상태 관리

상태 관리란?

기술적으로 모든 Vue 컴포넌트 인스턴스는 이미 자체 반응형 상태를 "관리"합니다. 간단한 카운터 컴포넌트를 예로 들어 보겠습니다:

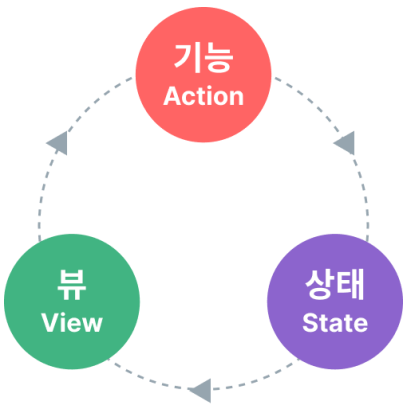
vue

```vue
<script setup>
import { ref } from 'vue'

// 상태(State)
const count = ref(0)

// 기능(Actions)
function increment() {
  count.value++
}
</script>

<!-- 뷰(view) -->
<template>{{ count }}</template>
```

이것은 다음과 같이 나눌 수 있는 독립된 장치 입니다:

상태(State): 앱 구동에 필요한 기본 데이터 소스.
뷰(View): 상태를 선언적으로 매핑하여 시각화.
기능(Actions): 뷰에서 사용자 입력에 대해 반응적으로 상태를 변경할 수 있게 정의된 동작.

이것은 "단방향 데이터 흐름" 개념의 간단한 표현입니다:



그러나 **여러 컴포넌트가 상태를 공유할 때** 단순성이 무너지기 시작합니다:

여러 뷰가 동일한 상태에 따라 달라질 수 있습니다.
서로 다른 뷰의 기능이 동일한 상태를 변경시킬 필요가 있을 수 있습니다.

사례 1의 경우, 가능한 해결 방법은 공유 상태를 공통 조상 컴포넌트로 "끌어올린" 다음 props로 전달하는 것입니다. 그러나 이것은 깊은 계층 구조를 가진 컴포넌트 트리에서 비효율적이 되며 **Prop** 드릴링으로 알려진 또 다른 문제로 이어집니다.

사례 2의 경우, 템플릿 refs를 통해 직접적인 부모/자식 인스턴스에 도달하거나, 발송(emit)된 이벤트를 통해 상태의 여러 복사본을 변경 및 동기화하려는 것과 같은 솔루션에 의존하는 경우가 많습니다. 이러한 패턴은 모두 깨지기 쉽고 빠르게 유지 관리할 수 없는 코드로 이어집니다.

더 간단하고 직관적인 솔루션은 컴포넌트에서 공유 상태를 추출하고 전역 싱글톤에서 관리하는 것입니다. 이를 통해 컴포넌트 트리는 큰 "뷰"가 되고 모든 컴포넌트는 트리의 위치에 관계없이 상태에 접근하거나 작업을 트리거할 수 있습니다!

반응형 API를 통한 간단한 상태 관리

여러 인스턴스에서 공유해야 하는 상태가 있는 경우, **reactive()** 를 사용하여 반응형 객체를 만든 다음 여러 컴포넌트에서 가져갈 수 있습니다.

```js
// store.js
import { reactive } from 'vue'

export const store = reactive({
  count: 0
})
```

```vue
<!-- ComponentA.vue -->
<script setup>
import { store } from './store.js'
</script>

<template>A 컴포넌트에서: {{ store.count }}</template>
```

```vue
<!-- ComponentB.vue -->
<script setup>
import { store } from './store.js'
</script>

<template>B 컴포넌트에서: {{ store.count }}</template>
```

이제 store 객체가 변경될 때마다 <ComponentA> 와 <ComponentB> 가 뷰를 자동으로 업데이트합니다. 이제 단일 기본 데이터 소스가 되었습니다.

그러나 이것은 store 를 가져오는 모든 컴포넌트가 원하는 대로 변경할 수 있음을 의미합니다.

```template
<template>
  <button @click="store.count++">
    B 컴포넌트에서: {{ store.count }}
  </button>
</template>
```

이것은 간단한 경우에는 문제없이 작동하지만, 컴포넌트에 의해 임의로 변경될 수 있는 전역 상태이므로 장기적인 유지 관리는 쉽지 않습니다. 상태 변경 로직이 상태 자체처럼 중앙 집중화되도록 하려면, 작업의 의도를 나타내는 이름으로 store 에 메서드를 정의하는 것이 좋습니다:

```js
// store.js
import { reactive } from 'vue'

export const store = reactive({
  count: 0,
  increment() {
    this.count++
  }
})
```

```template
<template>
  <button @click="store.increment()">
    B 컴포넌트에서: {{ store.count }}
  </button>
</template>
```

온라인 연습장으로 실행하기

TIP

클릭 핸들러는 괄호를 포함한 `store.increment()` 를 사용합니다. 이것은 컴포넌트 메서드가 아니기 때문에, 적절히 `this` 컨텍스트를 사용하여 메서드를 호출할 필요가 있습니다.

여기에서는 단일 반응형 객체를 `store` 로 사용하고 있지만, `ref()` 또는 `computed()` 와 같은 다른 반응형 **API**를 사용하여 생성된 반응형 상태를 공유하거나 컴포저블을 통해 전역 상태를 반환할 수도 있습니다:

```js
import { ref } from 'vue'

// 모듈 범위에서 생성된 전역 상태
const globalCount = ref(1)

export function useCount() {
  // 로컬 상태, 컴포넌트별로 생성됨
  const localCount = ref(1)

  return {
    globalCount,
    localCount
  }
}
```

Vue의 반응형 시스템이 컴포넌트 모델과 분리되어 있다는 사실은 Vue를 매우 유연하게 만듭니다.

SSR 고려 사항

서버 사이드 렌더링(SSR)을 활용하는 앱을 구축하는 경우, 위 패턴은 `store` 가 여러 요청에서 공유되는 싱글톤이기 때문에 문제를 일으킬 수 있습니다. 이것에 대한 자세한 내용은 **SSR** 가이드에서 설명합니다.

Pinia(피니아: 공식 상태관리 라이브러리)

간단한 시나리오에서는 수동 상태 관리 솔루션으로 충분하지만, 대규모 프로덕션 앱에서는 고려해야 할 사항이 더 많습니다:

- 팀 협업을 위한 더 강력한 규칙
- 타임라인, 컴포넌트 검사, time-travel 디버깅을 포함하여 Vue 개발자도구와 통합
- 핫 모듈 교체(HMR)
- 서버 사이드 렌더링 지원

Pinia는 위의 모든 것을 구현하는 상태 관리 라이브러리입니다. Vue 핵심 팀이 유지 관리하며 Vue 2 및 Vue 3에서 모두 작동합니다.

기존 Vue 사용자는 이전 공식 상태 관리 라이브러리인 **Vuex**에 익숙할 수 있습니다. Pinia가 생태계에서 동일한 역할을 수행하면서 이제 Vuex는 유지 관리 상태에 있습니다. 여전히 작동하지만 더 이상 새로운 기능이 추가되지 않습니다. 새로운 앱에는 Pinia를 사용하는 것이 좋습니다.

Pinia는 Vuex 5의 핵심 팀 논의에서 나온 많은 아이디어를 통합하여 Vuex의 다음이 어떤 모습일지 탐구하는 것으로 시작했습니다. 결국 우리는 Pinia가 이미 Vuex 5에서 원하는 것을 대부분 구현하고 있다는 것을 깨닫고 이것을 새로운 권장 사항으로 적용하기로 결정했습니다.

Vuex와 비해 Pinia는 더 간단한 API를 제공하고, Composition-API 스타일의 API를 제공하며, 가장 중요한 것은 TypeScript와 함께 사용할 때 견고한 유형 추론을 지원합니다.