컴포지션 API FAQ

TIP

이 FAQ는 Vue 2를 사용해본 경험이 있거나, 주로 옵션 API를 사용하고 있다는 것을 전제로 합니다.

컴포지션 API란?

Vue School의 무료 동영상 강의 보기

컴포지션(Composition) API는 옵션을 선언하는 대신 import 한 함수를 사용하여 Vue 컴포넌트를 작성할 수 있는 API 세트입니다. 이것은 아래 API를 다루는 포괄적인 용어입니다:

- 반응형(Reactivity) API: 예를 들어 ref() 및 reactive() 를 사용하여 반응형 상태, 계산된 상태 및 감시자를 직접 생성할 수 있습니다.
- 생명주기 훅: 예를 들어 onMounted() 및 onUnmounted() 를 사용하여 컴포넌트 생명주기에 프로그래밍 방식으로 연결할 수 있습니다.
- 의존성 주입(Dependency Injection): provide() 및 inject() 를 사용하면 반응형 API를 사용하는 동안 Vue의 의존성 주입 시스템을 활용할 수 있습니다.

컴포지션 API는 Vue 3 및 Vue 2.7에 내장된 기능입니다. 이전 Vue 2 버전의 경우 공식적으로 유지 관리되는 @vue/composition-api 플러그인을 사용하십시오. Vue 3에서는 주로 싱글 파일 컴포넌트에서 <스크립트 설정> 구문과 함께 사용되기도 합니다. 다음은 컴포지션 API를 사용하는 컴포넌트의 기본 예시입니다:

vue

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

// 반응형 상태
const count = ref(0)

// 상태를 변경하고 업데이트를 트리거하는 함수
function increment() {
  count.value++
}

// 생명주기 훅
onMounted(() => {
  console.log(`숫자를 세기 위한 초기값은 ${count.value} 입니다.`)
})
</script>

<template>
  <button @click="increment">숫자 세기: {{ count }}</button>
</template>
```

함수 구성에 기반한 API 스타일에도 불구하고 **컴포지션 API는 함수형 프로그래밍이 아닙니다**. 컴포지션 API는 Vue의 변경 가능하고 세분화된 반응성 패러다임을 기반으로 하는 반면 함수형 프로그래밍은 불변성을 강조합니다.

Vue를 컴포지션 API와 함께 사용하는 방법을 배우고 싶다면 왼쪽 사이드바 상단의 토글을 사용하여 사이트 전체 API 환경 설정을 컴포지션 API로 설정한 다음 처음부터 가이드를 살펴보세요.

왜 컴포지션 API인가요?

더 나은 로직 재사용성

컴포지션 API의 가장 큰 장점은 컴포저블 함수의 형태로 깔끔하고 효율적인 로직 재사용이 가능하다는 것입니다. 옵션 API의 기본 로직 재사용 메커니즘인 믹스인의 모든 단점을 해결합니다.

컴포지션 API의 로직 재사용 기능은 컴포저블 유틸리티의 계속 성장하는 컬렉션인 **VueUse**와 같은 인상적인 커뮤니티 프로젝트를 탄생시켰습니다. 또한 상태 저장 타사 서비스 또는 라이브러리를 불변 데이터, 상태 머신 및 **RxJS**와 같은 Vue의 반응형 시스템에 쉽게 통합하기 위한 깔끔한 메커니즘 역할을 합니다.

보다 유연한 코드 구성

많은 사용자는 기본적으로 옵션 API를 사용하여 조직화된 코드를 작성하는 것을 좋아합니다. 그러나 옵션 API는 단일 컴포넌트의 논리가 특정 복잡성 임계값을 초과하는 경우 심각한 제한을 가집니다. 이 제한은 여러 프로덕션 Vue 2 앱에서 직접 목격한 여러 **논리적 문제**를 처리해야 하는 컴포넌트에서 특히 두드러집니다.

Vue CLI의 GUI에서 폴더 탐색기 컴포넌트를 예로 들어 보겠습니다. 이 컴포넌트는 다음과 같은 논리적 문제를 야기합니다:

- 현재 폴더 상태 추적 및 내용 표시
- 폴더 탐색 처리(열기, 닫기, 새로 고침...)
- 새 폴더 생성 처리
- 즐겨찾기 폴더만 표시 전환
- 숨김 폴더 표시 전환
- 현재 작업 디렉터리 변경 처리

컴포넌트의 원본 버전은 옵션 API로 작성되었습니다. 다루는 논리적 문제를 기준으로 코드의 각 라인에 색상을 지정하면 다음과 같이 표시됩니다:

동일한 논리적 문제를 처리하는 코드가 파일의 다른 부분에 분할되어 있습니다. 수백 줄 길이의 컴포넌트에서 단일 논리적 문제를 이해하고 탐색하려면 파일을 지속적으로 위아래로 스크롤해야 하므로 어렵습니다. 또한 논리적 문제를 재사용 가능한 유틸리티로 추출하려는 경우, 파일의 다른 부분에서 올바른 코드 조각을 찾아 추출하는 데 상당한 노력이 필요합니다.

다음은 **Composition API**로 리팩터링 전후의 동일한 컴포넌트입니다:

Options API

Composition API

동일한 논리적 문제와 관련된 코드가 어떻게 그룹화 되었는지 보십시오. 특정 논리적 문제를 해결하는 동안 더 이상 다른 옵션 블록 사이를 이동할 필요가 없습니다. 또한, 추출을 위해 더 이상 코드를 섞을 필요가 없기 때문에 최소한의 노력으로 코드 그룹을 외부 파일로 이동할 수 있습니다. 리팩토링을 위한 소모 시간 감소는 대규모 코드베이스에서 장기적인 유지 관리의 핵심입니다.

더 나은 타입 추론

최근 몇 년 동안 점점 더 많은 프론트엔드 개발자들이 **TypeScript**를 채택하고 있습니다. 이는 우리가 보다 강력한 코드를 작성하고, 보다 자신 있게 변경하고, IDE 지원을 통해 뛰어난 개발 경험을 제공하는 데 도움이 되기 때문입니다. 그러나 원래 2013년에 구상된 옵션 API는 유형 추론을 염두에 두지 않고 설계되었습니다. 유형 추론이 옵션 API와 함께 작동하도록 하기 위해 일부 터무니없이 복잡한 유형 구조를 구현해야 했습니다. 이 모든 노력에도 불구하고 옵션 API에 대한 유형 추론은 여전히 믹스인 및 의존성 주입에 대해 분해될 수 있습니다.

이로 인해 TS와 함께 Vue를 사용하고자 하는 많은 개발자가 `vue-class-component` 로 구동되는 클래스 API에 의존하게 되었습니다. 그러나 클래스 기반 API는 2019년 Vue 3 개발 당시 2단계 제안에 불과했던 언어 기능인 ES 데코레이터에 크게 의존합니다. 저희는 불안정한 제안을 기반으로 공식 API를 만드는 것은 너무 위험하다고 생각했습니다. 그 이후로 데코레이터 제안은 또 한 번의 전면적인 개편을 거쳐 2022년에 마침내 3단계에 도달했습니다. 또한 클래스 기반 API는 옵션 API와 마찬가지로 로직 재사용 및 구성 제한이 있습니다.

이에 비해 컴포지션 API는 기본적으로 유형 친화적인 일반 변수와 함수를 주로 사용합니다. 컴포지션 API로 작성된 코드는 수동 유형 힌트가 거의 필요 없이 전체 유형 추론을 즐길 수 있습니다. 대부분의 경우 컴포지션 API 코드는 TypeScript와 일반 JavaScript에서 거의 동일하게 보입니다. 이것은 또한 일반 JavaScript 사용자가 부분적인 유형 추론의 이점을 누릴 수 있도록 합니다.

더 작은 프로덕션 번들 및 더 적은 오버헤드

컴포지션 API와 `<script setup>`으로 작성된 코드는 옵션 API에 비해 더 효율적이고 축소하기 쉽습니다. 이는 `<script setup>` 컴포넌트의 템플릿이 `<script setup>` 코드의 동일한 범위에 인라인된 함수로 컴파일되기 때문입니다. `this`의 속성 접근과 달리 컴파일된 템플릿 코드는 인스턴스 프록시 없이 `<script setup>` 내부에 선언된 변수에 직접 접근할 수 있습니다. 이것은 모든 변수 이름을 안전하게 단축할 수 있기 때문에 더 나은 축소로 이어집니다.

옵션 API와의 관계

Trade-offs

옵션 API에서 넘어온 일부 사용자들은 그들의 컴포지션 API 코드가 덜 구성적이라 생각하고, 컴포지션 API가 코드 구성 측면에서 "더 나쁘다"고 결론짓습니다. 이러한 의견을 가진 사용자라면 해당 문제를 다른 관점에서 볼 것을 권장합니다.

컴포지션 API가 더 이상 코드를 해당 버킷에 넣도록 안내하는 "가드 레일"(`export default {}`)을 제공하지 않는다는 것입니다. 따라서 일반적인 JavaScript를 작성하는 것처럼 컴포넌트 코드를 작성할 수 있습니다. 그러므로 **일반적인 JavaScript를 작성할 때와 마찬가지로 모든 코드 구성 모범 사례를 컴포지션 API 코드에 적용할 수 있고 적용해야 합니다**. 잘 구성된 자바스크립트를 작성할 수 있다면 잘 구성된 컴포지션 API 코드도 작성할 수 있어야 합니다.

옵션 API를 사용하면 컴포넌트 코드를 작성할 때 "생각을 덜"할 수 있으므로 많은 사용자가 이 API를 좋아합니다. 정신적 피로도는 줄어들지만, 다양성 없이 규정된 코드 구성 패턴에 갇히게 되므로 대규모 프로젝트에서 코드 품질을 리팩토링하거나 개선하기 어려울 수 있습니다. 이와 관련하여 컴포지션 API는 더 나은 장기적인(거시적인) 확장성을 제공합니다.

Composition API가 모든 사용 사례를 다루는가요?

상태 로직에 관한 측면에서는 그렇습니다. Composition API를 사용할 때, `props`, `emits`, `name`, 그리고 `inheritAttrs`와 같은 몇 가지 옵션이 여전히 필요할 수 있습니다.

> TIP
>
> 3.3부터 `<script setup>`에서 `defineOptions`를 직접 사용하여 컴포넌트 이름 또는 `inheritAttrs` 속성을 설정할 수 있습니다.

위에 나열된 옵션과 함께 컴포지션 API를 독점적으로 사용하려는 경우, Vue에서 옵션 API 관련 코드를 삭제하는 컴파일 타임 플래그를 통해 프로덕션 번들에서 몇 kb를 줄일 수 있습니다. 이것은 의존성의 Vue 컴포넌트에도 영향을 미칩니다.

동일한 컴포넌트에서 두 API를 모두 사용할 수 있습니까?

예. 옵션 API 컴포넌트에서 `setup()` 옵션을 통해 컴포지션 API를 사용할 수 있습니다.

그러나 컴포지션 API로 작성된 새 기능 또는 외부 라이브러리와 통합해야 하는 기존 옵션 API 코드베이스가 있는 경우에만 그렇게 하는 것이 좋습니다.

옵션 API가 더 이상 사용되지 않습니까?

아니요, 그렇게 할 계획이 없습니다. 옵션 API는 Vue의 필수적인 부분이며 많은 개발자들이 Vue를 좋아하는 이유입니다. 또한 컴포지션 API의 많은 이점은 대규모 프로젝트에서만 나타나고, 옵션 API는 복잡성이 낮은 여러 시나리오에서 여전히 확실한 선택이라는 것을 알고 있습니다.

클래스 API와의 관계

컴포지션 API가 추가 로직 재사용 및 코드 구성 이점과 함께 뛰어난 TypeScript 통합을 제공한다는 점을 감안할 때 Vue 3에서 Class API를 더 이상 사용하지 않는 것이 좋습니다.

React 훅과의 비교

컴포지션 API는 React 훅과 동일한 수준의 로직 구성 기능을 제공하지만 몇 가지 중요한 차이점이 있습니다.

React 훅은 컴포넌트가 업데이트될 때마다 반복적으로 호출됩니다. 이것은 노련한 React 개발자도 혼동할 수 있는 여러 가지 주의 사항을 만듭니다. 또한 개발 경험에 심각한 영향을 줄 수 있는 성능 최적화 문제로 이어집니다. 여기 몇 가지 예가 있습니다:

- Hooks 는 호출 순서에 민감하여 조건부로 사용할 수 없습니다.

- 리액트 컴포넌트 내에 선언된 변수들은 훅 클로저에 포함되고 개발자가 올바른 의존성 배열을 넘기지 않으면 "낡은" 상태가 될 수 있습니다. 이로 인해 리액트 개발자들은 ESLint 규칙에 의존하여 의존성 배열이 잘못되지 않았는지 확인해야합니다. 하지만 이 규칙이 충분히 똑똑하지 않아서 종종 정확도를 과하게 요구하여 불필요한 무효화를 발생시키거나, 엣지케이스를 만났을때 두통을 일으키곤 합니다.

- 비용이 많이 드는 계산은 `useMemo` 사용을 필요로 하며, 이 역시 올바른 종속성 배열을 수동으로 전달해야 합니다.

- 자식 컴포넌트에게 전달된 이벤트 핸들러는 불필요한 자식 컴포넌트의 업데이트를 발생시키고, 최적화를 위해 명시적인 `useCallback` 사용을 요구합니다. 이는 거의 대부분의 경우에 필요하고, 또 다시 정확한 의존성 배열을 요구합니다. 이를 소홀히 하면 앱을 오버 렌더링하여 눈치채지 못한 채로 성능 이슈를 발생시킬 수도 있습니다.

- 낡은 클로저 문제와 동시성 기능이 합쳐지면, 훅의 코드가 언제 그리고 왜 실행되는지 파악하기 어려워지며, 여러번의 렌더에서 영속되어야 하는 변경 가능한 상태(`useRef` 를 활용한)와 관련된 작업하는 것을 번거롭게 만듭니다.

이에 비해 Vue 컴포지션 API는 다음과 같습니다:

- `setup()` 함수 호출이나 `<script setup>` 호출 모두 단 한번만 발생합니다. 이로 인해 낡은 클로저 문제에 대해 걱정할 필요가 없기 때문에, 직관적인 기존의 자바스크립트와 코드가 더 잘 어울립니다. 컴포지션 API는 호출 순서에 민감하지 않고, 조건부로 호출될 수도 있습니다.

- Vue 의 런타임 반응성 시스템은 computed 속성과 watcher 에 사용되는 반응성 종속성을 자동으로 수집하므로 종속성을 수동으로 선언할 필요가 없습니다.

- 불필요한 하위 컴포넌트의 업데이트를 피하기 위해 콜백 함수를 수동으로 캐시할 필요가 없습니다. 일반적으로 Vue의 세분화된 반응성 시스템은 필요할 때만 하위 컴포넌트가 업데이트되도록 합니다. 수동 하위 컴포넌트 업데이트 최적화는 Vue 개발자에게 거의 문제가 되지 않습니다.

우리는 React 훅의 창의성을 인정하며 이는 컴포지션 API에 주요 영감의 원천입니다. 그러나 위에서 언급한 문제는 설계에 존재하며, Vue의 반응형 모델이 이러한 문제를 해결할 수 있는 방법을 제공한다는 것을 알게 되었습니다.