반응형 기초

API 기본설정

이 페이지와 이후 다른 가이드의 많은 챕터에는 옵션과 컴포지션 API에 대한 다양한 컨텐츠가 포함되어 있습니다. 현재 기본 설정은 **컴포지션 API** 입니다. 좌측 사이드바 상단에 있는 "API 스타일 설정" 스위치를 사용하여 API 스타일을 전환할 수 있습니다.

반응형 상태 선언

ref()

Composition API에서 반응형 상태를 선언하는 권장 방법은 `ref()` 함수를 사용하는 것입니다:

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

const count = ref(0)
```

`ref()` 는 인수를 가져와서 `.value` 속성이 있는 ref 객체에 래핑하여 반환합니다:

```js
const count = ref(0)

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

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

참고: **Refs** 타입 지정하기

컴포넌트 템플릿의 ref에 액세스하려면, 컴포넌트의 `setup()` 함수에서 선언하고 반환합니다.

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

export default {
  // `setup`은 Composition API 전용 특수 후크입니다.
  setup() {
    const count = ref(0)

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

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

템플릿에서 ref를 사용할 때 `.value` 를 추가할 필요가 없었습니다. 편의상 ref는 템플릿 내에서 사용될 때 자동으로 언래핑됩니다(몇 가지 주의 사항).

이벤트 핸들러에서 직접 참조를 변경할 수도 있습니다:

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

보다 복잡한 논리를 위해 동일한 범위에서 ref를 변경하고 상태와 함께 메서드로 노출하는 함수를 선언할 수 있습니다:

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

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

    function increment() {
      // JavaScript 에서 .value 는 필요합니다.
      count.value++
    }

    // 함수를 노출하는 것도 잊지 마세요.
    return {
      count,
      increment
    }
  }
}
```

그런 다음 노출된 메서드를 이벤트 처리기로 사용할 수 있습니다:

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

다음은 빌드 도구를 사용하지 않은 **Codepen**에 있는 예시입니다.

<script setup>

setup() 을 통해 상태와 메서드를 수동으로 노출하는 것은 장황할 수 있습니다. 다행히 단일 파일 컴포넌트(**SFC**)를 사용하면 피할 수 있습니다. <script setup> 으로 사용법을 단순화할 수 있습니다:

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

const count = ref(0)

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

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

온라인 연습장으로 실행하기

<script setup> 에서 선언된 최상위 수준 가져오기, 변수 및 함수는 동일한 컴포넌트의 템플릿에서 자동으로 사용할 수 있습니다. 템플릿을 동일한 범위에서 선언된 JavaScript 함수로 생각하십시오. 자연스럽게 함께 선언된 모든 항목에 액세스할 수 있습니다.

TIP

가이드의 나머지 부분에서는 Vue 개발자가 가장 일반적으로 사용하는 Composition API 코드 예제에 주로 SFC + `<script setup>` 구문을 사용합니다.

SFC를 사용하지 않는 경우에도 `setup()` 옵션과 함께 Composition API를 사용할 수 있습니다.

왜 Refs 입니까?

왜 우리가 일반 변수 대신 `.value` 가 있는 ref를 필요로 하는지 궁금해하실 수 있습니다. 이를 설명하기 위해서는 Vue의 반응성 시스템이 어떻게 작동하는지 간단하게 논의해야 합니다.

템플릿에서 ref를 사용하고 나중에 ref의 값을 변경하면, Vue는 자동으로 이 변경을 감지하고 DOM을 적절하게 업데이트합니다. 이는 의존성 추적 기반의 반응형 시스템으로 가능합니다. 컴포넌트가 처음 렌더링될 때, Vue는 렌더링 과정에서 사용된 모든 ref를 **추적**합니다. 나중에 ref가 변경되면, 이를 추적하는 컴포넌트에 대해 재렌더링을 **트리거**합니다.

표준 JavaScript에서는 일반 변수의 접근이나 변형을 감지하는 방법이 없습니다. 하지만, getter와 setter 메서드를 사용하여 객체의 속성의 get 및 set 연산을 가로챌 수 있습니다.

`.value` 속성은 Vue에게 ref가 액세스되거나 변경되었을 때를 감지할 기회를 줍니다. 내부적으로, Vue는 getter에서 추적을 수행하고, setter에서 트리거를 수행합니다. 개념적으로, ref를 다음과 같은 객체라고 생각할 수 있습니다:

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

refs의 또 다른 좋은 특성은 일반 변수와 달리 최신 값과 반응성 연결에 대한 액세스를 유지하면서 refs를 함수에 전달할 수 있다는 것입니다. 이는 복잡한 논리를 재사용 가능한 코드로 리팩터링할 때 특히 유용합니다.

반응성 시스템은 깊은 반응성 섹션에서 자세히 설명합니다.

깊은 반응형

Refs는 깊게 중첩된 개체, 배열 또는 `Map` 과 같은 JavaScript 내장 데이터 구조를 포함하여 모든 값 유형을 보유할 수 있습니다.

ref는 값을 깊이 반응하게 만듭니다. 즉, 중첩된 객체나 배열을 변경하더라도 변경 사항이 감지될 것으로 예상할 수 있습니다:

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

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

function mutateDeeply() {
  // 예상대로 작동합니다
  obj.value.nested.count++
  obj.value.arr.push('baz')
}
```

기본이 아닌 값은 아래에서 설명하는 **reactive()** 를 통해 반응형 프록시로 전환됩니다.

shallow refs(얕은 참조)를 사용하여 깊은 반응성을 옵트아웃할 수도 있습니다. 얕은 참조의 경우 반응성을 위해 `.value` 액세스만 추적됩니다. 얕은 참조는 큰 개체의 관찰 비용을 피하거나 외부 라이브러리에서 내부 상태를 관리하는 경우 성능을 최적화하는 데 사용할 수 있습니다.

추가 정보:

- 큰 불변 구조체에 대한 반응성 오버헤드 줄이기
- 외부 상태 시스템과 통합

DOM 업데이트 타이밍

반응 상태를 변경하면 DOM이 자동으로 업데이트됩니다. 하지만 DOM 업데이트는 동기적으로 적용되지 않는다는 점에 유의해야 합니다. 대신 Vue는 업데이트 주기의 "다음 틱"까지 버퍼링하여 얼마나 많은 상태 변경을 수행하든 각 컴포넌트가 한 번만 업데이트되도록 합니다.

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

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

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

reactive()

반응 상태를 선언하는 또 다른 방법은 `reactive()` API를 사용하는 것입니다. 내부 값을 특수 객체로 감싸는 ref와 달리 `reactive()` 는 객체 자체를 반응형으로 만듭니다:

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

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

> 참고: **Reactive** 타입 지정하기

템플릿에서의 사용법:

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

반응형 개체는 **JavaScript Proxies**이며 일반 개체처럼 작동합니다. 차이점은 Vue가 반응성 추적 및 트리거링을 위해 반응성 객체의 모든 속성에 대한 액세스 및 변형을 가로챌 수 있다는 것입니다.

`reactive()` 는 객체를 심층적으로 변환합니다. 중첩된 객체도 액세스할 때 `reactive()` 로 래핑됩니다. ref 값이 객체일 때 내부적으로 `ref()` 에 의해 호출되기도 합니다. 얕은 참조와 유사하게 깊은 반응성을 옵트아웃하기 위한 **shallowReactive()** API도 있습니다.

반응형 재정의 vs. 원본

`reactive()` 의 반환 값은 원본 객체와 같지 않고 원본 객체를 재정의한 프록시(Proxy)라는 점을 유의하는 것이 중요합니다.

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

```
// 반응형으로 재정의 된 것은 원본과 같지 않습니다.
console.log(proxy === raw) // false
```

프록시만 반응형입니다. 원본 객체를 변경해도 업데이트가 트리거되지 않습니다. 따라서 객체를 Vue의 반응형 시스템으로 작업할 때 가장 좋은 방법은 **상태를 재정의한 프록시만 사용**하는 것입니다.

프록시에 대한 일관된 접근을 보장하기 위해, 원본 객체를 `reactive()` 한 프록시와 프록시를 `reactive()` 한 프록시는 동일한 프록시를 반환하도록 동작합니다.

```js
// 객체를 reactive() 한 반환 값과 프록시는 동일합니다.
console.log(reactive(raw) === proxy) // true

// 프록시를 reactive()한 반환 값과 프록시는 동일합니다.
console.log(reactive(proxy) === proxy) // true
```

이 규칙은 중첩된 객체에도 적용됩니다. 내부 깊숙이까지 반응형이므로 반응형 객체 내부의 중첩된 객체도 프록시입니다:

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

const raw = {}
proxy.nested = raw

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

`reactive()` 의 제한 사항

`reactive()` API에는 몇 가지 제한 사항이 있습니다:

제한된 값 유형: 객체 유형(객체, 배열 및 컬렉션 유형에만 작동합니다. (예: `Map` 및 `Set`). 그러나 `string` , `number` 또는 `boolean` 과 같은 기본 유형을 보유할 수 없습니다.

전체 객체를 대체할 수 없음: Vue의 반응성 추적은 속성 액세스를 통해 작동하므로 반응 객체에 대한 동일한 참조를 항상 유지해야 합니다. 즉, 첫 번째 참조에 대한 반응성 연결이 끊어지기 때문에 반응성 개체를 쉽게 "대체(replace)"할 수 없습니다:

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

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

분해 할당에 친화적이지 않음: 반응형 객체의 원시 타입 속성을 지역 변수로 분해하거나, 그 속성을 함수에 전달할 때, 반응성 연결이 끊어집니다:

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

// count는 분해 할당 될 때 state.count에서 연결이 끊어집니다.
let { count } = state
// 원래 상태에 영향을 주지 않음
count++

// 함수는 일반 숫자를 수신하고
// state.count에 대한 변경 사항을 추적할 수 없습니다.
// 반응성을 유지하려면 전체 개체를 전달해야 합니다.
callSomeFunction(state.count)
```

이러한 제한으로 인해 반응 상태를 선언하기 위한 기본 API로 `ref()` 를 사용하는 것이 좋습니다.

추가적인 Ref 언래핑 세부 사항

Reactive 객체 프로퍼티

ref는 반응 객체의 속성으로 액세스되거나 변경될 때 자동으로 래핑 해제됩니다. 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
```

ref가 할당된 기존 속성에 새 ref를 할당하면 이전 ref는 대체됩니다:

```js
const otherCount = ref(2)

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

ref의 언래핑은 깊은 반응형 객체 내부에 중첩된 경우에만 발생합니다. 얕은 반응형 객체의 속성으로 접근하는 경우에는 적용되지 않습니다.

배열 및 컬렉션의 주의 사항

반응형 객체와 달리 ref가 반응형 배열의 요소 또는 `Map` 과 같은 기본 컬렉션 유형으로 액세스될 때 랩핑 해제가 수행되지 않습니다:

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

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

템플릿에서 래핑 해제 시 주의 사항

템플릿에서 ref 언래핑은 ref가 템플릿 렌더링 컨텍스트의 최상위 속성인 경우에만 적용됩니다.

아래 예에서 `count` 및 `object` 는 최상위 속성이지만 `object.id` 는 그렇지 않습니다.:

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

따라서 이 표현식은 예상대로 작동합니다:

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

...하지만 아래는 아닙니다:

template

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

표현식을 평가할 때 `object.id` 가 언래핑되지 않고 ref 객체로 남아 있기 때문에 렌더링된 결과는 `[object Object]1` 이 됩니다. 이 문제를 해결하기 위해 `id` 를 최상위 속성으로 분해해야 합니다.

js

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

template

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

이제 렌더링 결과는 `2` 가 됩니다.

주목해야 할 또 다른 사항은 ref가 텍스트 보간(예: `{{ }}` 태그)의 최종 평가 값인 경우 래핑되지 않으므로 다음은 `1` 을 렌더링한다는 것입니다:

template

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

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