컴포지션 API와 TypeScript

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

컴포넌트 Props 타입 지정하기

`<script setup>` 사용하기

`<script setup>` 을 사용할 때, `defineProps()` 매크로는 인자로 전달된 값에 따라 props 타입을 추론할 수 있습니다:

```vue
<script setup lang="ts">
const props = defineProps({
  foo: { type: String, required: true },
  bar: Number
})

props.foo // string
props.bar // number | undefined
</script>
```

이것을 "런타임 선언"이라고 하며, `defineProps()` 에 전달된 인자는 런타임의 `props` 옵션으로 사용됩니다.

하지만, 일반적으로 제네릭 타입 인자를 통해 순수 타입으로 props를 정의하는 것이 더 직관적입니다:

```vue
<script setup lang="ts">
const props = defineProps<{
  foo: string
```

```
})()
</script>
```

이것을 "타입 기반 선언"이라고 합니다. 컴파일러는 타입 인자를 기반으로 동등한 런타임 옵션을 최대한 추론하려고 시도합니다. 이 경우, 두 번째 예시는 첫 번째 예시와 정확히 동일한 런타임 옵션으로 컴파일됩니다.

타입 기반 선언 또는 런타임 선언 중 하나만 사용할 수 있으며, 동시에 둘 다 사용할 수는 없습니다.

props 타입을 별도의 인터페이스로 분리할 수도 있습니다:

```vue
<script setup lang="ts">
interface Props {
  foo: string
  bar?: number
}

const props = defineProps<Props>()
</script>
```

`Props` 가 외부 소스에서 import된 경우에도 동작합니다. 이 기능은 TypeScript가 Vue의 peer dependency로 필요합니다.

```vue
<script setup lang="ts">
import type { Props } from './foo'

const props = defineProps<Props>()
</script>
```

문법 제한 사항

3.2 버전 이하에서는 `defineProps()` 의 제네릭 타입 파라미터가 타입 리터럴 또는 로컬 인터페이스 참조로 제한되었습니다.

이 제한은 3.3에서 해결되었습니다. 최신 버전의 Vue는 타입 파라미터 위치에서 import된 타입과 제한된 복합 타입을 참조할 수 있습니다. 하지만 타입에서 런타임으로의 변환이 여전히 AST 기반이기 때문에, 조건부 타입 등 실제 타입 분석이 필요한 일부 복합 타입은 지원되지 않습니다. 조건부 타입은 단일 prop의 타입으로는 사용할 수 있지만, 전체 props 객체에는 사용할 수 없습니다.

Props 기본값

Props Destructure 3.5+ 를 사용하여 해결할 수 있습니다:

```ts
interface Props {
  msg?: string
  labels?: string[]
}

const { msg = 'hello', labels = ['one', 'two'] } = defineProps<Props>()
```

3.4 이하 버전에서는 Reactive Props Destructure가 기본적으로 활성화되어 있지 않습니다. 대안으로 `withDefaults` 컴파일러 매크로를 사용할 수 있습니다:

```ts
interface Props {
  msg?: string
  labels?: string[]
}

const props = withDefaults(defineProps<Props>(), {
  msg: 'hello',
  labels: () => ['one', 'two']
})
```

이 코드는 동등한 런타임 props `default` 옵션으로 컴파일됩니다. 추가로, `withDefaults` 헬퍼는 기본값에 대한 타입 검사를 제공하며, 기본값이 선언된 속성에 대해 반환된 `props` 타입에서 선택적 플래그를 제거합니다.

> ⓘ INFO
>
> 배열이나 객체와 같은 변경 가능한 참조 타입의 기본값은 `withDefaults` 를 사용할 때 함수로 감싸야 하며, 이는 실수로 인한 수정 및 외부 부작용을 방지합니다. 이렇게 하면 각 컴포넌트 인스턴스가 기본값의 자체 복사본을 갖게 됩니다. 구조 분해 할당을 사용할 때는 이 작업이 **필요하지 않습니다**.

`<script setup>` 없이

`<script setup>` 을 사용하지 않는 경우, props 타입 추론을 활성화하려면 `defineComponent()` 를 사용해야 합니다. `setup()` 에 전달되는 props 객체의 타입은 `props` 옵션에서 추론됩니다.

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

export default defineComponent({
  props: {
    message: String
  },
  setup(props) {
```

```
  }
})
```

복합 prop 타입

타입 기반 선언을 사용하면, prop에 복합 타입을 다른 타입과 마찬가지로 사용할 수 있습니다:

```vue
<script setup lang="ts">
interface Book {
  title: string
  author: string
  year: number
}

const props = defineProps<{
  book: Book
}>()
</script>
```

런타임 선언의 경우, `PropType` 유틸리티 타입을 사용할 수 있습니다:

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

const props = defineProps({
  book: Object as PropType<Book>
})
```

`props` 옵션을 직접 지정할 때도 거의 동일하게 동작합니다:

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

export default defineComponent({
  props: {
    book: Object as PropType<Book>
  }
})
```

`props` 옵션은 Options API에서 더 자주 사용되므로, Options API와 TypeScript 가이드에서 더 자세한 예시를 확인할 수 있습니다. 해당 예시에서 보여주는 기법은 `defineProps()` 를 사용하는 런타임 선언에도 적용됩니다.

컴포넌트 Emits 타입 지정하기

`<script setup>` 에서, `emit` 함수도 런타임 선언 또는 타입 선언을 통해 타입을 지정할 수 있습니다:

vue

```vue
<script setup lang="ts">
// 런타임
const emit = defineEmits(['change', 'update'])

// 옵션 기반
const emit = defineEmits({
  change: (id: number) => {
    // `true` 또는 `false`를 반환하여
    // 유효성 검사 통과/실패를 나타냄
  },
  update: (value: string) => {
    // `true` 또는 `false`를 반환하여
    // 유효성 검사 통과/실패를 나타냄
  }
})

// 타입 기반
const emit = defineEmits<{
  (e: 'change', id: number): void
  (e: 'update', value: string): void
}>()

// 3.3+: 대안, 더 간결한 문법
const emit = defineEmits<{
  change: [id: number]
  update: [value: string]
}>()
</script>
```

타입 인자는 다음 중 하나가 될 수 있습니다:

1. 호출 가능한 함수 타입이지만, Call Signatures로 작성된 타입 리터럴. 반환된 `emit` 함수의 타입으로 사용됩니다.
2. 이벤트 이름이 키이고, 값이 해당 이벤트에 대해 허용되는 추가 파라미터를 나타내는 배열/튜플 타입인 타입 리터럴. 위 예시는 각 인자가 명시적인 이름을 가질 수 있도록 명명된 튜플을 사용하고 있습니다.

타입 선언을 사용하면, emit되는 이벤트의 타입 제약을 훨씬 더 세밀하게 제어할 수 있습니다.

`<script setup>` 을 사용하지 않는 경우, `defineComponent()` 는 setup 컨텍스트에 노출된 `emit` 함수에 대해 허용된 이벤트를 추론할 수 있습니다:

```ts
export default defineComponent({
  emits: ['change'],
  setup(props, { emit }) {
    emit('change') // <-- 타입 검사 / 자동 완성
  }
})
```

`ref()` 타입 지정하기

ref는 초기값에서 타입을 추론합니다:

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

// 추론된 타입: Ref<number>
const year = ref(2020)

// => TS 오류: Type 'string'은(는) type 'number'에 할당할 수 없습니다.
year.value = '2020'
```

때로는 ref의 내부 값에 대해 복합 타입을 지정해야 할 수도 있습니다. 이럴 때는 `Ref` 타입을 사용할 수 있습니다:

```ts
import { ref } from 'vue'
import type { Ref } from 'vue'

const year: Ref<string | number> = ref('2020')

year.value = 2020 // ok!
```

또는, `ref()` 호출 시 제네릭 인자를 전달하여 기본 추론을 덮어쓸 수 있습니다:

```ts
// 결과 타입: Ref<string | number>
const year = ref<string | number>('2020')

year.value = 2020 // ok!
```

제네릭 타입 인자를 지정하고 초기값을 생략하면, 결과 타입은 `undefined` 를 포함하는 유니언 타입이 됩니다:

```ts
const n = ref<number>()
```

`reactive()` 타입 지정하기

`reactive()` 도 인자로부터 타입을 암시적으로 추론합니다:

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

// 추론된 타입: { title: string }
const book = reactive({ title: 'Vue 3 Guide' })
```

`reactive` 속성에 명시적으로 타입을 지정하려면, 인터페이스를 사용할 수 있습니다:

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

interface Book {
  title: string
  year?: number
}

const book: Book = reactive({ title: 'Vue 3 Guide' })
```

> **TIP**
>
> `reactive()` 의 제네릭 인자 사용은 권장되지 않습니다. 반환 타입(중첩 ref 언래핑 처리)이 제네릭 인자 타입과 다르기 때문입니다.

`computed()` 타입 지정하기

`computed()` 는 getter의 반환값을 기반으로 타입을 추론합니다:

```ts
import { ref, computed } from 'vue'

const count = ref(0)

// 추론된 타입: ComputedRef<number>
const double = computed(() => count.value * 2)
```

```
// => TS 오류: Property 'split' does not exist on type 'number'
const result = double.value.split('')
```

제네릭 인자를 통해 명시적으로 타입을 지정할 수도 있습니다:

```ts
const double = computed<number>(() => {
  // number를 반환하지 않으면 타입 오류 발생
})
```

이벤트 핸들러 타입 지정하기

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

```vue
<script setup lang="ts">
function handleChange(event) {
  // `event`는 암시적으로 `any` 타입을 가짐
  console.log(event.target.value)
}
</script>

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

타입 주석이 없으면, `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 인자에 명시적으로 타입을 지정하는 것이 권장됩니다. 또한, `event` 의 속성에 접근할 때 타입 단언을 사용해야 할 수도 있습니다:

```ts
function handleChange(event: Event) {
  console.log((event.target as HTMLInputElement).value)
}
```

Provide / Inject 타입 지정하기

지정하기 위해, Vue는 `InjectionKey` 인터페이스를 제공합니다. 이는 `Symbol` 을 확장한 제네릭 타입으로, 제공자와 소비자 간에 주입 값의 타입을 동기화하는 데 사용할 수 있습니다:

```ts
import { provide, inject } from 'vue'
import type { InjectionKey } from 'vue'

const key = Symbol() as InjectionKey<string>

provide(key, 'foo') // 문자열이 아닌 값을 제공하면 오류 발생

const foo = inject(key) // foo의 타입: string | undefined
```

주입 키는 별도의 파일에 두어 여러 컴포넌트에서 import할 수 있도록 하는 것이 좋습니다.

문자열 주입 키를 사용할 때는, 주입된 값의 타입이 `unknown` 이 되므로 제네릭 타입 인자를 통해 명시적으로 선언해야 합니다:

```ts
const foo = inject<string>('foo') // 타입: string | undefined
```

주입된 값은 여전히 `undefined` 일 수 있습니다. 이는 런타임에 제공자가 이 값을 제공한다는 보장이 없기 때문입니다.

기본값을 제공하면 `undefined` 타입을 제거할 수 있습니다:

```ts
const foo = inject<string>('foo', 'bar') // 타입: string
```

값이 항상 제공된다고 확신한다면, 값을 강제로 캐스팅할 수도 있습니다:

```ts
const foo = inject('foo') as string
```

템플릿 ref 타입 지정하기

Vue 3.5와 `@vue/language-tools` 2.1(IDE 언어 서비스와 `vue-tsc` 모두 지원)에서는 SFC에서 `useTemplateRef()` 로 생성된 ref의 타입이, 해당 `ref` 속성이 사용된 요소를 기반으로 **자동 추론**될 수 있습니다.

자동 추론이 불가능한 경우, 여전히 제네릭 인자를 통해 템플릿 ref를 명시적으로 타입 캐스팅할 수 있습니다:

▶ 3.5 이전 사용법

적절한 DOM 인터페이스를 얻으려면 MDN과 같은 페이지를 참고할 수 있습니다.

엄격한 타입 안전성을 위해서는 `el.value` 에 접근할 때 옵셔널 체이닝 또는 타입 가드를 사용하는 것이 필요합니다. 이는 초기 ref 값이 컴포넌트가 마운트될 때까지 `null` 이며, `v-if` 로 참조된 요소가 언마운트될 경우에도 `null` 이 될 수 있기 때문입니다.

컴포넌트 템플릿 ref 타입 지정하기

Vue 3.5와 `@vue/language-tools` 2.1(IDE 언어 서비스와 `vue-tsc` 모두 지원)에서는 SFC에서 `useTemplateRef()` 로 생성된 ref의 타입이, 해당 `ref` 속성이 사용된 요소나 컴포넌트를 기반으로 **자동 추론**될 수 있습니다.

자동 추론이 불가능한 경우(예: SFC가 아닌 사용, 동적 컴포넌트 등)에는 여전히 제네릭 인자를 통해 템플릿 ref를 명시적으로 타입 캐스팅할 수 있습니다.

import된 컴포넌트의 인스턴스 타입을 얻으려면, 먼저 `typeof` 로 타입을 얻은 후 TypeScript의 내장 `InstanceType` 유틸리티를 사용해 인스턴스 타입을 추출해야 합니다:

App.vue

vue

```vue
<script setup lang="ts">
import { useTemplateRef } from 'vue'
import Foo from './Foo.vue'
import Bar from './Bar.vue'

type FooType = InstanceType<typeof Foo>
type BarType = InstanceType<typeof Bar>

const compRef = useTemplateRef<FooType | BarType>('comp')
</script>

<template>
  <component :is="Math.random() > 0.5 ? Foo : Bar" ref="comp" />
</template>
```

컴포넌트의 정확한 타입이 없거나 중요하지 않은 경우, `ComponentPublicInstance` 를 대신 사용할 수 있습니다. 이 타입은 `$el` 과 같이 모든 컴포넌트가 공유하는 속성만 포함합니다:

```
import type { ComponentPublicInstance } from 'vue'

const child = useTemplateRef<ComponentPublicInstance>('child')
```

참조된 컴포넌트가 제네릭 컴포넌트인 경우, 예를 들어 `MyGenericModal` :

MyGenericModal.vue

```vue
<script setup lang="ts" generic="ContentType extends string | number">
import { ref } from 'vue'

const content = ref<ContentType | null>(null)

const open = (newContent: ContentType) => (content.value = newContent)

defineExpose({
  open
})
</script>
```

이 경우 `InstanceType` 이 동작하지 않으므로, `vue-component-type-helpers` 라이브러리의 `ComponentExposed` 를 사용해 참조해야 합니다.

App.vue

```vue
<script setup lang="ts">
import { useTemplateRef } from 'vue'
import MyGenericModal from './MyGenericModal.vue'
import type { ComponentExposed } from 'vue-component-type-helpers'

const modal = useTemplateRef<ComponentExposed<typeof MyGenericModal>>('modal')

const openModal = () => {
  modal.value?.open('newValue')
}
</script>
```

`@vue/language-tools` 2.1+에서는 정적 템플릿 ref의 타입이 자동으로 추론될 수 있으므로, 위와 같은 처리는 특수한 경우에만 필요합니다.

GitHub에서 이 페이지 편집

Previous
개요

Next
Options API와 TS