TypeScript와 Composition API

이 페이지는 이미 **Using Vue with TypeScript** 개요를 읽은 것을 가정합니다.

컴포넌트 Props의 타이핑

<script setup> 사용하기

<script setup> 을 사용할 때, defineProps() 매크로는 인수를 기반으로 Props 타입을 추론하는 기능을 지원합니다.

vue

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

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

이를 "런타임 선언"이라고 부르며, defineProps() 에 전달되는 인수가 런타임 props 옵션으로 사용됩니다.

하지만 보통은 제네릭 타입 인수를 사용하여 순수 타입으로 Props를 정의하는 것이 더 직관적입니다:

vue

```
<script setup lang="ts">
const props = defineProps<{
  foo: string
  bar?: number
}>()
</script>
```

이를 "타입 기반 선언"이라고 부릅니다. 컴파일러는 타입 인수를 기반으로 동일한 런타임 옵션을 추론하기 위해 최선을 다할 것입니다. 이 경우, 두 번째 예제는 첫 번째 예제와 정확히 동일한 런타임 옵션으로 컴파일됩니다.

타입 기반 선언이나 런타임 선언을 모두 사용할 수 있지만, 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>
```

이 기능은 Props 가 외부 소스에서 가져온 경우에도 작동합니다. 이 기능을 사용하려면 TypeScript가 Vue의 피어 종속성이어야 합니다.

vue

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

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

문법 제한

3.2 버전 이하에서는 `defineProps()` 의 제네릭 타입 매개변수는 타입 리터럴이나 로컬 인터페이스에 대한 참조로 제한되었습니다.

이 제한은 3.3에서 해결되었습니다. 최신 버전의 Vue는 가져온(imported) 및 일부 복잡한 타입을 유형 매개변수 위치에서 지원합니다. 그러나 유형을 런타임으로 변환하는 것은 여전히 AST 기반이므로 실제 유형 분석이 필요한 일부 복잡한 유형(조건부 유형 등)은 지원되지 않습니다. 조건부 유형은 단일 prop의 유형으로 사용할 수는 있지만, 전체 props 객체에는 사용할 수 없습니다.

Props 기본값

타입 기반 선언을 사용할 때, Props에 대한 기본값을 선언할 수 없습니다. 이는 `withDefaults` 컴파일러 매크로를 사용하여 해결할 수 있습니다:

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

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

이것은 동등한 런타임 props `default` 옵션으로 컴파일됩니다. 또한, `withDefaults` 헬퍼는 기본값에 대한 타입 체크를 제공하며, 기본값이 선언된 속성에 대한 선택적 플래그가 제거된 반환된 `props` 타입을 보장합니다.

`<script setup>` 을 사용하지 않는 경우

`<script setup>` 을 사용하지 않는 경우에는 `defineComponent()` 를 사용하여 Props 타입 추론을 가능하게 해야 합니다. `setup()` 에 전달되는 props 객체의 타입은 `props` 옵션을 기반으로 추론됩니다.

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

export default defineComponent({
  props: {
    message: String
  },
  setup(props) {
    props.message // <-- 타입: string
  }
})
```

복잡한 속성 유형

타입 기반 선언을 사용하면 prop도 다른 타입과 마찬가지로 복잡한 유형을 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 지정하는 경우에도 동일한 방식으로 작동합니다:

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

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

`props` 옵션은 주로 Options API와 함께 사용되므로, 옵션 **API**와 **TypeScript** 가이드에서 더 자세한 예제를 찾을 수 있습니다. 그 예제들에서 보여지는 기술은 `defineProps()` 를 사용한 런타임 선언에도 적용됩니다.

컴포넌트 이벤트의 타이핑

`<script setup>` 에서 `emit` 함수도 런타임 선언이나 타입 선언을 통해 타이핑할 수 있습니다:

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

// options 기반
const emit = defineEmits({
  change: (id: number) => {
    // `true` 또는 `false` 값을 반환하여
    // 유효성 검사 통과/실패 여부를 알려줌
  },
  update: (value: string) => {
    // `true` 또는 `false` 값을 반환하여
    // 유효성 검사 통과/실패 여부를 알려줌
  }
})

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

// 3.3+: 대체, 더 간결한 문법
const emit = defineEmits<{
  change: [id: number]
  update: [value: string]
```

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

타입 인수는 다음 중 하나일 수 있습니다:

호출 가능한 함수 타입입니다. **Call Signatures**와 함께 타입 리터럴로 작성됩니다. 이것은 반환된 `emit` 함수의 타입으로 사용됩니다.
키가 이벤트 이름이고 값이 추가 허용되는 이벤트의 매개변수를 나타내는 배열 또는 튜플 타입인 타입 리터럴입니다. 위의 예제는 명명된 튜플을 사용하므로 각 인수에 명시적인 이름을 부여할 수 있습니다.

보다 세부적인 이벤트의 타입 제약 조건을 지정할 수 있는 타입 선언을 사용하면 더욱 정교한 제어가 가능합니다.

`<script setup>` 을 사용하지 않는 경우, `defineComponent()` 을 사용하여 `setup()` 컨텍스트에서 노출된 `emit` 함수에 대한 허용된 이벤트를 컴파일러가 추론할 수 있습니다:

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

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

ref() 의 타이핑

`ref` 는 초기 값에서 타입을 추론합니다:

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

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

// => TS 에러: 'string' 타입은 'number'에 할당할 수 없습니다.
year.value = '2020'
```

때로는 ref의 내부 값에 대해 복잡한 유형을 지정해야 할 수도 있습니다. 이를 위해 `Ref` 타입을 사용할 수 있습니다:

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

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

year.value = 2020 // 정상적인 동작!
```

또는 `ref()` 를 호출할 때 제네릭 인수를 전달하여 기본 추론을 덮어쓸 수도 있습니다:

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

year.value = 2020 // 정상적인 동작!
```

제네릭 유형 인수를 지정하면서 초기 값은 생략하는 경우, 결과 타입은 `undefined` 를 포함하는 유니온 타입이 됩니다:

```ts
// 추론된 타입: Ref<number | undefined>
const n = ref<number>()
```

reactive() 의 타이핑

reactive() 는 인수에서 타입을 암묵적으로 추론합니다:

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

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

reactive 속성에 명시적인 타입을 지정하기 위해 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 인수를 사용하는 것은 권장되지 않습니다. 중첩된 ref 언래핑을 처리하는 반환된 타입은 제네릭 인수 타입과 다릅니다.

computed() 의 타이핑

computed() 는 getter의 반환 값에 따라 타입을 추론합니다:

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

const count = ref(0)

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

// => TS 에러: 'number'에는 'split' 속성이 존재하지 않습니다.
const result = double.value.split('')
```

제네릭 인수를 사용하여 명시적인 타입을 지정할 수도 있습니다:

```ts
const double = computed<number>(() => {
  // 반환 값이 number가 아니면 타입 에러
})
```

이벤트 핸들러의 타이핑

네이티브 DOM 이벤트를 다룰 때, 핸들러에 전달되는 인수의 타입을 올바르게 지정하는 것이 유용할 수 있습니다. 다음 예제를 살펴보세요:

```vue
<script setup lang="ts">
function handleChange(event) {
  // `event`의 타입은 암묵적으로 `any`입니다
  console.log(event.target.value)
}
</script>

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

타입 주석을 사용하지 않으면 `event` 인수는 암묵적으로 `any` 타입이 됩니다. 이는 `tsconfig.json` 에서 `"strict": true` 또는 `"noImplicitAny": true` 를 사용하는 경우 TS 에러로 이어집니다. 따라서 이벤트 핸들러의 인수를 명시적으로 주석으로 지정하는 것이 권장됩니다. 또한, `event` 의 속성에 접근할 때 타입 어설션(type assertion)을 사용해야 할 수도 있습니다:

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

provide / inject 의 타이핑

`provide` 와 `inject` 는 보통 서로 다른 컴포넌트에서 수행됩니다. 주입된(inject) 값의 타입을 적절히 지정하기 위해 Vue 는 `InjectionKey` 인터페이스를 제공하는데, 이는 `Symbol` 을 확장하는 제네릭 타입입니다. 이를 사용하여 제공자와 소비자 간에 주입된 값의 타입을 동기화할 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 키를 별도의 파일에 위치시켜 여러 컴포넌트에서 임포트할 수 있도록 하는 것이 좋습니다.

문자열 주입 키를 사용할 때, 주입된 값의 타입은 `unknown` 이 될 것이며, 제네릭 타입 인자를 통해 명시적으로 선언해야 합니다:

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

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

`undefined` 타입은 기본값을 제공함으로써 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 Ref의 타이핑

템플릿 ref는 명시적인 제네릭 타입 인수와 `null` 을 초기 값으로 사용하여 생성해야 합니다:

```vue
<script setup lang="ts">
import { ref, onMounted } from 'vue'

const el = ref<HTMLInputElement | null>(null)

onMounted(() => {
  el.value?.focus()
})
</script>

<template>
  <input ref="el" />
</template>
```

올바른 DOM 인터페이스를 얻기 위해서는 **MDN**과 같은 페이지를 확인할 수 있습니다.

Strict한 타입 안전성을 위해서는 `el.value` 에 접근할 때 옵셔널 체이닝 또는 타입 가드를 사용해야 합니다. 이는 초기 ref 값이 컴포넌트가 마운트될 때까지 `null` 이며, 참조된 요소가 `v-if` 에 의해 마운트 해제될 수도 있기 때문입니다.

컴포넌트 템플릿 Ref의 타이핑

자식 컴포넌트에 대한 템플릿 ref를 주석으로 지정하여 공개 메서드를 호출하기 위한 템플릿 ref를 타입으로 지정할 수도 있습니다. 예를 들어, `MyModal` 자식 컴포넌트가 모달을 열기 위한 메서드를 가지고 있다고 가정해보겠습니다:

```vue
<!-- MyModal.vue -->
<script setup lang="ts">
import { ref } from 'vue'

const isContentShown = ref(false)
const open = () => (isContentShown.value = true)

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

`MyModal` 의 인스턴스 타입을 가져오기 위해 `typeof` 를 통해 해당 컴포넌트의 유형을 먼저 얻은 다음, TypeScript의 내장 `InstanceType` 유틸리티를 사용하여 해당 인스턴스 타입을 추출해야 합니다:

```vue
<!-- App.vue -->
<script setup lang="ts">
import MyModal from './MyModal.vue'

const modal = ref<InstanceType<typeof MyModal> | null>(null)

const openModal = () => {
  modal.value?.open()
```

```
}
</script>
```

컴포넌트의 정확한 유형이 사용 불가능하거나 중요하지 않은 경우에는 `ComponentPublicInstance` 를 사용할 수도 있습니다. 이는 `$el` 과 같은 모든 컴포넌트에서 공유하는 속성만 포함합니다:

ts

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

const child = ref<ComponentPublicInstance | null>(null)
```