옵션 API에서 TypeScript 사용하기

> 이 페이지는 TypeScript와 함께 Vue 사용하기 개요를 이미 읽었다고 가정합니다.

TIP

Vue는 옵션 API와 함께 TypeScript 사용을 지원하지만, 더 간단하고 효율적이며 견고한 타입 추론을 제공하는 컴포지션 API를 통해 TypeScript와 함께 Vue를 사용하는 것이 권장됩니다.

컴포넌트 Props 타입 지정하기

옵션 API에서 props의 타입 추론을 위해서는 컴포넌트를 `defineComponent()` 로 감싸야 합니다. 이를 통해 Vue는 `props` 옵션을 기반으로, `required: true` 나 `default` 와 같은 추가 옵션을 고려하여 props의 타입을 추론할 수 있습니다:

ts

```ts
import { defineComponent } from 'vue'

export default defineComponent({
  // 타입 추론 활성화
  props: {
    name: String,
    id: [Number, String],
    msg: { type: String, required: true },
    metadata: null
  },
  mounted() {
    this.name // 타입: string | undefined
    this.id // 타입: number | string | undefined
    this.msg // 타입: string
    this.metadata // 타입: any
  }
})
```

된 속성을 가진 객체나 함수 호출 시그니처와 같은 복잡한 타입을 지정할 방법은 없습니다.

복잡한 props 타입을 명시하려면 `PropType` 유틸리티 타입을 사용할 수 있습니다:

```ts
import { defineComponent } from 'vue'
import type { PropType } from 'vue'

interface Book {
  title: string
  author: string
  year: number
}

export default defineComponent({
  props: {
    book: {
      // `Object`에 더 구체적인 타입 제공
      type: Object as PropType<Book>,
      required: true
    },
    // 함수도 타입 지정 가능
    callback: Function as PropType<(id: number) => void>
  },
  mounted() {
    this.book.title // string
    this.book.year // number

    // TS 오류: 'string' 타입의 인수는
    // 'number' 타입의 매개변수에 할당할 수 없음
    this.callback?.('123')
  }
})
```

주의사항

TypeScript 버전이 `4.7` 미만인 경우, `validator` 와 `default` prop 옵션에 함수 값을 사용할 때 주의해야 합니다. 반드시 화살표 함수를 사용하세요:

```ts
import { defineComponent } from 'vue'
import type { PropType } from 'vue'

interface Book {
  title: string
  year?: number
}

export default defineComponent({
  props: {
    bookA: {
```

```
      // TypeScript 4.7 버전에서는 반드시 화살표 함수 사용
      default: () => ({
        title: 'Arrow Function Expression'
      }),
      validator: (book: Book) => !!book.title
    }
  }
})
```

이렇게 하면 이러한 함수 내부에서 `this` 의 타입을 TypeScript가 추론해야 하는 상황을 방지할 수 있습니다. 이는 이전의 설계 제한이었으며, TypeScript 4.7에서 개선되었습니다.

컴포넌트 Emits 타입 지정하기

`emits` 옵션의 객체 문법을 사용하여 발생시킬 이벤트의 예상 페이로드 타입을 선언할 수 있습니다. 또한, 선언되지 않은 모든 이벤트를 발생시키면 타입 오류가 발생합니다:

ts

```ts
import { defineComponent } from 'vue'

export default defineComponent({
  emits: {
    addBook(payload: { bookName: string }) {
      // 런타임 유효성 검사 수행
      return payload.bookName.length > 0
    }
  },
  methods: {
    onSubmit() {
      this.$emit('addBook', {
        bookName: 123 // 타입 오류!
      })

      this.$emit('non-declared-event') // 타입 오류!
    }
  }
})
```

계산된 속성 타입 지정하기

계산된 속성은 반환값을 기반으로 타입을 추론합니다:

```
export default defineComponent({
  data() {
    return {
      message: 'Hello!'
    }
  },
  computed: {
    greeting() {
      return this.message + '!'
    }
  },
  mounted() {
    this.greeting // 타입: string
  }
})
```

경우에 따라 계산된 속성의 타입을 명시적으로 지정하여 구현이 올바른지 보장하고 싶을 수 있습니다:

```ts
import { defineComponent } from 'vue'

export default defineComponent({
  data() {
    return {
      message: 'Hello!'
    }
  },
  computed: {
    // 반환 타입을 명시적으로 지정
    greeting(): string {
      return this.message + '!'
    },

    // 쓰기 가능한 계산된 속성에 타입 지정
    greetingUppercased: {
      get(): string {
        return this.greeting.toUpperCase()
      },
      set(newValue: string) {
        this.message = newValue.toUpperCase()
      }
    }
  }
})
```

명시적 타입 지정은 TypeScript가 순환 추론 루프로 인해 계산된 속성의 타입을 추론하지 못하는 일부 예외적인 경우에도 필요할 수 있습니다.

이벤트 핸들러 타입 지정하기

네이티브 DOM 이벤트를 다룰 때, 핸들러에 전달하는 인자의 타입을 올바르게 지정하는 것이 유용할 수 있습니다. 다음 예제를 살펴봅시다:

```vue
<script lang="ts">
import { defineComponent } from 'vue'

export default defineComponent({
  methods: {
    handleChange(event) {
      // `event`는 암시적으로 `any` 타입을 가짐
      console.log(event.target.value)
    }
  }
})
</script>

<template>
  <input type="text" @change="handleChange" />
</template>
```

타입 지정을 하지 않으면 `event` 인자는 암시적으로 `any` 타입을 갖게 됩니다. 이는 `tsconfig.json` 에서 `"strict": true` 또는 `"noImplicitAny": true` 가 사용될 경우 TS 오류로 이어집니다. 따라서 이벤트 핸들러의 인자를 명시적으로 타입 지정하는 것이 권장됩니다. 또한, `event` 의 속성에 접근할 때 타입 단언을 사용해야 할 수도 있습니다:

```ts
import { defineComponent } from 'vue'

export default defineComponent({
  methods: {
    handleChange(event: Event) {
      console.log((event.target as HTMLInputElement).value)
    }
  }
})
```

전역 속성 확장하기

일부 플러그인은 `app.config.globalProperties` 를 통해 모든 컴포넌트 인스턴스에 전역적으로 사용할 수 있는 속성을 설치합니다. 예를 들어, 데이터 패칭을 위한 `this.$http` 나 국제화를 위한 `this.$translate` 를 설치할 수 있습니다. TypeScript와 잘 연동되도록, Vue는 TypeScript

니다:

```ts
import axios from 'axios'

declare module 'vue' {
  interface ComponentCustomProperties {
    $http: typeof axios
    $translate: (key: string) => string
  }
}
```

참고:

컴포넌트 타입 확장에 대한 TypeScript 단위 테스트

타입 보강 위치

이 타입 보강은 `.ts` 파일이나 프로젝트 전체에 적용되는 `*.d.ts` 파일에 둘 수 있습니다. 어느 쪽이든 `tsconfig.json` 에 포함되어야 합니다. 라이브러리/플러그인 작성자의 경우, 이 파일을 `package.json` 의 `types` 속성에 지정해야 합니다.

모듈 보강을 활용하려면, 보강이 TypeScript 모듈 내에 위치해야 합니다. 즉, 파일에 최상위 `import` 나 `export` 가 하나 이상 있어야 하며, 단순히 `export {}` 만 있어도 됩니다. 모듈 외부에 보강을 두면 원래 타입을 덮어써버리므로 주의하세요!

```ts
// 동작하지 않으며, 원래 타입을 덮어씁니다.
declare module 'vue' {
  interface ComponentCustomProperties {
    $translate: (key: string) => string
  }
}
```

```ts
// 올바르게 동작함
export {}

declare module 'vue' {
  interface ComponentCustomProperties {
    $translate: (key: string) => string
  }
}
```

커스텀 옵션 확장하기

니다:

```ts
import { defineComponent } from 'vue'

export default defineComponent({
  beforeRouteEnter(to, from, next) {
    // ...
  }
})
```

적절한 타입 보강이 없으면 이 훅의 인자들은 암시적으로 `any` 타입을 갖게 됩니다. 이러한 커스텀 옵션을 지원하려면 `ComponentCustomOptions` 인터페이스를 확장할 수 있습니다:

```ts
import { Route } from 'vue-router'

declare module 'vue' {
  interface ComponentCustomOptions {
    beforeRouteEnter?(to: Route, from: Route, next: () => void): void
  }
}
```

이제 `beforeRouteEnter` 옵션이 올바르게 타입 지정됩니다. 이는 단순한 예시일 뿐이며, `vue-router` 와 같이 타입이 잘 지정된 라이브러리는 자체 타입 정의에서 이러한 보강을 자동으로 수행해야 합니다.

이 보강의 위치는 전역 속성 보강과 동일한 제한을 받습니다.

참고:

- 컴포넌트 타입 확장에 대한 TypeScript 단위 테스트