Suspense

> ⚠ 실험적 기능
>
> `<Suspense>` 는 실험적 기능입니다. 안정적인 상태에 도달할 것이라는 보장이 없으며, 그 전에 API가 변경될 수 있습니다.

`<Suspense>` 는 컴포넌트 트리에서 비동기 의존성을 조율하기 위한 내장 컴포넌트입니다. 컴포넌트 트리 아래에 중첩된 여러 비동기 의존성이 해결될 때까지 로딩 상태를 렌더링할 수 있습니다.

비동기 의존성

`<Suspense>` 가 해결하려는 문제와 이 비동기 의존성과 어떻게 상호작용하는지 설명하기 위해, 다음과 같은 컴포넌트 계층 구조를 상상해봅시다:

```
<Suspense>
└ <Dashboard>
   ├ <Profile>
   │  └ <FriendStatus> (비동기 setup()을 가진 컴포넌트)
   └ <Content>
      ├ <ActivityFeed> (비동기 컴포넌트)
      └ <Stats> (비동기 컴포넌트)
```

컴포넌트 트리에는 렌더링이 먼저 해결되어야 하는 비동기 리소스에 의존하는 여러 중첩 컴포넌트가 있습니다. `<Suspense>` 없이 각 컴포넌트는 자체적으로 로딩/에러 및 로드 완료 상태를 처리해야 합니다. 최악의 경우, 페이지에 세 개의 로딩 스피너가 표시되고, 콘텐츠가 서로 다른 시점에 표시될 수 있습니다.

`<Suspense>` 컴포넌트를 사용하면 이러한 중첩된 비동기 의존성이 해결될 때까지 상위 수준의 로딩/에러 상태를 표시할 수 있습니다.

1. 비동기 `setup()` 훅을 가진 컴포넌트. 여기에는 최상위 `await` 표현식을 사용하는 `<script setup>` 컴포넌트도 포함됩니다.
2. 비동기 컴포넌트.

`async setup()`

Composition API 컴포넌트의 `setup()` 훅은 비동기로 만들 수 있습니다:

```js
export default {
  async setup() {
    const res = await fetch(...)
    const posts = await res.json()
    return {
      posts
    }
  }
}
```

`<script setup>` 을 사용하는 경우, 최상위 `await` 표현식이 있으면 해당 컴포넌트는 자동으로 비동기 의존성이 됩니다:

```vue
<script setup>
const res = await fetch(...)
const posts = await res.json()
</script>

<template>
  {{ posts }}
</template>
```

비동기 컴포넌트

비동기 컴포넌트는 기본적으로 **"suspensible"** 합니다. 즉, 부모 체인에 `<Suspense>` 가 있으면 해당 `<Suspense>` 의 비동기 의존성으로 처리됩니다. 이 경우, 로딩 상태는 `<Suspense>` 가 제어하며, 컴포넌트 자체의 로딩, 에러, 지연 및 타임아웃 옵션은 무시됩니다.

비동기 컴포넌트는 옵션에서 `suspensible: false` 를 지정하여 `Suspense` 제어를 비활성화하고 항상 자체적으로 로딩 상태를 제어할 수 있습니다.

로딩 상태

`<Suspense>` 컴포넌트에는 두 개의 슬롯: `#default` 와 `#fallback` 이 있습니다. 두 슬롯 모두 **하나의** 즉시 자식 노드만 허용합니다. 기본 슬롯의 노드는 가능하다면 표시됩니다. 그렇지 않으면 fallback 슬롯의 노드가 대신 표시됩니다.

template

```
<Suspense>
  <!-- 중첩된 비동기 의존성을 가진 컴포넌트 -->
  <Dashboard />

  <!-- #fallback 슬롯을 통한 로딩 상태 -->
  <template #fallback>
    로딩 중...
  </template>
</Suspense>
```

초기 렌더링 시, `<Suspense>` 는 기본 슬롯 콘텐츠를 메모리에서 렌더링합니다. 이 과정에서 비동기 의존성이 발견되면 **대기(pending)** 상태로 진입합니다. 대기 상태에서는 fallback 콘텐츠가 표시됩니다. 모든 비동기 의존성이 해결되면 `<Suspense>` 는 **해결(resolved)** 상태로 진입하고, 해결된 기본 슬롯 콘텐츠가 표시됩니다.

초기 렌더링 중 비동기 의존성이 발견되지 않으면 `<Suspense>` 는 바로 해결 상태로 진입합니다.

한 번 해결 상태에 들어가면, `<Suspense>` 는 `#default` 슬롯의 루트 노드가 교체될 때만 다시 대기 상태로 돌아갑니다. 트리에서 더 깊이 중첩된 새로운 비동기 의존성은 `<Suspense>` 가 다시 대기 상태로 돌아가게 하지 **않습니다**.

되돌림이 발생하면, fallback 콘텐츠가 즉시 표시되지 않습니다. 대신, `<Suspense>` 는 새 콘텐츠와 그 비동기 의존성이 해결될 때까지 이전 `#default` 콘텐츠를 표시합니다. 이 동작은 `timeout` prop으로 설정할 수 있습니다: 새 기본 콘텐츠 렌더링에 `timeout` 보다 오래 걸리면 `<Suspense>` 는 fallback 콘텐츠로 전환합니다. `timeout` 값이 `0` 이면 기본 콘텐츠가 교체될 때 fallback 콘텐츠가 즉시 표시됩니다.

이벤트

`<Suspense>` 컴포넌트는 3개의 이벤트를 발생시킵니다: `pending` , `resolve` , `fallback` . `pending` 이벤트는 대기 상태로 진입할 때 발생합니다. `resolve` 이벤트는 `default` 슬롯의 새 콘텐츠가 해결되었을 때 발생합니다. `fallback` 이벤트는 fallback 슬롯의 내용이 표시될 때 발생합니다.

하는 데 사용할 수 있습니다.

에러 처리

`<Suspense>` 는 현재 컴포넌트 자체를 통한 에러 처리를 제공하지 않습니다. 하지만, `errorCaptured` 옵션이나 `onErrorCaptured()` 훅을 사용하여 `<Suspense>` 의 부모 컴포넌트에서 비동기 에러를 포착하고 처리할 수 있습니다.

다른 컴포넌트와의 조합

`<Transition>` 및 `<KeepAlive>` 컴포넌트와 `<Suspense>` 를 함께 사용하는 경우가 많습니다. 이 컴포넌트들의 중첩 순서는 모두 올바르게 동작하도록 하는 데 중요합니다.

또한, 이 컴포넌트들은 Vue Router의 `<RouterView>` 컴포넌트와 함께 자주 사용됩니다.

다음 예시는 이 컴포넌트들을 중첩하여 모두 기대한 대로 동작하도록 하는 방법을 보여줍니다. 더 간단한 조합이 필요하다면 필요 없는 컴포넌트는 제거할 수 있습니다:

template

```
<RouterView v-slot="{ Component }">
  <template v-if="Component">
    <Transition mode="out-in">
      <KeepAlive>
        <Suspense>
          <!-- 메인 콘텐츠 -->
          <component :is="Component"></component>

          <!-- 로딩 상태 -->
          <template #fallback>
            로딩 중...
          </template>
        </Suspense>
      </KeepAlive>
    </Transition>
  </template>
</RouterView>
```

Vue Router는 동적 import를 사용하여 컴포넌트 지연 로딩을 기본적으로 지원합니다. 이는 비동기 컴포넌트와는 다르며, 현재로서는 `<Suspense>` 를 트리거하지 않습니다. 하지만, 이들 컴포넌트가 자식으로 비동기 컴포넌트를 가질 수 있고, 이 경우에는 평소와 같이 `<Suspense>` 를 트리거할 수 있습니다.

중첩 Suspense

3.3+에서만 지원

다음과 같이 여러 비동기 컴포넌트(중첩 또는 레이아웃 기반 라우트에서 흔함)가 있을 때:

template

```
<Suspense>
  <component :is="DynamicAsyncOuter">
    <component :is="DynamicAsyncInner" />
  </component>
</Suspense>
```

`<Suspense>` 는 예상대로 트리 아래의 모든 비동기 컴포넌트를 해결하는 경계를 만듭니다. 하지만, `DynamicAsyncOuter` 를 변경하면 `<Suspense>` 가 올바르게 대기하지만, `DynamicAsyncInner` 를 변경하면 중첩된 `DynamicAsyncInner` 가 해결될 때까지 빈 노드를 렌더링합니다(이전 노드나 fallback 슬롯 대신).

이를 해결하기 위해, 중첩된 컴포넌트의 패치를 처리할 중첩 suspense를 둘 수 있습니다. 예를 들면:

template

```
<Suspense>
  <component :is="DynamicAsyncOuter">
    <Suspense suspensible> <!-- 이 부분 -->
      <component :is="DynamicAsyncInner" />
    </Suspense>
  </component>
</Suspense>
```

`suspensible` prop을 설정하지 않으면, 내부 `<Suspense>` 는 부모 `<Suspense>` 에 의해 동기 컴포넌트로 처리됩니다. 즉, 자체 fallback 슬롯이 있으며, 두 `Dynamic` 컴포넌트가 동시에 변경되면 자식 `<Suspense>` 가 자체 의존성 트리를 로딩하는 동안 빈 노드와 여러 패치 사이클이 발생할 수 있습니다. 이는 바람직하지 않을 수 있습니다. 설정하면, 모든 비동기 의존성 처리는 부모 `<Suspense>` 에 위임되고(이벤트 발생 포함), 내부 `<Suspense>` 는 의존성 해결 및 패칭을 위한 또 다른 경계 역할만 하게 됩니다.

관련 문서

`<Suspense>` API 레퍼런스

Previous
Teleport

Next
싱글 파일 컴포넌트