반응성 기본

> **API Preference**
>
> 이 페이지와 이후 가이드의 많은 챕터들은 Options API와 Composition API에 따라 다른 내용을 포함하고 있습니다. 현재 선택된 선호도는 **Composition API**입니다. 왼쪽 사이드바 상단의 "API Preference" 스위치를 사용해 API 스타일을 전환할 수 있습니다.

반응형 상태 선언하기

`ref()`

Composition API에서는 `ref()` 함수를 사용해 반응형 상태를 선언하는 것이 권장됩니다:

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

const count = ref(0)
```

`ref()` 는 인자를 받아 `.value` 속성이 있는 ref 객체로 감싸 반환합니다:

```js
const count = ref(0)

console.log(count) // { value: 0 }
console.log(count.value) // 0

count.value++
console.log(count.value) // 1
```

> 참고: Ref 타입 지정 TS

컴포넌트의 템플릿에서 ref에 접근하려면, 컴포넌트의 `setup()` 함수에서 선언하고 반환해야 합니다:

```
export default {
  // `setup`은 Composition API를 위한 특별한 훅입니다.
  setup() {
    const count = ref(0)

    // ref를 템플릿에 노출
    return {
      count
    }
  }
}
```

template

```
<div>{{ count }}</div>
```

ref를 템플릿에서 사용할 때는 `.value` 를 붙일 필요가 **없다는 것**에 주목하세요. 편의를 위해, ref는 템플릿 내부에서 자동으로 언래핑됩니다(몇 가지 주의사항이 있습니다).

이벤트 핸들러에서 ref를 직접 변경할 수도 있습니다:

template

```
<button @click="count++">
  {{ count }}
</button>
```

더 복잡한 로직의 경우, 같은 스코프에서 ref를 변경하는 함수를 선언하고 상태와 함께 메서드로 노출할 수 있습니다:

js

```
import { ref } from 'vue'

export default {
  setup() {
    const count = ref(0)

    function increment() {
      // JavaScript에서는 .value가 필요합니다
      count.value++
    }

    // 함수도 반드시 노출해야 합니다.
    return {
      count,
      increment
    }
  }
}
```

노출된 메서드는 이벤트 핸들러로 사용할 수 있습니다:

```
  {{ count }}
</button>
```

이 예제는 빌드 도구 없이 Codepen에서 직접 확인할 수 있습니다.

`<script setup>`

`setup()` 을 통해 상태와 메서드를 수동으로 노출하는 것은 다소 장황할 수 있습니다. 다행히 싱글 파일 컴포넌트(SFC)를 사용할 때는 이를 피할 수 있습니다. `<script setup>` 을 사용하면 더 간단하게 작성할 수 있습니다:

vue

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

const count = ref(0)

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

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

Playground에서 실행해보기

`<script setup>` 에서 선언된 최상위 import, 변수, 함수는 해당 컴포넌트의 템플릿에서 자동으로 사용할 수 있습니다. 템플릿을 같은 스코프에 선언된 JavaScript 함수라고 생각하면, 자연스럽게 함께 선언된 모든 것에 접근할 수 있습니다.

> **TIP**
>
> 이후 가이드에서는 Composition API 코드 예제에 SFC + `<script setup>` 문법을 주로 사용할 예정입니다. 이는 Vue 개발자들이 가장 많이 사용하는 방식입니다.
>
> SFC를 사용하지 않는 경우에도 `setup()` 옵션으로 Composition API를 사용할 수 있습니다.

왜 Ref를 사용할까요?

의 반응성 시스템이 어떻게 동작하는지 간단히 살펴보겠습니다.

템플릿에서 ref를 사용하고, 이후 ref의 값을 변경하면, Vue는 변경을 자동으로 감지하고 DOM을 업데이트합니다. 이는 의존성 추적 기반의 반응성 시스템 덕분입니다. 컴포넌트가 처음 렌더링될 때, Vue는 렌더링에 사용된 모든 ref를 **추적**합니다. 이후 ref가 변경되면, 이를 추적 중인 컴포넌트에 **재렌더링**을 트리거합니다.

일반 JavaScript에서는 단순 변수의 접근이나 변경을 감지할 방법이 없습니다. 하지만 객체의 속성에 대해서는 getter와 setter를 사용해 get/set 연산을 가로챌 수 있습니다.

`.value` 속성은 Vue가 ref에 접근하거나 변경될 때 이를 감지할 기회를 제공합니다. 내부적으로 Vue는 getter에서 추적을, setter에서 트리거를 수행합니다. 개념적으로 ref는 다음과 같은 객체라고 생각할 수 있습니다:

```js
// 의사 코드, 실제 구현이 아닙니다
const myRef = {
  _value: 0,
  get value() {
    track()
    return this._value
  },
  set value(newValue) {
    this._value = newValue
    trigger()
  }
}
```

ref의 또 다른 장점은, 단순 변수와 달리 ref를 함수에 전달해도 최신 값과 반응성 연결을 유지할 수 있다는 점입니다. 이는 복잡한 로직을 재사용 가능한 코드로 리팩토링할 때 특히 유용합니다.

반응성 시스템에 대한 자세한 내용은 반응성 심층 섹션에서 다룹니다.

깊은 반응성

Ref는 깊게 중첩된 객체, 배열, 또는 `Map` 과 같은 JavaScript 내장 데이터 구조 등 어떤 값도 담을 수 있습니다.

ref는 자신의 값을 깊게 반응형으로 만듭니다. 즉, 중첩된 객체나 배열을 변경해도 변경 사항이 감지됩니다:

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

const obj = ref({
  nested: { count: 0 },
  arr: ['foo', 'bar']
```

```
function mutateDeeply() {
  // 아래 코드도 정상적으로 동작합니다.
  obj.value.nested.count++
  obj.value.arr.push('baz')
}
```

비원시 값은 아래에서 설명할 `reactive()` 를 통해 반응형 프록시로 변환됩니다.

shallow ref를 사용해 깊은 반응성을 비활성화할 수도 있습니다. shallow ref에서는 `.value` 접근만 반응성 추적이 됩니다. shallow ref는 대용량 객체의 관찰 비용을 피하거나, 내부 상태가 외부 라이브러리에 의해 관리되는 경우 성능 최적화에 사용할 수 있습니다.

더 읽어보기:

대형 불변 구조체의 반응성 오버헤드 줄이기
외부 상태 시스템과의 통합

DOM 업데이트 타이밍

반응형 상태를 변경하면 DOM이 자동으로 업데이트됩니다. 하지만 DOM 업데이트는 동기적으로 적용되지 않는다는 점에 유의해야 합니다. Vue는 업데이트를 "다음 틱"까지 버퍼링하여, 상태 변경이 몇 번 일어나든 각 컴포넌트가 한 번만 업데이트되도록 보장합니다.

상태 변경 후 DOM 업데이트가 완료될 때까지 기다리려면 nextTick() 전역 API를 사용할 수 있습니다:

```
import { nextTick } from 'vue'

async function increment() {
  count.value++
  await nextTick()
  // 이제 DOM이 업데이트되었습니다
}
```

js

---

reactive()

반응형 상태를 선언하는 또 다른 방법은 `reactive()` API를 사용하는 것입니다. ref가 내부 값을 특별한 객체로 감싸는 것과 달리, `reactive()` 는 객체 자체를 반응형으로 만듭니다:

```
const state = reactive({ count: 0 })
```

> 참고: Reactive 타입 지정 TS

템플릿에서 사용 예시:

```template
<button @click="state.count++">
  {{ state.count }}
</button>
```

반응형 객체는 JavaScript Proxy이며, 일반 객체처럼 동작합니다. 차이점은 Vue가 반응성 추적 및 트리거를 위해 반응형 객체의 모든 속성 접근과 변경을 가로챌 수 있다는 점입니다.

`reactive()` 는 객체를 깊게 변환합니다: 중첩 객체도 접근 시 `reactive()` 로 감싸집니다. ref의 값이 객체일 때 내부적으로도 호출됩니다. shallow ref와 유사하게, 깊은 반응성을 비활성화할 수 있는 `shallowReactive()` API도 있습니다.

반응형 프록시 vs. 원본

`reactive()` 가 반환하는 값은 원본 객체의 Proxy이며, 원본 객체와 같지 않다는 점에 유의해야 합니다:

```js
const raw = {}
const proxy = reactive(raw)

// proxy는 원본과 같지 않습니다.
console.log(proxy === raw) // false
```

프록시만 반응형입니다 - 원본 객체를 변경해도 업데이트가 트리거되지 않습니다. 따라서 Vue의 반응성 시스템을 사용할 때는 **반드시 프록시 버전의 상태만 사용**하는 것이 모범 사례입니다.

프록시에 일관되게 접근할 수 있도록, 같은 객체에 대해 `reactive()` 를 여러 번 호출해도 항상 같은 프록시가 반환되며, 이미 프록시인 객체에 `reactive()` 를 호출해도 자기 자신을 반환합니다:

```js
// 같은 객체에 reactive()를 호출하면 같은 프록시를 반환
console.log(reactive(raw) === proxy) // true

// 프록시에 reactive()를 호출하면 자기 자신을 반환
console.log(reactive(proxy) === proxy) // true
```

시입니다:

```js
const proxy = reactive({})

const raw = {}
proxy.nested = raw

console.log(proxy.nested === raw) // false
```

`reactive()` 의 한계

`reactive()` API에는 몇 가지 한계가 있습니다:

1. **제한된 값 타입:** 객체 타입(객체, 배열, `Map`, `Set` 등 컬렉션 타입)에만 동작합니다. 원시 타입(`string`, `number`, `boolean` 등)은 사용할 수 없습니다.

2. **전체 객체 교체 불가:** Vue의 반응성 추적은 속성 접근을 기반으로 하므로, 항상 같은 반응형 객체 참조를 유지해야 합니다. 즉, 반응형 객체를 "교체"하면 첫 번째 참조와의 반응성 연결이 끊깁니다:

   ```js
   let state = reactive({ count: 0 })

   // 위 참조({ count: 0 })는 더 이상 추적되지 않습니다
   // (반응성 연결이 끊어집니다!)
   state = reactive({ count: 1 })
   ```

3. **구조 분해에 불리함:** 반응형 객체의 원시 타입 속성을 로컬 변수로 구조 분해하거나, 해당 속성을 함수에 전달하면 반응성 연결이 끊깁니다:

   ```js
   const state = reactive({ count: 0 })

   // 구조 분해 시 count는 state.count와 연결이 끊깁니다.
   let { count } = state
   // 원본 state에는 영향 없음
   count++

   // 함수에 평범한 숫자가 전달되어
   // state.count의 변경을 추적할 수 없습니다
   // 반응성을 유지하려면 전체 객체를 전달해야 합니다
   callSomeFunction(state.count)
   ```

이러한 한계로 인해, 반응형 상태를 선언할 때는 `ref()` 를 기본 API로 사용하는 것을 권장합니다.

추가 Ref 언래핑 세부사항

반응형 객체 속성으로서

ref는 반응형 객체의 속성으로 접근하거나 변경할 때 자동으로 언래핑됩니다. 즉, 일반 속성처럼 동작합니다:

```js
const count = ref(0)
const state = reactive({
  count
})

console.log(state.count) // 0

state.count = 1
console.log(count.value) // 1
```

기존 ref에 연결된 속성에 새 ref를 할당하면, 이전 ref가 대체됩니다:

```js
const otherCount = ref(2)

state.count = otherCount
console.log(state.count) // 2
// 기존 ref는 이제 state.count와 연결이 끊어집니다
console.log(count.value) // 1
```

ref 언래핑은 깊은 반응형 객체 내부에 중첩된 경우에만 발생합니다. shallow 반응형 객체의 속성으로 접근할 때는 적용되지 않습니다.

배열 및 컬렉션에서의 주의사항

반응형 객체와 달리, 반응형 배열이나 `Map` 과 같은 네이티브 컬렉션 타입의 요소로 ref에 접근할 때는 **언래핑이 일어나지 않습니다**:

```js
const books = reactive([ref('Vue 3 Guide')])
// 여기서는 .value가 필요합니다
console.log(books[0].value)

const map = reactive(new Map([['count', ref(0)]]))
// 여기서도 .value가 필요합니다
console.log(map.get('count').value)
```

템플릿에서 ref 언래핑은 ref가 템플릿 렌더 컨텍스트의 최상위 속성일 때만 적용됩니다.

아래 예제에서, `count` 와 `object` 는 최상위 속성이지만, `object.id` 는 그렇지 않습니다:

```js
const count = ref(0)
const object = { id: ref(1) }
```

따라서, 이 표현식은 기대한 대로 동작합니다:

```template
{{ count + 1 }}
```

...하지만 이 표현식은 **동작하지 않습니다**:

```template
{{ object.id + 1 }}
```

렌더링 결과는 `[object Object]1` 이 됩니다. 이는 `object.id` 가 표현식 평가 시 언래핑되지 않고 ref 객체로 남기 때문입니다. 이를 해결하려면, `id` 를 최상위 속성으로 구조 분해하면 됩니다:

```js
const { id } = object
```

```template
{{ id + 1 }}
```

이제 렌더 결과는 `2` 가 됩니다.

또 한 가지 주의할 점은, ref가 텍스트 보간(즉, `{{ }}` 태그)의 최종 평가 값일 경우에는 언래핑이 일어난다는 것입니다. 따라서 아래 코드는 `1` 을 렌더링합니다:

```template
{{ object.id }}
```

이는 텍스트 보간의 편의 기능일 뿐이며, `{{ object.id.value }}` 와 동일합니다.