup>
```

```
import GrandChild from "@/components/GrandChild.vue";
</script>
```

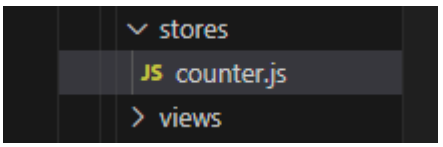
`GrandChild`

```
<template>
  <h3>GrandChild</h3>
</template>
```



`src/` 디렉터리를 클릭해서 열어보자. 해당 폴더 안에있는 `stores/` 디렉터리가 바로 `Pinia` 의 영역이이다.

`stores/` 디렉터리에는 프로젝트의 진행 중 필요에 따라 여러 개의 store가 존재할 수도 있다.



현재에는 `counter.js` 하나만 존재하며, 하나의 .js 파일 안에는

파일명과 일치하는 단 하나의 store, `counter` 가 존재할 것이다.

```
import { ref, computed } from 'vue'
import { defineStore } from 'pinia'

export const useCounterStore = defineStore('counter', () => {
  const count = ref(0)
  const doubleCount = computed(() => count.value * 2)
  function increment() {
    count.value++
  }

  return { count, doubleCount, increment }
})
```

기본으로 작성되어 있는 코드 내용을 하나씩 살펴 보자.

`export const useCounterStore = defineStore('counter', () =>`

- `defineStore` 함수를 사용해서 store를 정의한다.
  - `Pinia`는 `defineStore`라는 함수를 사용해서 각각의 파일마다 별도의 `store`를 정의한다.
- `defineStore` 의 첫번째 아규먼트는 store 의 이름이다. 파일명과 일치하게 네이밍을 하며, 이 이름은 전체 vue 프로젝트에서 중복되면 안된다.
- 두번째 아규먼트는 콜백함수다. 필요에 따라 state(상태), computed 그리고 action 등을 정의하며, 정의한 모든 것을 모아서 마지막에 리턴을 해줘야 한다.
  - pinia에서는 상태변경 및 로직을 처리하는 함수를 fuction이라고 부르기 보다는 action이라고 부른다.

다시 살펴보자.

`export const useCounterStore = defindStore('counter', ()⇒{`

- `defineStore` 는 컴포저블을 리턴하며, `use` + `Store이름` + `Store` 로 네이밍하고, 맨 앞에 `export` 를 붙여준다.
  - 컴포저블이란?
    Vue 3에서
    재사용 가능한 로직을 함수 형태로 정의하는것을 말한다.
  - https://v3-docs.vuejs-korea.org/guide/reusability/composables.html
- store는 컴포넌트에서 사용할 수 있는 "로직 덩어리"를 말한다. 이 로직 덩어리는 컴포넌트 안에서 마치 함수처럼 호출하여 사용할 수 있는데 나중에 실습을 통해서 살펴보자.

`useCounterStore` 컴포저블에서 정의하는 세 가지가 핵심이다.

- state : 지금까지 `ref` 로 정의해서 사용한 것과 같으며, 모든 컴포넌트에서 자유롭게 사용할 수 있다. 위의 예제에서는 `count` 에 해당한다.
- computed : state 의 값을 변경 시키지 않고, 다른 형태로 보여주기 위해 사용한다. 위에 작성된 예제 `doubleCount` 에서는 `count` 의 값을 변경 시키지는 않고, 두 배로 표시해서 보여준다.
- action : 특정한 로직을 정의하되, 반드시 state 변경과 관련이 있어야 한다. 위의 예에서는 `increment()` 함수가 될 것이다.

  (vue2에서는 increment()함수를 action이라는 객체 안에 정의 했지만, vue3부터는 action을 따로 작성하지 않아도 된다)

[중요]

즉, computed 와 action 의 차이는 state 를 실제로 변경 시키는지 아닌지의 차이라고 보면 된다. computed 는 state 를 특정한 처리를 해서 "보여줄" 뿐이고, action 은 state 를 반드시 "변경해야" 한다.

이제 컴포넌트에서 직접 사용해보자. `HomeView` 를 다음과 같이 정의한다.

```
<template>
  <h1>Home</h1>
  <div>count: {{ counter.count }}</div>
  <div>doubleCount: {{ counter.doubleCount }}</div>
  <HomeChild />
```

```
</template>

<script setup>
import { useCounterStore } from "@/stores/counter";
import HomeChild from "@/components/HomeChild.vue";

const counter = useCounterStore();

counter.count = 3;
counter.increment();
</script>
```

- import { useCounterStore } from "@/stores/counter";

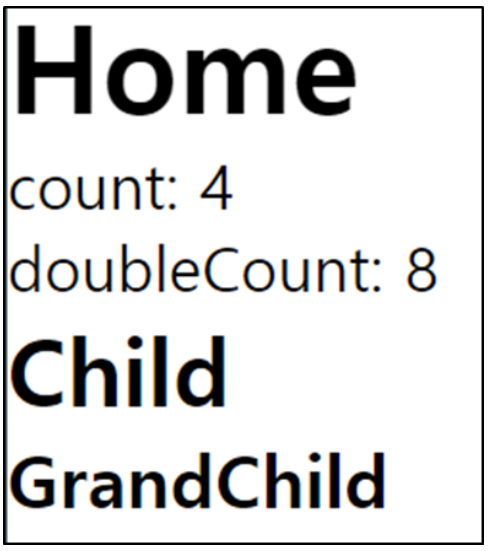
  `useCounterStore` 컴포저블을 store 경로에서 import를 먼저 한다.

- const counter = useCounterStore();

  `counter` 라는 객체를 하나 리턴 받는다.

  여기서 `counter` 는 `defineStore` 의 첫번째 아규먼트로 정의한 `counter` 라는 스토어 이름과 일치해야 한다.

- 각 구문별 사용예제
  - `counter.count = 3` state 직접 변경
  - `counter.increment()` 함수 실행. `counter.count` 는 1 증가할 것임
  - `{{ counter.count }}` , `{{ counter.doubleCount }}` 평소에 state 출력하듯 그대로 쓰면 된다.



Pinia 의 최대 장점은 더이상 `props` / `emit` 처럼 컴포넌트가 부모-자식 관계일 필요가 없다는 것이다. 아무 컴포넌트에서나 원하는 state, computed, action 을 사용할 수 있다.

- 컴포넌트 안에서 `counter.value++` 코드를 직접 사용할 수도 있지만, 굳이 count 값을 증가시키는 함수를 따로 만드는 이유는 무엇일까?

  state는 매우 중요한 데이터이기 때문에 아무 때나 직접 접근하도록 허용하면 여러 가지 문제가 발생할 수 있다. 상태를 직접 수정하는 대신 함수를 사용하면, 일관된 방식으로 상태를 변경할 수 있고, 동일한 로직을 여러 곳에서 중복하지 않고 한 곳에서만 관리할 수 있다.

  따라서 함수를 사용하면 버그를 줄이고 유지보수를 쉽게 할 수 있다는 장점이 있다.

이번에는 `GrandChild` 컴포넌트를 다음과 같이 변경해보자.

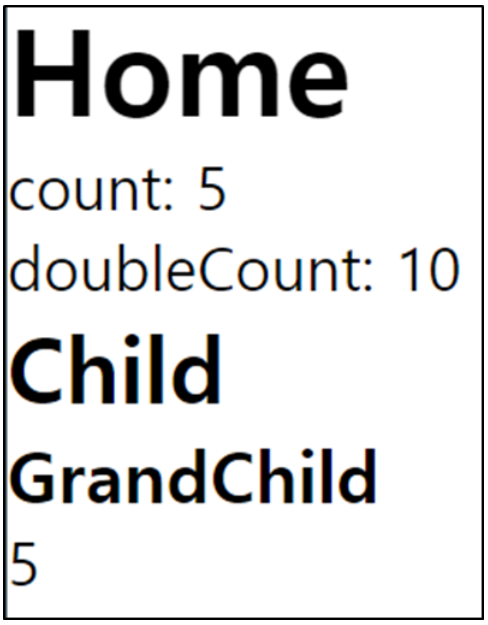
```
<template>
  <h3>GrandChild</h3>
  {{ counter.count }}
</template>

<script setup>
import { useCounterStore } from "@/stores/counter";

const counter = useCounterStore();
```

```
counter.count++;
</script>
```

`counter.count` 를 1 증가해보았다.



`GrandChild` 뿐만 아니라 `HomeView` 에서도 즉시 반영 되는 것을 알 수 있다.

`counter.value++` 를 직접 컴포넌트 안에 작성함으로써 store안에 있는 state값을 직접 바꿀 수 있지만 위 방법은 유지보수 차원에서 좋은 방법은 아니라고 했다.

이번에는 counter.js파일로 가서 `name` state 를 하나 더 추가해 보자.

아규먼트를 받아 state 를 변경하는 함수를 만들어보자.

```
import { ref, computed } from "vue";
import { defineStore } from "pinia";

export const useCounterStore = defineStore("counter", () => {
  const count = ref(0);
  const name = ref("minho");
  const doubleCount = computed(() => count.value * 2);
```

```
  function increment() {
    count.value++;
  }
  function changeName(newName) {
    name.value = newName;
  }

  return { count, name, doubleCount, increment, changeName };
});
```

`name` state 를 하나 추가했다. 즉, 하나의 store 는 여러 개의 state 를 가질 수 있다.

`changeName` function 도 정의했다. `newName` 로 인자값을 받아 `name` state 를 변경한다.

`return` 에 `name` , `changeName` 을 추가하는 것도 잊지 말자.

이번엔 `HomeChild` 에서 사용해보자.

```
<script setup>
import { useCounterStore } from "@/stores/counter.js";
import GrandChild from "@/components/GrandChild.vue";

const counter = useCounterStore();
counter.changeName("ssafy");
</script>

<template>
  <h2>Child</h2>
  {{ counter.name }}
  <GrandChild />
</template>
```

중간에 랜더링이 업데이트가 안될 수 있으니 새로고침을 자주 해주자.

원래 `name` 의 값은 `minho` 이였지만 `changeName` 에 의해 `ssafy` 로 잘 변경되어 출력 되는 것을 확인했다.

추가로, 만약 서버로부터 store 의 데이터를 가져올 일이 있다면, function에 axios 비동기 구문으로 작성해두었다가 호출하는 방식으로 사용이 될 것이다.