성능

개요

Vue는 대부분의 일반적인 사용 사례에서 별도의 수동 최적화 없이도 성능이 뛰어나도록 설계되었습니다. 하지만 항상 추가적인 미세 조정이 필요한 까다로운 상황이 존재합니다. 이 섹션에서는 Vue 애플리케이션의 성능과 관련하여 주의해야 할 점에 대해 논의합니다.

먼저, 웹 성능의 두 가지 주요 측면에 대해 살펴보겠습니다:

페이지 로드 성능: 애플리케이션이 처음 방문 시 얼마나 빠르게 콘텐츠를 표시하고 상호작용이 가능한 상태가 되는지. 이는 보통 Largest Contentful Paint (LCP) 및 Interaction to Next Paint와 같은 웹 바이탈 지표로 측정됩니다.

업데이트 성능: 사용자 입력에 반응하여 애플리케이션이 얼마나 빠르게 업데이트되는지. 예를 들어, 사용자가 검색창에 입력할 때 리스트가 얼마나 빠르게 업데이트되는지, 또는 SPA(싱글 페이지 애플리케이션)에서 사용자가 내비게이션 링크를 클릭할 때 페이지가 얼마나 빠르게 전환되는지 등이 있습니다.

이 두 가지를 모두 극대화하는 것이 이상적이지만, 프론트엔드 아키텍처에 따라 각 측면의 성능을 달성하는 난이도가 달라집니다. 또한, 개발하는 애플리케이션의 유형에 따라 성능에서 우선시해야 할 부분이 크게 달라집니다. 따라서 최적의 성능을 보장하는 첫 번째 단계는 개발하려는 애플리케이션 유형에 맞는 올바른 아키텍처를 선택하는 것입니다:

Vue 사용 방법을 참고하여 다양한 방식으로 Vue를 활용할 수 있는 방법을 확인하세요.

Jason Miller가 Application Holotypes에서 웹 애플리케이션의 유형과 각각의 이상적인 구현/전달 방식에 대해 논의합니다.

프로파일링 옵션

많이 있습니다:

프로덕션 배포의 로드 성능 프로파일링을 위한 도구:

- PageSpeed Insights
- WebPageTest

로컬 개발 중 성능 프로파일링을 위한 도구:

- Chrome DevTools Performance Panel
  `app.config.performance` 를 사용하면 Chrome DevTools의 성능 타임라인에서 Vue 전용 성능 마커를 활성화할 수 있습니다.

- Vue DevTools 확장도 성능 프로파일링 기능을 제공합니다.

페이지 로드 최적화

페이지 로드 성능을 최적화하는 프레임워크에 구애받지 않는 다양한 방법이 있습니다. 이 web.dev 가이드에서 종합적인 내용을 확인할 수 있습니다. 여기서는 Vue에 특화된 기법에 주로 초점을 맞추겠습니다.

올바른 아키텍처 선택하기

페이지 로드 성능에 민감한 사용 사례라면, 순수 클라이언트 사이드 SPA로 배포하는 것을 피하세요. 서버가 사용자가 보고자 하는 콘텐츠가 포함된 HTML을 직접 전송하도록 해야 합니다. 순수 클라이언트 렌더링은 콘텐츠 표시까지 시간이 느려지는 문제가 있습니다. 이는 서버 사이드 렌더링(SSR) 또는 정적 사이트 생성(SSG)으로 완화할 수 있습니다. Vue로 SSR을 수행하는 방법은 SSR 가이드를 참고하세요. 앱에 풍부한 상호작용 요구사항이 없다면, 전통적인 백엔드 서버에서 HTML을 렌더링하고 클라이언트에서 Vue로 향상시키는 방법도 사용할 수 있습니다.

메인 애플리케이션이 반드시 SPA여야 하지만, 마케팅 페이지(랜딩, 소개, 블로그 등)가 있다면 별도로 배포하세요! 마케팅 페이지는 SSG를 사용해 최소한의 JS로 정적 HTML로 배포하는 것이 이상적입니다.

번들 크기와 트리 셰이킹

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

- 가능하다면 빌드 단계를 사용하세요.

들어, 내장 `<Transition>` 컴포넌트를 사용하지 않으면 최종 프로덕션 번들에 포함되지 않습니다. 트리 셰이킹은 소스 코드에서 사용하지 않는 다른 모듈도 제거할 수 있습니다.

빌드 단계를 사용할 때 템플릿이 미리 컴파일되므로 Vue 컴파일러를 브라우저로 전송할 필요가 없습니다. 이렇게 하면 **14kb**(min+gzipped)의 JavaScript를 절약하고 런타임 컴파일 비용도 피할 수 있습니다.

새로운 의존성을 도입할 때 크기에 주의하세요! 실제 애플리케이션에서 번들이 비대해지는 가장 흔한 원인은 무거운 의존성을 인지하지 못한 채 도입하는 경우입니다.

빌드 단계를 사용하는 경우, ES 모듈 포맷을 제공하고 트리 셰이킹에 친화적인 의존성을 선호하세요. 예를 들어, `lodash` 대신 `lodash-es` 를 사용하세요.

의존성의 크기를 확인하고 제공하는 기능에 비해 가치가 있는지 평가하세요. 의존성이 트리 셰이킹에 친화적이라면, 실제 크기 증가는 실제로 import하는 API에 따라 달라집니다. bundlejs.com과 같은 도구로 빠르게 확인할 수 있지만, 실제 빌드 환경에서 측정하는 것이 항상 가장 정확합니다.

Vue를 주로 점진적 향상을 위해 사용하고 빌드 단계를 피하고 싶다면, petite-vue(단 **6kb**)를 사용하는 것도 고려해보세요.

코드 분할

코드 분할은 빌드 도구가 애플리케이션 번들을 여러 개의 더 작은 청크로 분할하여, 필요할 때마다 또는 병렬로 로드할 수 있도록 하는 것입니다. 적절한 코드 분할을 통해 페이지 로드 시 필요한 기능만 즉시 다운로드하고, 추가 청크는 필요할 때만 지연 로드하여 성능을 향상시킬 수 있습니다.

Rollup(Vite의 기반)이나 webpack과 같은 번들러는 ESM 동적 import 문법을 감지하여 자동으로 분할 청크를 생성할 수 있습니다:

```js
// lazy.js와 그 의존성은 별도의 청크로 분할되어
// `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 결합하여 컴포넌트 트리에 대한 분할 청크를 만들 수 있습니다:

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

// Foo.vue와 그 의존성에 대해 별도의 청크가 생성됩니다.
```

```
// 표정입니다.
const Foo = defineAsyncComponent(() => import('./Foo.vue'))
```

Vue Router를 사용하는 애플리케이션의 경우, 라우트 컴포넌트에 대해 지연 로딩을 사용하는 것이 강력히 권장됩니다. Vue Router는 `defineAsyncComponent` 와 별도로 지연 로딩을 명시적으로 지원합니다. 자세한 내용은 라우트 지연 로딩을 참고하세요.

업데이트 최적화

Props 안정성

Vue에서 자식 컴포넌트는 전달받은 props 중 하나라도 변경될 때만 업데이트됩니다. 다음 예시를 살펴보세요:

template

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

`<ListItem>` 컴포넌트 내부에서는 `id` 와 `activeId` props를 사용해 현재 활성화된 항목인지 판단합니다. 이 방식은 동작하지만, 문제는 `activeId` 가 변경될 때마다 리스트의 **모든** `<ListItem>` 이 업데이트되어야 한다는 점입니다!

이상적으로는 활성 상태가 변경된 항목만 업데이트되어야 합니다. 이를 위해 활성 상태 계산을 부모로 옮기고, `<ListItem>` 이 `active` prop을 직접 받도록 할 수 있습니다:

template

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

이제 대부분의 컴포넌트는 `active` prop이 `activeId` 가 변경되어도 동일하게 유지되므로 더 이상 업데이트할 필요가 없습니다. 일반적으로, 자식 컴포넌트에 전달하는 props를 최대한 안정적으로 유지하는 것이 좋습니다.

v-once

`v-once` 는 런타임 데이터에 의존하지만 더 이상 업데이트할 필요가 없는 콘텐츠를 렌더링할 때 사용할 수 있는 내장 디렉티브입니다. 사용된 전체 서브 트리는 이후 모든 업데이트에서 건너뜁

v-memo

`v-memo` 는 대형 서브 트리나 `v-for` 리스트의 업데이트를 조건부로 건너뛸 수 있는 내장 디렉티브입니다. 자세한 내용은 API 레퍼런스를 참고하세요.

계산 속성 안정성

Vue 3.4 이상에서는 계산 속성의 계산된 값이 이전 값과 달라졌을 때만 효과가 트리거됩니다. 예를 들어, 아래의 `isEven` 계산 속성은 반환 값이 `true` 에서 `false` 로, 또는 그 반대로 변경될 때만 효과를 트리거합니다:

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

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

// 계산된 값이 계속 `true`이므로 새로운 로그가 트리거되지 않음
count.value = 2
count.value = 4
```

이렇게 하면 불필요한 효과 트리거가 줄어들지만, 계산 속성이 매번 새로운 객체를 생성하는 경우에는 동작하지 않습니다:

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

매번 새로운 객체가 생성되기 때문에, 기술적으로는 항상 새로운 값이 이전 값과 다릅니다. `isEven` 속성이 동일하더라도, Vue는 이전 값과 새 값을 깊이 비교하지 않는 한 알 수 없습니다. 이러한 비교는 비용이 많이 들 수 있으므로 권장되지 않습니다.

대신, 새 값과 이전 값을 수동으로 비교하고, 변경 사항이 없다고 판단되면 이전 값을 반환하여 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
  })
```

플레이그라운드에서 직접 시도해보세요

항상 전체 계산을 수행한 후 이전 값과 비교 및 반환해야 하며, 그래야 매번 동일한 의존성이 수집됩니다.

일반 최적화

> 다음 팁들은 페이지 로드와 업데이트 성능 모두에 영향을 미칩니다.

대형 리스트 가상화

모든 프론트엔드 애플리케이션에서 가장 흔한 성능 문제 중 하나는 대형 리스트 렌더링입니다. 프레임워크가 아무리 성능이 좋아도, 수천 개의 항목이 있는 리스트를 렌더링하면 브라우저가 처리해야 하는 DOM 노드의 수가 많아져 **느려질 수밖에 없습니다**.

하지만 모든 노드를 미리 렌더링할 필요는 없습니다. 대부분의 경우, 사용자의 화면에는 대형 리스트 중 일부만 표시됩니다. **리스트 가상화**는 대형 리스트에서 현재 뷰포트에 있거나 가까운 항목만 렌더링하는 기법으로, 성능을 크게 향상시킬 수 있습니다.

리스트 가상화 구현은 쉽지 않지만, 다행히 바로 사용할 수 있는 커뮤니티 라이브러리가 있습니다:

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

대형 불변 구조의 반응성 오버헤드 줄이기

Vue의 반응성 시스템은 기본적으로 깊게 동작합니다. 이는 상태 관리를 직관적으로 만들어주지만, 데이터 크기가 클 때는 오버헤드가 발생할 수 있습니다. 모든 프로퍼티 접근이 프록시 트랩을 트리거하여 의존성 추적을 수행하기 때문입니다. 이는 특히 깊게 중첩된 객체의 대형 배열을 다루고, 한 번의 렌더링에서 10만 개 이상의 프로퍼티에 접근해야 하는 경우에 두드러집니다. 따라서 매우 특정한 사용 사례에만 영향을 미칩니다.

Vue는 `shallowRef()` 와 `shallowReactive()` 를 사용해 깊은 반응성을 비활성화할 수 있는 탈출구를 제공합니다. Shallow API는 루트 레벨에서만 반응성을 가지며, 모든 중첩 객체는 그대로 노

트 상태를 교체해야만 업데이트가 트리거된다는 트레이드오프가 있습니다:

```js
const shallowArray = shallowRef([
  /* 깊은 객체가 많은 대형 리스트 */
])

// 이것은 업데이트를 트리거하지 않음...
shallowArray.value.push(newObject)
// 이것은 트리거함:
shallowArray.value = [...shallowArray.value, newObject]

// 이것은 업데이트를 트리거하지 않음...
shallowArray.value[0].foo = 1
// 이것은 트리거함:
shallowArray.value = [
  {
    ...shallowArray.value[0],
    foo: 1
  },
  ...shallowArray.value.slice(1)
]
```

불필요한 컴포넌트 추상화 피하기

더 나은 추상화나 코드 구조화를 위해 렌더리스 컴포넌트나 고차 컴포넌트(즉, 다른 컴포넌트에 추가 props를 전달하여 렌더링하는 컴포넌트)를 만들 때가 있습니다. 이런 방식 자체는 문제가 없지만, 컴포넌트 인스턴스는 일반 DOM 노드보다 훨씬 비용이 크므로, 추상화 패턴으로 인해 너무 많은 인스턴스를 만들면 성능 비용이 발생할 수 있습니다.

몇 개의 인스턴스만 줄인다고 해서 눈에 띄는 효과가 있는 것은 아니므로, 앱에서 컴포넌트가 몇 번만 렌더링된다면 크게 신경 쓰지 않아도 됩니다. 이 최적화를 고려해야 할 가장 좋은 시나리오는 역시 대형 리스트입니다. 예를 들어, 100개의 항목이 있는 리스트에서 각 항목 컴포넌트가 많은 자식 컴포넌트를 포함하고 있다면, 불필요한 컴포넌트 추상화를 하나만 제거해도 수백 개의 컴포넌트 인스턴스가 줄어들 수 있습니다.

Previous
프로덕션 배포

Next
접근성