반응형 심화

Vue의 가장 독특한 기능 중 하나는 눈에 거슬리지 않는 반응성 시스템입니다. 컴포넌트 상태는 반응형 자바스크립트 객체로 구성됩니다. 이를 수정하면 뷰가 업데이트됩니다. 상태 관리를 간단하고 직관적으로 만들지만 몇 가지 일반적인 문제를 피하려면 작동 방식을 이해하는 것도 중요합니다. 이 섹션에서는 Vue의 반응성 시스템에 대한 몇 가지 하위 수준의 세부 사항을 살펴보겠습니다.

반응형이란?

요즘 프로그래밍에서 이 용어가 자주 등장하는데, 사람들이 이 말을 할 때 무슨 뜻일까요? 반응성은 선언적인 방식으로 변화에 적응할 수 있는 프로그래밍 패러다임입니다. 반응형 프로그래밍이 훌륭하기 때문에 사람들이 일반적으로 보여주는 대표적인 예는 Excel 스프레드시트입니다:

	A	B	C
0	1		
1	2		
2	3		

여기서 A2 셀은 `= A0 + A1` 의 수식을 통해 정의되므로(A2를 클릭하여 수식을 보거나 편집할 수 있음) 스프레드시트는 우리에게 3을 제공합니다. 그리고 A0 또는 A1을 업데이트하면, A2도 자동으로 업데이트됨을 알 수 있습니다.

JavaScript는 일반적으로 이렇게 작동하지 않습니다. JavaScript에서 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
```

`A0` 을 변경한다고 `A2` 가 자동으로 변경되지 않습니다.

그러면 JavaScript에서 이것을 어떻게 해야 할까요? 먼저 `A2` 를 업데이트하는 코드를 다시 실행하기 위해 다음과 같이 함수로 래핑합니다:

```js
let A2

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

그런 다음 몇 가지 용어를 정의해야 합니다:

- `update()` 함수는 프로그램의 상태를 수정하기 때문에 "**사이드 이펙트**" 줄여서 "**이펙트**"를 생성합니다.
- `A0` 및 `A1` 은 해당 값이 이펙트를 수행하는 데 사용되므로 이펙트의 "**의존성**"(dependency)으로 간주됩니다. 그 이펙트는 의존성에 대한 "**구독자**"(subscriber)라고 합니다.

우리에게 필요한 것은 `A0` 또는 `A1` (**의존성**)이 변경될 때마다 `update()` (**이펙트**)를 호출할 수 있는 함수입니다:

```js
whenDepsChange(update)
```

이 `whenDepsChange()` 함수에는 다음과 같은 작업이 있습니다:

- 변수가 읽힐 때를 추적합니다. 예를 들어 `A0 + A1` 표현식을 평가할 때 `A0` 과 `A1` 이 모두 읽힙니다.

현재 실행 중인 이펙트가 있을 때 변수를 읽으면, 해당 이펙트를 해당 변수의 구독자로 만듭니다. 예를 들어 `A0` 과 `A1` 은 `update()` 가 실행될 때 읽히기 때문에 `update()` 는 첫 번째 호출 이후에 `A0` 과 `A1` 의 구독자가 됩니다.

변수가 언제 변이되는지 감지합니다. 예를 들어 `A0` 에 새 값이 할당되면, 모든 구독자 이펙트에 다시 실행하도록 알립니다.

Vue에서 반응형 작동 방식

예제에서처럼 로컬 변수의 읽기 및 쓰기를 추적할 수는 없습니다. 바닐라 자바스크립트에는 이를 수행할 수 있는 메커니즘이 없기 때문입니다. 하지만 우리가 할 수 있는 것은 **객체 속성**의 읽기 및 쓰기를 가로채는 것입니다.

JavaScript에서 속성 접근을 가로채는 두 가지 방법이 있습니다: **getter / setter**와 **Proxy**입니다. Vue 2는 브라우저 지원 제한으로 인해 getter / setter만 사용했습니다. Vue 3에서는 반응형 객체에 Proxy를 사용하고, ref에 getter / setter를 사용합니다. 다음은 그들이 어떻게 작동하는지를 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 가능한 한 간단한 형태로 핵심 개념을 설명하기 위한 것이기 때문에 많은 세부 사항은 생략되고 예외적인 케이스는 무시됩니다.

아래는 우리가 기초 섹션에서 논의한 몇 가지 반응형의 제한 사항입니다:

반응형 객체의 속성을 로컬 변수에 할당하거나 해체할 때, 해당 변수에 접근하거나 할당하는 것은 비반응형(non-reactive)입니다. 이는 소스 객체의 get/set 프록시 트랩을 더 이상 트리거하지 않기 때문입니다. 이러한 "끊김(disconnect)"은 변수 바인딩에만 영향을 미칩니다. 변수가 객체와 같은 원시 타입이 아닌 값을 가리키는 경우, 해당 객체를 수정하는 것은 여전히 반응형으로 동작합니다.

`reactive()` 에서 반환된 프록시는 원본과 동일하게 동작하지만 `===` 연산자를 사용하여 원본과 비교하면 다른 ID를 갖습니다.

`track()` 내부에서 현재 실행 중인 이펙트가 있는지 확인합니다. 존재하는 경우, 추적 중인 속성에 대한 구독자 이펙트(Set에 저장됨)를 조회하고 이펙트를 Set에 추가합니다.

```js
// 이펙트가 실행되기 직전에 설정됩니다.
// 우리는 나중에 이것에 대해 다룰 것입니다.
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 Set이 발견되지 않은 경우(처음 추적 시) 생성됩니다. 이것이 `getSubscribersForProperty()` 함수가 하는 일입니다. 간단하게 하기 위해 자세한 내용은 건너뛰겠습니다.

`trigger()` 내부에서 속성에 대한 구독자 이펙트를 다시 조회합니다. 그러나 이번에는 우리가 그것들을 대신 호출합니다:

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

이제 다시 `whenDepsChange()` 함수로 돌아가 보겠습니다:

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

실제 업데이트를 실행하기 전에 자신을 현재 활성 이펙트로 설정하는 이펙트에 로우(raw) 업데이트 함수를 래핑합니다. 이것은 현재 활성 이펙트를 찾기 위해 업데이트 동안 `track()` 호출을 활성화합니다.

이 시점에서 의존성을 자동으로 추적하고 의존성이 변경될 때마다 다시 실행하는 이펙트를 만들었습니다. 우리는 이것을 **반응 이펙트**(Reactive Effect)라고 부릅니다.

Vue는 반응 이펙트를 생성할 수 있는 **`watchEffect()`** API를 제공합니다. 사실, 예제의 `whenDepsChange()` 와 매우 유사하게 작동한다는 것을 눈치채셨을 것입니다. 이제 실제 Vue API를 사용하여 원래 예제를 다시 작업할 수 있습니다:

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

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

watchEffect(() => {
  // A0과 A1을 추적함
  A2.value = A0.value + A1.value
})

// 이펙트가 트리거됨
A0.value = 2
```

반응 이펙트를 사용하여 ref를 변경하는 것이 가장 흥미로운 사용 사례는 아닙니다. 계산된 속성을 사용하면 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 무효화 및 재계산을 반응 이펙트를 사용하여 관리합니다.

그렇다면 일반적이고 유용한 반응 이펙트의 예는 무엇일까요? DOM을 업데이트 하는 것입니다! 다음과 같이 간단한 "반응형 렌더링"을 구현할 수 있습니다:

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

const count = ref(0)

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

// DOM 업데이트
count.value++
```

사실 이것은 Vue 컴포넌트가 상태와 DOM을 동기화 상태로 유지하는 방법과 매우 유사합니다. 각 컴포넌트 인스턴스는 DOM을 렌더링하고 업데이트하는 반응 이펙트를 생성합니다. 물론 Vue 컴포넌트는 `innerHTML` 보다 훨씬 더 효율적인 방법으로 DOM을 업데이트합니다. 이것은 렌더링 메커니즘에서 논의됩니다.

런타임 (실행 시) vs. 컴파일 타임 (컴파일 시) 반응형

Vue의 반응성 시스템은 주로 런타임 기반입니다. 추적 및 트리거는 코드가 브라우저에서 직접 실행되는 동안 수행됩니다. 런타임 반응성의 장점은 빌드 단계 없이 작동하며 예외 상황이 더 적습니다. 반면에 JavaScript의 구문 제한 때문에 구문 제한이 있는 JavaScript 값 컨테이너인 Vue refs와 같은 값 컨테이너의 필요성이 생깁니다.

Svelte와 같은 몇몇 프레임워크는 컴파일 중에 반응성을 구현하여 이러한 제한을 극복하기로 선택합니다. 이 프레임워크는 코드를 분석하고 변환하여 반응성을 시뮬레이션합니다. 컴파일 단계를 통해 프레임워크는 JavaScript 자체의 의미를 변경할 수 있습니다. 예를 들어 로컬로 정의된 변수에 대한 액세스 주변의 종속성 분석 및 효과 트리거를 수행하는 코드를 암시적으로 삽입할 수 있습니다. 단점은 이러한 변환이 빌드 단계를 필요로 하며, JavaScript 의미를 변경하는 것은 본질적으로 JavaScript처럼 보이지만 다른 것으로 컴파일되는 언어를 생성하는 것입니다.

Vue 팀은 이 방향을 탐색하기 위해 **Reactivity Transform**이라는 실험적인 기능을 통해 이를 탐색했지만, 결국 여기에서 설명된 이유 때문에 프로젝트에 적합하지 않다고 결정했습니다.

반응형 디버깅

Vue의 반응성 시스템이 종속성을 자동으로 추적하는 것은 훌륭하지만, 어떤 경우에는 추적 대상 또는 컴포넌트가 다시 렌더링되는 원인을 정확히 파악하고 싶을 수 있습니다.

컴포넌트 디버깅 훅

컴포넌트의 렌더링 중에 사용되는 의존성과 `onRenderTracked` 및 `onRenderTriggered` 생명 주기 훅을 사용하여 업데이트를 트리거하는 의존성을 디버그할 수 있습니다. 두 훅 모두 해당 의존성에 대한 정보가 포함된 디버거 이벤트를 수신합니다. 의존성을 대화식으로 검사하기 위해 콜백에 `debugger` 문을 배치하는 것이 좋습니다:

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

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

```
})

onRenderTriggered((event) => {
  debugger
})
</script>
```

TIP

컴포넌트 디버그 훅은 개발 모드에서만 작동합니다.

디버그 이벤트 객체의 타입은 다음과 같습니다:

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>
}
```

계산된 속성 디버깅

onTrack 및 onTrigger 콜백이 있는 두 번째 옵션 객체를 computed() 에 전달하여 계산된 속성을 디버그할 수 있습니다:

onTrack 은 반응 속성 또는 ref가 의존성으로 추적될 때 호출됩니다.
onTrigger 는 의존성의 변경에 의해 감시자 콜백이 트리거될 때 호출됩니다.

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

js

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

// plusOne에 접근, onTrack을 트리거해야 합니다.
console.log(plusOne.value)

// count.value를 변경, onTrigger를 트리거해야 합니다.
count.value++
```

TIP

계산된 속성의 onTrack 및 onTrigger 옵션은 개발 모드에서만 작동합니다.

감시자 디버깅

computed() 와 유사하게 감시자는 onTrack 및 onTrigger 옵션을 지원합니다:

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

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

TIP

감시자의 onTrack 및 onTrigger 옵션은 개발 모드에서만 작동합니다.

외부 상태 시스템과의 통합

Vue의 반응형 시스템은 일반 JavaScript 객체를 반응형 프록시로 깊이 변환하여 작동합니다. 깊은 변환은 외부 상태 관리 시스템과 통합할 때 필요하지 않거나 때때로 원하지 않을 수 있습니다(예: 외부 솔루션도 프록시를 사용하는 경우).

Vue의 반응형 시스템을 외부 상태 관리 솔루션과 통합하는 일반적인 아이디어는 외부 상태를 **shallowRef** 에 유지하는 것입니다. 얕은 참조는 .value 속성에 접근할 때만 반응합니다. 내부 값은 그대로 유지됩니다. 외부 상태가 변경되면 ref 값을 교체하여 업데이트를 트리거합니다.

불변 데이터

실행 취소/다시 실행 기능을 구현하는 경우, 사용자가 편집할 때마다 앱 상태의 스냅샷을 찍고 싶을 것입니다. 그러나 Vue의 변경 가능한 반응형 시스템은 상태 트리가 큰 경우 적합하지 않습니다. 모든 업데이트에서 전체 상태 객체를 직렬화하는 것은 CPU 및 메모리 비용 측면에서 비용이 많이 들 수 있기 때문입니다.

불변 데이터 구조는 상태 객체를 변경하지 않음으로써 이 문제를 해결합니다. 대신 이전 객체와 동일하고 변경되지 않은 부분을 공유하는 새 객체를 생성합니다. JavaScript에서 변경할 수 없는 데이터를 사용하는 방법에는 여러 가지가 있지만, Vue와 함께 **Immer**를 사용하는 것이 좋습니다. 왜냐하면 보다 인체 공학적이고 변경 가능한 문법을 유지하면서 변경할 수 없는 데이터를 사용할 수 있기 때문입니다.

간단한 컴포넌트를 통해 Immer를 Vue와 통합할 수 있습니다:

```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]
}
```

온라인 연습장으로 실행하기

상태 머신 (State Machine)

상태 머신은 앱이 어떤 상태에 있을 수 있는 모든 가능한 상태와 한 상태에서 다른 상태로 전환할 수 있는 모든 가능한 방법을 설명하기 위한 모델입니다. 단순한 컴포넌트에는 과도할 수 있지만, 복잡한 상태 흐름을 보다 강력하고 관리하기 쉽게 만드는 데 도움이 될 수 있습니다.

JavaScript에서 가장 널리 사용되는 상태 머신 구현 중 하나는 **XState**입니다. 다음은 이를 통합하는 컴포저블입니다:

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

export function useMachine(options) {
  const machine = createMachine(options)
  const state = shallowRef(machine.initialState)
  const service = interpret(machine)
    .onTransition((newState) => (state.value = newState))
    .start()
  const send = (event) => service.send(event)

  return [state, send]
}
```

온라인 연습장으로 실행하기

RxJS

RxJS는 비동기 이벤트 스트림을 다루기 위한 라이브러리입니다. **VueUse** 라이브러리는 Vue의 반응성 시스템과 RxJS 스트림을 연결하기 위한 `@vueuse/rxjs` 애드온을 제공합니다.

신호 연결

Vue의 Composition API에서 refs와 유사한 반응성 기본 요소를 "신호(signals)"라는 용어로 소개한 다른 프레임워크가 꽤 있습니다:

- **Solid** 신호
- **Angular** 신호
- **Preact** 신호
- **Qwik** 신호

기본적으로, 신호는 Vue refs와 같은 종류의 반응성 기본 요소입니다. 접근 시 주입(provide) 추적 및 변화(mutation) 시 사이드 이펙트 트리거링을 제공하는 값 컨테이너입니다. 이 반응성 기본 요소 기반 패러다임은 프론트엔드 세계에서 특히 새로운 개념이 아닙니다: **Knockout** 관찰가능 객체와 **Meteor Tracker**와 같은 구현으로 10년 이상 전으로 거슬러 올라갑니다. Vue Options API와 React 상태 관리 라이브러리 **MobX**도 같은 원칙에 기반하지만, 기본 요소를 객체 속성 뒤에 숨깁니다.

신호로 간주되기 위해 필수적인 특성은 아니지만, 오늘날 이 개념은 종종 세밀한 구독을 통해 업데이트가 수행되는 렌더링 모델과 함께 논의됩니다. Virtual DOM을 사용하기 때문에 Vue는 현재 유사한 최적화를 달성하기 위해 컴파일러에 의존합니다. 그러나 우리는 Virtual DOM에 의존하지 않고 Vue의 내장된 반응성 시스템을 더 활용하는 Solid에서 영감을 받은 새로운 컴파일 전략인 **Vapor Mode**도 탐구하고 있습니다.

API 디자인의 절충

Preact와 Qwik의 신호 디자인은 Vue의 **shallowRef**와 매우 유사합니다: 모두 `.value` 속성을 통해 변경 가능한 인터페이스를 제공합니다. 우리는 Solid와 Angular 신호에 대한 논의에 초점을 맞출 것입니다.

Solid 신호

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

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

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

`count` 신호가 setter 없이 전달될 수 있다는 점을 주목하세요. 이는 setter가 명시적으로 노출되지 않는 한 상태가 결코 변경될 수 없다는 것을 보장합니다. 이 안전 보장이 더 장황한 구문을 정당화하는지 여부는 프로젝트의 요구 사항과 개인의 취향에 따라 다를 수 있지만, 이 API 스타일을 선호하는 경우 Vue에서 쉽게 복제할 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]
}
```

온라인 연습장으로 실행하기

Angular 신호

Angular는 더티 체크(dirty-checking)를 포기하고 자체 반응성 기본 요소의 구현을 도입함으로써 일부 근본적인 변화를 겪고 있습니다. Angular 신호 API는 다음과 같습니다:

```js
const count = signal(0)

count() // 값에 접근
count.set(1) // 새로운 값 설정
count.update((v) => v + 1) // 이전 값에 기반한 업데이트
```

다시 한번, 우리는 Vue에서 API를 쉽게 복제할 수 있습니다:

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

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
}
```

온라인 연습장으로 실행하기

Vue refs와 비교할 때, Solid와 Angular의 getter 기반 API 스타일은 Vue 컴포넌트에서 사용될 때 몇 가지 흥미로운 절충을 제공합니다:

- `()` 는 `.value` 보다 약간 덜 장황하지만, 값을 업데이트하는 것은 더 장황합니다.
- ref-언래핑이 없습니다: 값을 접근할 때 항상 `()` 이 필요합니다. 이는 모든 곳에서 값 접근을 일관되게 합니다. 이는 또한 원시 신호를 컴포넌트 props로 내려보낼 수 있음을 의미합니다.

이러한 API 스타일이 여러분에게 맞는지는 어느 정도 주관적입니다. 우리의 목표는 이러한 다른 API 디자인 간의 근본적 유사성과 절충을 보여주는 것입니다. 또한 Vue가 유연하다는 것을 보여주고 싶습니다: 실제로 기존 API에 완전히 묶여 있지 않습니다. 필요한 경우, 더 구체적인 요구 사항에 맞게 자체 반응성 기본 요소 API를 만들 수 있습니다.