Watchers

기본 예제

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

Composition API에서는 `watch` 함수를 사용하여 반응형 상태가 변경될 때마다 콜백을 실행할 수 있습니다:

vue

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

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

// watch는 ref에 직접 사용할 수 있습니다
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 접근할 수 없습니다. ' + error
    } finally {
      loading.value = false
    }
  }
})
</script>

<template>
  <p>
    예/아니오로 대답할 수 있는 질문을 해보세요:
    <input v-model="question" :disabled="loading" />
```

```
  <p>{{ answer }}</p>
</template>
```

플레이그라운드에서 실행해보기

Watch 소스 타입

`watch` 의 첫 번째 인자는 다양한 타입의 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(`x + y의 합은: ${sum}`)
  }
)

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

반응형 객체의 속성을 아래와 같이 감시할 수는 없습니다:

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

// 이렇게 하면 동작하지 않습니다. watch()에 숫자를 전달하기 때문입니다.
watch(obj.count, (count) => {
  console.log(`Count는: ${count}`)
})
```

대신 getter를 사용하세요:

```js
// 대신 getter를 사용하세요:
watch(
  () => obj.count,
  (count) => {
```

```
  }
)
```

깊은 감시자(Deep Watchers)

`watch()` 를 반응형 객체에 직접 호출하면 암묵적으로 깊은 감시자가 생성됩니다 - 콜백은 모든 중첩된 변경에 대해 실행됩니다:

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

watch(obj, (newValue, oldValue) => {
  // 중첩된 속성 변경에도 실행됩니다
  // 참고: `newValue`와 `oldValue`는
  // 동일한 객체를 가리키므로 같습니다!
})

obj.count++
```

반응형 객체를 반환하는 getter와는 구분해야 합니다 - 이 경우에는 getter가 다른 객체를 반환할 때만 콜백이 실행됩니다:

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

하지만, 두 번째 경우에도 `deep` 옵션을 명시적으로 사용하여 깊은 감시자로 만들 수 있습니다:

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

Vue 3.5+에서는 `deep` 옵션에 최대 탐색 깊이를 나타내는 숫자를 지정할 수도 있습니다. 즉, Vue가 객체의 중첩 속성을 몇 단계까지 탐색할지 지정할 수 있습니다.

⚠ 주의해서 사용하세요

깊은 감시는 감시하는 객체의 모든 중첩 속성을 순회해야 하므로, 대용량 데이터 구조에 사용하면 비용이 많이 들 수 있습니다. 꼭 필요한 경우에만 사용하고, 성능에 주의하세요.

즉시 실행 감시자(Eager Watchers)

`watch` 는 기본적으로 지연(lazy) 실행됩니다: 감시하는 소스가 변경되기 전까지 콜백이 호출되지 않습니다. 하지만 경우에 따라 동일한 콜백 로직을 즉시 실행하고 싶을 수 있습니다. 예를 들어, 초기 데이터를 가져오고, 관련 상태가 변경될 때마다 다시 데이터를 가져오고 싶을 때가 있습니다.

`immediate: true` 옵션을 전달하여 감시자의 콜백을 즉시 실행할 수 있습니다:

```js
watch(
  source,
  (newValue, oldValue) => {
    // 즉시 실행되고, 이후 `source`가 변경될 때마다 다시 실행됨
  },
  { immediate: true }
)
```

1회성 감시자(Once Watchers)

3.4+에서만 지원

감시자의 콜백은 감시하는 소스가 변경될 때마다 실행됩니다. 만약 소스가 변경될 때 단 한 번만 콜백이 실행되길 원한다면, `once: true` 옵션을 사용하세요.

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

watchEffect()

감시자 콜백이 소스와 정확히 동일한 반응형 상태를 사용할 때가 많습니다. 예를 들어, 아래 코드는 `todoId` ref가 변경될 때마다 원격 리소스를 로드하기 위해 감시자를 사용합니다:

```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 소스로 한 번, 콜백 내부에서 한 번 더 사용하고 있다는 점에 주목하세요.

이 코드는 `watchEffect()` 로 더 간단하게 만들 수 있습니다. `watchEffect()` 는 콜백의 반응형 의존성을 자동으로 추적합니다. 위의 감시자는 다음과 같이 다시 쓸 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 명시할 필요가 없습니다. 실행 중에 `todoId.value` 가 의존성으로 자동 추적됩니다(계산 속성과 유사). `todoId.value` 가 변경될 때마다 콜백이 다시 실행됩니다. `watchEffect()` 를 사용하면 더 이상 소스 값을 명시적으로 전달할 필요가 없습니다.

`watchEffect()` 와 반응형 데이터 패칭이 실제로 동작하는 이 예제를 확인해보세요.

이처럼 의존성이 하나뿐인 경우에는 `watchEffect()` 의 이점이 크지 않습니다. 하지만 여러 의존성을 가진 감시자에서는 `watchEffect()` 를 사용하면 의존성 목록을 직접 관리할 필요가 없어집니다. 또한, 중첩 데이터 구조에서 여러 속성을 감시해야 할 때, `watchEffect()` 는 콜백에서 실제로 사용된 속성만 추적하므로, 모든 속성을 재귀적으로 추적하는 깊은 감시자보다 더 효율적일 수 있습니다.

TIP

`watchEffect` 는 **동기** 실행 중에만 의존성을 추적합니다. 비동기 콜백과 함께 사용할 때는, 첫 번째 `await` 이전에 접근한 속성만 추적됩니다.

`watch` vs. `watchEffect`

`watch` 와 `watchEffect` 모두 반응적으로 부수 효과를 수행할 수 있게 해줍니다. 두 함수의 주요 차이점은 반응형 의존성을 추적하는 방식에 있습니다:

`watch` 는 명시적으로 감시하는 소스만 추적합니다. 콜백 내부에서 접근한 값은 추적하지 않습니다. 또한, 소스가 실제로 변경될 때만 콜백이 실행됩니다. `watch` 는 의존성 추적과 부수 효과를 분리하여, 콜백이 언제 실행될지 더 정밀하게 제어할 수 있습니다.

반면, `watchEffect` 는 의존성 추적과 부수 효과를 하나의 단계로 결합합니다. 동기 실행 중에 접근한 모든 반응형 속성을 자동으로 추적합니다. 더 편리하고 코드가 간결해지지만, 반응형 의존성이 명시적이지 않게 됩니다.

부수 효과 정리(Side Effect Cleanup)

때때로 감시자에서 비동기 요청 등 부수 효과를 수행할 수 있습니다:

```js
watch(id, (newId) => {
  fetch(`/api/${newId}`).then(() => {
    // 콜백 로직
  })
})
```

하지만 요청이 완료되기 전에 `id` 가 변경된다면 어떻게 될까요? 이전 요청이 완료되면 이미 오래된 ID 값으로 콜백이 실행됩니다. 이상적으로는, `id` 가 새 값으로 변경될 때 이전 요청을 취소할 수 있으면 좋겠습니다.

`onWatcherCleanup()` 3.5+ API를 사용하면 감시자가 무효화되어 다시 실행되기 직전에 정리 함수를 등록할 수 있습니다:

```js
import { watch, onWatcherCleanup } from 'vue'

watch(id, (newId) => {
  const controller = new AbortController()

  fetch(`/api/${newId}`, { signal: controller.signal }).then(() => {
```

```
  })

  onWatcherCleanup(() => {
    // 오래된 요청 중단
    controller.abort()
  })
})
```

`onWatcherCleanup` 은 Vue 3.5+에서만 지원되며, 반드시 `watchEffect` 효과 함수나 `watch` 콜백 함수의 동기 실행 중에 호출해야 합니다. 비동기 함수에서 `await` 이후에 호출할 수 없습니다.

또는, 감시자 콜백의 3번째 인자, 그리고 `watchEffect` 효과 함수의 첫 번째 인자로 `onCleanup` 함수가 전달됩니다:

```js
watch(id, (newId, oldId, onCleanup) => {
  // ...
  onCleanup(() => {
    // 정리 로직
  })
})

watchEffect((onCleanup) => {
  // ...
  onCleanup(() => {
    // 정리 로직
  })
})
```

이 방식은 3.5 이전 버전에서도 동작합니다. 또한, 함수 인자로 전달된 `onCleanup` 은 감시자 인스턴스에 바인딩되어 있으므로, `onWatcherCleanup` 의 동기 실행 제약을 받지 않습니다.

콜백 실행 타이밍(Callback Flush Timing)

반응형 상태를 변경하면, Vue 컴포넌트 업데이트와 사용자가 만든 감시자 콜백이 모두 트리거될 수 있습니다.

컴포넌트 업데이트와 마찬가지로, 사용자가 만든 감시자 콜백도 중복 호출을 방지하기 위해 배치 처리됩니다. 예를 들어, 감시하는 배열에 동기적으로 1,000개 항목을 추가할 때 감시자가 1,000번 실행되는 것을 원하지 않을 것입니다.

기본적으로 감시자 콜백은 **상위 컴포넌트 업데이트 이후**(있다면), 그리고 **소유 컴포넌트의 DOM 업데이트 이전**에 호출됩니다. 즉, 감시자 콜백에서 소유 컴포넌트의 DOM에 접근하면, DOM이 업데이트되기 전 상태임을 의미합니다.

감시자 콜백에서 **Vue가 DOM을 업데이트한 후** 소유 컴포넌트의 DOM에 접근하고 싶다면, `flush: 'post'` 옵션을 지정해야 합니다:

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

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

후처리 `watchEffect()` 에는 편의상 `watchPostEffect()` 라는 별칭도 있습니다:

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

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

동기 감시자(Sync Watchers)

Vue가 관리하는 업데이트보다 먼저, 동기적으로 실행되는 감시자를 만들 수도 있습니다:

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

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

동기 `watchEffect()` 에는 편의상 `watchSyncEffect()` 라는 별칭도 있습니다:

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

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

⚠ 주의해서 사용하세요

없을 감시할 때는 편리하지만, 매일 등 동기적으로 어려운 변경될 수 있는 데이터 소스에는 사용을 피하세요.

감시자 중단하기(Stopping a Watcher)

`setup()` 또는 `<script setup>` 내부에서 동기적으로 선언한 감시자는 소유 컴포넌트 인스턴스에 바인딩되며, 소유 컴포넌트가 언마운트될 때 자동으로 중단됩니다. 대부분의 경우 감시자를 직접 중단할 필요가 없습니다.

여기서 중요한 점은 감시자가 **동기적으로** 생성되어야 한다는 것입니다: 감시자가 비동기 콜백에서 생성되면 소유 컴포넌트에 바인딩되지 않으므로, 메모리 누수를 방지하려면 직접 중단해야 합니다. 예시는 다음과 같습니다:

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

// 이 감시자는 자동으로 중단됩니다
watchEffect(() => {})

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

감시자를 수동으로 중단하려면 반환된 핸들 함수를 사용하세요. 이는 `watch` 와 `watchEffect` 모두에 적용됩니다:

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

// ...나중에 더 이상 필요 없을 때
unwatch()
```

비동기적으로 감시자를 생성해야 하는 경우는 매우 드물며, 가능하면 동기적으로 생성하는 것이 좋습니다. 비동기 데이터를 기다려야 한다면, 감시 로직을 조건부로 만들 수 있습니다:

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

watchEffect(() => {
  if (data.value) {
```

```
  }
})
```

GitHub에서 이 페이지 편집

Previous
폼 입력 바인딩

Next
템플릿 ref