렌더링 메커니즘

Vue는 템플릿을 어떻게 가져와 실제 DOM 노드로 변환합니까? Vue는 이러한 DOM 노드를 어떻게 효율적으로 업데이트합니까? Vue의 내부 렌더링 메커니즘에 대해 자세히 살펴봄으로써 이 질문들에 대해 설명하고자 합니다.

가상 DOM

Vue의 렌더링 시스템이 기반으로 하는 "가상 DOM(VDOM)"이라는 용어에 대해 들어본 적이 있을 것입니다.

가상 DOM은 UI의 이상적인 또는 "가상" 표현을 메모리에 저장하고, "실제" DOM과 동기화하는 프로그래밍 개념입니다. 이 개념은 **React**에 의해 개척되었으며, Vue를 비롯해 다른 많은 프레임워크에 적용되었습니다.

가상 DOM은 특정 기술보다 패턴에 가깝기 때문에 하나의 정식 구현이 없습니다. 간단한 예를 사용하여 이것을 설명할 수 있습니다:

```js
const vnode = {
  type: 'div',
  props: {
    id: 'hello'
  },
  children: [
    /* 더 많은 가상 노드(vnode)*/
  ]
}
```

여기서 `vnode` 는 `<div>` 엘리먼트를 나타내는 일반 JavaScript 객체("가상 노드")입니다. 여기에는 실제 엘리먼트를 만드는 데 필요한 모든 정보가 포함되어 있습니다. 또한 더 많은 자식 vnode를 포함하므로 가상 DOM 트리의 루트가 됩니다.

런타임 렌더러는 가상 DOM 트리를 탐색하고, 이 트리에서 실제 DOM 트리를 구성할 수 있습니다. 이 프로세스를 **마운트**(mount)라고 합니다.

가상 DOM 트리의 복사본이 두 개 있는 경우, 렌더러는 두 트리를 살펴보고 비교하여 차이점을 파악하고 이러한 변경 사항을 실제 DOM에 적용할 수도 있습니다. 이 프로세스를 **패치**(patch)라고 하며, "디핑(diffing)" 또는 "레컨실리에이션(reconciliation)"이라고도 합니다.

가상 DOM의 주요 장점은 개발자가 직접 DOM 조작을 렌더러에 맡기고 선언적 방식으로 원하는 UI 구조를 프로그래밍 방식으로 생성, 검사 및 구성할 수 있는 기능을 제공한다는 것입니다.

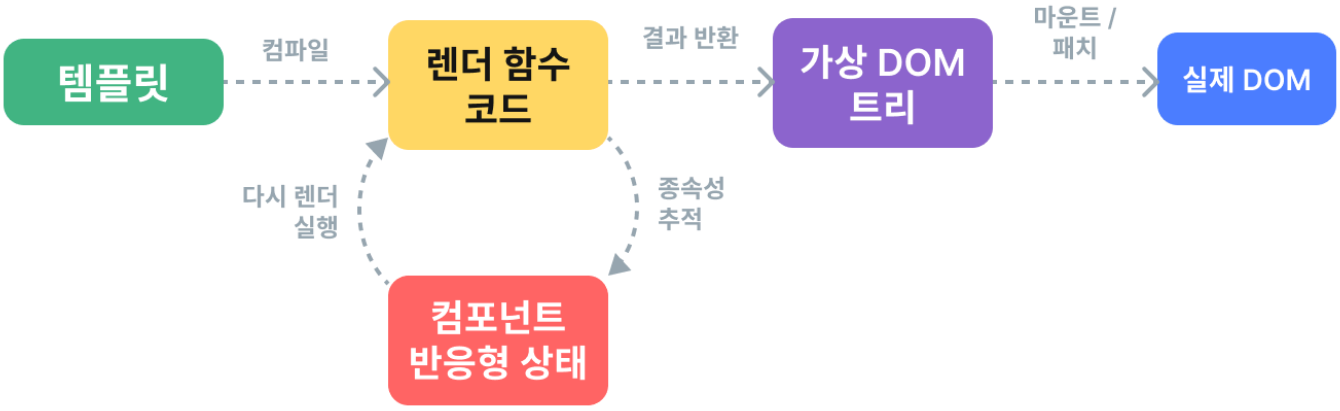
렌더 파이프라인

고수준에서 이것은 Vue 컴포넌트가 마운트될 때 발생합니다:

컴파일: Vue 템플릿은 **렌더 함수**로 컴파일 됨: 가상 DOM 트리를 반환하는 함수. 이 단계는 빌드 과정을 통해 미리 실행하거나 런타임 컴파일러를 사용하여 즉석에서 실행할 수 있습니다.

마운트: 런타임 렌더러는 렌더 함수를 호출하고, 반환된 가상 DOM 트리를 탐색하고, 이를 기반으로 실제 DOM 노드를 생성합니다. 이 단계는 반응 이펙트가 실행되므로 사용된 모든 반응형 의존성을 추적합니다.

패치: 마운트 중에 사용된 의존성이 변경되면 이펙트가 다시 실행됩니다. 이번에는 업데이트된 새로운 가상 DOM 트리가 생성됩니다. 런타임 렌더러는 새 트리를 탐색하고 이전 트리와 비교하고 필요한 업데이트를 실제 DOM에 적용합니다.



템플릿 vs. 렌더 함수

Vue 템플릿은 가상 DOM 렌더 함수로 컴파일됩니다. Vue는 또한 템플릿 컴파일 단계를 건너뛰고 렌더 함수를 직접 작성할 수 있는 API를 제공합니다. 렌더 함수는 JavaScript의 모든 함수를 사용하여 vnode로 작업할 수 있기 때문에 매우 동적인 로직을 다룰 때 템플릿보다 더 유연합니다.

그렇다면 Vue가 기본적으로 템플릿을 권장하는 이유는 무엇일까요? 여러 가지 이유가 있습니다:

> 템플릿은 실제 HTML에 더 가깝습니다. 이를 통해 기존 HTML 스니펫을 재사용하고, 접근성 모범 사례를 적용하고, CSS로 스타일을 지정하고, 디자이너가 이해하고 수정할 수 있습니다.
>
> 템플릿 문법은 정적으로 분석하기가 더 쉽습니다. 이를 통해 Vue의 템플릿 컴파일러는 가상 DOM의 성능을 향상시키기 위해 많은 컴파일 시간 최적화를 적용할 수 있습니다(아래에서 논의함).

실제로 대부분의 앱은 템플릿을 사용하는 것으로 충분합니다. 렌더링 함수는 일반적으로 매우 동적인 렌더링 로직을 처리해야 하는 재사용 가능한 컴포넌트에서만 사용됩니다. 렌더 함수 사용법은 렌더 함수 **&** JSX에서 자세히 설명합니다.

Compiler-Informed Virtual DOM

React와 대부분의 다른 가상 DOM 구현은 순전히 런타임입니다. 조정(reconciliation) 알고리즘에 할당되는 가상 DOM 트리에 대해 어떠한 가정할 수 없으므로, 정확성을 보장하기 위해 트리를 완전히 탐색하고 모든 vnode의 props를 비교해야 합니다. 또한 트리의 일부가 변경되지 않더라도 다시 렌더링할 때마다 항상 새로운 vnode가 생성되어 불필요한 메모리 부하가 발생합니다. 이것은 가상 DOM의 가장 비판적인 측면 중 하나입니다. 다소 무차별적인 조정 프로세스는 선언성과 정확성에 대한 대가로 효율성을 희생합니다.

하지만 꼭 그럴 필요는 없습니다. Vue에서 프레임워크는 컴파일러와 런타임을 모두 제어합니다. 이를 통해 밀접하게 연결된 렌더러만 활용할 수 있는 많은 컴파일-타임 최적화를 구현할 수 있습니다. 컴파일러는 템플릿을 정적으로 분석하고, 생성된 코드에 힌트를 남겨 런타임이 가능할 때마다 바로 가기를 사용할 수 있도록 합니다. 동시에 에지(edge) 케이스에서 보다 직접적인 제어를 위해, 사용자가 렌더 함수 레이어로 드롭다운할 수 있는 기능을 계속 유지합니다. 우리는 이 하이브리드 접근 방식을 **Compiler-Informed Virtual DOM**(컴파일러에 알려진 가상 DOM)이라고 부릅니다.

아래에서는 가상 DOM의 런타임 성능을 향상시키기 위해, Vue 템플릿 컴파일러가 수행하는 몇 가지 주요 최적화에 대해 설명합니다.

정적 호이스팅 (Static Hoisting)

템플릿에는 동적 바인딩이 포함되지 않은 부분이 있는 경우가 많습니다:

template

```
<div>
  <div>foo</div> <!-- 호이스트(hoist) 됨 -->
  <div>bar</div> <!-- 호이스트 됨 -->
```

```
  <div>{{ dynamic }}</div>
</div>
```

템플릿 탐색기에서 확인하기

foo 및 bar div는 정적이므로 리렌더에서 vnode를 다시 생성하고 이를 비교하는 것은 불필요합니다. Vue 컴파일러는 vnode 생성 함수 호출 시 자동으로 호이스트하여 모든 렌더에서 동일한 vnode를 재사용합니다. 렌더러는 이전 vnode와 새 vnode가 동일함을 발견하면 비교를 건너뛸 수 있습니다.

또한 정적 엘리먼트가 충분히 연속적으로 존재하면, 이러한 모든 노드에 대한 일반 HTML 문자열을 포함하는 단일 "정적 vnode"로 압축됩니다(예제). 이러한 정적 vnode는 innerHTML 을 직접 설정하여 마운트됩니다. 또한 초기 마운트 시 해당 DOM 노드를 캐시합니다. 동일한 컨텐츠가 앱의 다른 곳에서 재사용되는 경우 기본 cloneNode() 를 사용하여 새 DOM 노드가 생성되며 이는 매우 효율적입니다.

패치 플래그

동적 바인딩이 있는 단일 엘리먼트의 경우 컴파일 시 많은 정보를 추론할 수도 있습니다:

template

```
<!-- class만 바인딩 -->
<div :class="{ active }"></div>

<!-- id와 value만 바인딩 -->
<input :id="id" :value="value">

<!-- text만 only -->
<div>{{ dynamic }}</div>
```

템플릿 탐색기에서 확인하기

이러한 엘리먼트에 대한 렌더 함수 코드를 생성할 때, Vue는 vnode 생성 호출에서 직접 각 엘리먼트에 필요한 업데이트 유형을 인코딩합니다:

js

```
createElementVNode("div", {
  class: _normalizeClass({ active: _ctx.active })
}, null, 2 /* CLASS */)
```

마지막 인자 2 는 패치 플래그입니다. 엘리먼트는 단일 숫자로 병합될 여러 패치 플래그를 가질 수 있습니다. 그런 다음 런타임 렌더러는 비트 연산을 사용하여 플래그에 대해 검사하여 특정 작업을 수행해야 하는지 여부를 결정할 수 있습니다.

js

```
if (vnode.patchFlag & PatchFlags.CLASS /* 2 */) {
  // 엘리먼트의 클래스 업데이트
}
```

비트 단위 검사는 매우 빠릅니다. 패치 플래그를 사용하면 Vue는 동적 바인딩으로 엘리먼트를 업데이트할 때 필요한 최소한의 작업을 수행할 수 있습니다.

Vue는 또한 vnode가 가진 자식 유형을 인코딩합니다. 예를 들어 여러 루트 노드가 있는 템플릿은 프래그먼트(fragment)로 표시됩니다. 대부분의 경우 이러한 루트 노드의 순서가 절대 변경되지 않는다는 것을 알고 있으므로, 이 정보를 패치 플래그로 런타임에 제공할 수도 있습니다:

js

```
export function render() {
  return (_openBlock(), _createElementBlock(_Fragment, null, [
    /* children */
  ], 64 /* STABLE_FRAGMENT */))
}
```

따라서 런타임은 루트 프래그먼트에 대한 자식 순서 조정을 완전히 건너뛸 수 있습니다:

Tree Flattening

이전 예제에서 생성된 코드를 다시 살펴보면, 반환된 가상 DOM 트리의 루트가 `createElementBlock()` 호출로 생성되었음을 알 수 있습니다:

```js
export function render() {
  return (_openBlock(), _createElementBlock(_Fragment, null, [
    /* children */
  ], 64 /* STABLE_FRAGMENT */))
}
```

개념적으로 "블록(block)"은 내부 구조가 안정적인 템플릿의 일부입니다. 이 경우 전체 템플릿은 `v-if` 및 `v-for` 와 같은 구조적 디렉티브를 포함하지 않기 때문에 단일 블록을 갖습니다.

각 블록은 패치 플래그가 있는 모든 하위 노드(직계 자식뿐만 아니라)를 추적합니다. 예를 들어:

```template
<div> <!-- 루트 블록 -->
  <div>...</div>         <!-- 추적하지 않음 -->
  <div :id="id"></div>   <!-- 추적함 -->
  <div>                  <!-- 추적하지 않음 -->
    <div>{{ bar }}</div> <!-- 추적함 -->
  </div>
</div>
```

그 결과 동적 하위 노드만 포함하는 병합된(flattened) 배열이 생성됩니다:

```
div (block root)
- div with :id binding
- div with {{ bar }} binding
```

이 컴포넌트를 다시 렌더링해야 할 때 전체 트리 대신 병합된 트리만 탐색하면 됩니다. 이를 **트리 병합**(tree flattening)이라고 하며, 가상 DOM 조정 중에 통과해야 하는 노드의 수를 크게 줄입니다. 템플릿의 모든 정적인 부분은 효과적으로 건너뜁니다.

`v-if` 및 `v-for` 디렉티브는 새 블록 노드를 생성합니다.

```template
<div> <!-- 루트 블록 -->
  <div>
    <div v-if> <!-- if 블록 -->
      ...
    <div>
  </div>
</div>
```

자식 블록은 부모 블록의 동적 자손 배열 내에서 추적됩니다. 이것은 상위 블록에 대한 안정적인 구조를 유지합니다.

SSR 하이드레이션(hydration)에서의 영향

패치 플래그와 트리 병합은 Vue의 **SSR** 하이드레이션 성능도 크게 향상시킵니다.

단일 엘리먼트 하이드레이션은 해당 vnode의 패치 플래그를 기반으로 빠른 경로를 취할 수 있습니다.

블록 노드와 해당 동적 자손만 하이드레이션 중에 통과되야 템플릿 수준에서 부분 하이드레이션를 효과적으로 달성할 수 있습니다.