성능 (Performance)

개요

Vue는 수동 최적화가 크게 필요하지 않은 가장 일반적인 사용 사례에 적합하도록 설계되었습니다. 그러나 추가 미세 조정이 필요한 어려운 시나리오가 항상 있습니다. 이 섹션에서는 Vue 앱 성능과 관련하여 주의해야 할 사항에 대해 설명합니다.

먼저 웹 성능의 두 가지 주요 측면에 대해 논의해 보겠습니다:

- **페이지 로드 성능**: 앱이 컨텐츠를 표시하고 초기 방문 시 대화형이 되는 속도. 이는 일반적으로 최대 컨텐츠 풀 페인트**(LCP: Largest Contentful Paint)** 및 최초 입력 지연**(FID: First Input Delay)**와 같은 웹 필수 측정기들을 사용하여 측정됩니다.

- **업데이트 성능**: 사용자 입력에 대한 응답으로 앱이 업데이트되는 속도. 예를 들어 사용자가 검색 상자에 입력할 때 목록이 업데이트되는 속도 또는 사용자가 싱글 페이지 앱(SPA)에서 탐색 링크를 클릭할 때 페이지가 전환되는 속도입니다.

두 가지를 모두 최대화하는 것이 이상적이지만, 서로 다른 프론트엔드 아키텍처는 성능을 얼마나 쉽게 달성할 수 있는지에 영향을 미칩니다. 또한 성능의 우선순위를 어떻게 지정했냐에 따라 구축하는 앱이 크게 영향을 받습니다. 따라서 최적의 성능을 보장하는 첫 번째 단계는 구축 중인 앱에 적합한 아키텍처를 선택하는 것입니다.

- Vue를 다양한 방식으로 활용하는 방법을 보려면 **Vue** 사용 방법을 참고하세요.

- Jason Miller의 블로그 **Application Holotypes**(앱 유형)에서 웹 앱의 유형과 이상적인 구현 및 제공에 대해 논의합니다.

프로파일링 옵션 (profiling options)

성능을 개선하려면 먼저 측정 방법을 알아야 합니다. 이와 관련하여 도움이 될 수 있는 훌륭한 도구가 많이 있습니다.

프로덕션 배포의 로드 성능 프로파일링:

- **PageSpeed Insights**
- **WebPageTest**

로컬 개발 중 성능 프로파일링:

- 크롬 개발자도구 성능**(Performance)** 패널
  - **app.config.performance** 는 크롬 개발자도구의 성능 타임라인에서 Vue 관련 성능 마커를 활성화합니다.

- **Vue** 개발자도구 확장은 성능 프로파일링 기능도 제공합니다.

페이지 로드 최적화

페이지 로드 성능을 최적화하는 것은 프레임워크에 구애받지 않는 경우가 많습니다. 포괄적인 정리 내용은 **web.dev** 가이드를 확인하세요. 여기서는 주로 Vue에 특화된 기술에 초점을 맞출 것입니다.

올바른 아키텍처 선택

사용 사례가 페이지 로드 성능에 민감한 경우 순수한 클라이언트 측 SPA로 전송하지 마세요. 사용자가 보고자 하는 콘텐츠가 포함된 HTML을 서버가 직접 전송해야 합니다. 순수 클라이언트 측 렌더링은 콘텐츠에 도달하는 시간이 느립니다. 서버 측 렌더링**(SSR)** 또는 정적 사이트 생성**(SSG)**을 사용하면 이를 완화할 수 있습니다. **SSR** 가이드를 확인하여 Vue로 SSR을 수행하는 방법에 대해 알아보세요. 앱에 풍부한 인터랙티브 요구 사항이 없는 경우 기존 백엔드 서버를 사용하여 HTML을 렌더링하고 클라이언트에서 Vue를 사용하여 향상시킬 수도 있습니다.

기본 애플리케이션이 SPA여야 하지만 마케팅 페이지(랜딩, 정보, 블로그)가 있는 경우 마케팅 페이지를 별도로 제공하세요! 마케팅 페이지는 SSG를 사용하여 최소한의 JS가 포함된 정적 HTML로 배포하는 것이 이상적입니다.

번들 크기 및 트리 쉐이킹 (tree-shaking)

페이지 로드 성능을 향상시키는 가장 효과적인 방법 중 하나는 더 작은 JavaScript 번들을 제공하는 것입니다. Vue를 사용할 때 번들 크기를 줄이는 몇 가지 방법은 다음과 같습니다:

가능하면 빌드 과정을 사용하십시오.

Vue의 많은 API는 최신 빌드 도구를 통해 번들로 제공되는 경우, "트리 쉐이킹"합니다. 예를 들어 내장된 `<Transition>` 컴포넌트를 사용하지 않으면, 최종 프로덕션 번들에 포함되지 않습니다. 트리 쉐이킹은 소스 코드에서 사용하지 않는 다른 모듈도 제거할 수 있습니다.

빌드 과정을 사용할 때, 템플릿이 미리 컴파일되므로 Vue 컴파일러를 브라우저에 제공할 필요가 없습니다. 이것은 **14kb** 만큼을 절약 후 min+gzip으로 압축된 JavaScript로 런타임 컴파일 비용을 방지합니다.

새로운 의존성을 도입할 때 크기에 주의하십시오! 실제 앱에서 비대해진 번들은 대부분 자신도 모르는 사이에 과도한 의존성을 도입한 결과입니다.

빌드 과정을 사용하는 경우, ES 모듈 형식을 제공하고, 트리 쉐이킹 친화적인 의존성을 선호하십시오. 예를 들어 `lodash` 보다 `lodash-es` 를 선호합니다.

의존성의 크기를 확인하고 그것이 제공하는 기능의 가치가 있는지 평가하십시오. 의존성이 트리 쉐이킹에 친화적이면 실제 크기 증가는 가져오는 API에 따라 달라집니다. **bundlejs.com**과 같은 도구를 사용하여 빠르게 확인할 수 있지만, 실제 빌드 설정으로 측정하는 것이 항상 가장 정확합니다.

주로 빌드 과정 없이 Vue를 사용하고 있다면, **petite-vue(6kb)**를 사용하는 것이 좋습니다.

코드 분할

코드 분할은 빌드 도구가 앱 번들을 여러 개의 작은 청크로 분할하는 것으로, 로드 또는 요청 시 병렬로 로드할 수 있는 곳입니다. 적절한 코드 분할을 사용하면 페이지 로드 시, 필요한 기능을 즉시 다운로드할 수 있으며, 추가 청크는 필요할 때만 지연 로드되므로 성능이 향상됩니다.

Rollup(Vite의 기반) 또는 webpack과 같은 번들러는 ESM 동적 가져오기 문법을 감지하여 자동으로 분할 청크를 생성할 수 있습니다.

```js
// lazy.js와 그 의존성은 별도의 청크로 분할되며,
// `loadLazy()`가 호출될 때만 로드됩니다.
function loadLazy() {
  return import('./lazy.js')
}
```

지연 로딩은 초기 페이지 로드 후 즉시 필요하지 않은 기능에 사용하는 것이 가장 좋습니다. Vue 애플리케이션에서는 Vue의 비동기 컴포넌트 기능과 함께 사용하여 컴포넌트 트리에 대한 분할 청크를 만들 수 있습니다:

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

// Foo.vue 및 해당 의존성에 대해 별도의 청크가 생성됩니다.
// 비동기 컴포넌트가 페이지에서 렌더링될 때에만
// 요청하여 가져옵니다.
const Foo = defineAsyncComponent(() => import('./Foo.vue'))
```

Vue 라우터를 사용하는 애플리케이션의 경우 라우팅 컴포넌트에 지연 로딩을 사용할 것을 강력히 권장합니다. Vue 라우터는 `defineAsyncComponent` 와는 별도로 지연 로딩을 명시적으로 지원합니다. 자세한 내용은 지연 로딩 라우트를 참고하세요.

업데이트 최적화

Props 안정성

Vue에서 자식 컴포넌트는 수신된 props 중 하나 이상이 변경된 경우에만 업데이트됩니다. 다음 예제를 봅시다:

```template
<ListItem
  v-for="item in list"
  :id="item.id"
  :active-id="activeId" />
```

<ListItem> 컴포넌트 내에서 id 및 activeId prop을 사용하여 현재 활성 아이템인지 여부를 결정합니다. 이것이 작동하는 동안 문제는 activeId 가 변경될 때마다 목록의 **모든** <ListItem> 이 업데이트되어야 한다는 것입니다!

이상적으로는 활성 상태가 변경된 아이템만 업데이트되어야 합니다. 활성 상태 계산을 부모로 이동하고, <ListItem> 이 active prop을 직접 받도록 하여 이를 달성할 수 있습니다:

```template
<ListItem
  v-for="item in list"
  :id="item.id"
  :active="item.id === activeId" />
```

이제 대부분의 컴포넌트에서 active prop은 activeId 가 변경될 때 동일하게 유지되므로, 더 이상 업데이트할 필요가 없습니다. 이 개념은 일반적으로 자식 컴포넌트에 전달되는 prop을 가능한 한 안정적으로 유지하는 것입니다.

v-once

v-once 는 런타임 데이터에 의존하지만, 업데이트할 필요가 없는 컨텐츠를 렌더링하는 데 사용할 수 있는 내장 디렉티브입니다. 이것을 사용하는 컴포넌트 내 전체 하위 트리는 이후 모든 업데이트를 건너뜁니다. 자세한 내용은 **API** 레퍼런스를 참고하세요.

v-memo

v-memo 는 큰 하위 트리 또는 v-for 목록의 업데이트를 조건부로 건너뛰는 데 사용할 수 있는 내장 디렉티브입니다. 자세한 내용은 **API** 레퍼런스를 참고하세요.

계산된 안정성

3.4 버전부터, 계산된 속성은 이전과 다른 계산된 값이 나올 때만 효과를 발생시킵니다. 예를 들어, 다음과 같은 isEven 계산은 반환된 값이 true 에서 false 로, 또는 그 반대로 변경될 때만 효과를 발생시킵니다:

```js
const count = ref(0)
const isEven = computed(() => count.value % 2 === 0)

watchEffect(() => console.log(isEven.value)) // true

// 계산된 값이 `true`로 유지되기 때문에 새로운 로그를 발생시키지 않습니다
count.value = 2
count.value = 4
```

이는 불필요한 효과 발생을 줄이지만, 계산된 값이 매번 새로운 객체를 생성하는 경우에는 작동하지 않습니다:

```js
const computedObj = computed(() => {
  return {
    isEven: count.value % 2 === 0
  }
})
```

매번 새로운 객체가 생성되므로, 새로운 값은 기술적으로 항상 이전 값과 다릅니다. `isEven` 속성이 동일하게 유지되더라도 Vue는 오래된 값과 새로운 값을 깊은 비교(deep comparison)를 하지 않는 한 알 수 없습니다. 이러한 비교는 비용이 많이 들고 가치가 없을 수 있습니다.

대신, 새로운 값과 오래된 값을 수동으로 비교하여, 변경되지 않았다는 것을 알고 있을 경우 오래된 값을 반환하도록 최적화할 수 있습니다:

```js
const computedObj = computed((oldValue) => {
  const newValue = {
    isEven: count.value % 2 === 0
  }
  if (oldValue && oldValue.isEven === newValue.isEven) {
    return oldValue
  }
  return newValue
})
```

Playground 예제

항상 오래된 값을 비교하고 반환하기 전에 전체 계산을 수행해야 하므로, 매번 실행될 때 동일한 의존성이 수집될 수 있습니다.

일반적인 최적화

다음 팁은 페이지 로드와 업데이트 성능에 모두 영향을 미칩니다.

대규모 목록 가상화

모든 프론트엔드 앱에서 가장 일반적인 성능 문제 중 하나는, 큰 목록을 렌더링하는 것입니다. 프레임워크의 성능이 아무리 뛰어나더라도 수천 개의 아이템이 포함된 목록을 렌더링하는 것은 **브라우저가 처리해야 하는 DOM 노드의 수 때문에 느려질 것**입니다.

그러나 모든 노드를 미리 렌더링할 필요는 없습니다. 대부분의 경우, 사용자의 화면 크기는 큰 목록의 작은 하위 집합만 표시할 수 있습니다. 큰 목록에서 현재 표시 영역에 있거나 가까운 아이템만 렌더링하는 기술인 **목록 가상화**로 성능을 크게 향상할 수 있습니다.

목록 가상화를 구현하는 것은 쉽지 않습니다. 다행히 직접 사용할 수 있는 기존 커뮤니티 라이브러리가 있습니다:

- **vue-virtual-scroller**
- **vue-virtual-scroll-grid**
- **vueuc/VVirtualList**

큰 불변 구조에 대한 반응성 오버헤드 감소

Vue의 반응형 시스템은 기본적으로 깊습니다. 이러면 상태 관리가 직관적이지만, 데이터 크기가 클 경우 특정 수준의 오버헤드가 발생하는데, 이것은 모든 속성 접근이 의존성 추적을 수행하는 프록시 트랩을 트리거하기 때문입니다. 이것은 일반적으로 깊이 중첩된 객체의 큰 배열을 처리할 때, 단일 렌더가 100,000개 이상의 속성에 접근하는 경우 눈에 띄게 나타나므로, 매우 특정한 사용 사례에만 영향을 미칩니다.

Vue는 **`shallowRef()`** 및 **`shallowReactive()`** 를 사용하여, 깊은 반응형을 opt-out하는 탈출구를 제공합니다. 얕은 API는 루트 수준에서만 반응하는 상태를 생성하고, 모든 중첩된 객체를 그대로 노출합니다. 이렇게 하면 중첩된 속성에 빠르게 접근할 수 있으며, 모든 중첩된 객체는 변경할 수 없는 것처럼 처리해야 하고, 루트 상태를 교체해야만 업데이트가 트리거될 수 있습니다:

```js
const shallowArray = shallowRef([
  /* 깊은 객체의 큰 목록 */
])

// 업데이트가 실행되지 않습니다...
shallowArray.value.push(newObject)
// 이것은 다음을 수행합니다:
shallowArray.value = [...shallowArr.value, newObject]

// 업데이트가 실행되지 않습니다...
```

```
shallowArray.value[0].foo = 1
// 이것은 다음을 수행합니다:
shallowArray.value = [
  {
    ...shallowArray.value[0],
    foo: 1
  },
  ...shallowArray.value.slice(1)
]
```

불필요한 컴포넌트 추상화 방지

때때로 더 나은 추상화 또는 코드 구성을 위해 렌더링 없는 컴포넌트 또는 상위 컴포넌트(즉, 추가 프로퍼티가 있는 다른 컴포넌트를 렌더링하는 컴포넌트)를 만들 수 있습니다. 이것이 잘못된 것은 아니지만 컴포넌트 인스턴스는 일반 DOM 노드보다 훨씬 비싸고 추상화 패턴으로 인해 너무 많이 생성하면 성능 비용이 발생할 수 있다는 점을 명심하세요.

몇 개의 인스턴스만 줄이는 것은 눈에 띄는 효과가 없으므로, 컴포넌트가 앱에서 몇 번만 렌더링되는 경우, 이것을 불필요하게 추상화하지 않아도 됩니다. 이 최적화를 고려하는 가장 좋은 시나리오는 "**큰 목록**"입니다. 100개의 아이템 목록이 있고, 각 아이템 컴포넌트에 많은 하위 컴포넌트가 포함되어 있다고 상상해 보십시오. 여기서 불필요한 컴포넌트 추상화 하나를 제거하면, 수백 개의 컴포넌트 인스턴스가 줄어들 수 있습니다.