렌더 함수와 JSX

Vue는 대부분의 경우에 템플릿을 사용하여 애플리케이션을 구축하는 것을 권장합니다. 그러나 JavaScript의 프로그래밍적인 기능이 필요한 경우가 있습니다. 이때 사용할 수 있는 것이 **렌더 함수**입니다.

가상 DOM 및 렌더 함수 개념이 처음이라면 먼저 렌더링 메커니즘 챕터를 읽어보세요.

기본 사용법

Vnode 생성하기

Vue는 Vnode를 생성하기 위해 `h()` 함수를 제공합니다.

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

const vnode = h(
  'div', // 타입
  { id: 'foo', class: 'bar' }, // 속성
  [
    /* 자식 요소들 */
  ]
)
```

`h()` 는 **하이퍼스크립트**(hyperscript)의 약자입니다. 이는 "HTML(하이퍼텍스트 마크업 언어)를 생성하는 JavaScript"를 의미합니다. 이 이름은 많은 가상 DOM 구현에서 공유하는 규칙으로부터 유래되었습니다. 더 구체적인 이름은 `createVnode()` 가 될 수 있지만, 더 짧은 이름은 렌더 함수에서 이 함수를 여러 번 호출해야 할 때 유용합니다.

`h()` 함수는 매우 유연하게 설계되었습니다:

```js
// 타입을 제외한 모든 인자는 선택적입니다.
h('div')
h('div', { id: 'foo' })

// 속성으로 속성과 프로퍼티를 모두 사용할 수 있습니다.
// Vue는 할당하는 올바른 방법을 자동으로 선택합니다.
h('div', { class: 'bar', innerHTML: 'hello' })

// `.prop`과 `.attr`와 같은 속성 변경자를 사용할 수 있습니다.
// 접두사인 `.`와 `^`를 사용하여 추가합니다.
h('div', { '.name': 'some-name', '^width': '100' })

// class와 style은 템플릿과 동일한
// 객체/배열 값을 지원합니다.
h('div', { class: [foo, { bar }], style: { color: 'red' } })

// 이벤트 리스너는 onXxx로 전달해야 합니다.
h('div', { onClick: () => {} })

// 자식 요소로 문자열을 사용할 수 있습니다.
h('div', { id: 'foo' }, 'hello')

// 속성이 없는 경우 속성을 생략할 수 있습니다.
h('div', 'hello')
h('div', [h('span', 'hello')])
```

```
// 자식 요소 배열에는 혼합된 Vnode와 문자열이 포함될 수 있습니다.
h('div', ['hello', h('span', 'hello')])
```

결과로 나오는 Vnode의 구조는 다음과 같습니다.

```js
const vnode = h('div', { id: 'foo' }, [])

vnode.type // 'div'
vnode.props // { id: 'foo' }
vnode.children // []
vnode.key // null
```

> 주의
>
> 전체 `VNode` 인터페이스에는 여러 가지 다른 내부 속성이 있지만, 여기에서 나열된 속성 이외의 속성에 의존하는 것을 피하는 것이 권장됩니다. 내부 속성이 변경되면 의도하지 않은 오류가 발생할 수 있습니다.

렌더 함수 선언하기

구성 API를 사용할 때 `setup()` 훅의 반환 값은 템플릿과 데이터를 연결하기 위해 사용됩니다. 그러나 렌더 함수를 사용하는 경우에는 직접 렌더 함수를 반환할 수 있습니다.

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

export default {
  props: {
    /* ... */
  },
  setup(props) {
    const count = ref(1)

    // 렌더 함수를 반환합니다.
    return () => h('div', props.msg + count.value)
  }
}
```

렌더 함수는 `setup()` 내부에서 선언되므로 동일한 스코프에서 선언된 속성 및 반응성 있는 상태에 접근할 수 있습니다.

단일 Vnode를 반환하는 것 외에도 문자열이나 배열을 반환할 수도 있습니다.

```js
export default {
  setup() {
    return () => 'hello world!'
  }
}
```

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

export default {
  setup() {
    // 여러 개의 루트 노드를 반환하기 위해 배열을 사용합니다.
    return () => [
      h('div'),
      h('div'),
      h('div')
    ]
```

```
  }
}
```

TIP

값을 직접 반환하는 대신에 함수를 반환해야 합니다! `setup()` 함수는 컴포넌트당 한 번 호출되지만, 반환된 렌더 함수는 여러 번 호출될 수 있습니다.

상태가 없는 렌더 함수 컴포넌트의 경우, 간결성을 위해 직접 함수로 선언할 수도 있습니다.

```js
function Hello() {
  return 'hello world!'
}
```

맞습니다. 이것은 유효한 Vue 컴포넌트입니다! 이 구문에 대한 자세한 내용은 함수형 컴포넌트를 참조하세요.

Vnode는 고유해야 합니다

컴포넌트 트리의 모든 Vnode는 고유해야 합니다. 다음과 같은 렌더 함수는 잘못된 예입니다.

```js
function render() {
  const p = h('p', 'hi')
  return h('div', [
    // 중복된 Vnode입니다!
    p,
    p
  ])
}
```

동일한 요소/컴포넌트를 여러 번 복제하려는 경우 팩토리 함수를 사용할 수 있습니다. 예를 들어, 다음과 같은 렌더 함수는 동일한 단락(p)을 20개 복제하는 데에 완벽히 유효합니다.

```js
function render() {
  return h(
    'div',
    Array.from({ length: 20 }).map(() => {
      return h('p', 'hi')
    })
  )
}
```

JSX / TSX

JSX는 JavaScript에 XML과 유사한 확장을 제공하여 다음과 같은 코드를 작성할 수 있도록 합니다.

```jsx
const vnode = <div>hello</div>
```

JSX 표현식 내에서 동적인 값을 포함하려면 중괄호를 사용합니다.

```jsx
const vnode = <div id={dynamicId}>hello, {userName}</div>
```

create-vue 및 Vue CLI는 미리 구성된 JSX 지원으로 프로젝트를 스캐폴딩할 수 있는 옵션을 제공합니다. JSX를 수동으로 구성하는 경우 **@vue/babel-plugin-jsx** 의 문서를 참조하시기 바랍니다.

실제로 JSX는 처음에 React에서 소개되었지만, JSX는 런타임에서 정의된 구체적인 의미론을 가지고 있지 않으며 다양한 출력으로 컴파일될 수 있습니다. JSX를 사용한 경험이 있다면 **Vue의 JSX 변환은 React의 JSX 변환과 다릅니다**. 따라서 Vue 애플리케이션에서 React의 JSX 변환을 사용할 수 없습니다. React JSX와의 주요 차이점은 다음과 같습니다.

- class 및 for 와 같은 HTML 속성을 속성으로 사용할 수 있습니다. className 이나 htmlFor 를 사용할 필요가 없습니다.
- 컴포넌트에 자식 요소(즉, 슬롯)를 전달하는 방식이 다릅니다.

Vue의 타입 정의는 TSX 사용에 대한 타입 추론을 제공합니다. TSX를 사용하는 경우, Vue JSX 변환을 처리하기 위해 TypeScript가 JSX 구문을 그대로 남겨두도록 tsconfig.json 에 "jsx": "preserve" 를 지정해야 합니다.

JSX 타입 추론

Vue의 JSX도 변환과 마찬가지로 다른 타입 정의가 필요합니다.

Vue 3.4부터 Vue는 더 이상 전역 JSX 네임스페이스를 암시적으로 등록하지 않습니다. TypeScript에 Vue의 JSX 타입 정의를 사용하도록 지시하려면 tsconfig.json 에 다음을 포함시켜야 합니다:

```json
{
  "compilerOptions": {
    "jsx": "preserve",
    "jsxImportSource": "vue"
    // ...
  }
}
```

파일 상단에 /* @jsxImportSource vue */ 주석을 추가하여 파일별로 선택적으로 사용할 수도 있습니다.

전역 JSX 네임스페이스의 존재에 의존하는 코드가 있다면, 프로젝트에서 vue/jsx 를 명시적으로 가져오거나 참조하여 3.4 이전의 정확한 전역 동작을 유지할 수 있습니다. 이는 전역 JSX 네임스페이스를 등록합니다.

렌더 함수 레시피

다음에서는 몇 가지 일반적인 템플릿 기능을 렌더 함수 / JSX의 동등한 형태로 구현하는 몇 가지 레시피를 제공합니다.

v-if

템플릿:

```template
<div>
  <div v-if="ok">yes</div>
  <span v-else>no</span>
</div>
```

동등한 렌더 함수 / JSX:

```js
h('div', [ok.value ? h('div', 'yes') : h('span', 'no')])
```

```jsx
<div>{ok.value ? <div>yes</div> : <span>no</span>}</div>
```

v-for

템플릿:

```template
<ul>
  <li v-for="{ id, text } in items" :key="id">
    {{ text }}
  </li>
</ul>
```

동등한 렌더 함수 / JSX:

```js
h(
  'ul',
  // `items`가 배열 값을 갖는 ref인 경우를 가정합니다.
  items.value.map(({ id, text }) => {
    return h('li', { key: id }, text)
  })
)
```

```jsx
<ul>
  {items.value.map(({ id, text }) => {
    return <li key={id}>{text}</li>
  })}
</ul>
```

v-on

on 다음에 대문자로 시작하는 속성 이름은 이벤트 리스너로 처리됩니다. 예를 들어 onClick 은 템플릿의 @click 과 동일합니다.

```js
h(
  'button',
  {
    onClick(event) {
      /* ... */
    }
  },
  'Click Me'
)
```

```jsx
<button
  onClick={(event) => {
    /* ... */
  }}
>
  Click Me
</button>
```

이벤트 수정자

.passive , .capture , .once 이벤트 수정자의 경우에는 이벤트 이름 뒤에 camelCase로 연결하여 추가할 수 있습니다.

예를 들어:

```js
h('input', {
  onClickCapture() {
    /* 캡처 모드에서 리스너 */
  },
  onKeyupOnce() {
    /* 한 번만 트리거 */
  },
```

```
  onMouseoverOnceCapture() {
    /* 한 번만 트리거 + 캡처 모드 */
  }
})
```

jsx

```
<input
  onClickCapture={() => {}}
  onKeyupOnce={() => {}}
  onMouseoverOnceCapture={() => {}}
/>
```

다른 이벤트와 키 수정자의 경우 **withModifiers** 헬퍼를 사용할 수 있습니다.

js

```
import { withModifiers } from 'vue'

h('div', {
  onClick: withModifiers(() => {}, ['self'])
})
```

jsx

```
<div onClick={withModifiers(() => {}, ['self'])} />
```

컴포넌트

컴포넌트의 vnode를 생성하려면 h() 의 첫 번째 인수에 컴포넌트 정의를 전달해야 합니다. 따라서 렌더 함수에서 컴포넌트를 사용할 때는 컴포넌트를 직접 가져와서 사용하면 됩니다.

js

```
import Foo from './Foo.vue'
import Bar from './Bar.jsx'

function render() {
  return h('div', [h(Foo), h(Bar)])
}
```

jsx

```
function render() {
  return (
    <div>
      <Foo />
      <Bar />
    </div>
  )
}
```

h 를 사용하여 다양한 파일 형식에서 가져온 컴포넌트를 사용할 수 있습니다.

동적 컴포넌트는 렌더 함수에서 간단하게 처리할 수 있습니다.

js

```
import Foo from './Foo.vue'
import Bar from './Bar.jsx'

function render() {
  return ok.value ? h(Foo) : h(Bar)
}
```

jsx

```
function render() {
  return ok.value ? <Foo /> : <Bar />
}
```

컴포넌트를 이름으로 등록하고 직접 가져올 수 없는 경우(예: 라이브러리에서 전역으로 등록되는 경우), **resolveComponent()** 헬퍼를 사용하여 프로그래밍적으로 해결할 수 있습니다.

슬롯 렌더링

렌더 함수에서 슬롯에 접근하려면 setup() 컨텍스트에서 슬롯에 액세스할 수 있는 slots 객체를 사용해야 합니다. slots 객체의 각 슬롯은 **Vnode 배열을 반환하는 함수**입니다.

```js
export default {
  props: ['message'],
  setup(props, { slots }) {
    return () => [
      // 기본 슬롯:
      // <div><slot /></div>
      h('div', slots.default()),

      // 네임드 슬롯:
      // <div><slot name="footer" :text="message" /></div>
      h(
        'div',
        slots.footer({
          text: props.message
        })
      )
    ]
  }
}
```

JSX 동등 코드:

```jsx
// default
<div>{slots.default()}</div>

// named
<div>{slots.footer({ text: props.message })}</div>
```

슬롯 전달하기

컴포넌트에 자식 요소를 전달하는 것은 요소에 자식 요소를 전달하는 것과 약간 다릅니다. 배열 대신 슬롯 함수나 슬롯 함수의 객체를 전달해야 합니다. 슬롯 함수는 일반적인 렌더 함수에서 반환할 수 있는 모든 것을 반환할 수 있습니다. 자식 컴포넌트에서 접근할 때 항상 VNode 배열로 정규화됩니다.

```js
// 단일 기본 슬롯
h(MyComponent, () => 'hello')

// 네임드 슬롯
// 슬롯 객체가 props로 처리되지 않도록
// `null`을 전달해야 함에 유의하세요.
h(MyComponent, null, {
  default: () => 'default slot',
  foo: () => h('div', 'foo'),
  bar: () => [h('span', 'one'), h('span', 'two')]
})
```

JSX 동등 코드:

```jsx
// default
<MyComponent>{() => 'hello'}</MyComponent>
```

```
// named
<MyComponent>{{
  default: () => 'default slot',
  foo: () => <div>foo</div>,
  bar: () => [<span>one</span>, <span>two</span>]
}}</MyComponent>
```

슬롯을 함수로 전달하면 자식 컴포넌트에서 지연 호출될 수 있습니다. 이를 통해 슬롯의 종속성은 부모가 아닌 자식에 의해 추적되며, 더 정확하고 효율적인 업데이트가 이루어집니다.

범위 지정 슬롯

부모 컴포넌트에서 범위 지정 슬롯을 렌더링하려면, 자식 컴포넌트에 슬롯이 전달됩니다. 이제 슬롯이 `text` 라는 매개변수를 가지고 있음에 주목하세요. 이 슬롯은 자식 컴포넌트에서 호출되며 자식 컴포넌트의 데이터가 부모 컴포넌트로 전달됩니다.

```js
// 부모 컴포넌트
export default {
  setup() {
    return () => h(MyComp, null, {
      default: ({ text }) => h('p', text)
    })
  }
}
```

슬롯이 속성(props)으로 취급되지 않도록 `null` 을 전달하는 것을 잊지 마세요.

```js
// 자식 컴포넌트
export default {
  setup(props, { slots }) {
    const text = ref('hi')
    return () => h('div', null, slots.default({ text: text.value }))
  }
}
```

JSX와 동등:

```jsx
<MyComponent>{{
  default: ({ text }) => <p>{ text }</p>
}}</MyComponent>
```

내장 컴포넌트

`<KeepAlive>` , `<Transition>` , `<TransitionGroup>` , `<Teleport>` , `<Suspense>` 과 같은 내장 컴포넌트는 렌더 함수에서 사용하기 위해 가져와야 합니다:

```js
import { h, KeepAlive, Teleport, Transition, TransitionGroup } from 'vue'

export default {
  setup () {
    return () => h(Transition, { mode: 'out-in' }, /* ... */)
  }
}
```

v-model

v-model 지시자는 템플릿 컴파일 중에 modelValue 와 onUpdate:modelValue 프롭으로 확장되며, 이 프롭을 직접 제공해야 합니다:

```js
export default {
  props: ['modelValue'],
  emits: ['update:modelValue'],
  setup(props, { emit }) {
    return () =>
      h(SomeComponent, {
        modelValue: props.modelValue,
        'onUpdate:modelValue': (value) => emit('update:modelValue', value)
      })
  }
}
```

사용자 정의 디렉티브

withDirectives 를 사용하여 VNode에 사용자 정의 디렉티브를 적용할 수 있습니다:

```js
import { h, withDirectives } from 'vue'

// 사용자 정의 디렉티브
const pin = {
  mounted() { /* ... */ },
  updated() { /* ... */ }
}

// <div v-pin:top.animate="200"></div>
const vnode = withDirectives(h('div'), [
  [pin, 200, 'top', { animate: true }]
])
```

디렉티브가 이름으로 등록되고 직접 가져올 수 없는 경우 **resolveDirective** 도우미를 사용하여 해결할 수 있습니다.

템플릿 레퍼런스

구성 API에서는 템플릿 레퍼런스를 ref() 자체를 vnode의 프롭으로 전달하여 생성합니다:

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

export default {
  setup() {
    const divEl = ref()

    // <div ref="divEl">
    return () => h('div', { ref: divEl })
  }
}
```

함수형 컴포넌트

함수형 컴포넌트는 자체 상태가 없는 컴포넌트의 대체 형식입니다. 이들은 순수 함수처럼 동작합니다. 속성을 받아서 VNode를 반환합니다. 컴포넌트 인스턴스(즉, this)를 생성하지 않고, 일반적인 컴포넌트 라이프사이클 훅도 사용하지 않습니다.

함수형 컴포넌트를 만들기 위해 옵션 객체 대신 일반 함수를 사용합니다. 이 함수는 사실상 컴포넌트의 render 함수 역할을 합니다.

함수형 컴포넌트의 시그니처는 setup() 훅과 동일합니다:

```js
function MyComponent(props, { slots, emit, attrs }) {
  // ...
}
```

대부분의 일반적인 컴포넌트 구성 옵션은 함수형 컴포넌트에서 사용할 수 없습니다. 그러나 **props** 와 **emits** 를 정의하여 **props** 와 **emits** 을 지정할 수 있습니다.

```js
MyComponent.props = ['value']
MyComponent.emits = ['click']
```

props 옵션이 지정되지 않으면 함수에 전달된 props 객체에 모든 속성이 포함되며, attrs 와 동일합니다. props 이름은 props 옵션이 지정되지 않으면 camelCase로 정규화되지 않습니다.

명시적인 props 를 가진 함수형 컴포넌트의 경우, 속성 전달은 일반 컴포넌트와 마찬가지로 작동합니다. 그러나 props 를 명시적으로 지정하지 않은 함수형 컴포넌트의 경우, attrs 에서 기본적으로 class , style , onXxx 이벤트 리스너만 상속됩니다. 어느 경우에도 inheritAttrs 를 false 로 설정하여 속성 상속을 비활성화할 수 있습니다.

```js
MyComponent.inheritAttrs = false
```

함수형 컴포넌트는 일반 컴포넌트와 마찬가지로 등록하고 사용할 수 있습니다. h() 에 첫 번째 인수로 함수를 전달하면 함수형 컴포넌트로 처리됩니다.

함수형 컴포넌트에 대한 타이핑

함수형 컴포넌트는 이름이 있는지 또는 익명인지에 따라 타입 지정될 수 있습니다. **Vue** - 공식 확장은 SFC 템플릿에서 사용할 때 적절히 타입이 지정된 함수형 컴포넌트의 타입 체킹도 지원합니다.

기명 함수형 컴포넌트

```tsx
import type { SetupContext } from 'vue'
type FComponentProps = {
  message: string
}

type Events = {
  sendMessage(message: string): void
}

function FComponent(
  props: FComponentProps,
  context: SetupContext<Events>
) {
  return (
    <button onClick={() => context.emit('sendMessage', props.message)}>
      {props.message} {' '}
    </button>
  )
}

FComponent.props = {
  message: {
    type: String,
    required: true
  }
}

FComponent.emits = {
  sendMessage: (value: unknown) => typeof value === 'string'
}
```

익명 함수형 컴포넌트

tsx

```tsx
import type { FunctionalComponent } from 'vue'

type FComponentProps = {
  message: string
}

type Events = {
  sendMessage(message: string): void
}

const FComponent: FunctionalComponent<FComponentProps, Events> = (
  props,
  context
) => {
  return (
    <button onClick={() => context.emit('sendMessage', props.message)}>
      {props.message} {' '}
    </button>
  )
}

FComponent.props = {
  message: {
    type: String,
    required: true
  }
}

FComponent.emits = {
  sendMessage: (value) => typeof value === 'string'
}
```