반응성 심층 분석

Vue의 가장 두드러진 특징 중 하나는 눈에 띄지 않는 반응성 시스템입니다. 컴포넌트 상태는 반응형 JavaScript 객체로 구성됩니다. 이 객체를 수정하면 뷰가 업데이트됩니다. 이는 상태 관리를 간단하고 직관적으로 만들어주지만, 몇 가지 일반적인 함정들을 피하기 위해서도 그 동작 방식을 이해하는 것이 중요합니다. 이 섹션에서는 Vue의 반응성 시스템의 저수준 세부 사항을 파고들어 보겠습니다.

반응성이란?

이 용어는 요즘 프로그래밍에서 자주 등장하지만, 사람들이 이 말을 할 때 실제로 무엇을 의미할까요? 반응성은 선언적인 방식으로 변화에 적응할 수 있게 해주는 프로그래밍 패러다임입니다. 사람들이 보통 보여주는 대표적인 예시는, 훌륭한 예시이기도 한, 엑셀 스프레드시트입니다:

	A	B	C
0	1		
1	2		
2	3		

여기서 셀 A2는 `= A0 + A1` 이라는 수식으로 정의되어 있습니다(A2를 클릭하면 수식을 보고 수정할 수 있습니다). 그래서 스프레드시트는 3을 보여줍니다. 놀랄 일은 없습니다. 하지만 A0나 A1을 업데이트하면, A2도 자동으로 업데이트되는 것을 볼 수 있습니다.

JavaScript는 보통 이렇게 동작하지 않습니다. 만약 JavaScript로 비슷한 것을 작성한다면:

```js
let A0 = 1
let A1 = 2
let A2 = A0 + A1

console.log(A2) // 3

A0 = 2
console.log(A2) // 여전히 3
```

그렇다면 JavaScript에서는 어떻게 해야 할까요? 먼저, `A2` 를 업데이트하는 코드를 다시 실행할 수 있도록 함수로 감싸봅시다:

```js
let A2

function update() {
  A2 = A0 + A1
}
```

그리고 몇 가지 용어를 정의해야 합니다:

> `update()` 함수는 프로그램의 상태를 변경하므로 **부수 효과** 또는 줄여서 **이펙트**를 발생시킵니다.
>
> `A0` 와 `A1` 은 이 이펙트의 **의존성**입니다. 이 값들이 이펙트를 수행하는 데 사용되기 때문입니다. 이펙트는 자신의 의존성에 **구독자**가 되었다고 할 수 있습니다.

우리가 필요한 것은 `A0` 이나 `A1` (즉, **의존성**)이 변경될 때마다 `update()` (**이펙트**)를 호출해주는 마법 같은 함수입니다:

```js
whenDepsChange(update)
```

이 `whenDepsChange()` 함수는 다음과 같은 작업을 해야 합니다:

1. 변수가 읽힐 때를 추적합니다. 예를 들어, `A0 + A1` 표현식을 평가할 때 `A0` 와 `A1` 이 모두 읽힙니다.
2. 현재 실행 중인 이펙트가 있을 때 변수가 읽히면, 그 이펙트를 해당 변수의 구독자로 만듭니다. 예를 들어, `update()` 가 실행될 때 `A0` 와 `A1` 이 읽히므로, 첫 호출 이후 `update()` 는 `A0` 와 `A1` 모두의 구독자가 됩니다.
3. 변수가 변경될 때를 감지합니다. 예를 들어, `A0` 에 새 값이 할당되면, 모든 구독자 이펙트에 다시 실행하라고 알립니다.

Vue에서의 반응성 동작 방식

예제처럼 지역 변수의 읽기와 쓰기를 실제로 추적할 수는 없습니다. 순수 JavaScript에는 이를 위한 메커니즘이 없습니다. 하지만 **객체 속성**의 읽기와 쓰기를 가로챌 수는 있습니다.

다. Vue 2는 브라우저 지원 제한 때문에 getter / setter만 사용했습니다. Vue 3에서는 Proxy가 반응형 객체에 사용되고, getter / setter는 ref에 사용됩니다. 아래는 그 동작 방식을 보여주는 의사 코드입니다:

```js
function reactive(obj) {
  return new Proxy(obj, {
    get(target, key) {
      track(target, key)
      return target[key]
    },
    set(target, key, value) {
      target[key] = value
      trigger(target, key)
    }
  })
}

function ref(value) {
  const refObject = {
    get value() {
      track(refObject, 'value')
      return value
    },
    set value(newValue) {
      value = newValue
      trigger(refObject, 'value')
    }
  }
  return refObject
}
```

> ⓘ TIP
>
> 여기와 아래의 코드 스니펫은 핵심 개념을 최대한 단순하게 설명하기 위한 것이므로, 많은 세부 사항이 생략되어 있고, 예외적인 경우도 무시되어 있습니다.

이것은 우리가 기본 섹션에서 논의했던 반응형 객체의 몇 가지 제한 사항을 설명해줍니다:

반응형 객체의 속성을 지역 변수에 할당하거나 구조 분해 할당하면, 그 변수에 접근하거나 할당해도 더 이상 원본 객체의 get / set 프록시 트랩을 트리거하지 않으므로 반응형이 아닙니다. 이 "연결 해제"는 변수 바인딩에만 영향을 미치며, 만약 변수가 객체와 같은 비원시 값을 가리킨다면, 그 객체를 변경하는 것은 여전히 반응형입니다.

`reactive()` 에서 반환된 프록시는 원본과 거의 동일하게 동작하지만, `===` 연산자로 비교하면 원본과는 다른 정체성을 가집니다.

한 구독자 이펙트(집합에 저장됨)를 찾아서 그 이펙트를 집합에 추가합니다:

```js
// 이 값은 이펙트가 실행되기 직전에 설정됩니다.
// 이에 대해서는 나중에 다루겠습니다.
let activeEffect

function track(target, key) {
  if (activeEffect) {
    const effects = getSubscribersForProperty(target, key)
    effects.add(activeEffect)
  }
}
```

이펙트 구독은 전역 `WeakMap<target, Map<key, Set<effect>>>` 데이터 구조에 저장됩니다. 속성에 대한 구독 이펙트 집합이 없다면(처음 추적되는 경우), 새로 생성됩니다. 이것이 `getSubscribersForProperty()` 함수가 하는 일입니다. 단순화를 위해 세부 구현은 생략합니다.

`trigger()` 내부에서는 다시 한 번 해당 속성의 구독자 이펙트를 찾습니다. 하지만 이번에는 그것들을 호출합니다:

```js
function trigger(target, key) {
  const effects = getSubscribersForProperty(target, key)
  effects.forEach((effect) => effect())
}
```

이제 다시 `whenDepsChange()` 함수로 돌아가 봅시다:

```js
function whenDepsChange(update) {
  const effect = () => {
    activeEffect = effect
    update()
    activeEffect = null
  }
  effect()
}
```

이 함수는 실제 `update` 함수를 이펙트로 감싸서, 실행 전에 자신을 현재 활성 이펙트로 설정합니다. 이렇게 하면 업데이트 중에 `track()` 호출이 현재 활성 이펙트를 찾을 수 있습니다.

이 시점에서, 우리는 자신의 의존성을 자동으로 추적하고, 의존성이 변경될 때마다 다시 실행되는 이펙트를 만들었습니다. 이를 **반응형 이펙트**라고 부릅니다.

Vue는 반응형 이펙트를 생성할 수 있는 API를 제공합니다: `watchEffect()` . 사실, 이것이 예제의 마법 같은 `whenDepsChange()` 와 매우 비슷하게 동작한다는 것을 눈치챘을 수도 있습니다. 이제 실제 Vue API를 사용하여 원래 예제를 다시 작성해볼 수 있습니다:

```
const A0 = ref(0)
const A1 = ref(1)
const A2 = ref()

watchEffect(() => {
  // A0와 A1을 추적합니다
  A2.value = A0.value + A1.value
})

// 이펙트를 트리거합니다
A0.value = 2
```

반응형 이펙트를 사용해 ref를 변경하는 것은 그다지 흥미로운 사용 사례는 아닙니다. 사실, 계산 속성을 사용하는 것이 더 선언적입니다:

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

const A0 = ref(0)
const A1 = ref(1)
const A2 = computed(() => A0.value + A1.value)

A0.value = 2
```

내부적으로, `computed` 는 반응형 이펙트를 사용해 무효화와 재계산을 관리합니다.

그렇다면 일반적이고 유용한 반응형 이펙트의 예시는 무엇일까요? 바로 DOM 업데이트입니다! 다음과 같이 간단한 "반응형 렌더링"을 구현할 수 있습니다:

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

const count = ref(0)

watchEffect(() => {
  document.body.innerHTML = `Count is: ${count.value}`
})

// DOM을 업데이트합니다
count.value++
```

실제로, 이것은 Vue 컴포넌트가 상태와 DOM을 동기화하는 방식과 매우 유사합니다. 각 컴포넌트 인스턴스는 렌더링과 DOM 업데이트를 위해 반응형 이펙트를 생성합니다. 물론, Vue 컴포넌트는 `innerHTML` 보다 훨씬 효율적인 방법으로 DOM을 업데이트합니다. 이에 대해서는 렌더링 메커니즘에서 다룹니다.

런타임 vs. 컴파일타임 반응성

Vue의 반응성 시스템은 주로 런타임 기반입니다: 추적과 트리거링이 모두 브라우저에서 코드가 실행되는 동안 수행됩니다. 런타임 반응성의 장점은 빌드 단계 없이도 동작할 수 있고, 예외적인 경우가 적다는 점입니다. 반면, 이는 JavaScript의 문법적 한계에 제약을 받게 하여, Vue ref와 같은 값 컨테이너가 필요하게 만듭니다.

Svelte와 같은 일부 프레임워크는 컴파일 시점에 반응성을 구현하여 이러한 한계를 극복합니다. 코드를 분석하고 변환하여 반응성을 시뮬레이션합니다. 컴파일 단계에서는 프레임워크가 JavaScript 자체의 의미를 변경할 수 있습니다. 예를 들어, 지역 변수 접근 시 의존성 분석과 이펙트 트리거링을 수행하는 코드를 암묵적으로 삽입할 수 있습니다. 단점은 이러한 변환이 빌드 단계를 필요로 하고, JavaScript의 의미를 변경하는 것은 본질적으로 JavaScript처럼 보이지만 실제로는 다른 것으로 컴파일되는 언어를 만드는 것과 같습니다.

Vue 팀도 Reactivity Transform이라는 실험적 기능을 통해 이 방향을 탐구했지만, 여기서 설명한 이유로 인해 프로젝트에 적합하지 않다고 판단했습니다.

반응성 디버깅

Vue의 반응성 시스템이 자동으로 의존성을 추적해주는 것은 훌륭하지만, 경우에 따라 정확히 무엇이 추적되고 있는지, 또는 어떤 것이 컴포넌트의 리렌더를 유발하는지 알아내고 싶을 수 있습니다.

컴포넌트 디버깅 훅

컴포넌트의 렌더링 중 어떤 의존성이 사용되고, 어떤 의존성이 업데이트를 트리거하는지 디버깅하려면 `onRenderTracked` 와 `onRenderTriggered` 라이프사이클 훅을 사용할 수 있습니다. 두 훅 모두 해당 의존성에 대한 정보를 담은 디버거 이벤트를 받습니다. 콜백에 `debugger` 문을 넣어 상호작용적으로 의존성을 검사하는 것이 좋습니다:

vue

```
<script setup>
import { onRenderTracked, onRenderTriggered } from 'vue'

onRenderTracked((event) => {
  debugger
})

onRenderTriggered((event) => {
  debugger
```

```
</script>
```

> **TIP**
>
> 컴포넌트 디버그 훅은 개발 모드에서만 동작합니다.

디버그 이벤트 객체는 다음과 같은 타입을 가집니다:

```ts
type DebuggerEvent = {
  effect: ReactiveEffect
  target: object
  type:
    | TrackOpTypes /* 'get' | 'has' | 'iterate' */
    | TriggerOpTypes /* 'set' | 'add' | 'delete' | 'clear' */
  key: any
  newValue?: any
  oldValue?: any
  oldTarget?: Map<any, any> | Set<any>
}
```

계산 속성 디버깅

`computed()` 에 두 번째 옵션 객체로 `onTrack` 과 `onTrigger` 콜백을 전달하여 계산 속성을 디버깅할 수 있습니다:

`onTrack` 은 반응형 속성이나 ref가 의존성으로 추적될 때 호출됩니다.

`onTrigger` 는 의존성의 변경으로 인해 watcher 콜백이 트리거될 때 호출됩니다.

두 콜백 모두 컴포넌트 디버그 훅과 동일한 형식의 디버거 이벤트를 받습니다:

```js
const plusOne = computed(() => count.value + 1, {
  onTrack(e) {
    // count.value가 의존성으로 추적될 때 트리거됨
    debugger
  },
  onTrigger(e) {
    // count.value가 변경될 때 트리거됨
    debugger
  }
})

// plusOne에 접근하면 onTrack이 트리거됨
console.log(plusOne.value)
```

```
count.value++
```

> ⓘ TIP
>
> `onTrack` 과 `onTrigger` 계산 속성 옵션은 개발 모드에서만 동작합니다.

워처 디버깅

`computed()` 와 마찬가지로, 워처도 `onTrack` 과 `onTrigger` 옵션을 지원합니다:

```js
watch(source, callback, {
  onTrack(e) {
    debugger
  },
  onTrigger(e) {
    debugger
  }
})

watchEffect(callback, {
  onTrack(e) {
    debugger
  },
  onTrigger(e) {
    debugger
  }
})
```

> ⓘ TIP
>
> `onTrack` 과 `onTrigger` 워처 옵션은 개발 모드에서만 동작합니다.

외부 상태 시스템과의 통합

Vue의 반응성 시스템은 일반 JavaScript 객체를 깊게 변환하여 반응형 프록시로 만듭니다. 외부 상태 관리 시스템과 통합할 때(예: 외부 솔루션도 Proxy를 사용하는 경우), 깊은 변환이 불필요하거나 원치 않을 수 있습니다.

Vue의 반응성 시스템을 외부 상태 관리 솔루션과 통합하는 일반적인 방법은 외부 상태를 `shallowRef` 에 보관하는 것입니다. shallow ref는 `.value` 속성에 접근할 때만 반응형이며, 내

다.

불변 데이터

실행 취소 / 다시 실행 기능을 구현하려면, 사용자가 편집할 때마다 애플리케이션의 상태 스냅샷을 저장하고 싶을 것입니다. 하지만 Vue의 변경 가능한 반응성 시스템은 상태 트리가 크면 적합하지 않습니다. 매번 전체 상태 객체를 직렬화하는 것은 CPU와 메모리 비용이 많이 들 수 있기 때문입니다.

불변 데이터 구조는 상태 객체를 절대 변경하지 않고, 대신 이전 객체와 동일한 변경되지 않은 부분을 공유하는 새 객체를 만듭니다. JavaScript에서 불변 데이터를 사용하는 방법은 여러 가지가 있지만, Immer를 Vue와 함께 사용하는 것을 추천합니다. Immer를 사용하면 더 편리한 변경 가능한 문법을 유지하면서 불변 데이터를 사용할 수 있습니다.

Immer를 Vue와 통합하는 간단한 컴포저블은 다음과 같습니다:

```js
import { produce } from 'immer'
import { shallowRef } from 'vue'

export function useImmer(baseState) {
  const state = shallowRef(baseState)
  const update = (updater) => {
    state.value = produce(state.value, updater)
  }

  return [state, update]
}
```

플레이그라운드에서 직접 사용해보기

상태 머신

상태 머신은 애플리케이션이 가질 수 있는 모든 상태와, 한 상태에서 다른 상태로 전이할 수 있는 모든 방법을 설명하는 모델입니다. 단순한 컴포넌트에는 과할 수 있지만, 복잡한 상태 흐름을 더 견고하고 관리하기 쉽게 만들어줍니다.

JavaScript에서 가장 인기 있는 상태 머신 구현 중 하나는 XState입니다. 다음은 XState와 통합하는 컴포저블 예시입니다:

```js
import { createMachine, interpret } from 'xstate'
import { shallowRef } from 'vue'

export function useMachine(options) {
  const machine = createMachine(options)
```

```
  const service = interpret(machine)
    .onTransition((newState) => (state.value = newState))
    .start()
  const send = (event) => service.send(event)

  return [state, send]
}
```

플레이그라운드에서 직접 사용해보기

RxJS

RxJS는 비동기 이벤트 스트림을 다루는 라이브러리입니다. VueUse 라이브러리는 RxJS 스트림을 Vue의 반응성 시스템과 연결해주는 `@vueuse/rxjs` 애드온을 제공합니다.

시그널과의 연결

다른 여러 프레임워크가 Vue의 컴포지션 API의 ref와 유사한 반응성 프리미티브를 "시그널"이라는 용어로 도입했습니다:

- Solid Signals
- Angular Signals
- Preact Signals
- Qwik Signals

근본적으로, 시그널은 Vue ref와 동일한 종류의 반응성 프리미티브입니다. 값 컨테이너로서 접근 시 의존성 추적을 제공하고, 변경 시 부수 효과를 트리거합니다. 이러한 반응성 프리미티브 기반 패러다임은 프론트엔드 세계에서 특별히 새로운 개념이 아닙니다. 10년이 넘은 Knockout observables과 Meteor Tracker와 같은 구현까지 거슬러 올라갑니다. Vue 옵션 API와 React 상태 관리 라이브러리 MobX도 동일한 원리에 기반하지만, 프리미티브를 객체 속성 뒤에 숨깁니다.

시그널로 분류되기 위해 반드시 필요한 특성은 아니지만, 오늘날 이 개념은 종종 미세한 구독을 통해 업데이트가 수행되는 렌더링 모델과 함께 논의됩니다. Virtual DOM을 사용하는 Vue는 현재 컴파일러를 통해 유사한 최적화를 달성합니다. 하지만, Vue는 Virtual DOM에 의존하지 않고 Vue의 내장 반응성 시스템을 더 많이 활용하는 Vapor Mode라는 Solid에서 영감을 받은 새로운 컴파일 전략도 탐구하고 있습니다.

API 설계의 트레이드오프

속성을 통한 변경 가능한 인터페이스를 제공합니다. 여기서는 Solid와 Angular 시그널에 초점을 맞추겠습니다.

Solid Signals

Solid의 `createSignal()` API 설계는 읽기/쓰기 분리를 강조합니다. 시그널은 읽기 전용 getter와 별도의 setter로 노출됩니다:

```js
const [count, setCount] = createSignal(0)

count() // 값에 접근
setCount(1) // 값 업데이트
```

`count` 시그널을 setter 없이 하위로 전달할 수 있다는 점에 주목하세요. 이렇게 하면 setter를 명시적으로 노출하지 않는 한 상태를 절대 변경할 수 없습니다. 이 안전성 보장이 더 장황한 문법을 정당화하는지는 프로젝트 요구사항과 개인 취향에 따라 다를 수 있습니다. 만약 이 API 스타일을 선호한다면, Vue에서도 쉽게 구현할 수 있습니다:

```js
import { shallowRef, triggerRef } from 'vue'

export function createSignal(value, options) {
  const r = shallowRef(value)
  const get = () => r.value
  const set = (v) => {
    r.value = typeof v === 'function' ? v(r.value) : v
    if (options?.equals === false) triggerRef(r)
  }
  return [get, set]
}
```

플레이그라운드에서 직접 사용해보기

Angular Signals

Angular는 더티 체킹을 포기하고 자체 반응성 프리미티브 구현을 도입하는 등 근본적인 변화를 겪고 있습니다. Angular Signal API는 다음과 같습니다:

```js
const count = signal(0)

count() // 값에 접근
count.set(1) // 새 값 설정
count.update((v) => v + 1) // 이전 값을 기반으로 업데이트
```

역시, Vue에서 이 API를 쉽게 구현할 수 있습니다:

```
export function signal(initialValue) {
  const r = shallowRef(initialValue)
  const s = () => r.value
  s.set = (value) => {
    r.value = value
  }
  s.update = (updater) => {
    r.value = updater(r.value)
  }
  return s
}
```

플레이그라운드에서 직접 사용해보기

Vue ref와 비교할 때, Solid와 Angular의 getter 기반 API 스타일은 Vue 컴포넌트에서 다음과 같은 흥미로운 트레이드오프를 제공합니다:

`()` 는 `.value` 보다 약간 덜 장황하지만, 값을 업데이트하는 것은 더 장황합니다.
ref 언래핑이 없습니다: 값에 접근할 때 항상 `()` 가 필요합니다. 이는 어디서나 값 접근이 일관됨을 의미합니다. 또한, 원시 시그널을 컴포넌트 props로 그대로 전달할 수 있습니다.

이러한 API 스타일이 본인에게 맞는지는 어느 정도 주관적입니다. 여기서의 목표는 이러한 다양한 API 설계 간의 근본적인 유사성과 트레이드오프를 보여주는 것입니다. 또한 Vue가 유연하다는 점도 보여주고자 합니다. 기존 API에 얽매이지 않고, 필요하다면 더 구체적인 요구에 맞는 자체 반응성 프리미티브 API를 만들 수도 있습니다.