Watchers

기본 예제

계산 속성은 선언적으로 파생 값을 계산할 수 있게 해줍니다. 그러나 상태 변경에 반응하여 "부수 효과"를 수행해야 하는 경우가 있습니다. 예를 들어, DOM을 변경하거나 비동기 작업 결과에 따라 다른 상태를 변경하는 경우가 그렇습니다.

Composition API를 사용하면 반응형 상태가 변경될 때마다 콜백을 트리거하는 **watch** 함수를 사용할 수 있습니다:

vue

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

const question = ref('')
const answer = ref('질문에는 보통 물음표가 포함됩니다. ;-)')
const loading = ref(false)

// watch는 ref에 직접 작동합니다
watch(question, async (newQuestion, oldQuestion) => {
  if (newQuestion.includes('?')) {
    loading.value = true
    answer.value = '생각 중...'
    try {
      const res = await fetch('https://yesno.wtf/api')
      answer.value = (await res.json()).answer
    } catch (error) {
      answer.value = '오류! API에 도달할 수 없습니다. ' + error
    } finally {
      loading.value = false
    }
  }
})
</script>

<template>
  <p>
    예/아니오 질문을 하세요:
    <input v-model="question" :disabled="loading" />
  </p>
  <p>{{ answer }}</p>
</template>
```

온라인 연습장으로 실행하기

Watch 소스 타입

watch 의 첫 번째 인수는 다양한 유형의 반응형 "소스"가 될 수 있습니다: ref(계산된 ref 포함), 반응형 객체, **getter** 함수, 또는 여러 소스의 배열이 될 수 있습니다:

js

```
const x = ref(0)
const y = ref(0)

// 단일 ref
watch(x, (newX) => {
  console.log(`x는 ${newX}입니다`)
})

// getter
watch(
```

```
  () => x.value + y.value,
  (sum) => {
    console.log(`x와 y의 합은: ${sum}입니다`)
  }
)

// 여러 소스의 배열
watch([x, () => y.value], ([newX, newY]) => {
  console.log(`x는 ${newX}이고 y는 ${newY}입니다`)
})
```

반응형 객체의 속성을 다음과 같이 감시할 수 없습니다:

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

// 이것은 작동하지 않습니다. watch()에 숫자를 전달하고 있기 때문입니다.
watch(obj.count, (count) => {
  console.log(`Count는: ${count}입니다`)
})
```

대신, getter를 사용하십시오:

```js
// 대신, getter를 사용하십시오:
watch(
  () => obj.count,
  (count) => {
    console.log(`Count는: ${count}입니다`)
  }
)
```

깊은 감시자

반응형 객체에서 직접 `watch()` 를 호출하면 암시적으로 깊은 감시자가 생성됩니다 - 콜백은 모든 중첩된 변경에 대해 트리거됩니다:

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

watch(obj, (newValue, oldValue) => {
  // 중첩된 속성 변경 시 실행됩니다
  // 참고: `newValue`는 여기서 `oldValue`와 동일합니다.
  // 두 값 모두 동일한 객체를 가리키기 때문입니다!
})

obj.count++
```

반응형 객체를 반환하는 getter와는 차이가 있습니다 - 후자의 경우, getter가 다른 객체를 반환할 때만 콜백이 실행됩니다:

```js
watch(
  () => state.someObject,
  () => {
    // state.someObject가 교체될 때만 실행됩니다.
  }
)
```

그러나, `deep` 옵션을 명시적으로 사용하여 두 번째 경우를 깊은 감시자로 만들 수 있습니다:

```js
watch(
  () => state.someObject,
  (newValue, oldValue) => {
    // 참고: state.someObject가 교체되지 않는 한,
    // 여기서 `newValue`는 `oldValue`와 동일합니다.
  },
  { deep: true }
)
```

주의해서 사용

깊은 감시는 감시된 객체의 모든 중첩된 속성을 순회해야 하므로, 대규모 데이터 구조에서 사용할 경우 비용이 많이 들 수 있습니다. 필요할 때만 사용하고 성능에 미치는 영향을 주의하십시오.

Eager 감시자

`watch` 는 기본적으로 지연 모드입니다: 감시된 소스가 변경될 때까지 콜백이 호출되지 않습니다. 그러나 경우에 따라 동일한 콜백 로직을 즉시 실행하고 싶을 수 있습니다 - 예를 들어, 초기 데이터를 가져오고, 관련 상태가 변경될 때마다 데이터를 다시 가져오고 싶을 때가 그렇습니다.

`immediate: true` 옵션을 전달하여 감시자의 콜백을 즉시 실행하도록 강제할 수 있습니다:

```js
watch(
  source,
  (newValue, oldValue) => {
    // 즉시 실행된 후, `source`가 변경될 때 다시 실행됩니다.
  },
  { immediate: true }
)
```

Once 감시자

감시자의 콜백은 감시된 소스가 변경될 때마다 실행됩니다. 콜백이 소스가 변경될 때 한 번만 트리거되기를 원한다면, `once: true` 옵션을 사용하십시오.

```js
watch(
  source,
  (newValue, oldValue) => {
    // `source`가 변경될 때 한 번만 트리거됩니다.
  },
  { once: true }
)
```

watchEffect()

감시자 콜백이 소스와 동일한 반응형 상태를 사용하는 경우가 일반적입니다. 예를 들어, `todoId` `ref`가 변경될 때마다 원격 리소스를 로드하는 감시자를 고려해 보세요:

```js
const todoId = ref(1)
const data = ref(null)

watch(
  todoId,
  async () => {
    const response = await fetch(
      `https://jsonplaceholder.typicode.com/todos/${todoId.value}`
    )
    data.value = await response.json()
  },
  { immediate: true }
)
```

특히, 감시자가 `todoId` 를 두 번 사용하는 방법에 주목하세요, 한 번은 소스로, 그리고 다시 콜백 내에서 사용됩니다.

이것은 **`watchEffect()`** 를 사용하여 간소화할 수 있습니다. `watchEffect()` 는 콜백의 반응형 종속성을 자동으로 추적할 수 있게 해줍니다. 위의 감시자는 다음과 같이 다시 작성할 수 있습니다:

```js
watchEffect(async () => {
  const response = await fetch(
    `https://jsonplaceholder.typicode.com/todos/${todoId.value}`
  )
  data.value = await response.json()
})
```

여기서 콜백은 즉시 실행되며, `immediate: true` 를 지정할 필요가 없습니다. 실행 중에는 `todoId.value` 를 종속성으로 자동으로 추적합니다(계산 속성과 유사). `todoId.value` 가 변경될 때마다 콜백이 다시 실행됩니다. `watchEffect()` 를 사용하면 소스 값으로 `todoId` 를 명시적으로 전달할 필요가 없습니다.

`watchEffect()` 와 반응형 데이터 가져오기 작업의 실제 예시는 이 예제를 참고하세요.

이와 같은 예에서, 종속성이 하나만 있는 경우 `watchEffect()` 의 이점은 상대적으로 적습니다. 그러나 종속성이 여러 개인 감시자의 경우, `watchEffect()` 를 사용하면 종속성 목록을 수동으로 유지하는 부담을 덜 수 있습니다. 또한, 중첩된 데이터 구조에서 여러 속성을 감시해야 할 경우, `watchEffect()` 는 콜백에서 사용된 속성만 추적하므로 깊은 감시자보다 더 효율적일 수 있습니다.

> **TIP**
>
> `watchEffect` 는 **동기적** 실행 동안에만 종속성을 추적합니다. 비동기 콜백과 함께 사용할 때는 첫 번째 `await` 틱 전에 접근한 속성만 추적됩니다.

`watch` 와 `watchEffect` 비교

`watch` 와 `watchEffect` 는 모두 반응형으로 부수 효과를 수행할 수 있게 해줍니다. 주요 차이점은 반응형 종속성을 추적하는 방식입니다:

`watch` 는 명시적으로 감시된 소스만 추적합니다. 콜백 내부에서 접근한 것은 추적하지 않습니다. 또한, 소스가 실제로 변경되었을 때만 콜백이 실행됩니다. `watch` 는 종속성 추적을 부수 효과와 분리하여 콜백이 실행될 시기를 더 정확하게 제어할 수 있습니다.

`watchEffect` 는 종속성 추적과 부수 효과를 하나의 단계로 결합합니다. 동기적 실행 동안 접근한 모든 반응형 속성을 자동으로 추적합니다. 이는 더 편리하며 일반적으로 더 간결한 코드를 작성할 수 있게 해주지만, 반응형 종속성이 덜 명확합니다.

콜백 플러시 타이밍

반응형 상태를 변경하면 Vue 컴포넌트 업데이트와 사용자가 생성한 감시자 콜백을 모두 트리거할 수 있습니다.

컴포넌트 업데이트와 유사하게, 사용자가 생성한 감시자 콜백은 중복 호출을 피하기 위해 배치됩니다. 예를 들어, 감시되는 배열에 항목을 동기적으로 1,000개 푸시할 때 감시자가 1,000번 실행되는 것은 원하지 않을 것입니다.

기본적으로, 감시자의 콜백은 부모 컴포넌트 업데이트 후(있는 경우), 소유자 컴포넌트의 DOM 업데이트 전 호출됩니다. 이는 감시 자 콜백 내부에서 소유자 컴포넌트의 DOM에 접근하려고 할 때, DOM이 업데이트 전 상태일 것임을 의미합니다.

후처리 감시자

감시자 콜백에서 Vue가 DOM을 업데이트한 후 소유자 컴포넌트의 DOM에 접근하려면, `flush: 'post'` 옵션을 지정해야 합니다:

```js
watch(source, callback, {
  flush: 'post'
})

watchEffect(callback, {
  flush: 'post'
})
```

후처리 `watchEffect()` 에는 편리한 별칭인 `watchPostEffect()` 가 있습니다:

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

watchPostEffect(() => {
  /* Vue 업데이트 후 실행됨 */
})
```

동기 감시자

Vue 관리 업데이트 이전에 동기적으로 실행되는 감시자를 만들 수도 있습니다:

```js
watch(source, callback, {
  flush: 'sync'
})

watchEffect(callback, {
  flush: 'sync'
})
```

동기 `watchEffect()` 에는 편리한 별칭인 `watchSyncEffect()` 가 있습니다:

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

watchSyncEffect(() => {
  /* 반응형 데이터 변경 시 동기적으로 실행됨 */
})
```

> 주의해서 사용
>
> 동기 감시자는 배치가 없으며 반응형 변이가 감지될 때마다 트리거됩니다. 간단한 불리언 값을 감시할 때는 사용해도 괜찮지만, 배열과 같이 동기적으로 여러 번 변경될 수 있는 데이터 소스에 사용하지 마십시오.

감시자 중지

`setup()` 이나 `<script setup>` 내에서 동기적으로 선언된 감시자는 소유자 컴포넌트 인스턴스에 바인딩되며, 소유자 컴포넌트가 마운트 해제될 때 자동으로 중지됩니다. 대부분의 경우 감시자를 직접 중지할 필요가 없습니다.

여기서 중요한 점은 감시자가 **동기적으로** 생성되어야 한다는 것입니다: 감시자가 비동기 콜백 내에서 생성되면 소유자 컴포넌트에 바인딩되지 않으며, 메모리 누수를 피하기 위해 수동으로 중지해야 합니다. 다음은 예시입니다:

vue

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

// 이 감시자는 자동으로 중지됩니다.
watchEffect(() => {})

// ...이 감시자는 중지되지 않습니다!
setTimeout(() => {
  watchEffect(() => {})
}, 100)
</script>
```

감시자를 수동으로 중지하려면 반환된 핸들 함수를 사용하십시오. 이는 `watch` 와 `watchEffect` 모두에 대해 작동합니다:

js

```js
const unwatch = watchEffect(() => {})

// ...나중에, 더 이상 필요하지 않을 때
unwatch()
```

비동기적으로 감시자를 생성해야 하는 경우는 매우 드물며, 가능하면 동기적 생성을 선호해야 합니다. 비동기 데이터를 기다려야 하는 경우, 감시 로직을 조건부로 만들 수 있습니다:

js

```js
// 비동기적으로 로드될 데이터
const data = ref(null)

watchEffect(() => {
  if (data.value) {
    // 데이터가 로드될 때 무언가를 수행합니다.
  }
})
```