

REACT Docs - 漢語



Table of contents

• Accessibility	5
• Add react /to /a /website	17
• Addons animation	22
• Addons create /fragment	30
• Addons perf	32
• Addons pure /render /mixin	36
• Addons shallow /compare	37
• Addons shallow /renderer	38
• Addons test /utils	40
• Addons two /way /binding /helpers	47
• Addons update	50
• Addons	53
• Cdn links	55
• Code splitting	56
• Codebase overview	61
• Components and /props	65
• Composition vs /inheritance	70
• Conditional rendering	73
• Context	78
• Create a /new /react /app	84
• Cross origin /errors	87
• Design principles	89
• Error boundaries	96
• Error decoder	100
• Faq ajax	101
• Faq build	104
• Faq functions	105
• Faq internals	112
• Faq state	113
• Faq structure	116
• Faq styling	118
• Faq versioning	119
• Forms	122
• Forwarding refs	128

• Fragments	130
• Getting started	133
• Handling events	136
• Hello world	139
• Higher order /components	141
• Hooks custom	150
• Hooks effect	155
• Hooks faq	165
• Hooks intro	184
• Hooks overview	188
• Hooks reference	194
• Hooks rules	210
• Hooks state	213
• How to /contribute	219
• Implementation notes	224
• Integrating with /other /libraries	242
• Introducing jsx	251
• Jsx in /depth	255
• Legacy context	263
• Legacy event /pooling	268
• Lifting state /up	269
• Lists and /keys	276
• Optimizing performance	281
• Portals	290
• React without /es6	293
• React without /jsx	297
• Reconciliation	298
• Reference dom /elements	302
• Reference events	306
• Reference glossary	314
• Reference javascript /environment /requirements	318
• Reference profiler	319
• Reference pure /render /mixin	322
• Reference react /component	324
• Reference react /dom /client	338
• Reference react /dom /server	340

• Reference react /dom	345
• Reference react	350
• Reference test /renderer	358
• Refs and /the /dom	364
• Release channels	370
• Render props	373
• Rendering elements	379
• State and /lifecycle	381
• Static type /checking	389
• Strict mode	396
• Testing environments	401
• Testing recipes	403
• Testing	414
• Thinking in /react	415
• Typechecking with /proptypes	420
• Uncontrolled components	425
• Web components	427

無障礙功能

為什麼要有無障礙功能？

網路無障礙功能（又稱為 [a11y](#)）的概念是設計並打造所有人都能使用的網站。我們必須支援無障礙功能，才能使用輔助科技解讀網頁。

React 能完整支援無障礙網站的建構。這些網站通常都使用標準的 HTML 技術。

標準及規範

WCAG

[網路內容無障礙功能指南](#) (WCAG) 提供了建立無障礙網頁的規範。

以下的 WCAG 檢查清單提供了概觀：

- [Wuhcag 的 WCAG 檢查清單](#)
- [WebAIM 的 WCAG 檢查清單](#)
- [The A11Y Project 的檢查清單](#)

WAI-ARIA

這份[網路無障礙功能倡議 - 無障礙網路應用程式](#)文件包含了許多架設無障礙功能 JavaScript 的小工具。

請注意，所有的 `aria-*` HTML attribute 在 JSX 中都是支援的。相較於 React 中大部分駝峰式大小寫的 DOM property 和 attribute，這些 attribute 則應該像在純 HTML 中一樣使用帶連字符式寫法（又稱為 kebab-case、lisp-case 等）：

```
<input  
  type="text"  
  aria-label={labelText}  
  aria-required="true"  
  onChange={onchangeHandler}  
  value={inputValue}  
  name="name"  
/>
```

Semantic HTML

Semantic HTML 是無障礙網頁應用程式的基礎。使用不同的 HTML element 來加強網站中資訊的意義可以在不用花費的情況下讓所有人造訪你的網站。

- [MDN HTML element 參考](#)

有時候當我們新增 `<div>` element 到 JSX 讓 React 程式可以運作時，我們會違反 HTML 的語義，尤特別是在當我們處理列表 (``, `` 和 `<dl>`) 以及 HTML 表格 `<table>` 的時候。在這些情況下我們應該使用 [React Fragment](#) 將數個 element 組織在一起。

例如：

```
import React, { Fragment } from 'react';

function ListItem({ item }) {
  return (
    <Fragment>
      <dt>{item.term}</dt>
      <dd>{item.description}</dd>
    </Fragment>
  );
}

function Glossary(props) {
  return (
    <dl>
      {props.items.map(item => (
        <ListItem item={item} key={item.id} />
      ))}
    </dl>
  );
}
```

你可以使用 map 將一個 collection 中的每一個 item 與一個 fragment 相對應，就如同處理其他的 element 一樣：

```
function Glossary(props) {
  return (
    <dl>
      {props.items.map(item => (
        // 當你 map 一個 collection 時，Fragment 也應該要有一個 `key` prop
        <Fragment key={item.id}>
          <dt>{item.term}</dt>
          <dd>{item.description}</dd>
        </Fragment>
      ))}
    </dl>
  );
}
```

如果你的 Fragment tag 不需要任何 prop，你也可以使用[簡寫語法](#)，如果你的工具支援這個語法的話：

```
function ListItem({ item }) {
  return (
    <>
      <dt>{item.term}</dt>
      <dd>{item.description}</dd>
    </>
  );
}
```

請參考 [Fragment 文件](#) 了解更多詳情。

無障礙表格

標記

每一個 HTML 的表格控制，例如 `<input>` 和 `<textarea>`，都需要無障礙標記。我們需要提供敘述性的、能讓螢幕閱讀器識別的標記。

以下的資源讓我們知道如何標記：

- [W3C 解釋如何標記 element](#)
- [WebAIM 解釋如何標記 element](#)
- [Paciello Group 解釋無障礙名稱](#)

雖然這些標準的 HTML 用法能直接在 React 中使用，請注意 `for` attribute 在 JSX 中是寫作 `htmlFor`：

```
<label htmlFor="namedInput">Name:</label>
<input id="namedInput" type="text" name="name"/>
```

通知用戶錯誤訊息

錯誤發生的狀況需要被所有使用者了解。以下連結解釋如何讓螢幕閱讀器也能識別錯誤訊息：

- [W3C 解釋用戶通知](#)
- [WebAIM 解釋表格驗證](#)

焦點控制

確保你的網路應用程式能完全只用鍵盤操作：

- [WebAIM 談鍵盤無障礙功能](#)

鍵盤焦點和焦點輪廓

鍵盤焦點指的是目前在 DOM 中的 element 被選取以接受來自鍵盤的輸入。我們到處可以見到類似下面這張圖示內的焦點輪廓：



如果你打算用另一種方式做焦點輪廓，請使用 CSS 來移除這個輪廓，例如設定 `outline: 0`。

跳到指定內容的機制

請提供某種方式讓使用者可以跳過應用程式中的導覽部分，因為這樣可以協助加速鍵盤導覽。

跳過連結或跳過導覽連結是隱藏式的導覽連結，只有在鍵盤使用者與網頁互動時才會顯現。它們十分容易用內部頁面锚和一些 styling 做出來：

- [WebAIM - 跳過導覽連結](#)

你也可以使用像 `<main>` 和 `<aside>` 這樣的 landmark element 和 role 來標記頁面上的區域，因為輔助科技會快速導覽使用者到這些區域。

在這裡你可以閱讀更多關於這些 element 增加無障礙功能的用法：

- [無障礙的 Landmark](#)

Programmatically managing focus

Our React applications continuously modify the HTML DOM during runtime, sometimes leading to keyboard focus being lost or set to an unexpected element. In order to repair this, we need to programmatically nudge the keyboard focus in the right direction. For example, by resetting keyboard focus to a button that opened a modal window after that modal window is closed.

[MDN Web Docs](#) takes a look at this and describes how we can build [keyboard-navigable JavaScript widgets](#).

To set focus in React, we can use [Refs to DOM elements](#).

Using this, we first create a ref to an element in the JSX of a component class:

```
class CustomTextInput extends React.Component {
  constructor(props) {
    super(props);
    // Create a ref to store the textInput DOM element
    this.textInput = React.createRef();
  }
  render() {
    // Use the `ref` callback to store a reference to the text input DOM
    // element in an instance field (for example, this.textInput).
    return (
      <input
        type="text"
        ref={this.textInput}
      />
    );
  }
}
```

Then we can focus it elsewhere in our component when needed:

```
focus() {
  // Explicitly focus the text input using the raw DOM API
  // Note: we're accessing "current" to get the DOM node
  this.textInput.current.focus();
}
```

Sometimes a parent component needs to set focus to an element in a child component. We can do this by [exposing DOM refs to parent components](#) through a special prop on the child component that forwards the parent's ref to the child's DOM node.

```
function CustomTextInput(props) {
  return (
    <div>
```

```

        <input ref={props.inputRef} />
    </div>
);
}

class Parent extends React.Component {
constructor(props) {
    super(props);
    this.inputElement = React.createRef();
}
render() {
    return (
        <CustomTextInput inputRef={this.inputElement} />
    );
}
}

// Now you can set focus when required.
this.inputElement.current.focus();

```

When using a [HOC](#) to extend components, it is recommended to [forward the ref](#) to the wrapped component using the `forwardRef` function of React. If a third party HOC does not implement ref forwarding, the above pattern can still be used as a fallback.

A great focus management example is the [react-aria-modal](#). This is a relatively rare example of a fully accessible modal window. Not only does it set initial focus on the cancel button (preventing the keyboard user from accidentally activating the success action) and trap keyboard focus inside the modal, it also resets focus back to the element that initially triggered the modal.

Note:

While this is a very important accessibility feature, it is also a technique that should be used judiciously. Use it to repair the keyboard focus flow when it is disturbed, not to try and anticipate how users want to use applications.

Mouse and pointer events

Ensure that all functionality exposed through a mouse or pointer event can also be accessed using the keyboard alone. Depending only on the pointer device will lead to many cases where keyboard users cannot use your application.

To illustrate this, let's look at a prolific example of broken accessibility caused by click events. This is the outside click pattern, where a user can disable an opened popover by clicking outside the element.

Select an option

Load the option

Remove the option

This is typically implemented by attaching a `click` event to the `window` object that closes the popover:

```
class OuterClickExample extends React.Component {
  constructor(props) {
    super(props);

    this.state = { isOpen: false };
    this.toggleContainer = React.createRef();

    this.onClickHandler = this.onClickHandler.bind(this);
    this.onClickOutsideHandler = this.onClickOutsideHandler.bind(this);
  }

  componentDidMount() {
    window.addEventListener('click', this.onClickOutsideHandler);
  }

  componentWillUnmount() {
    window.removeEventListener('click', this.onClickOutsideHandler);
  }

  onClickHandler() {
    this.setState(currentState => ({
      isOpen: !currentState.isOpen
    }));
  }

  onClickOutsideHandler(event) {
    if (this.state.isOpen && !this.toggleContainer.current.contains(event.target))
    {
      this.setState({ isOpen: false });
    }
  }

  render() {
    return (
      <div ref={this.toggleContainer}>
        <button onClick={this.onClickHandler}>Select an option</button>
        {this.state.isOpen && (
          <ul>
            <li>Option 1</li>
            <li>Option 2</li>
            <li>Option 3</li>
          </ul>
        )}
      </div>
    );
  }
}
```

```

    );
}
}
```

This may work fine for users with pointer devices, such as a mouse, but operating this with the keyboard alone leads to broken functionality when tabbing to the next element as the `window` object never receives a `click` event. This can lead to obscured functionality which blocks users from using your application.

Select an option

Load the option

Remove the option

The same functionality can be achieved by using appropriate event handlers instead, such as `onBlur` and `onFocus`:

```

class BlurExample extends React.Component {
  constructor(props) {
    super(props);

    this.state = { isOpen: false };
    this.timeOutId = null;

    this.onClickHandler = this.onClickHandler.bind(this);
    this.onBlurHandler = this.onBlurHandler.bind(this);
    this.onFocusHandler = this.onFocusHandler.bind(this);
  }

  onClickHandler() {
    this.setState(currentState => ({
      isOpen: !currentState.isOpen
    }));
  }

  // We close the popover on the next tick by using setTimeout.
  // This is necessary because we need to first check if
  // another child of the element has received focus as
  // the blur event fires prior to the new focus event.
  onBlurHandler() {
    this.timeOutId = setTimeout(() => {
      this.setState({
        isOpen: false
      });
    });
  }

  // If a child receives focus, do not close the popover.
  onFocusHandler() {
    clearTimeout(this.timeOutId);
  }
}
```

```

render() {
  // React assists us by bubbling the blur and
  // focus events to the parent.
  return (
    <div onBlur={this.onBlurHandler}
      onFocus={this.onFocusHandler}>
      <button onClick={this.onClickHandler}
        aria-haspopup="true"
        aria-expanded={this.state.isOpen}>
        Select an option
      </button>
      {this.state.isOpen && (
        <ul>
          <li>Option 1</li>
          <li>Option 2</li>
          <li>Option 3</li>
        </ul>
      )}
    </div>
  );
}

```

This code exposes the functionality to both pointer device and keyboard users. Also note the added `aria-*` props to support screen-reader users. For simplicity's sake the keyboard events to enable `arrow key` interaction of the popover options have not been implemented.

Select an option

Load the option

Remove the option

This is one example of many cases where depending on only pointer and mouse events will break functionality for keyboard users. Always testing with the keyboard will immediately highlight the problem areas which can then be fixed by using keyboard aware event handlers.

More Complex Widgets

A more complex user experience should not mean a less accessible one. Whereas accessibility is most easily achieved by coding as close to HTML as possible, even the most complex widget can be coded accessibly.

Here we require knowledge of [ARIA Roles](#) as well as [ARIA States and Properties](#). These are toolboxes filled with HTML attributes that are fully supported in JSX and enable us to construct fully accessible, highly functional React components.

Each type of widget has a specific design pattern and is expected to function in a certain way by users and user agents alike:

- [WAI-ARIA Authoring Practices - Design Patterns and Widgets](#)
- [Heydon Pickering - ARIA Examples](#)
- [Inclusive Components](#)

Other Points for Consideration

Setting the language

Indicate the human language of page texts as screen reader software uses this to select the correct voice settings:

- [WebAIM - Document Language](#)

Setting the document title

Set the document `<title>` to correctly describe the current page content as this ensures that the user remains aware of the current page context:

- [WCAG - Understanding the Document Title Requirement](#)

We can set this in React using the [React Document Title Component](#).

Color contrast

Ensure that all readable text on your website has sufficient color contrast to remain maximally readable by users with low vision:

- [WCAG - Understanding the Color Contrast Requirement](#)
- [Everything About Color Contrast And Why You Should Rethink It](#)
- [A11yProject - What is Color Contrast](#)

It can be tedious to manually calculate the proper color combinations for all cases in your website so instead, you can [calculate an entire accessible color palette with Colorable](#).

Both the aXe and WAVE tools mentioned below also include color contrast tests and will report on contrast errors.

If you want to extend your contrast testing abilities you can use these tools:

- [WebAIM - Color Contrast Checker](#)
- [The Paciello Group - Color Contrast Analyzer](#)

Development and Testing Tools

There are a number of tools we can use to assist in the creation of accessible web applications.

The keyboard

By far the easiest and also one of the most important checks is to test if your entire website can be reached and used with the keyboard alone. Do this by:

1. Disconnecting your mouse.
2. Using `Tab` and `Shift+Tab` to browse.
3. Using `Enter` to activate elements.
4. Where required, using your keyboard arrow keys to interact with some elements, such as menus and dropdowns.

Development assistance

We can check some accessibility features directly in our JSX code. Often intellisense checks are already provided in JSX aware IDE's for the ARIA roles, states and properties. We also have access to the following tool:

eslint-plugin-jsx-a11y

The `eslint-plugin-jsx-a11y` plugin for ESLint provides AST linting feedback regarding accessibility issues in your JSX. Many IDE's allow you to integrate these findings directly into code analysis and source code windows.

[Create React App](#) has this plugin with a subset of rules activated. If you want to enable even more accessibility rules, you can create an `.eslintrc` file in the root of your project with this content:

```
{
  "extends": ["react-app", "plugin:jsx-a11y/recommended"],
  "plugins": ["jsx-a11y"]
}
```

Testing accessibility in the browser

A number of tools exist that can run accessibility audits on web pages in your browser. Please use them in combination with other accessibility checks mentioned here as they can only test the technical accessibility of your HTML.

aXe, aXe-core and react-axe

Deque Systems offers [aXe-core](#) for automated and end-to-end accessibility tests of your applications. This module includes integrations for Selenium.

The [Accessibility Engine](#) or aXe, is an accessibility inspector browser extension built on `aXe-core`.

You can also use the [@axe-core/react](#) module to report these accessibility findings directly to the console while developing and debugging.

WebAIM WAVE

The [Web Accessibility Evaluation Tool](#) is another accessibility browser extension.

Accessibility inspectors and the Accessibility Tree

The [Accessibility Tree](#) is a subset of the DOM tree that contains accessible objects for every DOM element that should be exposed to assistive technology, such as screen readers.

In some browsers we can easily view the accessibility information for each element in the accessibility tree:

- [Using the Accessibility Inspector in Firefox](#)
- [Using the Accessibility Inspector in Chrome](#)
- [Using the Accessibility Inspector in OS X Safari](#)

Screen readers

Testing with a screen reader should form part of your accessibility tests.

Please note that browser / screen reader combinations matter. It is recommended that you test your application in the browser best suited to your screen reader of choice.

Commonly Used Screen Readers

NVDA in Firefox

[NonVisual Desktop Access](#) or NVDA is an open source Windows screen reader that is widely used.

Refer to the following guides on how to best use NVDA:

- [WebAIM - Using NVDA to Evaluate Web Accessibility](#)
- [Deque - NVDA Keyboard Shortcuts](#)

VoiceOver in Safari

VoiceOver is an integrated screen reader on Apple devices.

Refer to the following guides on how to activate and use VoiceOver:

- [WebAIM - Using VoiceOver to Evaluate Web Accessibility](#)
- [Deque - VoiceOver for OS X Keyboard Shortcuts](#)
- [Deque - VoiceOver for iOS Shortcuts](#)

JAWS in Internet Explorer

[Job Access With Speech](#) or JAWS, is a prolifically used screen reader on Windows.

Refer to the following guides on how to best use JAWS:

- [WebAIM - Using JAWS to Evaluate Web Accessibility](#)
- [Deque - JAWS Keyboard Shortcuts](#)

Other Screen Readers

ChromeVox in Google Chrome

ChromeVox is an integrated screen reader on Chromebooks and is available [as an extension](#) for Google Chrome.

Refer to the following guides on how best to use ChromeVox:

- [Google Chromebook Help - Use the Built-in Screen Reader](#)
- [ChromeVox Classic Keyboard Shortcuts Reference](#)

[Go to TOC](#)

將 React 加入到網頁

按自己所需可多可少的採用 React。

React 在剛推出的時候就容許被逐步採用，你可以按自己所需可多可少的採用 React。或許你只想在現存的網頁上增加少量的互動性。採用 React component 會是個非常好的選擇。

大部分的網頁不是，也不需要是 single-page 的應用程式。你只需要**幾行的程式碼，並且不需要任何建立工具**，就可以在你一小部分的網頁上嘗試採用 React。你可以選擇逐步擴大它的應用範圍，或是只使用在少部分的可變 widget 上。

- [一分鐘內加入 React](#)
- 可選：嘗試 React 與 JSX（不需要 bundler！）

一分鐘內加入 React

在本章節裡，我們會示範如何在 HTML 網頁上加入一個 React component。你可以利用自己的網頁，或建立一個空白的 HTML 文件來跟隨着我們練習。

我們不需要任何複雜的工具或安裝需求 —— 你只需要連接到網路和一分鐘的時間，就能完成本章節。

可選：[下載完整範例 \(2KB zipped\)](#)

第一步：在 HTML 上加入一個 DOM Container

首先，打開你想編輯的 HTML 網頁。在你想利用 React 來展示內容的位置上，加上一個空白的 `<div>` 標籤。例如：

```
<!-- ... 現存 HTML ... -->
<div id="like_button_container"></div>
<!-- ... 現存 HTML ... -->
```

我們給這個 `<div>` 加上一個獨一無二的 `id` HTML attribute。這會容許我們稍後在 JavaScript 程式碼裡找到它，並且在裡面展示一個 React component。

提示

你可以放置像這樣的「container」`<div>` 在 `<body>` 標籤裡的**任何地方**。你也可以按需要，在一頁裡放置多個獨立的 DOM container。它們通常都是空白的 —— React 會替換 DOM container 裡任何現存的內容。

第二步：加上 Script 標籤

下一步，在 HTML 網頁的 `</body>` 結束標籤前，加上三個 `<script>` 標籤。

```
<!-- ... 其他 HTML ... -->
<!-- 載入 React。 -->
```

```

<!-- 注意：在發佈應用程式前，請把「development.js」替換成「production.min.js」。 -->
<script src="https://unpkg.com/react@18/umd/react.development.js" crossorigin>
</script>
<script src="https://unpkg.com/react-dom@18/umd/react-dom.development.js" crossorigin></script>

<!-- 載入我們的 React component。-->
<script src="like_button.js"></script>

</body>

```

前兩個的標籤會載入 React。第三個會載入你的 component 程式碼。

第三步：建立一個 React Component

在你的 HTML 網頁旁邊，建立一個名為 `like_button.js` 的文件

打開[這個 starter code](#)，然後把它貼上到你剛建立的文件裡。

提示

這程式碼定立了一個名為 `LikeButton` 的 React component。請別擔心如果你還沒明白它 —— 我們會在[實用指南](#)和[主要概念指南](#)裡解構 React 的基礎。目前，我們就先讓它展示在畫面上吧！

在 [starter code](#) 之後，請在 `like_button.js` 的底部加上以三行的程式碼：

```

// ... 你貼上的 starter code ...

const domContainer = document.querySelector('#like_button_container');
const root = ReactDOM.createRoot(domContainer);
root.render(e(LikeButton));

```

這三行的程式碼會找我們在第一步所加入的 `<div>`，用它建立一個 React app 然後在裡面顯示我們的「Like」按鈕 React component。

大功告成！

沒有第四步了。你剛剛已經將第一個 React component 加入到你的網頁上。

查看後續的章節了解更多有關採用 React 的提示。

[按這裡看完整範例的程式碼](#)

[下載完整範例 \(2KB zipped\)](#)

提示：重用 Component

通常，你會想把 React component 展示在 HTML 網頁上的不同地方。以下的範例是展示我們的「Like」按鈕三次，再各自傳送資料給它們：

[按這裡看完整範例的程式碼](#)

[下載完整範例 \(2KB zipped\)](#)

注意

當網頁上以 React 驅動的部分是互相獨立的時候，這種策略會非常有用。在 React 的程式碼裡，使用 [component composition](#) 會反而比較容易。

提示：為線上環境壓縮 JavaScript

在部署你的網頁到線上環境之前，要留意沒壓縮的 JavaScript 會明顯的減慢使用者載入網頁的速度。

如果你已經壓縮了應用程式的 scripts，而且確保了你發佈的 HTML 是載入了以 `production.min.js` 結尾的 React 版本，那麼你的網頁已經發佈就緒：

```
<script src="https://unpkg.com/react@18/umd/react.production.min.js" crossorigin>
</script>
<script src="https://unpkg.com/react-dom@18/umd/react-dom.production.min.js"
crossorigin></script>
```

如果你沒有一個為你的 scripts 進行壓縮的步驟，[這裡有個方法設定](#)。

可選：嘗試 React 與 JSX

上面所接觸的範例，我們只倚靠着瀏覽器自身所支援的特性。這就是為什麼我們會用一個 JavaScript function call 來告訴 React 要展示什麼：

```
const e = React.createElement;

// 展示一個「Like」<button>
return e(
  'button',
  { onClick: () => this.setState({ liked: true }) },
  'Like'
);
```

不過，React 也提供了一個使用 [JSX](#) 的選擇：

```
// 展示一個「Like」<button>
return (
  <button onClick={() => this.setState({ liked: true })}>
    Like
  </button>
);
```

這兩段的程式碼是等同的。雖然 **JSX 是完全可選的**，但許多人也覺得它有助編寫 UI 程式碼 —— 無論是使用 React 或其他函式庫。

你可以使用[這個線上轉換器](#)來嘗試 JSX。

快速嘗試 JSX

在你的項目中，嘗試 JSX 最快的方法就是將這個 `<script>` 標籤加入你的網頁上：

```
<script src="https://unpkg.com/babel-standalone@6/babel.min.js"></script>
```

現在你就可以在任何 `<script>` 標籤裡使用 JSX，方法就是為它們加上 `type="text/babel"` 的 attribute。這裡是個 [HTML 文件與 JSX 的範例](#)，你可以下載來嘗試一下。

現在你可以透過加入 `type="text/babel"` attribute 在任何 `<script>` tag 中使用 JSX。這裡是一個帶有 JSX 的 [HTML 範例](#)，你可以下載並且試試看。

這種方式最適合用來學習和建立簡單的示範。然而，它會令你的網頁變慢，而且並不適合發佈到線上環境。當你準備好到下一步，請移除掉你剛加入的 `<script>` 標籤和 `type="text/babel"` attribute。在下一個章節，你會設定一個 JSX preprocessor 來自動轉換所有的 `<script>` 標籤。

將 JSX 加入到項目

將 JSX 加入到項目裡並不需要複雜的工具，例如一個 bundler 或開發伺服器。本質上，加入 JSX 就像加入一個 CSS preprocessor。這只需要你安裝 [Node.js](#) 到你的電腦裡。

在 terminal 轉到你的項目文件夾裡，再貼上以下的兩行指令：

1. 第一步：執行 `npm init -y` (如果失敗了，[這裡有方法解決](#))
2. 第二步：執行 `npm install babel-cli@6 babel-preset-react-app@3`

提示

我們只使用 `npm` 來安裝 JSX preprocessor，之後你並不再需要它。React 和你的應用程式碼都可以留在 `<script>` 標籤裡，並不需要進行修改。

恭喜你！你剛剛為你的項目加入了發佈就緒的 JSX 設定了。

執行 JSX Preprocessor

建立一個名為 `src` 的文件夾，然後執行這個終端指令：

```
npx babel --watch src --out-dir . --presets react-app/prod
```

Note

`npx` 不是拼寫錯誤 —— 它是一個 [npm 5.2+](#) 附帶的 package 執行器。

如果你看見一個寫着「You have mistakenly installed the `babel` package」的錯誤訊息，你或許跳過了[上一步](#)。在同一個文件夾裡執行它，然後再重新嘗試。

請不要等待它完成 —— 這個指令會啟動一個 JSX 自動監測器。

如果你現在利用這個 [JSX starter code](#) 建立一個名為的 `src/like_button.js` 文件，監測器會建立一個預先處理過，由普通 JavaScript 程式碼組成，並且適合瀏覽器載入的 `like_button.js`。當你使用 JSX 來編輯文件時，轉換過程會自動重新執行。

再者，這也容許我們在舊瀏覽器上，使用現代 JavaScript 的語法特性，例如 class。我們剛加入的工具叫 Babel，你可以在[它的官方文件](#)裡了解更多。

如果你發現自己習慣了使用各種建立工具，而且希望它們為你做更多，[下一個章節](#)會介紹一些受歡迎和容易上手的 toolchains。不過，只利用 script 標籤也游刃有餘！

Animation Add-Ons

Note:

`ReactTransitionGroup` and `ReactCSSTransitionGroup` have been moved to the `react-transition-group` package that is maintained by the community. Its 1.x branch is completely API-compatible with the existing addons. Please file bugs and feature requests in the [new repository](#).

The `ReactTransitionGroup` add-on component is a low-level API for animation, and `ReactCSSTransitionGroup` is an add-on component for easily implementing basic CSS animations and transitions.

High-level API: `ReactCSSTransitionGroup`

`ReactCSSTransitionGroup` is a high-level API based on `ReactTransitionGroup` and is an easy way to perform CSS transitions and animations when a React component enters or leaves the DOM. It's inspired by the excellent [ng-animate](#) library.

Importing

```
import ReactCSSTransitionGroup from 'react-transition-group'; // ES6
var ReactCSSTransitionGroup = require('react-transition-group'); // ES5 with npm
```

```
class TodoList extends React.Component {
  constructor(props) {
    super(props);
    this.state = {items: ['hello', 'world', 'click', 'me']};
    this.handleAdd = this.handleAdd.bind(this);
  }

  handleAdd() {
    const newItems = this.state.items.concat([
      prompt('Enter some text')
    ]);
    this.setState({items: newItems});
  }

  handleRemove(i) {
    let newItems = this.state.items.slice();
    newItems.splice(i, 1);
    this.setState({items: newItems});
  }

  render() {
    const items = this.state.items.map((item, i) => (
      <div key={i} onClick={() => this.handleRemove(i)}>
        {item}
      </div>
    )));
  }
}
```

```

    return (
      <div>
        <button onClick={this.handleAdd}>Add Item</button>
        <ReactCSSTransitionGroup
          transitionName="example"
          transitionEnterTimeout={500}
          transitionLeaveTimeout={300}>
          {items}
        </ReactCSSTransitionGroup>
      </div>
    );
  }
}

```

Note:

You must provide the `key` attribute for all children of `ReactCSSTransitionGroup`, even when only rendering a single item. This is how React will determine which children have entered, left, or stayed.

In this component, when a new item is added to `ReactCSSTransitionGroup` it will get the `example-enter` CSS class and the `example-enter-active` CSS class added in the next tick. This is a convention based on the `transitionName` prop.

You can use these classes to trigger a CSS animation or transition. For example, try adding this CSS and adding a new list item:

```

.example-enter {
  opacity: 0.01;
}

.example-enter.example-enter-active {
  opacity: 1;
  transition: opacity 500ms ease-in;
}

.example-leave {
  opacity: 1;
}

.example-leave.example-leave-active {
  opacity: 0.01;
  transition: opacity 300ms ease-in;
}

```

You'll notice that animation durations need to be specified in both the CSS and the render method; this tells React when to remove the animation classes from the element and -- if it's leaving -- when to remove the element from the DOM.

Animate Initial Mounting

`ReactCSSTransitionGroup` provides the optional prop `transitionAppear`, to add an extra transition phase at the initial mount of the component. There is generally no transition phase at the initial mount as the default value of `transitionAppear` is `false`. The following is an example which passes the prop `transitionAppear` with the value `true`.

```
render() {
  return (
    <ReactCSSTransitionGroup
      transitionName="example"
      transitionAppear={true}
      transitionAppearTimeout={500}
      transitionEnter={false}
      transitionLeave={false}>
      <h1>Fading at Initial Mount</h1>
    </ReactCSSTransitionGroup>
  );
}
```

During the initial mount `ReactCSSTransitionGroup` will get the `example-appear` CSS class and the `example-appear-active` CSS class added in the next tick.

```
.example-appear {
  opacity: 0.01;
}

.example-appear.example-appear-active {
  opacity: 1;
  transition: opacity .5s ease-in;
}
```

At the initial mount, all children of the `ReactCSSTransitionGroup` will `appear` but not `enter`. However, all children later added to an existing `ReactCSSTransitionGroup` will `enter` but not `appear`.

Note:

The prop `transitionAppear` was added to `ReactCSSTransitionGroup` in version 0.13. To maintain backwards compatibility, the default value is set to `false`.

However, the default values of `transitionEnter` and `transitionLeave` are `true` so you must specify `transitionEnterTimeout` and `transitionLeaveTimeout` by default. If you don't need either enter or leave animations, pass `transitionEnter={false}` or `transitionLeave={false}`.

Custom Classes

It is also possible to use custom class names for each of the steps in your transitions. Instead of passing a string into `transitionName` you can pass an object containing either the `enter` and `leave` class names, or an object containing the `enter`, `enter-active`, `leave-active`, and `leave` class names. If only the enter and leave classes are provided, the enter-active and leave-active classes will be determined by appending '-active' to the end of the class name. Here are two examples using custom classes:

```
// ...
<ReactCSSTransitionGroup
  transitionName={{ 
    enter: 'enter',
    enterActive: 'enterActive',
    leave: 'leave',
    leaveActive: 'leaveActive',
    appear: 'appear',
    appearActive: 'appearActive'
  }}>
  {item}
</ReactCSSTransitionGroup>

<ReactCSSTransitionGroup
  transitionName={{ 
    enter: 'enter',
    leave: 'leave',
    appear: 'appear'
  }}>
  {item2}
</ReactCSSTransitionGroup>
// ...
```

Animation Group Must Be Mounted To Work

In order for it to apply transitions to its children, the `ReactCSSTransitionGroup` must already be mounted in the DOM or the prop `transitionAppear` must be set to `true`.

The example below would **not** work, because the `ReactCSSTransitionGroup` is being mounted along with the new item, instead of the new item being mounted within it. Compare this to the [Getting Started](#) section above to see the difference.

```
render() {
  const items = this.state.items.map((item, i) => (
    <div key={item} onClick={() => this.handleRemove(i)}>
      <ReactCSSTransitionGroup transitionName="example">
        {item}
      </ReactCSSTransitionGroup>
    </div>
  )));
  return (
    <div>
      <button onClick={this.handleAdd}>Add Item</button>
      {items}
    </div>
  );
}
```

Animating One or Zero Items

In the example above, we rendered a list of items into `ReactCSSTransitionGroup`. However, the children of `ReactCSSTransitionGroup` can also be one or zero items. This makes it possible to animate a single element entering or leaving. Similarly, you can animate a new element replacing the current element. For example, we can implement a simple image carousel like this:

```
import ReactCSSTransitionGroup from 'react-transition-group';

function ImageCarousel(props) {
  return (
    <div>
      <ReactCSSTransitionGroup
        transitionName="carousel"
        transitionEnterTimeout={300}
        transitionLeaveTimeout={300}>
        <img src={props.imageSrc} key={props.imageSrc} />
      </ReactCSSTransitionGroup>
    </div>
  );
}
```

Disabling Animations

You can disable animating `enter` or `leave` animations if you want. For example, sometimes you may want an `enter` animation and no `leave` animation, but `ReactCSSTransitionGroup` waits for an animation to complete before removing your DOM node. You can add `transitionEnter={false}` or `transitionLeave={false}` props to `ReactCSSTransitionGroup` to disable these animations.

Note:

When using `ReactCSSTransitionGroup`, there's no way for your components to be notified when a transition has ended or to perform any more complex logic around animation. If you want more fine-grained control, you can use the lower-level `ReactTransitionGroup` API which provides the hooks you need to do custom transitions.

Low-level API: `ReactTransitionGroup`

Importing

```
import ReactTransitionGroup from 'react-addons-transition-group' // ES6
var ReactTransitionGroup = require('react-addons-transition-group') // ES5 with npm
```

`ReactTransitionGroup` is the basis for animations. When children are declaratively added or removed from it (as in the [example above](#)), special lifecycle methods are called on them.

- `componentWillAppear()`

- `componentDidAppear()`
- `componentWillEnter()`
- `componentDidEnter()`
- `componentWillLeave()`
- `componentDidLeave()`

Rendering a Different Component

`ReactTransitionGroup` renders as a `span` by default. You can change this behavior by providing a `component` prop. For example, here's how you would render a ``:

```
<ReactTransitionGroup component="ul">
  {/* ... */}
</ReactTransitionGroup>
```

Any additional, user-defined, properties will become properties of the rendered component. For example, here's how you would render a `` with CSS class:

```
<ReactTransitionGroup component="ul" className="animated-list">
  {/* ... */}
</ReactTransitionGroup>
```

Every DOM component that React can render is available for use. However, `component` does not need to be a DOM component. It can be any React component you want; even ones you've written yourself! Just write `component={List}` and your component will receive `this.props.children`.

Rendering a Single Child

People often use `ReactTransitionGroup` to animate mounting and unmounting of a single child such as a collapsible panel. Normally `ReactTransitionGroup` wraps all its children in a `span` (or a custom `component` as described above). This is because any React component has to return a single root element, and `ReactTransitionGroup` is no exception to this rule.

However if you only need to render a single child inside `ReactTransitionGroup`, you can completely avoid wrapping it in a `` or any other DOM component. To do this, create a custom component that renders the first child passed to it directly:

```
function FirstChild(props) {
  const childrenArray = React.Children.toArray(props.children);
  return childrenArray[0] || null;
}
```

Now you can specify `FirstChild` as the `component` prop in `<ReactTransitionGroup>` props and avoid any wrappers in the result DOM:

```
<ReactTransitionGroup component={FirstChild}>
  {someCondition ? <MyComponent /> : null}
</ReactTransitionGroup>
```

This only works when you are animating a single child in and out, such as a collapsible panel. This approach wouldn't work when animating multiple children or replacing the single child with another child, such as an image carousel. For an image carousel, while the current image is animating out, another image will animate in, so `<ReactTransitionGroup>` needs to give them a common DOM parent. You can't avoid the wrapper for multiple children, but you can customize the wrapper with the `component` prop as described above.

Reference

`componentWillAppear()` **{#componentwillappear}**

`componentWillAppear(callback)`

This is called at the same time as `componentDidMount()` for components that are initially mounted in a `TransitionGroup`. It will block other animations from occurring until `callback` is called. It is only called on the initial render of a `TransitionGroup`.

`componentDidAppear()` **{#componentdidappear}**

`componentDidAppear()`

This is called after the `callback` function that was passed to `componentWillAppear` is called.

`componentWillEnter()` **{#componentwillenter}**

`componentWillEnter(callback)`

This is called at the same time as `componentDidMount()` for components added to an existing `TransitionGroup`. It will block other animations from occurring until `callback` is called. It will not be called on the initial render of a `TransitionGroup`.

`componentDidEnter()` **{#componentdidenter}**

`componentDidEnter()`

This is called after the `callback` function that was passed to `componentWillEnter()` is called.

`componentWillLeave()` **{#componentwillleave}**

`componentWillLeave(callback)`

This is called when the child has been removed from the `ReactTransitionGroup`. Though the child has been removed, `ReactTransitionGroup` will keep it in the DOM until `callback` is called.

componentDidLeave() {#componentdidleave}

`componentDidLeave()`

This is called when the `willLeave` `callback` is called (at the same time as `componentWillUnmount()`).

[Go to TOC](#)

Keyed Fragments

Note:

`React.addons` entry point is deprecated as of React v15.5. We now have first class support for fragments which you can read about [here](#).

Importing

```
import createFragment from 'react-addons-create-fragment'; // ES6
var createFragment = require('react-addons-create-fragment'); // ES5 with npm
```

Overview

In most cases, you can use the `key` prop to specify keys on the elements you're returning from `render`. However, this breaks down in one situation: if you have two sets of children that you need to reorder, there's no way to put a key on each set without adding a wrapper element.

That is, if you have a component such as:

```
function Swapper(props) {
  let children;
  if (props.swapped) {
    children = [props.rightChildren, props.leftChildren];
  } else {
    children = [props.leftChildren, props.rightChildren];
  }
  return <div>{children}</div>;
}
```

The children will unmount and remount as you change the `swapped` prop because there aren't any keys marked on the two sets of children.

To solve this problem, you can use the `createFragment` add-on to give keys to the sets of children.

```
Array<ReactNode> createFragment(object children) {#arrayreactnode-createfragmentobject-children}
```

Instead of creating arrays, we write:

```
import createFragment from 'react-addons-create-fragment';

function Swapper(props) {
  let children;
  if (props.swapped) {
    children = createFragment({
      right: props.rightChildren,
      left: props.leftChildren
    });
  }
  return <div>{children}</div>;
}
```

```
    } else {
      children = createFragment({
        left: props.leftChildren,
        right: props.rightChildren
      });
    }
    return <div>{children}</div>;
}
```

The keys of the passed object (that is, `left` and `right`) are used as keys for the entire set of children, and the order of the object's keys is used to determine the order of the rendered children. With this change, the two sets of children will be properly reordered in the DOM without unmounting.

The return value of `createFragment` should be treated as an opaque object; you can use the `React.Children` helpers to loop through a fragment but should not access it directly. Note also that we're relying on the JavaScript engine preserving object enumeration order here, which is not guaranteed by the spec but is implemented by all major browsers and VMs for objects with non-numeric keys.

Performance Tools

Note:

As of React 16, `react-addons-perf` is not supported. Please use [your browser's profiling tools](#) to get insight into which components re-render.

Importing

```
import Perf from 'react-addons-perf'; // ES6
var Perf = require('react-addons-perf'); // ES5 with npm
```

Overview

React is usually quite fast out of the box. However, in situations where you need to squeeze every ounce of performance out of your app, it provides a `shouldComponentUpdate()` method where you can add optimization hints to React's diff algorithm.

In addition to giving you an overview of your app's overall performance, `Perf` is a profiling tool that tells you exactly where you need to put these methods.

See these articles for an introduction to React performance tooling:

- ["How to Benchmark React Components"](#)
- ["Performance Engineering with React"](#)
- ["A Deep Dive into React Perf Debugging"](#)

Development vs. Production Builds

If you're benchmarking or seeing performance problems in your React apps, make sure you're testing with the [minified production build](#). The development build includes extra warnings that are helpful when building your apps, but it is slower due to the extra bookkeeping it does.

However, the perf tools described on this page only work when using the development build of React. Therefore, the profiler only serves to indicate the *relatively* expensive parts of your app.

Using Perf

The `Perf` object can be used with React in development mode only. You should not include this bundle when building your app for production.

Getting Measurements

- `start()`
- `stop()`

- `getLastMeasurements()`

Printing Results

The following methods use the measurements returned by `Perf.getLastMeasurements()` to pretty-print the result.

- `printInclusive()`
 - `printExclusive()`
 - `printWasted()`
 - `printOperations()`
 - `printDOM()`
-

Reference

`start() {#start}`

`stop() {#stop}`

```
Perf.start()  
// ...  
Perf.stop()
```

Start/stop the measurement. The React operations in-between are recorded for analyses below. Operations that took an insignificant amount of time are ignored.

After stopping, you will need `Perf.getLastMeasurements()` to get the measurements.

`getLastMeasurements() {#getlastmeasurements}`

```
Perf.getLastMeasurements()
```

Get the opaque data structure describing measurements from the last start-stop session. You can save it and pass it to the other print methods in `Perf` to analyze past measurements.

Note

Don't rely on the exact format of the return value because it may change in minor releases. We will update the documentation if the return value format becomes a supported part of the public API.

`printInclusive() {#printinclusive}`

```
Perf.printInclusive(measurements)
```

Prints the overall time taken. When no arguments are passed, `printInclusive` defaults to all the measurements from the last recording. This prints a nicely formatted table in the console, like so:

(index)	Owner > component	Inclusive time (ms)	Instances
0	"<root> > App"	15.31	37
1	"App > Box"	6.47	37
2	"Box > Box2"	1.96	37

printExclusive() `{#printexclusive}`

`Perf.printExclusive(measurements)`

"Exclusive" times don't include the times taken to mount the components: processing props, calling `componentWillMount` and `componentDidMount`, etc.

(index)	Component c...	Total inclus...	Exclusive m...	Exclusive r...	Mount time ...	Render time...	Instances
0	"App"	15.31	1.76	1.23	0.04	0.03	37

printWasted() `{#printwasted}`

`Perf.printWasted(measurements)`

The most useful part of the profiler.

"Wasted" time is spent on components that didn't actually render anything, e.g. the render stayed the same, so the DOM wasn't touched.

(index)	Owner > component	Wasted time (ms)	Instances
0	"<root> > App"	15.317999947001226	37
1	"App > Box"	6.479999952716753	37
2	"Box > Box2"	1.9630000460892916	37

printOperations() `{#printoperations}`

`Perf.printOperations(measurements)`

Prints the underlying DOM manipulations, e.g. "set innerHTML" and "remove".

(index)	data-reactid	type	args
0	"0"	"set innerHTML"	"<div data-reactid=".0\..."

printDOM() `{#printdom}`

`Perf.printDOM(measurements)`

This method has been renamed to `printOperations()`. Currently `printDOM()` still exists as an alias but it prints a deprecation warning and will eventually be removed.

PureRenderMixin

Note:

`PureRenderMixin` is a legacy add-on. Use `React.PureComponent` instead.

Importing

```
import PureRenderMixin from 'react-addons-pure-render-mixin'; // ES6
var PureRenderMixin = require('react-addons-pure-render-mixin'); // ES5 with npm
```

Overview

If your React component's render function renders the same result given the same props and state, you can use this mixin for a performance boost in some cases.

Example:

```
const createReactClass = require('create-react-class');

createReactClass({
  mixins: [PureRenderMixin],

  render: function() {
    return <div className={this.props.className}>foo</div>;
  }
});
```

Under the hood, the mixin implements `shouldComponentUpdate`, in which it compares the current props and state with the next ones and returns `false` if the equalities pass.

Note:

This only shallowly compares the objects. If these contain complex data structures, it may produce false-negatives for deeper differences. Only mix into components which have simple props and state, or use `forceUpdate()` when you know deep data structures have changed. Or, consider using [immutable objects](#) to facilitate fast comparisons of nested data.

Furthermore, `shouldComponentUpdate` skips updates for the whole component subtree. Make sure all the children components are also "pure".

[Go to TOC](#)

Shallow Compare

Note:

`shallowCompare` is a legacy add-on. Use `React.memo` or `React.PureComponent` instead.

Importing

```
import shallowCompare from 'react-addons-shallow-compare'; // ES6
var shallowCompare = require('react-addons-shallow-compare'); // ES5 with npm
```

Overview

Before `React.PureComponent` was introduced, `shallowCompare` was commonly used to achieve the same functionality as `PureRenderMixin` while using ES6 classes with React.

If your React component's render function is "pure" (in other words, it renders the same result given the same props and state), you can use this helper function for a performance boost in some cases.

Example:

```
export class SampleComponent extends React.Component {
  shouldComponentUpdate(nextProps, nextState) {
    return shallowCompare(this, nextProps, nextState);
  }

  render() {
    return <div className={this.props.className}>foo</div>;
  }
}
```

`shallowCompare` performs a shallow equality check on the current `props` and `nextProps` objects as well as the current `state` and `nextState` objects.

It does this by iterating on the keys of the objects being compared and returning true when the values of a key in each object are not strictly equal.

`shallowCompare` returns `true` if the shallow comparison for props or state fails and therefore the component should update.

`shallowCompare` returns `false` if the shallow comparison for props and state both pass and therefore the component does not need to update.

[Go to TOC](#)

Shallow Renderer

如何 Import

```
import ShallowRenderer from 'react-test-renderer/shallow'; // ES6
var ShallowRenderer = require('react-test-renderer/shallow'); // ES5 with npm
```

概觀

當在為 React 寫單元測試時，shallow render 十分有用。Shallow render 可以只 render 「第一層」的 component，並且對 component 的 render 方法的回傳值進行 assert，不必擔心 child component 的行為，child component 並沒有被實例化或被 render。Shallow render 不依賴 DOM。

例如，假設你有下列 component：

```
function MyComponent() {
  return (
    <div>
      <span className="heading">Title</span>
      <Subcomponent foo="bar" />
    </div>
  );
}
```

你可以使用 assert：

```
import ShallowRenderer from 'react-test-renderer/shallow';

// in your test:
const renderer = new ShallowRenderer();
renderer.render(<MyComponent />);
const result = renderer.getRenderOutput();

expect(result.type).toBe('div');
expect(result.props.children).toEqual([
  <span className="heading">Title</span>,
  <Subcomponent foo="bar" />
]);
```

Shallow testing 目前有些限制，並不支援 refs。

注意：

我們建議你可以查看 Enzyme 的 [Shallow Rendering API](#)。它在相同功能的基礎上提供更棒更高階的 API。

參考

`shallowRenderer.render() {#shallowrendererrender}`

你可以把 `shallowRenderer` 想成一個用來 `render` 測試中 `component` 的「空間」，並且可以從中提取該 `component` 輸出的內容。

`shallowRenderer.render()` 類似於 `root.render()`，但它不依賴 DOM 且只 `render` 一層。這意味著你可以對測試的 `component` 及其 child `component` 進行隔離測試。

`shallowRenderer.getRenderOutput() {#shallowrenderergetrenderoutput}`

在 `shallowRenderer.render()` 被呼叫後，你可以使用 `shallowRenderer.getRenderOutput()` 來取得該 `component` 第一層的輸出內容。

然後，就可以對輸出的內容進行 `assert` 操作。

Test Utilities

如何 Import

```
import ReactTestUtils from 'react-dom/test-utils'; // ES6  
var ReactTestUtils = require('react-dom/test-utils'); // ES5 with npm
```

概觀

`ReactTestUtils` 使你可以輕鬆在你選擇的測試框架中測試 React component。在 Facebook，我們使用 [Jest](#) 以方便地進行 JavaScript 測試。你可以從 Jest 網站的 [React 教學](#)學習如何使用 Jest。

注意：

我們推薦使用 [React Testing Library](#)，它促使你寫出的測試能像使用者一樣地使用 component。

對於 React ≤ 16 的版本，[Enzyme](#) 的測試工具，讓你能輕易 assert、操作及遍歷 React component 的輸出。

- `act()`
- `mockComponent()`
- `isElement()`
- `isElementOfType()`
- `isDOMComponent()`
- `isCompositeComponent()`
- `isCompositeComponentWithType()`
- `findAllInRenderedTree()`
- `scryRenderedDOMComponentsWithClass()`
- `findRenderedDOMComponentWithClass()`
- `scryRenderedDOMComponentsWithTag()`
- `findRenderedDOMComponentWithTag()`
- `scryRenderedComponentsWithType()`
- `findRenderedComponentWithType()`
- `renderIntoDocument()`
- `Simulate`

參考資料

act() {#act}

為了準備讓 component 進行 assert，將 render component 及執行更新的程式碼放在 `act()` 中。這讓你的測試更貼近 React 在瀏覽器中的運作方式。

注意

如果你使用 `react-test-renderer`，它也提供行為相同的 `act` function。

舉例來說，假設我們有個 `Counter` component：

```
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {count: 0};
    this.handleClick = this.handleClick.bind(this);
  }
  componentDidMount() {
    document.title = `You clicked ${this.state.count} times`;
  }
  componentDidUpdate() {
    document.title = `You clicked ${this.state.count} times`;
  }
  handleClick() {
    this.setState(state => ({
      count: state.count + 1,
    }));
  }
  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={this.handleClick}>
          Click me
        </button>
      </div>
    );
  }
}
```

我們可以這樣測試：

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import { act } from 'react-dom/test-utils';
import Counter from './Counter';

let container;

beforeEach(() => {
  container = document.createElement('div');
  document.body.appendChild(container);
});
```

```

afterEach(() => {
  document.body.removeChild(container);
  container = null;
});

it('can render and update a counter', () => {
  // Test first render and componentDidMount
  act(() => {
    ReactDOM.createRoot(container).render(<Counter />);
  });
  const button = container.querySelector('button');
  const label = container.querySelector('p');
  expect(label.textContent).toBe('You clicked 0 times');
  expect(document.title).toBe('You clicked 0 times');

  // Test second render and componentDidUpdate
  act(() => {
    button.dispatchEvent(new MouseEvent('click', {bubbles: true}));
  });
  expect(label.textContent).toBe('You clicked 1 times');
  expect(document.title).toBe('You clicked 1 times');
});

```

- 不要忘記，只有在 DOM container 已加到 `document` 裡面時，才可以 dispatch DOM event。你可以使用如 `react-testing-library` 的 helper 來減少 boilerplate 程式碼。
- `recipes` 說明文件內包含了 `act()` 的詳細資訊，包含範例以及用法。

mockComponent() {#mockcomponent}

```

mockComponent(
  componentClass,
  [mockTagName]
)

```

傳遞一個被 mock 的 component module 到這個方法後，它會增加有用的方法，讓它能做為虛擬的 React component。component 不會像平常一樣 render，它會變成一個簡單的 `<div>`（或其他標籤，如果有提供 `mockTagName`），包含任何提供的 children。

注意：

`mockComponent()` 是 legacy API。我們建議以 `jest.mock()` 作為替代。

isElement() {#iselement}

```
isElement(element)
```

如果 `element` 是 React element 的話就回傳 `true`。

isElementOfType() {#iselementoftype}

```
isElementOfType(
  element,
  componentClass
)
```

如果 `element` 是 type 為 `componentClass` 的 React element 就回傳 `true`。

isDOMComponent() {#isdomcomponent}

```
isDOMComponent(instance)
```

如果 `instance` 是 DOM component (如 `<div>` 或 ``) 就回傳 `true`。

isCompositeComponent() {#iscompositecomponent}

```
isCompositeComponent(instance)
```

如果 `instance` 是使用者定義的 component, 例如 class 或 function, 就回傳 `true`。

isCompositeComponentWithType() {#iscompositecomponentwithtype}

```
isCompositeComponentWithType(
  instance,
  componentClass
)
```

如果 `instance` 是 type 為 `componentClass` 的 component 就回傳 `true`。

findAllInRenderedTree() {#findallinrenderedtree}

```
findAllInRenderedTree(
  tree,
  test
)
```

遍歷 `tree` 中的所有 component, 並收集 `test(component)` 為 `true` 的所有 component。這個方法本身不是那麼好用, 但是它被其他測試工具做為基礎使用。

scryRenderedDOMComponentsWithClass() {#scryrendereddomcomponentswithclass}

```
scryRenderedDOMComponentsWithClass(
  tree,
  className
)
```

在已經被 render 的 tree 中尋找所有 DOM element，回傳 class 名稱符合 `className` 的 DOM component。

`findRenderedDOMComponentWithClass()` {#findrendereddomcomponentwithclass}

```
findRenderedDOMComponentWithClass(  
  tree,  
  className  
)
```

與 `scryRenderedDOMComponentsWithTag()` 相似，不過預期只有一個結果。如果符合預期則回傳那個結果，否則拋出例外。

`scryRenderedDOMComponentsWithTag()` {#scryrendereddomcomponentswithtag}

```
scryRenderedDOMComponentsWithTag(  
  tree,  
  tagName  
)
```

在已經被 render 的 tree 中尋找所有 DOM element，回傳 tag 名稱符合 `tagName` 的 DOM component。

`findRenderedDOMComponentWithTag()` {#findrendereddomcomponentwithtag}

```
findRenderedDOMComponentWithTag(  
  tree,  
  tagName  
)
```

與 `scryRenderedDOMComponentsWithTag()` 相似，不過預期只有一個結果。如果符合預期則回傳那個結果，否則拋出例外。

`scryRenderedComponentsWithType()` {#scryrenderedcomponentswithtype}

```
scryRenderedComponentsWithType(  
  tree,  
  componentClass  
)
```

尋找所有 component type 與 `componentClass` 相同的 instance。

`findRenderedComponentWithType()` {#findrenderedcomponentwithtype}

```
findRenderedComponentWithType(
  tree,
  componentClass
)
```

與 `scryRenderedComponentsWithType()` 相似，不過預期只有一個結果。如果符合預期則回傳那個結果，否則拋出例外。

`renderIntoDocument()` {#renderintodocument}

```
renderIntoDocument(element)
```

Render React element 到 document 中獨立的 DOM node 裡。**這個 function 需要 DOM。** 它等效於：

```
const domContainer = document.createElement('div');
ReactDOM.createRoot(domContainer).render(element);
```

注意：

在 `import React` 前，你需要讓 `window`、`window.document` 和 `window.document.createElement` 在全域可以使用。否則 React 會認為它無法存取 DOM，像 `setState` 之類的方法也將無法運作。

其他工具

`Simulate` {#simulate}

```
Simulate.{eventName}(
  element,
  [eventData]
)
```

在 DOM node 上用可選的 `eventData` 事件資料模擬 event dispatch。

每一個 React 支援的事件在 `Simulate` 都有對應的方法。

點擊 element

```
// <button ref={(node) => this.button = node}>...</button>
const node = this.button;
ReactTestUtils.Simulate.click(node);
```

更改輸入欄位的值，然後按 ENTER 鍵。

```
// <input ref={(node) => this.textInput = node} />
const node = this.textInput;
node.value = 'giraffe';
ReactTestUtils.Simulate.change(node);
ReactTestUtils.Simulate.keyDown(node, {key: "Enter", keyCode: 13, which: 13});
```

注意：

你需要提供所有在你的 component 中有使用的事件屬性（如 keyCode、which 等等），因為 React 不會為你建立這些東西。

[Go to TOC](#)

Two-way Binding Helpers

Note:

`LinkedStateMixin` is deprecated as of React v15. The recommendation is to explicitly set the value and change handler, instead of using `LinkedStateMixin`.

Importing

```
import LinkedStateMixin from 'react-addons-linked-state-mixin'; // ES6
var LinkedStateMixin = require('react-addons-linked-state-mixin'); // ES5 with npm
```

Overview

`LinkedStateMixin` is an easy way to express two-way binding with React.

In React, data flows one way: from owner to child. We think that this makes your app's code easier to understand. You can think of it as "one-way data binding."

However, there are lots of applications that require you to read some data and flow it back into your program. For example, when developing forms, you'll often want to update some React `state` when you receive user input. Or perhaps you want to perform layout in JavaScript and react to changes in some DOM element size.

In React, you would implement this by listening to a "change" event, read from your data source (usually the DOM) and call `setState()` on one of your components. "Closing the data flow loop" explicitly leads to more understandable and easier-to-maintain programs. See [our forms documentation](#) for more information.

Two-way binding -- implicitly enforcing that some value in the DOM is always consistent with some React `state` -- is concise and supports a wide variety of applications. We've provided `LinkedStateMixin`: syntactic sugar for setting up the common data flow loop pattern described above, or "linking" some data source to React `state`.

Note:

`LinkedStateMixin` is just a thin wrapper and convention around the `onChange / setState()` pattern. It doesn't fundamentally change how data flows in your React application.

LinkedStateMixin: Before and After

Here's a simple form example without using `LinkedStateMixin`:

```
var createReactClass = require('create-react-class');

var NoLink = createReactClass({
  getInitialState: function() {
    return {message: 'Hello!'};
  },
  handleChange: function(event) {
    this.setState({message: event.target.value});
  },
  render: function() {
    var message = this.state.message;
    return <input type="text" value={message} onChange={this.handleChange}>;
  }
});
```

This works really well and it's very clear how data is flowing, however, with a lot of form fields it could get a bit verbose. Let's use `LinkedStateMixin` to save us some typing:

```
var createReactClass = require('create-react-class');

var WithLink = createReactClass({
  mixins: [LinkedStateMixin],
  getInitialState: function() {
    return {message: 'Hello!'};
  },
  render: function() {
    return <input type="text" valueLink={this.linkState('message')} />;
  }
});
```

`LinkedStateMixin` adds a method to your React component called `linkState()`. `linkState()` returns a `valueLink` object which contains the current value of the React state and a callback to change it.

`valueLink` objects can be passed up and down the tree as props, so it's easy (and explicit) to set up two-way binding between a component deep in the hierarchy and state that lives higher in the hierarchy.

Note that checkboxes have a special behavior regarding their `value` attribute, which is the value that will be sent on form submit if the checkbox is checked (defaults to `on`). The `value` attribute is not updated when the checkbox is checked or unchecked. For checkboxes, you should use `checkedLink` instead of `valueLink`:

```
<input type="checkbox" checkedLink={this.linkState('booleanValue')} />
```

Under the Hood

There are two sides to `LinkedStateMixin`: the place where you create the `valueLink` instance and the place where you use it. To prove how simple `LinkedStateMixin` is, let's rewrite each side separately to be more explicit.

valueLink Without LinkedStateMixin

```
var createReactClass = require('create-react-class');

var WithoutMixin = createReactClass({
```

```

getInitialState: function() {
  return {message: 'Hello!'};
},
handleChange: function(newValue) {
  this.setState({message: newValue});
},
render: function() {
  var valueLink = {
    value: this.state.message,
    requestChange: this.handleChange
  };
  return <input type="text" valueLink={valueLink} />;
}
);

```

As you can see, `valueLink` objects are very simple objects that just have a `value` and `requestChange` prop. And `LinkedStateMixin` is similarly simple: it just populates those fields with a `value` from `this.state` and a callback that calls `this.setState()`.

LinkedStateMixin Without valueLink

```

var LinkedStateMixin = require('react-addons-linked-state-mixin');
var createReactClass = require('create-react-class');

var WithoutLink = createReactClass({
  mixins: [LinkedStateMixin],
  getInitialState: function() {
    return {message: 'Hello!'};
  },
  render: function() {
    var valueLink = this.linkState('message');
    var handleChange = function(e) {
      valueLink.requestChange(e.target.value);
    };
    return <input type="text" value={valueLink.value} onChange={handleChange} />;
  }
});

```

The `valueLink` prop is also quite simple. It simply handles the `onChange` event and calls `this.props.valueLink.requestChange()` and also uses `this.props.valueLink.value` instead of `this.props.value`. That's it!

Immutability Helpers

Note:

`update` is a legacy add-on. Use `immutability-helper` instead.

Importing

```
import update from 'react-addons-update'; // ES6
var update = require('react-addons-update'); // ES5 with npm
```

Overview

React lets you use whatever style of data management you want, including mutation. However, if you can use immutable data in performance-critical parts of your application it's easy to implement a fast `shouldComponentUpdate()` method to significantly speed up your app.

Dealing with immutable data in JavaScript is more difficult than in languages designed for it, like [Clojure](#). However, we've provided a simple immutability helper, `update()`, that makes dealing with this type of data much easier, *without* fundamentally changing how your data is represented. You can also take a look at Facebook's [Immutable-js](#) and the [Advanced Performance](#) section for more detail on Immutable-js.

The Main Idea

If you mutate data like this:

```
myData.x.y.z = 7;
// or...
myData.a.b.push(9);
```

You have no way of determining which data has changed since the previous copy has been overwritten. Instead, you need to create a new copy of `myData` and change only the parts of it that need to be changed. Then you can compare the old copy of `myData` with the new one in `shouldComponentUpdate()` using triple-equals:

```
const newData = deepCopy(myData);
newData.x.y.z = 7;
newData.a.b.push(9);
```

Unfortunately, deep copies are expensive, and sometimes impossible. You can alleviate this by only copying objects that need to be changed and by reusing the objects that haven't changed. Unfortunately, in today's JavaScript this can be cumbersome:

```
const newData = extend(myData, {
  x: extend(myData.x, {
    y: extend(myData.x.y, {z: 7}),
```

```
  },
  a: extend(myData.a, {b: myData.a.b.concat(9)})
});
```

While this is fairly performant (since it only makes a shallow copy of `log n` objects and reuses the rest), it's a big pain to write. Look at all the repetition! This is not only annoying, but also provides a large surface area for bugs.

update() {#update}

`update()` provides simple syntactic sugar around this pattern to make writing this code easier. This code becomes:

```
import update from 'react-addons-update';

const newData = update(myData, {
  x: {y: {z: {$set: 7}}},
  a: {b: {$push: [9]}}
});
```

While the syntax takes a little getting used to (though it's inspired by MongoDB's query language) there's no redundancy, it's statically analyzable and it's not much more typing than the mutative version.

The `$`-prefixed keys are called *commands*. The data structure they are "mutating" is called the *target*.

Available Commands

- `={$push: array} push()` all the items in `array` on the target.
- `={$unshift: array} unshift()` all the items in `array` on the target.
- `={$splice: array of arrays} for each item in arrays call splice()` on the target with the parameters provided by the item.
- `={$set: any} replace the target entirely.`
- `={$merge: object} merge the keys of object with the target.`
- `={$apply: function} passes in the current value to the function and updates it with the new returned value.`

Examples

Simple push

```
const initialArray = [1, 2, 3];
const newArray = update(initialArray, {$push: [4]}); // => [1, 2, 3, 4]
```

`initialArray` is still `[1, 2, 3]`.

Nested collections

```
const collection = [1, 2, {a: [12, 17, 15]}];
const newCollection = update(collection, {2: {a: {$splice: [[1, 1, 13, 14]]}}});
// => [1, 2, {a: [12, 13, 14, 15]}]
```

This accesses `collection`'s index `2`, key `a`, and does a splice of one item starting from index `1` (to remove `17`) while inserting `13` and `14`.

Updating a value based on its current one

```
const obj = {a: 5, b: 3};
const newObj = update(obj, {b: {$apply: function(x) {return x * 2;}}});
// => {a: 5, b: 6}
// This is equivalent, but gets verbose for deeply nested collections:
const newObj2 = update(obj, {b: {$set: obj.b * 2}});
```

(Shallow) Merge

```
const obj = {a: 5, b: 3};
const newObj = update(obj, {$merge: {b: 6, c: 7}}); // => {a: 5, b: 6, c: 7}
```

Add-Ons

Note:

`React.addons` entry point is deprecated as of React v15.5. The add-ons have moved to separate modules, and some of them have been deprecated.

The React add-ons are a collection of useful utility modules for building React apps. **These should be considered experimental** and tend to change more often than the core.

- `createFragment`, to create a set of externally-keyed children.

The add-ons below are in the development (unminified) version of React only:

- `Perf`, a performance profiling tool for finding optimization opportunities.
- `ReactTestUtils`, simple helpers for writing test cases.

Legacy Add-ons

The add-ons below are considered legacy and their use is discouraged. They will keep working in observable future, but there is no further development.

- `PureRenderMixin`. Use `React.PureComponent` instead.
- `shallowCompare`, a helper function that performs a shallow comparison for props and state in a component to decide if a component should update. We recommend using `React.PureComponent` instead.
- `update`. Use `kolodny/immutability-helper` instead.
- `ReactDOMFactories`, pre-configured DOM factories to make React easier to use without JSX.

Deprecated Add-ons

- `LinkedStateMixin` has been deprecated.
- `TransitionGroup` and `CSSTransitionGroup` have been deprecated in favor of [their drop-in replacements](#).

Using React with Add-ons

You can install the add-ons individually from npm (e.g. `npm install react-addons-create-fragment`) and import them:

```
import createFragment from 'react-addons-create-fragment'; // ES6
var createFragment = require('react-addons-create-fragment'); // ES5 with npm
```

When using React 15 or earlier from a CDN, you can use `react-with-addons.js` instead of `react.js`:

```
<script src="https://unpkg.com/react@15/dist/react-with-addons.js"></script>
```

The add-ons will be available via the `React.addons` global (e.g. `React.addons.TestUtils`).

[Go to TOC](#)

CDN 連結

通過 CDN 載入 React 和 React DOM。

```
<script crossorigin src="https://unpkg.com/react@18/umd/react.development.js">
</script>
<script crossorigin src="https://unpkg.com/react-dom@18/umd/react-
dom.development.js"></script>
```

以上的版本只適用於開發環境，並不適合用於線上環境。你可以在以下找到已壓縮和最佳化的 React 線上版本：

```
<script crossorigin src="https://unpkg.com/react@18/umd/react.production.min.js">
</script>
<script crossorigin src="https://unpkg.com/react-dom@18/umd/react-
dom.production.min.js"></script>
```

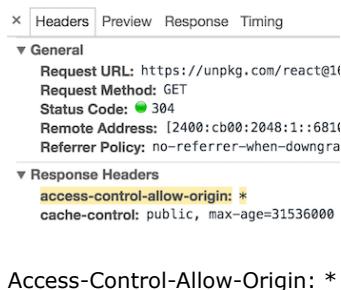
請更改版本號碼 18 來載入指定版本的 react 和 react-dom。

為什麼要使用 `crossorigin` Attribute?

如果你通過 CDN 載入 React，我們建議你保留 `crossorigin` 的設定：

```
<script crossorigin src="... "></script>
```

我們也建議你驗證你所使用的 CDN 是否設定了 `Access-Control-Allow-Origin: *` HTTP header：



Access-Control-Allow-Origin: *

這能為 React 16 或以上的版本帶來更好的錯誤處理體驗。

Code-Splitting

Bundling

大部分 React 應用程式會使用像是 [Webpack](#)、[Rollup](#) 或 [Browserify](#) 的工具來 bundle 它們的檔案。Bundle 是將 import 的檔案合併為單一的檔案過程：「Bundle」。網頁可以引入 bundle，以一次載入整個應用程式。

Bundle 是將被 import 的檔案合併成一個單一的檔案：「bundle」。這個 bundle 檔案可以被引入到網頁內來載入整個應用程式。

範例

應用程式：

```
// app.js
import { add } from './math.js';

console.log(add(16, 26)); // 42
```

```
// math.js
export function add(a, b) {
  return a + b;
}
```

Bundle：

```
function add(a, b) {
  return a + b;
}

console.log(add(16, 26)); // 42
```

注意：

你的 bundle 後的最終結果看起來會與此不同。

如果你使用 [Create React App](#)、[Next.js](#)、[Gatsby](#)，或者是類似的工具，會有一個內建的 Webpack 設定來 bundle 你的應用程式。

如果沒有的話，你需要自己設定 bundle。例如，拜訪 Webpack 文件的 [Installation](#) 和 [Getting Started](#) 指南。

Code-Splitting

Bundle 非常棒，但隨著你的應用程式成長，你的 bundle 也將會隨著增長。特別是你引入了大量的第三方函式庫。你需要隨時留意 bundle 後的程式碼，這樣你就不會得意外的讓 bundle 檔案變得太大的，以至於你的應用程式需要很長的時間才能被載入。

為了避免 bundle 的結果過大，最好的解決問題的方式是開始「split」你的 bundle。Code-Splitting 是透過由像是 Webpack、Rollup 和 Browserify（經由 factor-bundle）的 bundler 所支援的功能，它會建立多個 bundle，可以在 runtime 時動態的被載入。

Code-splitting 可以幫助你「延遲載入」目前使用者所需要的東西，這可以大幅提供你的應用程式效能。雖然你還沒有減少應用程式的程式碼總數量，但你可以避免載入使用者目前使用不到的程式碼，來減少初始載入應用程式的時間。

import() {#import}

將 code-splitting 引入到你的應用程式最好的方式是透過動態 import() 語法。

加入前：

```
import { add } from './math';
console.log(add(16, 26));
```

加入後：

```
import("./math").then(math => {
  console.log(math.add(16, 26));
});
```

當 Webpack 遇到這種語法時，它將自動的在你的應用程式啟動 code-splitting。如果你使用 Create React App 的話，它已經幫你設定好了，你可以立即的[使用它](#)。在 Next.js 也內建支援這個功能。

如果你是自行設定 Webpack，你可以閱讀 Webpack 的 [code-splitting 指南](#)。你的 Webpack 設定看起來應該像這樣。

當使用 Babel 時，你將需要確保 Babel 可以解析動態的 import 語法而不是去轉換它。你可能會需要 [@babel/plugin-syntax-dynamic-import](#)。

React.lazy {#reactlazy}

`React.lazy` 讓你 render 一個動態 import 的 component 作為正常的 component。

加入前：

```
import OtherComponent from './OtherComponent';
```

加入後：

```
const OtherComponent = React.lazy(() => import('./OtherComponent'));
```

當首次 render 這個 component 時，將會自動的載入包含 `OtherComponent` 的 bundle。

`React.lazy` 接受一個必須呼叫一個動態 `import()` 的 function。它必須回傳一個 `Promise`，`resolve` 一個包含 React component 的 `default` export 的 module。

lazy component 應在 `Suspense` component 內 render，這使我們可以在等待 lazy component 載入時，顯示一些 fallback 內容（像是一個載入的符號）。

```
import React, { Suspense } from 'react';

const OtherComponent = React.lazy(() => import('./OtherComponent'));

function MyComponent() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <OtherComponent />
      </Suspense>
    </div>
  );
}
```

`fallback` prop 接受在等待 component 載入時要 render 的任何 React element。你可以將 `Suspense` component 放在 lazy component 上方的任何位置。你甚至可以包覆多個 lazy component 到 `Suspense` component 內。

```
import React, { Suspense } from 'react';

const OtherComponent = React.lazy(() => import('./OtherComponent'));
const AnotherComponent = React.lazy(() => import('./AnotherComponent'));

function MyComponent() {
  return (
    <div>
      <Suspense fallback={<div>Loading...</div>}>
        <section>
          <OtherComponent />
          <AnotherComponent />
        </section>
      </Suspense>
    </div>
  );
}
```

避免 Fallbacks

任何 component 都可能因 render 而暫停，即使是已經向使用者顯示的 component。為了讓螢幕內容始終保持一致，如果一個已經顯示的 component suspend，React 必須將 tree 隱藏到最近的 `<Suspense>` 邊界。但是，從使用者的角度來看，這可能會讓人迷惑。

思考這個 tab switcher：

```
import React, { Suspense } from 'react';
import Tabs from './Tabs';
import Glimmer from './Glimmer';

const Comments = React.lazy(() => import('./Comments'));
const Photos = React.lazy(() => import('./Photos'));

function MyComponent() {
  const [tab, setTab] = React.useState('photos');

  function handleTabSelect(tab) {
    setTab(tab);
  };
}
```

```

return (
  <div>
    <Tabs onTabSelect={handleTabSelect} />
    <Suspense fallback={<Glimmer />}>
      {tab === 'photos' ? <Photos /> : <Comments />}
    </Suspense>
  </div>
);
}

```

在這個範例中，如果 tab 從 `'photos'` 變更為 `'comments'`，但是 `Comments` 暫停，使用者會看到一個 `Glimmer`。這很合理因為使用者不想要看到 `Photos`，`Comments` component 還沒準備 render 任何東西，React 需要保持使用者體驗一致，所以它別無選擇，只能顯示上面的 `Glimmer`。

但是，有時候這樣的使用者體驗並不理想。在特定場景下，有時候在新的 UI 準備好之前，顯示「舊」的 UI 會更好。你可以使用新的 `startTransition` API 來讓 React 執行此操作：

```

function handleTabSelect(tab) {
  startTransition(() => {
    setTab(tab);
  });
}

```

這裡，你告訴 React 設定 tab 成 `'comments'` 不是一個緊急更新，但它是一個 `transition` 所以可能需要一些時間。React 將會保持舊的 UI 交互，並且在準備好時切換成顯示 `<Comments />`。更多資訊請參考 [Transitions](#)。

錯誤邊界

如果其他的 module 載入失敗（例如，因為網路失敗），它將會觸發一個錯誤。你可以透過錯誤邊界處理這些錯誤來呈現一個好的使用者體驗和管理恢復。一旦你建立了你的錯誤邊界，你可以在任何的 lazy component 上方使用它，當網路發生錯誤時可以顯示一個錯誤狀態。

```

import React, { Suspense } from 'react';
import MyErrorBoundary from './MyErrorBoundary';

const OtherComponent = React.lazy(() => import('./OtherComponent'));
const AnotherComponent = React.lazy(() => import('./AnotherComponent'));

const MyComponent = () => (
  <div>
    <MyErrorBoundary>
      <Suspense fallback={<div>Loading...</div>}>
        <section>
          <OtherComponent />
          <AnotherComponent />
        </section>
      </Suspense>
    </MyErrorBoundary>
  </div>
);

```

基於 Route 的 Code-Splitting

在你的應用程式決定採用 code-splitting 可能有點棘手。你想要確保選擇的位置可以適當的 split bundle，但不會破壞使用者的體驗。

Route 是一個開始的好地方。Web 上大多數的人都習慣花一些時間來等待頁面的過渡。你也傾向於重新 render 一次整個頁面，所以你的使用者不能同時與頁面上的其他 element 做互動。

這裡是如何在你的應用程式使用像是 `React Router` 的函式庫與 `React.lazy` 來設定基於 route 的 code-splitting。

```
import React, { Suspense, lazy } from 'react';
import { BrowserRouter as Router, Routes, Route } from 'react-router-dom';

const Home = lazy(() => import('./routes/Home'));
const About = lazy(() => import('./routes/About'));

const App = () => (
  <Router>
    <Suspense fallback={<div>Loading...</div>}>
      <Routes>
        <Route path="/" element={<Home />} />
        <Route path="/about" element={<About />} />
      </Routes>
    </Suspense>
  </Router>
);
```

Named Exports

`React.lazy` 目前只支援 default exports。如果你想 import 的 module 使用 named export，你可以建立一個中介 module 來重新 export 它做為預設。這可以確保 tree shaking 不會 pull 無用的 component。

```
// ManyComponents.js
export const MyComponent = /* ... */;
export const MyUnusedComponent = /* ... */;
```

```
// MyComponent.js
export { MyComponent as default } from "./ManyComponents.js";
```

```
// MyApp.js
import React, { lazy } from 'react';
const MyComponent = lazy(() => import("./MyComponent.js"));
```

Codebase Overview

This section will give you an overview of the React codebase organization, its conventions, and the implementation.

If you want to [contribute to React](#) we hope that this guide will help you feel more comfortable making changes.

We don't necessarily recommend any of these conventions in React apps. Many of them exist for historical reasons and might change with time.

Top-Level Folders

After cloning the [React repository](#), you will see a few top-level folders in it:

- `packages` contains metadata (such as `package.json`) and the source code (`src` subdirectory) for all packages in the React repository. **If your change is related to the code, the `src` subdirectory of each package is where you'll spend most of your time.**
- `fixtures` contains a few small React test applications for contributors.
- `build` is the build output of React. It is not in the repository but it will appear in your React clone after you [build it](#) for the first time.

The documentation is hosted [in a separate repository from React](#).

There are a few other top-level folders but they are mostly used for the tooling and you likely won't ever encounter them when contributing.

Colocated Tests

We don't have a top-level directory for unit tests. Instead, we put them into a directory called `__tests__` relative to the files that they test.

For example, a test for `setInnerHTML.js` is located in `__tests__/setInnerHTML-test.js` right next to it.

Warnings and Invariants

The React codebase uses `console.error` to display warnings:

```
if (__DEV__) {
  console.error('Something is wrong.');
}
```

Warnings are only enabled in development. In production, they are completely stripped out. If you need to forbid some code path from executing, use `invariant` module instead:

```
var invariant = require('invariant');

invariant(
```

```
2 + 2 === 4,
'You shall not pass!'
);
```

The invariant is thrown when the `invariant` condition is `false`.

"Invariant" is just a way of saying "this condition always holds true". You can think about it as making an assertion.

It is important to keep development and production behavior similar, so `invariant` throws both in development and in production. The error messages are automatically replaced with error codes in production to avoid negatively affecting the byte size.

Development and Production

You can use `__DEV__` pseudo-global variable in the codebase to guard development-only blocks of code.

It is inlined during the compile step, and turns into `process.env.NODE_ENV !== 'production'` checks in the CommonJS builds.

For standalone builds, it becomes `true` in the unminified build, and gets completely stripped out with the `if` blocks it guards in the minified build.

```
if (__DEV__) {
  // This code will only run in development.
}
```

Flow

We recently started introducing [Flow](#) checks to the codebase. Files marked with the `@flow` annotation in the license header comment are being typechecked.

We accept pull requests [adding Flow annotations to existing code](#). Flow annotations look like this:

```
ReactRef.detachRefs = function(
  instance: ReactInstance,
  element: ReactElement | string | number | null | false,
): void {
  // ...
}
```

When possible, new code should use Flow annotations. You can run `yarn flow` locally to check your code with Flow.

Multiple Packages

React is a [monorepo](#). Its repository contains multiple separate packages so that their changes can be coordinated together, and issues live in one place.

React Core

The "core" of React includes all the top-level `React APIs`, for example:

- `React.createElement()`
- `React.Component`
- `React.Children`

React core only includes the APIs necessary to define components. It does not include the [reconciliation](#) algorithm or any platform-specific code. It is used both by React DOM and React Native components.

The code for React core is located in `packages/react` in the source tree. It is available on npm as the `react` package. The corresponding standalone browser build is called `react.js`, and it exports a global called `React`.

Renderers

React was originally created for the DOM but it was later adapted to also support native platforms with [React Native](#). This introduced the concept of "renderers" to React internals.

Renderers manage how a React tree turns into the underlying platform calls.

Renderers are also located in `packages/`:

- [React DOM Renderer](#) renders React components to the DOM. It implements [top-level ReactDOM APIs](#) and is available as `react-dom` npm package. It can also be used as standalone browser bundle called `react-dom.js` that exports a `ReactDOM` global.
- [React Native Renderer](#) renders React components to native views. It is used internally by React Native.
- [React Test Renderer](#) renders React components to JSON trees. It is used by the [Snapshot Testing](#) feature of [Jest](#) and is available as `react-test-renderer` npm package.

The only other officially supported renderer is `react-art`. It used to be in a separate [GitHub repository](#) but we moved it into the main source tree for now.

Note:

Technically the `react-native-renderer` is a very thin layer that teaches React to interact with React Native implementation. The real platform-specific code managing the native views lives in the [React Native repository](#) together with its components.

Reconcilers

Even vastly different renderers like React DOM and React Native need to share a lot of logic. In particular, the [reconciliation](#) algorithm should be as similar as possible so that declarative rendering, custom components, state, lifecycle methods, and refs work consistently across platforms.

To solve this, different renderers share some code between them. We call this part of React a "reconciler". When an update such as `setState()` is scheduled, the reconciler calls `render()` on components in the tree and mounts, updates, or unmounts them.

Reconcilers are not packaged separately because they currently have no public API. Instead, they are exclusively used by renderers such as React DOM and React Native.

Stack Reconciler

The "stack" reconciler is the implementation powering React 15 and earlier. We have since stopped using it, but it is documented in detail in the [next section](#).

Fiber Reconciler

The "fiber" reconciler is a new effort aiming to resolve the problems inherent in the stack reconciler and fix a few long-standing issues. It has been the default reconciler since React 16.

Its main goals are:

- Ability to split interruptible work in chunks.
- Ability to prioritize, rebase and reuse work in progress.
- Ability to yield back and forth between parents and children to support layout in React.
- Ability to return multiple elements from `render()`.
- Better support for error boundaries.

You can read more about React Fiber Architecture [here](#) and [here](#). While it has shipped with React 16, the async features are not enabled by default yet.

Its source code is located in `packages/react-reconciler`.

Event System

React implements a layer over native events to smooth out cross-browser differences. Its source code is located in `packages/react-dom/src/events`.

What Next?

Read the [next section](#) to learn about the pre-React 16 implementation of reconciler in more detail. We haven't documented the internals of the new reconciler yet.

Components 與 Props

Component 使你可以將 UI 拆分成獨立且可複用的程式碼，並且專注於各別程式碼的思考。本章節旨在介紹 component 的相關概念，你也可以在此參閱[詳細的 API 文件](#)。

概念上來說，component 就像是 JavaScript 的 function，它接收任意的參數（稱之為「props」）並且回傳描述畫面的 React element。

Function Component 與 Class Component

定義 component 最簡單的方法即是撰寫一個 Javascript function：

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

此 function 是一個符合規範的 React component，因為它接受一個「props」（指屬性 properties）物件且回傳一個 React element。我們稱之為 function component，因為它本身就是一個 JavaScript function。

同樣的，你也可以使用 [ES6 Class](#) 來定義 component：

```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

上述兩種 component 在 React 中是同等的。

Function 和 Class component 兩者都擁有額外的特性，我們將會在[下一個章節](#)探討。

Render 一個 Component

在此之前，我們只見過這種相當於 DOM 標籤的 React element：

```
const element = <div />;
```

不過，React element 也可以是使用者自定義的 component：

```
const element = <Welcome name="Sara" />;
```

當 React 看到由使用者定義 component 的 element 時，它將 JSX 屬性和 children 作為 single object 傳遞給該 component。我們稱這個 object 為「props」。

舉例來說，這段程式碼會在頁面上 render 出「Hello, Sara」：

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

```
const root = ReactDOM.createRoot(document.getElementById('root'));
const element = <Welcome name="Sara" />;
root.render(element);
```

[在 CodePen 上試試看吧！](#)

讓我們來複習一下這個例子發生了什麼事：

1. 我們對 `<Welcome name="Sara" />` 這個 element 呼叫了 `root.render()`。
2. React 以 `{name: 'Sara'}` 作為 props 傳入 `Welcome` component 並呼叫。
3. `Welcome` component 回傳了 `<h1>Hello, Sara</h1>` 這個 element 作為返回值。
4. React DOM 有效的將 DOM 更新為 `<h1>Hello, Sara</h1>`。

注意：Component 的字首須為大寫字母

React 會將小寫字母開頭的組件視為原始 DOM 標籤，舉例來說，`<div />` 就會被視為是 HTML 的 `div` 標籤，但是 `<Welcome />` 則是一個 component，而且需要在作用域中使用 `Welcome`。

想要了解更多關於此慣例的原因，請參閱 [JSX In Depth](#) 章節。

組合 Component

Component 可以在輸出中引用其他 component。我們可以在任何層次中抽象化相同的 component，按鈕、表單、對話框、甚至是整個畫面，在 React 應用程式中都將以 component 的方式呈現。

舉例來說，我們可以建立一個 render 多次 `Welcome` 的 `App` component：

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}

function App() {
  return (
    <div>
      <Welcome name="Sara" />
      <Welcome name="Cahal" />
      <Welcome name="Edite" />
    </div>
  );
}
```

[在 CodePen 上試試看吧！](#)

通常來說，每個 React 應用程式都有一個最高層級的 `App` component。然而，如果你將 React 結合至現存的應用程式中，你可能需要使用像 `Button` 這樣的小型 component，並由下往上，逐步應用到畫面的最高層級。

抽離 Component

別害怕將 component 拆分成更小的 component。

舉例來說，我們看這個 `Comment` 的 component：

```
function Comment(props) {
  return (
    <div className="Comment">
      <div className="UserInfo">
        <img className="Avatar"
          src={props.author.avatarUrl}
          alt={props.author.name}>
      />
      <div className="UserInfo-name">
        {props.author.name}
      </div>
    </div>
    <div className="Comment-text">
      {props.text}
    </div>
    <div className="Comment-date">
      {formatDate(props.date)}
    </div>
  </div>
);
}
```

[在 CodePen 上試試看吧！](#)

它接受 `author` (一個物件)、`text` (一個字串)、還有 `date` (一個日期) 作為它的 props。它的作用是在一個社交網站上 render 一則評論。

這個 component 可能因為太多的巢狀關係而難以更動，而且也難以複用獨立的部分。讓我們把一些 component 從中分離吧。

首先，我們將 `Avatar` 分離出來：

```
function Avatar(props) {
  return (
    <img className="Avatar"
      src={props.user.avatarUrl}
      alt={props.user.name}>
  );
}
```

這個 `Avatar` 並不需知道它會被 render 在 `Comment` 中。這是為什麼我們給他一個更為一般的名子：`user` 而不是 `author`。

我們建議從 component 的角度為 props 命名，而不是它的使用情境。

現在我們可以稍微簡化 `Comment`：

```
function Comment(props) {
  return (
    <div className="Comment">
      <div className="UserInfo">
        <Avatar user={props.author} />
        <div className="UserInfo-name">
          {props.author.name}
        </div>
      </div>
    </div>
  );
}
```

```

        </div>
    </div>
    <div className="Comment-text">
        {props.text}
    </div>
    <div className="Comment-date">
        {formatDate(props.date)}
    </div>
</div>
);
}
}

```

接下來，我們將 `UserInfo` component 也抽離出來，它會在使用者名稱旁邊 render `Avatar` component：

```

function UserInfo(props) {
    return (
        <div className="UserInfo">
            <Avatar user={props.user} />
            <div className="UserInfo-name">
                {props.user.name}
            </div>
        </div>
    );
}

```

讓我們將 `Comment` 更加簡化：

```

function Comment(props) {
    return (
        <div className="Comment">
            <UserInfo user={props.author} />
            <div className="Comment-text">
                {props.text}
            </div>
            <div className="Comment-date">
                {formatDate(props.date)}
            </div>
        </div>
    );
}

```

在 [CodePen](#) 上試試看吧！

在一開始，將 component 抽離出來可能是一件繁重的工作，但是在較大的應用程式中，建構可複用的 component 是非常值得。以經驗來說，如果一個 UI 中有一部分會被重複使用很多次（`Button`、`Panel`、`Avatar`），或者它足夠複雜（`App`、`FeedStory`、`Comment`），則可以將它提取到獨立的 component。

Props 是唯讀的

不管你使用 `function` 或是 `class` 來宣告 component，都絕不能修改自己的 props。例如這個 `sum` function：

```

function sum(a, b) {
    return a + b;
}

```

像這樣的 function 是 `Pure function` 的，因為他們並沒有改變輸入，而且相同的輸入總是回傳一樣的結果。

相反地，這個 function 並非 Pure function，因為它更改了它的參數：

```
function withdraw(account, amount) {  
  account.total -= amount;  
}
```

React 是很彈性的，但有一條嚴格的規定：

所有的 React component 都必須像 Pure function 一般保護他的 props

當然，應用程式的 UI 是動態的，而且總是隨著時間改變。在[下個章節](#)，我們會介紹一個新的概念「state」。State 可以在不違反上述規則的前提下，讓 React component 隨使用者操作、網路回應、或是其他方式改變輸出內容。

Composition vs 繼承

React 具有強大的 composition 模型，我們建議你在 component 之間使用 composition 來複用你的程式碼，而不是使用繼承。

在這個章節中，我們將考慮一些新 React 開發者常常遇到的繼承問題，並示範如何透過 composition 來解決它們。

包含

有些 component 不會提早知道它們的 children 有些什麼。對於像是 `Sidebar` 或 `Dialog` 這類通用的「box」component 特別常見。

我們建議這些 component 使用特殊的 `children` prop 將 children element 直接傳入到它們的輸出：

```
function FancyBorder(props) {
  return (
    <div className={'FancyBorder FancyBorder-' + props.color}>
      {props.children}
    </div>
  );
}
```

這讓其他的 component 透過巢狀的 JSX 將任意的 children 傳遞給它們：

```
function WelcomeDialog() {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">
        Welcome
      </h1>
      <p className="Dialog-message">
        Thank you for visiting our spacecraft!
      </p>
    </FancyBorder>
  );
}
```

[在 CodePen 試試看吧！](#)

任何在 `<FancyBorder>` JSX tag 內的內容都被作為 `children` prop 被傳遞給 `FancyBorder` component。由於 `FancyBorder` 在 `<div>` 內 render `{props.children}`，被傳遞的 element 會在最終的輸出出現。

雖然這個情況不常見，但有時候你可能需要在 component 中使用多個「hole」。在這種情況下，你可以使用你慣用的方法，而不是使用 `children`：

```
function SplitPane(props) {
  return (
    <div className="SplitPane">
      <div className="SplitPane-left">
        {props.left}
      </div>
      <div className="SplitPane-right">
```

```

        {props.right}
      </div>
    </div>
  );
}

function App() {
  return (
    <SplitPane
      left={
        <Contacts />
      }
      right={
        <Chat />
      } />
  );
}

```

[在 CodePen 上試試吧！](#)

React 的 element 像是 `<Contacts />` 和 `<Chat />` 都只是 object，所以你可以像其它任何的資料一樣，將它們作為 props 傳遞。這種方法可能會提醒你其他函式庫中的「slot」，但對於你可以在 React 中作為 prop 傳遞的內容沒有限制。

特別化

有時候，我們需要考慮 component 會不會是其他 component 的「特別情況」。例如，我們可能會說 `WelcomeDialog` 是 `Dialog` 的一個特定情況。

在 React 中，這也可以透過 composition 被實現，其中更「特別」的 component render 更多「通用」的 component，並使用 prop 對其進行設定：

```

function Dialog(props) {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">
        {props.title}
      </h1>
      <p className="Dialog-message">
        {props.message}
      </p>
    </FancyBorder>
  );
}

function WelcomeDialog() {
  return (
    <Dialog
      title="Welcome"
      message="Thank you for visiting our spacecraft!" />
  );
}

```

[在 CodePen 上試試看吧！](#)

對於使用 class 定義的 component，composition 一樣有效：

```

function Dialog(props) {
  return (
    <FancyBorder color="blue">
      <h1 className="Dialog-title">
        {props.title}
      </h1>
      <p className="Dialog-message">
        {props.message}
      </p>
      {props.children}
    </FancyBorder>
  );
}

class SignUpDialog extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.handleSignUp = this.handleSignUp.bind(this);
    this.state = {login: ''};
  }

  render() {
    return (
      <Dialog title="Mars Exploration Program"
        message="How should we refer to you?">
        <input value={this.state.login}
          onChange={this.handleChange} />
        <button onClick={this.handleSignUp}>
          Sign Me Up!
        </button>
      </Dialog>
    );
  }

  handleChange(e) {
    this.setState({login: e.target.value});
  }

  handleSignUp() {
    alert(`Welcome aboard, ${this.state.login}!`);
  }
}

```

[在 CodePen 上試試看吧！](#)

那麼關於繼承呢？

在 Facebook 中，我們使用 React 在成千上萬個 component，我們找不到任何使用案例來推薦你建立繼承結構的 component。

Prop 和 composition 提供你明確和安全的方式來自訂 component 的外觀和行為所需的靈活性。請記得，component 可以接受任意的 prop，包含 primitive value、React element，或者是 function。

如果你想要在 component 之間複用非 UI 的功能，我們建議抽離它到一個獨立的 JavaScript 模組。Component 可以 import 並使用它的 function、object，或者是 class，而不需要繼承它。

[Go to TOC](#)

條件 Render

在 React 中，你可以建立不同的 component 來封裝你需要的行為。接著，你可以根據你的應用程式的 state，來 render 其中的一部份。

React 中的條件 rendering 跟 JavaScript 一致。使用 JavaScript 中的運算子如 `if` 或者 [三元運算子](#) 來建立表示目前 state 的 element，然後讓 React 根據它們來更新 UI。

先看以下兩個 component：

```
function UserGreeting(props) {
  return <h1>Welcome back!</h1>;
}

function GuestGreeting(props) {
  return <h1>Please sign up.</h1>;
}
```

我們將會建立一個 `Greeting` component，它會根據使用者是否已登入來顯示其中之一：

```
function Greeting(props) {
  const isLoggedIn = props.isLoggedIn;
  if (isLoggedIn) {
    return <UserGreeting />;
  }
  return <GuestGreeting />;
}

const root = ReactDOM.createRoot(document.getElementById('root'));
// 試改為 isLoggedIn={true}:
root.render(<Greeting isLoggedIn={false} />);
```

[在 CodePen 上試試看吧！](#)

這範例根據 `isLoggedIn` prop 的值來 render 不同的問候語。

Element 變數

你可以用變數來儲存 element。它可以幫助你有條件地 render 一部份的 component，而保持其他輸出不變。

思考這兩個新的登入和登出按鈕 component：

```
function LoginButton(props) {
  return (
    <button onClick={props.onClick}>
      Login
    </button>
  );
}

function LogoutButton(props) {
  return (
    <button onClick={props.onClick}>
      Logout
    </button>
  );
}
```

```

    </button>
);
}

```

在下列範例，我們將建立一個名為 `LoginControl` 的 **stateful component**。

它將根據目前的 state 來 render `<LoginButton />` 或 `<LogoutButton />`。而且也會 render 前面範例的 `<Greeting />`。

```

class LoginControl extends React.Component {
  constructor(props) {
    super(props);
    this.handleLoginClick = this.handleLoginClick.bind(this);
    this.handleLogoutClick = this.handleLogoutClick.bind(this);
    this.state = {isLoggedIn: false};
  }

  handleLoginClick() {
    this.setState({isLoggedIn: true});
  }

  handleLogoutClick() {
    this.setState({isLoggedIn: false});
  }

  render() {
    const isLoggedIn = this.state.isLoggedIn;
    let button;

    if (isLoggedIn) {
      button = <LogoutButton onClick={this.handleLogoutClick} />;
    } else {
      button = <LoginButton onClick={this.handleLoginClick} />;
    }

    return (
      <div>
        <Greeting isLoggedIn={isLoggedIn} />
        {button}
      </div>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<LoginControl />);

```

[在 CodePen 上試試看吧！](#)

雖然宣告變數並使用 `if` 語句來有條件 render component 是一個不錯的方式，但有時你也想使用更簡潔的語法。在 JSX 中有以下幾種方法：

Inline If 與 `&&` 邏輯運算子

你可以透過大括號在 JSX 中嵌入表達式，包括 JavaScript 的 `&&` 邏輯運算子，可以方便 render 有條件的 element：

```

function Mailbox(props) {
  const unreadMessages = props.unreadMessages;
  return (
    <div>
      <h1>Hello!</h1>
      {unreadMessages.length > 0 &&
        <h2>
          You have {unreadMessages.length} unread messages.
        </h2>
      }
    </div>
  );
}

const messages = ['React', 'Re: React', 'Re:Re: React'];

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Mailbox unreadMessages={messages} />);

```

[在 CodePen 上試試看吧！](#)

能夠這樣做是因為在 JavaScript 中，`true && expression` 總是回傳 `expression`，而 `false && expression` 總是回傳 `false`。

所以，當條件為 `true` 時，`&&` 右側的 element 會出現在輸出中，如果是 `false`，React 會忽略並跳過它。

請注意，回傳 falsy expression 仍會導致 `&&` 之後的 element 被忽略，但依舊回傳 falsy expression，在下面的範例中，`render` 將會回傳 `<div>0</div>`。

```

render() {
  const count = 0;
  return (
    <div>
      {count && <h1>Messages: {count}</h1>}
    </div>
  );
}

```

Inline If-Else 與三元運算子

另一個有條件 render element 的方式是透過 JavaScript 的三元運算子 `condition ? true : false`。

在下面的範例，我們會用它來有條件地 render 一小段文字。

```

render() {
  const isLoggedIn = this.state.isLoggedIn;
  return (
    <div>
      The user is <b>{isLoggedIn ? 'currently' : 'not'}logged in.
    </div>
  );
}

```

它也可以被用在較複雜的表達式上，雖然不是太明顯：

```

render() {
  const isLoggedIn = this.state.isLoggedIn;
  return (
    <div>
      {isLoggedIn
        ? <LogoutButton onClick={this.handleLogoutClick} />
        : <LoginButton onClick={this.handleLoginClick} />
      }
    </div>
  );
}

```

就跟在 JavaScript 中一樣，你可以根據團隊習慣來選擇更合適的風格。還要記著如果條件變得過於複雜，也許是個好時機來[抽離 component](#) 了。

防止 Component Render

在少數的情況下，你可能希望 component 隱藏自己本身，即便它是由另一個 component 被 render。可以透過回傳 `null` 而不是它的 render 輸出。

在下面的範例中，`<WarningBanner />` 的 render 取決於 `warn` prop 的值。如果 prop 是 `false`，它就不會 render。

```

function WarningBanner(props) {
  if (!props.warn) {
    return null;
  }

  return (
    <div className="warning">
      Warning!
    </div>
  );
}

class Page extends React.Component {
  constructor(props) {
    super(props);
    this.state = {showWarning: true};
    this.handleToggleClick = this.handleToggleClick.bind(this);
  }

  handleToggleClick() {
    this.setState(state => ({
      showWarning: !state.showWarning
    }));
  }

  render() {
    return (
      <div>
        <WarningBanner warn={this.state.showWarning} />
        <button onClick={this.handleToggleClick}>
          {this.state.showWarning ? 'Hide' : 'Show'}
        </button>
      </div>
    );
  }
}

```

```
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Page />);
```

[在 CodePen 上試試看吧！](#)

在 component 中回傳 `null` 並不會影響 component 的生命週期方法。例如 `componentDidUpdate` 依然可以被呼叫。

[Go to TOC](#)

Context

Context provides a way to pass data through the component tree without having to pass props down manually at every level.

In a typical React application, data is passed top-down (parent to child) via props, but such usage can be cumbersome for certain types of props (e.g. locale preference, UI theme) that are required by many components within an application. Context provides a way to share values like these between components without having to explicitly pass a prop through every level of the tree.

- [When to Use Context](#)
- [Before You Use Context](#)
- [API](#)
 - [React.createContext](#)
 - [Context.Provider](#)
 - [Class.contextType](#)
 - [Context.Consumer](#)
 - [Context.displayName](#)
- [Examples](#)
 - [Dynamic Context](#)
 - [Updating Context from a Nested Component](#)
 - [Consuming Multiple Contexts](#)
- [Caveats](#)
- [Legacy API](#)

When to Use Context

Context is designed to share data that can be considered "global" for a tree of React components, such as the current authenticated user, theme, or preferred language. For example, in the code below we manually thread through a "theme" prop in order to style the Button component:

```
embed:context/motivation-problem.js
```

Using context, we can avoid passing props through intermediate elements:

```
embed:context/motivation-solution.js
```

Before You Use Context

Context is primarily used when some data needs to be accessible by *many* components at different nesting levels. Apply it sparingly because it makes component reuse more difficult.

If you only want to avoid passing some props through many levels, [component composition](#) is often a simpler solution than context.

For example, consider a `Page` component that passes a `user` and `avatarSize` prop several levels down so that deeply nested `Link` and `Avatar` components can read it:

```
<Page user={user} avatarSize={avatarSize} />
// ... which renders ...
<PageLayout user={user} avatarSize={avatarSize} />
// ... which renders ...
<NavigationBar user={user} avatarSize={avatarSize} />
// ... which renders ...
<Link href={user.permalink}>
  <Avatar user={user} size={avatarSize} />
</Link>
```

It might feel redundant to pass down the `user` and `avatarSize` props through many levels if in the end only the `Avatar` component really needs it. It's also annoying that whenever the `Avatar` component needs more props from the top, you have to add them at all the intermediate levels too.

One way to solve this issue **without context** is to [pass down the `Avatar` component itself](#) so that the intermediate components don't need to know about the `user` or `avatarSize` props:

```
function Page(props) {
  const user = props.user;
  const userLink = (
    <Link href={user.permalink}>
      <Avatar user={user} size={props.avatarSize} />
    </Link>
  );
  return <PageLayout userLink={userLink} />;
}

// Now, we have:
<Page user={user} avatarSize={avatarSize} />
// ... which renders ...
<PageLayout userLink={...} />
// ... which renders ...
<NavigationBar userLink={...} />
// ... which renders ...
{props.userLink}
```

With this change, only the top-most `Page` component needs to know about the `Link` and `Avatar` components' use of `user` and `avatarSize`.

This *inversion of control* can make your code cleaner in many cases by reducing the amount of props you need to pass through your application and giving more control to the root components. Such inversion, however, isn't the right choice in every case; moving more complexity higher in the tree makes those higher-level components more complicated and forces the lower-level components to be more flexible than you may want.

You're not limited to a single child for a component. You may pass multiple children, or even have multiple separate "slots" for children, [as documented here](#):

```
function Page(props) {
  const user = props.user;
  const content = <Feed user={user} />;
  const topBar = (
```

```

<NavigationBar>
  <Link href={user.permalink}>
    <Avatar user={user} size={props.avatarSize} />
  </Link>
</NavigationBar>
);
return (
  <PageLayout
    topBar={topBar}
    content={content}
  />
);
}
}

```

This pattern is sufficient for many cases when you need to decouple a child from its immediate parents. You can take it even further with `render props` if the child needs to communicate with the parent before rendering.

However, sometimes the same data needs to be accessible by many components in the tree, and at different nesting levels. Context lets you "broadcast" such data, and changes to it, to all components below. Common examples where using context might be simpler than the alternatives include managing the current locale, theme, or a data cache.

API

`React.createContext` {#reactcreatecontext}

```
const MyContext = React.createContext(defaultValue);
```

Creates a Context object. When React renders a component that subscribes to this Context object it will read the current context value from the closest matching `Provider` above it in the tree.

The `defaultValue` argument is **only** used when a component does not have a matching Provider above it in the tree. This default value can be helpful for testing components in isolation without wrapping them. Note: passing `undefined` as a Provider value does not cause consuming components to use `defaultValue`.

`Context.Provider` {#contextprovider}

```
<MyContext.Provider value={/* some value */}>
```

Every Context object comes with a Provider React component that allows consuming components to subscribe to context changes.

The Provider component accepts a `value` prop to be passed to consuming components that are descendants of this Provider. One Provider can be connected to many consumers. Providers can be nested to override values deeper within the tree.

All consumers that are descendants of a Provider will re-render whenever the Provider's `value` prop changes. The propagation from Provider to its descendant consumers (including `.contextType` and `useContext`) is not subject to the `shouldComponentUpdate` method, so the consumer is updated even when an ancestor component skips an update.

Changes are determined by comparing the new and old values using the same algorithm as `Object.is`.

Note

The way changes are determined can cause some issues when passing objects as `value`: see [Caveats](#).

`Class.contextType` [\[#classcontexttype\]](#)

```
class MyClass extends React.Component {
  componentDidMount() {
    let value = this.context;
    /* perform a side-effect at mount using the value of MyContext */
  }
  componentDidUpdate() {
    let value = this.context;
    /* ... */
  }
  componentWillUnmount() {
    let value = this.context;
    /* ... */
  }
  render() {
    let value = this.context;
    /* render something based on the value of MyContext */
  }
}
MyClass.contextType = MyContext;
```

The `contextType` property on a class can be assigned a Context object created by `React.createContext()`. Using this property lets you consume the nearest current value of that Context type using `this.context`. You can reference this in any of the lifecycle methods including the render function.

Note:

You can only subscribe to a single context using this API. If you need to read more than one see [Consuming Multiple Contexts](#).

If you are using the experimental [public class fields syntax](#), you can use a **static** class field to initialize your `contextType`.

```
class MyClass extends React.Component {
  static contextType = MyContext;
  render() {
    let value = this.context;
    /* render something based on the value */
  }
}
```

Context.Consumer {#contextconsumer}

```
<MyContext.Consumer>
  {value => /* render something based on the context value */}
</MyContext.Consumer>
```

A React component that subscribes to context changes. Using this component lets you subscribe to a context within a [function component](#).

Requires a [function as a child](#). The function receives the current context value and returns a React node. The `value` argument passed to the function will be equal to the `value` prop of the closest Provider for this context above in the tree. If there is no Provider for this context above, the `value` argument will be equal to the `defaultValue` that was passed to `createContext()`.

Note

For more information about the 'function as a child' pattern, see [render props](#).

Context.displayName {#contextdisplayname}

Context object accepts a `displayName` string property. React DevTools uses this string to determine what to display for the context.

For example, the following component will appear as `MyDisplayName` in the DevTools:

```
const MyContext = React.createContext(/* some value */);
MyContext.displayName = 'MyDisplayName';

<MyContext.Provider> // "MyDisplayName.Provider" in DevTools
<MyContext.Consumer> // "MyDisplayName.Consumer" in DevTools
```

Examples

Dynamic Context

A more complex example with dynamic values for the theme:

theme-context.js `embed:context/theme-detailed-theme-context.js`

themed-button.js `embed:context/theme-detailed-themed-button.js`

app.js `embed:context/theme-detailed-app.js`

Updating Context from a Nested Component

It is often necessary to update the context from a component that is nested somewhere deeply in the component tree. In this case you can pass a function down through the context to allow consumers to update the context:

theme-context.js [embed:context/updating-nested-context-context.js](#)

theme-toggler-button.js [embed:context/updating-nested-context-theme-toggler-button.js](#)

app.js [embed:context/updating-nested-context-app.js](#)

Consuming Multiple Contexts

To keep context re-rendering fast, React needs to make each context consumer a separate node in the tree.

[embed:context/multiple-contexts.js](#)

If two or more context values are often used together, you might want to consider creating your own render prop component that provides both.

Caveats

Because context uses reference identity to determine when to re-render, there are some gotchas that could trigger unintentional renders in consumers when a provider's parent re-renders. For example, the code below will re-render all consumers every time the Provider re-renders because a new object is always created for `value`:

[embed:context/reference-caveats-problem.js](#)

To get around this, lift the value into the parent's state:

[embed:context/reference-caveats-solution.js](#)

Legacy API

Note

React previously shipped with an experimental context API. The old API will be supported in all 16.x releases, but applications using it should migrate to the new version. The legacy API will be removed in a future major React version. Read the [legacy context docs here](#).

建立全新的 React 應用程式

使用整合的 toolchain 以得到最佳的使用者和開發者體驗。

本頁將介紹一些受歡迎的 React toolchain，它們有助於以下的工作：

- 擴大文件和 component 的規模。
- 使用來至 npm 的第三方函式庫。
- 偵測早期常見的錯誤。
- 實時在開發環境裡編輯 CSS 和 JS。
- 最佳化線上環境輸出。

本頁所建議的 toolchain 都不需要進行任何設定就能開始使用。

你可能不需要 Toolchain

如果你沒有碰到以上所提到的問題，或者還沒有習慣使用 JavaScript 的工具，你可以考慮在 HTML 網頁裡加上 `<script>` 標籤來使用 React，以及使用可選的 JSX。

這也是將 React 加上到現存網頁裡最簡單的方法。當你覺得使用大型的 toolchain 對你有幫助，你也可以隨時把它們加入！

推薦的 Toolchain

React 團隊主要推薦以下的方案：

- 如果你正在學習 React 或建立全新的 single-page 應用程式，請使用 Create React App。
- 如果你正在建立一個使用 Node.js 的 server-rendered 網頁，請使用 Next.js。
- 如果你正在建立一個靜態內容的網頁，請使用 Gatsby。
- 如果你正在建立一個 component 函式庫或與現存程式碼倉庫進行接軌，請使用更靈活的 Toolchain。

Create React App

Create React App 是一個適合學習 React 的環境，而且也是使用 React 建立一個全新的 single-page 應用程式的最佳方法。

它會為你設定好開發環境，以便你能夠使用最新的 JavaScript 特性，提供良好的開發者體驗，並且為線上環境最佳化你的應用程式。你需要在你的機器上安裝 `Node >= 14.0.0` and `npm >= 5.6`。要建立項目，請執行：

```
npx create-react-app my-app  
cd my-app  
npm start
```

提示

npx 不是拼寫錯誤 —— 它是一個 npm 5.2+ 附帶的 package 執行器。

Create React App 並不會處理 backend 邏輯或資料庫；它只會建立一個 frontend build pipeline，以便你配合任何 backend 來使用。基本上，它是使用 [Babel](#) 和 [webpack](#)，但你不需要了解任何關於它們的細節。

當你準備好發佈到線上環境，執行 `npm run build` 會在 `build` 文件夾裡建立一個你的應用程式的最佳化版本，你可以從 Create React App 的 [README](#) 和 [使用者指南](#) 了解更多資訊。

Next.js

[Next.js](#) 是一個受歡迎和輕量的框架，用於使用 React 所建立的**靜態**和**server-rendered**的應用程式。它自身已包括了**styling**和**routing**的方案，而且它假設你在使用 [Node.js](#) 作為伺服器環境。

從 [Next.js 的官方指南](#) 了解更多。

Gatsby

[Gatsby](#) 是使用 React 建立**靜態網頁**的最佳方法。它讓你使用 React component，但會輸出 pre-rendered HTML 和 CSS 來保證最快的載入時間。

從 [Gatsby 的官方指南](#)和 [starter kit 範例集](#) 了解更多。

更靈活的 Toolchain

以下的 toolchain 會提供更多的靈活性和選擇。我們推薦給比較有經驗的使用者：

- [Neutrino](#) 結合了 [webpack](#) 強大的功能與簡單的預設，並包括了 [React 應用程式](#)和 [React component](#) 的預設。
- [Nx](#) 是一套用於 full-stack monorepo 開發的 toolkit，內建支援 React、Next.js、Express 以及更多。
- [Parcel](#) 是一個快速、零設定的網路應用程式 bundler，並且可以配合 React 一起使用。
- [Razzle](#) 是一個不需要設定的 server-rendering 框架，但它比 Next.js 提供更多的靈活性。

從零開始建立 Toolchain

一個 JavaScript 的建立 toolchain 通常包括：

- 一個 **package 管理員**，例如 [Yarn](#) 或 [npm](#)。它能讓你充分利用數量龐大的第三方 package，並且輕鬆的安裝或更新它們。
- 一個 **bundler**，例如 [webpack](#) 或 [Parcel](#)。它能讓你編寫模組化的程式碼，並將它們組合成小小的 package 以最佳化載入時間。
- 一個 **compiler**，例如 [Babel](#)。它能讓你編寫現代 JavaScript 程式碼，並可以在舊版本的瀏覽器裡使用。

如果你傾向從零開始設定屬於自己的 JavaScript toolchain，請[查看這指南](#)，它會重新建立一些 Create React App 的功能。

別忘了確保你的自訂 toolchain 有為線上環境進行正確的設定。

react

[Go to TOC](#)

Cross-origin Errors

Note:

The following section applies only to the development mode of React. Error handling in production mode is done with regular try/catch statements.

In [development mode](#), React uses a global `error` event handler to preserve the "pause on exceptions" behavior of browser DevTools. It also logs errors to the developer console.

If an error is thrown from a [different origin](#) the browser will mask its details and React will not be able to log the original error message. This is a security precaution taken by browsers to avoid leaking sensitive information.

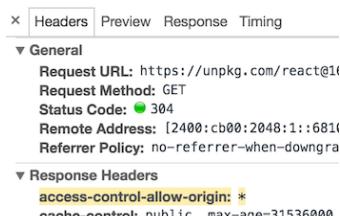
You can simplify the development/debugging process by ensuring that errors are thrown with a same-origin policy. Below are some common causes of cross-origin errors and ways to address them.

CDN

When loading React (or other libraries that might throw errors) from a CDN, add the `crossorigin` attribute to your `<script>` tags:

```
<script crossorigin src="..."></script>
```

Also ensure the CDN responds with the `Access-Control-Allow-Origin: *` HTTP header:



`Access-Control-Allow-Origin: *`

Webpack

Source maps

Some JavaScript bundlers may wrap the application code with `eval` statements in development. (For example Webpack will do this if `devtool` is set to any value containing the word "eval".) This may cause errors to be treated as cross-origin.

If you use Webpack, we recommend using the `cheap-module-source-map` setting in development to avoid this problem.

Code splitting

If your application is split into multiple bundles, these bundles may be loaded using JSONP. This may cause errors thrown in the code of these bundles to be treated as cross-origin.

To resolve this, use the `crossOriginLoading` setting in development to add the `crossorigin` attribute to the `<script>` tags generated for the JSONP requests.

[Go to TOC](#)

Design Principles

We wrote this document so that you have a better idea of how we decide what React does and what React doesn't do, and what our development philosophy is like. While we are excited to see community contributions, we are not likely to choose a path that violates one or more of these principles.

Note:

This document assumes a strong understanding of React. It describes the design principles of *React itself*, not React components or applications.

For an introduction to React, check out [Thinking in React](#) instead.

Composition

The key feature of React is composition of components. Components written by different people should work well together. It is important to us that you can add functionality to a component without causing rippling changes throughout the codebase.

For example, it should be possible to introduce some local state into a component without changing any of the components using it. Similarly, it should be possible to add some initialization and teardown code to any component when necessary.

There is nothing "bad" about using state or lifecycle methods in components. Like any powerful feature, they should be used in moderation, but we have no intention to remove them. On the contrary, we think they are integral parts of what makes React useful. We might enable [more functional patterns](#) in the future, but both local state and lifecycle methods will be a part of that model.

Components are often described as "just functions" but in our view they need to be more than that to be useful. In React, components describe any composable behavior, and this includes rendering, lifecycle, and state. Some external libraries like [Relay](#) augment components with other responsibilities such as describing data dependencies. It is possible that those ideas might make it back into React too in some form.

Common Abstraction

In general we [resist adding features](#) that can be implemented in userland. We don't want to bloat your apps with useless library code. However, there are exceptions to this.

For example, if React didn't provide support for local state or lifecycle methods, people would create custom abstractions for them. When there are multiple abstractions competing, React can't enforce or take advantage of the properties of either of them. It has to work with the lowest common denominator.

This is why sometimes we add features to React itself. If we notice that many components implement a certain feature in incompatible or inefficient ways, we might prefer to bake it into React. We don't do it lightly. When we do it, it's because we are confident that raising the abstraction level benefits the whole ecosystem. State, lifecycle methods, cross-browser event normalization are good examples of this.

We always discuss such improvement proposals with the community. You can find some of those discussions by the "[big picture](#)" label on the React issue tracker.

Escape Hatches

React is pragmatic. It is driven by the needs of the products written at Facebook. While it is influenced by some paradigms that are not yet fully mainstream such as functional programming, staying accessible to a wide range of developers with different skills and experience levels is an explicit goal of the project.

If we want to deprecate a pattern that we don't like, it is our responsibility to consider all existing use cases for it and [educate the community about the alternatives](#) before we deprecate it. If some pattern that is useful for building apps is hard to express in a declarative way, we will [provide an imperative API](#) for it. If we can't figure out a perfect API for something that we found necessary in many apps, we will [provide a temporary subpar working API](#) as long as it is possible to get rid of it later and it leaves the door open for future improvements.

Stability

We value API stability. At Facebook, we have more than 50 thousand components using React. Many other companies, including [Twitter](#) and [Airbnb](#), are also heavy users of React. This is why we are usually reluctant to change public APIs or behavior.

However we think stability in the sense of "nothing changes" is overrated. It quickly turns into stagnation. Instead, we prefer the stability in the sense of "It is heavily used in production, and when something changes, there is a clear (and preferably automated) migration path."

When we deprecate a pattern, we study its internal usage at Facebook and add deprecation warnings. They let us assess the impact of the change. Sometimes we back out if we see that it is too early, and we need to think more strategically about getting the codebases to the point where they are ready for this change.

If we are confident that the change is not too disruptive and the migration strategy is viable for all use cases, we release the deprecation warning to the open source community. We are closely in touch with many users of React outside of Facebook, and we monitor popular open source projects and guide them in fixing those deprecations.

Given the sheer size of the Facebook React codebase, successful internal migration is often a good indicator that other companies won't have problems either. Nevertheless sometimes people point out additional use cases we haven't thought of, and we add escape hatches for them or rethink our approach.

We don't deprecate anything without a good reason. We recognize that sometimes deprecations warnings cause frustration but we add them because deprecations clean up the road for the improvements and new features that we and many people in the community consider valuable.

For example, we added a [warning about unknown DOM props](#) in React 15.2.0. Many projects were affected by this. However fixing this warning is important so that we can introduce the support for [custom attributes](#) to React. There is a reason like this behind every deprecation that we add.

When we add a deprecation warning, we keep it for the rest of the current major version, and [change the behavior in the next major version](#). If there is a lot of repetitive manual work involved, we release a [codemod](#) script that automates most of the change. Codemods enable us to move forward without stagnation in a massive codebase, and we encourage you to use them as well.

You can find the codemods that we released in the [react-codemod](#) repository.

Interoperability

We place high value in interoperability with existing systems and gradual adoption. Facebook has a massive non-React codebase. Its website uses a mix of a server-side component system called XHP, internal UI libraries that came before React, and React itself. It is important to us that any product team can [start using React for a small feature](#) rather than rewrite their code to bet on it.

This is why React provides escape hatches to work with mutable models, and tries to work well together with other UI libraries. You can wrap an existing imperative UI into a declarative component, and vice versa. This is crucial for gradual adoption.

Scheduling

Even when your components are described as functions, when you use React you don't call them directly. Every component returns a [description of what needs to be rendered](#), and that description may include both user-written components like `<LikeButton>` and platform-specific components like `<div>`. It is up to React to "unroll" `<LikeButton>` at some point in the future and actually apply changes to the UI tree according to the render results of the components recursively.

This is a subtle distinction but a powerful one. Since you don't call that component function but let React call it, it means React has the power to delay calling it if necessary. In its current implementation React walks the tree recursively and calls render functions of the whole updated tree during a single tick. However in the future it might start [delaying some updates to avoid dropping frames](#).

This is a common theme in React design. Some popular libraries implement the "push" approach where computations are performed when the new data is available. React, however, sticks to the "pull" approach where computations can be delayed until necessary.

React is not a generic data processing library. It is a library for building user interfaces. We think that it is uniquely positioned in an app to know which computations are relevant right now and which are not.

If something is offscreen, we can delay any logic related to it. If data is arriving faster than the frame rate, we can coalesce and batch updates. We can prioritize work coming from user interactions (such as an animation caused by a button click) over less important background work (such as rendering new content just loaded from the network) to avoid dropping frames.

To be clear, we are not taking advantage of this right now. However the freedom to do something like this is why we prefer to have control over scheduling, and why `setState()` is asynchronous. Conceptually, we think of it as "scheduling an update".

The control over scheduling would be harder for us to gain if we let the user directly compose views with a "push" based paradigm common in some variations of [Functional Reactive Programming](#). We want to own the "glue" code.

It is a key goal for React that the amount of the user code that executes before yielding back into React is minimal. This ensures that React retains the capability to schedule and split work in chunks according to what it knows about the UI.

There is an internal joke in the team that React should have been called "Schedule" because React does not want to be fully "reactive".

Developer Experience

Providing a good developer experience is important to us.

For example, we maintain [React DevTools](#) which let you inspect the React component tree in Chrome and Firefox. We have heard that it brings a big productivity boost both to the Facebook engineers and to the community.

We also try to go an extra mile to provide helpful developer warnings. For example, React warns you in development if you nest tags in a way that the browser doesn't understand, or if you make a common typo in the API. Developer warnings and the related checks are the main reason why the development version of React is slower than the production version.

The usage patterns that we see internally at Facebook help us understand what the common mistakes are, and how to prevent them early. When we add new features, we try to anticipate the common mistakes and warn about them.

We are always looking out for ways to improve the developer experience. We love to hear your suggestions and accept your contributions to make it even better.

Debugging

When something goes wrong, it is important that you have breadcrumbs to trace the mistake to its source in the codebase. In React, props and state are those breadcrumbs.

If you see something wrong on the screen, you can open React DevTools, find the component responsible for rendering, and then see if the props and state are correct. If they are, you know that the problem is in the component's `render()` function, or some function that is called by `render()`. The problem is isolated.

If the state is wrong, you know that the problem is caused by one of the `setState()` calls in this file. This, too, is relatively simple to locate and fix because usually there are only a few `setState()` calls in a single file.

If the props are wrong, you can traverse the tree up in the inspector, looking for the component that first "poisoned the well" by passing bad props down.

This ability to trace any UI to the data that produced it in the form of current props and state is very important to React. It is an explicit design goal that state is not "trapped" in closures and combinators, and is available to React directly.

While the UI is dynamic, we believe that synchronous `render()` functions of props and state turn debugging from guesswork into a boring but finite procedure. We would like to preserve this constraint in React even though it makes some use cases, like complex animations, harder.

Configuration

We find global runtime configuration options to be problematic.

For example, it is occasionally requested that we implement a function like `React.configure(options)` or `React.register(component)`. However this poses multiple problems, and we are not aware of good solutions to them.

What if somebody calls such a function from a third-party component library? What if one React app embeds another React app, and their desired configurations are incompatible? How can a third-party component specify that it requires a particular configuration? We think that global configuration doesn't work well with composition. Since composition is central to React, we don't provide global configuration in code.

We do, however, provide some global configuration on the build level. For example, we provide separate development and production builds. We may also [add a profiling build](#) in the future, and we are open to considering other build flags.

Beyond the DOM

We see the value of React in the way it allows us to write components that have fewer bugs and compose together well. DOM is the original rendering target for React but [React Native](#) is just as important both to Facebook and the community.

Being renderer-agnostic is an important design constraint of React. It adds some overhead in the internal representations. On the other hand, any improvements to the core translate across platforms.

Having a single programming model lets us form engineering teams around products instead of platforms. So far the tradeoff has been worth it for us.

Implementation

We try to provide elegant APIs where possible. We are much less concerned with the implementation being elegant. The real world is far from perfect, and to a reasonable extent we prefer to put the ugly code into the library if it means the user does not have to write it. When we evaluate new code, we are looking for an implementation that is correct, performant and affords a good developer experience. Elegance is secondary.

We prefer boring code to clever code. Code is disposable and often changes. So it is important that it [doesn't introduce new internal abstractions unless absolutely necessary](#). Verbose code that is easy to move around, change and remove is preferred to elegant code that is prematurely abstracted and hard to change.

Optimized for Tooling

Some commonly used APIs have verbose names. For example, we use `componentDidMount()` instead of `didMount()` or `onMount()`. This is [intentional](#). The goal is to make the points of interaction with the library highly visible.

In a massive codebase like Facebook, being able to search for uses of specific APIs is very important. We value distinct verbose names, and especially for the features that should be used sparingly. For example, `dangerouslySetInnerHTML` is hard to miss in a code review.

Optimizing for search is also important because of our reliance on [codemods](#) to make breaking changes. We want it to be easy and safe to apply vast automated changes across the codebase, and unique verbose names help us achieve this. Similarly, distinctive names make it easy to write custom [lint rules](#) about using React without worrying about potential false positives.

[JSX](#) plays a similar role. While it is not required with React, we use it extensively at Facebook both for aesthetic and pragmatic reasons.

In our codebase, JSX provides an unambiguous hint to the tools that they are dealing with a React element tree. This makes it possible to add build-time optimizations such as [hoisting constant elements](#), safely lint and codemod internal component usage, and [include JSX source location](#) into the warnings.

Dogfooding

We try our best to address the problems raised by the community. However we are likely to prioritize the issues that people are *also* experiencing internally at Facebook. Perhaps counter-intuitively, we think this is the main reason why the community can bet on React.

Heavy internal usage gives us the confidence that React won't disappear tomorrow. React was created at Facebook to solve its problems. It brings tangible business value to the company and is used in many of its products. [Dogfooding](#) it means that our vision stays sharp and we have a focused direction going forward.

This doesn't mean that we ignore the issues raised by the community. For example, we added support for [web components](#) and [SVG](#) to React even though we don't rely on either of them internally. We are actively [listening to your pain points](#) and [address them](#) to the best of our ability. The community is what makes React special to us, and we are honored to contribute back.

After releasing many open source projects at Facebook, we have learned that trying to make everyone happy at the same time produced projects with poor focus that didn't grow well. Instead, we found that picking a small audience and focusing on making them happy brings a positive net effect. That's exactly what we did with React, and so far solving the problems encountered by Facebook product teams has translated well to the open source community.

The downside of this approach is that sometimes we fail to give enough focus to the things that Facebook teams don't have to deal with, such as the "getting started" experience. We are acutely aware of this, and we are thinking of how to improve in a way that would benefit everyone in the community without making the same mistakes we did with open source projects before.

錯誤邊界

在過去，component 裡 JavaScript 的錯誤常常會破壞 React 的內部 state，並使它在下次 render 的時候發生神秘的錯誤。這些錯誤總是被應用程式的程式碼裡更早發生的錯誤所導致，但 React 並沒有提供在 component 裡優雅處理它們的方式，而且也無法從錯誤中恢復。

引入錯誤邊界

一個介面裡的某一個 JavaScript 的錯誤不應該毀了整個應用程式。為了替 React 使用者解決這個問題，React 16 引入了一個新的概念：「錯誤邊界」。

錯誤邊界是一個 React component，它捕捉了任何在它的 **child component tree** 裡發生的 JavaScript 的錯誤，記錄那些錯誤，然後顯示在一個 **fallback** 的使用介面，而非讓整個 component tree 崩壞。錯誤邊界會在 render 的時候、在生命週期函式內、以及底下整個 component tree 裡的 constructor 內捕捉錯誤。

注意

錯誤邊界不會在以下情況捕捉錯誤：

- Event handlers ([學習更多](#))
- 非同步的程式碼 (例如 `setTimeout` 或 `requestAnimationFrame` callback)
- Server side rendering
- 在錯誤邊界裡丟出的錯誤 (而不是在它底下的 children)

一個 class component 如果定義了 `static getDerivedStateFromError()` 或 `componentDidCatch()` 其中一種（或兩種都有）生命週期，它就會變成錯誤邊界。在錯誤被丟出去之後，我們使用 `static getDerivedStateFromError()` 來 render fallback 的 UI，以及使用 `componentDidCatch()` 來記錄錯誤的資訊。

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    // 更新 state 以至於下一個 render 會顯示 fallback UI
    return { hasError: true };
  }

  componentDidCatch(error, errorInfo) {
    // 你也可以把錯誤記錄到一個錯誤回報系統服務
    logErrorToMyService(error, errorInfo);
  }

  render() {
    if (this.state.hasError) {
      // 你可以 render 任何客製化的 fallback UI
      return <h1>Something went wrong.</h1>;
    }
  }
}
```

```

    return this.props.children;
}
}

```

然後你就可以把它當成一般的 component 來使用：

```

<ErrorBoundary>
  <MyWidget />
</ErrorBoundary>

```

錯誤邊界就如同 JavaScript 的 `catch {}`，但它是給 component 使用的。只有 class component 可以成為錯誤邊界。實務上，大部分的時間你只會想要宣告錯誤邊界 component 一次，然後在你的應用程式裡重複使用它。

要注意錯誤邊界只會捕捉它底下 **component tree** 裡的 **component** 的錯誤。錯誤邊界無法捕捉它自己本身的錯誤。如果一個錯誤邊界在 render 錯誤訊息的時候失敗了，這個錯誤會被傳遞到在它之上最近的錯誤邊界。這個也與 JavaScript 的 `catch {}` 的運作方式類似。

Live Demo

[查看這個宣告與使用錯誤邊界的範例。](#)

該把錯誤邊界放在哪裡

錯誤邊界的精確度取決於你自己。你可以把它包在最上層的 route component 藉以顯示「發生了一些錯誤」的訊息給使用者，就如同 server-side framework 裡常常處理錯誤的方式。你也可以把它包在個別的小工具外，藉以保護它們不受應用程式裡發生的其他錯誤的影響。

對於未捕捉到的錯誤的新行為

這個改變有重要的意義。在 **React 16**，沒有被錯誤邊界所捕捉到的錯誤會 **unmount** 整個 **React component tree**。

我們為了這個決定辯論過，但在我們的經驗裡，留下壞掉的 UI 比完全移除它更糟。舉例來說，在像 Messenger 一樣的產品裡，留下壞掉的 UI 可能會導致某人傳送訊息給錯誤的對象。相似地，在支付軟體裡，顯示錯誤的金額比 render 空白畫面來得更糟。

這個改變代表著，如果你遷移到 React 16，你有可能會發掘出應用程式裡以前沒注意過但已經存在的錯誤。加上錯誤邊界使你在錯誤發生時能夠提供更好的使用者體驗。

例如，Facebook Messenger 用分開的錯誤邊界包住了側欄位的內容、資訊面板、對話紀錄、和訊息輸入欄。如果某個在其中一個 UI 裡的 component 壞了，其他的部分仍會保持能夠互動的狀態。

我們也鼓勵你使用 JS 的錯誤回報服務（或建立一個你自己的服務），這樣你可以從上線的程式裡學習未處理的 exception 並修理它們。

Component Stack Traces

React 16 把所有發生在 render 時的錯誤在開發時印出在 console 裡，即使應用程式不小心吞掉了這些錯誤。除了錯誤訊息和 JavaScript 的 stack 以外，它也提供了 component stack trace。現在你可以看到錯誤在哪個 component 裡發生的確切位置：

```
▶ React caught an error thrown by BuggyCounter. You should fix this error in your code. react-dom.development.js:7708
React will try to recreate this component tree from scratch using the error boundary you provided, ErrorBoundary.

Error: I crashed!

The error is located at:
in BuggyCounter (created by App)
in ErrorBoundary (created by App)
in div (created by App)
in App
```

你也可以在 component stack trace 裡看見檔案名稱和行數。這個在 [Create React App](#) 裡是預設行為：

```
▶ React caught an error thrown by BuggyCounter. You should fix this error in your code. react-dom.development.js:7708
React will try to recreate this component tree from scratch using the error boundary you provided, ErrorBoundary.

Error: I crashed!

The error is located at:
in BuggyCounter (at App.js:26)
in ErrorBoundary (at App.js:21)
in div (at App.js:8)
in App (at index.js:5)
```

如果你沒有使用 Create React App，你可以手動在 Babel 設定加上[這個 plugin](#)。注意它是被設計用來在開發模式使用的，且**必須在正式環境被關掉**。

注意

在 stack trace 裡顯示的 component 名稱是由 `Function.name` attribute 所決定的。如果你支援沒有原生提供它的舊瀏覽器和裝置（例如 IE 11），試著考慮把 `Function.name` polyfill 到你的應用程式，例如 `function.name-polyfill`。或著，你可以另外在你所有的 component 裡設定 `displayName` attribute。

那 `try/catch` 呢？

`try / catch` 很棒，但它只作用在命令式程式碼 (imperative code)：

```
try {
  showButton();
} catch (error) {
  // ...
}
```

然而，React component 是宣告式 (declarative) 的，且指明了什麼必須被 render：

```
<Button />
```

錯誤邊界保有了 React 宣告式的天性，且如你所預期的運行。例如，即使在某個 tree 裡很深的地方被 `setState` 所導致的 `componentDidUpdate` 的錯誤，它仍然會正確的被傳遞到最近的錯誤邊界。

那 Event Handler 呢？

錯誤邊界不會捕捉 event handler 裡所發生的錯誤。

React 不需要從 event handler 裡發生的錯誤恢復。不像 render 和其他生命週期的函式一樣，event handler 不會發生在 render 的時候。所以如果它們丟出錯誤，React 仍然知道該顯示什麼在畫面上。

如果你需要捕捉一個 event handler 裡的錯誤，只要使用一般 JavaScript 的 `try / catch` 就可以了：

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = { error: null };
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    try {
      // 做某些可以拋出錯誤的事情
    } catch (error) {
      this.setState({ error });
    }
  }

  render() {
    if (this.state.error) {
      return <h1>Caught an error.</h1>
    }
    return <button onClick={this.handleClick}>Click Me</button>
  }
}
```

注意以上的範例是用來示範一般 JavaScript 的行為，並沒有用到錯誤邊界。

從 React 15 發生的名稱改變

React 15 用不同的函式名稱支援了非常有限的錯誤邊界功能：`unstable_handleError`。這個函式不能用了，而且從 16 beta 開始，你需要將它改成 `componentDidCatch`。

我們為這個改變提供了 [codemod](#) 來自動遷移你的程式碼。

Error Decoder

In the minified production build of React, we avoid sending down full error messages in order to reduce the number of bytes sent over the wire.

We highly recommend using the development build locally when debugging your app since it tracks additional debug info and provides helpful warnings about potential problems in your apps, but if you encounter an exception while using the production build, this page will reassemble the original text of the error.

AJAX 和 APIs

AJAX 怎麼用？

你可以使用任何你喜歡的 AJAX 函式庫來與 React 搭配。一些流行的函式庫有：[Axios](#)、[jQuery AJAX](#) 以及瀏覽器內建的 `window.fetch`。

應該在 Component 的哪個生命週期中使用 AJAX？

你應該在 `componentDidMount` 的生命週期方法內，使用 AJAX 呼叫來填充資料。如此一來，你可以在收到資料時，使用 `setState` 來更新 Component。

範例：利用 AJAX 的回傳值來設定狀態

你應該在 `componentDidMount` 的生命週期方法內，使用 AJAX 呼叫來填充資料。如此一來，你可以在收到資料時，使用 `setState` 來更新 Component。

假設 API 回傳的 JSON 物件如下：

```
{
  "items": [
    { "id": 1, "name": "Apples", "price": "$2" },
    { "id": 2, "name": "Peaches", "price": "$5" }
  ]
}
```

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      error: null,
      isLoading: false,
      items: []
    };
  }

  componentDidMount() {
    fetch("https://api.example.com/items")
      .then(res => res.json())
      .then(
        (result) => {
          this.setState({
            isLoading: true,
            items: result.items
          });
        },
        // Note: it's important to handle errors here
        // instead of a catch() block so that we don't swallow
        // exceptions from actual bugs in components.
        (error) => {
          this.setState({
            isLoading: true,
            error
          });
        }
      );
  }
}
```

```

        )
    }

    render() {
        const { error, isLoading, items } = this.state;
        if (error) {
            return <div>Error: {error.message}</div>;
        } else if (!isLoading) {
            return <div>Loading...</div>;
        } else {
            return (
                <ul>
                    {items.map(item => (
                        <li key={item.id}>
                            {item.name} {item.price}
                        </li>
                    ))}
                </ul>
            );
        }
    }
}

```

Here is the equivalent with [Hooks](#):

```

function MyComponent() {
    const [error, setError] = useState(null);
    const [isLoading, setIsLoaded] = useState(false);
    const [items, setItems] = useState([]);

    // Note: the empty deps array [] means
    // this useEffect will run once
    // similar to componentDidMount()
    useEffect(() => {
        fetch("https://api.example.com/items")
            .then(res => res.json())
            .then(
                (result) => {
                    setIsLoaded(true);
                    setItems(result);
                },
                // Note: it's important to handle errors here
                // instead of a catch() block so that we don't swallow
                // exceptions from actual bugs in components.
                (error) => {
                    setIsLoaded(true);
                    setError(error);
                }
            )
    }, [])

    if (error) {
        return <div>Error: {error.message}</div>;
    } else if (!isLoading) {
        return <div>Loading...</div>;
    } else {
        return (
            <ul>
                {items.map(item => (
                    <li key={item.id}>
                        {item.name} {item.price}

```

```
</li>
});}
</ul>
);
}
}
```

Babel、JSX 和構建步驟

我需要在 React 中使用 JSX 嗎？

不需要！請參考[沒有 JSX 的 React](#)來了解更多資訊。

我需要在 React 裡使用 ES6 (+) 嗎？

不需要！請參考[沒有 ES6 的 React](#)來了解更多資訊。

如何在 JSX 中編寫註解？

```
<div>
  /* Comment goes here */
  Hello, {name}!
</div>
```

```
<div>
  /* It also works
   for multi-line comments. */
  Hello, {name}!
</div>
```

Passing Functions to Components

How do I pass an event handler (like onClick) to a component?

Pass event handlers and other functions as props to child components:

```
<button onClick={this.handleClick}>
```

If you need to have access to the parent component in the handler, you also need to bind the function to the component instance (see below).

How do I bind a function to a component instance?

There are several ways to make sure functions have access to component attributes like `this.props` and `this.state`, depending on which syntax and build steps you are using.

Bind in Constructor (ES2015)

```
class Foo extends Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
  }
  handleClick() {
    console.log('Click happened');
  }
  render() {
    return <button onClick={this.handleClick}>Click Me</button>;
  }
}
```

Class Properties (ES2022)

```
class Foo extends Component {
  handleClick = () => {
    console.log('Click happened');
  };
  render() {
    return <button onClick={this.handleClick}>Click Me</button>;
  }
}
```

Bind in Render

```
class Foo extends Component {
  handleClick() {
    console.log('Click happened');
  }
  render() {
    return <button onClick={this.handleClick.bind(this)}>Click Me</button>;
  }
}
```

Note:

Using `Function.prototype.bind` in render creates a new function each time the component renders, which may have performance implications (see below).

Arrow Function in Render

```
class Foo extends Component {
  handleClick() {
    console.log('Click happened');
  }
  render() {
    return <button onClick={() => this.handleClick()}>Click Me</button>;
  }
}
```

Note:

Using an arrow function in render creates a new function each time the component renders, which may break optimizations based on strict identity comparison.

Is it OK to use arrow functions in render methods?

Generally speaking, yes, it is OK, and it is often the easiest way to pass parameters to callback functions.

If you do have performance issues, by all means, optimize!

Why is binding necessary at all?

In JavaScript, these two code snippets are **not** equivalent:

```
obj.method();
```

```
var method = obj.method;
method();
```

Binding methods helps ensure that the second snippet works the same way as the first one.

With React, typically you only need to bind the methods you *pass* to other components. For example, `<button onClick={this.handleClick}>` passes `this.handleClick` so you want to bind it. However, it is unnecessary to bind the `render` method or the lifecycle methods: we don't pass them to other components.

[This post by Yehuda Katz](#) explains what binding is, and how functions work in JavaScript, in detail.

Why is my function being called every time the component renders?

Make sure you aren't *calling the function* when you pass it to the component:

```
render() {
  // Wrong: handleClick is called instead of passed as a reference!
  return <button onClick={this.handleClick()}>Click Me</button>
}
```

Instead, *pass the function itself* (without parens):

```
render() {
  // Correct: handleClick is passed as a reference!
  return <button onClick={this.handleClick}>Click Me</button>
}
```

How do I pass a parameter to an event handler or callback?

You can use an arrow function to wrap around an event handler and pass parameters:

```
<button onClick={() => this.handleClick(id)} />
```

This is equivalent to calling `.bind`:

```
<button onClick={this.handleClick.bind(this, id)} />
```

Example: Passing params using arrow functions

```
const A = 65 // ASCII character code

class Alphabet extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      justClicked: null,
      letters: Array.from({length: 26}, (_, i) => String.fromCharCode(A + i))
    };
  }
  handleClick(letter) {
    this.setState({ justClicked: letter });
  }
  render() {
    return (
      <div>
        Just clicked: {this.state.justClicked}
        <ul>
          {this.state.letters.map(letter =>
            <li key={letter} onClick={() => this.handleClick(letter)}>
              {letter}
            </li>
          )}
        </ul>
      </div>
    );
  }
}
```

Example: Passing params using data-attributes

Alternately, you can use DOM APIs to store data needed for event handlers. Consider this approach if you need to optimize a large number of elements or have a render tree that relies on `React.PureComponent` equality checks.

```
const A = 65 // ASCII character code

class Alphabet extends React.Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
    this.state = {
      justClicked: null,
      letters: Array.from({length: 26}, (_, i) => String.fromCharCode(A + i))
    };
  }

  handleClick(e) {
    this.setState({
      justClicked: e.target.dataset.letter
    });
  }

  render() {
    return (
      <div>
        Just clicked: {this.state.justClicked}
        <ul>
          {this.state.letters.map(letter =>
            <li key={letter} data-letter={letter} onClick={this.handleClick}>
              {letter}
            </li>
          )}
        </ul>
      </div>
    );
  }
}
```

How can I prevent a function from being called too quickly or too many times in a row?

If you have an event handler such as `onClick` or `onScroll` and want to prevent the callback from being fired too quickly, then you can limit the rate at which callback is executed. This can be done by using:

- **throttling**: sample changes based on a time based frequency (eg `_.throttle`)
- **debouncing**: publish changes after a period of inactivity (eg `_.debounce`)
- **requestAnimationFrame throttling**: sample changes based on `requestAnimationFrame` (eg `raf-schd`)

See [this visualization](#) for a comparison of `throttle` and `debounce` functions.

Note:

`_.debounce`, `_.throttle` and `raf-schd` provide a `cancel` method to cancel delayed callbacks. You should either call this method from `componentWillUnmount` or check to ensure that the component is still mounted within the delayed function.

Throttle

Throttling prevents a function from being called more than once in a given window of time. The example below throttles a "click" handler to prevent calling it more than once per second.

```
import throttle from 'lodash.throttle';

class LoadMoreButton extends React.Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
    this.handleClickThrottled = throttle(this.handleClick, 1000);
  }

  componentWillMount() {
    this.handleClickThrottled.cancel();
  }

  render() {
    return <button onClick={this.handleClickThrottled}>Load More</button>;
  }

  handleClick() {
    this.props.loadMore();
  }
}
```

Debounce

Debouncing ensures that a function will not be executed until after a certain amount of time has passed since it was last called. This can be useful when you have to perform some expensive calculation in response to an event that might dispatch rapidly (eg scroll or keyboard events). The example below debounces text input with a 250ms delay.

```
import debounce from 'lodash.debounce';

class Searchbox extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.emitChangeDebounced = debounce(this.emitChange, 250);
  }

  componentWillMount() {
    this.emitChangeDebounced.cancel();
  }

  render() {
    return (
      <input
        type="text"
        onChange={this.handleChange}
    
```

```

        placeholder="Search...""
        defaultValue={this.props.value}
      />
    );
}

handleChange(e) {
  this.emitChangeDebounced(e.target.value);
}

emitChange(value) {
  this.props.onChange(value);
}
}

```

`requestAnimationFrame` throttling

`requestAnimationFrame` is a way of queuing a function to be executed in the browser at the optimal time for rendering performance. A function that is queued with `requestAnimationFrame` will fire in the next frame. The browser will work hard to ensure that there are 60 frames per second (60 fps). However, if the browser is unable to it will naturally *limit* the amount of frames in a second. For example, a device might only be able to handle 30 fps and so you will only get 30 frames in that second. Using `requestAnimationFrame` for throttling is a useful technique in that it prevents you from doing more than 60 updates in a second. If you are doing 100 updates in a second this creates additional work for the browser that the user will not see anyway.

Note:

Using this technique will only capture the last published value in a frame. You can see an example of how this optimization works on [MDN](#)

```

import rafSchedule from 'raf-schd';

class ScrollListener extends React.Component {
  constructor(props) {
    super(props);

    this.handleScroll = this.handleScroll.bind(this);

    // Create a new function to schedule updates.
    this.scheduleUpdate = rafSchedule(
      point => this.props.onScroll(point)
    );
  }

  handleScroll(e) {
    // When we receive a scroll event, schedule an update.
    // If we receive many updates within a frame, we'll only publish the latest
    // value.
    this.scheduleUpdate({ x: e.clientX, y: e.clientY });
  }

  componentWillUnmount() {
    // Cancel any pending updates since we're unmounting.
  }
}

```

```
    this.scheduleUpdate.cancel();
}

render() {
  return (
    <div
      style={{ overflow: 'scroll' }}
      onScroll={this.handleScroll}
    >
      
    </div>
  );
}

}
```

Testing your rate limiting

When testing your rate limiting code works correctly it is helpful to have the ability to fast forward time. If you are using `jest` then you can use `mock timers` to fast forward time. If you are using `requestAnimationFrame` throttling then you may find `raf-stub` to be a useful tool to control the ticking of animation frames.

Virtual DOM and Internals

What is the Virtual DOM?

The virtual DOM (VDOM) is a programming concept where an ideal, or "virtual", representation of a UI is kept in memory and synced with the "real" DOM by a library such as ReactDOM. This process is called [reconciliation](#).

This approach enables the declarative API of React: You tell React what state you want the UI to be in, and it makes sure the DOM matches that state. This abstracts out the attribute manipulation, event handling, and manual DOM updating that you would otherwise have to use to build your app.

Since "virtual DOM" is more of a pattern than a specific technology, people sometimes say it to mean different things. In React world, the term "virtual DOM" is usually associated with [React elements](#) since they are the objects representing the user interface. React, however, also uses internal objects called "fibers" to hold additional information about the component tree. They may also be considered a part of "virtual DOM" implementation in React.

Is the Shadow DOM the same as the Virtual DOM?

No, they are different. The Shadow DOM is a browser technology designed primarily for scoping variables and CSS in web components. The virtual DOM is a concept implemented by libraries in JavaScript on top of browser APIs.

What is "React Fiber"?

Fiber is the new reconciliation engine in React 16. Its main goal is to enable incremental rendering of the virtual DOM. [Read more](#).

Component State

useState 的作用？

`useState()` 安排對 component `state` object 的更新。當 state 變更時，component 會藉由重新 render 來回應。

state 和 props 有什麼不同？

`props` (「properties」的簡寫) 和 `state` 都是純 JavaScript object。雖然兩者都擁有會影響 render 輸出的資訊，但在一個重要方向上有所不同：`props` 是被傳遞進 component (類似於 function 的參數)，而 `state` 是在 component 內部被管理 (類似於在 function 中宣告中的變數)。

這裡有一些很棒的資源，可以進一步閱讀來了解使用 `props` 和 `state` 的時機：

- [Props vs State](#)
- [ReactJS: Props vs. State](#)

為什麼 useState 會出現錯誤的值？

在 React，`this.props` 和 `this.state` 都代表該 `render` 的值。也就是目前螢幕上的內容。

呼叫 `setState` 是非同步的 — 不要在呼叫 `setState` 後立即依賴 `this.state` 去反映新的值。如果需要基於目前 state 來計算值，請傳遞一個更新用的 function 而不是 object (細節請看下方)。

不會符合預期結果的程式碼範例：

```
incrementCount() {
  // Note: this will *not* work as intended.
  this.setState({count: this.state.count + 1});
}

handleSomething() {
  // Let's say `this.state.count` starts at 0.
  this.incrementCount();
  this.incrementCount();
  this.incrementCount();
  // When React re-renders the component, `this.state.count` will be 1, but you
  // expected 3.

  // This is because `incrementCount()` function above reads from
  // `this.state.count`,
  // but React doesn't update `this.state.count` until the component is re-
  // rendered.
  // So `incrementCount()` ends up reading `this.state.count` as 0 every time, and
  // sets it to 1.

  // The fix is described below!
}
```

關於如何修正此問題的資訊請參考下方。

如何使用目前 state 的值來更新 state？

將一個 function 而非 object 傳遞給 `setState`，來確保呼叫時始終是使用最新版本的 state（請看下方）。

在 `setState` 中傳遞 object 或 function 有何不同？

傳遞一個更新用的 function 允許你在 updater 內存取目前 state 的值。由於 `setState` 是批次處理呼叫的，這讓你能夠鏈結更新並確保它們依序建立而不會產生衝突：

```
incrementCount() {
  this.setState((state) => {
    // Important: read `state` instead of `this.state` when updating.
    return {count: state.count + 1}
  });
}

handleSomething() {
  // Let's say `this.state.count` starts at 0.
  this.incrementCount();
  this.incrementCount();
  this.incrementCount();

  // If you read `this.state.count` now, it would still be 0.
  // But when React re-renders the component, it will be 3.
}
```

[學習更多關於 `setState`](#)

`setState` 何時是非同步？

目前 `setState` 在 event handler 中是非同步。

這會確保 `Child` 不會重新 render 兩次，像是 `Parent` 和 `Child` 在一個單次 click 事件中同時呼叫 `setState` 的例子。取而代之，React 會在瀏覽器事件結束時「刷新」state 的更新。這在大型應用程式中能產生顯著的效能提升。

這是一個實作細節所以請避免直接依賴它。未來的版本中，React 將在更多情況下預設批次處理更新。

為什麼 React 不同步地更新 `this.state`？

如同在前面篇幅解釋的，在開始重新 render 前，React 意圖「等待」直到所有的 component 都在其 event handler 中呼叫 `setState()`。藉由避免不必要的重新 render 來增加效能。

不過，你也許仍想知道，為什麼 React 不在不重新 render 的情況下立即更新 `this.state` 就好。

有 2 個主要原因：

- 這會破壞 `props` 和 `state` 間的一致性，造成難以除錯的問題。
- 這會造成我們正在研究的一些新功能難以實現。

這則 [GitHub 評論](#)以具體範例來深入探討。

我應該使用 Redux 或 MobX 之類的狀態管理函式庫嗎？

看情況。

在加入額外的函式庫前先來了解 React 是個好主意。因為你可以只用 React 就建立一個夠複雜的應用程式。

File Structure

有任何建構 React 專案的推薦方式嗎？

React 對你如何將檔案放入資料夾中並無意見。不過你可以考慮幾種在生態系中常見的方法。

以功能或路徑分類

一種建構專案的常見方式是將 CSS, JS, 和測試檔案放在以功能或路徑分類的資料夾中。

```
common/
  Avatar.js
  Avatar.css
  APIUtils.js
  APIUtils.test.js
feed/
  index.js
  Feed.js
  Feed.css
  FeedStory.js
  FeedStory.test.js
  FeedAPI.js
profile/
  index.js
  Profile.js
  ProfileHeader.js
  ProfileHeader.css
  ProfileAPI.js
```

「功能」的定義並不是統一的，且其粒度由你決定。如果你無法列出最高階層的資料夾，你可以詢問你的產品使用者，產品包含哪些主要部分，然後以他們的心智模型作為藍圖。

以檔案類型分類

另一個受歡迎的建構專案方式是將相似的檔案組在一起，例如：

```
api/
  APIUtils.js
  APIUtils.test.js
  ProfileAPI.js
  UserAPI.js
components/
  Avatar.js
  Avatar.css
  Feed.js
  Feed.css
  FeedStory.js
  FeedStory.test.js
  Profile.js
  ProfileHeader.js
  ProfileHeader.css
```

有些人也喜歡進一步將 component 按照其在應用程式中的角色分進不同的資料夾裡。例如，[Atomic Design](#) 便是一個基於此原則的設計方法。記得，把類似的方法視為有幫助的例子而非必須遵守的嚴格規定，通常會更有成效。

避免太多巢狀

JavaScript 專案中有很多和深度巢狀目錄有關的痛點。撰寫其中的相對導入或是在檔案移動後更新這些導入變得越來越難。除非你有非常令人信服的原因必須使用深度巢狀資料夾，不然請考慮限制你自己在一個專案中使用最多三層或四層資料夾。當然，這只是一個建議，且也不一定會和你的專案有關。

不要想得太多

如果你才剛開始一個專案，[請不要花超過五分鐘](#)在選擇檔案結構。選擇以上任何方法（或想一個你自己的）然後開始寫程式碼！畢竟在你寫了一些真正的程式碼後你很可能會想重新思考。

如果你感到完全卡住，從把所有檔案放在一個資料夾開始。最終它會變大到你會想將某些檔案和剩下的分開。在那時你就會有足夠的認知去分辨哪些檔案你最常一起更改。一般來說，把經常一起更改的檔案放得和彼此近一些是個好主意。這個原則稱為「colocation」。

當專案變得更大時，實務上他們通常會使用上述兩種方法的混合。所以在一開始選擇「正確的」那一個並不是非常重要。

Styling and CSS

如何在 Component 中加入 CSS Class?

傳遞一個 string 作為 `className` 的 prop :

```
render() {
  return <span className="menu navigation-menu">Menu</span>
}
```

通常 CSS class 會依賴 component 的 prop 或 state :

```
render() {
  let className = 'menu';
  if (this.props.isActive) {
    className += ' menu-active';
  }
  return <span className={className}>Menu</span>
}
```

小技巧

如果你發現你常寫類似的程式碼，可使用 `classnames` 函式庫簡化。

可以使用 Inline Style 嗎？

可以，請見[此處](#)文件關於 styling 部分。

Inline Style 不好嗎？

CSS class 通常比 inline style 效能更好。

CSS-in-JS 是什麼？

「CSS-in-JS」指的是使用 JavaScript 組成 CSS 而非將其定義於外部檔案的一種模式。

請注意此功能為第三方函式庫所提供之部分，並非 React 的一部分。React 對 style 是如何定義並無意見。若有疑問，照常地將你的 style 定義於另一個 `*.css` 檔案中然後使用 `className` 去引用會是個好的開始。

我可以在 React 中做動畫嗎？

React 可以支援動畫。範例請見 [React Transition Group](#) 和 [React Motion](#)、[React Spring](#) 或是 [Framer Motion](#)。

[Go to TOC](#)

Versioning Policy

React follows [semantic versioning \(semver\)](#) principles.

That means that with a version number **x.y.z**:

- When releasing **critical bug fixes**, we make a **patch release** by changing the **z** number (ex: 15.6.2 to 15.6.3).
- When releasing **new features** or **non-critical fixes**, we make a **minor release** by changing the **y** number (ex: 15.6.2 to 15.7.0).
- When releasing **breaking changes**, we make a **major release** by changing the **x** number (ex: 15.6.2 to 16.0.0).

Major releases can also contain new features, and any release can include bug fixes.

Minor releases are the most common type of release.

This versioning policy does not apply to prerelease builds in the Next or Experimental channels. [Learn more about prereleases](#).

Breaking Changes

Breaking changes are inconvenient for everyone, so we try to minimize the number of major releases – for example, React 15 was released in April 2016 and React 16 was released in September 2017, and React 17 was released in October 2020.

Instead, we release new features in minor versions. That means that minor releases are often more interesting and compelling than majors, despite their unassuming name.

Commitment to Stability

As we change React over time, we try to minimize the effort required to take advantage of new features. When possible, we'll keep an older API working, even if that means putting it in a separate package. For example, [mixins have been discouraged for years](#) but they're supported to this day [via create-react-class](#) and many codebases continue to use them in stable, legacy code.

Over a million developers use React, collectively maintaining millions of components. The Facebook codebase alone has over 50,000 React components. That means we need to make it as easy as possible to upgrade to new versions of React; if we make large changes without a migration path, people will be stuck on old versions. We test these upgrade paths on Facebook itself – if our team of less than 10 people can update 50,000+ components alone, we hope the upgrade will be manageable for anyone using React. In many cases, we write [automated scripts](#) to upgrade component syntax, which we then include in the open-source release for everyone to use.

Gradual Upgrades via Warnings

Development builds of React include many helpful warnings. Whenever possible, we add warnings in preparation for future breaking changes. That way, if your app has no warnings on the latest release, it will be compatible with the next major release. This allows you to upgrade your apps one component at a time.

Development warnings won't affect the runtime behavior of your app. That way, you can feel confident that your app will behave the same way between the development and production builds -- the only differences are that the production build won't log the warnings and that it is more efficient. (If you ever notice otherwise, please file an issue.)

What Counts as a Breaking Change?

In general, we *don't* bump the major version number for changes to:

- **Development warnings.** Since these don't affect production behavior, we may add new warnings or modify existing warnings in between major versions. In fact, this is what allows us to reliably warn about upcoming breaking changes.
- **APIs starting with `unstable_`.** These are provided as experimental features whose APIs we are not yet confident in. By releasing these with an `unstable_` prefix, we can iterate faster and get to a stable API sooner.
- **Alpha and canary versions of React.** We provide alpha versions of React as a way to test new features early, but we need the flexibility to make changes based on what we learn in the alpha period. If you use these versions, note that APIs may change before the stable release.
- **Undocumented APIs and internal data structures.** If you access internal property names like `__SECRET_INTERNALS_DO_NOT_USE_OR_YOU_WILL_BE_FIRED` or `__reactInternalInstance$uk43rzhitjg`, there is no warranty. You are on your own.

This policy is designed to be pragmatic: certainly, we don't want to cause headaches for you. If we bumped the major version for all of these changes, we would end up releasing more major versions and ultimately causing more versioning pain for the community. It would also mean that we can't make progress in improving React as fast as we'd like.

That said, if we expect that a change on this list will cause broad problems in the community, we will still do our best to provide a gradual migration path.

If a Minor Release Includes No New Features, Why Isn't It a Patch?

It's possible that a minor release will not include new features. [This is allowed by semver](#), which states "[a minor version] MAY be incremented if substantial new functionality or improvements are introduced within the private code. It MAY include patch level changes."

However, it does raise the question of why these releases aren't versioned as patches instead.

The answer is that any change to React (or other software) carries some risk of breaking in unexpected ways. Imagine a scenario where a patch release that fixes one bug accidentally introduces a different bug. This would not only be disruptive to developers, but also harm their confidence in future patch releases. It's especially regrettable if the original fix is for a bug that is rarely encountered in practice.

We have a pretty good track record for keeping React releases free of bugs, but patch releases have an even higher bar for reliability because most developers assume they can be adopted without adverse consequences.

For these reasons, we reserve patch releases only for the most critical bugs and security vulnerabilities.

If a release includes non-essential changes — such as internal refactors, changes to implementation details, performance improvements, or minor bugfixes — we will bump the minor version even when there are no new features.

表單

HTML 表單的 element 和 React 中其他的 DOM element 不太一樣，因為表單的 element 很自然地有一些內部的 state。例如，這個表單在下面簡單的 HTML 中接受一個名稱：

```
<form>
  <label>
    Name:
    <input type="text" name="name" />
  </label>
  <input type="submit" value="Submit" />
</form>
```

當使用者提交表單時，此表單具有瀏覽到新頁面的預設 HTML 表單行為。如果你想要在 React 中也有這樣的行為的話，直接用 HTML 是可行的。但是在大多數的情況中，有一個 JavaScript function 來處理提交表單的功能並讀取使用者在表單中填入的資料是很方便的。要做到這樣，標準的方法是使用「controlled component」：

Controlled Component

在 HTML 中，表單的 element 像是 `<input>`、`<textarea>` 和 `<select>` 通常會維持它們自身的 state，並根據使用者的輸入來更新 state。在 React 中，可變的 state 通常是被維持在 component 中的 state property，並只能以 `setState()` 來更新。

我們可以透過將 React 的 state 變成「唯一真相來源」來將這兩者結合。如此，render 表單的 React component 同時也掌握了後續使用者的輸入對表單帶來的改變。像這樣一個輸入表單的 element，被 React 用這樣的方式來控制它的值，就被稱為「controlled component」。

例如，如果我們想要讓上一個範例在一個名字被輸入表單時印出，我們可以把這個表單寫成一個 controlled component：

```
class NameForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: ''};

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({value: event.target.value});
  }

  handleSubmit(event) {
    alert('A name was submitted: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name:
          <input type="text" value={this.state.value} onChange={this.handleChange}>
        </label>
      </form>
    );
  }
}
```

```

    />
    </label>
    <input type="submit" value="Submit" />
  </form>
);
}
}
}

```

[在 CodePen 上試試看吧！](#)

由於 `value` attribute 是被設定在我們的表單 element 上，顯示的 value 會永遠是 `this.state.value`，這使得 React 的 state 成為了資料來源。由於 `handleChange` 在每一次鍵盤被敲擊時都會被執行，並更新 React 的 state，因此被顯示的 value 將會在使用者打字的同時被更新。

在這樣的 controlled component 中，顯示的 value 始終由 React 的 state 驅動，雖然這意味著你必須寫更多的 code，但現在你同時可以將 value 傳遞給其他的 UI element，或是從其他 event handler 重置。

Textarea 標籤

在 HTML 中，一個 `<textarea>` 的 element 是經由它的 children 來定義它的文字：

```

<textarea>
  Hello there, this is some text in a text area
</textarea>

```

在 React 中，`<textarea>` 則是使用一個 `value` 的 attribute。如此一來，一個使用 `<textarea>` 的表單可以使用非常類似單行的 input 方法來寫成：

```

class EssayForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: 'Please write an essay about your favorite DOM element.'
    };

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({value: event.target.value});
  }

  handleSubmit(event) {
    alert('An essay was submitted: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Essay:
          <textarea value={this.state.value} onChange={this.handleChange} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}

```

```

    );
}
}

```

請注意 `this.state.value` 是在 constructor 內被初始化的，所以上述的 text area 一開始會有一些文字。

Select 標籤

在 HTML 中，`<select>` 會建立一個下拉式選單。例如，這個 HTML 會建立一個有各種水果的下拉式選單：

```

<select>
  <option value="grapefruit">Grapefruit</option>
  <option value="lime">Lime</option>
  <option selected value="coconut">Coconut</option>
  <option value="mango">Mango</option>
</select>

```

請注意在這裡，椰子的選項是一開始就被選定的，因為它有一個 `selected` attribute。但是在 React 中並不是用 `selected` attribute，而是在 `select` 的標籤上用一個 `value` attribute。對一個 controlled component 來說這是比較方便的，因為你只需要在一個地方更新它。例如：

```

class FlavorForm extends React.Component {
  constructor(props) {
    super(props);
    this.state = {value: 'coconut'};

    this.handleChange = this.handleChange.bind(this);
    this.handleSubmit = this.handleSubmit.bind(this);
  }

  handleChange(event) {
    this.setState({value: event.target.value});
  }

  handleSubmit(event) {
    alert('Your favorite flavor is: ' + this.state.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Pick your favorite flavor:
          <select value={this.state.value} onChange={this.handleChange}>
            <option value="grapefruit">Grapefruit</option>
            <option value="lime">Lime</option>
            <option value="coconut">Coconut</option>
            <option value="mango">Mango</option>
          </select>
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}

```

[在 CodePen 上試試看吧！](#)

整體來說，這使得 `<input type="text">`，`<textarea>` 和 `<select>` 使用起來都很類似。它們全都會接收一個你在實作一個 controlled component 時會使用到的 `value` attribute。

注意

你可以將一個 array 傳給 `value` 這個 attribute，這使得你可以在一個 `select` 中選取多個選項：

```
<select multiple={true} value={['B', 'C']}>
```

檔案 input 標籤

在 HTML 中，`<input type="file">` 讓使用者從它們的儲存裝置中選擇一個至多個檔案，並把它們上傳到伺服器或透過 File API 被 JavaScript 處理。

```
<input type="file" />
```

由於它的值是唯讀，它在 React 中是一個 **uncontrolled** component。在稍後的文件中有其他關於它和其他 uncontrolled component 的討論。

處理多個輸入

當你需要處理多個 controlled `input` element，你可以在每個 element 中加入一個 `name` attribute，並讓 handler function 選擇基於 `event.target.name` 的值該怎麼做：

例如：

```
class Reservation extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      isGoing: true,
      number0fGuests: 2
    };
    this.handleInputChange = this.handleInputChange.bind(this);
  }

  handleInputChange(event) {
    const target = event.target;
    const value = target.type === 'checkbox' ? target.checked : target.value;
    const name = target.name;

    this.setState({
      [name]: value
    });
  }

  render() {
    return (
      <div>
        <input checked="" name="isGoing" type="checkbox"/> Are you going?
        <input type="checkbox" name="isGoing" type="checkbox"/> Are you going?
        <br/>
        <input type="text" name="number0fGuests" value="2"/> Number of guests
      </div>
    );
  }
}
```

```

<form>
  <label>
    Is going:
    <input
      name="isGoing"
      type="checkbox"
      checked={this.state.isGoing}
      onChange={this.handleInputChange} />
  </label>
  <br />
  <label>
    Number of guests:
    <input
      name="numberOfGuests"
      type="number"
      value={this.state.numberOfGuests}
      onChange={this.handleInputChange} />
  </label>
</form>
);
}
}

```

在 [CodePen 上試試看吧！](#)

注意我們使用了 ES6 的 `computed property name` 語法來更新與輸入中的 `name` 相對應的 `state key`：

```

this.setState({
  [name]: value
});

```

這和下面的 ES5 程式碼是一樣的：

```

var partialState = {};
partialState[name] = value;
this.setState(partialState);

```

此外，由於 `setState()` 自動 `merge` 一部分的 `state` 和現在的 `state`，我們只需要在有改變的地方呼叫它即可。

Controlled 輸入值為 Null

在一箇 `controlled component` 上指明 `value prop` 可避免使用者改變輸入，除非你希望使用者這樣做。如果你已經指明了 `value` 但輸入仍然是可以被修改的，你很可能是不小心將 `value` 的值設定為 `undefined` 或 `null`。

下面的程式碼就是一個範例。（輸入原先是被鎖住的，但在短暫的延遲後，變得可以被修改了。）

```

ReactDOM.createRoot(mountNode).render(<input value="hi" />);

setTimeout(function() {
  ReactDOM.createRoot(mountNode).render(<input value={null} />);
}, 1000);

```

Controlled component 的替代方案

有時候使用 controlled component 是很乏味的，因為你需要為每一個資訊可以改變的方式寫一個 event handler，並將所有的輸入 state 透過一個 React component 來傳遞。這在你將一個舊的 codebase 改寫成 React 時或將一個 React 的應用程式與一個非 React 的函式庫整合時會變得特別麻煩。在這種情況中，你也許會想參考 [uncontrolled component](#)，也就是另一種取代輸入表格的方式。

成熟的解決方案

如果你想找出一個完整的、包含驗證、可追蹤拜訪欄位並能處理提交表單等功能的解決方案，[Formik](#) 是一個很熱門的選擇。然而，它是在與 controlled component 和維持 state 相同的原則上所建立的，所以別忘了學習它。

傳送 Ref

傳送 ref 是一種自動把 `ref` 從一個 component 傳遞到它底下的其中一個 child 的技巧。通常來說，應用程式裡大部分的 component 都不需要用到它。然而，對某些種類的 component 來說它很有用，特別是能夠重複使用的函式庫。以下會解釋最常見的情形。

傳送 ref 到 DOM component

試著考慮一個叫做 `FancyButton` 的 component，它會 render 一個原生的 `button` DOM element：
`embed:forwarding-refs/fancy-button-simple.js`

React component 會隱藏包含 `render` 的結果在內的實作細節。其他使用到 `FancyButton` 的 component **通常不需要** 獲得內部按鈕的 DOM element 的 `ref`。這樣帶來的好處是，我們能夠避免讓其他 component 過度依賴彼此的 DOM 的結構。

雖然這樣的封裝對於應用程式層級的 component（像是 `FeedStory` 或 `Comment`）來說是我們所希望擁有的，但對於常用的「末端」component，像是 `FancyButton` 或 `MyTextInput` 來說，可能會變得不方便使用。這些末端 component 通常會像普通的 DOM `button` 和 `input` 一樣，在應用程式的各個地方被使用，在處理 `focus`、選取或動畫時，取得它們的 DOM 節點可能是不可避免的。

傳送 ref 是個選擇性的功能，它能夠讓某些 component 利用它們收到的 `ref` 來傳遞到底下的 child component。

在下面的例子中，`FancyButton` 藉由 `React.forwardRef` 來獲取傳遞到它身上的 `ref`，然後再傳遞到它 render 的 DOM `button` 上：

`embed:forwarding-refs/fancy-button-simple-ref.js`

這樣一來，使用 `FancyButton` 的 component 可以獲得它底下的 `button` 的 DOM 節點的 `ref`，並可以在需要的時候獲取它——就如同直接使用 `button` DOM 一樣。

以下一步一步的解釋上面的例子到底發生了什麼事：

1. 我們藉由呼叫 `React.createRef` 產生了一個 `React ref`，然後將它賦值於叫做 `ref` 的變數裡。
2. 我們藉由把 `ref` 當成一個 JSX attribute 來把它傳遞到 `<FancyButton ref={ref}>`。
3. React 把 `ref` 當作第二個變數傳到 `forwardRef` 裡的 `(props, ref) => ...` function。
4. 我們藉由把這個 `ref` 當作 JSX attribute 來傳遞到更下面的 `<button ref={ref}>`。
5. 當 `ref` 被附上之後，`ref.current` 會指向 `<button>` DOM 節點。

注意

第二個 `ref` 變數只會在你用 `React.forwardRef` 呼叫定義一個 component 的時候存在。一般 function 或 class component 不會獲得 `ref` 變數，且在 `props` 裡 `ref` 也不存在。。

傳送 ref 不侷限於 DOM 元件。你可以也將 `ref` 傳遞到 class component。

對於 component 函式庫維護者的提醒

當你在函式庫開始使用 `forwardRef` 時，你應該要把它當作重大變化，然後為你的元件庫發佈新的主要版號。因為你的函式庫可能會有可見的不同行為（像是 `ref` 被指定在哪，或是什麼型別會被輸出），然後它可能會破壞那些依賴於舊有行為的應用程式或其他套件。

有條件的在 `ref` 存在時才使用 `React.forwardRef` 也是不推薦的方式，相同的原因：它改變了你的套件的行為，且當你的使用者升級 React 之後可能會破壞使用者的應用程式。

在 Higher-Order Component 內傳送 ref

在 [Higher-Order Component](#)（也叫做 HOC）裡，這樣的技巧會特別有用。讓我們用一個會把 component 的 prop 記錄到 `console` 的 HOC 為例：`embed:forwarding-refs/log-props-before.js`

這個「`logProps`」HOC 把所有的 `prop` 傳遞到他所包裹的 component，所以被 `render` 出的結果會是一樣的。舉例來說，我們可以用這個 HOC 來記錄所有經過「`fancy button`」元件的 prop：`embed:forwarding-refs/fancy-button.js`

以上的範例有個警告：`ref` 不會被傳遞過去。因為 `ref` 不是個 prop。就像 `key` 一樣，React 用不同的方式處理它。如果你想要在 HOC 加上一個 `ref`，`ref` 會被指定到最外層的 component，而不是直接包裹它的那個 component。

這代表對於 `FancyButton` component 所用的 `ref` 會直接被附到 `LogProps` component：`embed:forwarding-refs/fancy-button-ref.js`

幸運的是，我們可以刻意利用 `React.forwardRef` API 把 `ref` 傳送到裡面的 `FancyButton` component。`React.forwardRef` 接受一個用到 `props` 和 `ref` 參數的 `render` 的 function 並回傳一個 React 節點。例如：`embed:forwarding-refs/log-props-after.js`

在 DevTool 裡顯示客製化的名稱

`React.forwardRef` 接受一個 `render` function。React DevTool 使用這個 function 來決定這個傳送 `ref` 的 component 該顯示什麼。

例如，在 DevTool 裡，這個傳送的 component 會顯示為「`ForwardRef`」：

`embed:forwarding-refs/wrapped-component.js`

如果你對 `render` function 取名，DevTool 會包含它的名字（例如：「`ForwardRef(myFunction)`」）：

`embed:forwarding-refs/wrapped-component-with-function-name.js`

你也可以設定 function 的 `displayName` 來包裹你的 component：

`embed:forwarding-refs/customized-display-name.js`

Fragments

React 其中一種常見的使用情況是在一個 component 中回傳多個 element, fragment 讓你能夠在不用增加額外 DOM 節點的情況下，重新組合 child component。

```
render() {
  return (
    <React.Fragment>
      <ChildA />
      <ChildB />
      <ChildC />
    </React.Fragment>
  );
}
```

還有一種簡寫語法可以用來宣告 fragment。

動機

常見的情況是在 component 裡回傳一連串的 child element，看看這個 React 的程式碼片段：

```
class Table extends React.Component {
  render() {
    return (
      <table>
        <tr>
          <Columns />
        </tr>
      </table>
    );
  }
}
```

為了使 render 出來的 HTML 是有效的，`<Columns />` 需要回傳多個 `<td>` element。如果將 parent div 元素放在 `<Columns />` 中的 `render()` 區塊，將會使生成的 HTML 無效。

```
class Columns extends React.Component {
  render() {
    return (
      <div>
        <td>Hello</td>
        <td>World</td>
      </div>
    );
  }
}
```

在 `<Table />` 內的輸出如下：

```
<table>
  <tr>
    <div>
      <td>Hello</td>
```

```

<td>World</td>
</div>
</tr>
</table>

```

這個問題交給 fragment 解決。

使用方式

```

class Columns extends React.Component {
  render() {
    return (
      <React.Fragment>
        <td>Hello</td>
        <td>World</td>
      </React.Fragment>
    );
  }
}

```

會讓 `<Table />` 得到一個正確的輸出：

```

<table>
  <tr>
    <td>Hello</td>
    <td>World</td>
  </tr>
</table>

```

簡寫語法

你可以用新的簡寫語法來宣告 fragment, 它看起來就像空標籤：

```

class Columns extends React.Component {
  render() {
    return (
      <>
        <td>Hello</td>
        <td>World</td>
      </>
    );
  }
}

```

你可以像使用其他元素一樣使用 `<></>`, 但值得注意的是它並不支援 key 和 attribute。

Keyed Fragments

透過明確宣告 `<React.Fragment>` 的 fragment 可能會遇到帶有 key 的情況。一個使用案例是將它 mapping 到 fragment array。舉例來說，像下方程式碼一樣建立一個敘述列表：

```

function Glossary(props) {
  return (
    <dl>
      {props.items.map(item => (
        // 如果缺少 key , React 會發出一個缺少 key 的警告

```

```
<React.Fragment key={item.id}>
  <dt>{item.term}</dt>
  <dd>{item.description}</dd>
</React.Fragment>
) );
);
}
```

目前 `key` 是唯一可以傳遞給 `Fragment` 的 attribute。之後我們可能會支援更多的 attribute，像是 event handler。

Live Demo

你可以透過 [CodePen](#) 嘗試新的 JSX fragment 語法。

[Go to TOC](#)

開始

- "download.html"
- "downloads.html"

本章節是 React 文件與相關資訊的概覽。

React 是一個實作使用者介面的 JavaScript 函式庫。請到我們的[主頁](#)或[教學](#)中學習什麼是 React。

- [嘗試 React](#)
- [學習 React](#)
- [了解最新消息](#)
- [版本化文件](#)
- [找不到相關資訊？](#)

嘗試 React

React 在剛推出的時候就容許被逐步採用，[你可以按自己所需可多可少的採用 React](#)。不管你是想初步嘗試 React、在簡單的 HTML 網頁上加入互動性，或是實作一個使用 React 驅動的複雜應用程式，這章節裏的連結會幫助你開始。

線上體驗

如果你有興趣嘗試 React，你可以使用一些線上編輯器。在 [CodePen](#)、[CodeSandbox](#)，或是 [Stackblitz](#) 實作一個 Hello World 的範例。

如果你偏向使用自己的編輯器，你可以[下載此 HTML 文件](#)進行修改，並從本機文件系統裡把它打開到瀏覽器。這會進行一個緩慢的 runtime 程式碼轉換，因此我們建議只把這文件用作簡單的示範。

將 React 加入到網頁

你可以在一分鐘內將 React 加入到一個 HTML 網頁上。你可以選擇逐步擴大它的應用範圍，或是只使用在少部分的可變 widget 上。

實作全新的 React 應用程式

當你剛實作全新的 React 應用程式時，一個[簡單的 HTML 網頁配上 script 標籤](#)或許依然是最好的選擇。這樣我們就可以在一分鐘內設定好。

當你的應用程式逐漸擴大，你或許要考慮使用一個比較綜合的設定方法。我們推薦使用一些 JavaScript toolchain 來設定大型的應用程式。它們只需要一些設定，甚至不需要，就能讓我們充份使用豐富的 React ecosystem。[學習如何設定](#)

學習 React

學習 React 的人來至於不同的背景，以及不同的學習方式。無論你是偏向理論化，或是實踐化的方式，我們希望本章節會對你有所幫助。

- 如果你喜歡動手做，請從我們的[實用指南](#)開始。

- 如果你喜歡**按部就班學習概念**，請從我們的[主要概念指南](#)開始。

與學習其他全新的科技一樣，學習 React 會有一定的難度。我們相信只要多練習和添加多一份耐心，你一定會掌握到 React。

第一個範例

[React 首頁](#)包含了幾個小型的 React 範例與實時編輯器。即使你還沒完全了解 React，你也可以嘗試變更它們的程式碼，看看結果會怎樣。

初學者的 React

如果你覺得 React 官方文件學習的節奏比較快，不太適應，請先看看 [Tania Rascia 所編寫的 React 概覽](#)。她會以初學者的角度，介紹 React 的主要概念。當你看完了，嘗試再回來繼續學習吧！

設計師的 React

如果你是來至於設計背景的，[這些資源](#)會是一個很好的起點。

JavaScript 的資源

我們假設你有一些 JavaScript 的基礎知識。你不需要是 JavaScript 的專家，只是同時學習 React 和 JavaScript 會比較困難。

我們建議先看[這裏的 JavaScript 概覽](#)來測試自己的認識程度。這需要花費大概 30 分鐘到 1 小時，但這會增加你學習 React 的信心。

提示

當你遇到 JavaScript 的困難時，[MDN](#) 和 [javascript.info](#) 會是非常好的查閱資源。你也可以在[社群支援論壇](#)上尋找幫助。

實用指南

如果你喜歡**動手做**，請參考我們的[實用指南](#)。我們會利用 React 實作一個井字遊戲。你可能對遊戲開發不感興趣，因而很想跳過這個部份，但請給它一個機會。在這指南所學習的技巧，將會是實作任何 React 應用程式的重要基礎，掌握到它們會讓你更了解 React。

按部就班學習指南

如果你喜歡**按部就班學習概念**，我們的[主要概念指南](#)會是一個很好的起點。指南裏每個章節都是建基於上一個章節的學習內容，這樣你就不會錯過任何內容。

用 React 思考

許多 React 使用者都表示在閱讀了[用 React 思考](#)後對 React 恍然大悟。它可能是最源遠流長的 React 指南，至今依然極具價值。

課程推薦

相比官方文件，有時候大家會覺得第三方的書籍和影片教學對學習更有幫助。因此我們整合了一系列的[課程推薦](#)，有些課程更是免費的。

進階概念

當你對[主要概念](#)有所掌握，並嘗試使用了一點 React，你可能會對比較進階的課題感興趣。進階指南會介紹一些強大但不常用的 React 特點，例如 [context](#) 和 [refs](#)。

API 參考

當你想了解更多有關特定 React API 的資訊時，這章節會非常有用。例如 [React.Component API 參考](#) 會解構 [setState\(\)](#) 是怎樣運作，以及介紹不同生命週期方法的使用方法。

術語表與常見問題

[術語表](#)包含了 React 文件裏最常見的術語。為了解答大家對於一些常見課題的疑問，包括 [發送 AJAX 請求](#)、[component state](#) 和[文件結構](#)，我們因此也提供了一個常見問題的章節。

了解最新消息

[React 部落格](#)是 React 團隊發佈更新消息的官方渠道。任何重要消息，包括更新日誌或棄用通知，都會在這裏先發佈。

你也可以在 Twitter 追蹤 [@reactjs 帳號](#)，只閱讀官方部落格也不會令你錯過重要消息。

並非所有 React 版本發佈值得我們在部落格裡發佈文章，但你可以在 [React 的程式碼倉庫中的 CHANGELOG.md 文件](#) 或 [Release](#) 頁面，找到每個版本發佈的詳細更新日誌。

版本化文件

官方文件會一直與最新穩定版本的 React 保持同步。至 React 16 以來，你可以在這[另一頁](#)找到舊版本的 React 文件。注意，舊版本的文件是取至該版本發佈時的狀態，並不會有持續的更新。

找不到相關資訊？

如果你找不到相關的資訊，或是對部份的內容有疑惑，請連同你的建議，在[此文件倉庫提出 issue](#) 或是在 Twitter 上提及 [@reactjs 帳號](#)。我們樂意聆聽你的意見。

事件處理

使用 React element 處理事件跟使用 DOM element 處理事件是十分相似的。它們有一些語法上的差異：

- 事件的名稱在 React 中都是 camelCase，而在 HTML DOM 中則是小寫。
- 事件的值在 JSX 中是一個 function，而在 HTML DOM 中則是一個 string。

例如，在 HTML 中的語法：

```
<button onclick="activateLasers()">
  Activate Lasers
</button>
```

和在 React 中的語法有些微的不同：

```
<button onClick={activateLasers}>
  Activate Lasers
</button>
```

另外一個差異是，在 React 中，你不能夠使用 `return false` 來避免瀏覽器預設行為。你必須明確地呼叫 `preventDefault`。例如，在純 HTML 中，若要避免連結開啟新頁的預設功能，你可以這樣寫：

```
<form onsubmit="console.log('You clicked submit.'); return false">
  <button type="submit">Submit</button>
</form>
```

在 React 中，你則可以這樣寫：

```
function Form() {
  function handleSubmit(e) {
    e.preventDefault();
    console.log('You clicked submit.');
  }

  return (
    <form onSubmit={handleSubmit}>
      <button type="submit">Submit</button>
    </form>
  );
}
```

在這裡，`e` 是一個 synthetic 事件 (synthetic event)。React 根據 [W3C 規範](#)來定義這些 synthetic 事件，React event 與 native event 工作的方式不盡然相同。若想了解更多這方面的資訊，請參考 [SyntheticEvent](#)。

當使用 React 時，你不需要在建立一個 DOM element 後再使用 `addEventListener` 來加上 listener。你只需要在這個 element 剛開始被 render 時就提供一個 listener。

當你使用 [ES6 class](#) 來定義 Component 時，常見的慣例是把 event handler 當成那個 class 的方法。例如，這個 `Toggle` Component 會 render 一個按鈕，讓使用者可以轉換 state 中的「開」與「關」：

```

class Toggle extends React.Component {
  constructor(props) {
    super(props);
    this.state = {isToggleOn: true};

    // 為了讓 `this` 能在 callback 中被使用，這裡的綁定是必要的：
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    this.setState(prevState => ({
      isToggleOn: !prevState.isToggleOn
    }));
  }

  render() {
    return (
      <button onClick={this.handleClick}>
        {this.state.isToggleOn ? 'ON' : 'OFF'}
      </button>
    );
  }
}

```

在 [CodePen 上試試看吧！](#)

請特別注意 `this` 在 JSX callback 中的意義。在 JavaScript 中，class 的方法在預設上是沒有被綁定（bound）的。如果你忘了綁定 `this.handleClick` 並把它傳遞給 `onClick` 的話，`this` 的值將會在該 function 被呼叫時變成 `undefined`。

這並非是 React 才有的行為，而是 [function 在 JavaScript 中的運作模式](#)。總之，當你使用一個方法，卻沒有在後面加上 `()` 之時（例如當你使用 `onClick={this.handleClick}` 時），你應該要綁定這個方法。

如果呼叫 `bind` 對你來說很麻煩的話，你可以用別的方式。你可以用 public class field 來正確的綁定 callback：

```

class LoggingButton extends React.Component {
  // 這個語法確保 `this` 是在 handleClick 中被綁定。
  handleClick = () => {
    console.log('this is:', this);
  };

  render() {
    return (
      <button onClick={this.handleClick}>
        Click me
      </button>
    );
  }
}

```

這個語法在 [Create React App](#) 中是預設成可行的。

如果你並沒有使用 class field 的語法的話，你則可以在 callback 中使用 [arrow function](#)：

```

class LoggingButton extends React.Component {
  handleClick() {
    console.log('this is:', this);
  }
}

```

```

render() {
  // 這個語法確保 `this` 是在 handleClick 中被綁定：
  return (
    <button onClick={() => this.handleClick()}>
      Click me
    </button>
  );
}

```

這個語法的問題是每一次 `LoggingButton` `render` 的時候，就會建立一個不同的 callback。大多時候，這是無所謂的。然而，如果這個 callback 被當作一個 prop 傳給下層的 component 的話，其他的 component 也許會做些多餘的 re-render。原則上來說，我們建議在 `constructor` 內綁定，或使用 class field 語法，以避免這類的性能問題。

將參數傳給 Event Handler

在一個迴圈中，我們常常會需要傳遞一個額外的參數給 event handler。例如，如果 `id` 是每一行的 ID 的話，下面兩種語法都可行：

```

<button onClick={(e) => this.deleteRow(id, e)}>Delete Row</button>
<button onClick={this.deleteRow.bind(this, id)}>Delete Row</button>

```

以上這兩行程式是相同的。一個使用 `arrow functions`，另一個則使用了 `Function.prototype.bind`。

以這兩個例子來說，`e` 這個參數所代表的 React 事件將會被當作 ID 之後的第二個參數被傳遞下去。在使用 `arrow function` 時，我們必須明確地將它傳遞下去，但若使用 `bind` 語法，未來任何的參數都將會自動被傳遞下去。

Hello World

React 最簡單的範例看起來像是：

```
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<h1>Hello, world!</h1>);
```

在頁面上顯示「Hello, world!」。

[在 CodePen 上試試看吧！](#)

點擊上方連結開啟線上編輯器。請隨意的變更程式碼並觀察它們的輸出變化。這個指南大部分的範例都可以像這樣編輯。

如何閱讀本指南

在這份指南中，我們會研究 React 應用程式的基本組成：元素與組件。一旦理解它們之後，你可以從小巧且可覆用的 component，拼湊出複雜的應用程式。

Tip

本篇指南設計給喜歡**按部就班**的朋友。若喜歡動手做的朋友，請參考我們的[實用指南](#)。你可能會發現本篇指南和教學課程其實是相輔相成的。

本篇是 React 主要概念的第一章節。你可以在側邊的導航欄中找到所有章節的列表。如果你從手機裝置上瀏覽，可以透過點擊螢幕右下角的按鈕來瀏覽選單。

本篇指南的每個章節都是基於前一章節的知識。**你可以按照側邊欄中顯示的順序閱讀「主要概念」指南章節，來了解 React 大部分的內容。**舉例來說，這篇章節的下一章是[「JSX 入門」](#)。

需要的知識

React 是一個 JavaScript 的函式庫，而且我們假設你有一些 JavaScript 的基礎知識。**如果你感到沒有自信的話，我們推薦讀過一遍這個 [JavaScript 教學課程](#)來確認你的知識水平如何，確保你可以跟上本篇指南。**這需要花費大概 30 分鐘到 1 小時，但你不會覺得你在學習 React，同時又在學習 JavaScript。

注意

本篇指南偶爾會使用較新的 JavaScript 語法。若近年內沒有接觸過 JavaScript，只要看一下[這三個心法](#)即可。

馬上開始吧！

繼續滾動頁面至底部，你會在網頁的右下角找到進入[下一章的連結](#)。

react

[Go to TOC](#)

Higher-Order Components

A higher-order component (HOC) is an advanced technique in React for reusing component logic. HOCs are not part of the React API, per se. They are a pattern that emerges from React's compositional nature.

Concretely, a **higher-order component is a function that takes a component and returns a new component.**

```
const EnhancedComponent = higherOrderComponent(WrappedComponent);
```

Whereas a component transforms props into UI, a higher-order component transforms a component into another component.

HOCs are common in third-party React libraries, such as Redux's `connect` and Relay's `createFragmentContainer`.

In this document, we'll discuss why higher-order components are useful, and how to write your own.

Use HOCs For Cross-Cutting Concerns

Note

We previously recommended mixins as a way to handle cross-cutting concerns. We've since realized that mixins create more trouble than they are worth. [Read more](#) about why we've moved away from mixins and how you can transition your existing components.

Components are the primary unit of code reuse in React. However, you'll find that some patterns aren't a straightforward fit for traditional components.

For example, say you have a `CommentList` component that subscribes to an external data source to render a list of comments:

```
class CommentList extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {
      // "DataSource" is some global data source
      comments: DataSource.getComments()
    };
  }

  componentDidMount() {
    // Subscribe to changes
    DataSource.addChangeListener(this.handleChange);
  }

  componentWillUnmount() {
```

```
// Clean up listener
DataSource.removeChangeListener(this.handleChange);
}

handleChange() {
// Update component state whenever the data source changes
this.setState({
  comments: DataSource.getComments()
});
}

render() {
  return (
    <div>
      {this.state.comments.map((comment) => (
        <Comment comment={comment} key={comment.id} />
      ))}
    </div>
  );
}
}
```

Later, you write a component for subscribing to a single blog post, which follows a similar pattern:

```
class BlogPost extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {
      blogPost: DataSource.getBlogPost(props.id)
    };
  }

  componentDidMount() {
    DataSource.addChangeListener(this.handleChange);
  }

  componentWillUnmount() {
    DataSource.removeChangeListener(this.handleChange);
  }

  handleChange() {
    this.setState({
      blogPost: DataSource.getBlogPost(this.props.id)
    });
  }

  render() {
    return <TextBlock text={this.state.blogPost} />;
  }
}
```

`CommentList` and `BlogPost` aren't identical — they call different methods on `DataSource`, and they render different output. But much of their implementation is the same:

- On mount, add a change listener to `DataSource`.
- Inside the listener, call `setState` whenever the data source changes.
- On unmount, remove the change listener.

You can imagine that in a large app, this same pattern of subscribing to `DataSource` and calling `setState` will occur over and over again. We want an abstraction that allows us to define this logic in a single place and share it across many components. This is where higher-order components excel.

We can write a function that creates components, like `CommentList` and `BlogPost`, that subscribe to `DataSource`. The function will accept as one of its arguments a child component that receives the subscribed data as a prop. Let's call the function `withSubscription`:

```
const CommentListWithSubscription = withSubscription(
  CommentList,
  (DataSource) => DataSource.getComments()
);

const BlogPostWithSubscription = withSubscription(
  BlogPost,
  (DataSource, props) => DataSource.getBlogPost(props.id)
);
```

The first parameter is the wrapped component. The second parameter retrieves the data we're interested in, given a `DataSource` and the current props.

When `CommentListWithSubscription` and `BlogPostWithSubscription` are rendered, `CommentList` and `BlogPost` will be passed a `data` prop with the most current data retrieved from `DataSource`:

```
// This function takes a component...
function withSubscription(WrappedComponent, selectData) {
  // ...and returns another component...
  return class extends React.Component {
    constructor(props) {
      super(props);
      this.handleChange = this.handleChange.bind(this);
      this.state = {
        data: selectData(DataSource, props)
      };
    }

    componentDidMount() {
      // ... that takes care of the subscription...
      DataSource.addChangeListener(this.handleChange);
    }

    componentWillUnmount() {
      DataSource.removeChangeListener(this.handleChange);
    }

    handleChange() {
      this.setState({
        data: selectData(DataSource, this.props)
      });
    }

    render() {
      // ... and renders the wrapped component with the fresh data!
      // Notice that we pass through any additional props
      return <WrappedComponent data={this.state.data} {...this.props} />;
    }
  };
}
```

```

    }
};

}

```

Note that a HOC doesn't modify the input component, nor does it use inheritance to copy its behavior. Rather, a HOC *composes* the original component by *wrapping* it in a container component. A HOC is a pure function with zero side-effects.

And that's it! The wrapped component receives all the props of the container, along with a new prop, `data`, which it uses to render its output. The HOC isn't concerned with how or why the data is used, and the wrapped component isn't concerned with where the data came from.

Because `withSubscription` is a normal function, you can add as many or as few arguments as you like. For example, you may want to make the name of the `data` prop configurable, to further isolate the HOC from the wrapped component. Or you could accept an argument that configures `shouldComponentUpdate`, or one that configures the data source. These are all possible because the HOC has full control over how the component is defined.

Like components, the contract between `withSubscription` and the wrapped component is entirely props-based. This makes it easy to swap one HOC for a different one, as long as they provide the same props to the wrapped component. This may be useful if you change data-fetching libraries, for example.

Don't Mutate the Original Component. Use Composition.

Resist the temptation to modify a component's prototype (or otherwise mutate it) inside a HOC.

```

function logProps(InputComponent) {
  InputComponent.prototype.componentDidUpdate = function(prevProps) {
    console.log('Current props: ', this.props);
    console.log('Previous props: ', prevProps);
  };
  // The fact that we're returning the original input is a hint that it has
  // been mutated.
  return InputComponent;
}

// EnhancedComponent will log whenever props are received
const EnhancedComponent = logProps(InputComponent);

```

There are a few problems with this. One is that the input component cannot be reused separately from the enhanced component. More crucially, if you apply another HOC to `EnhancedComponent` that *also* mutates `componentDidUpdate`, the first HOC's functionality will be overridden! This HOC also won't work with function components, which do not have lifecycle methods.

Mutating HOCs are a leaky abstraction—the consumer must know how they are implemented in order to avoid conflicts with other HOCs.

Instead of mutation, HOCs should use composition, by wrapping the input component in a container component:

```

function logProps(WrappedComponent) {
  return class extends React.Component {
    componentDidUpdate(prevProps) {
      console.log('Current props: ', this.props);
      console.log('Previous props: ', prevProps);
    }
    render() {
      // Wraps the input component in a container, without mutating it. Good!
      return <WrappedComponent {...this.props} />;
    }
  }
}

```

This HOC has the same functionality as the mutating version while avoiding the potential for clashes. It works equally well with class and function components. And because it's a pure function, it's composable with other HOCs, or even with itself.

You may have noticed similarities between HOCs and a pattern called **container components**. Container components are part of a strategy of separating responsibility between high-level and low-level concerns. Containers manage things like subscriptions and state, and pass props to components that handle things like rendering UI. HOCs use containers as part of their implementation. You can think of HOCs as parameterized container component definitions.

Convention: Pass Unrelated Props Through to the Wrapped Component

HOCs add features to a component. They shouldn't drastically alter its contract. It's expected that the component returned from a HOC has a similar interface to the wrapped component.

HOCs should pass through props that are unrelated to its specific concern. Most HOCs contain a render method that looks something like this:

```

render() {
  // Filter out extra props that are specific to this HOC and shouldn't be
  // passed through
  const { extraProp, ...passThroughProps } = this.props;

  // Inject props into the wrapped component. These are usually state values or
  // instance methods.
  const injectedProp = someStateOrInstanceMethod;

  // Pass props to wrapped component
  return (
    <WrappedComponent
      injectedProp={injectedProp}
      {...passThroughProps}
    />
  );
}

```

This convention helps ensure that HOCs are as flexible and reusable as possible.

Convention: Maximizing Composability

Not all HOCs look the same. Sometimes they accept only a single argument, the wrapped component:

```
const NavbarWithRouter = withRouter(Navbar);
```

Usually, HOCs accept additional arguments. In this example from Relay, a config object is used to specify a component's data dependencies:

```
const CommentWithRelay = Relay.createContainer(Comment, config);
```

The most common signature for HOCs looks like this:

```
// React Redux's `connect`
const ConnectedComment = connect(commentSelector, commentActions)(CommentList);
```

What?! If you break it apart, it's easier to see what's going on.

```
// connect is a function that returns another function
const enhance = connect(commentListSelector, commentListActions);
// The returned function is a HOC, which returns a component that is connected
// to the Redux store
const ConnectedComment = enhance(CommentList);
```

In other words, `connect` is a higher-order function that returns a higher-order component!

This form may seem confusing or unnecessary, but it has a useful property. Single-argument HOCs like the one returned by the `connect` function have the signature `Component => Component`. Functions whose output type is the same as its input type are really easy to compose together.

```
// Instead of doing this...
const EnhancedComponent = withRouter(connect(commentSelector)(WrappedComponent))

// ... you can use a function composition utility
// compose(f, g, h) is the same as (...args) => f(g(h(...args)))
const enhance = compose(
  // These are both single-argument HOCs
  withRouter,
  connect(commentSelector)
)
const EnhancedComponent = enhance(WrappedComponent)
```

(This same property also allows `connect` and other enhancer-style HOCs to be used as decorators, an experimental JavaScript proposal.)

The `compose` utility function is provided by many third-party libraries including `lodash` (as `lodash.flowRight`), `Redux`, and `Ramda`.

Convention: Wrap the Display Name for Easy Debugging

The container components created by HOCs show up in the [React Developer Tools](#) like any other component. To ease debugging, choose a display name that communicates that it's the result of a HOC.

The most common technique is to wrap the display name of the wrapped component. So if your higher-order component is named `withSubscription`, and the wrapped component's display name is `CommentList`, use the display name `WithSubscription(CommentList)`:

```
function withSubscription(WrappedComponent) {
  class WithSubscription extends React.Component {/* ... */}
  WithSubscription.displayName =
    `WithSubscription(${getDisplayName(WrappedComponent)})`;
  return WithSubscription;
}

function getDisplayName(WrappedComponent) {
  return WrappedComponent.displayName || WrappedComponent.name || 'Component';
}
```

Caveats

Higher-order components come with a few caveats that aren't immediately obvious if you're new to React.

Don't Use HOCs Inside the render Method

React's diffing algorithm (called [Reconciliation](#)) uses component identity to determine whether it should update the existing subtree or throw it away and mount a new one. If the component returned from `render` is identical (`==`) to the component from the previous render, React recursively updates the subtree by diffing it with the new one. If they're not equal, the previous subtree is unmounted completely.

Normally, you shouldn't need to think about this. But it matters for HOCs because it means you can't apply a HOC to a component within the `render` method of a component:

```
render() {
  // A new version of EnhancedComponent is created on every render
  // EnhancedComponent1 !== EnhancedComponent2
  const EnhancedComponent = enhance(MyComponent);
  // That causes the entire subtree to unmount/remount each time!
  return <EnhancedComponent />;
}
```

The problem here isn't just about performance — remounting a component causes the state of that component and all of its children to be lost.

Instead, apply HOCs outside the component definition so that the resulting component is created only once. Then, its identity will be consistent across renders. This is usually what you want, anyway.

In those rare cases where you need to apply a HOC dynamically, you can also do it inside a component's lifecycle methods or its constructor.

Static Methods Must Be Copied Over

Sometimes it's useful to define a static method on a React component. For example, Relay containers expose a static method `getFragment` to facilitate the composition of GraphQL fragments.

When you apply a HOC to a component, though, the original component is wrapped with a container component. That means the new component does not have any of the static methods of the original component.

```
// Define a static method
WrappedComponent.staticMethod = function() {/*...*/}
// Now apply a HOC
const EnhancedComponent = enhance(WrappedComponent);

// The enhanced component has no static method
typeof EnhancedComponent.staticMethod === 'undefined' // true
```

To solve this, you could copy the methods onto the container before returning it:

```
function enhance(WrappedComponent) {
  class Enhance extends React.Component {/*...*/}
  // Must know exactly which method(s) to copy :(
  Enhance.staticMethod = WrappedComponent.staticMethod;
  return Enhance;
}
```

However, this requires you to know exactly which methods need to be copied. You can use [hoist-non-react-statics](#) to automatically copy all non-React static methods:

```
import hoistNonReactStatic from 'hoist-non-react-statics';
function enhance(WrappedComponent) {
  class Enhance extends React.Component {/*...*/}
  hoistNonReactStatic(Enhance, WrappedComponent);
  return Enhance;
}
```

Another possible solution is to export the static method separately from the component itself.

```
// Instead of...
MyComponent.someFunction = someFunction;
export default MyComponent;

// ...export the method separately...
export { someFunction };

// ...and in the consuming module, import both
import MyComponent, { someFunction } from './MyComponent.js';
```

Refs Aren't Passed Through

While the convention for higher-order components is to pass through all props to the wrapped component, this does not work for refs. That's because `ref` is not really a prop — like `key`, it's handled specially by React. If you add a ref to an element whose component is the result of a HOC, the ref refers to an instance of the outermost container component, not the wrapped component.

The solution for this problem is to use the `React.forwardRef` API (introduced with React 16.3). [Learn more about it in the forwarding refs section.](#)

打造你自己的 Hook

Hook 是 React 16.8 中增加的新功能。它讓你不必寫 class 就能使用 state 以及其他 React 的功能。

打造你自己的 Hook 可以將 component 邏輯提取到可重複使用的 function 中。

當我們在學習關於[使用 Effect Hook](#) 時，我們從一個聊天應用程式看到這個 component，顯示一條訊息說明朋友是否還在線上：

```
import React, { useState, useEffect } from 'react';

function FriendStatus(props) {
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}
```

現在，讓我們的聊天應用程式也有一個聯繫列表，而且想要透過綠色來 render 在線上的使用者名稱。我們可以從上方複製並貼上相似的邏輯到 `FriendListItem` component，但它並不理想：

```
import React, { useState, useEffect } from 'react';

function FriendListItem(props) {
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  return (
    <li style={{ color: isOnline ? 'green' : 'black' }}>
      {props.friend.name}
    </li>
  );
}
```

相反的，我們想要在 `FriendStatus` 和 `FriendList` 共享這個邏輯。

習慣上，我們有兩種常用的方式在 component 來共享 stateful 邏輯：[render props](#) 和 [higher-order component](#)。我們現在將看到 Hook 如何在不強迫 tree 加入更多 component 的情況下，解決許多相同的問題。

提取一個自定義的 Hook

當我們想要共享邏輯在兩個 JavaScript function 之間時，我們提取它成為第三個 function。Component 和 Hook 兩者都是 function，所以這也適用於它們！

一個自定義的 Hook 是以「`use`」為開頭命名的 JavaScript function，而且它可能也呼叫其他的 Hook。例如，以下是我第一個字定義的 `useFriendStatus` Hook：

```
import { useState, useEffect } from 'react';

function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    ChatAPI.subscribeToFriendStatus(friendID, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(friendID, handleStatusChange);
    };
  });

  return isOnline;
}
```

這裡沒有新加入的東西 -- 邏輯是複製於上方的 component。就像在 component 內一樣，確保只在自定義的 Hook 頂層無條件的呼叫其他 Hook。

不像 React component，一個自定義的 Hook 不需要一個特定的宣告。我們可以決定它需要接受什麼參數，以及它應該回傳什麼（如果有的話）。換句話說，它就像一個普通的 function。它的命名開頭應該總是為 `use`，所以你可以一眼就看出 [Hook 的規則](#)適用於它。

我們的 `useFriendStatus` Hook 目的是訂閱我們朋友的狀態。這也是為什麼它接受 `friendID` 作為一個參數，並回傳朋友是否在線上：

```
function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);

  // ...

  return isOnline;
}
```

現在，讓我們來看如何使用我們自定義的 Hook。

使用一個自定義的 Hook

在一開始的時候，我們的目標是從 `FriendStatus` 和 `FriendListItem` component 中移除重複的邏輯。這兩者都想要知道朋友是否在線上。

現在，我們提取了邏輯到 `useFriendStatus` hook，我們可以使用它：

```
function FriendStatus(props) {
  const isOnline = useFriendStatus(props.friend.id);

  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}

function FriendListItem(props) {
  const isOnline = useFriendStatus(props.friend.id);

  return (
    <li style={{ color: isOnline ? 'green' : 'black' }}>
      {props.friend.name}
    </li>
  );
}
```

這個程式碼相等於原始的範例嗎？是的，它們執行的方式是相同的。如果你仔細看的話，你會注意到我們沒有改變任何的行為。我們所做的只是在兩個 function 中提取共同的程式碼讓它成為一個獨立的 function。自定義的 Hook 是自然遵循 Hook 設計的規範，而不是 React 的功能。

請問我必須以「`use`」開頭命名我自定義的 Hook 嗎？請這麼做。這個規範非常的重要。沒有它的話，我們無法自動的檢查違反 Hook 規則的行為，因為我們無法判斷某個 function 中是否包含對 Hook 的呼叫。

請問兩個 component 使用相同的 Hook 是共享 state 的嗎？不是的。自定義的 Hook 有一個機制重複使用 `stateful` 邏輯（例如設定訂閱並記住目前的值），但每次你使用自定義的 Hook 時，所有內部的 state 和 effect 都是完全獨立的。

自定義的 Hook 是如何隔離 state 的？ 每個呼叫 Hook 的都會得到獨立的 state。因為我們直接呼叫 `useFriendStatus`，從 React 的角度來看，我們的 component 只呼叫 `useState` 和 `useEffect`。正如我們之前所理解的，我們可以在一個 component 呼叫 `useState` 和 `useEffect` 多次，而且它們都是完全獨立的。

提示：在 Hook 之間傳遞資訊

由於 Hook 是 function，我們可以在它們之間傳遞資訊。

為了說明這點，我們將使用我們假設的聊天範例中的另一個 component。這是一個聊天訊息收件人的選擇器，顯示目前選擇的朋友是否在線上：

```
const friendList = [
];

function ChatRecipientPicker() {
  const [recipientID, setRecipientID] = useState(1);
  const isRecipientOnline = useFriendStatus(recipientID);
```

```

return (
  <>
    <Circle color={isRecipientOnline ? 'green' : 'red'} />
    <select
      value={recipientID}
      onChange={e => setRecipientID(Number(e.target.value))}>
    >
      {friendList.map(friend => (
        <option key={friend.id} value={friend.id}>
          {friend.name}
        </option>
      ))}
    </select>
  </>
);
}

```

我們將目前選擇的朋友 ID 存在 `recipientID` state 變數中，如果使用者在 `<select>` 選擇器中選擇不同的使用者它將會更新。

因為叫了 `useState` Hook 為我們提供了 `recipientID` state 變數的最新值，我們可以將它作為變數傳遞到我們自定義的 `useFriendStatus`：

```

const [recipientID, setRecipientID] = useState(1);
const isRecipientOnline = useFriendStatus(recipientID);

```

這讓我們知道目前選擇的朋友是否在線上。如果我們選擇不同的朋友並更新 `recipientID` state 變數，我們的 `useFriendStatus` Hook 將會從先前選擇的朋友中取消訂閱，並訂閱最新選擇的狀態。

useYourImagination() {#useyourimagination}

自定義的 Hook 提供了共享邏輯的靈活性，這在以前的 React component 是不可能的。你可以撰寫自定義的 Hook 涵蓋廣泛的場景，像是表格處理、動畫、陳述式訂閱 (Declarative Subscription)、計時器還有更多我們沒有考慮過的。更重要的是，你可以打造與 React 的內建一樣易於使用的 Hook。

盡量不要過早地加入抽象。現在 function component 可以做更多的事，在你 codebase 中的 function component 程式碼平均可能都會變得更長。這都是正常的 -- 不要覺得你必須馬上把它拆分成 Hook。但我們也鼓勵你開始發現自定義的 Hook 可以隱藏簡單 interface 背後的複雜邏輯情況，或者幫忙解開一個混亂的 component。

例如，你可能有一個複雜的 component，它包含許多以一個特殊目的 (ad-hoc) 方式來管理的 local state。`useState` 沒辦法讓更新邏輯集中化，所以你可能更傾向將其寫為 Redux 的 reducer：

```

function todosReducer(state, action) {
  switch (action.type) {
    case 'add':
      return [...state, {
        text: action.text,
        completed: false
      }];
    // ... 其他 action ...
    default:
      return state;
  }
}

```

Reducer 是非常方便於獨立測試的，而且可以表達複雜的更新邏輯。如果有需要的話，你可以將它們拆成更小的 reducer。然而，你可能也喜歡使用 React local state 的好處，或者你不想要安裝其他的函式庫。

那麼，如果我們可以撰寫一個 `useReducer` Hook，讓我們用 reducer 管理 component 的 *local state* 呢？它的簡化版本看起來如下：

```
function useReducer(reducer, initialState) {
  const [state, setState] = useState(initialState);

  function dispatch(action) {
    const nextState = reducer(state, action);
    setState(nextState);
  }

  return [state, dispatch];
}
```

現在我們在其他的 component 使用它，讓 reducer 驅動它的 state 管理：

```
function Todos() {
  const [todos, dispatch] = useReducer(todosReducer, []);

  function handleAddClick(text) {
    dispatch({ type: 'add', text });
  }

  // ...
}
```

在複雜的 component 中使用 reducer 管理 local state 的需求很常見，我們已經將 `useReducer` Hook 內建在 React 中。你可以在 [Hooks API 參考](#)中找到它與其他內建的 Hook。

使用 Effect Hook

Hook 是 React 16.8 中增加的新功能。它讓你不必寫 class 就能使用 state 以及其他 React 的功能。

Effect Hook 讓你可以使用 function component 中的 side effect：

```
import React, { useState, useEffect } from 'react';

function Example() {
  const [count, setCount] = useState(0);

  // 相似於 componentDidMount 和 componentDidUpdate:
  useEffect(() => {
    // 使用瀏覽器 API 更新文件標題
    document.title = `You clicked ${count} times`;
  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

這個範例基於[上一頁的計數器範例](#)，但是我們增加了一個新的功能：我們把網頁標題設定為包含點擊次數的自訂訊息。

資料 `fetch`、設定 `subscription`、或手動改變 React component 中的 DOM 都是 side effect 的範例。無論你是否習慣將這些操作稱為「side effect」（或簡稱「effect」），你之前可能已經在 component 中執行了這些操作。

提示

如果你熟悉 React class 的生命週期方法，你可以把 `useEffect` 視為 `componentDidMount`，`componentDidUpdate` 和 `componentWillUnmount` 的組合。

React component 有兩種常見的 side effect：一種不需要執行清除，另一種則需要。讓我們仔細看看它們區別。

無需清除的 Effect

有時候，我們希望在 React 更新 DOM 之後執行一些額外的程式碼。網路請求、手動變更 DOM、和 logging，它們都是無需清除 effect 的常見範例。我們之所以這樣說，是因為我們可以執行它們，並立即忘記它們。讓我們比較一下 class 和 Hooks 如何讓我們表達這樣的 side effect。

使用 Class 的範例

在 React class component 中，`render` 方法本身不應該觸發 side effect。這太早了 — 我們通常希望在 React 更新 DOM 之後執行我們的 effect。

這就是為什麼在 React class 中，我們將 side effect 放入 `componentDidMount` 和 `componentDidUpdate`。回到我們的範例，這是一個 React class component 的計數器，它在 React 對 DOM 進行變更後立即更新網頁標題：

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  componentDidMount() {
    document.title = `You clicked ${this.state.count} times`;
  }

  componentDidUpdate() {
    document.title = `You clicked ${this.state.count} times`;
  }

  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Click me
        </button>
      </div>
    );
  }
}
```

注意我們如何必須在 `class` 中複製這兩個生命週期方法之間的程式碼。

這是因為在許多情況下，我們希望執行相同的 side effect，無論 component 是剛被 mount 還是已經被更新。概念上，我們希望它在每次 render 之後發生 — 但是 React class component 沒有這樣的方法。我們可以提取一個單獨的方法，但我們仍然需要在兩個地方呼叫它。

現在來看看我們可以如何使用 `useEffect` Hook 做同樣的事情。

使用 Hook 的範例

我們已經在本頁頂部看到了這個範例，但讓我們來仔細看看它：

```
import React, { useState, useEffect } from 'react';

function Example() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

```

        </div>
    );
}

```

useEffect 有什麼作用？ 透過使用這個 Hook，你告訴 React 你的 component 需要在 render 後做一些事情。React 將記住你傳遞的 function（我們將其稱為「effect」），並在執行 DOM 更新之後呼叫它。在這個 effect 中，我們設定了網頁的標題，但我們也可以執行資料提取或呼叫其他命令式 API。

為什麼在 component 內部呼叫 useEffect ？ 在 component 中放置 `useEffect` 讓我們可以直接從 effect 中存取 `count` state 變數（或任何 props）。我們不需要特殊的 API 來讀取它 — 它已經在 function 範圍內了。Hook 摊抱 JavaScript closure，並避免在 JavaScript 已經提供解決方案的情況下引入 React 特定的 API。

每次 render 後都會執行 useEffect 嗎？ 是的！預設情況下，它在第一個 render 和隨後每一個更新之後執行。（我們稍後會談到[如何自定義](#)。）你可能會發現把 effect 想成發生在「render 之後」更為容易，而不是考慮「mount」和「更新」。React 保證 DOM 在執行 effect 時已被更新。

詳細說明

現在我們對 effect 有了更多的了解，應該可以理解這幾行程式碼：

```

function Example() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });
}

```

我們宣告 `count` state 變數，然後告訴 React 我們需要使用一個 effect。我們將一個 function 傳入給 `useEffect` Hook。我們傳入的這個 function 就是我們的 effect。在 effect 內部，我們使用瀏覽器 API `document.title` 設定了網頁標題。我們可以讀取 effect 中最新的 `count`，因為它在我們 function 的範圍內。當 React render 我們的 component 時，它會記住我們使用的 effect，然後在更新 DOM 後執行我們的 effect。每次 render 都是這樣，包括第一次。

有經驗的 JavaScript 開發人員可能會注意到，傳遞給 `useEffect` 的 function 在每次 render 時都會有所不同。這是刻意的。實際上，這是讓我們可以從 effect 內部讀取 `count` 數值，且不必擔心數值過時的原因。每次重新 render 時，我們都會安排一個 *different* effect 來替代上一個。在某種程度上，這使 effect 的行為更像是 render 結果的一部分 — 每個 effect 都「屬於」特定的 render。我們將在本頁稍後更清楚地看到為什麼這很有用。

提示

與 `componentDidMount` 或 `componentDidUpdate` 不同，使用 `useEffect` 安排的 effect 不會阻止瀏覽器更新螢幕。這使你的應用程式感覺起來響應更快。大多數 effect 不需要同步發生。在少見的需要同步發生的情況下（例如測量 layout），有另外一個 `useLayoutEffect` Hook，它的 API 與 `useEffect` 相同。

需要清除的 Effect

先前，我們理解了怎樣表達不需要任何清除的 side effect。但是，有些 effect 需要。例如，**我們可能想要設定**對某些外部資料來源的 subscription。在這種情況下，請務必進行清除，以免造成 memory leak！讓我們比較一下我們可以如何用 class 和 Hook 做到這一點。

使用 Class 的範例

在 React class 中，你通常會在 `componentDidMount` 中設定一個 subscription，然後在 `componentWillUnmount` 中把它清除。例如，假設我們有一個 `ChatAPI` module 可讓我們訂閱朋友的線上狀態。我們可能會這樣用 class 來訂閱和顯示該狀態：

```
class FriendStatus extends React.Component {
  constructor(props) {
    super(props);
    this.state = { isOnline: null };
    this.handleStatusChange = this.handleStatusChange.bind(this);
  }

  componentDidMount() {
    ChatAPI.subscribeToFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }

  componentWillUnmount() {
    ChatAPI.unsubscribeFromFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }

  handleStatusChange(status) {
    this.setState({
      isOnline: status.isOnline
    });
  }

  render() {
    if (this.state.isOnline === null) {
      return 'Loading...';
    }
    return this.state.isOnline ? 'Online' : 'Offline';
  }
}
```

請注意 `componentDidMount` 和 `componentWillUnmount` 需要如何相互呼應。生命週期方法迫使我們拆開這個邏輯，即使概念上它們的程式碼都與同一個 effect 相關。

注意

敏銳的讀者可能會注意到，要做到完全正確，這個範例還需要 `componentDidUpdate`。我們現在將暫時忽略這一點，但在這頁的稍後部分我們會再次討論這點。

使用 Hook 的範例

讓我們看看如何使用 Hook 撰寫這個 component。

你可能會認為我們需要一個單獨的 effect 來執行清除。但是新增和移除 subscription 的程式碼緊密相關，因此 `useEffect` 的設計在將其保持在一起。如果你的 effect 回傳了一個 function，React 將在需要清除時執行它：

```
import React, { useState, useEffect } from 'react';

function FriendStatus(props) {
  const [isOnline, setIsOnline] = useState(null);

  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    // 指定如何在這個 effect 之後執行清除：
    return function cleanup() {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}
```

為什麼我們從 effect 中回傳一個 function？ 這是 effect 的可選清除機制。每個 effect 都可以回傳一個會在它之後執行清除的 function。這使我們可以把新增和移除 subscription 的邏輯彼此保持靠近。它們都屬於同一個 effect！

React 到底什麼時候會清除 effect？ 在 component unmount 時，React 會執行清除。但是，正如我們之前看到的，effect 會在每個 render 中執行，而不僅僅是一次。這是為什麼 React 還可以在下次執行 effect 之前清除前一個 render 的 effect 的原因。我們會在下面討論**為什麼這有助於避免 bug** 以及**如果出現效能問題，如何選擇退出此行為**。

注意

我們不必從 effect 中回傳命名了的 function。我們在這裡將其稱為 `cleanup` 以明確它的目的，但是你可以回傳 arrow function 或者叫它別的名字。

總結

我們看到了 `useEffect` 可以讓我們在 component render 後表達不同類型的 side effect。某些 effect 可能需要進行清除，因此它們回傳一個 function：

```
useEffect(() => {
  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
  return () => {
    ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
  };
});
```

其他 effect 可能沒有清除的階段，並且不回傳任何內容。

```
useEffect(() => {
  document.title = `You clicked ${count} times`;
});
```

Effect Hook 通過單個 API 統一了這兩種使用情境。

如果你對 Effect Hook 的執行方式有不錯的理解，或者感到不知所措，你可以立即跳到[下一頁有關 Hook 的規則](#)。

使用 Effect 的提示

我們將在這一頁繼續深入研究 `useEffect` 的某些方面，有經驗的 React 使用者可能會對這些感到好奇。不要覺得現在一定要去研究它們。你可以隨時回來此頁面來了解有關 Effect Hook 的更多詳細資訊。

提示：使用多個 Effect 來分離關注點

對於 Hook，我們在[動機](#)中概述的問題之一是 class 生命週期方法通常包含不相關的邏輯，但是相關的邏輯卻被分成了幾個方法。這是一個結合了前面範例中的計數器和好友狀態指示器邏輯的 component：

```
class FriendStatusWithCounter extends React.Component {
  constructor(props) {
    super(props);
    this.state = { count: 0, isOnline: null };
    this.handleStatusChange = this.handleStatusChange.bind(this);
  }

  componentDidMount() {
    document.title = `You clicked ${this.state.count} times`;
    ChatAPI.subscribeToFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }

  componentDidUpdate() {
    document.title = `You clicked ${this.state.count} times`;
  }

  componentWillUnmount() {
    ChatAPI.unsubscribeFromFriendStatus(
      this.props.friend.id,
      this.handleStatusChange
    );
  }
}
```

```

}
handleStatusChange(status) {
  this.setState({
    isOnline: status.isOnline
  });
}
// ...

```

注意設定 `document.title` 的邏輯是如何在 `componentDidMount` 和 `componentDidUpdate` 之間分配的。subscription 的邏輯也分佈在 `componentDidMount` 和 `componentWillUnmount` 之間。而且 `componentDidMount` 包含了兩個工作的程式碼。

那麼，Hook 可以怎麼解決這個問題？就像[你可以多次使用 State Hook](#)，你同樣可以用多個 effect。這使我們可以將無關的邏輯分為不同的 effect：

```

function FriendStatusWithCounter(props) {
  const [count, setCount] = useState(0);
  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });

  const [isOnline, setIsOnline] = useState(null);
  useEffect(() => {
    function handleStatusChange(status) {
      setIsOnline(status.isOnline);
    }

    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });
}
// ...

```

Hook 讓我們根據程式碼的作用來拆分程式碼，而不是用生命週期方法的名字。React 將按照指定的順序執行 component 所使用的每一個 effect。

解釋：為什麼 Effect 在每次更新時執行

如果你習慣了 class，那麼你可能想知道為什麼 effect 的清除階段會在每次重新 render 後發生，而不僅僅是在 unmounting 過程中發生一次。讓我們看一個實際的範例，看看為什麼這種設計可以幫我們寫出 bug 更少的 component。

在[本頁前面](#)，我們介紹了一個 `FriendStatus` component 的範例，這個 component 顯示朋友是否在線上。我們的 class 從 `this.props` 中抓取 `friend.id`，在 component mount 後訂閱好友狀態，並在 unmount 期間取消訂閱：

```

componentDidMount() {
  ChatAPI.subscribeToFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}

componentWillUnmount() {

```

```
ChatAPI.unsubscribeFromFriendStatus(
  this.props.friend.id,
  this.handleStatusChange
);
}
```

但是如果 **component** 顯示在螢幕上時，**friend prop** 發生變化，會發生什麼呢？我們的 component 將繼續顯示其他好友的線上狀態。這是一個 bug。Unmount 時，由於取消訂閱的呼叫會使用錯誤的朋友 ID，因此也會導致 memory leak 或 crash。

在 class component 中，我們需要加入 `componentDidUpdate` 來處理這種情況：

```
componentDidMount() {
  ChatAPI.subscribeToFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}

componentDidUpdate(prevProps) {
  // 從先前的 friend.id 取消訂閱
  ChatAPI.unsubscribeFromFriendStatus(
    prevProps.friend.id,
    this.handleStatusChange
  );
  // 訂閱下一個 friend.id
  ChatAPI.subscribeToFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}

componentWillUnmount() {
  ChatAPI.unsubscribeFromFriendStatus(
    this.props.friend.id,
    this.handleStatusChange
  );
}
```

忘記正確處理 `componentDidUpdate` 是 React 應用程式中常見的 bug 來源。

現在考慮這個使用 Hook 的 component 版本：

```
function FriendStatus(props) {
  // ...
  useEffect(() => {
    // ...
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });
}
```

它沒有受這個 bug 的困擾。（但我們也沒有對它進行任何更改。）

因為 `useEffect` 會預設處理更新，所以沒有專門用於處理更新的程式碼。在應用下一個 effect 之前，它將清除之前的 effect。為了說明這一點，下面是這個 component 隨時間推移可能產生的一系列訂閱和取消訂閱的呼叫：

```

ChatAPI.subscribeToFriendStatus(100, handleStatusChange); // 執行第一個 effect
ChatAPI.unsubscribeFromFriendStatus(100, handleStatusChange); // 清除前一個 effect
ChatAPI.subscribeToFriendStatus(200, handleStatusChange); // 執行下一個 effect
ChatAPI.unsubscribeFromFriendStatus(200, handleStatusChange); // 清除前一個 effect
ChatAPI.subscribeToFriendStatus(300, handleStatusChange); // 執行下一個 effect
// Unmount
ChatAPI.unsubscribeFromFriendStatus(300, handleStatusChange); // 清除最後一個 effect

```

此行為預設確保程式碼一致性，並防止 class component 中常見的由於缺少更新邏輯而導致的 bug。

提示：通過忽略 Effect 來最佳化效能

在某些情況下，每次 render 後清除或執行 effect 可能會導致效能問題。在 class component 中，我們可以通過在 `componentDidUpdate` 內部的 `prevProps` 或 `prevState` 撰寫一個額外的比對條件來解決此問題：

```

componentDidUpdate(prevProps, prevState) {
  if (prevState.count !== this.state.count) {
    document.title = `You clicked ${this.state.count} times`;
  }
}

```

這個要求很常見，所以已內建在 `useEffect` 的 Hook API 中。如果在重新 render 之間某些值沒有改變，你可以讓 React 忽略 effect。為此，請將 array 作為可選的第二個參數傳遞給 `useEffect`：

```

useEffect(() => {
  document.title = `You clicked ${count} times`;
}, [count]); // 僅在計數更改時才重新執行 effect

```

在上面的範例中，我們將 `[count]` 作為第二個參數傳遞。這是什麼意思？如果 `count` 是 `5`，然後我們的 component 重新 render，`count` 仍然等於 `5`，React 將比對前一個 render 的 `[5]` 和下一個 render 的 `[5]`。因為 array 中的每一項都相同 (`5 === 5`)，所以 React 將忽略這個 effect。那就是我們的最佳化。

當我們 render 時將 `count` 更新為 `6`，React 將比對前一個 render 的 array `[5]` 與下一個 render 的 array `[6]`。這次，React 將重新執行 effect，因為 `5 !== 6`。如果 array 中有多個項目，即使其中一項不同，React 也會重新執行 effect。

這也適用於有清除階段的 effect：

```

useEffect(() => {
  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
  return () => {
    ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
  };
}, [props.friend.id]); // 僅在 props.friend.id 變更時重新訂閱

```

未來，第二個參數可能會透過 build-time transformation 自動被加入。

注意

如果你使用此最佳化，請確保 array 包括了 **component** 範圍內隨時間變化並被 **effect** 用到的所有值（例如 **props** 和 **state**）。否則，你的程式碼將引用先前 render 中的舊值。了解更多[如何處理 function](#) 和[如果 array 經常變化的話該怎麼辦](#)。

如果你想執行一個 effect 並且僅（在 mount 和 unmount 時）將其清除一次，則可以傳遞一個空 array ([]) 作為第二個參數。這告訴 React 你的 effect 不依賴於任何 props 或 state 的值，因此它不需要重新執行。這不屬於特殊情況 — 依賴項目 array 一直這樣工作。

如果你傳遞一個空 array ([])，effect 中的 props 和 state 始終具有其初始值。儘管將 [] 作為第二個參數傳遞更接近於我們熟悉的 `componentDidMount` 和 `componentWillUnmount` 的模式，但通常有[更好的解決方案](#)可以避免過於頻繁地重新執行 effect。另外，別忘了 React 在瀏覽器繪製完成之後才延遲執行 `useEffect`，所以做額外的工作沒有很大的問題。

我們建議使用 `exhaustive-deps` 規則作為我們 `eslint-plugin-react-hooks` package 的一部分。當不正確地指定依賴時，它會發出警告，並提出修改建議。

下一步

恭喜！這一頁很長，但是希望讀到最後，你絕大多數的問題都有了答案。你已經學過 State Hook 和 Effect Hook，把兩者結合起來，你已經能做到很多東西。它們涵蓋了 class 的絕大多數的使用案例 — 如果沒有涵蓋到，[額外的 Hook](#) 或許會幫到你。

我們也開始看到 Hook 如何解決[動機](#)中概述的問題。我們已經看到了 effect 清除如何避免在 `componentDidUpdate` 和 `componentWillUnmount` 中重複，如何使相關程式碼更緊密地結合在一起，並幫助我們避免 bug。我們還看到了我們可以如何根據 effect 的目的來區分 effect，這是我們在 class 中根本無法做到的。

現在，你可能會質疑 Hook 的工作方式。React 怎麼知道哪個 `useState` 呼叫對應於 re-render 之間的哪個 state 變數？React 如何在每次更新中「匹配」上一個和下一個 effect？[在下一頁，我們會學習 Hook 的規則 — 它們對於 Hook 的正常執行至關重要](#)。

Hooks 常見問題

Hooks 是 React 16.8 中增加的新功能。它讓你不必寫 class 就能使用 state 以及其他 React 的功能。

這一頁會解答一些關於 [Hook](#) 常見的問題。

- [採用策略](#)

- React 哪一個版本中包含 Hook ?
- 我需要重寫所有的 Class component 嗎 ?
- 我可以在 Hook 做什麼是我在 Class 所不能做的 ?
- Hook 與我的 React 知識有多少保持相關性 ?
- 我應該使用 Hook 、 Class 或是兩者兼具 ?
- Hook 包含所有 Class 的使用情境嗎 ?
- Hook 可以取代 Render Props 和 Higher-Order Component 嗎 ?
- Hook 對於 Redux connect() 和 React Router 等等其他流行的 API 意味著什麼 ?
- Hook 可以使用靜態型別嗎 ?
- 如何測試使用 Hook 的 component ?
- Lint 規則究竟強制了些什麼 ?

- [從 Class 到 Hook](#)

- 生命週期方法與 Hook 如何對應 ?
- 我如何使用 Hook fetch 資料 ?
- 是否有類似 Instance 變數的東西 ?
- 我應該使用一個或是多個 state 變數 ?
- 我可以只在更新時執行 effect 嗎 ?
- 如何取得先前的 prop 或 state ?
- 為什麼我在 function 內看到舊的 prop 或 state ?
- 我該如何實作 getDerivedStateFromProps ?
- 有類似 forceUpdate 的東西嗎 ?
- 我可以對 function component 建立一個 ref 嗎 ?
- 我該如何測量一個 DOM node ?
- const [thing, setThing] = useState() 是什麼意思 ?

- [效能最佳化](#)

- 我可以在更新時忽略 effect 嗎 ?
- 在依賴項目的列表中忽略 function 是安全的嗎 ?
- 如果我的 effect 依賴項目經常變化的話該怎麼辦 ?
- 我該如何實作 shouldComponentUpdate ?
- 如何 memoize 計算 ?
- 如何延遲建立昂貴的 object ?
- 在 render 時建立 function, Hooks 會變慢嗎 ?
- 如何避免向下傳遞 callback ?
- 如何從 useCallback 讀取一個經常變化的值 ?

- [深入理解](#)

- React 如何將 Hook 呼叫與 component 關聯 ?

- Hook 現有的技術是什麼？

採用策略

React 哪一個版本中包含 Hook？

從 16.8.0 開始，React 包含一個穩定的 React Hooks 實作：

- React DOM
- React Native
- React DOM Server
- React Test Renderer
- React Shallow Renderer

注意，若要啟動 Hook，所有 React package 需要升級到 **16.8.0** 或是更高的版本。例如你忘了升級 React DOM，Hook 將無法正常執行。

React Native 0.59 以上的版本支援 Hooks。

我需要重寫所有的 Class component 嗎？

不需要。React [沒有計劃](#) 移除 class -- 我們需要讓產品保持運作，重寫的成本很高，我們建議你在新的程式碼中嘗試 Hook。

我可以在 Hook 做什麼是我在 Class 所不能做的？

Hook 在 component 之間提供強大而富有表現力的新方式來重複使用功能。在「[打造你的 Hook](#)」提供了一個可行的方式。[這篇文章](#)是由 React 核心團隊成員深入研究 Hook 解鎖的新功能。

Hook 與我的 React 知識有多少保持相關性？

Hook 已經有許多你知道的方式可以直接使用 React 的功能 -- 像是 state、lifecycle、context 以及 ref。基本上它們並沒有改變 React 的運作方式，而且你對於 component、prop 以及上至下的資料流了解也一樣重要。

Hook 的確有它本身的學習曲線。如果在這份文件中缺漏了些什麼，[提出一個 issue](#)，我們將會嘗試幫助你。

我應該使用 Hook、Class 或是兩者兼具？

當你準備好時，我們鼓勵開始使用 Hook 撰寫你新的 component。確保你團隊的成員們使用 Hook 並熟悉本文件。我們並不鼓勵你重寫現有的 class component 成 Hook，除非你已經計劃重寫它們（例如：修正 bug）。

你不可以在 class component 內使用 Hook，但你絕對可以在單個 tree 中將 class 和 function component 與 Hook 混合使用。無論是 class 或 function component，使用 Hook 是該 component 實作的細節。從長遠來看，我們期待 Hook 可以是大家撰寫 React component 的主要方式。

Hook 包含所有 Class 的使用情境嗎？

我們的目標是讓 Hook 盡快能涵蓋 class 的所有使用情境。對於不常見的 `getSnapshotBeforeUpdate` 和 `componentDidCatch` 的生命週期並沒有等價的 Hook 方式，但我們計劃很快會加入它們。

Hook 可以取代 Render Props 和 Higher-Order Component 嗎？

我們思考 Hook 是一個更簡單的方式來提供這個使用情境。這兩種模式仍然有它的用處（例如：一個 virtual scroller component 可能有一個 `renderItem` prop，或是一個 virtual container component 可能有它本身的 DOM 結構）。但在大部分的情況下，Hook 就可以滿足了，而且可以幫助你減少在 tree 內的巢狀 component。

Hook 對於 Redux `connect()` 和 React Router 等等其他流行的 API 意味著什麼？

你可以繼續使用與以往完全相同的 API；它們依然可以正常使用。

React Redux 從 v7.1.0 [支援 Hooks API](#) 並提供像是 `useDispatch` 或 `useSelector` 的 hooks。

React Router 從 v5.1 版本後[支援 Hooks](#)。

其他像是 React Router 可能在未來會支援 Hooks。

Hook 可以使用靜態型別嗎？

Hook 的設計考慮到了靜態型別。因為它們是 function，比 Higher-Order Component 等其他模式的 component 更容易正確的定義。最新的 Flow 和 TypeScript 定義包含對 React Hook 的支援。

重要的是，如果你想要以某種嚴格的方式定義 React API，自訂的 Hook 讓你有權利限制。React 為你提供了 primitive，但你可以將它與我們提供的方式，用不同的方式組合在一起。

如何測試使用 Hook 的 component？

從 React 的角度來看，使用 Hook 的 component 就只是一個正常的 component。如果你的測試方案不依賴在 React 內部，測試使用 Hook 的 component 通常與你測試 component 的方式相同。

注意

[測試方法](#)包含許多範例，讓你可以複製貼上。

例如，如果我們有一個計數器 component：

```
function Example() {
  const [count, setCount] = useState(0);
  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });
  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

我們將會使用 React DOM 測試它。為了確保它在瀏覽器發生的行為，我們將會把程式碼透過呼叫 `ReactTestUtils.act()` 來 render 和更新：

```
import React from 'react';
import ReactDOM from 'react-dom/client';
import { act } from 'react-dom/test-utils';
import Counter from './Counter';

let container;

beforeEach(() => {
  container = document.createElement('div');
  document.body.appendChild(container);
});

afterEach(() => {
  document.body.removeChild(container);
  container = null;
});

it('can render and update a counter', () => {
  // 第一次測試 render 和 effect
  act(() => {
    ReactDOM.createRoot(container).render(<Counter />);
  });
  const button = container.querySelector('button');
  const label = container.querySelector('p');
  expect(label.textContent).toBe('You clicked 0 times');
  expect(document.title).toBe('You clicked 0 times');

  // 第二次測試 render 和 effect
  act(() => {
    button.dispatchEvent(new MouseEvent('click', {bubbles: true}));
  });
  expect(label.textContent).toBe('You clicked 1 times');
  expect(document.title).toBe('You clicked 1 times');
});
```

對 `act()` 的呼叫也會更新它內部的 effect。

如果你需要測試自定義的 Hook，你可以透過在測試中建立你的 component，並從中使用 Hook。接著你可以測試你撰寫的 component。

為了減少 boilerplate，我們推薦使用 `react-testing-library`，它的設計理念是鼓勵你撰寫像使用者在使用 component 的測試。

更多資訊，請參考[測試方法](#)。

Lint 規則究竟強制了些什麼？

我們提供一個 [ESLint plugin](#) 強制 Hook 的規則 來避免 bug。它假設任何以「`use`」為開頭的 function 和緊跟在它之後的大寫字母是 Hook。我們認知到這個啟發式的搜尋不是完美而且可能有一些誤判，但是如果沒有一個全生態系統的慣例，就沒有辦法讓 Hook 良好的運作 -- 而且，較長的命名會阻礙人們採用 Hook 或是遵循慣例。

特別是，該規範強制執行：

- 呼叫 Hook 要麼是在一個 `PascalCase` function (假設是一個 component) 內，或者是其他 `useSomething` function (假設是一個字定義的 Hook)。
- 在每次的 `render` 上以相同的順序呼叫 Hook。

這裡還有一些啟發式的方法，當我們發現錯誤並微調規則以平衡避免誤判時，這些規則可能會隨著時間而改變。

從 Class 到 Hook

生命週期方法與 Hook 如何對應？

- `constructor`：Function component 不需要 `constructor`。你可以在呼叫 `useState` 時初始化 state。如果初始化 state 的操作代價很高，你可以傳遞一個 function 到 `useState`。
- `getDerivedStateFromProps`：改為在 `render` 時安排更新。
- `shouldComponentUpdate`：參考 `React.memo` 如下。
- `render`：這是 function component body 本身。
- `componentDidMount`、`componentDidUpdate`、`componentWillUnmount`：`useEffect` Hook 可以表達這些所有的組合（包含少見和常見）的情況
- `getSnapshotBeforeUpdate`、`componentDidCatch` 和 `getDerivedStateFromError`：現在沒有 Hook 等價於這些方法，但是它們未來很快會被加入。

我如何使用 Hook fetch 資料？

這裡有一個範例讓你可以開始。想要學習更多，閱讀這篇關於使用 Hook fetch 資料的文章。

是否有類似 Instance 變數的東西？

是的！`useRef()` Hook 不只是針對 DOM 的 ref。「ref」object 是一個 generic container，其 `current` 屬性是可變的，可以保存任何值，類似於 class 上的 `instance` 屬性。

你可以從 `useEffect` 內撰寫它：

```
function Timer() {
  const intervalRef = useRef();

  useEffect(() => {
    const id = setInterval(() => {
      // ...
    });
    intervalRef.current = id;
    return () => {
      clearInterval(intervalRef.current);
    };
  });
  // ...
}
```

如果我們想要設定一個計時器，我們不需要 ref (`id` 可能是在 effect)，但是如果我們想要從一個 event handler 清除定時器它會很有幫助：

```
// ...
function handleCancelClick() {
  clearInterval(intervalRef.current);
}
// ...
```

概念上，你可以將 ref 視為類似於 class 中 instance 的變數。除非你正在做[延遲初始化](#)，避免在 render 時設定 ref -- 這可能會造成非預期的行為。相反的，通常你會在 event handler 和 effect 中修改 ref。

我應該使用一個或是多個 state 變數？

如果你原來是使用 class component，你可能會想要呼叫 `useState()` 並一次放入所有的 state 到一個 object，你想要的話可以這麼做。這裡是一個隨著滑鼠移動的 component 範例。我們保持它的位置和大小在 local state 內：

```
function Box() {
  const [state, setState] = useState({ left: 0, top: 0, width: 100, height: 100 });
}
// ...
```

現在我們想要撰寫一些邏輯，當使用者移動滑鼠時，改變 `left` 和 `top`。注意我們如何手動合併這些欄位至先前的 state object：

```
// ...
useEffect(() => {
  function handleWindowMouseMove(e) {
    // 使用展開運算子「...state」確保我們不會「漏掉」width 和 height
    setState(state => ({ ...state, left: e.pageX, top: e.pageY }));
  }
  // 注意：這個實作是相當簡化的
  window.addEventListener('mousemove', handleWindowMouseMove);
  return () => window.removeEventListener('mousemove', handleWindowMouseMove);
}, []);
// ...
```

這是因為當我們更新 state 變數時，我們替換它的值。這與 class 中的 `this.setState` 不同，將被更新的欄位合併到 object 中。

如果你漏掉了自動合併，你可以撰寫一個自訂的 `useLegacyState` Hook 來合併 object state 的更新。然而，**我們建議根據哪些值是趨於一起變化的，拆分為多個 state 變數**

例如，我們可以拆分 component state 成 `position` 和 `size` object，並總是替換 `position` 而不需要合併：

```
function Box() {
  const [position, setPosition] = useState({ left: 0, top: 0 });
  const [size, setSize] = useState({ width: 100, height: 100 });

  useEffect(() => {
    function handleWindowMouseMove(e) {
      setPosition({ left: e.pageX, top: e.pageY });
    }
  }, []);
```

分離獨立的 state 變數也有另一個好處。稍後可以輕鬆地將一些相關邏輯提取到自定義的 Hook 中，例如：

```
function Box() {
  const position = useWindowPosition();
  const [size, setSize] = useState({ width: 100, height: 100 });
  // ...
}

function useWindowPosition() {
  const [position, setPosition] = useState({ left: 0, top: 0 });
  useEffect(() => {
    // ...
  }, []);
  return position;
}
```

注意我們是如何移動 `position` state 變數的 `useState` 呼叫，並且在不改變我們程式碼的情況下，將相關 effect 寫到自定義的 Hook。如果所有的 state 在一個單一的 object，提取它會變得困難。

以上兩者都將所有 state 放到各自的 `useState`，並可以被呼叫。當你在這兩個極端之間找到平衡時，component 往往最具可讀性，並且將相關 state 變成一些獨立的 state 變數。如果 state 邏輯變得複雜，我們推薦用 [reducer 管理](#)或者是一個自定義的 Hook。

我可以只在更新時執行 effect 嗎？

這是一個特殊的情況，如果你有需要的話，可以[使用 mutable ref](#) 來手動的儲存對應於第一次或是後續的 render，然後檢查在你 effect 內的 flag。（如果你發現自己經常這樣做，可以為其建立一個自定義的 Hook。）

如何取得先前的 prop 或 state？

有兩種情況下，你可能想取得先前的 prop 或 state。

有時候，你需要先前的 prop 來 [清除一個 effect](#)。例如，你可能有一個基於 `userId` prop 來 subscribe 一個 socket 的 effect，你想要 unsubscribe 先前的 `userId` 以及 subscribe 下一個。對於這樣的工作你不需要做任何事：

```
useEffect(() => {
  ChatAPI.subscribeToSocket(props.userId);
  return () => ChatAPI.unsubscribeFromSocket(props.userId);
}, [props.userId]);
```

在上面的範例中，如果 `userId` 從 `3` 變成 `4`，`ChatAPI.unsubscribeFromSocket(3)` 會回傳第一個，接著執行 `ChatAPI.subscribeToSocket(4)`。這裡不需要取得「先前的」 `userId`，因為 cleanup function 會在它的 closure 捕獲它。

其他時候，你可能需要基於 prop 或 state 的變化來調整 state。這很少需要，而且這通常是個訊號你有一些重複或多餘的 state。然而，在極少數的情況下你需要這個模式，你可以 [儲存先前的 state 或 prop 在 state 並且在 rendering 期間更新它們](#)。

我們之前提出了一個自定義的 hook 叫做 `usePrevious` 來儲存前一個值。然而，我們發現大部分的使用案例都是向上面所描述的兩種模式。如果你的使用情境非常不同，你可以 [儲存一個值在 ref](#) 並且當它需要的時候手動更新它。避免在 rendering 中讀取和更新 ref，因為這讓你 component 的行為難以預測和理解。

為什麼我在 function 內看到舊的 prop 或 state？

任何在 component 內的 function，包括 event handler 和 effect，從被建立的 render 中「看見」props 和 state。例如，思考一下這個程式碼：

```
function Example() {
  const [count, setCount] = useState(0);

  function handleAlertClick() {
    setTimeout(() => {
      alert('You clicked on: ' + count);
    }, 3000);
  }

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
      <button onClick={handleAlertClick}>
        Show alert
      </button>
    </div>
  );
}
```

如果你第一次點擊「Show alert」並且遞增 counter，那麼 alert 將會顯示你當時點擊「Show alert」按鈕時的 count 變數。假設 prop 和 state 沒有改變的話，這可以避免 bug。

如果你想要從一些非同步的 callback 讀取最新的 state，你可以把它保留在 ref，mutate 它，並從中讀取。

最後，如果你使用「dependency array」做最佳化，但沒有正確的指定所有的依賴，你可能會看到舊的 props 或是 state。例如，如果在 effect 指定 [] 作為第二個參數，但是在內部讀取 someProp，它將只會「看到」 someProp 的初始值。解決方式是移除 dependency array 或者是修正它。這裡是[你如何處理這些 function](#)，以及其他常見的策略在不錯誤地跳過依賴項的情況下減少執行 effect。

注意

我們提供了一個 exhaustive-deps 的 ESLint 規則作為 eslint-plugin-react-hooks package 的一部份。它會在指定錯誤的依賴時，發出警告並提供建議修復。

我該如何實作 getDerivedStateFromProps？

雖然你可能不需要它，但在極少數情況下（例如實作 <Transition> component），你可以在 render 期間更新 state。React 將會在第一次的 render 退出後，重新執行 component 並且立即更新 state，這個操作代價不會很昂貴。

在這裡，我們將 row prop 先前的值存儲在 state 變數中，讓我們可以方便的比較：

```
function ScrollView({row}) {
  const [isScrollingDown, setIsScrollingDown] = useState(false);
  const [prevRow, setPrevRow] = useState(null);
```

```

if (row !== prevRow) {
  // Row 在最後一次 render 被改變。更新 isScrollingDown
  setIsScrollingDown(prevRow === null && row > prevRow);
  setPrevRow(row);
}

return `Scrolling down: ${isScrollingDown}`;
}

```

這個第一眼看起來可能很奇怪，但是 render 過程中的更新正是 `getDerivedStateFromProps` 的概念。

有類似 `forceUpdate` 的東西嗎？

如果新的值與先前相同的話，`useState` 和 `useReducer` Hook 兩者都可以從 [更新中跳脫](#)。

通常來說，你不應該在 React 內 mutate local state，然而，你可以使用一個 increment counter 來作為跳脫方式，強迫重新 render，即使 state 沒有改變。

```

const [ignored, forceUpdate] = useReducer(x => x + 1, 0);

function handleClick() {
  forceUpdate();
}

```

如果可以的話，盡量避免使用這個方式。

我可以對 function component 建立一個 ref 嗎？

雖然你不應該經常這樣做，但是你可以使用 `useImperativeHandle` Hook expose 一些 imperative 方法給 parent component。

我該如何測量一個 DOM node？

為了測量 DOM node 的位置或是大小，你可以使用 `callback ref`。只要 ref 被 attach 到不同的 node，React 將會呼叫這個 callback。這裡是一個 [簡易的範例](#)：

```

function MeasureExample() {
  const [height, setHeight] = useState(0);

  const measuredRef = useCallback(node => {
    if (node !== null) {
      setHeight(node.getBoundingClientRect().height);
    }
  }, []);

  return (
    <>
      <h1 ref={measuredRef}>Hello, world</h1>
      <h2>The above header is {Math.round(height)}px tall</h2>
    </>
  );
}

```

在這個範例我們不選擇 `useRef` 是因為一個 object 的 ref 不會通知我們目前的 ref 值的改變。使用一個 callback ref 確保即使 child component 延遲顯示測量的 node (例如：在 response click)，我們仍然會在 parent component 中收到有關它的通知，並可以更新測量結果。

這確保我們的 ref callback 不會在 re-render 時改變，因此 React 不需要呼叫它。

在這個範例，callback ref 只會在當 component mount 以及 unmount 時被呼叫，由於被 render 的 `<h1>` component 在所有 render 都保持存在。如果你想要在任何時候在 component resize 時被通知，你可以使用 `ResizeObserver` 或其他第三方的 Hook。

如果你希望呼叫它的話，你可以抽出這個邏輯變成一個可重複使用的 Hook：

```
function MeasureExample() {
  const [rect, ref] = useClientRect();
  return (
    <>
      <h1 ref={ref}>Hello, world</h1>
      {rect !== null &&
        <h2>The above header is {Math.round(rect.height)}px tall</h2>
      }
    </>
  );
}

function useClientRect() {
  const [rect, setRect] = useState(null);
  const ref = useCallback(node => {
    if (node !== null) {
      setRect(node.getBoundingClientRect());
    }
  }, []);
  return [rect, ref];
}
```

`const [thing, setThing] = useState()` 是什麼意思？

如果你不熟悉這個語法的話，可以參考在 State Hook 文件的解釋。

效能最佳化

我可以在更新時忽略 effect 嗎？

可以的。請參考條件式觸發 effect。請注意，忘記處理更新通常導致 bug，這就是為什麼這不是預設行為。

在依賴項目的列表中忽略 function 是安全的嗎？

一般來說，不是。

```
function Example({ someProp }) {
  function doSomething() {
    console.log(someProp);
  }

  useEffect(() => {
```

```
    doSomething();
}, []); // 🚫 這個不安全 (呼叫 `doSomething` 的 function 使用了 `someProp` )
```

要記住 effect 之外的 function 使用了哪些 props 或 state 是很困難的。這也是為什麼通常你需要在 **effect** 內宣告所需要的 **function**。然後可以很容易的看出 effect 依賴了 component 範圍內的值：

```
function Example({ someProp }) {
useEffect(() => {
  function doSomething() {
    console.log(someProp);
  }

  doSomething();
}, [someProp]); // ✅ OK (我們的 effect 只使用 `someProp`)
}
```

如果之後我們仍然不使用 component 範圍內的任何值，則可以安全的指定為 `[]`：

```
useEffect(() => {
  function doSomething() {
    console.log('hello');
  }

  doSomething();
}, []); // ✅ OK，在這個範例中，因為我們不使用 component 範圍中的*任何*值
```

根據你的使用情境，還有一些選項如下所述。

注意

我們提供了 `exhaustive-deps` ESLint 規則作為 `eslint-plugin-react-hooks` package 的一部份。它可以幫助你找到不一致的處理更新的 component。

讓我們看看這個為什麼很重要。

如果你在 `useEffect`、`useLayoutEffect`、`useMemo`、`useCallback` 或是 `useImperativeHandle` 的最後指定了[依賴項目的列表](#)，它必須包含在 callback 內使用的值以及參與 React 的資料流。包含了 props、state 和從它們取得的任何值。

如果沒有任何內容（或由它呼叫的 function）reference 到 props、state 或是從它們取得的值，那麼從依賴項目中省略一個 function 是唯一安全的。這個範例有一個 bug：

```
function ProductPage({ productId }) {
  const [product, setProduct] = useState(null);

  async function fetchProduct() {
    const response = await fetch(`http://myapi/product/${productId}`); // 使用
    productId prop
    const json = await response.json();
    setProduct(json);
  }
}
```

```
useEffect(() => {
  fetchProduct();
}, []); // 🚫 無效，因為 `fetchProduct` 使用 `productId`  
// ...
}
```

推薦修正的方法是將 function 移動到你的 effect 內部。這樣可以很容易地看到你的 effect 使用了哪些 props 或 state，並確保他們都被宣告：

```
function ProductPage({ productId }) {
  const [product, setProduct] = useState(null);

  useEffect(() => {
    // 透過將這個 function 移動到 effect 內部，我們可以清楚地看到值的使用。
    async function fetchProduct() {
      const response = await fetch('http://myapi/product/' + productId);
      const json = await response.json();
      setProduct(json);
    }

    fetchProduct();
  }, [productId]); // ✅ 有效，因為我們的 effect 只使用 productId
  // ...
}
```

這也可以允許你處理在 effect 內使用 local 變數處理無序的 response：

```
useEffect(() => {
  let ignore = false;
  async function fetchProduct() {
    const response = await fetch('http://myapi/product/' + productId);
    const json = await response.json();
    if (!ignore) setProduct(json);
  }

  fetchProduct();
  return () => { ignore = true };
}, [productId]);
```

我們搬移 function 到 effect 內部，所以它不需要被加入在依賴列表。

Tip

查看[這個簡易的範例](#)以及[這篇文章](#)來學習關於如何使用 Hooks 來取得資料。

如果有一些因素讓你不能搬移 function 到 effect 內，這裡有一些其他的選項：

- 你可以嘗試將 function 搬移到 component 之外。在這個情況下，function 可以保證不 reference 到任何的 props 或 state，而且也不需要在依賴項目的列表中。
- 如果你正在呼叫的 function 是 pure 的計算，而且可以在 render 時被安全的呼叫，**你可以在 effect 外呼叫它**，並讓 effect 取決於回傳的值。
- 作為最後的手段，你可以加入一個 function 到 effect 依賴項目，但是封裝它的定義成 `useCallback` Hook。這可以確保它不會在每次 render 時改變，除非它自己的依賴項目也改變：

```

function ProductPage({ productId }) {
  // ✅ 藉由 useCallback 封裝可以避免在每次 render 時改變
  const fetchProduct = useCallback(() => {
    // ... Does something with productId ...
  }, [productId]); // ✅ 所有 useCallback 依賴已經被指定

  return <ProductDetails fetchProduct={fetchProduct} />;
}

function ProductDetails({ fetchProduct })
  useEffect(() => {
    fetchProduct();
  }, [fetchProduct]); // ✅ 所有 useEffect 依賴已經被指定
  // ...
}

```

注意，在上面的範例中，我們需要保持 function 在依賴列表內。這可以確保 `ProductPage` 中的 `productId` 變更自動會觸發 `ProductDetails` component 的 refetch。

如果我的 effect 依賴項目經常變化的話該怎麼辦？

有時候，你的 effect 可能使用 state 而且它經常變更。你可能想從依賴的項目列表中省略該 state，但這通常會導致 bug：

```

function Counter() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    const id = setInterval(() => {
      setCount(count + 1); // 這個 effect 依賴於 `count` state
    }, 1000);
    return () => clearInterval(id);
  }, []); // 🚨 Bug : `count` 沒有被指定作為一個依賴

  return <h1>{count}</h1>;
}

```

設定空的依賴 `[]`，意味著 effect 將只會在 component mount 的時候只執行一次，而不是在每次 re-render。問題在於內部的 `setInterval` callback，`count` 值並不會變更，因為我們已經建立一個 closure 並且設定 `count` 為 `0`，就像執行 effect callback 時一樣。在每秒鐘，這個 callback 呼叫 `setCount(0 + 1)`，所以 `count` 永遠不會大於 `1`。

指定 `[count]` 作為依賴項目可以修正這個 bug，但會導致每次更改時重置間隔。實際上，每個 `setInterval` 在被清除之前都有一次機會執行（類似於 `setTimeout`。）這可能並不理想。要修正這個問題，我們可以使用 `useState` 的 functional 更新的形式。它允許我們指定如何 state 需要改變而不引用目前的 state：

```

function Counter() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    const id = setInterval(() => {
      setCount(c => c + 1); // ✅ 這不會依賴於外部的 `count` 變數
    }, 1000);
    return () => clearInterval(id);
  }, []); // ✅ 我們的 effect 不使用 component 範圍內的任何變數

  return <h1>{count}</h1>;
}

```

(`setCount` function 的 identity 的保證是穩定的，因此省略它是安全的。)

現在，`setInterval` callback 每秒執行一次，但每次對 `setCount` 內部的呼叫都可以使用 `count` 最新的值（在 callback 中被呼叫的 `c`。）

在許多複雜的情況下（例如，一個 state 依賴另一個 state），嘗試使用 `useReducer` Hook 將 state 的更新邏輯搬移到 effect 外。這篇文章提供了一個如何做到的範例。`useReducer` 中的 `dispatch` function 的 identity 是穩定的 — 即使 reducer function 被宣告在 component 內並讀取它的 props。

作為最後的手段，如果你需要像是 class 內的 `this`，你可以使用 `ref` 來持有一個 mutable 變數。然後你可以寫入和讀取它。例如：

```
function Example(props) {
  // 保持最新的 props 在 ref。
  const latestProps = useRef(props);
  useEffect(() => {
    latestProps.current = props;
  });

  useEffect(() => {
    function tick() {
      // 在任何時候讀取最新的 props
      console.log(latestProps.current);
    }

    const id = setInterval(tick, 1000);
    return () => clearInterval(id);
  }, []); // 這個 effect 永遠不會重新執行
}
```

如果你真的找不到其他更好的方式才這麼做，因為依賴 mutation 會讓 component 的可預測性降低。如果有一個特定的模式不能很好地轉換，請提出 issue 並附上可執行的範例程式碼，我們可以嘗試提供幫助。

我該如何實作 `shouldComponentUpdate`？

你可以藉由 `React.memo` 封裝 function 來對它的 props 進行淺比較：

```
const Button = React.memo((props) => {
  // 你的 component
});
```

它不是一個 Hook，因為它並不像 Hooks 那樣的組成。`React.memo` 相等於 `PureComponent`，但是它只比較 props。（你也可以加入第二參數，指定一個舊 props 和新 props 的比較 function。如果它回傳 `true`，則跳過更新。）

`React.memo` 不比較 state，因為沒有單一的 state object 可以比較。但你也可以讓 children 變成 pure，甚至可以透過 `useMemo` 來最佳化個別的 children。

如何 memoize 計算？

`useMemo` Hook 讓你可以透過「記住」先前的計算來快取多個 render 之間的計算：

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

這個程式碼呼叫 `computeExpensiveValue(a, b)`。但是如果 `[a, b]` 依賴自上一個值沒有改變，那麼 `useMemo` 會跳過第二次的呼叫並只重複使用它回傳的最後一個值。

記住，被傳到 `useMemo` 的 function 會在 render 期間執行。不要在 render 期間做一些通常不會做的事情。例如，side effects 屬於在 `useEffect` 被處理，而不是 `useMemo`。

你可以依賴 `useMemo` 作為效能的最佳化，而不是依賴語意的保證。 未來 React 可能會選擇「忘記」一些先前 memoize 的值，並在下一次 render 重新計算，例如，釋放螢幕以外的 component 記憶體。撰寫你的程式碼，讓它在沒有 `useMemo` 的狀況下依然可以執行 - 並且加上它來最佳化效能。(對於極少數的情況，值永遠不會被重新計算，你可以[延遲初始化 ref](#)。)

方便的是，`useMemo` 也可以讓你跳過一個 child 昂貴的 re-render：

```
function Parent({ a, b }) {
  // 只會在 `a` 變更時 re-render :
  const child1 = useMemo(() => <Child1 a={a} />, [a]);
  // 只會在 `b` 變更時 re-render :
  const child2 = useMemo(() => <Child2 b={b} />, [b]);
  return (
    <>
      {child1}
      {child2}
    </>
  )
}
```

注意，這個方法在一個迴圈中無法執行，因為 Hook [不能](#)被放在迴圈內。但是你可以提取一個清單列表的獨立 component，並在這裡呼叫 `useMemo`。

如何延遲建立昂貴的 object?

如果依賴相同的話，`useMemo` 讓你可以 [memoize 一個昂貴的計算](#)。然而，它只是個提示，並不能保證計算不會重新執行。但有時候你需要確保一個 object 只被建立一次。

第一個常見的情況是建立昂貴的初始 state：

```
function Table(props) {
  // ⚠️ createRows() 在每次 render 被呼叫
  const [rows, setRows] = useState(createRows(props.count));
  // ...
}
```

為了避免重新建立初始 state，我們可以傳遞一個 **function** 給 `useState`：

```
function Table(props) {
  // ✅ createRows() 只會被呼叫一次
  const [rows, setRows] = useState(() => createRows(props.count));
  // ...
}
```

React 只會在第一次 render 時呼叫這個 function。參考 [useState API](#)。

你偶爾可能也想要避免重新建立 `useRef()` 初始值。例如，或許你想要確保某些 imperative class 只被建立一次：

```
function Image(props) {
  // ⚠️ IntersectionObserver 在每次 render 時被建立
  const ref = useRef(new IntersectionObserver(onIntersect));
  // ...
}
```

`useRef` 不接受像 `useState` 這樣特殊的重載 function。你可以撰寫你自己的 function 來建立並延遲設定：

```
function Image(props) {
  const ref = useRef(null);

  // ✅ IntersectionObserver 會延遲被建立一次
  function getObserver() {
    if (ref.current === null) {
      ref.current = new IntersectionObserver(onIntersect);
    }
    return ref.current;
  }

  // 當你需要它的時候，呼叫 getObserver()
  // ...
}
```

這是避免了在第一次真正需要之前建立昂貴的 object。如果你使用 Flow 或者是 TypeScript，為了方便你也可以給定 `getObserver()` 一個 non-nullable 的型別。

在 render 時建立 function, Hooks 會變慢嗎？

不會，在現代瀏覽器中，除了在極端情況下，closure 的原生效能與 class 相較之下沒有明顯的差異。

此外，考慮到 Hooks 的設計在以下幾個方面很有效：

- Hook 可以避免 class 的大量開銷，像是建立 class instance 並在 constructor 繩定 event handler。
- 習慣使用 Hooks 的程式碼後，就不需要深層的巢狀 component，這在 higher-order component、render props 和 context 等其他流行函式庫中普遍存在，使用較小的 component tree，React 可以減少更多的工作量。

傳統上，在 React inline function 的效能問題與如何在 child component 中，在每個 render 打破 `shouldComponentUpdate` 最佳化傳遞新的 callback 有關。Hooks 從三個方面來處理這個問題。

- The `useCallback` Hook 讓你可以在重新 render 之間保持相同的 callback，所以 `shouldComponentUpdate` 依然可以運作：

```
// 除非 `a` 或 `b` 變更，否則不改變
const memoizedCallback = useCallback(() => {
  doSomething(a, b);
}, [a, b]);
```

- `useMemo` Hook 讓獨立的 children 更新時，可以更容易的被控制，減少對 pure component 的需求。
- 最後，`useReducer` Hook 減少傳遞深層的 callback 需要，我們將會在下面解釋。

如何避免向下傳遞 callback？

我們發現大部分的人不喜歡透過 component tree 的每一層手動傳遞 callback。即使它是更明確，它可以感覺像是有許多「管道」。

在大型的 component tree 中，我們推薦另一個方法是透過 context 從 `useReducer` 傳遞一個 `dispatch` function：

```
const TodosDispatch = React.createContext(null);

function TodosApp() {
  // 注意：`dispatch` 在 re-render 之間不會改變
  const [todos, dispatch] = useReducer(todosReducer);

  return (
    <TodosDispatch.Provider value={dispatch}>
      <DeepTree todos={todos} />
    </TodosDispatch.Provider>
  );
}
```

任何在 `TodosApps` 內的 child component 可以使用 `dispatch` function 來傳遞 action 到 `TodosApp`：

```
function DeepChild(props) {
  // 如果我們想要執行一個 action，我們可以從 context 取得 dispatch。
  const dispatch = useContext(TodosDispatch);

  function handleClick() {
    dispatch({ type: 'add', text: 'hello' });
  }

  return (
    <button onClick={handleClick}>Add todo</button>
  );
}
```

從維護的角度來看更方便（不需要持有轉發 callback），並且完全避免了 callback 問題。像這樣傳遞 `dispatch` 是深度更新的推薦模式。

請注意，你仍然可以選擇是否將應用程式的 `state` 向下傳遞為 props（更明確）還是作為 context（對於非常深的更新更方便）。如果你也使用 context 傳遞 state，使用兩個不同的 context type -- `dispatch` context 永遠不會改變，因此讀取它的 component 不需要重新 render，除非它們也需要應用程式的 state。

如何從 `useCallback` 讀取一個經常變化的值？

注意

我們建議在 context 中傳遞 `dispatch`，而不是在 props 中傳遞單獨的 callback。下面的方法僅在此處提及完整性和跳脫方法。

在極少數的情況下你可能會透過 `useCallback` memoize 一個 callback，但是因為內部 function 必須常常被重新建立，所以 memoize 沒有辦法很好個運作。如果你要 memoize 的 function 是一個 event handler，而且它不會被在 render 時被使用，你可以使用 `ref` 作為一個 instance 變數，並手動儲存最後被 commit 的值：

```
function Form() {
  const [text, updateText] = useState('');
  const textRef = useRef();

  useEffect(() => {
    textRef.current = text; // 將它寫到 ref
  });

  const handleSubmit = useCallback(() => {
    const currentText = textRef.current; // 從 ref 讀取它
    alert(currentText);
  }, [textRef]); // 不要像 [text] 那樣重新建立 handleSubmit

  return (
    <>
      <input value={text} onChange={e => updateText(e.target.value)} />
      <ExpensiveTree onSubmit={handleSubmit} />
    </>
  );
}
```

這是一個相當複雜的模式，但如果你需要，可以執行這個跳脫最佳化。如果你提取它到自訂的 Hook：

```
function Form() {
  const [text, updateText] = useState('');
  // 如果 `text` 改變，將會被 memoize：
  const handleSubmit = useEventCallback(() => {
    alert(text);
  }, [text]);

  return (
    <>
      <input value={text} onChange={e => updateText(e.target.value)} />
      <ExpensiveTree onSubmit={handleSubmit} />
    </>
  );
}

function useEventCallback(fn, dependencies) {
  const ref = useRef(() => {
    throw new Error('Cannot call an event handler while rendering.');
  });

  useEffect(() => {
    ref.current = fn;
  }, [fn, ...dependencies]);

  return useCallback(() => {
    const fn = ref.current;
    return fn();
  }, [ref]);
}
```

在任何一種情況下，我們都不推薦這種模式，僅在此呈現完整性。相反的，最好避免在深處傳遞 callback。

深入理解

React 如何將 Hook 呼叫與 component 關聯？

React 會持續追蹤目前 render 的 component。感謝 [Hooks 的規則](#)，我們知道 Hook 只能從 React component（或自訂的 Hook -- 它們也只能從 React component 中被呼叫）被呼叫。

每一個 component 有一個「memory cell」的內部列表。它們只是我們可以放入一些資料的 JavaScript object。當你呼叫像是 `useState()` 的 Hook，它會讀取目前的 cell（或在第一次 render 時初始化它），並將指標移動到下一個。這就是多個 `useState()` 的呼叫，取得每個獨立的 local state。

Hook 現有的技術是什麼？

Hooks 綜合了幾個不同來源的想法：

- 在 [react-future repository](#) 中，我們使用 functional APIs 的舊實驗。
- React 社群使用 render props APIs 的實驗，包括 [Ryan Florence](#) 的 [Reactions Component](#)。
- [Dominic Gannaway](#) 的 `adopt keyword` 提案作為 render props 的語法糖。
- [DisplayScript](#) 中的 state 變數以及 state 單元。
- ReasonReact 中的 [Reducer components](#)。
- Rx 中的 [Subscriptions](#)。
- Multicore OCaml 中的 [Algebraic effects](#)。

[Sebastian Markbåge](#) 想出了 Hook 的原始設計，之後由 [Andrew Clark](#)、[Sophie Alpert](#)、[Dominic Gannaway](#) 以及其他 React 團隊的成員加以完善。

介紹 Hook

Hook 是 React 16.8 中增加的新功能。它讓你不必寫 class 就能使用 state 以及其他 React 的功能。

```
import React, { useState } from 'react';

function Example() {
  // 宣告一個新的 state 變數，我們稱作為「count」。
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

這個新 function `useState` 將會是我們學習的第一個「Hook」，不過這只是一個初窺的範例。如果這現在看起來並不合理，請別擔心！

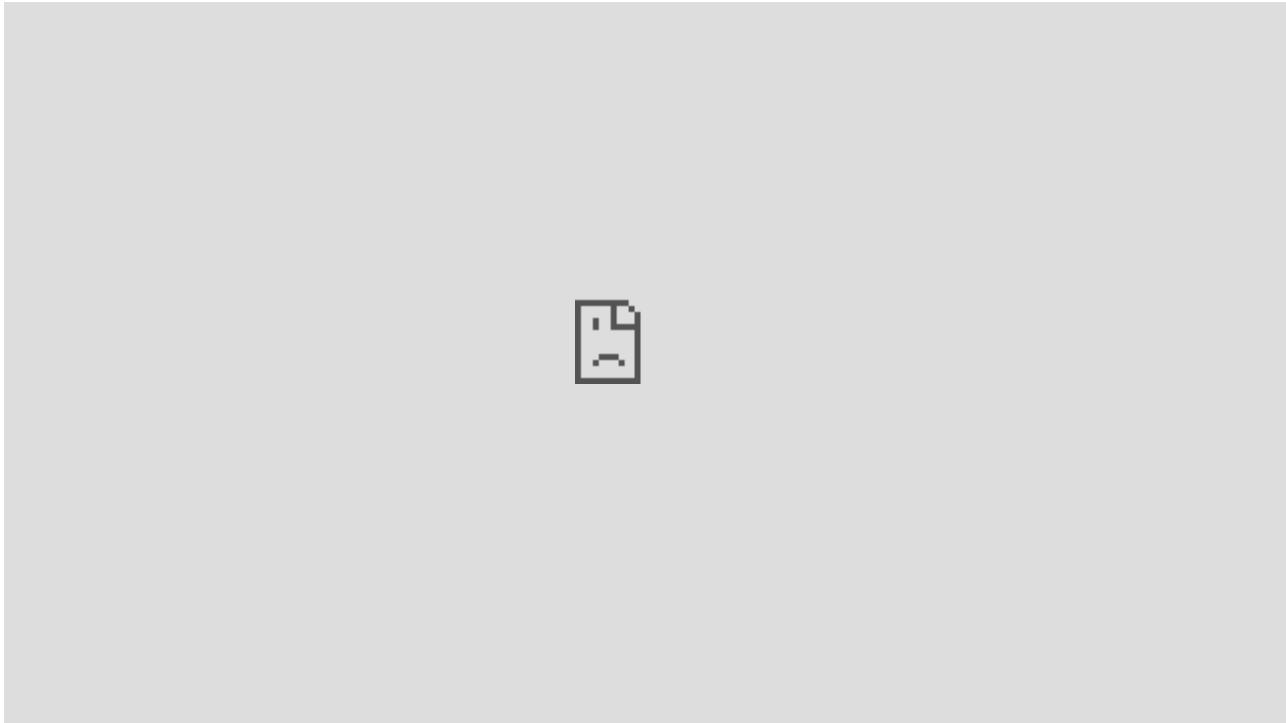
你可以開始在[下一頁學習 Hook](#)。在這一頁，我們將會繼續解釋為什麼要把 Hook 加到 React 以及他們如何幫助你寫出好的應用程式。

注意

React 16.8.0 是第一個支援 Hook 的版本。在升級時，記得不要忘記升級所有的套件包括 React DOM。React Native 則是已從 [0.59 的版本開始](#)支援 Hook。

介紹影片

在 React Conf 2018, Sophie Alpert 和 Dan Abramov 介紹了 Hook, 接著 Ryan Florence 展示了如何用它們來重構應用程式。在這裡觀看影片：



沒有 Breaking Change

在我們繼續之前, 請注意 Hook 是：

- **完全自由選擇使用。**你可以在幾個 component 中試用 Hook 而不用重寫任何既有的程式碼。不過如果你不想要, 並不需要現在學習或使用 Hook。
- **100% 向下相容。**Hook 沒有任何 breaking change。
- **現在即可使用。**隨著 v16.8.0 發佈, 現在即可使用 Hook。

目前沒有計畫要從 React 移除 class。你可以在這個頁面[下面的章節](#)閱讀更多跟逐步採用 Hook 有關的內容。

Hook 不會取代你對 React 概念的了解。相反的, Hook 是對你已經熟悉的 React 概念：props、state、context、refs 以及 lifecycle, 提供了一個更直接的 API。正如我們稍後將展示的那樣, Hook 還提供了一種新的強大方式來組合他們。

如果你只是想要開始學習 Hook, 可以自由地[直接跳到下一頁！](#)你也可以繼續閱讀這一頁來了解更多關於我們為什麼要加入 Hook, 以及我們如何在不重寫應用程式的情況下開始使用它們。

動機

Hook 解決了 React 中我們過去五年在編寫與維護數萬個 component 時所遇到的各種看似不相關的問題。無論你是在學習 React、每天使用它、還是更喜歡有著相似 component 模型的不同函式庫, 你可能都會發現其中一些問題。

在 Component 之間重用 Stateful 的邏輯很困難

React 沒有提供一個方法來把可重用的行為「附加」到一個 component 上（舉例來說，把它連結到一個 store）。如果你已經使用 React 一段時間，你或許會熟悉像是 [render props](#) 以及 [higher-order components](#)，這些試著解決這個問題的模式。但是這些模式要求你在使用它們時重新架構你的 component，這可能很麻煩，而且使程式碼更難追蹤。如果你在 React DevTools 上查看一個典型的 React 應用程式，你很可能會發現一個 component 的「包裝地獄」，被 provider、consumer、higher-order component、render props 以及其他抽象給層層圍繞。現在我們可以在 [DevTools 裡把他們過濾掉](#)，不過這指出了一個更根本的問題：React 需要一個更好的 primitive 來共用 stateful 邏輯。

使用 Hook，你可以從 component 抽取 stateful 的邏輯，如此一來它就可以獨立地被測試和重複使用。**Hook 讓你不需要改變 component 階層就能重用 stateful 的邏輯**。這讓在許多 component 之間共用或是與社群共用 Hook 很簡單。

我們將會在[打造你自己的 Hook](#) 討論更多相關內容。

複雜的 component 變得很難理解

我們時常必須維護那些一開始非常簡單，但後來變成充滿無法管理的 stateful 邏輯和 side effect 的 component。每個 lifecycle 方法常常包含不相關的邏輯混合在一起。舉例來說，component 可能會在 `componentDidMount` 和 `componentDidUpdate` 中抓取資料。但是，同一個 `componentDidMount` 方法可能也包含一些設置 event listener 的不相關邏輯，並在 `componentWillUnmount` 執行清除它們。會一起改變且彼此相關的程式碼被拆分，但完全不相關的程式碼卻放在同一個方法裡。這讓它很容易製造 bug 和不一致性。

在許多情況下，因為到處都是 stateful 邏輯，不可能把這些 component 拆分成更小的 component。而測試它們也很困難。這是許多人偏愛把 React 跟一個獨立的 state 管理函式庫結合的其中一個理由。然而，這常常引入了太多的抽象，要求你在不同檔案間跳來跳去，而且讓重用 component 更加困難。

為了解決這個問題，**Hook 讓你把一個 component 拆分成更小的 function，這基於什麼部分是相關的（像是設置一個 subscription 或是抓取資料）**，而不是強制基於 lifecycle 方法來分拆。你還可以選擇使用 reducer 來管理 component 的內部 state，使其更具可預測性。

我們將會在[使用 Effect Hook](#) 討論更多相關內容。

Class 讓人們和電腦同時感到困惑

除了使重用、組織程式碼更加困難以外，我們發現 class 可能是學習 React 的一大障礙。你必須了解 `this` 在 JavaScript 中如何運作，而這跟它在大部分程式語言中的運作方式非常不同。你必須記得 bind 那些 event handler。如果沒有 [ES2022 public class fields](#)，撰寫的程式碼會非常繁瑣。人們可以很好的理解 props、state 以及從上而下的資料流，但仍然在跟 class 斥鬥。React 中的 function component 和 class component 之間的差異以及什麼時候該使用哪一個，甚至在經驗豐富的 React 開發者之間也存在意見分歧。

此外，React 已經出現了大約五年，而我們想要確保它在下一個五年保持競爭力。如同 [Svelte](#)、[Angular](#)、[Glimmer](#)，以及其他所展示的，component 的[提前編譯](#)有很大的未來潛力。特別是如果它不侷限在模板上。最近，我們在實驗使用 [Prepack](#) 來做 [component folding](#)，而我們已經看到大有可為的早期結果。然而，我們發現使用 class component 會鼓勵一些不是故意的模式，這會讓這些最佳化回到一條比較慢的路。Class 在現在的工具上也有不少問題。例如，class 沒有辦法很好的壓縮，而且它讓 hot reload 變得脆弱而且不可靠。我們想要提出一個可以讓程式碼更可能留在可最佳化的路徑上的 API。

為了解決這些問題，**Hook 讓你不需要 class 就能使用更多 React 的功能。** 從概念上來看，React component 一直都更接近 function。Hook 擁抱 function，但沒有犧牲 React 的實際精神。Hook 提供取用 imperative 技術的辦法且不要求你學習複雜的 functional 或 reactive programming 技術。

範例

[Hook 總覽](#)是一個開始學習 Hook 的好地方。

逐步的採用策略

長話短說：目前沒有計畫要從 React 移除 class。

我們知道 React 的開發者們專注在交付產品，沒有時間仔細去看每一個被釋出的新 API。Hook 非常新，所以在考慮學習或採用它們之前，等待更多範例和教學可能會更好。

我們也了解要添加一個新的 primitive 到 React 的標準非常高。我們已經準備了一個[詳盡的 RFC](#)給好奇的讀者，它藉由更多細節深入探討動機，並針對特定的設計決策以及先關的既有技術提供額外的觀點。

至關重要的是，Hook 可以與既有的程式碼一起運作，因此你可以逐步採用它們。 不用急著轉換到 Hook。我們建議避免任何「巨大的改寫」，尤其是那些既有、複雜的 class component。要開始「從 Hook 的角度思考」需要一些思維上的轉變。根據我們的經驗，最好先在新的且重要性較低的 component 中練習使用 Hook，並確保團隊中的每個人都不會對它感到不舒服。在你嘗試了 Hook 之後，請自由地[給我們一些回饋](#)，不管是正面的還是負面的可以。

我們有意讓 Hook 能涵蓋 class 所有既有的使用案例，**但我們會在可見的未來繼續支援 class component。** 在 Facebook 裡，我們有數以萬計的 component 是用 class 寫的，而我們絕對沒有計劃要改寫它們。取而代之，我們開始在新的程式碼中使用 Hook 並讓它們跟 class 共存。

常見問題

我們準備了一個 [Hook 的常見問題頁面](#)來回答 Hook 最常見到的問題。

下一步

看到這頁的最後面，你應該已經大致了解 Hook 正在解決什麼問題，不過許多細節你可能都還不大清楚。別擔心！讓我們馬上前往[下一頁](#)，我們可以在那裡開始藉由範例學習 Hook。

Hook 概觀

Hook 是 React 16.8 中增加的新功能。它讓你不必寫 class 就能使用 state 以及其他 React 的功能。

Hook [向後兼容](#)。這一頁提供 Hook 的簡介給有經驗的 React 使用者們。這是一個快節奏的簡介，如果你有疑惑，看一下黃色框框如下：

詳細解釋

閱讀 [Hook 介紹的動機部分](#)來了解為什麼我們將 Hook 加入 React。

↑↑↑ 每一個章節結束都有一個黃色框框像這樣。他們連到詳細解釋。

📌 State Hook

這個範例 render 一個計數器。當你按下按鈕，他會增加數值：

```
import React, { useState } from 'react';

function Example() {
  // 宣告一個新的 state 變數，我們叫他「count」
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

這裡，`useState` 是一個 *Hook*（我們馬上會聊到他是什麼意思）。我們在 function component 中呼叫他們用來加入一些 local state。React 會在重新 render 的頁面之間保留這些 state。`useState` 回傳一組數值：目前 state 數值和一個可以讓你更新 state 的 function。你可以從 event handler 或其他地方呼叫這個 function 來更新他。很類似 `this.setState` 在 class 當中的用法，除了他不會將舊的與新的 state 合併在一起。（我們將會在[使用 State Hook](#) 中示範一個範例比較 `useState` 與 `this.state`。）

`useState` 唯一的 argument 是初始狀態。在上面的例子中，他是 `0` 因為我們的計數器從零開始。注意不像 `this.state`，state 在這裡不需要一定是 object，雖然你要也可以。初始狀態 argument 只有在第一次 render 的時候會被用到。

宣告多個 state 變數

你可以在一個 component 中使用 State Hook 不只一次：

```
function ExampleWithManyStates() {
  // 告多個 state 變數!
  const [age, setAge] = useState(42);
  const [fruit, setFruit] = useState('banana');
  const [todos, setTodos] = useState([{ text: 'Learn Hooks' }]);
  // ...
}
```

陣列解構賦值語法讓我們可以給用呼叫 `useState` 宣告的 state 變數不同的名字，這些名字不是 `useState` API 的一部份。相反地，React 假設如果你呼叫 `useState` 多次，你在每次 render 用同樣的順序。我們之後將會回來討論為什麼可以這樣做與什麼時候他很有用。

但是什麼是 Hook？

Hook 是 function，他讓你可以從 function component 「hook into」React state 與生命週期功能。Hook 在 class 裡面沒有辦法用——他們讓你不用 class 就能使用 React。（我們不建議你通宵重寫現存的 component 但是如果想要的話，你可以開始在新的 component 當中使用。）

React 提供一些內建 Hook 像是 `useState`。你也可以打造你自己的 Hook 用來在不同的 component 之間重複使用 stateful 邏輯。我們先來看看內建 Hook。

[詳細解釋](#)

你可以在[使用 State Hook](#) 了解更多 State Hook。

⚡ Effect Hook

你從前可能在 React component 做過 fetch 資料、訂閱、或手動改變 DOM。我們稱這些操作「side effect」（或簡稱 effect）因為他們可以影響其他 component 且在 render 期間無法完成。

Effect Hook `useEffect` 在 function component 中加入運作 side effect 的能力。他和 `componentDidMount`，`componentDidUpdate`，與 `componentWillUnmount` 有著同樣的宗旨，但整合進一個單一的 API。（我們將在[使用 Effect Hook](#) 中用範例比較 `useEffect` 和這些方法。）

舉例來說，這個 component 在 React 更新 DOM 之後設定文件標題：

```
import React, { useState, useEffect } from 'react';

function Example() {
  const [count, setCount] = useState(0);

  // 與 componentDidMount 和 componentDidUpdate 類似：
  useEffect(() => {
    // 使用瀏覽器 API 更新文件標題
    document.title = `You clicked ${count} times`;
  });

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

```

        </button>
    </div>
);
}

```

當你呼叫 `useEffect` 時，你告訴 React 刷新 DOM 變動之後執行你的「effect」。Effect 在 component 裡面被宣告所以他們有權限訪問他的 props 和 state。預設之下，React 在每一次 render 之後執行 effect —— 包括第一次 render。(我們會在[使用 Effect Hook](#) 做更多他和 class lifecycle 的比較。)

Effect 可以透過回傳一個 function 選擇性的定義如何「清理」。舉例來說，這個 component 使用 effect 來訂閱朋友的上線狀態，並在取消訂閱之後清理。

```

import React, { useState, useEffect } from 'react';

function FriendStatus(props) {
  const [isOnline, setIsOnline] = useState(null);

  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  useEffect(() => {
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);

    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}

```

在這個例子中，當 component unmount 和重新運行 effect 的時候，React 會取消訂閱我們的 `ChatAPI` 因為一連串依序的 render。(如果你想的話，如果我們傳遞給 `ChatAPI` 的 `props.friend.id` 沒有改變，有辦法告訴 React 跳過重新訂閱。)

就像是 `useState` 一樣，你可以在一個 component 中使用超過一個 effect：

```

function FriendStatusWithCounter(props) {
  const [count, setCount] = useState(0);
  useEffect(() => {
    document.title = `You clicked ${count} times`;
  });

  const [isOnline, setIsOnline] = useState(null);
  useEffect(() => {
    ChatAPI.subscribeToFriendStatus(props.friend.id, handleStatusChange);
    return () => {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id, handleStatusChange);
    };
  });

  function handleStatusChange(status) {

```

```

    setIsOnline(status.isOnline);
}
// ...

```

Hook 讓你在 component 中透過彼此相關的部分組織 side effect (像是加入或移除訂閱)，而不是強迫根據生命週期方法分散各處。

詳細解釋

你可以在[使用 Effect Hook](#) 了解更多 Effect Hook。

👉 Hook 的規則

Hook 是 JavaScript function，但是他們強加了兩條額外的規則：

- 只在**最上層**呼叫 Hook。不要在迴圈、判斷式、或是嵌套 function 中呼叫 Hook。
- 只在 **React function component** 呼叫 Hook。不要在一般 JavaScript function 中呼叫 Hook。（只有一個其他有效的地方可以呼叫 Hook——你自己的客製化 Hook。我們馬上會學到他們。）

我們提供了一個 [linter plugin](#) 自動強制套用這些規則。我們了解這些規則第一眼看起來很多限制或是很讓人疑惑，但是他們是讓 Hook 成功運作的必要條件。

詳細解釋

你可以在[Hook 的規則](#)了解 more Hook 的規則。

💡 打造你自己的 Hook

有時候，我們想要在 component 之間重複使用某些 stateful 邏輯。傳統上，這個問題有兩種熱門的答案：[higher-order components](#) 與 [render props](#)。客製化 Hook 讓你不用在你的 tree 中加入更多 component 就能做到。

在這一頁前面，我們提到 `FriendStatus` component 呼叫 `useState` 與 `useEffect` Hook 來訂閱朋友的線上狀態。假如我們也想要在另一個 component 重複使用這個訂閱邏輯。

首先，我們抽離這個邏輯到客製化 Hook `useFriendState` 之中：

```

import React, { useState, useEffect } from 'react';

function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);

  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  useEffect(() => {
    ChatAPI.subscribeToFriendStatus(friendID, handleStatusChange);
    return () => {

```

```

        ChatAPI.unsubscribeFromFriendStatus(friendID, handleStatusChange);
    };
});

return isOnline;
}

```

他接收 `friendID` 做為 argument 且回傳我們的朋友是否在線上。

現在我們在兩個 component 中都可以使用：

```

function FriendStatus(props) {
  const isOnline = useFriendStatus(props.friend.id);

  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}

function FriendListItem(props) {
  const isOnline = useFriendStatus(props.friend.id);

  return (
    <li style={{ color: isOnline ? 'green' : 'black' }}>
      {props.friend.name}
    </li>
  );
}

```

這兩個 component 的 state 是完全獨立的。Hook 是一種重複使用 *stateful* 邏輯的方式，不是 state 本身。實際上，每一次呼叫一個 Hook 都有一個完全獨立且隔離的 state —— 所以你甚至可以在一個 component 使用同一個客製化 Hook 兩次。

客製化 Hook 比較像慣例而不是功能，如果一個 function 的名字是「`use`」開頭且他呼叫其他 Hook，我們將他稱之為客製化 Hook。使用 Hook 時，`useSomething` 的命名慣例是我們的 linter plugin 如何能夠在程式碼中找到 bug 的原因。

你可以寫客製化 Hook 涵蓋廣泛地使用案例像是表單處理、動畫、宣告式訂閱、計時器和許多我們可能沒有想到的。我們很興奮看到 React 社群會想到什麼樣的 Hook。

詳細解釋

你可以在[打造你自己的 Hook](#) 了解更多客製化 Hook。

💡 其他 Hook

還有一些比較少用的內建 Hook 你可能會覺得很有用。舉例來說，`useContext` 讓你不需要巢狀化就可以訂閱 React context：

```
function Example() {  
  const locale = useContext(LocaleContext);  
  const theme = useContext(ThemeContext);  
  // ...  
}
```

`useReducer` 讓你在複雜的 component 中用 reducer 管理 local state：

```
function Todos() {  
  const [todos, dispatch] = useReducer(todosReducer);  
  // ...  
}
```

詳細解釋

你可以在 [Hook API 參考](#) 了解更多所有的內建 Hook。

下一步

呼！講很快！如果有什麼讓你覺得沒道理或是你想了解更多細節，你可以閱讀下一頁，從 [State Hook](#) 文件開始。

你也可以看看 [Hook API reference](#) 與 [Hook 常見問題](#)。

最後，不要漏了 [Hook 介紹頁面](#)，這裡解釋了為什麼我們加入了 Hook 與我們如何和 class 一起使用——不用重寫我們的應用程式。

[Go to TOC](#)

Hooks API 參考

Hook 是 React 16.8 中增加的新功能。它讓你不必寫 class 就能使用 state 以及其他 React 的功能。

本頁面描述 React 中內建 Hook 的 API。

如果你剛開始接觸 Hook，你可能會想先查閱 [Hook 概論](#)。你也可以在 [Hook 常見問題](#)中找到有用的資訊。

- [基礎的 Hook](#)
 - `useState`
 - `useEffect`
 - `useContext`
- [額外的 Hook](#)
 - `useReducer`
 - `useCallback`
 - `useMemo`
 - `useRef`
 - `useImperativeHandle`
 - `useLayoutEffect`
 - `useDebugValue`
 - `useDeferredValue`
 - `useTransition`
 - `useId`
- [Library Hooks](#)
 - `useSyncExternalStore`
 - `useInsertionEffect`

基礎的 Hook

`useState` {#usestate}

```
const [state, setState] = useState(initialState);
```

回傳一個 state 的值，以及更新 state 的 function。

在首次 render 時，回傳的 `state` 的值會跟第一個參數 (`initialState`) 一樣。

`setState` function 是用來更新 state。它接收一個新的 state 並將 component 的重新 render 排進隊列。

```
setState(newState);
```

在後續的重新 render，`useState` 回傳的第一個值必定會是最後更新的 state。

注意

React 確保 `setState` function 本身是穩定的，而且不會在重新 render 時改變。這就是為什麼可以安全地從 `useEffect` 或 `useCallback` 的依賴列表省略它。

函數式更新

如果新的 state 是用先前的 state 計算出，你可以傳遞一個 function 到 `useState`。該 function 將接收先前的 state，並回傳一個已更新的值。下列的計算器 component 範例示範了 `useState` 的兩種用法。

```
function Counter({initialCount}) {
  const [count, setCount] = useState(initialCount);
  return (
    <>
      Count: {count}
      <button onClick={() => setCount(initialCount)}>Reset</button>
      <button onClick={() => setCount(prevCount => prevCount - 1)}>-</button>
      <button onClick={() => setCount(prevCount => prevCount + 1)}>+</button>
    </>
  );
}
```

「+」和「-」按鈕使用了函數式形式，因為被更新的值是基於先前的值。但是「Reset」按鈕則使用普通形式，因為它總是把 count 設定回初始值。

如果你的 update 函式回傳與目前的 state 相同的值，後續的 render 將會被完整跳過。

注意

與 class component 的 `setState` 方法不同，`useState` 不會自動合併更新 object。你可以用函數式更新的形式結合 object spread 語法來達到相同效果：

```
const [state, setState] = useState({});
setState(prevState => {
  // 也可以使用 Object.assign
  return {...prevState, ...updatedValues};
});
```

另一個選擇是 `useReducer`，它更適合用於管理有多個子數值的 state object。

惰性初始 state

`initialState` 參數只會在初始 render 時使用，在後續 render 時會被忽略。如果初始 state 需要通過複雜的計算來獲得，你可以傳入一個 function，該 function 只會在初始 render 時被調用：

```
const [state, setState] = useState(() => {
  const initialState = someExpensiveComputation(props);
  return initialState;
});
```

跳過 state 更新

如果你使用與目前 state 相同值來更新 State Hook，React 將會跳過子 component 的 render 及 effect 的執行。
(React 使用 `Object.is` 來比較。)

請注意 React 可能仍需要在跳過 render 之前 render 該 component。這不應該是個問題，因為 React 不會不必要的「深入」到 component tree 中。如果你在 render 當中執行了昂貴的計算，你可以使用 `useMemo` 來最佳化。

Batching of state updates

React may group several state updates into a single re-render to improve performance. Normally, this improves performance and shouldn't affect your application's behavior.

Before React 18, only updates inside React event handlers were batched. Starting with React 18, [batching is enabled for all updates by default](#). Note that React makes sure that updates from several *different* user-initiated events -- for example, clicking a button twice -- are always processed separately and do not get batched. This prevents logical mistakes.

In the rare case that you need to force the DOM update to be applied synchronously, you may wrap it in `flushSync`. However, this can hurt performance so do this only where needed.

`useEffect` {#useeffect}

```
useEffect(didUpdate);
```

接受一個包含指令式，且可能有副作用程式碼的 function。

在 function component (指的是 React 的 render 階段) 的 function 內，mutation、subscription、timer、日誌記錄、以及其他 side effect 是不被允許的。因為這可能會導致容易混淆的 bug 和不一致的 UI。

相反的，使用 `useEffect`。傳遞到 `useEffect` 的 function 會在 render 到螢幕之後執行。可以把 effect 看作 React 從純函式世界通往指令式世界的跳脫方式。

在預設情況下，effect 會在每一個完整 render 後執行，但你也可以選擇它們在[某些值改變的時候](#)才執行。

清除一個 effect

通常來說，在 component 離開螢幕之前需要清除 effect 所建立的資源，例如像是 subscription 或計時器的 ID。要做到這一點，傳遞到 `useEffect` 的 function 可以回傳一個清除的 function。例如，要建立一個 subscription：

```
useEffect(() => {
  const subscription = props.source.subscribe();
  return () => {
    // Clean up the subscription
    subscription.unsubscribe();
  };
});
```

清除 function 會在 component 從 UI 被移除前執行，來防止 memory leak。此外，如果 component render 了數次（它們通常會這樣），在執行下一個 effect 前，上一個 effect 就已被清除。在上述的例子中，意味著每一次更新都會建立一個新 subscription。要避免每次更新都觸發 effect 的執行，請參閱下一個章節。

Effect 的時機

與 `componentDidMount` 和 `componentDidUpdate` 不同，在延遲事件期間，傳遞給 `useEffect` 的 function 會在 `layout` 和 `render` 之後觸發。這使它適用於很多常見的 side effect，例如設定 subscription 和 event handler，因為絕大部份的工作都不應該阻礙瀏覽器更新畫面。

然而，不是所有的 effect 都可以被延後。例如，使用者可見的 DOM 變更必須在下一次繪製之前同步觸發，這樣使用者才不會感覺到視覺不一致。（概念上類似被動和主動 event listener 的區別。）為這類型的 effect，React 提供了一個額外的 `useLayoutEffect` Hook。它和 `useEffect` 的結構相同，只是執行的時機不同而已。

此外，從 React 18 開始，傳給 `useEffect` 的 function 將在 `layout` 和 `paint` 之前 同步的觸發，當它是一個離散的使用者輸入（像是點擊）或當它是一個被 wrap 在 `flushSync` 的更新結果。這個行為讓 event system 或 `flushSync` 的 caller 觀察 effect 的結果。

注意

這只會影響傳遞給 `useEffect` 的 function 的被呼叫時間 - 在這些 effect 中安排的更新仍然會被延遲。這與 `useLayoutEffect` 不同，後者會觸發 function 並立即處理其更新。

雖然 `useEffect` 會被延遲直到瀏覽器繪制完成，但會保證在任何新 `render` 前執行。React 會在開始新一個更新前刷新上一輪 `render` 的 effect。

有條件的觸發 effect

effect 的預設行為是在每次完成 `render` 後觸發 effect。這樣的話，如果其中一個依賴有改變，則會重新建立一個 effect。

然而，在某些情況下這可能矯枉過正，例如在上一章節的 subscription 範例。我們只需要在 `source` prop 改變後才重新建立 subscription，而不需要在每次更新後。

要實現這一點，可以向 `useEffect` 傳遞第二個參數，它是該 effect 所依賴的值 array。我們更新後的範例如下：

```
useEffect(
  () => {
    const subscription = props.source.subscribe();
    return () => {
      subscription.unsubscribe();
    };
  },
  [props.source],
);
```

現在只有當 `props.source` 改變時才會重新建立 subscription。

注意

若你使用這個最佳化的方式，請確保該 array 包含了所有在該 component 中會隨時間而變的值（例如 `props` 和 `state`）以及在該 effect 所使用到的值。否則，你的程式碼會引用先前 `render` 的舊變數。了解更多[如何處理 function](#) 和當 array 的值頻繁變化時的處理。

如果你想 effect 只執行和清除一次（在 mount 和 unmount），你可以傳遞一個空的 array ([]) 作為第二個參數。這告訴 React 你的 effect 沒有依賴任何在 props 或 state 的值，所以它永遠不需被再次執行。這並不是一個特殊處理 -- 它依然遵循依賴 array 的運作方式。

如果你傳入了一個空的 array ([])，effect 內部的 props 和 state 就一直擁有其初始值。儘管傳入 [] 作為第二個參數有點類似 `componentDidMount` 和 `componentWillUnmount` 的思維模式，但其實有[更好的方法](#)來避免過於頻繁地重複執行 effect。而且，不要忘記 React 會延後執行 `useEffect` 直至瀏覽器完成繪制，所以額外的工作也不會是太大問題。

我們建議使用 `eslint-plugin-react-hooks` 中的 `exhaustive-deps` 規則。它會在依賴錯誤時發出警告並提出修正建議。

依賴 array 並不作為傳到 effect function 的參數。但從概念上來說，這是它所代表的：所有在 effect function 中引用的值都應該出現在依賴 array 中。在未來，一個足夠先進的編譯器可以自動建立這個 array。

`useContext` {#usecontext}

```
const value = useContext(MyContext);
```

接收一個 context object (`React.createContext` 的回傳值) 並回傳該 context 目前的值。Context 目前的值是取決於由上層 component 距離最近的 `<MyContext.Provider>` 的 `value` prop。

當 component 上層最近的 `<MyContext.Provider>` 更新時，該 hook 會觸發重新 render，並使用最新傳遞到 `MyContext` 的 context value 傳送到 `MyContext` provider。即便 ancestor 使用 `React.memo` 或 `shouldComponentUpdate`，重新 render 仍然從使用 `useContext` 的 component 本身開始。

不要忘記 `useContext` 的參數必需為 `context object` 自己：

- 正確: `useContext(MyContext)`
- 錯誤: `useContext(MyContext.Consumer)`
- 錯誤: `useContext(MyContext.Provider)`

呼叫 `useContext` 的 component 總是會在 context 值更新時重新 render。如果重新 render component 的操作很昂貴，你可以[透過 memoization 來最佳化](#)。

注意

假若你在接觸 hook 前已熟悉 context API，`useContext(MyContext)` 就相等於 class 中的 `static contextType = MyContext` 或 `<MyContext.Consumer>`。

`useContext(MyContext)` 只能讓你讀取 context 及訂閱其變更。你仍然需要在 tree 的上層使用 `<MyContext.Provider>` 來提供 context 的值。

與 `Context.Provider` 放在一起

```

const themes = {
  light: {
    foreground: "#000000",
    background: "#eeeeee"
  },
  dark: {
    foreground: "#ffffff",
    background: "#222222"
  }
};

const ThemeContext = React.createContext(themes.light);

function App() {
  return (
    <ThemeContext.Provider value={themes.dark}>
      <Toolbar />
    </ThemeContext.Provider>
  );
}

function Toolbar(props) {
  return (
    <div>
      <ThemedButton />
    </div>
  );
}

function ThemedButton() {
  const theme = useContext(ThemeContext);

  return (
    <button style={{ background: theme.background, color: theme.foreground }}>
      I am styled by theme context!
    </button>
  );
}

```

此為使用 Hook 修改之前 [Context Advanced Guide](#) 中的範例，你可以在那裡了解更多 Context 的資訊，像是何時以及如何使用 Context。

額外的 Hooks

以下的 Hook，有些是上一節中基礎的 Hook 的變異，有些則是在特殊情況下使用。不用特地預先學習它們。

`useReducer {#usereducer}`

```
const [state, dispatch] = useReducer(reducer, initialArg, init);
```

`useState` 的替代方案。接受一個 `(state, action) => newState` 的 reducer，然後回傳現在的 state 以及其配套的 `dispatch` 方法。（如果你熟悉 Redux，你已經知道這如何運作。）

當你需要複雜的 state 邏輯而且包括多個子數值或下一個 state 依賴之前的 state，`useReducer` 會比 `useState` 更適用。而且 `useReducer` 可以讓你觸發深層更新的 component 作效能的最佳化，因為你可以傳遞 `dispatch` 而不是 callback。

以下是用 reducer 重寫 `useState` 一節的範例：

```
const initialState = {count: 0};

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return {count: state.count + 1};
    case 'decrement':
      return {count: state.count - 1};
    default:
      throw new Error();
  }
}

function Counter() {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <>
      Count: {state.count}
      <button onClick={() => dispatch({type: 'decrement'})}>-</button>
      <button onClick={() => dispatch({type: 'increment'})}>+</button>
    </>
  );
}
```

注意

React 確保 `dispatch` function 本身是穩定的，而且不會在重新 render 時改變。這就為什麼可以安全地從 `useEffect` 或 `useCallback` 的依賴列表省略它。

指定初始 state

有兩種不同初始化 `useReducer` state 的方法。你可以根據使用場景選擇任何一種。最簡單的方法就是把初始 state 作為第二個參數傳入：

```
const [state, dispatch] = useReducer(
  reducer,
  {count: initialCount}
);
```

注意

React 不使用 `state = initialState` 這個由 Redux 推廣開來的參數慣例。初始值有時需要依賴於 prop，因此需要在呼叫 Hook 時指定。如果你較偏愛上述的慣例，你可以呼叫 `useReducer(reducer, undefined, reducer)` 來模擬 Redux 的行為，但這是不鼓勵的。

惰性初始化

你也可以惰性的建立初始 state。你可以傳入 `init` function 作為第三個參數。初始的 state 會被設定為 `init(initialArg)`。

這樣讓你可以將計算初始 state 的邏輯提取到 reducer 外。而且也方便了將來處理重置 state 的 action：

```
function init(initialCount) {
  return {count: initialCount};
}

function reducer(state, action) {
  switch (action.type) {
    case 'increment':
      return {count: state.count + 1};
    case 'decrement':
      return {count: state.count - 1};
    case 'reset':
      return init(action.payload);
    default:
      throw new Error();
  }
}

function Counter({initialCount}) {
  const [state, dispatch] = useReducer(reducer, initialCount, init);
  return (
    <>
      Count: {state.count}
      <button onClick={() => dispatch({type: 'reset', payload: initialCount})}>
        Reset
      </button>
      <button onClick={() => dispatch({type: 'decrement'})}>-</button>
      <button onClick={() => dispatch({type: 'increment'})}>+</button>
    </>
  );
}
```

跳過 `dispatch`

如果你在 Reducer Hook 回傳的值與目前的 state 相同，React 將會跳過 child component 的 render 及 effect 的執行。（React 使用 `Object.is` 來比較。）

請注意 React 可能仍需要在跳過 render 之前 render 該 component。這不應該是個問題，因為 React 不會不必要的「深入」到 tree 中。如果你在 render 當中執行了昂貴的計算，你可以使用 `useMemo` 來最佳化。

`useCallback` {#usecallback}

```
const memoizedCallback = useCallback(
  () => {
    doSomething(a, b);
  },
  [a, b],
);
```

回傳一個 `memoized` 的 callback。

傳遞一個 inline callback 及依賴 array。`useCallback` 會回傳該 callback 的 memoized 版本，它僅在依賴改變時才會更新。當傳遞 callback 到已經最佳化的 child component 時非常有用，這些 child component 依賴於引用相等性來防止不必要的 render (例如，`shouldComponentUpdate`)

`useCallback(fn, deps)` 相等於 `useMemo(() => fn, deps)`。

注意

依賴 array 並不作為傳到 callback 的參數。但從概念上來說，這是它所代表的：所有在 callback 中引用的值都應該出現在依賴 array 中。在未來，一個足夠先進的編譯器可以自動建立這個 array。

我們建議使用 `eslint-plugin-react-hooks` 中的 `exhaustive-deps` 規則。它會在依賴錯誤時發出警告並提出修正建議。

useMemo {#usememo}

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b), [a, b]);
```

回傳一個 `memoized` 的值。

傳遞一個「建立」function 及依賴 array。`useMemo` 只會在依賴改變時才重新計算 `memoized` 的值。這個最佳化可以避免在每次 render 都進行昂貴的計算。

要謹記傳到 `useMemo` 的 function 會在 render 期間執行。不要做一些通常不會在 render 期間做的事情。例如，處理 side effect 屬於 `useEffect`，而不是 `useMemo`。

如果沒有提供 array，每次 render 時都會計算新的值。

你可以把 `useMemo` 作為效能最佳化的手段，但請不要把它當作成語意上的保證。在將來，React 可能會選擇「忘記」某些之前已 memorize 的值並在下一次 render 時重新計算，例如，為已離開螢幕的 component 釋放記憶體。先撰寫沒有 `useMemo` 也可執行的代碼 — 然後再加入它來做效能最佳化。

注意

依賴 array 並不作為傳到 function 的參數。但從概念上來說，這是它所代表的：所有在 callback 中引用的值都應該出現在依賴 array 中。在未來，一個足夠先進的編譯器可以自動建立這個 array。

我們建議使用 `eslint-plugin-react-hooks` 中的 `exhaustive-deps` 規則。它會在依賴錯誤時發出警告並提出修正建議。

useRef {#useref}

```
const refContainer = useRef(initialValue);
```

`useRef` 回傳一個 mutable 的 ref object, `.current` 屬性被初始為傳入的參數 (`initialValue`)。回傳的 object 在 component 的生命週期將保持不變。

一個常見的使用情境就是命令式的訪問 child component :

```
function TextInputWithFocusButton() {
  const inputEl = useRef(null);
  const onButtonClick = () => {
    // `current` points to the mounted text input element
    inputEl.current.focus();
  };
  return (
    <>
      <input ref={inputEl} type="text" />
      <button onClick={onButtonClick}>Focus the input</button>
    </>
  );
}
```

本質上, `useRef` 就像一個可以持有 mutable 的值在 `.current` 屬性的「盒子」。

你應該熟悉 `ref`, 它一種主要是用來訪問 DOM 的方式。如果你在 React 中以 `<div ref={myRef} />` 傳入 `ref` object, 無論節點如何改變, React 都會將其 `.current` 屬性設為相應的 DOM 節點。

然而, `useRef()` 比 `ref` 屬性更有用。它可以很方便地持有任何 mutable 的值, 跟 class 中的 instance field 類似。

這是因為 `useRef()` 會建立一個普通的 JavaScript object。`useRef()` 和自建一個 `{current: ...}` object 的唯一不同是, `useRef` 在每次 render 時都會給你同一個的 ref object。

請記住 `useRef` 在其內容有變化時並不會通知你。變更 `.current` 屬性不會觸發重新 render。如果你想要在 React 繩定或解綁 DOM 節點的 ref 時執行程式碼, 你可能需要使用 `callback ref` 來實現。

useImperativeHandle {#useimperativehandle}

```
useImperativeHandle(ref, createHandle, [deps])
```

`useImperativeHandle` 可以讓使用 `ref` 時能向父 component 暴露自定義的 instance 值。一如既往, 在大多數的情況下應避免使用 `ref` 的命令式代碼。`useImperativeHandle` 應與 `forwardRef` 一同使用：

```
function FancyInput(props, ref) {
  const inputRef = useRef();
  useImperativeHandle(ref, () => ({
    focus: () => {
      inputRef.current.focus();
    }
  }));
  return <input ref={inputRef} ... />;
}
FancyInput = forwardRef(FancyInput);
```

在這個範例中, `render <FancyInput ref={inputRef} />` 的父 component 能呼叫 `inputRef.current.focus()`。

useLayoutEffect {#uselayouteffect}

與宣告 `useEffect` 本身相同，但它會在所有 DOM 改變後，同步調用。使用它來讀取 DOM layout 並同步重新 render。在瀏覽器執行繪製之前，`useLayoutEffect` 內部的更新將被同步刷新。

盡可能使用標準的 `useEffect` 來避免阻礙視覺上的更新。

注意

如果你是從 class component migrate 程式碼，請注意 `useLayoutEffect` 與 `componentDidMount` 和 `componentDidUpdate` 的呼叫時機是一樣。不過，我們建議先使用 `useEffect`，只當它有問題時才嘗試使用 `useLayoutEffect`。

如果你使用伺服器 render，請記住 `useLayoutEffect` 或 `useEffect` 均不會執行，直到 JavaScript 完成載入。這是為什麼在伺服器 render 的 component 包含 `useLayoutEffect` 時 React 會發出警告。要解決這問題，把該邏輯搬到 `useEffect` 裡（如果首次 render 不需要該邏輯），或把 component 延遲到客戶端完成 render 後才出現（如果直到 `useLayoutEffect` 執行前 HTML 都會錯亂的情況下）。

要在伺服器 render 的 HTML 排除需要 layout effect 的 component，可以利用 `showChild && <Child />` 進行條件 render，並使用 `useEffect(() => { setShowChild(true); }, [])` 來延遲顯示。這樣，UI 就不會在完成 render 之前顯示錯亂了。

useDebugValue {#usedebugvalue}

`useDebugValue(value)`

`useDebugValue` 可以用來在 React DevTools 中顯示自訂義 hook 的標籤。

例如，在「[打造你的 Hook](#)」中提及的 `useFriendStatus` 自定義 Hook：

```
function useFriendStatus(friendID) {
  const [isOnline, setIsOnline] = useState(null);

  // ...

  // Show a label in DevTools next to this Hook
  // e.g. "FriendStatus: Online"
  useDebugValue(isOnline ? 'Online' : 'Offline');

  return isOnline;
}
```

注意

我們不建議在每個自定義 Hook 都加上 debug 值。它在自定義 Hook 的共享函式庫中才是最有價值的。

延遲格式化 Debug Value

在某些情況下，格式化一個顯示值可能會是昂貴的操作。除非實際檢查 Hook，否則沒有必要。

因此，`useDebugValue` 接受一個格式化 function 作為可選的第二個參數。該 function 只有在 Hook 被檢查時才會被呼叫。它接受 debug 值作為參數，然後回傳一個被格式化的顯示值。

例如，一個回傳 `Date` 值的自定義 Hook 可以通過以下的格式化 function 來避免不必要的呼叫 `toDateString` function：

```
useDebugValue(date, date => date.toDateString());
```

useDeferredValue {#usedeferredvalue}

```
const deferredValue = useDeferredValue(value);
```

`useDeferredValue` 接受一個值，並且回傳值的新拷貝，新的拷貝會被推遲到更緊急的更新。如果目前的 render 是緊急更新的結果，像是使用者輸入，React 將會回傳先前的值，並接著在緊急更新完成後，render 新的值。

這個 hook 類似 user-space 的 hook，被用在 debouncing 或 throttling 來延遲更新。使用 `useDeferredValue` 的好處是 React 將會在其他工作完成後，立即進行更新（而不是等待一個任意的時間），像是 `startTransition`，延遲的值可以暫停而不會觸發一個現有內容的 unexpected fallback。

Memoizing 延遲的 Children

`useDeferredValue` 只延遲你傳送給它的值。如果你想要防止一個 child component 在一個緊急更新中不斷重新 render，你必須使用 `React.memo` 或 `React.useMemo` 來 memoize component：

```
function Typeahead() {
  const query = useSearchQuery('');
  const deferredQuery = useDeferredValue(query);

  // Memoizing tells React to only re-render when deferredQuery changes,
  // not when query changes.
  const suggestions = useMemo(() =>
    <SearchSuggestions query={deferedQuery} />,
    [deferedQuery]
  );

  return (
    <>
      <SearchInput query={query} />
      <Suspense fallback="Loading results...">
        {suggestions}
      </Suspense>
    </>
  );
}
```

Memoizing children 告訴 React 它只需要在 `deferedQuery` 重新 render，而不是在 `query` 變更時。這個警告不是 `useDeferredValue` 獨有的，它與你使用的 debouncing 或 throttling 類似的 hook 使用的模式相同。

useTransition {#usetransition}

```
const [isPending, startTransition] = useTransition();
```

回傳一個 transition pending 狀態的 stateful 值，以及一個啟動 function。

`startTransition` lets you mark updates in the provided callback as transitions: 讓你在提供的 callback 中標記更新為 transitions：

```
startTransition(() => {
  setCount(count + 1);
})
```

`isPending` 表示當一個 transition 是 active 時顯示 pending state：

```
function App() {
  const [isPending, startTransition] = useTransition();
  const [count, setCount] = useState(0);

  function handleClick() {
    startTransition(() => {
      setCount(c => c + 1);
    })
  }

  return (
    <div>
      {isPending && <Spinner />}
      <button onClick={handleClick}>{count}</button>
    </div>
  );
}
```

注意：

Updates in a transition yield to more urgent updates such as clicks.

Updates in a transition will not show a fallback for re-suspended content. This allows the user to continue interacting with the current content while rendering the update.

useId {#useid}

```
const id = useId();
```

`useId` is a hook for generating unique IDs that are stable across the server and client, while avoiding hydration mismatches.

Note

`useId` is **not** for generating [keys](#) in a list. Keys should be generated from your data.

For a basic example, pass the `id` directly to the elements that need it:

```
function Checkbox() {
  const id = useId();
  return (
    <>
      <label htmlFor={id}>Do you like React?</label>
      <input id={id} type="checkbox" name="react"/>
    </>
  );
}
```

For multiple IDs in the same component, append a suffix using the same `id`:

```
function NameFields() {
  const id = useId();
  return (
    <div>
      <label htmlFor={id + '-firstName'}>First Name</label>
      <div>
        <input id={id + '-firstName'} type="text" />
      </div>
      <label htmlFor={id + '-lastName'}>Last Name</label>
      <div>
        <input id={id + '-lastName'} type="text" />
      </div>
    </div>
  );
}
```

注意：

`useId` generates a string that includes the `:` token. This helps ensure that the token is unique, but is not supported in CSS selectors or APIs like `querySelectorAll`.

`useId` supports an `identifierPrefix` to prevent collisions in multi-root apps. To configure, see the options for `hydrateRoot` and `ReactDOMServer`.

Library Hooks

The following Hooks are provided for library authors to integrate libraries deeply into the React model, and are not typically used in application code.

`useSyncExternalStore` [#usesyncexternalstore](#)

```
const state = useSyncExternalStore(subscribe, getSnapshot[, getServerSnapshot]);
```

`useSyncExternalStore` is a hook recommended for reading and subscribing from external data sources in a way that's compatible with concurrent rendering features like selective hydration and time slicing.

This method returns the value of the store and accepts three arguments:

- `subscribe` : function to register a callback that is called whenever the store changes.
- `getSnapshot` : function that returns the current value of the store.
- `getServerSnapshot` : function that returns the snapshot used during server rendering.

The most basic example simply subscribes to the entire store:

```
const state = useSyncExternalStore(store.subscribe, store.getSnapshot);
```

However, you can also subscribe to a specific field:

```
const selectedField = useSyncExternalStore(
  store.subscribe,
  () => store.getSnapshot().selectedField,
);
```

When server rendering, you must serialize the store value used on the server, and provide it to `useSyncExternalStore`. React will use this snapshot during hydration to prevent server mismatches:

```
const selectedField = useSyncExternalStore(
  store.subscribe,
  () => store.getSnapshot().selectedField,
  () => INITIAL_SERVER_SNAPSHOT.selectedField,
);
```

注意：

`getSnapshot` must return a cached value. If `getSnapshot` is called multiple times in a row, it must return the same exact value unless there was a store update in between.

A shim is provided for supporting multiple React versions published as `use-sync-external-store/shim`. This shim will prefer `useSyncExternalStore` when available, and fallback to a user-space implementation when it's not.

As a convenience, we also provide a version of the API with automatic support for memoizing the result of `getSnapshot` published as `use-sync-external-store/with-selector`.

useInsertionEffect {#useinsertioneffect}

```
useInsertionEffect(didUpdate);
```

The signature is identical to `useEffect`, but it fires synchronously *before* all DOM mutations. Use this to inject styles into the DOM before reading layout in `useLayoutEffect`. Since this hook is limited in scope, this hook does not have access to refs and cannot schedule updates.

注意：

`useInsertionEffect` should be limited to css-in-js library authors. Prefer `useEffect` or `useLayoutEffect` instead.

Hook 的規則

Hook 是 React 16.8 新加入的功能，它們讓你可以不用寫 class 就能使用 state 與其他 React 的功能。

Hook 是 JavaScript function，當你使用它們時需要遵守兩個規則。我們提供了一個 [linter plugin](#) 來自動化地實行這些規則：

只在最上層呼叫 Hook

不要在迴圈、條件式或是巢狀的 function 內呼叫 Hook。 相反的，在任何 early return 之前，請在 React function top level 使用 Hook。藉由遵循這些規則，你可以確保當每次一個 component render 時 Hook 都依照正確的順序被呼叫。正是這個使得 React 有辦法在多個 `useState` 和 `useEffect` 呼叫間，正確地保持 Hook 的 state。（如果你感到好奇，我們將在下方深入的解釋它。）

只在 React Function 中呼叫 Hook

別在一般的 JavaScript function 中呼叫 Hook。 相反的，你可以：

- 在 React function component 中呼叫 Hook。
- 在自定義的 Hook 中呼叫 Hook。（我們將會在下頁了解它們）。

透過遵循這些規則，你確保了在 component 中所有的 stateful 邏輯在其原始碼中可以清楚地被看見。

ESLint Plugin

我們發佈了一個 ESLint plugin 叫做 `eslint-plugin-react-hooks` 來強制施行這兩個規則。如果你想嘗試的話，可以將這個 plugin 加入到你的專案中：

Create React App 預設包含此 plugin。

```
npm install eslint-plugin-react-hooks --save-dev

// 你的 ESLint 配置
{
  "plugins": [
    // ...
    "react-hooks"
  ],
  "rules": {
    // ...
    "react-hooks/rules-of-hooks": "error", // 檢查 Hook 的規則
    "react-hooks/exhaustive-deps": "warn" // 檢查 effect 的相依性
  }
}
```

你現在可以先跳過，下頁將解釋如何打造**你自己的 Hook**。在這頁，我們將會繼續解釋這些規則背後的原因。

解說

如我們先前所學到的，我們可以在單一的 component 中使用多個 State 或 Effect Hook：

```

function Form() {
  // 1. 使用 name state 變數
  const [name, setName] = useState('Mary');

  // 2. 使用一個 effect 來保存表單
  useEffect(function persistForm() {
    localStorage.setItem('formData', name);
  });

  // 3. 使用 surname state 變數
  const [surname, setSurname] = useState('Poppins');

  // 4. 使用一個 effect 來更新標題
  useEffect(function updateTitle() {
    document.title = name + ' ' + surname;
  });

  // ...
}

```

所以 React 是如何知道哪個 state 要對應到哪個 `useState` 的呼叫？答案是 **React 仰賴於 Hook 被呼叫的順序**。我們的範例能執行是因為在每一次的 render 中 Hook 都是依照一樣的順序被呼叫：

```

// -----
// 第一次 render
// -----
useState('Mary')          // 1. 用 'Mary' 來初始化 name state 變數
useEffect(persistForm)    // 2. 增加一個 effect 來保存表單
useState('Poppins')       // 3. 用 'Poppins' 來初始化 surname state 變數
useEffect(updateTitle)    // 4. 增加一個 effect 來更新標題

// -----
// 第二次 render
// -----
useState('Mary')          // 1. 讀取 name state 變數 (參數被忽略了)
useEffect(persistForm)    // 2. 替換了用來保存表單的 effect
useState('Poppins')       // 3. 讀取 surname state 變數 (參數被忽略了)
useEffect(updateTitle)    // 4. 替換了用來更新標題的 effect

// ...

```

只要 Hook 在 render 時被呼叫的順序是一致的，React 可以將一些 local state 和它們一一聯繫在一起。但如果我們把一個 Hook 呼叫（例如，`persistForm` effect）放在條件式中會發生什麼事呢？

```

// ● 我們違反了第一個規則，在條件式中使用 Hook
if (name !== '') {
  useEffect(function persistForm() {
    localStorage.setItem('formData', name);
  });
}

```

這個 `name !== ''` 條件式在初次 render 時為 `true`，所以我們執行了此 Hook。然而，在下一次 render 時使用者可能清除了表單，使得條件式變為 `false`。而現在我們在 render 期間跳過了這一個 Hook，Hook 的呼叫順序有所不同：

```

useState('Mary')          // 1. 讀取 name state 變數 (參數被忽略了)
// useEffect(persistForm) // ● 這個 Hook 被跳過了！
useState('Poppins')       // ● 2 (但之前是 3). 未能讀取 surname state 變數
useEffect(updateTitle)    // ● 3 (但之前是 4). 未能取代 effect

```

React 不會知道第二個 `useState` Hook 呼叫回傳什麼。React 預期在這個 component 中的第二個 Hook 呼叫和 `persistForm` effect 是相對應的，就如同在前一次的 render 一樣，但它不再一樣了。從那時起，在我們跳過的那個 Hook 後面，每下一個 Hook 呼叫都會 shift 一個，導致 bug 的發生。

這就是為何必須在我們的 **component** 之上層來呼叫 **Hook**。如果我們想要有條件地執行 effect，我們可以把那個條件式放在我們的 Hook 裡：

```
useEffect(function persistForm() {
  // 👍 我們不再違反第一個規則
  if (name !== '') {
    localStorage.setItem('formData', name);
  }
});
```

注意你不需要擔心這個問題，如果你使用提供的 **lint 規則**。但現在你也了解為何 Hook 是這樣運作的，和這些用來避免而制定的規則。

下一步

最後，我們準備好學習撰寫你自己的 Hook！自定義的 Hook 讓你能結合由 React 提供的 Hook 到你自己的抽象化中，而且在不同的 component 間重複使用相同的 stateful 邏輯。

[Go to TOC](#)

使用 State Hook

Hook 是 React 16.8 中增加的新功能。它讓你不必寫 class 就能使用 state 以及其他 React 的功能。

介紹 *Hook* 使用過這個範例讓我們熟悉 *Hook*：

```
import React, { useState } from 'react';

function Example() {
  // 告知一個新的 state 變數，我們稱作為「count」。
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

我們會藉由在同樣的範例中使用 class 時有什麼差別來學習 *Hook*。

相等的 Class 範例

如果你以前在 React 中使用過 class，這段程式應該對你不陌生：

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }

  render() {
    return (
      <div>
        <p>You clicked {this.state.count} times</p>
        <button onClick={() => this.setState({ count: this.state.count + 1 })}>
          Click me
        </button>
      </div>
    );
  }
}
```

State 從 `{ count: 0 }` 開始，當使用者點擊按鈕時，我們藉由呼叫 `this.setState()` 增加 `state.count`。在這一整頁，我們會使用這個 class 中的片段來說明。

注意

你也許會好奇我們為什麼要用一個計數器，而不是一個更實際的例子。其實當我們還在學習怎麼使用 Hook 的時候，這有助於我們專注在 API 上面。

Hook 和 Function Component

提醒一下，在 React 中 function component 看起來像這樣：

```
const Example = (props) => {
  // 你可以在這裡使用 Hook !
  return <div />;
}
```

或是：

```
function Example(props) {
  // 你可以在這裡使用 Hook !
  return <div />;
}
```

也許你知道它會被稱作「stateless component」。但因為現在這些 component 也可以使用 React state 了，我們會比較喜歡「function component」這個稱呼。

Hook 不會在 class 裡運作。但你可以使用它來取代 class。

什麼是 Hook？

就從 React 引入 `useState` 來開始新的範例吧：

```
import React, { useState } from 'react';
function Example() {
  // ...
}
```

什麼是 Hook？ Hook 是一個讓你可以使用 React 各項功能的特殊 function。舉例來說，`useState` 是一個讓你增加 React state 到 function component 的 Hook。在後面的章節我們會學習其他 Hook。

什麼時候該使用 Hook？ 以前當你寫一個 function component 需要增加一些 state 時，你必須轉換成 class。現在你可以直接在 function component 中使用 Hook。我們現在就來試試看吧！

注意

關於在 component 中哪裡可以使用 Hook，有一些特別的規則。我們會在 [Hook 的規則](#) 中學習。

宣告一個 State 變數

在 class 中，藉由在 constructor 設定 `this.state` 成 `{ count: 0 }` 把 `count` 這個 state 起始值設為 0。

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }
}
```

在 function component 中，我們沒有 `this`，所以我們沒辦法指定或讀取 `this.state`。相反地，我們可以直接在 component 中呼叫 `useState` Hook。

```
import React, { useState } from 'react';

function Example() {
  // 宣告一個新的 state 變數，我們稱作為「count」。
  const [count, setCount] = useState(0);
```

呼叫 `useState` 做了什麼？它宣告了一個「state 變數」。我們的變數叫做 `count`，也可以有其他稱呼，像是 `banana`。這是一個在 function 呼叫中「保留」變數的方法—`useState` 就像是 class 中 `this.state` 的功能一樣。一般情況下，變數會在 function 結束時「消失」，但 state 變數會被 React 保留起來。

我們傳入什麼參數給 `useState`？唯一需要傳入 `useState()` Hook 的參數就是 state 的起始值。不像 class，這個 state 不需要是一個 object。我們可以使用我們所需的 number 或是 string。在範例中，我們只需要一個 number 記錄使用者點擊了多少次，所以我們傳入 `0` 來當作起始值。（如果要在 state 儲存兩個不同的值，可以呼叫 `useState()` 兩次。）

`useState` 回傳了什麼？它回傳了一對值：目前的 state 跟一個可以更新 state 的 function。這就是為什麼我們寫 `const [count, setCount] = useState()` 的原因。除了你是直接拿到一對，這跟 class 中的 `this.state.count` 和 `this.setState` 很類似。如果你對我們用的語法不熟悉，我們稍後會在[本頁的底部](#)討論它。

現在我們知道 `useState` Hook 做了什麼，我們的範例會看起來更有感覺：

```
import React, { useState } from 'react';

function Example() {
  // 宣告一個新的 state 變數，我們稱作為「count」。
  const [count, setCount] = useState(0);
```

我們宣告了一個叫做 `count` 的 state 變數，並將起始值設成了 `0`。React 在 re-render 間會記住目前的值，並將它提供給我們的 function。如果我們需要更新目前的 `count`，我們可以呼叫 `setCount`。

注意

你也許會好奇：為什麼 `useState` 不叫做 `createState`？

`state` 只會在第一次 render 時被建立，所以「Create」並不會比較精準。而在其後的 render，`useState` 只會給我們目前的 `state`。不然這就不叫做 `state` 啦！另外還有一個理由，為什麼 Hook 總是以 `use` 起頭。我們稍後會在[Hook 規則](#)中了解。

讀取 State

在 class 中我們要顯示目前的計數，我們使用 `this.state.count`：

```
<p>You clicked {this.state.count} times</p>
```

在 function 中，我們可以直接使用 `count`：

```
<p>You clicked {count} times</p>
```

更新 State

在 class 中，我們需要呼叫 `this.setState()` 來更新 `count` state：

```
<button onClick={() => this.setState({ count: this.state.count + 1 })}>
  Click me
</button>
```

在 function 中，我們已經有 `setCount` 和 `count` 變數，所以我們不需要 `this`：

```
<button onClick={() => setCount(count + 1)}>
  Click me
</button>
```

總結

我們來一行一行總結一下我們學了什麼，並確認我們都懂了。

```
1: import React, { useState } from 'react';
2:
3: function Example() {
4:   const [count, setCount] = useState(0);
5:
6:   return (
7:     <div>
8:       <p>You clicked {count} times</p>
9:       <button onClick={() => setCount(count + 1)}>
10:         Click me
11:       </button>
12:     </div>
13:   );
14: }
```

- 第一行：**我們從 React 引入 `useState` Hook。它讓我們可以在 function component 保留 local state。
- 第四行：**在 `Example` component 裡，我們呼叫 `useState` Hook 宣告了一個新的 state 變數。並回傳了一對由我們命名的值。我們將代表點擊數的變數命名為 `count`。我們將起始值設為 `0` 並傳入 `useState` 當作唯一參數。第二個回傳的值是個可以更新 `count` 的 function，所以我們命名為 `setCount`。
- 第九行：**當使用者點擊，我們就呼叫 `setCount` 並傳入新的值。然後 React 就會 re-render `Example` component，並傳入新的 `count` 值。

別急！也許你需要好好消化一番。如果還是有些不懂，你可以試著再將上面那段程式從頭到尾看一下。我們保證一旦你試著把 class 的 state 運作方式「遺忘」，這一切會合理起來。

提示：方括號代表什麼？

你也許注意到了，當我們宣告 state 變數時使用了方括號：

```
const [count, setCount] = useState(0);
```

等號左邊的並不是 React 的 API。你可以命名你自己的 state 變數：

```
const [fruit, setFruit] = useState('banana');
```

這個 JavaScript 語法叫做陣列解構賦值。這代表我們宣告了兩個新的變數 `fruit` 和 `setFruit`，`fruit` 被設為 `useState` 回傳的第一個值，`setFruit` 則是第二個。跟下面這段程式相同：

```
var fruitStateVariable = useState('banana'); // 回傳一對值
var fruit = fruitStateVariable[0]; // 第一個值
var setFruit = fruitStateVariable[1]; // 第二個值
```

當我們使用 `useState` 宣告 state 變數，他會回傳一對在 array 裡的值。第一個值是目前 state 的值，第二個是一個可以更新 state 的 function。因為它們有特殊的意義，只用 `[0]` 和 `[1]` 來存取它們的話會令人困惑。所以我們使用陣列解構賦值來命名它們。

注意

你也許會好奇既然我們沒有傳入任何東西像是 `this`，React 要怎麼知道哪個 component 對應到哪個 `useState`。我們會在 FAQ 章節解答[這個問題](#)和其他問題。

提示：使用多個 State 變數

把 state 變數宣告成一對 `[something, setSomething]` 同時也很便利，因為如果想要使用超過一個 state 變數，這能讓我們對不同的 state 變數有不同的命名

```
function ExampleWithManyStates() {
  // 宣告多個 state 變數！
  const [age, setAge] = useState(42);
  const [fruit, setFruit] = useState('banana');
  const [todos, setTodos] = useState([{ text: 'Learn Hooks' }]);
}
```

在上面的 component 中，我們的 local 變數有 `age`、`fruit` 和 `todos`，而且我們可以單獨更新它們。

```
function handleOrangeClick() {
  // 類似於 this.setState({ fruit: 'orange' })
  setFruit('orange');
}
```

你不需要使用很多 state 變數。state 變數可以是 object 或是 array，所以你可以把相關的資料放在一起。然而，不像 class 裡的 `this.setState` 會合併原本的 state，這裡更新 state 變數會直接取代。

我們在 [FAQ 章節](#) 提提供更多拆分獨立的 state 變數的建議。

下一步

在這一頁，我們學習了其中一個 React 提供的 Hook—`useState`。我們有時也稱它做「State Hook」。前所未有的，它讓我們能夠增加 local state 到 React function component！

我們同時也瞭解了更多什麼是 Hook。Hook 是可以讓你在 function component 使用 React 各項功能的 function。還有更多 Hook 我們還沒認識，而 Hook 的名字都會以 `use` 起頭。

現在我們繼續學習下個 Hook : `useEffect`。 就像 class 的生命週期方法，它讓你能夠在 component 使用 side effect。

[Go to TOC](#)

How to Contribute

React is one of Facebook's first open source projects that is both under very active development and is also being used to ship code to everybody on [facebook.com](#). We're still working out the kinks to make contributing to this project as easy and transparent as possible, but we're not quite there yet. Hopefully this document makes the process for contributing clear and answers some questions that you may have.

Code of Conduct {#code-of-conduct}

Facebook has adopted the [Contributor Covenant](#) as its Code of Conduct, and we expect project participants to adhere to it. Please read [the full text](#) so that you can understand what actions will and will not be tolerated.

Open Development

All work on React happens directly on [GitHub](#). Both core team members and external contributors send pull requests which go through the same review process.

Semantic Versioning

React follows [semantic versioning](#). We release patch versions for critical bugfixes, minor versions for new features or non-essential changes, and major versions for any breaking changes. When we make breaking changes, we also introduce deprecation warnings in a minor version so that our users learn about the upcoming changes and migrate their code in advance. Learn more about our commitment to stability and incremental migration in [our versioning policy](#).

Every significant change is documented in the [changelog file](#).

Branch Organization

Submit all changes directly to the `main branch`. We don't use separate branches for development or for upcoming releases. We do our best to keep `main` in good shape, with all tests passing.

Code that lands in `main` must be compatible with the latest stable release. It may contain additional features, but no breaking changes. We should be able to release a new minor version from the tip of `main` at any time.

Feature Flags

To keep the `main` branch in a releasable state, breaking changes and experimental features must be gated behind a feature flag.

Feature flags are defined in `packages/shared/ReactFeatureFlags.js`. Some builds of React may enable different sets of feature flags; for example, the React Native build may be configured differently than React DOM. These flags are found in `packages/shared/forks`. Feature flags are statically typed by Flow, so you can run `yarn flow` to confirm that you've updated all the necessary files.

React's build system will strip out disabled feature branches before publishing. A continuous integration job runs on every commit to check for changes in bundle size. You can use the change in size as a signal that a feature was gated correctly.

Bugs

Where to Find Known Issues

We are using [GitHub Issues](#) for our public bugs. We keep a close eye on this and try to make it clear when we have an internal fix in progress. Before filing a new task, try to make sure your problem doesn't already exist.

Reporting New Issues

The best way to get your bug fixed is to provide a reduced test case. This [JSFiddle template](#) is a great starting point.

Security Bugs

Facebook has a [bounty program](#) for the safe disclosure of security bugs. With that in mind, please do not file public issues; go through the process outlined on that page.

How to Get in Touch

- IRC: [#reactjs on freenode](#)
- [Discussion forums](#)

There is also [an active community of React users on the Discord chat platform](#) in case you need help with React.

Proposing a Change

If you intend to change the public API, or make any non-trivial changes to the implementation, we recommend [filing an issue](#). This lets us reach an agreement on your proposal before you put significant effort into it.

If you're only fixing a bug, it's fine to submit a pull request right away but we still recommend to file an issue detailing what you're fixing. This is helpful in case we don't accept that specific fix but want to keep track of the issue.

Your First Pull Request

Working on your first Pull Request? You can learn how from this free video series:

[How to Contribute to an Open Source Project on GitHub](#)

To help you get your feet wet and get you familiar with our contribution process, we have a list of [good first issues](#) that contain bugs that have a relatively limited scope. This is a great place to get started.

If you decide to fix an issue, please be sure to check the comment thread in case somebody is already working on a fix. If nobody is working on it at the moment, please leave a comment stating that you intend to work on it so other people don't accidentally duplicate your effort.

If somebody claims an issue but doesn't follow up for more than two weeks, it's fine to take it over but you should still leave a comment.

Sending a Pull Request

The core team is monitoring for pull requests. We will review your pull request and either merge it, request changes to it, or close it with an explanation. For API changes we may need to fix our internal uses at Facebook.com, which could cause some delay. We'll do our best to provide updates and feedback throughout the process.

Before submitting a pull request, please make sure the following is done:

1. Fork [the repository](#) and create your branch from `main`.
2. Run `yarn` in the repository root.
3. If you've fixed a bug or added code that should be tested, add tests!
4. Ensure the test suite passes (`yarn test`). Tip: `yarn test --watch TestName` is helpful in development.
5. Run `yarn test --prod` to test in the production environment.
6. If you need a debugger, run `yarn debug-test --watch TestName`, open `chrome://inspect`, and press "Inspect".
7. Format your code with `prettier` (`yarn prettier`).
8. Make sure your code lints (`yarn lint`). Tip: `yarn linc` to only check changed files.
9. Run the `Flow` typechecks (`yarn flow`).
10. If you haven't already, complete the CLA.

Contributor License Agreement (CLA)

In order to accept your pull request, we need you to submit a CLA. You only need to do this once, so if you've done this for another Facebook open source project, you're good to go. If you are submitting a pull request for the first time, just let us know that you have completed the CLA and we can cross-check with your GitHub username.

[Complete your CLA here.](#)

Contribution Prerequisites

- You have [Node](#) installed at LTS and [Yarn](#) at v1.2.0+.
- You have [JDK](#) installed.
- You have `gcc` installed or are comfortable installing a compiler if needed. Some of our dependencies may require a compilation step. On OS X, the Xcode Command Line Tools will cover this. On Ubuntu, `apt-get install build-essential` will install the required packages. Similar commands should work on other Linux distros. Windows will require some additional steps, see the [node-gyp installation instructions](#) for details.

- You are familiar with Git.

Development Workflow

After cloning React, run `yarn` to fetch its dependencies. Then, you can run several commands:

- `yarn lint` checks the code style.
- `yarn linc` is like `yarn lint` but faster because it only checks files that differ in your branch.
- `yarn test` runs the complete test suite.
- `yarn test --watch` runs an interactive test watcher.
- `yarn test --prod` runs tests in the production environment.
- `yarn test <pattern>` runs tests with matching filenames.
- `yarn debug-test` is just like `yarn test` but with a debugger. Open `chrome://inspect` and press "Inspect".
- `yarn flow` runs the Flow typechecks.
- `yarn build` creates a `build` folder with all the packages.
- `yarn build react/index,react-dom/index --type=UMD` creates UMD builds of just React and ReactDOM.

We recommend running `yarn test` (or its variations above) to make sure you don't introduce any regressions as you work on your change. However, it can be handy to try your build of React in a real project.

First, run `yarn build`. This will produce pre-built bundles in `build` folder, as well as prepare npm packages inside `build/packages`.

The easiest way to try your changes is to run `yarn build react/index,react-dom/index --type=UMD` and then open `fixtures/packaging/babel-standalone/dev.html`. This file already uses `react.development.js` from the `build` folder so it will pick up your changes.

If you want to try your changes in your existing React project, you may copy `build/node_modules/react/umd/react.development.js`, `build/node_modules/react-dom/umd/react-dom.development.js`, or any other build products into your app and use them instead of the stable version.

If your project uses React from npm, you may delete `react` and `react-dom` in its dependencies and use `yarn link` to point them to your local `build` folder. Note that **instead of `--type=UMD` you'll want to pass `--type=NODE` when building**. You'll also need to build the `scheduler` package:

```
cd ~/path_to_your_react_clone/
yarn build react/index,react/jsx,react-dom/index,scheduler --type=NODE

cd build/node_modules/react
yarn link
cd build/node_modules/react-dom
yarn link

cd ~/path/to/your/project
yarn link react react-dom
```

Every time you run `yarn build` in the React folder, the updated versions will appear in your project's `node_modules`. You can then rebuild your project to try your changes.

If some package is still missing (e.g. maybe you use `react-dom/server` in your project), you can always do a full build with `yarn build`. Note that running `yarn build` without options takes a long time.

We still require that your pull request contains unit tests for any new functionality. This way we can ensure that we don't break your code in the future.

Style Guide

We use an automatic code formatter called [Prettier](#). Run `yarn prettier` after making any changes to the code.

Then, our linter will catch most issues that may exist in your code. You can check the status of your code styling by simply running `yarn linc`.

However, there are still some styles that the linter cannot pick up. If you are unsure about something, looking at [Airbnb's Style Guide](#) will guide you in the right direction.

Request for Comments (RFC)

Many changes, including bug fixes and documentation improvements can be implemented and reviewed via the normal GitHub pull request workflow.

Some changes though are "substantial", and we ask that these be put through a bit of a design process and produce a consensus among the React core team.

The "RFC" (request for comments) process is intended to provide a consistent and controlled path for new features to enter the project. You can contribute by visiting the [rfcs repository](#).

License

By contributing to React, you agree that your contributions will be licensed under its MIT license.

What Next?

Read the [next section](#) to learn how the codebase is organized.

Implementation Notes

This section is a collection of implementation notes for the [stack reconciler](#).

It is very technical and assumes a strong understanding of React public API as well as how it's divided into core, renderers, and the reconciler. If you're not very familiar with the React codebase, read [the codebase overview](#) first.

It also assumes an understanding of the [differences between React components, their instances, and elements](#).

The stack reconciler was used in React 15 and earlier. It is located at [src/renderers/shared/stack/reconciler](#).

Video: Building React from Scratch

[Paul O'Shannessy](#) gave a talk about [building React from scratch](#) that largely inspired this document.

Both this document and his talk are simplifications of the real codebase so you might get a better understanding by getting familiar with both of them.

Overview

The reconciler itself doesn't have a public API. [Renderers](#) like React DOM and React Native use it to efficiently update the user interface according to the React components written by the user.

Mounting as a Recursive Process

Let's consider the first time you mount a component:

```
const root = ReactDOM.createRoot(rootEl);
root.render(<App />);
```

`root.render` will pass `<App />` along to the reconciler. Remember that `<App />` is a React element, that is, a description of *what* to render. You can think about it as a plain object:

```
console.log(<App />);
// { type: App, props: {} }
```

The reconciler will check if `App` is a class or a function.

If `App` is a function, the reconciler will call `App(props)` to get the rendered element.

If `App` is a class, the reconciler will instantiate an `App` with `new App(props)`, call the `componentWillMount()` lifecycle method, and then will call the `render()` method to get the rendered element.

Either way, the reconciler will learn the element `App` "rendered to".

This process is recursive. `App` may render to a `<Greeting />`, `Greeting` may render to a `<Button />`, and so on. The reconciler will "drill down" through user-defined components recursively as it learns what each component renders to.

You can imagine this process as a pseudocode:

```

function isClass(type) {
  // React.Component subclasses have this flag
  return (
    Boolean(type.prototype) &&
    Boolean(type.prototype.isReactComponent)
  );
}

// This function takes a React element (e.g. <App />)
// and returns a DOM or Native node representing the mounted tree.
function mount(element) {
  var type = element.type;
  var props = element.props;

  // We will determine the rendered element
  // by either running the type as function
  // or creating an instance and calling render().
  var renderedElement;
  if (isClass(type)) {
    // Component class
    var publicInstance = new type(props);
    // Set the props
    publicInstance.props = props;
    // Call the lifecycle if necessary
    if (publicInstance.componentWillMount) {
      publicInstance.componentWillMount();
    }
    // Get the rendered element by calling render()
    renderedElement = publicInstance.render();
  } else {
    // Component function
    renderedElement = type(props);
  }

  // This process is recursive because a component may
  // return an element with a type of another component.
  return mount(renderedElement);
}

// Note: this implementation is incomplete and recurses infinitely!
// It only handles elements like <App /> or <Button />.
// It doesn't handle elements like <div /> or <p /> yet.
}

var rootEl = document.getElementById('root');
var node = mount(<App />);
rootEl.appendChild(node);

```

Note:

This really *is* a pseudo-code. It isn't similar to the real implementation. It will also cause a stack overflow because we haven't discussed when to stop the recursion.

Let's recap a few key ideas in the example above:

- React elements are plain objects representing the component type (e.g. `App`) and the props.
- User-defined components (e.g. `App`) can be classes or functions but they all "render to" elements.
- "Mounting" is a recursive process that creates a DOM or Native tree given the top-level React element (e.g. `<App />`).

Mounting Host Elements

This process would be useless if we didn't render something to the screen as a result.

In addition to user-defined ("composite") components, React elements may also represent platform-specific ("host") components. For example, `Button` might return a `<div />` from its render method.

If element's `type` property is a string, we are dealing with a host element:

```
console.log(<div />);
// { type: 'div', props: {} }
```

There is no user-defined code associated with host elements.

When the reconciler encounters a host element, it lets the renderer take care of mounting it. For example, React DOM would create a DOM node.

If the host element has children, the reconciler recursively mounts them following the same algorithm as above. It doesn't matter whether children are host (like `<div><hr /></div>`), composite (like `<div><Button /></div>`), or both.

The DOM nodes produced by the child components will be appended to the parent DOM node, and recursively, the complete DOM structure will be assembled.

Note:

The reconciler itself is not tied to the DOM. The exact result of mounting (sometimes called "mount image" in the source code) depends on the renderer, and can be a DOM node (React DOM), a string (React DOM Server), or a number representing a native view (React Native).

If we were to extend the code to handle host elements, it would look like this:

```
function isClass(type) {
  // React.Component subclasses have this flag
  return (
    Boolean(type.prototype) &&
    Boolean(type.prototype.isReactComponent)
  );
}

// This function only handles elements with a composite type.
// For example, it handles <App /> and <Button />, but not a <div />.
function mountComposite(element) {
```

```

var type = element.type;
var props = element.props;

var renderedElement;
if (isClass(type)) {
  // Component class
  var publicInstance = new type(props);
  // Set the props
  publicInstance.props = props;
  // Call the lifecycle if necessary
  if (publicInstance.componentWillMount) {
    publicInstance.componentWillMount();
  }
  renderedElement = publicInstance.render();
} else if (typeof type === 'function') {
  // Component function
  renderedElement = type(props);
}

// This is recursive but we'll eventually reach the bottom of recursion when
// the element is host (e.g. <div />) rather than composite (e.g. <App />):
return mount(renderedElement);
}

// This function only handles elements with a host type.
// For example, it handles <div /> and <p /> but not an <App />.
function mountHost(element) {
  var type = element.type;
  var props = element.props;
  var children = props.children || [];
  if (!Array.isArray(children)) {
    children = [children];
  }
  children = children.filter(Boolean);

  // This block of code shouldn't be in the reconciler.
  // Different renderers might initialize nodes differently.
  // For example, React Native would create iOS or Android views.
  var node = document.createElement(type);
  Object.keys(props).forEach(propName => {
    if (propName !== 'children') {
      node.setAttribute(propName, props[propName]);
    }
  });

  // Mount the children
  children.forEach(childElement => {
    // Children may be host (e.g. <div />) or composite (e.g. <Button />).
    // We will also mount them recursively:
    var childNode = mount(childElement);

    // This line of code is also renderer-specific.
    // It would be different depending on the renderer:
    node.appendChild(childNode);
  });

  // Return the DOM node as mount result.
  // This is where the recursion ends.
  return node;
}

function mount(element) {

```

```

var type = element.type;
if (typeof type === 'function') {
  // User-defined components
  return mountComposite(element);
} else if (typeof type === 'string') {
  // Platform-specific components
  return mountHost(element);
}
}

var rootEl = document.getElementById('root');
var node = mount(<App />);
rootEl.appendChild(node);

```

This is working but still far from how the reconciler is really implemented. The key missing ingredient is support for updates.

Introducing Internal Instances

The key feature of React is that you can re-render everything, and it won't recreate the DOM or reset the state:

```

root.render(<App />);
// Should reuse the existing DOM:
root.render(<App />);

```

However, our implementation above only knows how to mount the initial tree. It can't perform updates on it because it doesn't store all the necessary information, such as all the `publicInstances`, or which DOM `node`s correspond to which components.

The stack reconciler codebase solves this by making the `mount()` function a method and putting it on a class. There are drawbacks to this approach, and we are going in the opposite direction in the [ongoing rewrite of the reconciler](#). Nevertheless this is how it works now.

Instead of separate `mountHost` and `mountComposite` functions, we will create two classes: `DOMComponent` and `CompositeComponent`.

Both classes have a constructor accepting the `element`, as well as a `mount()` method returning the mounted node. We will replace a top-level `mount()` function with a factory that instantiates the correct class:

```

function instantiateComponent(element) {
  var type = element.type;
  if (typeof type === 'function') {
    // User-defined components
    return new CompositeComponent(element);
  } else if (typeof type === 'string') {
    // Platform-specific components
    return new DOMComponent(element);
  }
}

```

First, let's consider the implementation of `CompositeComponent`:

```

class CompositeComponent {
  constructor(element) {
    this.currentElement = element;
    this.renderedComponent = null;
    this.publicInstance = null;
  }

  getPublicInstance() {
    // For composite components, expose the class instance.
    return this.publicInstance;
  }

  mount() {
    var element = this.currentElement;
    var type = element.type;
    var props = element.props;

    var publicInstance;
    var renderedElement;
    if (isClass(type)) {
      // Component class
      publicInstance = new type(props);
      // Set the props
      publicInstance.props = props;
      // Call the lifecycle if necessary
      if (publicInstance.componentWillMount) {
        publicInstance.componentWillMount();
      }
      renderedElement = publicInstance.render();
    } else if (typeof type === 'function') {
      // Component function
      publicInstance = null;
      renderedElement = type(props);
    }

    // Save the public instance
    this.publicInstance = publicInstance;

    // Instantiate the child internal instance according to the element.
    // It would be a DOMComponent for <div /> or <p />,
    // and a CompositeComponent for <App /> or <Button />:
    var renderedComponent = instantiateComponent(renderedElement);
    this.renderedComponent = renderedComponent;

    // Mount the rendered output
    return renderedComponent.mount();
  }
}

```

This is not much different from our previous `mountComposite()` implementation, but now we can save some information, such as `this.currentElement`, `this.renderedComponent`, and `this.publicInstance`, for use during updates.

Note that an instance of `CompositeComponent` is not the same thing as an instance of the user-supplied `element.type`. `CompositeComponent` is an implementation detail of our reconciler, and is never exposed to the user. The user-defined class is the one we read from `element.type`, and `CompositeComponent` creates an instance of it.

To avoid the confusion, we will call instances of `CompositeComponent` and `DOMComponent` "internal instances". They exist so we can associate some long-lived data with them. Only the renderer and the reconciler are aware that they exist.

In contrast, we call an instance of the user-defined class a "public instance". The public instance is what you see as `this` in the `render()` and other methods of your custom components.

The `mountHost()` function, refactored to be a `mount()` method on `DOMComponent` class, also looks familiar:

```
class DOMComponent {
  constructor(element) {
    this.currentElement = element;
    this.renderedChildren = [];
    this.node = null;
  }

  getPublicInstance() {
    // For DOM components, only expose the DOM node.
    return this.node;
  }

  mount() {
    var element = this.currentElement;
    var type = element.type;
    var props = element.props;
    var children = props.children || [];
    if (!Array.isArray(children)) {
      children = [children];
    }

    // Create and save the node
    var node = document.createElement(type);
    this.node = node;

    // Set the attributes
    Object.keys(props).forEach(propName => {
      if (propName !== 'children') {
        node.setAttribute(propName, props[propName]);
      }
    });

    // Create and save the contained children.
    // Each of them can be a DOMComponent or a CompositeComponent,
    // depending on whether the element type is a string or a function.
    var renderedChildren = children.map(instantiateComponent);
    this.renderedChildren = renderedChildren;

    // Collect DOM nodes they return on mount
    var childNodes = renderedChildren.map(child => child.mount());
    childNodes.forEach(childNode => node.appendChild(childNode));

    // Return the DOM node as mount result
    return node;
  }
}
```

The main difference after refactoring from `mountHost()` is that we now keep `this.node` and `this.renderedChildren` associated with the internal DOM component instance. We will also use them for applying non-destructive updates in the future.

As a result, each internal instance, composite or host, now points to its child internal instances. To help visualize this, if a function `<App>` component renders a `<Button>` class component, and `Button` class renders a `<div>`, the internal instance tree would look like this:

```
[object CompositeComponent] {
  currentElement: <App />,
  publicInstance: null,
  renderedComponent: [object CompositeComponent] {
    currentElement: <Button />,
    publicInstance: [object Button],
    renderedComponent: [object DOMComponent] {
      currentElement: <div />,
      node: [object HTMLDivElement],
      renderedChildren: []
    }
  }
}
```

In the DOM you would only see the `<div>`. However the internal instance tree contains both composite and host internal instances.

The composite internal instances need to store:

- The current element.
- The public instance if element type is a class.
- The single rendered internal instance. It can be either a `DOMComponent` or a `CompositeComponent`.

The host internal instances need to store:

- The current element.
- The DOM node.
- All the child internal instances. Each of them can be either a `DOMComponent` or a `CompositeComponent`.

If you're struggling to imagine how an internal instance tree is structured in more complex applications, [React DevTools](#) can give you a close approximation, as it highlights host instances with grey, and composite instances with purple:

```

<TodoApp>
  ▼<div>
    <h3>TODO</h3>
    ▼<TodoList items=[{...}]>
      ▼<ul>
        <li key="1475260226151">Buy milk</li>
      </ul>
    </TodoList>
    ▼<form onSubmit=bound handleSubmit()>
      <input onChange=bound onChange() value="" />
      <button>Add #2</button>
    </form>
  </div>
</TodoApp>

```

To complete this refactoring, we will introduce a function that mounts a complete tree into a container node and a public instance:

```

function mountTree(element, containerNode) {
  // Create the top-level internal instance
  var rootComponent = instantiateComponent(element);

  // Mount the top-level component into the container
  var node = rootComponent.mount();
  containerNode.appendChild(node);

  // Return the public instance it provides
  var publicInstance = rootComponent.getPublicInstance();
  return publicInstance;
}

var rootEl = document.getElementById('root');
mountTree(<App />, rootEl);

```

Unmounting

Now that we have internal instances that hold onto their children and the DOM nodes, we can implement unmounting. For a composite component, unmounting calls a lifecycle method and recurses.

```

class CompositeComponent {
  // ...

  unmount() {
    // Call the lifecycle method if necessary
    var publicInstance = this.publicInstance;
    if (publicInstance) {
      if (publicInstance.componentWillUnmount) {
        publicInstance.componentWillUnmount();
      }
    }
  }
}

```

```
//Unmount the single rendered component
var renderedComponent = this.renderedComponent;
renderedComponent.unmount();
}
```

For `DOMComponent`, unmounting tells each child to unmount:

```
class DOMComponent {
  // ...
  unmount() {
    //Unmount all the children
    var renderedChildren = this.renderedChildren;
    renderedChildren.forEach(child => child.unmount());
  }
}
```

In practice, unmounting DOM components also removes the event listeners and clears some caches, but we will skip those details.

We can now add a new top-level function called `unmountTree(containerNode)` that is similar to `ReactDOM.unmountComponentAtNode()`:

```
function unmountTree(containerNode) {
  // Read the internal instance from a DOM node:
  // (This doesn't work yet, we will need to change mountTree() to store it.)
  var node = containerNode.firstChild;
  var rootComponent = node._internalInstance;

  //Unmount the tree and clear the container
  rootComponent.unmount();
  containerNode.innerHTML = '';
}
```

In order for this to work, we need to read an internal root instance from a DOM node. We will modify `mountTree()` to add the `_internalInstance` property to the root DOM node. We will also teach `mountTree()` to destroy any existing tree so it can be called multiple times:

```
function mountTree(element, containerNode) {
  //Destroy any existing tree
  if (containerNode.firstChild) {
    unmountTree(containerNode);
  }

  //Create the top-level internal instance
  var rootComponent = instantiateComponent(element);

  //Mount the top-level component into the container
  var node = rootComponent.mount();
  containerNode.appendChild(node);

  //Save a reference to the internal instance
  node._internalInstance = rootComponent;

  //Return the public instance it provides
}
```

```

    var publicInstance = rootComponent.getPublicInstance();
    return publicInstance;
}

```

Now, running `unmountTree()`, or running `mountTree()` repeatedly, removes the old tree and runs the `componentWillUnmount()` lifecycle method on components.

Updating

In the previous section, we implemented unmounting. However React wouldn't be very useful if each prop change unmounted and mounted the whole tree. The goal of the reconciler is to reuse existing instances where possible to preserve the DOM and the state:

```

var rootEl = document.getElementById('root');

mountTree(<App />, rootEl);
// Should reuse the existing DOM:
mountTree(<App />, rootEl);

```

We will extend our internal instance contract with one more method. In addition to `mount()` and `unmount()`, both `DOMComponent` and `CompositeComponent` will implement a new method called `receive(nextElement)`:

```

class CompositeComponent {
  // ...

  receive(nextElement) {
    // ...
  }
}

class DOMComponent {
  // ...

  receive(nextElement) {
    // ...
  }
}

```

Its job is to do whatever is necessary to bring the component (and any of its children) up to date with the description provided by the `nextElement`.

This is the part that is often described as "virtual DOM diffing" although what really happens is that we walk the internal tree recursively and let each internal instance receive an update.

Updating Composite Components

When a composite component receives a new element, we run the `componentWillUpdate()` lifecycle method.

Then we re-render the component with the new props, and get the next rendered element:

```

class CompositeComponent {
  // ...

  receive(nextElement) {
    var prevProps = this.currentElement.props;
    var publicInstance = this.publicInstance;
    var prevRenderedComponent = this.renderedComponent;
    var prevRenderedElement = prevRenderedComponent.currentElement;

    // Update *own* element
    this.currentElement = nextElement;
    var type = nextElement.type;
    var nextProps = nextElement.props;

    // Figure out what the next render() output is
    var nextRenderedElement;
    if (isClass(type)) {
      // Component class
      // Call the lifecycle if necessary
      if (publicInstance.componentWillUpdate) {
        publicInstance.componentWillUpdate(nextProps);
      }
      // Update the props
      publicInstance.props = nextProps;
      // Re-render
      nextRenderedElement = publicInstance.render();
    } else if (typeof type === 'function') {
      // Component function
      nextRenderedElement = type(nextProps);
    }
  }
}

```

Next, we can look at the rendered element's `type`. If the `type` has not changed since the last render, the component below can also be updated in place.

For example, if it returned `<Button color="red" />` the first time, and `<Button color="blue" />` the second time, we can just tell the corresponding internal instance to `receive()` the next element:

```

// ...

// If the rendered element type has not changed,
// reuse the existing component instance and exit.
if (prevRenderedElement.type === nextRenderedElement.type) {
  prevRenderedComponent.receive(nextRenderedElement);
  return;
}

// ...

```

However, if the next rendered element has a different `type` than the previously rendered element, we can't update the internal instance. A `<button>` can't "become" an `<input>`.

Instead, we have to unmount the existing internal instance and mount the new one corresponding to the rendered element type. For example, this is what happens when a component that previously rendered a `<button />` renders an `<input />`:

```
// ...

// If we reached this point, we need to unmount the previously
// mounted component, mount the new one, and swap their nodes.

// Find the old node because it will need to be replaced
var prevNode = prevRenderedComponent.getHostNode();

// Unmount the old child and mount a new child
prevRenderedComponent.unmount();
var nextRenderedComponent = instantiateComponent(nextRenderedElement);
var nextNode = nextRenderedComponent.mount();

// Replace the reference to the child
this.renderedComponent = nextRenderedComponent;

// Replace the old node with the new one
// Note: this is renderer-specific code and
// ideally should live outside of CompositeComponent:
prevNode.parentNode.replaceChild(nextNode, prevNode);

}

}
```

To sum this up, when a composite component receives a new element, it may either delegate the update to its rendered internal instance, or unmount it and mount a new one in its place.

There is another condition under which a component will re-mount rather than receive an element, and that is when the element's `key` has changed. We don't discuss `key` handling in this document because it adds more complexity to an already complex tutorial.

Note that we needed to add a method called `getHostNode()` to the internal instance contract so that it's possible to locate the platform-specific node and replace it during the update. Its implementation is straightforward for both classes:

```
class CompositeComponent {
// ...

getHostNode() {
  // Ask the rendered component to provide it.
  // This will recursively drill down any composites.
  return this.renderedComponent.getHostNode();
}

class DOMComponent {
// ...

getHostNode() {
  return this.node;
}
}
```

Updating Host Components

Host component implementations, such as `DOMComponent`, update differently. When they receive an element, they need to update the underlying platform-specific view. In case of React DOM, this means updating the DOM attributes:

```

class DOMComponent {
// ...

receive(nextElement) {
  var node = this.node;
  var prevElement = this.currentElement;
  var prevProps = prevElement.props;
  var nextProps = nextElement.props;
  this.currentElement = nextElement;

  // Remove old attributes.
  Object.keys(prevProps).forEach(propName => {
    if (propName !== 'children' && !nextProps.hasOwnProperty(propName)) {
      node.removeAttribute(propName);
    }
  });
  // Set next attributes.
  Object.keys(nextProps).forEach(propName => {
    if (propName !== 'children') {
      node.setAttribute(propName, nextProps[propName]);
    }
  });
}

// ...

```

Then, host components need to update their children. Unlike composite components, they might contain more than a single child.

In this simplified example, we use an array of internal instances and iterate over it, either updating or replacing the internal instances depending on whether the received `type` matches their previous `type`. The real reconciler also takes element's `key` in the account and track moves in addition to insertions and deletions, but we will omit this logic.

We collect DOM operations on children in a list so we can execute them in batch:

```

// ...

// These are arrays of React elements:
var prevChildren = prevProps.children || [];
if (!Array.isArray(prevChildren)) {
  prevChildren = [prevChildren];
}
var nextChildren = nextProps.children || [];
if (!Array.isArray(nextChildren)) {
  nextChildren = [nextChildren];
}
// These are arrays of internal instances:
var prevRenderedChildren = this.renderedChildren;
var nextRenderedChildren = [];

// As we iterate over children, we will add operations to the array.
var operationQueue = [];

// Note: the section below is extremely simplified!
// It doesn't handle reorders, children with holes, or keys.
// It only exists to illustrate the overall flow, not the specifics.

for (var i = 0; i < nextChildren.length; i++) {
  // Try to get an existing internal instance for this child

```

```

var prevChild = prevRenderedChildren[i];

// If there is no internal instance under this index,
// a child has been appended to the end. Create a new
// internal instance, mount it, and use its node.
if (!prevChild) {
  var nextChild = instantiateComponent(nextChildren[i]);
  var node = nextChild.mount();

  // Record that we need to append a node
  operationQueue.push({type: 'ADD', node});
  nextRenderedChildren.push(nextChild);
  continue;
}

// We can only update the instance if its element's type matches.
// For example, <Button size="small" /> can be updated to
// <Button size="large" /> but not to an <App />.
var canUpdate = prevChildren[i].type === nextChildren[i].type;

// If we can't update an existing instance, we have to unmount it
// and mount a new one instead of it.
if (!canUpdate) {
  var prevNode = prevChild.getHostNode();
  prevChild.unmount();

  var nextChild = instantiateComponent(nextChildren[i]);
  var nextNode = nextChild.mount();

  // Record that we need to swap the nodes
  operationQueue.push({type: 'REPLACE', prevNode, nextNode});
  nextRenderedChildren.push(nextChild);
  continue;
}

// If we can update an existing internal instance,
// just let it receive the next element and handle its own update.
prevChild.receive(nextChildren[i]);
nextRenderedChildren.push(prevChild);
}

// Finally, unmount any children that don't exist:
for (var j = nextChildren.length; j < prevChildren.length; j++) {
  var prevChild = prevRenderedChildren[j];
  var node = prevChild.getHostNode();
  prevChild.unmount();

  // Record that we need to remove the node
  operationQueue.push({type: 'REMOVE', node});
}

// Point the list of rendered children to the updated version.
this.renderedChildren = nextRenderedChildren;

// ...

```

As the last step, we execute the DOM operations. Again, the real reconciler code is more complex because it also handles moves:

```
// ...

// Process the operation queue.
while (operationQueue.length > 0) {
  var operation = operationQueue.shift();
  switch (operation.type) {
    case 'ADD':
      this.node.appendChild(operation.node);
      break;
    case 'REPLACE':
      this.node.replaceChild(operation.nextNode, operation.prevNode);
      break;
    case 'REMOVE':
      this.node.removeChild(operation.node);
      break;
  }
}
}
```

And that is it for updating host components.

Top-Level Updates

Now that both `CompositeComponent` and `DOMComponent` implement the `receive(nextElement)` method, we can change the top-level `mountTree()` function to use it when the element `type` is the same as it was the last time:

```
function mountTree(element, containerNode) {
  // Check for an existing tree
  if (containerNode.firstChild) {
    var prevNode = containerNode.firstChild;
    var prevRootComponent = prevNode._internalInstance;
    var prevElement = prevRootComponent.currentElement;

    // If we can, reuse the existing root component
    if (prevElement.type === element.type) {
      prevRootComponent.receive(element);
      return;
    }

    // Otherwise, unmount the existing tree
    unmountTree(containerNode);
  }

  // ...
}
```

Now calling `mountTree()` two times with the same type isn't destructive:

```
var rootEl = document.getElementById('root');

mountTree(<App />, rootEl);
// Reuses the existing DOM:
mountTree(<App />, rootEl);
```

These are the basics of how React works internally.

What We Left Out

This document is simplified compared to the real codebase. There are a few important aspects we didn't address:

- Components can render `null`, and the reconciler can handle "empty slots" in arrays and rendered output.
- The reconciler also reads `key` from the elements, and uses it to establish which internal instance corresponds to which element in an array. A bulk of complexity in the actual React implementation is related to that.
- In addition to composite and host internal instance classes, there are also classes for "text" and "empty" components. They represent text nodes and the "empty slots" you get by rendering `null`.
- Renderers use [injection](#) to pass the host internal class to the reconciler. For example, React DOM tells the reconciler to use `ReactDOMComponent` as the host internal instance implementation.
- The logic for updating the list of children is extracted into a mixin called `ReactMultiChild` which is used by the host internal instance class implementations both in React DOM and React Native.
- The reconciler also implements support for `setState()` in composite components. Multiple updates inside event handlers get batched into a single update.
- The reconciler also takes care of attaching and detaching refs to composite components and host nodes.
- Lifecycle methods that are called after the DOM is ready, such as `componentDidMount()` and `componentDidUpdate()`, get collected into "callback queues" and are executed in a single batch.
- React puts information about the current update into an internal object called "transaction". Transactions are useful for keeping track of the queue of pending lifecycle methods, the current DOM nesting for the warnings, and anything else that is "global" to a specific update. Transactions also ensure React "cleans everything up" after updates. For example, the transaction class provided by React DOM restores the input selection after any update.

Jumping into the Code

- `ReactMount` is where the code like `mountTree()` and `unmountTree()` from this tutorial lives. It takes care of mounting and unmounting top-level components. `ReactNativeMount` is its React Native analog.
- `ReactDOMComponent` is the equivalent of `DOMComponent` in this tutorial. It implements the host component class for React DOM renderer. `ReactNativeBaseComponent` is its React Native analog.
- `ReactCompositeComponent` is the equivalent of `CompositeComponent` in this tutorial. It handles calling user-defined components and maintaining their state.
- `instantiateReactComponent` contains the switch that picks the right internal instance class to construct for an element. It is equivalent to `instantiateComponent()` in this tutorial.

- `ReactReconciler` is a wrapper with `mountComponent()`, `receiveComponent()`, and `unmountComponent()` methods. It calls the underlying implementations on the internal instances, but also includes some code around them that is shared by all internal instance implementations.
- `ReactChildReconciler` implements the logic for mounting, updating, and unmounting children according to the `key` of their elements.
- `ReactMultiChild` implements processing the operation queue for child insertions, deletions, and moves independently of the renderer.
- `mount()`, `receive()`, and `unmount()` are really called `mountComponent()`, `receiveComponent()`, and `unmountComponent()` in React codebase for legacy reasons, but they receive elements.
- Properties on the internal instances start with an underscore, e.g. `_currentElement`. They are considered to be read-only public fields throughout the codebase.

Future Directions

Stack reconciler has inherent limitations such as being synchronous and unable to interrupt the work or split it in chunks. There is a work in progress on the [new Fiber reconciler](#) with a [completely different architecture](#). In the future, we intend to replace stack reconciler with it, but at the moment it is far from feature parity.

Next Steps

Read the [next section](#) to learn about the guiding principles we use for React development.

Integrating with Other Libraries

React can be used in any web application. It can be embedded in other applications and, with a little care, other applications can be embedded in React. This guide will examine some of the more common use cases, focusing on integration with [jQuery](#) and [Backbone](#), but the same ideas can be applied to integrating components with any existing code.

Integrating with DOM Manipulation Plugins

React is unaware of changes made to the DOM outside of React. It determines updates based on its own internal representation, and if the same DOM nodes are manipulated by another library, React gets confused and has no way to recover.

This does not mean it is impossible or even necessarily difficult to combine React with other ways of affecting the DOM, you just have to be mindful of what each is doing.

The easiest way to avoid conflicts is to prevent the React component from updating. You can do this by rendering elements that React has no reason to update, like an empty `<div />`.

How to Approach the Problem

To demonstrate this, let's sketch out a wrapper for a generic jQuery plugin.

We will attach a `ref` to the root DOM element. Inside `componentDidMount`, we will get a reference to it so we can pass it to the jQuery plugin.

To prevent React from touching the DOM after mounting, we will return an empty `<div />` from the `render()` method. The `<div />` element has no properties or children, so React has no reason to update it, leaving the jQuery plugin free to manage that part of the DOM:

```
class SomePlugin extends React.Component {
  componentDidMount() {
    this.$el = $(this.el);
    this.$el.somePlugin();
  }

  componentWillUnmount() {
    this.$el.somePlugin('destroy');
  }

  render() {
    return <div ref={el => this.el = el} />;
  }
}
```

Note that we defined both `componentDidMount` and `componentWillUnmount` lifecycle methods. Many jQuery plugins attach event listeners to the DOM so it's important to detach them in `componentWillUnmount`. If the plugin does not provide a method for cleanup, you will probably have to provide your own, remembering to remove any event listeners the plugin registered to prevent memory leaks.

Integrating with jQuery Chosen Plugin

For a more concrete example of these concepts, let's write a minimal wrapper for the plugin [Chosen](#), which augments `<select>` inputs.

Note:

Just because it's possible, doesn't mean that it's the best approach for React apps. We encourage you to use React components when you can. React components are easier to reuse in React applications, and often provide more control over their behavior and appearance.

First, let's look at what Chosen does to the DOM.

If you call it on a `<select>` DOM node, it reads the attributes off of the original DOM node, hides it with an inline style, and then appends a separate DOM node with its own visual representation right after the `<select>`. Then it fires jQuery events to notify us about the changes.

Let's say that this is the API we're striving for with our `<Chosen>` wrapper React component:

```
function Example() {
  return (
    <Chosen onChange={value => console.log(value)}>
      <option>vanilla</option>
      <option>chocolate</option>
      <option>strawberry</option>
    </Chosen>
  );
}
```

We will implement it as an [uncontrolled component](#) for simplicity.

First, we will create an empty component with a `render()` method where we return `<select>` wrapped in a `<div>`:

```
class Chosen extends React.Component {
  render() {
    return (
      <div>
        <select className="Chosen-select" ref={el => this.el = el}>
          {this.props.children}
        </select>
      </div>
    );
  }
}
```

Notice how we wrapped `<select>` in an extra `<div>`. This is necessary because Chosen will append another DOM element right after the `<select>` node we passed to it. However, as far as React is concerned, `<div>` always only has a single child. This is how we ensure that React updates won't conflict with the extra DOM node appended by Chosen. It is important that if you modify the DOM outside of React flow, you must ensure React doesn't have a reason to touch those DOM nodes.

Next, we will implement the lifecycle methods. We need to initialize Chosen with the ref to the `<select>` node in `componentDidMount`, and tear it down in `componentWillUnmount`:

```
componentDidMount() {
  this.$el = $(this.el);
  this.$el.chosen();
}

componentWillUnmount() {
  this.$el.chosen('destroy');
}
```

[Try it on CodePen](#)

Note that React assigns no special meaning to the `this.el` field. It only works because we have previously assigned this field from a `ref` in the `render()` method:

```
<select className="Chosen-select" ref={el => this.el = el}>
```

This is enough to get our component to render, but we also want to be notified about the value changes. To do this, we will subscribe to the jQuery `change` event on the `<select>` managed by Chosen.

We won't pass `this.props.onChange` directly to Chosen because component's props might change over time, and that includes event handlers. Instead, we will declare a `handleChange()` method that calls `this.props.onChange`, and subscribe it to the jQuery `change` event:

```
componentDidMount() {
  this.$el = $(this.el);
  this.$el.chosen();

  this.handleChange = this.handleChange.bind(this);
  this.$el.on('change', this.handleChange);
}

componentWillUnmount() {
  this.$el.off('change', this.handleChange);
  this.$el.chosen('destroy');
}

handleChange(e) {
  this.props.onChange(e.target.value);
}
```

[Try it on CodePen](#)

Finally, there is one more thing left to do. In React, props can change over time. For example, the `<Chosen>` component can get different children if parent component's state changes. This means that at integration points it is important that we manually update the DOM in response to prop updates, since we no longer let React manage the DOM for us.

Chosen's documentation suggests that we can use jQuery `trigger()` API to notify it about changes to the original DOM element. We will let React take care of updating `this.props.children` inside `<select>`, but we will also add a `componentDidUpdate()` lifecycle method that notifies Chosen about changes in the children list:

```
componentDidUpdate(prevProps) {
  if (prevProps.children !== this.props.children) {
    this.$el.trigger("chosen:updated");
  }
}
```

This way, Chosen will know to update its DOM element when the `<select>` children managed by React change.

The complete implementation of the `Chosen` component looks like this:

```
class Chosen extends React.Component {
  componentDidMount() {
    this.$el = $(this.el);
    this.$el.chosen();

    this.handleChange = this.handleChange.bind(this);
    this.$el.on('change', this.handleChange);
  }

  componentDidUpdate(prevProps) {
    if (prevProps.children !== this.props.children) {
      this.$el.trigger("chosen:updated");
    }
  }

  componentWillUnmount() {
    this.$el.off('change', this.handleChange);
    this.$el.chosen('destroy');
  }

  handleChange(e) {
    this.props.onChange(e.target.value);
  }

  render() {
    return (
      <div>
        <select className="Chosen-select" ref={el => this.el = el}>
          {this.props.children}
        </select>
      </div>
    );
  }
}
```

[Try it on CodePen](#)

Integrating with Other View Libraries

React can be embedded into other applications thanks to the flexibility of `createRoot()`.

Although React is commonly used at startup to load a single root React component into the DOM, `createRoot()` can also be called multiple times for independent parts of the UI which can be as small as a button, or as large as an app.

In fact, this is exactly how React is used at Facebook. This lets us write applications in React piece by piece, and combine them with our existing server-generated templates and other client-side code.

Replacing String-Based Rendering with React

A common pattern in older web applications is to describe chunks of the DOM as a string and insert it into the DOM like so: `$('#el').html(htmlString)`. These points in a codebase are perfect for introducing React. Just rewrite the string based rendering as a React component.

So the following jQuery implementation...

```
$('#container').html('<button id="btn">Say Hello</button>');
$('#btn').click(function() {
  alert('Hello!');
});
```

...could be rewritten using a React component:

```
function Button() {
  return <button id="btn">Say Hello</button>;
}

$('#btn').click(function() {
  alert('Hello!');
});
```

From here you could start moving more logic into the component and begin adopting more common React practices. For example, in components it is best not to rely on IDs because the same component can be rendered multiple times. Instead, we will use the [React event system](#) and register the click handler directly on the React `<button>` element:

```
function Button(props) {
  return <button onClick={props.onClick}>Say Hello</button>;
}

function HelloButton() {
  function handleClick() {
    alert('Hello!');
  }
  return <Button onClick={handleClick} />;
}
```

[Try it on CodePen](#)

You can have as many such isolated components as you like, and use `ReactDOM.createRoot()` to render them to different DOM containers. Gradually, as you convert more of your app to React, you will be able to combine them into larger components, and move some of the `ReactDOM.createRoot()` calls up the hierarchy.

Embedding React in a Backbone View

[Backbone](#) views typically use HTML strings, or string-producing template functions, to create the content for their DOM elements. This process, too, can be replaced with rendering a React component.

Below, we will create a Backbone view called `ParagraphView`. It will override Backbone's `render()` function to render a React `<Paragraph>` component into the DOM element provided by Backbone (`this.el`). Here, too, we are using `ReactDOM.createRoot()`:

```
function Paragraph(props) {
  return <p>{props.text}</p>;
}

const ParagraphView = Backbone.View.extend({
  initialize(options) {
    this.reactRoot = ReactDOM.createRoot(this.el);
  },
  render() {
    const text = this.model.get('text');
    this.reactRoot.render(<Paragraph text={text} />);
    return this;
  },
  remove() {
    this.reactRoot.unmount();
    Backbone.View.prototype.remove.call(this);
  }
});
```

Try it on [CodePen](#)

It is important that we also call `root.unmount()` in the `remove` method so that React unregisters event handlers and other resources associated with the component tree when it is detached.

When a component is removed *from within* a React tree, the cleanup is performed automatically, but because we are removing the entire tree by hand, we must call this method.

Integrating with Model Layers

While it is generally recommended to use unidirectional data flow such as [React state](#), [Flux](#), or [Redux](#), React components can use a model layer from other frameworks and libraries.

Using Backbone Models in React Components

The simplest way to consume [Backbone](#) models and collections from a React component is to listen to the various change events and manually force an update.

Components responsible for rendering models would listen to `'change'` events, while components responsible for rendering collections would listen for `'add'` and `'remove'` events. In both cases, call `this.forceUpdate()` to rerender the component with the new data.

In the example below, the `List` component renders a Backbone collection, using the `Item` component to render individual items.

```
class Item extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
  }

  handleChange() {
    this.forceUpdate();
  }

  componentDidMount() {
    this.props.model.on('change', this.handleChange);
  }

  componentWillUnmount() {
    this.props.model.off('change', this.handleChange);
  }

  render() {
    return <li>{this.props.model.get('text')}</li>;
  }
}

class List extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
  }

  handleChange() {
    this.forceUpdate();
  }

  componentDidMount() {
    this.props.collection.on('add', 'remove', this.handleChange);
  }

  componentWillUnmount() {
    this.props.collection.off('add', 'remove', this.handleChange);
  }

  render() {
    return (
      <ul>
        {this.props.collection.map(model => (
          <Item key={model.cid} model={model} />
        ))}
      </ul>
    );
  }
}
```

[Try it on CodePen](#)

Extracting Data from Backbone Models

The approach above requires your React components to be aware of the Backbone models and collections. If you later plan to migrate to another data management solution, you might want to concentrate the knowledge about Backbone in as few parts of the code as possible.

One solution to this is to extract the model's attributes as plain data whenever it changes, and keep this logic in a single place. The following is a [higher-order component](#) that extracts all attributes of a Backbone model into state, passing the data to the wrapped component.

This way, only the higher-order component needs to know about Backbone model internals, and most components in the app can stay agnostic of Backbone.

In the example below, we will make a copy of the model's attributes to form the initial state. We subscribe to the `change` event (and unsubscribe on unmounting), and when it happens, we update the state with the model's current attributes. Finally, we make sure that if the `model` prop itself changes, we don't forget to unsubscribe from the old model, and subscribe to the new one.

Note that this example is not meant to be exhaustive with regards to working with Backbone, but it should give you an idea for how to approach this in a generic way:

```
function connectToBackboneModel(WrappedComponent) {
  return class BackboneComponent extends React.Component {
    constructor(props) {
      super(props);
      this.state = Object.assign({}, props.model.attributes);
      this.handleChange = this.handleChange.bind(this);
    }

    componentDidMount() {
      this.props.model.on('change', this.handleChange);
    }

    componentWillReceiveProps(nextProps) {
      this.setState(Object.assign({}, nextProps.model.attributes));
      if (nextProps.model !== this.props.model) {
        this.props.model.off('change', this.handleChange);
        nextProps.model.on('change', this.handleChange);
      }
    }

    componentWillUnmount() {
      this.props.model.off('change', this.handleChange);
    }

    handleChange(model) {
      this.setState(model.changedAttributes());
    }

    render() {
      const propsExceptModel = Object.assign({}, this.props);
      delete propsExceptModel.model;
      return <WrappedComponent {...propsExceptModel} {...this.state} />;
    }
  };
}
```

```

        }
    }
}
```

To demonstrate how to use it, we will connect a `NameInput` React component to a Backbone model, and update its `firstName` attribute every time the input changes:

```

function NameInput(props) {
  return (
    <p>
      <input value={props.firstName} onChange={props.handleChange} />
      <br />
      My name is {props.firstName}.
    </p>
  );
}

const BackboneNameInput = connectToBackboneModel(NameInput);

function Example(props) {
  function handleChange(e) {
    props.model.set('firstName', e.target.value);
  }

  return (
    <BackboneNameInput
      model={props.model}
      handleChange={handleChange}
    />
  );
}

const model = new Backbone.Model({ firstName: 'Frodo' });
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Example model={model} />);
```

[Try it on CodePen](#)

This technique is not limited to Backbone. You can use React with any model library by subscribing to its changes in the lifecycle methods and, optionally, copying the data into the local React state.

[Go to TOC](#)

介紹 JSX

考慮下面這個變數宣告：

```
const element = <h1>你好，世界！</h1>;
```

這個有趣的標籤語法不是一個字串也不是 HTML。

這個語法叫做 JSX，是一個 JavaScript 的語法擴充。我們推薦你在寫 React 的時候透過這個語法來描述使用者介面的外觀。JSX 可能為讓你想到一些樣板語言，但不一樣的地方是 JSX 允許你使用 JavaScript 所有的功能。

執行 JSX 會產生 React 「element」。我們會在[下一個章節](#)深入如何將這些輸出 render 到 DOM 裡頭。接下來，我們將帶您了解 JSX 的基礎。

為什麼要用 JSX？

React 摊抱了 render 邏輯從根本上就得跟其他 UI 邏輯綁在一起的事實：事件要怎麼處理？隨著時間經過 state 會如何變化？以及要怎麼將資料準備好用於顯示？

與其刻意的將技術拆開，把標籤語法跟邏輯拆放於不同檔案之中，React [關注點分離](#)的方法是將其拆分為很多同時包含 UI 與邏輯的 component，而彼此之間很少互相依賴。我們會在[之後的章節](#)中回來探討 component 這個主題，但如果你還沒被說服接受將標籤語法寫在 JS 裡頭，[這個演講](#)或許會說服你。

React [並不要求](#)使用 JSX，但大部分人覺得在 JavaScript 程式碼中撰寫使用者介面的同時，這是一個很好的視覺輔助。這也允許 React 顯示更有用的錯誤及警告訊息。

好，說完這個了，讓我們開始吧！

在 JSX 中嵌入 Expression

在下面這個範例中，我們宣告一個名為 `name` 的變數，並在 JSX 中透過將其名稱包在大括號中使用：

```
const name = 'Josh Perez';
const element = <h1>Hello, {name}</h1>;
```

你可以在 JSX 的大括號中寫入任何合法的 [JavaScript expression](#)。舉例來說，`2 + 2`、`user.firstName` 以及 `formatName(user)` 都是合法的 JavaScript expression。

接下來的範例中，我們將嵌入呼叫 JavaScript function (`formatName(user)`) 的回傳值到一個 `<h1>` element 中。

```
function formatName(user) {
  return user.firstName + ' ' + user.lastName;
}

const user = {
  firstName: 'Harper',
  lastName: 'Perez'
};

const element = (
  <h1>
```

```
Hello, {formatName(user)}!
  </h1>
);
```

在 [CodePen 上試試看吧！](#)

為了方便閱讀，我們將 JSX 拆成很多行表達。雖然這並不需要，我們建議將多行 JSX 包在括號中來避免遇到[自動分號補足](#)的麻煩。

JSX 本身也是 Expression

在編譯之後，JSX expressions 就變成了一般的 JavaScript function 呼叫並回傳 JavaScript 物件。這表示你也可以在 `if` 跟 `for` 迴圈中使用 JSX，將其指定到一個變數，使用 JSX 作為參數並由 function 中回傳。

```
function getGreeting(user) {
  if (user) {
    return <h1>Hello, {formatName(user)}!</h1>;
  }
  return <h1>Hello, Stranger.</h1>;
}
```

在 JSX 中指定屬性

你可以使用引號將字串設定為屬性：

```
const element = <a href="https://www.reactjs.org"> link </a>;
```

你也可以在屬性中使用大括號來嵌入一個 JavaScript expression：

```
const element = <img src={user.avatarUrl}></img>;
```

不要在嵌入 JavaScript expression 作為屬性的時候同時使用引號或是大括號。你應該要在使用字串屬性的時候使用引號，使用 expressions 的時候使用大括號，但不要同時使用。

注意：

由於 JSX 比較接近 JavaScript 而不是 HTML，React DOM 使用 `camelCase` 來命名屬性而不是使用慣有的 HTML 屬性名稱。

舉例來說：在 JSX 之中，`class` 變成了 `className` 而 `tabindex` 變成了 `tabIndex`。

在 JSX 中指定 Children

就像是在 XML 之中，如果一個標籤是空白的，你可以用 `/>` 立刻關閉這個標籤：

```
const element = <img src={user.avatarUrl} />;
```

JSX 標籤也可以包含 children：

```
const element = (
  <div>
    <h1>Hello!</h1>
    <h2>Good to see you here.</h2>
  </div>
);
```

JSX 防範注入攻擊

在 JSX 之中可以安全的直接嵌入使用者輸入：

```
const title = response.potentiallyMaliciousInput;
// 這是安全的：
const element = <h1>{title}</h1>;
```

React DOM 預設會在 render 之前 escape 所有嵌入在 JSX 中的變數。這保證你永遠不會不小心注入任何不是直接寫在你的應用程式中的東西。所有變數都會在 render 之前轉為字串，這可以避免 XSS（跨網站指令碼）攻擊。

JSX 表示物件

Babel 將 JSX 編譯為呼叫 `React.createElement()` 的程式。

下面這兩個例子完全相同：

```
const element = (
  <h1 className="greeting">
    Hello, World!
  </h1>
);
```

```
const element = React.createElement(
  'h1',
  {className: 'greeting'},
  'Hello, World!'
);
```

`React.createElement()` 會進行一些檢查以幫助你寫出沒有 bug 的程式，但基本上它會產生類似下面的物件：

```
// 注意：這是簡化過的結構
const element = {
  type: 'h1',
  props: {
    className: 'greeting',
    children: 'Hello, world!'
  }
};
```

這種物件被稱呼為「React element」。你可以想像他們描述的是你想要在螢幕上看到的東西，React 會讀取這些物件並用這些描述來產生 DOM 並保持他們在最新狀態。

我們會在[下一個章節](#)探討如何把 React element render 到 DOM 之中。

提示：

我們推薦你在編輯器中使用 [「Babel」語法](#), 這樣可以確保 ES6 跟 JSX 都能夠正確的被語法突顯。

[Go to TOC](#)

深入 JSX

基本上，JSX 單純只是 `React.createElement(component, props, ...children)` function 的一個語法糖。以下 JSX 程式碼：

```
<MyButton color="blue" shadowSize={2}>
  Click Me
</MyButton>
```

會編譯成：

```
React.createElement(
  MyButton,
  {color: 'blue', shadowSize: 2},
  'Click Me'
)
```

如果沒有 children 的話你也可以使用閉合的標籤形式。例如：

```
<div className="sidebar" />
```

會編譯成：

```
React.createElement(
  'div',
  {className: 'sidebar'}
)
```

如果你想測試某些特定的 JSX 會轉換成什麼樣的 JavaScript，你可以在線上 [Babel 編譯器](#) 進行測試。

指定 React Element 類型

JSX 標籤的第一個部分決定 React element 的類型。

大寫字母的 JSX 標籤代表它們是 React Component。這些標籤會編譯成指向命名變數的 reference，所以當你使用 JSX `<Foo />` 表達式時，`Foo` 就必須在作用域內。

React 必須在作用域內

因為 JSX 會編譯成呼叫 `React.createElement` 的形式，`React` 函式庫必須同時與你的 JSX 程式碼在相同的作用域內。

舉例來說，雖然 `React` 與 `CustomButton` 並沒有在 JavaScript 中被直接使用，但是在此程式碼中它們必須被導入：

```
import React from 'react';
import CustomButton from './CustomButton';

function WarningButton() {
  // return React.createElement(CustomButton, {color: 'red'}, null);
  return <CustomButton color="red" />;
}
```

如果你不使用任何 JavaScript bundler 並從 `<script>` 標籤來載入 React，那麼它就已經以 `React` 存在於作用域中了。

在 JSX 類型中使用點記法

你也可以在 JSX 中使用點記法來指向一個 React Component。這對當你有一個會導出許多 React component 的 module 來講是十分方便的。舉例來說，如果 `MyComponents.DatePicker` 是一個 component，那麼你可以在 JSX 中直接這樣使用：

```
import React from 'react';

const MyComponents = {
  DatePicker: function DatePicker(props) {
    return <div>Imagine a {props.color} datepicker here.</div>;
  }
}

function BlueDatePicker() {
  return <MyComponents.DatePicker color="blue" />;
}
```

使用者定義的 Component 必須由大寫字母開頭

當一個 element 為小寫字母開頭時，它就會指向內建的 component，例如 `<div>` 或 `` 會生產成字串 `'div'` 或 `'span'` 並傳遞給 `React.createElement`。像 `<Foo />` 大寫字母開頭的 element 會編譯成 `React.createElement(Foo)` 並且在你的 JavaScript 檔案裡對應自定義或導入的 component。

我們建議以大寫字母開頭來命名 component。如果你有一個小寫字母開頭的 component，請在 JSX 裡使用之前把它賦值給一個大寫字母開頭的變數。

例如，以下程式碼並不會按照預期運行：

```
import React from 'react';

// 錯誤！這是一個 component 並且應該由大寫字母開頭：
function hello(props) {
  // 正確！因為 div 是一個有效的 HTML 標籤，所以使用<div> 是可行的：
  return <div>Hello {props.toWhat}</div>;
}

function HelloWorld() {
  // 錯誤！React 會因為非大寫字母開頭而認為 <hello /> 是一個 HTML 標籤：
  return <hello toWhat="World" />;
}
```

為了解決這個問題，我們將重新命名 `hello` 成 `Hello` 並且以 `<Hello />` 來使用它：

```
import React from 'react';

// 正確！這是一個 component 並且應該由大寫字母開頭：
function Hello(props) {
  // 正確！因為 div 是一個有效的 HTML 標籤，所以使用 <div> 是可行的：
  return <div>Hello {props.toWhat}</div>;
}
```

```
function HelloWorld() {
  // 正確！React 會因為大寫字母開頭而了解 <Hello /> 是一個 component。
  return <Hello toWhat="World" />;
}
```

在 Runtime 時選擇類型

你不能用通用表達式當作 React element 的類型。如果你想要用通用表達式來表示 element 的類型，你可以先把它賦值給一個大寫字母開頭的變數。這是在當你想要根據一個 prop 來決定 render 不同 component 時常常發生的：

```
import React from 'react';
import { PhotoStory, VideoStory } from './stories';

const components = {
  photo: PhotoStory,
  video: VideoStory
};

function Story(props) {
  // 錯誤！JSX 不能是表達式。
  return <components[props.storyType] story={props.story} />;
}
```

為了解決這個問題，我們首先將類型賦值給一個大寫字母開頭的變數：

```
import React from 'react';
import { PhotoStory, VideoStory } from './stories';

const components = {
  photo: PhotoStory,
  video: VideoStory
};

function Story(props) {
  // 正確！JSX 類型可以是大寫字母開頭的變數。
  const SpecificStory = components[props.storyType];
  return <SpecificStory story={props.story} />;
}
```

JSX 中的 Props

在 JSX 中有不同的方式可以指定 props。

JavaScript 表達式作為 Props

你可以用 `{}` 包住任何 JavaScript 表達式作為一個 prop 傳遞。例如，在以下 JSX 中：

```
<MyComponent foo={1 + 2 + 3 + 4} />
```

對 `MyComponent` 來說，因為 `1 + 2 + 3 + 4` 會被解讀，所以 `props.foo` 的值會是 `10`。

因為 `if` 語法與 `for` 迴圈都不屬於 JavaScript 表達式，所以它們並不能在 JSX 中被直接使用。不過，你可以在它周圍外的程式碼中使用。例如：

```
function NumberDescriber(props) {
  let description;
  if (props.number % 2 == 0) {
    description = <strong>even</strong>;
  } else {
    description = <i>odd</i>;
  }
  return <div>{props.number} is an {description} number</div>;
}
```

你可以在對應的段落中了解更多關於[條件式 render 與迴圈](#)。

字串字面值

你可以傳遞一個字串字面值作為 prop。以下兩個 JSX 表達式是相等的：

```
<MyComponent message="hello world" />
<MyComponent message={['hello world']} />
```

當你傳遞一個字串字面值時，它的值是未經 HTML 轉義的。所以以下兩個 JSX 表達式是相等的：

```
<MyComponent message=&lt;3&gt; />
<MyComponent message={['<3'']} />
```

這種行為通常是無關緊要的。在此只是為了完整性而提及。

Props 預設為「True」

如果你沒給 prop 賦值，那麼它的預設值就是 `true`。以下兩個 JSX 表達式是相等的：

```
<MyTextBox autocomplete />
<MyTextBox autocomplete={true} />
```

一般來說，我們不建議不傳遞 prop 的值，因為容易把它跟 [ES6 object shorthand](#) 混淆，`{foo}` 是 `{foo: foo}` 的簡寫而不是 `{foo: true}`，所以並不建議這樣使用。這種行為存在只是為了相配 HTML 的行為。

展開屬性

如果你已經有了一個 `props` 的 object，並且想把它傳遞進 JSX，你可以使用 `...` 作為展開運算子來傳遞整個 props object。以下兩個 component 是相等的：

```
function App1() {
  return <Greeting firstName="Ben" lastName="Hector" />;
}

function App2() {
  const props = {firstName: 'Ben', lastName: 'Hector'};
  return <Greeting {...props} />;
}
```

你也可以使用展開運算子來分開並挑選 component 所需的 props。

```

const Button = props => {
  const { kind, ...other } = props;
  const className = kind === "primary" ? "PrimaryButton" : "SecondaryButton";
  return <button className={className} {...other} />;
};

const App = () => {
  return (
    <div>
      <Button kind="primary" onClick={() => console.log("clicked!")}>
        Hello World!
      </Button>
    </div>
  );
};

```

在以上的範例中，`kind` prop 被安全地挑出並且不會被傳遞進 DOM 中的 `<button>` element。所有其它的 props 藉由 `...other` object 被傳遞，讓 component 的應用非常具有彈性。你可以看見它傳遞一個 `onClick` 與 `children` props。

展開運算子不但可以如此靈活地使用，它能讓我們輕易挑選出對於 component 不重要且多餘的 props，也能讓我們傳遞無效的 HTML 屬性到 DOM 裡。

JSX 中的 Children

在 JSX 表達式有包含開始與結束標籤的情形下，夾在兩者之間的內容會被傳遞為特別的 prop：`props.children`。有幾種不同的方法來傳遞 children：

字串字面值

你可以在兩個標籤之間放置字串，而 `props.children` 就會是那個字串。這對許多內建的 HTML element 是很有用的。例如：

```
<MyComponent>Hello world!</MyComponent>
```

這是有效的 JSX，而 `props.children` 在 `MyComponent` 中單純就會是 `"Hello world!"`。HTML 是未經轉義的，所以你可以這樣如同 HTML 來寫 JSX：

```
<div>This is valid HTML &amp; JSX at the same time.</div>
```

JSX 會把開頭與結尾的空白去除，也會去除空行。與標籤相鄰的新行會去除；在字串字面值中間的新行則會被壓縮成單一空格。所以以下都會 render 同樣的結果：

```

<div>Hello World</div>

<div>
  Hello World
</div>

<div>
  Hello
  World
</div>

```

```
<div>
  Hello World
</div>
```

JSX Children

你可以提供許多 JSX element 作為 children。這在顯示巢狀 component 時是非常實用的：

```
<MyContainer>
  <MyFirstComponent />
  <MySecondComponent />
</MyContainer>
```

你可以混合不同類型的 children，所以你也能夠同時使用字串字面值與 JSX children。這也是 JSX 與 HTML 另一相似的點，而以下是有效的 JSX 同時也是有效的 HTML：

```
<div>
  Here is a list:
  <ul>
    <li>Item 1</li>
    <li>Item 2</li>
  </ul>
</div>
```

一個 React component 也能夠回傳一個陣列 element：

```
render() {
  // 沒有必要把多餘的 list items 包在 element 裡頭！
  return [
    // 別忘了加 keys :)
    <li key="A">First item</li>,
    <li key="B">Second item</li>,
    <li key="C">Third item</li>,
  ];
}
```

JavaScript 表達式作為 Children

你可以傳遞任何封裝在 `{ }` 內的 JavaScript 表達式作為 children。例如，以下表達式皆相等：

```
<MyComponent>foo</MyComponent>
<MyComponent>{'foo'}</MyComponent>
```

這在要 render 任意長度的 JSX 表達式列表時是非常實用的。例如，這會 render 一個 HTML 列表：

```
function Item(props) {
  return <li>{props.message}</li>;
}

function TodoList() {
  const todos = ['finish doc', 'submit pr', 'nag dan to review'];
  return (
    <ul>
      {todos.map((message) => <Item key={message} message={message} />)}
```

```

        </ul>
    );
}

```

JavaScript 表達式可以與不同類型的 children 混合。這在不使用樣板字串時非常實用：

```

function Hello(props) {
    return <div>Hello {props.addressee}!</div>;
}

```

Functions 作為 Children

正常來說，在 JSX 中的 JavaScript 表達式會被轉換成字串、React element、或者包含這些的列表。不過，`props.children` 就像其它 prop 一樣可以傳遞任何類型的資料，而並不局限於只有 React 知道如何 render 的資料。舉例來說，假如你有一個自訂 component，你可以把 callback 作為 `props.children` 傳遞：

```

// numTimes 次呼叫 children callback 來重複生成 component
function Repeat(props) {
    let items = [];
    for (let i = 0; i < props.numTimes; i++) {
        items.push(props.children(i));
    }
    return <div>{items}</div>;
}

function ListOfTenThings() {
    return (
        <Repeat numTimes={10}>
            {(index) => <div key={index}>This is item {index} in the list</div>}
        </Repeat>
    );
}

```

被傳遞進自訂 component 的 children 可以是任何東西，只要 component 能夠在 render 之前把它轉換成 React 能夠理解的東西就可以了。這種用法並不普遍，但能夠顯示出 JSX 的延伸性。

Booleans, Null, 與 Undefined 會被忽略

`false`, `null`, `undefined`, 與 `true` 都是有效的 children。它們只是單純不會被 render。以下 JSX 表達式皆會 render 相同的結果：

```

<div />
<div></div>
<div>{false}</div>
<div>{null}</div>
<div>{undefined}</div>
<div>{true}</div>

```

這在需要條件式 render 不同 React elements 時非常方便。以下 JSX 只會在 `showHeader` 為 `true` 時 render `<Header />`：

```
<div>
  {showHeader && <Header />}
  <Content />
</div>
```

值得注意的是有一些像是數字 `0` 的「**falsy**」值仍然會被 React 給 render。舉例來說，以下的程式碼可能不會如同你預期般地運作，因為當 `props.messages` 是一個空 array 時，`0` 會被印出：

```
<div>
  {props.messages.length &&
    <MessageList messages={props.messages} />
  }
</div>
```

為了解決這個問題，確保在 `&&` 之前的表達式為 boolean：

```
<div>
  {props.messages.length > 0 &&
    <MessageList messages={props.messages} />
  }
</div>
```

相反地，如果你想要印出 `false`、`true`、`null` 或者 `undefined` 時，你必須要先把它轉換成一個字符串：

```
<div>
  My JavaScript variable is {String(myVariable)}.
</div>
```

Legacy Context

Note:

The legacy context API will be removed in a future major version. Use the [new context API](#) introduced with version 16.3. The legacy API will continue working for all 16.x releases.

How To Use Context

This section documents a legacy API. See the [new API](#).

Suppose you have a structure like:

```
class Button extends React.Component {
  render() {
    return (
      <button style={{background: this.props.color}}>
        {this.props.children}
      </button>
    );
  }
}

class Message extends React.Component {
  render() {
    return (
      <div>
        {this.props.text} <Button color={this.props.color}>Delete</Button>
      </div>
    );
  }
}

class MessageList extends React.Component {
  render() {
    const color = "purple";
    const children = this.props.messages.map((message) =>
      <Message text={message.text} color={color} />
    );
    return <div>{children}</div>;
  }
}
```

In this example, we manually thread through a `color` prop in order to style the `Button` and `Message` components appropriately. Using context, we can pass this through the tree automatically:

```
import PropTypes from 'prop-types';

class Button extends React.Component {
```

```

render() {
  return (
    <button style={{background: this.context.color}}>
      {this.props.children}
    </button>
  );
}

Button.contextTypes = {
  color: PropTypes.string
};

class Message extends React.Component {
  render() {
    return (
      <div>
        {this.props.text} <Button>Delete</Button>
      </div>
    );
  }
}

class MessageList extends React.Component {
  getChildContext() {
    return {color: "purple"};
  }

  render() {
    const children = this.props.messages.map((message) =>
      <Message text={message.text} />
    );
    return <div>{children}</div>;
  }
}

MessageList.childContextTypes = {
  color: PropTypes.string
};

```

By adding `childContextTypes` and `getChildContext` to `MessageList` (the context provider), React passes the information down automatically and any component in the subtree (in this case, `Button`) can access it by defining `contextTypes`.

If `contextTypes` is not defined, then `context` will be an empty object.

Note:

`React.PropTypes` has moved into a different package since React v15.5. Please use [the prop-type library instead](#) to define `contextTypes`.

We provide a [codemod script](#) to automate the conversion.

Parent-Child Coupling

This section documents a legacy API. See the [new API](#).

Context can also let you build an API where parents and children communicate. For example, one library that works this way is [React Router V4](#):

```
import { BrowserRouter as Router, Route, Link } from 'react-router-dom';

const BasicExample = () => (
  <Router>
    <div>
      <ul>
        <li><Link to="/">Home</Link></li>
        <li><Link to="/about">About</Link></li>
        <li><Link to="/topics">Topics</Link></li>
      </ul>

      <hr />

      <Route exact path="/" component={Home} />
      <Route path="/about" component={About} />
      <Route path="/topics" component={Topics} />
    </div>
  </Router>
);
```

By passing down some information from the `Router` component, each `Link` and `Route` can communicate back to the containing `Router`.

Before you build components with an API similar to this, consider if there are cleaner alternatives. For example, you can pass entire React components as props if you'd like to.

Referencing Context in Lifecycle Methods

This section documents a legacy API. See the [new API](#).

If `contextTypes` is defined within a component, the following [lifecycle methods](#) will receive an additional parameter, the `context` object:

- `constructor(props, context)`
- `componentWillReceiveProps(nextProps, nextContext)`
- `shouldComponentUpdate(nextProps, nextState, nextContext)`
- `componentWillUpdate(nextProps, nextState, nextContext)`

Note:

As of React 16, `componentDidUpdate` no longer receives `prevContext`.

Referencing Context in Function Components

This section documents a legacy API. See the [new API](#).

Function components are also able to reference `context` if `contextTypes` is defined as a property of the function. The following code shows a `Button` component written as a function component.

```
import PropTypes from 'prop-types';

const Button = ({children}, context) =>
  <button style={{background: context.color}}>
    {children}
  </button>

Button.contextTypes = {color: PropTypes.string};
```

Updating Context

This section documents a legacy API. See the [new API](#).

Don't do it.

React has an API to update context, but it is fundamentally broken and you should not use it.

The `getChildContext` function will be called when the state or props changes. In order to update data in the context, trigger a local state update with `this.setState`. This will trigger a new context and changes will be received by the children.

```
import PropTypes from 'prop-types';

class MediaQuery extends React.Component {
  constructor(props) {
    super(props);
    this.state = {type: 'desktop'};
  }

  getChildContext() {
    return {type: this.state.type};
  }

  componentDidMount() {
    const checkMediaQuery = () => {
      const type = window.matchMedia("(min-width: 1025px)").matches ? 'desktop' : 'mobile';
      if (type !== this.state.type) {
        this.setState({type});
      }
    };
    checkMediaQuery();
    window.addEventListener('resize', checkMediaQuery);
  }
}
```

```
};

  window.addEventListener('resize', checkMediaQuery);
  checkMediaQuery();
}

render() {
  return this.props.children;
}

MediaQuery.childContextTypes = {
  type: PropTypes.string
};
```

The problem is, if a context value provided by component changes, descendants that use that value won't update if an intermediate parent returns `false` from `shouldComponentUpdate`. This is totally out of control of the components using context, so there's basically no way to reliably update the context. [This blog post](#) has a good explanation of why this is a problem and how you might get around it.

Event Pooling

Note

This page is only relevant for React 16 and earlier, and for React Native.

React 17 on the web **does not** use event pooling.

[Read more](#) about this change in React 17.

The `SyntheticEvent` objects are pooled. This means that the `SyntheticEvent` object will be reused and all properties will be nullified after the event handler has been called. For example, this won't work:

```
function handleChange(e) {
  // This won't work because the event object gets reused.
  setTimeout(() => {
    console.log(e.target.value); // Too late!
  }, 100);
}
```

If you need to access event object's properties after the event handler has run, you need to call `e.persist()`:

```
function handleChange(e) {
  // Prevents React from resetting its properties:
  e.persist();

  setTimeout(() => {
    console.log(e.target.value); // Works
  }, 100);
}
```

提升 State

通常來說，有一些 component 需要反映相同的資料變化。我們建議將共享的 state 提升到最靠近它們的共同 ancestor。讓我們來看這是如何運作的。

在這個章節，我們將建立一個溫度計算器來計算水是否會在給定的溫度下沸騰。

我們將會建立一個 component 叫做 `BoilingVerdict`。它接受 `celsius` 溫度作為一個 prop，並印出它是否足夠煮沸開水：

```
function BoilingVerdict(props) {
  if (props.celsius >= 100) {
    return <p>The water would boil.</p>;
  }
  return <p>The water would not boil.</p>;
}
```

接下來，我們將會建立一個叫做 `Calculator` 的 component。它 render 一個 `<input>` 讓你輸入溫度，並且將它的值儲存在 `this.state.temperature`。

此外，它 render 目前輸入值在 `BoilingVerdict`。

```
class Calculator extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {temperature: ''};
  }

  handleChange(e) {
    this.setState({temperature: e.target.value});
  }

  render() {
    const temperature = this.state.temperature;
    return (
      <fieldset>
        <legend>Enter temperature in Celsius:</legend>
        <input
          value={temperature}
          onChange={this.handleChange} />
        <BoilingVerdict
          celsius={parseFloat(temperature)} />
      </fieldset>
    );
  }
}
```

[在 CodePen 上試試看吧！](#)

加入第二個輸入

這是我們的新需求，除了攝氏輸入，我們提供一個華氏輸入，它們彼此保持同步。

我們會從 `Calculator` 抽離出一個 `TemperatureInput` component。我們將會加入一個新的 `scale` prop，它可以是「c」或「f」：

```
const scaleNames = {
  c: 'Celsius',
  f: 'Fahrenheit'
};

class TemperatureInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {temperature: ''};
  }

  handleChange(e) {
    this.setState({temperature: e.target.value});
  }

  render() {
    const temperature = this.state.temperature;
    const scale = this.props.scale;
    return (
      <fieldset>
        <legend>Enter temperature in {scaleNames[scale]}:</legend>
        <input value={temperature}>
          onChange={this.handleChange}
        </input>
      </fieldset>
    );
  }
}
```

我們現在可以改變 `Calculator` 來 render 兩個獨立的溫度輸入：

```
class Calculator extends React.Component {
  render() {
    return (
      <div>
        <TemperatureInput scale="c" />
        <TemperatureInput scale="f" />
      </div>
    );
  }
}
```

[在 CodePen 上試試看吧！](#)

現在我們有兩個輸入，但是當你輸入其中一個溫度輸入時，另外一個輸入並沒有更新。這和我們的需求產生了矛盾：我們希望它們可以保持同步。

我們也無法從 `Calculator` 顯示 `BoilingVerdict`。`Calculator` 並不知道目前的溫度，因為它被隱藏在 `TemperatureInput` 內。

撰寫轉換 Function

首先，我們將撰寫兩個 function 來轉換攝氏和華氏：

```

function toCelsius(fahrenheit) {
  return (fahrenheit - 32) * 5 / 9;
}

function toFahrenheit(celsius) {
  return (celsius * 9 / 5) + 32;
}

```

這兩個 function 轉換 number。我們將撰寫其他 function，它將 `temperature` string 和轉換 function 作為參數，並回傳 string。我們將使用它來計算基於另一個輸入的輸入值。

當 `temperattrue` 無效時，回傳一個空的 string，它的輸出會四捨五入到小數點後第三位：

```

function tryConvert(temperature, convert) {
  const input = parseFloat(temperature);
  if (Number.isNaN(input)) {
    return '';
  }
  const output = convert(input);
  const rounded = Math.round(output * 1000) / 1000;
  return rounded.toString();
}

```

例如，`tryConvert('abc', toCelsius)` 回傳一個空的 string，而 `tryConvert('10.22', toFahrenheit)` 回傳 `'50.396'`。

Lifting State Up

目前兩個 `TemperatureInput` component 都有它們各自的 local state：

```

class TemperatureInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
    this.state = {temperature: ''};
  }

  handleChange(e) {
    this.setState({temperature: e.target.value});
  }

  render() {
    const temperature = this.state.temperature;
    // ...
  }
}

```

然而，我們想要這兩個輸入可以彼此同步。當我們更新攝氏輸入時，華氏輸入應該反映被轉換後的溫度，反之亦然。

在 React 中，共享 state 是透過將 state 搬移到需要它的 component 共同最近的 ancestor 來完成的。這被稱為「提升 state」。我們將從 `TemperatureInput` 移除 local state 並且搬移它到 `Calculator`。

如果 `Calculator` 擁有共享 state，它將成為目前兩個溫度輸入的「真相來源」。這可以說明它們兩者具有一致的值。由於這兩個 `TemperatureInput` component 的 prop 都是來自相同的 `Calculator` parent component，所以這兩個輸入會彼此同步。

讓我們來一步一步看它是如何執行的。

首先，我們將會把 `TemperatureInput` component 的 `this.state.temperature` 替換為 `this.props.temperature`。現在，讓我們假設 `this.props.temperature` 已經存在，雖然我們之後需要從 `Calculator` 傳遞它：

```
render() {
  // 先前: const temperature = this.state.temperature;
  const temperature = this.props.temperature;
  // ...
```

我們知道 `prop` 是唯讀的。當 `temperature` 在 local state 時，`TemperatureInput` 可以呼叫 `this.setState()` 來改變它。然而，現在 `temperature` prop 是來自它的 parent，`TemperatureInput` 無法控制它。

在 React 中，這通常透過讓 component 「被控制」來解決。就像 DOM `<input>` 同時接受 `value` 和 `onChange` prop，所以可以自訂 `TemperatureInput` 同時接受來自 `Calculator` parent component 的 `temperature` 和 `onTemperatureChange` prop。

現在，當 `TemperatureInput` 想要更新溫度時，它呼叫 `this.props.onTemperatureChange`：

```
handleChange(e) {
  // 先前: this.setState({temperature: e.target.value});
  this.props.onTemperatureChange(e.target.value);
  // ...
```

注意：

Component 中自訂的 `temperature` 或 `onTemperatureChange` prop 名稱並沒有特殊含義。我們可以將它們命名為任何名稱，像是 `value` 和 `onChange`，這是常見的慣例。

`onTemperatureChange` prop 與 `temperature` prop 將會由 `Calculator` parent component 提供。它將透過修改本身的 local state 來處理更改，因此會重新 render 兩個輸入與新的值。我們將快速地看一下 `Calculator` 的實作。

在深入改變 `Calculator` 之前，讓我們回顧先前對 `TemperatureInput` 的修改。我們移除了它的 local state，並且不讀取 `this.state.temperature`，我們現在讀取 `this.props.temperature`。當我們想要改變時不呼叫 `this.setState()`，我們現在呼叫 `this.props.onTemperatureChange()`，它是由 `Calculator` 提供的：

```
class TemperatureInput extends React.Component {
  constructor(props) {
    super(props);
    this.handleChange = this.handleChange.bind(this);
  }

  handleChange(e) {
    this.props.onTemperatureChange(e.target.value);
  }

  render() {
    const temperature = this.props.temperature;
    const scale = this.props.scale;
    return (
      <fieldset>
        <legend>Enter temperature in {scaleNames[scale]}:</legend>
```

```

        <input value={temperature}
               onChange={this.handleChange} />
    </fieldset>
);
}
}

```

現在讓我們轉回到 `Calculator` component。

我們將 store 目前輸入的 `temperature` 和 `scale` 在它的 local state。這是我們從輸入「提升」的 state，而且同時為輸入的「真相來源」。它是我們為了 render 兩個輸入而需要知道的最小表示資料。

例如，如果我們輸入 37 到攝氏輸入，`calculator` component 的 state 將會是：

```

{
  temperature: '37',
  scale: 'c'
}

```

如果我們之後更改華氏欄位為 212，`calculator` component 的 state 將會是：

```

{
  temperature: '212',
  scale: 'f'
}

```

我們可以 store 兩個輸入的值，但事實證明它是不需要的。它只要 store 最近修改的輸入值，以及溫度單位就夠了我們可以根據目前的 `temperature` 和 `scale` 來推斷其他輸入值。

輸入值會保持同步是因為它們的計算都是來自相同的 state：

```

class Calculator extends React.Component {
  constructor(props) {
    super(props);
    this.handleChangeCelsius = this.handleChangeCelsius.bind(this);
    this.handleChangeFahrenheit = this.handleChangeFahrenheit.bind(this);
    this.state = {temperature: '', scale: 'c'};
  }

  handleChangeCelsius(temperature) {
    this.setState({scale: 'c', temperature});
  }

  handleChangeFahrenheit(temperature) {
    this.setState({scale: 'f', temperature});
  }

  render() {
    const scale = this.state.scale;
    const temperature = this.state.temperature;
    const celsius = scale === 'f' ? tryConvert(temperature, toCelsius) :
      temperature;
    const fahrenheit = scale === 'c' ? tryConvert(temperature, toFahrenheit) :
      temperature;

    return (
      <div>
        <TemperatureInput

```

```

        scale="c"
        temperature={celsius}
        onTemperatureChange={this.handleCelsiusChange} />
    <TemperatureInput
        scale="f"
        temperature={fahrenheit}
        onTemperatureChange={this.handleFahrenheitChange} />
    <BoilingVerdict
        celsius={parseFloat(celsius)} />
</div>
);
}
}

```

[在 CodePen 上試試看吧！](#)

現在，無論輸入如何改變，在 `Calculator` 的 `this.state.temperature` 和 `this.state.scale` 都會被更新。其中一個輸入保留任何使用者的輸入，並且總是根據使用者的輸入重新計算另一個輸入值。

讓我們回顧一下當你編輯輸入時會發生什麼事情：

- React 在 DOM `<input>` 上呼叫被指定為 `onChange` 的函式。在我們的範例中，這是在 `TemperatureInput component` 內的 `handleChange` 方法。
- 在 `TemperatureInput component` 的 `handleChange` 方法呼叫 `this.props.onTemperatureChange()` 與新的期望值。它的 prop 包含 `onTemperatureChange`，是由 `Calculator` parent component 所提供的。
- 當它被 render 之前，`Calculator` 指定攝氏 `TemperatureInput` 的 `onTemperatureChange` 是 `Calculator` 的 `handleCelsiusChange` 方法，而華氏溫度的 `TemperatureInput` 的 `onTemperatureChange` 方法是 `Calculator` 的 `handleFahrenheitChange` 方法。因此根據我們編輯的輸入呼叫這兩個 `Calculator` 方法中的其中一個。
- 在這些方法中，`Calculator component` 要求 React 根據我們編輯的新輸入值和目前的溫度單位的輸入呼叫 `this.setState()` 來重新 render 本身。
- React 呼叫 `Calculator component` 的 `render` 方法來了解 UI 應該是怎麼樣子。根據目前溫度和溫度單位重新計算兩個輸入的值。溫度轉換會在這裡執行。
- 透過 `Calculator` 指定新的 prop，React 呼叫各個 `TemperatureInput component` 的 `render` 方法，它們應該了解 UI 是什麼樣子。
- React 呼叫 `BoilingVerdict component` 的 `render` 方法，以攝氏溫度做為 prop。
- React DOM 使用沸騰判定更新 DOM 並匹配所需的輸入值。我們剛剛編輯的輸入它接收目前的值，而另一個輸入被更新成轉換後的溫度。

每次更新都會執行相同的步驟，保持輸入的同步。

經驗學習

在 React 應用程式中，對於資料的變化只能有一個唯一的「真相來源」。通常來說，`state` 會優先被加入到需要 `render` 的 component。接著，如果其他的 component 也需要的話，你可以提升 `state` 到共同最靠近的 ancestor。你應該依賴[上至下的資料流](#)，而不是嘗試在不同 component 之間同步 `state`。

提升 `state` 涉及撰寫更多的「boilerplate」程式碼，而不是雙向綁定的方法，但它對於隔離和尋找 bug 時更加容易。由於任何 `state` 「存活」在一些 component 中，而且 component 本身可以改變它，bug 的產生大幅的減少。此外，你也可以實作任何自訂的邏輯來拒絕或轉換使用者的輸入。

如果某樣東西可以從 prop 或 state 被取得，它可能不應該在 state。例如，我們只 store 最後編輯的 `temperature` 和它的 `scale`，而不是 store `celsiusValue` 和 `fahrenheitValue`。其他輸入的值總是可以從它們的 `render()` 方法被計算出來。這讓我們可以清除或將四捨五入應用於另一個欄位，而不會在使用者輸入中失去任何精度。

當你在 UI 上看到一些錯誤時，你可以使用 [React Developer Tools](#) 來檢查 prop 並往 tree 的上方尋找，直到找到負責更新 state 的 component。這讓你可以追蹤到錯誤的來源：

The screenshot shows a user interface with two input fields. The first input field has the placeholder text "Enter temperature in Celsius:". The second input field has the placeholder text "Enter temperature in Fahrenheit:". Both fields are empty.

The water would not boil.

The screenshot shows the React Developer Tools interface. The left pane displays the component tree:

```

<> <Calculator>
  <> <div>
    <> <TemperatureInput scale="c" temperature="" onTemperatureChange="..."/>
    <> <TemperatureInput scale="f" temperature="" onTemperatureChange="..."/>
    <> <BoilingVerdict celsius=null>...</BoilingVerdict>
  </div>
</Calculator>

```

The right pane shows the component's props and state:

- Props**: An empty object.
- State**:
 - `scale: "c"`
 - `temperature: ""`

列表與 Key

首先，讓我們複習一下在 JavaScript 中如何改變列表。

在以下的程式碼中，我們使用 `map()` function 來接收 `numbers` array，並將其中的每個值乘以兩倍。我們將 `map()` 回傳的新 array 設定為變數 `doubled` 的值並印出：

```
const numbers = [1, 2, 3, 4, 5];
const doubled = numbers.map((number) => number * 2);
console.log(doubled);
```

這段程式碼在 `console` 中印出 `[2, 4, 6, 8, 10]`。

在 React 中，將 array 轉變成 element 列表幾乎是一樣的方式。

Render 多個 Component

你可以建立一系列的 element 並用大括號 `{}` 將它們包含在 JSX 裡面。

下面，我們會用 JavaScript 的 `map()` function 迭代 `numbers` array，我們每次都會回傳一個 `` element。最後，我們會把結果產生的 element array 設定為 `listItems`：

```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) =>
  <li>{number}</li>
);
```

接著，我們會把整個 `listItems` array 包含在一個 `` element 內：

```
<ul>{listItems}</ul>
```

[在 CodePen 上試試看吧！](#)

這段程式碼展示了一個數字列表，從 1 到 5。

基本列表 Component

一般來說，你會在一個 component 內 render 列表。

我們可以將上面的範例改寫為一個接收 `numbers` array 並輸出一個沒有排序的 element 列表的 component。

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <li>{number}</li>
  );
  return (
    <ul>{listItems}</ul>
  );
}
```

```
const numbers = [1, 2, 3, 4, 5];
const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<NumberList numbers={numbers} />);
```

當你執行這段程式碼時，你會收到一個關於你應該提供 key 給每一個列表項目的警告。「key」是當你在建立一個 element 列表時必須使用的特殊的 string attribute。在下一個段落中，我們將會討論其重要性。

讓我們為 `numbers.map()` 列表中的每個項目分配一個 `key`，並修正遺漏 key 的問題。

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <li key={number.toString()}>
      {number}
    </li>
  );
  return (
    <ul>{listItems}</ul>
  );
}
```

[在 CodePen 上試試看吧！](#)

Key

Key 幫助 React 分辨哪些項目被改變、增加或刪除。在 array 裡面的每個 element 都應該要有一個 key，如此才能給予每個 element 一個固定的身份：

```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) =>
  <li key={number.toString()}>
    {number}
  </li>
);
```

選擇 key 最佳的方法是在列表中使用唯一識別字串來區別 sibling 項目。通常，你會使用資料的 ID 作為 key：

```
const todoItems = todos.map((todo) =>
  <li key={todo.id}>
    {todo.text}
  </li>
);
```

當你 render 的項目沒有固定的 ID 且你也沒有更好的辦法時，你可以使用項目的索引做為 key：

```
const todoItems = todos.map((todo, index) =>
  // 請在項目沒有固定的 ID 時才這樣做
  <li key={index}>
    {todo.text}
  </li>
);
```

我們並不建議你使用索引作為 key，尤其如果項目的順序會改變的話。這會對效能產生不好的影響，也可能會讓 component state 產生問題。請參考 Robin Pokorny 這篇[深入剖析使用索引作為 key 的負面效應](#)一文。如果你選擇不明確分配 key 到列表項目時，React 預設將會使用索引作為 key。

如果你想了解更多，請參考[深度剖析 key 的必要性](#)一文。

用 Key 抽離 Component

Key 只有在周遭有 array 的情境中才有意義。

例如，如果你要抽離一個 `ListItem` component 的話，你應該把 key 放在 array 裡的 `<ListItem />` element 上，而不是把它放在 `ListItem` 裡面的 `` element 上。

範例：Key 的錯誤使用方式

```
function ListItem(props) {
  const value = props.value;
  return (
    // 錯！你不需要在這裡指出 key :
    <li key={value.toString()}>
      {value}
    </li>
  );
}

function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    // 錯！你應該要在這裡指出 key :
    <ListItem value={number} />
  );
  return (
    <ul>
      {listItems}
    </ul>
  );
}
```

範例：Key 的正確使用方式

```
function ListItem(props) {
  // 正確！你不需要在這裡指出 key :
  return <li>{props.value}</li>;
}

function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    // 正確！Key 應該在 array 內被指定。
    <ListItem key={number.toString()} value={number} />
  );
  return (
    <ul>
      {listItems}
    </ul>
  );
}
```

[在 CodePen 上試試看吧！](#)

一個好的經驗法則是，在 `map()` 呼叫中的每個 element 都會需要 key。

Key 必須在 Sibling 中是唯一的

在 array 中使用的 key 應該要是唯一的值。然而，它們不必在全域中唯一。當我們產生兩個不同的 array 時，我們仍然可以使用相同的 key：

```
function Blog(props) {
  const sidebar = (
    <ul>
      {props.posts.map((post) =>
        <li key={post.id}>
          {post.title}
        </li>
      )}
    </ul>
  );
  const content = props.posts.map((post) =>
    <div key={post.id}>
      <h3>{post.title}</h3>
      <p>{post.content}</p>
    </div>
  );
  return (
    <div>
      {sidebar}
      <hr />
      {content}
    </div>
  );
}

const posts = [
];

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Blog posts={posts} />);
```

[在 CodePen 上試試看吧！](#)

Key 的功能是提示 React，但它們不會被傳遞到你的 component。如果你在 component 中需要同樣的值，你可以直接把這個值用一個不同的名稱作為 prop 傳下去：

```
const content = posts.map((post) =>
  <Post
    key={post.id}
    id={post.id}
    title={post.title} />
);
```

在上面的例子中，`Post` component 可以讀取 `props.id`，但不能讀取 `props.key`。

在 JSX 中嵌入 map()

在上面的例子中，我們宣告了另一個 `listItems` 變數並把它包含在 JSX 中：

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <ListItem key={number.toString()} value={number} />
  );
  return (
    <ul>
      {listItems}
    </ul>
  );
}
```

JSX 讓你在大括號中 [嵌入任何表達式](#)，所以我們能夠 inline `map()` 的結果：

```
function NumberList(props) {
  const numbers = props.numbers;
  return (
    <ul>
      {numbers.map((number) =>
        <ListItem key={number.toString()} value={number} />
      )}
    </ul>
  );
}
```

在 [CodePen](#) 上試試看吧！

有時候這會產生更乾淨的程式碼，但這種風格也可能被濫用。就像 JavaScript 一樣，是否要將變數抽取出來以增加可讀性完全是看你的決定。請記得，如果 `map()` 的程式碼層級變得過度巢狀，也許就是使用[抽離 component](#)的時候了。

Optimizing Performance

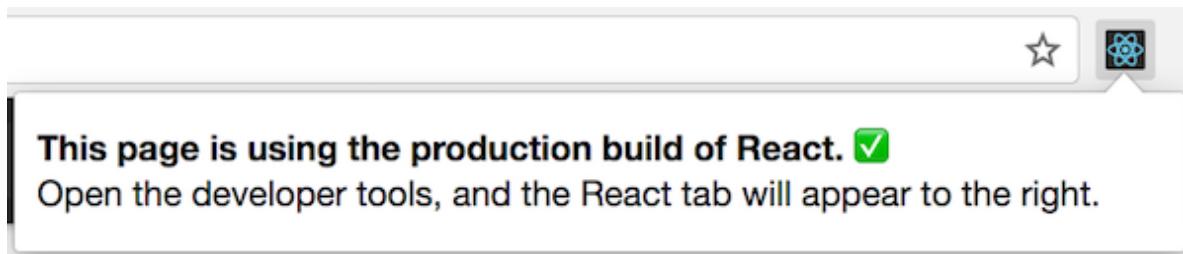
Internally, React uses several clever techniques to minimize the number of costly DOM operations required to update the UI. For many applications, using React will lead to a fast user interface without doing much work to specifically optimize for performance. Nevertheless, there are several ways you can speed up your React application.

Use the Production Build

If you're benchmarking or experiencing performance problems in your React apps, make sure you're testing with the minified production build.

By default, React includes many helpful warnings. These warnings are very useful in development. However, they make React larger and slower so you should make sure to use the production version when you deploy the app.

If you aren't sure whether your build process is set up correctly, you can check it by installing [React Developer Tools for Chrome](#). If you visit a site with React in production mode, the icon will have a dark background:



If you visit a site with React in development mode, the icon will have a red background:



It is expected that you use the development mode when working on your app, and the production mode when deploying your app to the users.

You can find instructions for building your app for production below.

Create React App

If your project is built with [Create React App](#), run:

```
npm run build
```

This will create a production build of your app in the `build/` folder of your project.

Remember that this is only necessary before deploying to production. For normal development, use `npm start`.

Single-File Builds

We offer production-ready versions of React and React DOM as single files:

```
<script src="https://unpkg.com/react@18/umd/react.production.min.js"></script>
<script src="https://unpkg.com/react-dom@18/umd/react-dom.production.min.js">
</script>
```

Remember that only React files ending with `.production.min.js` are suitable for production.

Brunch

For the most efficient Brunch production build, install the `terser-brunch` plugin:

```
# If you use npm
npm install --save-dev terser-brunch

# If you use Yarn
yarn add --dev terser-brunch
```

Then, to create a production build, add the `-p` flag to the `build` command:

```
brunch build -p
```

Remember that you only need to do this for production builds. You shouldn't pass the `-p` flag or apply this plugin in development, because it will hide useful React warnings and make the builds much slower.

Browserify

For the most efficient Browserify production build, install a few plugins:

```
# If you use npm
npm install --save-dev envify terser uglifyify

# If you use Yarn
yarn add --dev envify terser uglifyify
```

To create a production build, make sure that you add these transforms (**the order matters**):

- The `envify` transform ensures the right build environment is set. Make it global (`-g`).

- The `uglifyify` transform removes development imports. Make it global too (`-g`).
- Finally, the resulting bundle is piped to `terser` for mangling ([read why](#)).

For example:

```
browserify ./index.js \
-g [ envify --NODE_ENV production ] \
-g uglifyify \
| terser --compress --mangle > ./bundle.js
```

Remember that you only need to do this for production builds. You shouldn't apply these plugins in development because they will hide useful React warnings, and make the builds much slower.

Rollup

For the most efficient Rollup production build, install a few plugins:

```
# If you use npm
npm install --save-dev rollup-plugin-commonjs rollup-plugin-replace rollup-plugin-terser

# If you use Yarn
yarn add --dev rollup-plugin-commonjs rollup-plugin-replace rollup-plugin-terser
```

To create a production build, make sure that you add these plugins (**the order matters**):

- The `replace` plugin ensures the right build environment is set.
- The `commonjs` plugin provides support for CommonJS in Rollup.
- The `terser` plugin compresses and mangles the final bundle.

```
plugins: [
  // ...
  require('rollup-plugin-replace')({
    'process.env.NODE_ENV': JSON.stringify('production')
  }),
  require('rollup-plugin-commonjs')(),
  require('rollup-plugin-terser')(),
  // ...
]
```

For a complete setup example [see this gist](#).

Remember that you only need to do this for production builds. You shouldn't apply the `terser` plugin or the `replace` plugin with `'production'` value in development because they will hide useful React warnings, and make the builds much slower.

webpack

Note:

If you're using Create React App, please follow [the instructions above](#).
This section is only relevant if you configure webpack directly.

Webpack v4+ will minify your code by default in production mode.

```
const TerserPlugin = require('terser-webpack-plugin');

module.exports = {
  mode: 'production',
  optimization: {
    minimizer: [new TerserPlugin({ /* additional options here */ })],
  },
};
```

You can learn more about this in [webpack documentation](#).

Remember that you only need to do this for production builds. You shouldn't apply `TerserPlugin` in development because it will hide useful React warnings, and make the builds much slower.

Profiling Components with the DevTools Profiler

`react-dom` 16.5+ and `react-native` 0.57+ provide enhanced profiling capabilities in DEV mode with the React DevTools Profiler. An overview of the Profiler can be found in the blog post "[Introducing the React Profiler](#)". A video walkthrough of the profiler is also [available on YouTube](#).

If you haven't yet installed the React DevTools, you can find them here:

- [Chrome Browser Extension](#)
- [Firefox Browser Extension](#)
- [Standalone Node Package](#)

Note

A production profiling bundle of `react-dom` is also available as `react-dom/profiling`. Read more about how to use this bundle at fb.me/react-profiling

Note

Before React 17, we use the standard [User Timing API](#) to profile components with the chrome performance tab. For a more detailed walkthrough, check out [this article by Ben Schwarz](#).

Virtualize Long Lists

If your application renders long lists of data (hundreds or thousands of rows), we recommend using a technique known as "windowing". This technique only renders a small subset of your rows at any given time, and can dramatically reduce the time it takes to re-render the components as well as the number of DOM nodes created.

[react-window](#) and [react-virtualized](#) are popular windowing libraries. They provide several reusable components for displaying lists, grids, and tabular data. You can also create your own windowing component, like [Twitter did](#), if you want something more tailored to your application's specific use case.

Avoid Reconciliation

React builds and maintains an internal representation of the rendered UI. It includes the React elements you return from your components. This representation lets React avoid creating DOM nodes and accessing existing ones beyond necessity, as that can be slower than operations on JavaScript objects. Sometimes it is referred to as a "virtual DOM", but it works the same way on React Native.

When a component's props or state change, React decides whether an actual DOM update is necessary by comparing the newly returned element with the previously rendered one. When they are not equal, React will update the DOM.

Even though React only updates the changed DOM nodes, re-rendering still takes some time. In many cases it's not a problem, but if the slowdown is noticeable, you can speed all of this up by overriding the lifecycle function `shouldComponentUpdate`, which is triggered before the re-rendering process starts. The default implementation of this function returns `true`, leaving React to perform the update:

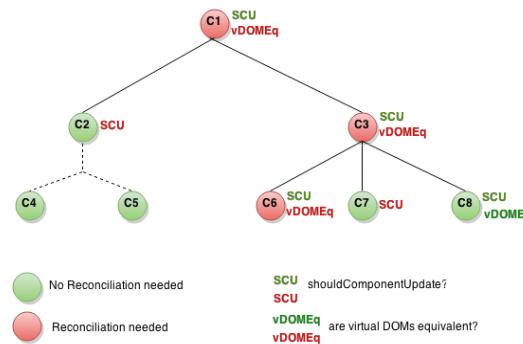
```
shouldComponentUpdate(nextProps, nextState) {
  return true;
}
```

If you know that in some situations your component doesn't need to update, you can return `false` from `shouldComponentUpdate` instead, to skip the whole rendering process, including calling `render()` on this component and below.

In most cases, instead of writing `shouldComponentUpdate()` by hand, you can inherit from `React.PureComponent`. It is equivalent to implementing `shouldComponentUpdate()` with a shallow comparison of current and previous props and state.

shouldComponentUpdate In Action

Here's a subtree of components. For each one, `SCU` indicates what `shouldComponentUpdate` returned, and `vDOMEq` indicates whether the rendered React elements were equivalent. Finally, the circle's color indicates whether the component had to be reconciled or not.



Since `shouldComponentUpdate` returned `false` for the subtree rooted at C2, React did not attempt to render C2, and thus didn't even have to invoke `shouldComponentUpdate` on C4 and C5.

For C1 and C3, `shouldComponentUpdate` returned `true`, so React had to go down to the leaves and check them. For C6 `shouldComponentUpdate` returned `true`, and since the rendered elements weren't equivalent React had to update the DOM.

The last interesting case is C8. React had to render this component, but since the React elements it returned were equal to the previously rendered ones, it didn't have to update the DOM.

Note that React only had to do DOM mutations for C6, which was inevitable. For C8, it bailed out by comparing the rendered React elements, and for C2's subtree and C7, it didn't even have to compare the elements as we bailed out on `shouldComponentUpdate`, and `render` was not called.

Examples

If the only way your component ever changes is when the `props.color` or the `state.count` variable changes, you could have `shouldComponentUpdate` check that:

```
class CounterButton extends React.Component {
  constructor(props) {
    super(props);
    this.state = {count: 1};
  }

  shouldComponentUpdate(nextProps, nextState) {
    if (this.props.color !== nextProps.color) {
      return true;
    }
    if (this.state.count !== nextState.count) {
      return true;
    }
    return false;
  }

  render() {
    return (
      <button
        color={this.props.color}
        onClick={() => this.setState(state => ({count: state.count + 1}))}>
        Count: {this.state.count}
      </button>
    );
  }
}
```

```

        </button>
    );
}
}

```

In this code, `shouldComponentUpdate` is just checking if there is any change in `props.color` or `state.count`. If those values don't change, the component doesn't update. If your component got more complex, you could use a similar pattern of doing a "shallow comparison" between all the fields of `props` and `state` to determine if the component should update. This pattern is common enough that React provides a helper to use this logic - just inherit from `React.PureComponent`. So this code is a simpler way to achieve the same thing:

```

class CounterButton extends React.PureComponent {
  constructor(props) {
    super(props);
    this.state = {count: 1};
  }

  render() {
    return (
      <button
        color={this.props.color}
        onClick={() => this.setState(state => ({count: state.count + 1}))}>
        Count: {this.state.count}
      </button>
    );
  }
}

```

Most of the time, you can use `React.PureComponent` instead of writing your own `shouldComponentUpdate`. It only does a shallow comparison, so you can't use it if the props or state may have been mutated in a way that a shallow comparison would miss.

This can be a problem with more complex data structures. For example, let's say you want a `ListOfWords` component to render a comma-separated list of words, with a parent `WordAdder` component that lets you click a button to add a word to the list. This code does *not* work correctly:

```

class ListOfWords extends React.PureComponent {
  render() {
    return <div>{this.props.words.join(',')}</div>;
  }
}

class WordAdder extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      words: ['marklar']
    };
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    // This section is bad style and causes a bug
    const words = this.state.words;
    words.push('marklar');
  }
}

```

```

    this.setState({words: words});
}

render() {
  return (
    <div>
      <button onClick={this.handleClick} />
      <ListOfWords words={this.state.words} />
    </div>
  );
}
}

```

The problem is that `PureComponent` will do a simple comparison between the old and new values of `this.props.words`. Since this code mutates the `words` array in the `handleClick` method of `WordAdder`, the old and new values of `this.props.words` will compare as equal, even though the actual words in the array have changed. The `ListOfWords` will thus not update even though it has new words that should be rendered.

The Power Of Not Mutating Data

The simplest way to avoid this problem is to avoid mutating values that you are using as props or state. For example, the `handleClick` method above could be rewritten using `concat` as:

```

handleClick() {
  this.setState(state => ({
    words: state.words.concat(['marklar'])
  }));
}

```

ES6 supports a [spread syntax](#) for arrays which can make this easier. If you're using Create React App, this syntax is available by default.

```

handleClick() {
  this.setState(state => ({
    words: [...state.words, 'marklar'],
  }));
}

```

You can also rewrite code that mutates objects to avoid mutation, in a similar way. For example, let's say we have an object named `colormap` and we want to write a function that changes `colormap.right` to be `'blue'`. We could write:

```

function updateColorMap(colormap) {
  colormap.right = 'blue';
}

```

To write this without mutating the original object, we can use `Object.assign` method:

```

function updateColorMap(colormap) {
  return Object.assign({}, colormap, {right: 'blue'});
}

```

`updateColorMap` now returns a new object, rather than mutating the old one. `Object.assign` is in ES6 and requires a polyfill.

[Object spread syntax](#) makes it easier to update objects without mutation as well:

```
function updateColorMap(colormap) {  
  return {...colormap, right: 'blue'};  
}
```

This feature was added to JavaScript in ES2018.

If you're using Create React App, both `Object.assign` and the object spread syntax are available by default.

When you deal with deeply nested objects, updating them in an immutable way can feel convoluted. If you run into this problem, check out [Immer](#) or [immutability-helper](#). These libraries let you write highly readable code without losing the benefits of immutability.

Portal

Portal 提供一個優秀方法來讓 child 可以 render 到 parent component DOM 樹以外的 DOM 節點。

`ReactDOM.createPortal(child, container)`

第一個參數 (`child`) 是任何可 render 的 React child，例如 element、string 或者 fragment。第二個參數 (`container`) 則是一個 DOM element。

使用方式

通常當 component 的 render 方法回傳一個 element 時，此 element 會作為 child 被 mount 進最接近的 parent 節點中：

```
render() {
  // React mount 一個新的 div 並將 children render 進去
  return (
    <div>
      {this.props.children}
    </div>
  );
}
```

然而在某些狀況下，將 child 插入不同位置的 DOM 內十分好用：

```
render() {
  // React *不會* 建立新的 div。它會將 children render 進 `domNode` 中。
  // `domNode` 可以是任何在隨意位置的合法 DOM node。
  return ReactDOM.createPortal(
    this.props.children,
    domNode
  );
}
```

一個典型的 portal 使用案例是，當 parent component 有 `overflow: hidden` 或者 `z-index` 的樣式時，卻仍需要 child 在視覺上「跳出」其容器的狀況。例如 dialog、hovercard 與 tooltip 都屬於此案例。

注意：

當使用 portal 時，請留意控管鍵盤 focus 對於無障礙功能會變得非常重要。

使用跳窗 dialog 時，應確保每個人都可以依照 [WAI-ARIA Modal 開發規範](#) 定義的方式與其互動。

[在 CodePen 上試試看吧！](#)

透過 Portal 進行 Event Bubbling

雖然 portal 可以被放置在 DOM tree 中的任何位置，但 portal 的其他行為與一般的 React child 別無二致。像是 context 等功能的運作方式並不會因為 child 是 portal 而有所不同，因為不論 portal 在 DOM tree 中的位置為何，它都存在於 *React tree* 中。

相同的行為也包括 event bubbling。一個在 portal 內觸發的 event 會傳遞到涵蓋它的 *React tree* 祖先中，就算涵蓋它的那些 element 並不是 DOM tree 上的祖先。假設存在以下 HTML 結構：

```
<html>
  <body>
    <div id="app-root"></div>
    <div id="modal-root"></div>
  </body>
</html>
```

一個在 `#app-root` 中的 `Parent` component 可以捕捉從 `sibling` 節點 `#modal-root` bubble 上來且未被接收過的 event。

```
// 這兩個 container 是 DOM 上的 sibling
const appRoot = document.getElementById('app-root');
const modalRoot = document.getElementById('modal-root');

class Modal extends React.Component {
  constructor(props) {
    super(props);
    this.el = document.createElement('div');
  }

  componentDidMount() {
    // Portal element 會在 Modal 的 children 被
    // mount 之後才插入 DOM tree 中，這代表 children
    // 會被 mount 在一個分離的 DOM 節點上。如果一個
    // child component 需要在 mount 結束時馬上連接到 DOM tree 中，
    // 例如測量一個 DOM node，或者在子節點中使用 'autoFocus' 等狀況，
    // 則應將 state 加入 Modal 中，並只在 Modal 插入 DOM tree 後
    // 才 render children。
    modalRoot.appendChild(this.el);
  }

  componentWillUnmount() {
    modalRoot.removeChild(this.el);
  }

  render() {
    return ReactDOM.createPortal(
      this.props.children,
      this.el
    );
  }
}

class Parent extends React.Component {
  constructor(props) {
    super(props);
    this.state = {clicks: 0};
    this.handleClick = this.handleClick.bind(this);
  }
```

```

handleClick() {
  // 這會在 Child 中的 button 被點擊時觸發並更新 Parent 的 state,
  // 就算 Child 的 button 並不是 DOM 中的直接後代。
  this.setState(state => ({
    clicks: state.clicks + 1
  }));
}

render() {
  return (
    <div onClick={this.handleClick}>
      <p>Number of clicks: {this.state.clicks}</p>
      <p>
        Open up the browser DevTools
        to observe that the button
        is not a child of the div
        with the onClick handler.
      </p>
      <Modal>
        <Child />
      </Modal>
    </div>
  );
}

function Child() {
  // 這個 button 中的 click event 會被 bubble 到 parent 中,
  // 因為這邊並沒有定義 'onClick' attribute
  return (
    <div className="modal">
      <button>Click</button>
    </div>
  );
}

const root = ReactDOM.createRoot(appRoot);
root.render(<Parent />);

```

[在 CodePen 上試試看吧！](#)

Parent component 可以捕捉從 portal bubble 上來的 event 能使開發具有不直接依賴於 portal 的更彈性化抽象性。舉例來說，如果你 render 了一個 `<Modal />` component，則不論 Modal 是否是使用 portal 實作，它的 parent 皆可以捕捉到其 event。

[Go to TOC](#)

不使用 ES6 開發 React

通常你會把 React component 定義成一個 JavaScript class：

```
class Greeting extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

如果你還沒開始使用 ES6 的話，則可以考慮用 `create-react-class` 來代替：

```
var createReactClass = require('create-react-class');
var Greeting = createReactClass({
  render: function() {
    return <h1>Hello, {this.props.name}</h1>;
  }
});
```

ES6 class 與 `createReactClass()` 的 API 十分相似，只有以下幾個不同處。

宣告預設 Props

使用 function 與 ES6 class 語法時，`defaultProps` 會被定義為 component 上的一個屬性：

```
class Greeting extends React.Component {
  // ...
}

Greeting.defaultProps = {
  name: 'Mary'
};
```

而使用 `createReactClass()` 語法時，你則需要在傳遞的物件上定義 `getDefaultsProps()` 方法：

```
var Greeting = createReactClass({
  getDefaultsProps: function() {
    return {
      name: 'Mary'
    };
  },
  // ...
});
```

設定初始 State

在 ES6 class 語法中，你可以藉由在 `constructor` 中設定 `this.state` 來定義初始 state：

```
class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {count: props.initialCount};
  }
  // ...
}
```

而使用 `createReactClass()` 語法時，你則需要另外提供一個會回傳初始 state 的 `getInitialState` 方法：

```
var Counter = createReactClass({
  getInitialState: function() {
    return {count: this.props.initialCount};
  },
  // ...
});
```

自動綁定

在使用 ES6 class 宣告 React component 時，其方法的語義與一般的 ES6 class 相同。也就是說，這些方法不會自動綁定 `this` 到 instance 上。你會需要明確地在 `constructor` 中使用 `.bind(this)` 來綁定 `this`：

```
class SayHello extends React.Component {
  constructor(props) {
    super(props);
    this.state = {message: 'Hello!'};
    // 這行很重要！
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    alert(this.state.message);
  }

  render() {
    // 因為 `this.handleClick` 已經被綁定了，所以我們才可以把它當作 event handler 使用。
    return (
      <button onClick={this.handleClick}>
        Say hello
      </button>
    );
  }
}
```

而使用 `createReactClass()` 時就不需要這麼做，因為它會自動綁定所有方法：

```
var SayHello = createReactClass({
  getInitialState: function() {
    return {message: 'Hello!'};
  },

  handleClick: function() {
    alert(this.state.message);
  },

  render: function() {
    return (
      <button onClick={this.handleClick}>
```

```

    Say hello
  </button>
);
}
});

```

這代表使用 ES6 class 時會需要為 event handler 多寫一些 boilerplate 程式碼，但這種寫法的好處是在大型的應用程式中會有稍微好一點的效能。

如果寫 boilerplate 程式碼對你來說實在是很沒有吸引力的話，你可以使用 [ES2022 Class Properties](#) 語法：

```

class SayHello extends React.Component {
  constructor(props) {
    super(props);
    this.state = {message: 'Hello!'};
  }

  handleClick = () => {
    alert(this.state.message);
  };

  render() {
    return (
      <button onClick={this.handleClick}>
        Say hello
      </button>
    );
  }
}

```

如果你想要保險一點，你有以下幾個選擇：

- 在 constructor 中綁定方法。
- 使用 arrow function，例如 `onClick={(e) => this.handleClick(e)}`。
- 繼續使用 `createReactClass`。

Mixin

注意：

ES6 並沒有支援任何 mixin 語法。因此當你在 React 中使用 ES6 class 時也不支援使用 mixin。

我們也發現在程式中使用 **mixin** 會造成很多問題，[因此不建議在新的程式碼中使用 mixin](#)。

此段落內容僅供參考。

有時候完全不同的 component 間可能會共有相同的功能。這種狀況有時候被稱為 [橫切關注點](#) 問題。`createReactClass` 可以讓你使用舊有的 `mixins` 機制來解決此問題。

一個常見的使用情境是，一個 component 想要每隔一段時間就更新自己一次。使用 `setInterval()` 就可以很容易的實現這個功能，但重要的是要在不需要 `setInterval()` 之後把它取消掉以保留記憶體空間。React 提供 [生命週期方法](#)讓你知道一個 component 什麼時候會被建立或銷毀。讓我們用這些方法建立一個簡易 mixin，以提供簡單的 `setInterval()` 功能，並讓這個 `setInterval()` 在你的 component 被銷毀時自動清除掉。

```
var SetIntervalMixin = {
  componentWillMount: function() {
    this.intervals = [];
  },
  setInterval: function() {
    this.intervals.push(setInterval.apply(null, arguments));
  },
  componentWillUnmount: function() {
    this.intervals.forEach(clearInterval);
  }
};

var createReactClass = require('create-react-class');

var TickTock = createReactClass({
  mixins: [SetIntervalMixin], // 使用 mixin
  getInitialState: function() {
    return {seconds: 0};
  },
  componentDidMount: function() {
    this.setInterval(this.tick, 1000); // 呼叫一個 mixin 中的方法
  },
  tick: function() {
    this.setState({seconds: this.state.seconds + 1});
  },
  render: function() {
    return (
      <p>
        React has been running for {this.state.seconds} seconds.
      </p>
    );
  }
});

const root = ReactDOM.createRoot(document.getElementById('example'));
root.render(<TickTock />);
```

如果一個 component 使用了複數個 mixin，且其中的多個 mixin 同時定義了相同的生命週期方法（例如，多個 mixin 都想要在 component 被銷毀時做一些清除的動作），則所有的生命週期方法都保證會被呼叫到。這些 Mixin 中的方法會依照它們的排列順序依序執行，並會在最後呼叫 component 上的對應方法。

[Go to TOC](#)

不使用 JSX 開發 React

JSX 對於使用 React 並不是不可或缺的。當你不想在開發環境中設置編譯時，不使用 JSX 開發 React 格外方便。

每個 JSX 元素都只是呼叫 `React.createElement(component, props, ...children)` 的語法糖。所有任何你能用 JSX 做的事，你都能用純 JavaScript 做到。

比如，這段用 JSX 寫成的編碼：

```
class Hello extends React.Component {
  render() {
    return <div>Hello {this.props.toWhat}</div>;
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Hello toWhat="World" />);
```

可以編譯成這段沒有 JSX 的程式碼：

```
class Hello extends React.Component {
  render() {
    return React.createElement('div', null, `Hello ${this.props.toWhat}`);
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(React.createElement(Hello, {toWhat: 'World'}, null));
```

如果你想看更多 JSX 是如何轉換成 JavaScript 的範例，你可以嘗試[線上 Babel 編譯器](#)。

`Component` 可以是一個字串、可以是 `React.Component` 的 subclass，或是個 plain function。

如果你已經疲於不斷重複寫 `React.createElement`，一個常見的方式是賦予一個縮寫：

```
const e = React.createElement;

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(e('div', null, 'Hello World'));
```

如果你使用這個 `React.createElement` 縮寫的格式，他可以跟不使用 JSX 開發 React 一樣方便。

此外，你也可以參考社群專案像是 `react-hyperscript` 和 `hyperscript-helpers`，這些專案提供了更為簡潔的語法。

Reconciliation

React 提供聲明式 (declarative) 的 API，讓開發者在使用 React 時，可以不用關注每次 Component 更新時底層有什麼改變。這讓開發應用程式簡單許多，但也可能讓開發者對 React 底層的相關實作不夠瞭解。因此在這篇文章中描述了實作 React 底層中「diffing」演算法時，我們採取什麼策略讓 component 的更新是可預測的，同時可以滿足要求高效能的應用程式。

動機

在使用 React 時，每次呼叫 `render()` 函式，我們都可以當成是建立了一顆由 React element 構成的樹狀結構。而在每一次有 state 或 props 更新時，`render()` 函式就會回傳一顆不同的 tree。因此，React 需要判斷如何有效率的把 UI 從舊的 tree 更新成新的 tree。

對於這個「如何用最少操作去將舊的 tree 轉換成新的 tree」的演算法問題有一些通用的解法，但即使是目前[最先進的演算法](#)都還需要 $O(n^3)$ 的時間複雜度 (n 為 tree 中 element 的數量)。

假設 React 使用這種演算法，則呈現 1000 個 element 需要 10 億次的比較。因為這個比較成本實在太高，所以 React 在以下兩個假設下採用了一個 $O(n)$ 的啓發式 (heuristic) 演算法：

1. 兩個不同類型的 element 會產生出不同的 tree
2. 開發者可以通過 `key` prop 來指出哪些子 element 在不同的 render 下能保持不變；

而事實上，以上假設在幾乎所有實務上會出現的場景都成立。

Diffing 演算法

當比對兩顆 tree 時，React 首先比較兩棵 tree 的 root element。不同類型的 root element 會有不同的處理方式。

比對不同類型的 Element

當比對的兩個 root element 為不同類型的元素時，React 會將原有的 tree 整顆拆掉並且重新建立起新的 tree。例如，當一個元素從 `<a>` 變成 ``、從 `<Article>` 變成 `<Comment>`、或從 `<Button>` 變成 `<div>` 時，都會觸發一個完整的重建流程。

當拆掉一顆 tree 時，舊的 DOM 節點會被銷毀，且該 component instance 會執行 `componentWillUnmount()` 函式。當建立一顆新的 tree 時，新建立的 DOM 節點會被插入到 DOM 中，且該 component instance 會依次執行 `UNSAFE_componentWillMount()` 與 `componentDidMount()` 方法。而所有跟之前舊的 tree 所關聯的 state 也會被銷毀。

任何在 root 以下的 component 也會被 unmount，它們的狀態會被銷毀。例如，當比對以下變更時：

```
<div>
  <Counter />
</div>

<span>
  <Counter />
</span>
```

React 會 destroy 舊的 Counter 並且重新建立一個新的。

筆記：

這個方法已經被認為已過時，你在寫新程式應避免使用他們：

- UNSAFE_componentWillMount()

比對同一類型的 DOM Element

當比對兩個相同類型的 React element 時，React 會保留 DOM 節點，只比對及更新有改變的 attribute。例如：

```
<div className="before" title="stuff" />
<div className="after" title="stuff" />
```

透過比對這兩個 element，React 知道只需要修改 DOM 節點上的 className。

當更新 style 時，React 僅更新有所更變的屬性。例如：

```
<div style={{color: 'red', fontWeight: 'bold'}} />
<div style={{color: 'green', fontWeight: 'bold'}} />
```

透過比對這兩個 element，React 知道只需要修改 color 的樣式，而不需修改 fontWeight。

在處理完目前節點之後，React 會繼續對 children 進行遞迴處理。

比對同類型的 Component Element

當一個 component 更新時，該 component 的 instance 保持不變，這樣 state 能夠被保留在不同次的 render 中。React 會更新該 component instance 的 props 以跟最新的 element 保持一致，並且呼叫該 instance 的 UNSAFE_componentWillReceiveProps()、UNSAFE_componentWillUpdate() 和 componentDidUpdate() 方法。

接下來，該 instance 會再呼叫 render() 方法，而 diff 算法將會遞迴處理舊的結果以及新的結果。

筆記：

下列方法已過時，你在寫新程式應避免使用他們：

- UNSAFE_componentWillUpdate()
- UNSAFE_componentWillReceiveProps()

對 Children 進行遞迴處理

在預設條件下，當遞迴處理 DOM 節點的 children 時，React 只會同時遍歷兩個 children 的 array，並在發現差異時，產生一個 mutation。

例如，當在 children 的 array 尾端新增一個 element 時，在這兩個 tree 之間的轉換效果很好：

```
<ul>
  <li>first</li>
  <li>second</li>
</ul>

<ul>
  <li>first</li>
  <li>second</li>
  <li>third</li>
</ul>
```

React 會先匹配兩個 `first` 對應的 tree，然後匹配第二個元素 `second` 對應的 tree，最後插入第三個元素 `third` 的 tree。

如果只是單純的實作，則在 array 開頭插入新元素會讓效能變差。例如，在兩個 tree 之間的轉換效果很差：

```
<ul>
  <li>Duke</li>
  <li>Villanova</li>
</ul>

<ul>
  <li>Connecticut</li>
  <li>Duke</li>
  <li>Villanova</li>
</ul>
```

這個情況下，React 會針對每個 child 都進行 mutate，而不是讓兩個相同的 `Duke` 和 `Villanova` subtree 不參與 mutate。這種低效率的情況下可能會帶來效能問題。

Keys

為了解決以上問題，React 提供了 key 屬性。當 children 擁有 key 屬性時，React 使用 key 來匹配原有 tree 上的 children 以及後續 tree 的 children。例如，以下範例在新增 key 屬性之後，可以改善在上個例子中發生的效能問題：

```
<ul>
  <li key="2015">Duke</li>
  <li key="2016">Villanova</li>
</ul>

<ul>
  <li key="2014">Connecticut</li>
  <li key="2015">Duke</li>
  <li key="2016">Villanova</li>
</ul>
```

現在 React 知道只有帶著 '2014' key 的 element 是新的，而帶著 '2015' 以及 '2016' key 的 element 只是相對位置移動了。

實際上，找出一個 key 通常並不困難。你要顯示的 element 可能已經具有唯一的 ID 了，這個 key 可能是來自於你的資料：

```
<li key={item.id}>{item.name}</li>
```

當以上情況不成立時，你可以新增一個 ID 字串到你的 Model 中，或者利用一部分內容作為 hash 來產生一個 key。這個 key 不需要是全域唯一，但在 array 中需要保持唯一。

最後，你也可以使用元素在 array 中的索引值作為 key。這個方法在元素不進行重新排序時比較合適，但如果有序修改，diff 就會變慢。

當使用 array 索引值作為 key 的 component 進行重新排序時，component state 可能會遇到一些問題。由於 component instance 是基於它們的 key 來決定是否更新以及重複使用，如果 key 是一個索引值，那麼修改順序時會修改目前的 key，導致 component 的 state（例如不受控制輸入框）可能相互篡改導致無法預期的變動。

下面是一個在 Codepen 上的範例，[示範使用索引值作為 key 時導致的問題](#)，以及[一個修復了重新排列、排序、以及在 array 開頭插入的問題，且不使用索引值作為 key 的版本](#)。

權衡

請記得 Reconciliation 只是一個實作細節，即使 React 在每個動作後之後對整個應用進行重新 render，得到的最終結果也會和進行 Reconciliation 後是一樣的。在這個情形下，重新 render 表示在所有 component 內呼叫 `render`，這不代表 React 會 `unmount` 或重新 `mount` 它們。React 只會基於以上提到的規則來決定如何進行差異的合併。

我們會定期改善啟發式演算法，讓常見的使用場景能夠更有效率地執行。在目前的實作中，可以理解為一棵 subtree 能在其相同階層的兄弟之間移動，但不能移動到其他位置。如果移動到其他位置，則演算法會重新 render 整棵 subtree。

由於 React 依賴啟發式的演算法，因此當以下假設沒有得到滿足，效能將會有所影響。

1. 該演算法不會嘗試匹配不同 component 類型的 subtree。如果你發現你在兩種不同類型的 component 中切換，但輸出非常相似的內容，建議把它們改成同一類型。實際上，我們沒有發現在改成同一種類型後會發生問題。
2. Key 應該具有穩定、可預測、以及 array 內唯一的特質。不穩定的 key（例如透過 `Math.random()` 隨機生成的）會導致許多 component instance 和 DOM 節點被不必要的重新建立，這可能導致效能低下和 child component 中的 state 丟失。

DOM Elements

為了能兼具效能與跨瀏覽器的兼容性，React 實作了一套獨立於瀏覽器的 DOM 系統，我們藉由這個機會解決瀏覽器 DOM 的美中不足之處。

在 React 裡，所有 DOM property 和 attribute（包括 event handler）都應該以 camelCase 命名。舉例來說，HTML 的 attribute `tabindex` 在 React 中對應到 `tabIndex`。`aria-*` 和 `data-*` attribute 則是例外，需要保持全部小寫。舉例來說，`aria-label` 保持原樣即可。

Attribute 相異之處

有些 attribute 在 React 和 HTML 之間運作的方式略有不同：

`checked`

當 `<input>` 的 type 是 `checkbox` 或是 `radio` 時，可以使用 `checked` 這個 attribute 來設定 component 是否被選取。這對建立 controlled component 很有幫助。`defaultChecked` 則是使用在 uncontrolled component，在初始 mount 設定 component 是否被選取。

`className`

要指定一個 CSS class 時，使用 `className` attribute。這在所有標準的 DOM 和 SVG element 像是 `<div>`、`<a>` 其他的都能適用。

如果你在 React 裡使用 Web Components（這不是常見的狀況），則使用 `class`。

`dangerouslySetInnerHTML`

`dangerouslySetInnerHTML` 是 React 用來替代 DOM 的 `innerHTML`。普遍來說，從程式碼中注入 HTML 是個冒險的行為，你會很輕易地讓使用者暴露在 cross-site scripting (XSS) 攻擊風險之下。所以在 React 裡你還是可以直接注入 HTML，但是你必須使用 `dangerouslySetInnerHTML`，然後傳入一個有 `__html` 為 key 的 object，藉此來提醒你自己這樣做具有風險。例如：

```
function createMarkup() {
  return {__html: 'First &middot; Second'};
}

function MyComponent() {
  return <div dangerouslySetInnerHTML={createMarkup()} />;
}
```

`htmlFor`

React element 使用 `htmlFor` 來替代 `for`，因為 `for` 在 JavaScript 是保留字。

onChange

`onChange` event 會表現的跟你預期一樣：每當表格欄位值改變，這個 event 跟著發生。原先的 `onChange` 是個用詞不夠精確的行為，且 React 依靠這個 event 來處理使用者的即時輸入，所以我們表現出不同於瀏覽器的預設行為。

selected

如果你想要將 `<option>` 標記成已選擇，請在 `<select>` 的 `value` 中引用該選項的值。更多細節說明請參考 [「The select Tag」](#)。

style

注意

為了方便在文件裡某些範例會使用 `style`，但是**基本上不推薦使用 `style attribute` 作為初步修飾 elements 的手段**。在大部分的情形，應該使用 `className` 來對應定義在外部 CSS stylesheet 的 class。`style` 通常在 React 應用中會被用做動態 `style` 的增加方式。請參考 [FAQ: Styling and CSS](#)。

`style` attribute 接收一個 JavaScript object 內含以 camelCase 命名的 property，而不是 CSS string。這與 DOM `style` JavaScript property 一致，但效率更高，並且可以防止 XSS 安全漏洞。例如：

```
const divStyle = {
  color: 'blue',
  backgroundImage: 'url(' + imgUrl + ')',
};

function HelloWorldComponent() {
  return <div style={divStyle}>Hello World!</div>;
}
```

注意 `style` 不會自動加上前綴。為了要支援舊的瀏覽器，你必須自行提供相對應的 `style` property。

```
const divStyle = {
  WebkitTransition: 'all', // 注意這裡是大寫的「W」
  msTransition: 'all' // 「ms」是唯一需要使用小寫開頭的瀏覽器引擎前綴
};

function ComponentWithTransition() {
  return <div style={divStyle}>This should work cross-browser</div>;
}
```

為了與使用 JavaScript 來存取 DOM node 一致，`style` key 也以 camelCase 命名 (e.g. `node.style.backgroundImage`)。瀏覽器引擎前綸除了 `ms` 都應該以一個大寫字母開頭。這就是 `WebkitTransition` 為什麼有一個大寫「W」的原因。

React 會為某些數字型態的 `style` property 自動加上「px」。如果你想要使用其他單位，以 string 的形式加上單位。舉例來說：

```
// 輸出 style: '10px'
<div style={{ height: 10 }}>
  Hello World!
</div>

// 輸出 style: '10%'
<div style={{ height: '10%' }}>
  Hello World!
</div>
```

注意所有 `style` property 都會轉換成 pixel string。只有某些會保持沒有單位，如 `zoom`、`order`、`flex`。沒有單位的 property 完整列表可以在[這裡](#)查看。

suppressContentEditableWarning

通常來說，當一個 element 的 children 設定 `contentEditable` 時，因為不會有作用，所以會有警示訊息。這個 attribute 會抑制警示訊息。除非你在建立一個像是 [Draft.js](#) 的 library，否則不要使用它。

suppressHydrationWarning

如果你使用 server-side React render 時，通常 server 和 client render 不同時，會有一個警示訊息。然而，在一些少數的案例，很難去保證 server 和 client side 會完全符合。例如，像是 timestamp 就無法在 server 和 client 保持相同。

如果你設定了 `suppressHydrationWarning` 為 `true`，attribute 以及 element 內容不一樣時，React 就不會有警示訊息。這只作用在一層深度，且需要有計畫性地使用，請勿濫用。你可以在 [ReactDOM.hydrateRoot\(\) documentation](#) 讀到更多關於 hydration。

value

`value` attribute 可以使用在 `<input>`、`<select>` 和 `<textarea>` component。你可以使用它來設定 component value。這對建立 controlled component 很有幫助。`defaultValue` 則是使用在 uncontrolled component，在初始 mount 後設定 component 的 value。

可以使用的 HTML Attribute

在 React 16 中，任何標準或自訂的 DOM attribute 都可以使用。

React 為 DOM 提供了一套以 JavaScript 為中心的 API。因為 React component 通常會有自訂或跟 DOM 相關的 prop，React 使用像 DOM API 一樣的 `camelCase`。

```
<div tabIndex={-1} /> // 就像 node.tabIndex DOM API
<div className="Button" /> // 就像 node.className DOM API
<input readOnly={true} /> // 就像 node.readOnly DOM API
```

除了上述文件提到的，這些 prop 跟對應的 HTML attribute 運作方式一樣。

React 中可以使用這些 DOM attribute：

```
accept acceptCharset accessKey action allowFullScreen alt autoComplete
autoFocus autoPlay capture cellPadding cellSpacing challenge charSet checked
cite classID className cols content contentEditable contextMenu controls
```

```
controlsList coords crossOrigin data dateTime default defer dir disabled
download draggable encType form formAction formEncType formMethod formNoValidate
formTarget frameBorder headers height hidden high href hrefLang htmlFor
httpEquiv icon id inputMode integrity is keyParams keyType kind label lang list
loop low manifest marginHeight marginWidth max maxLength media mediaGroup method
min minLength multiple muted name noValidate nonce open optimum pattern
placeholder poster preload profile radioGroup readOnly rel required reversed
role rowSpan rows sandbox scope scoped scrolling seamless selected shape size
sizes span spellCheck src srcDoc srcLang srcSet start step style summary
tabIndex target title type useMap value width wmode wrap
```

同樣地，所有 SVG attribute 都可以使用：

```
accentHeight accumulate additive alignmentBaseline allowReorder alphabetic
amplitude arabicForm ascent attributeName attributeType autoReverse azimuth
baseFrequency baseProfile baselineShift bbox begin bias by calcMode capHeight
clip clipPath clipPathUnits clipRule colorInterpolation
colorInterpolationFilters colorProfile colorRendering contentScriptType
contentStyleType cursor cx cy d decelerate descent diffuseConstant direction
display divisor dominantBaseline dur dx dy edgeMode elevation enableBackground
end exponent externalResourcesRequired fill fillOpacity fillRule filter
filterRes filterUnits floodColor floodOpacity focusable fontFamily fontSize
fontSizeAdjust fontStretch fontStyle fontVariant fontWeight format from fx fy
g1 g2 glyphName glyphOrientationHorizontal glyphOrientationVertical glyphRef
gradientTransform gradientUnits hanging horizAdvX horizOriginX ideographic
imageRendering in in2 intercept k k1 k2 k3 k4 kernelMatrix kernelUnitLength
kerning keyPoints keySplines keyTimes lengthAdjust letterSpacing lightingColor
limitingConeAngle local markerEnd markerHeight markerMid markerStart
markerUnits markerWidth mask maskContentUnits maskUnits mathematical mode
numOctaves offset opacity operator order orient orientation origin overflow
overlinePosition overlineThickness paintOrder panose1 pathLength
patternContentUnits patternTransform patternUnits pointerEvents points
pointsAtX pointsAtY pointsAtZ preserveAlpha preserveAspectRatio primitiveUnits
r radius refX refY renderingIntent repeatCount repeatDur requiredExtensions
requiredFeatures restart result rotate rx ry scale seed shapeRendering slope
spacing specularConstant specularExponent speed spreadMethod startOffset
stdDeviation stemh stemv stitchTiles stopColor stopOpacity
strikethroughPosition strikethroughThickness string stroke strokeDasharray
strokeDashoffset strokeLinecap strokeLinejoin strokeMiterlimit strokeOpacity
strokeWidth surfaceScale systemLanguage tableValues targetX targetY textAnchor
textDecoration textLength textRendering to transform u1 u2 underlinePosition
underlineThickness unicode unicodeBidi unicodeRange unitsPerEm vAlphabetic
vHanging vIdeographic vMathematical values vectorEffect version vertAdvY
vertOriginX vertOriginY viewBox viewTarget visibility widths wordSpacing
writingMode x x1 x2 xChannelSelector xHeight xlinkActuate xlinkArcrole
xlinkHref xlinkRole xlinkShow xlinkTitle xlinkType xmlns xmlnsXlink xmlBase
xmlLang xmlSpace y y1 y2 yChannelSelector z zoomAndPan
```

你也可以使用自訂的 attribute，只要它們是完全小寫的。

SyntheticEvent

這份參考指南紀錄了 `SyntheticEvent` 這個形成 React 事件系統的 wrapper。想了解更多，請參考[事件處理](#)。

概觀

你的 event handler 將會是 `SyntheticEvent` 被傳遞的 instance，它是一個跨瀏覽器的、瀏覽器原生事件的 wrapper。它和瀏覽器原生事件有相同的介面，包含 `stopPropagation()` 和 `preventDefault()`，除了原生事件在所有的瀏覽器都以相同的方式運作這點以外。

如果你發現因為某些原因你需要使用瀏覽器的底層事件，你只需要使用 `nativeEvent` 這個 attribute 即可。Synthetic event 不同於瀏覽器的 native event 並不會直接 mapping，例如在 `onMouseLeave` `event.nativeEvent` 將會指向到 `mouseout` event。特定 mapping 不是公開 API 的一部分，並且可以隨時更改。每個 `SyntheticEvent` object 都有下列的 attribute：

```
boolean bubbles
boolean cancelable
DOMEventTarget currentTarget
boolean defaultPrevented
number eventPhase
boolean isTrusted
DOMEvent nativeEvent
void preventDefault()
boolean isDefaultPrevented()
void stopPropagation()
boolean isPropagationStopped()
void persist()
DOMEventTarget target
number timeStamp
string type
```

注意：

從 v17 開始，`e.persist()` 將不會再有任何作用，因為 `SyntheticEvent` 已不再被 pool 了。

注意：

截至 v0.14 為止，從 event handler 回傳 `false` 並不會停止事件冒泡（event propagation）。因此你可以視情況手寫觸發 `e.stopPropagation()` 或 `e.preventDefault()`。

支援的事件

React 將事件規格化，已讓它們在不同的瀏覽器中有一致的屬性。

以下的 event handler 會在冒泡階段時被一個事件觸發。如果你想註冊捕獲階段的 event handler，請在事件名稱的後面加上 `Capture`。例如，假設你想在捕獲階段捕捉 click 事件的話，你需要用 `onClickCapture` 而不是 `onClick`：

- [Clipboard Events](#)
 - [Composition Events](#)
 - [Keyboard Events](#)
 - [Focus Events](#)
 - [Form Events](#)
 - [Generic Events](#)
 - [Mouse Events](#)
 - [Pointer Events](#)
 - [Selection Events](#)
 - [Touch Events](#)
 - [UI Events](#)
 - [Wheel Events](#)
 - [Media Events](#)
 - [Image Events](#)
 - [Animation Events](#)
 - [Transition Events](#)
 - [Other Events](#)
-

參考

剪貼板事件

事件名稱：

`onCopy onCut onPaste`

屬性：

`DOMDataTransfer clipboardData`

組合事件

事件名稱：

`onCompositionEnd onCompositionStart onCompositionUpdate`

屬性：

`string data`

鍵盤事件

事件名稱：

`onKeyDown onKeyPress onKeyUp`

屬性：

```
boolean altKey
number charCode
boolean ctrlKey
boolean getModifierState(key)
string key
number keyCode
string locale
number location
boolean metaKey
boolean repeat
boolean shiftKey
number which
```

`key` 屬性可以接受 DOM Level 3 Events spec 內記錄的任意值。

焦點事件

事件名稱：

`onFocus onBlur`

這些焦點事件在 React DOM 中所有的 element 上都可以使用，不限於表單 element。

屬性：

`DOMEventTarget relatedTarget`

onFocus

當 element (或是 element 裡面的 element) 受到 focus 時，`onFocus` 事件會被呼叫。例如，下方的 `onFocus` 事件會在當使用者點擊一個文字輸入時觸發。

```
function Example() {
  return (
    <input
      onFocus={(e) => {
        console.log('Focused on input');
      }}
      placeholder="onFocus is triggered when you click this input."
    />
  )
}
```

onBlur

The `onBlur` event handler is called when focus has left the element (or left some element inside of it). For example, it's called when the user clicks outside of a focused text input.

```
function Example() {
  return (
    <input
      onBlur={(e) => {
        console.log('Triggered because this input lost focus');
      }}
      placeholder="onBlur is triggered when you click this input and then you
      click outside of it."
    />
  )
}
```

Detecting Focus Entering and Leaving

You can use the `currentTarget` and `relatedTarget` to differentiate if the focusing or blurring events originated from *outside* of the parent element. Here is a demo you can copy and paste that shows how to detect focusing a child, focusing the element itself, and focus entering or leaving the whole subtree.

```
function Example() {
  return (
    <div
      tabIndex={1}
      onFocus={(e) => {
        if (e.currentTarget === e.target) {
          console.log('focused self');
        } else {
          console.log('focused child', e.target);
        }
        if (!e.currentTarget.contains(e.relatedTarget)) {
          // Not triggered when swapping focus between children
          console.log('focus entered self');
        }
      }}
      onBlur={(e) => {
        if (e.currentTarget === e.target) {
          console.log('unfocused self');
        } else {
          console.log('unfocused child', e.target);
        }
        if (!e.currentTarget.contains(e.relatedTarget)) {
          // Not triggered when swapping focus between children
          console.log('focus left self');
        }
      }}
    >
      <input id="1" />
      <input id="2" />
    </div>
  );
}
```

表單事件

事件名稱：

`onChange onInput onInvalid onReset onSubmit`

想了解關於 `onChange` 事件的資訊，請參考[表單](#)。

Generic 事件

事件名稱：

`onError onLoad`

滑鼠事件

事件名稱：

`onClick onContextMenu onDoubleClick onDrag onDragEnd onDragEnter onDragExit
onDragLeave onDragOver onDragStart onDrop onMouseDown onMouseEnter onMouseLeave
onMouseMove onMouseOut onMouseOver onMouseUp`

`onMouseEnter` 和 `onMouseLeave` 事件從離開的 element 向正在進入的 element 傳播，而不是正常的冒泡，也沒有捕獲階段。

屬性：

```
boolean altKey
number button
number buttons
number clientX
number clientY
boolean ctrlKey
boolean getModifierState(key)
boolean metaKey
number pageX
number pageY
DOMEventTarget relatedTarget
number screenX
number screenY
boolean shiftKey
```

指標事件

事件名稱：

`onPointerDown onPointerMove onPointerUp onPointerCancel onGotPointerCapture
onLostPointerCapture onPointerEnter onPointerLeave onPointerOver onPointerOut`

`onPointerEnter` 和 `onPointerLeave` 事件從離開的 element 向正在進入的 element 傳播，而不是正常的冒泡，也沒有捕獲階段。

屬性：

如同在 [W3 spec](#) 內定義的，指標事件是滑鼠事件的延伸，並帶有以下屬性：

```
number pointerId
number width
number height
number pressure
number tangentialPressure
number tiltX
number tiltY
number twist
string pointerType
boolean isPrimary
```

關於跨瀏覽器支援的說明：

目前為止，並非所有的瀏覽器都支援指標事件（在這篇文章撰寫的時候，支援該事件的瀏覽器有：Chrome、Firefox、Edge 以及 Internet Explorer）。React 刻意不通過 polyfill 的方式支援其他瀏覽器，因為符合標準的 polyfill 會明顯地增加 `react-dom` 的 bundle 大小。

如果你的應用程式需要指標事件，我們建議你加上第三方的指針事件 polyfill。

選擇事件

事件名稱：

```
onSelect
```

觸摸事件

事件名稱：

```
onTouchCancel onTouchEnd onTouchMove onTouchStart
```

屬性：

```
boolean altKey
DOMTouchList changedTouches
boolean ctrlKey
boolean getModifierState(key)
boolean metaKey
boolean shiftKey
DOMTouchList targetTouches
DOMTouchList touches
```

UI 事件

事件名稱：

```
onScroll
```

注意：

從 React 17 開始，`onScroll` 事件就不是冒泡了，這與瀏覽器的行為相符，並且避免了當巢狀滾動元件觸發事件時，在遠處 parent 元件上的混亂現象。

屬性：

`number detail`
`DOMAbstractView view`

滾輪事件

事件名稱：

`onWheel`

屬性：

`number deltaMode`
`number deltaX`
`number deltaY`
`number deltaZ`

媒體事件

事件名稱：

`onAbort` `onCanPlay` `onCanPlayThrough` `onDurationChange` `onEmptied` `onEncrypted`
`onEnded` `onError` `onLoadedData` `onLoadedMetadata` `onLoadStart` `onPause` `onPlay`
`onPlaying` `onProgress` `onRateChange` `onSeeked` `onSeeking` `onStalled` `onSuspend`
`onTimeUpdate` `onVolumeChange` `onWaiting`

圖片事件

事件名稱：

`onLoad` `onError`

動畫事件

事件名稱：

`onAnimationStart` `onAnimationEnd` `onAnimationIteration`

屬性：

```
string animationName  
string pseudoElement  
float elapsedTime
```

過渡事件

事件名稱：

```
onTransitionEnd
```

屬性：

```
string propertyName  
string pseudoElement  
float elapsedTime
```

其他事件

事件名稱：

```
onToggle
```

React 術語表

單頁應用程式

單頁應用程式會載入一個 HTML 頁面和其他所需要的資源（例如 JavaScript 和 CSS），之後在頁面或後續頁面上的互動都不需要往返伺服器，這表示該頁面不會被重新載入。

縱使你可以用 React 來建立單頁應用程式，但這不是必需的。React 也可以被用在現有網站的一小部分以增強交互性。用 React 編寫的代碼可以與伺服器 render（例如 PHP）的 markup 或其他客戶端 library 完美兼容。事實上，React 在 Facebook 中也是這樣使用的。

ES6, ES2015, ES2016, 等

這些縮略詞都是指最新版本的 ECMAScript 語言規範標準，而 JavaScript 語言就是它們的一個實現。ES6 版本（也被稱為 ES2015）包含了許多新特性，例如：arrow function, class, 樣板字面值，`let` 與 `const` 變數聲明。你可以在[這裏](#)了解更多版本的新特性。

編譯器

JavaScript 編譯器接受 JavaScript 代碼，對其進行轉換並返回不同的格式的 JavaScript 代碼。最常見的使用例子是把 ES6 語法轉換為一些舊瀏覽器能夠辨識的語法。[Babel](#) 是 React 上最常用的編譯器。

Bundler

Bundlers 將 JavaScript 和 CSS 程式碼作為獨立的模組撰寫（通常有數百個），並將它們合併成數個檔案並針對瀏覽器進行最佳化。在 React 應用程式常用的 bundlers 包括 [Webpack](#) 和 [Browserify](#)。

套件管理工具

套件管理工具是用來管理項目的 dependency。[npm](#) 和 [Yarn](#) 都是在 React 上常用的套件管理工具。它們都是使用相同 npm 套件註冊表的客戶端。

CDN

CDN 即是 內容傳遞網路。CDN 從全球各地的伺服器提供靜態內容的緩存。

JSX

JSX 是一種 JavaScript 語法的擴展。它跟模板語言類似，但具有 JavaScript 的全部功能。JSX 會被編譯為 `React.createElement()`，然後返回被稱為 React element 的 JavaScript object。JSX 的基本簡介可以參見[這裏](#)，更深入的教程可以參見[這裏](#)。

React DOM 使用 camelCase 來命名 HTML 屬性名稱。例如，`tabindex` 在 JSX 中寫作 `tabIndex`。而 `class` 因為是 JavaScript 中的保留字，所以寫作 `className`：

```
<h1 className="hello">My name is Clementine!</h1>
```

Element {#elements}

React element 是 React 應用程式的建構模塊。有人可能會將 element 跟更廣為人知的 component 概念混淆。element 描述你在屏幕上看到甚麼。React element 是不可變的。

```
const element = <h1>Hello, world</h1>;
```

通常 element 不會被直接使用，而是在 component 中被返回。

Components {#components}

React component 是一段小，可重複使用的代碼，用來返回一個被 render 在頁面的 React element。最簡單的 React component 是一個普通的 JavaScript function，返回一個 React element。

```
function Welcome(props) {
  return <h1>Hello, {props.name}</h1>;
}
```

Component 也可以是 ES6 class：

```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
}
```

Component 可以分解為不同的功能部件，並在其他 component 中使用。Component 可以返回其他 component, array, string 和 number。一個很好的經驗法則是，如果你的 UI 中有一部份被重複使用多次 (Button, Panel, Avatar)，或自身就足夠複雜 (App, FeedStory, Comment)，這些都是成為可重複使用 component 的好選擇。Component 的名字必需以大寫字母開始 (`<Wrapper/>` 而不是 `<wrapper/>`)。參見[這裏](#)來了解更多 render component 的資料。

props {#props}

`props` 是 React component 的輸入。它們是從父 component 傳遞到子 component 的數據。

請記著 `props` 是只讀的。不應該以任何方式來修改它們。

```
// 錯誤!
props.number = 42;
```

如果你需要修改某些數值來反映用戶輸入或網絡響應，請使用 `state` 來代替。

props.children {#propschildren}

`props.children` 在每個 component 上都可用。它包含 component 開始至完結標記之間的內容。例如：

```
<Welcome>Hello world!</Welcome>
```

字符串 `Hello world!` 在 `Welcome` component 可以在 `props.children` 中獲取。

```
function Welcome(props) {
  return <p>{props.children}</p>;
}
```

以 class 定義的 component，使用 `this.props.children`：

```
class Welcome extends React.Component {
  render() {
    return <p>{this.props.children}</p>;
  }
}
```

state {#state}

當一個 component 與其相關的數據會隨時間而改變，就需要 `state`。例如，一個 `Checkbox` component 可能需要 `isChecked` 的 state，而 `NewFeed` component 可能希望追蹤 `fetchedPosts` 在它的 state 中。

`state` 和 `props` 最主要的區別是 `props` 是由父 component 傳遞，而 `state` 是由 component 本身管理的。一個 component 不能改變自己的 `props`，但就可以改變自己的 `state`。

對於每個特定變化的數據，應該只有一個 component 「擁有」它在 `state` 中。不要試圖同步兩個不同 component 的 `state`。反而，把 `state 提升`至最接近它們的共同 ancestor，並把它以 `props` 的形式傳遞至它們兩個。

生命週期方法 {#lifecycle-methods}

生命週期方法是用來在 component 不同階段來執行自訂功能。以下的事件都有生命週期方法：當 component 被建立和加入在 DOM 裏 (`mounting`)，當 component 更新，以及從 DOM 中 `unmount` 或移除時。

Controlled 與 Uncontrolled Components

React 有兩個不同的方案處理表格輸入。

表格輸入 element 的值是由 React 來控制，就被稱為 *controlled component*。當用戶輸入數據到 controlled component，就會觸發一個 event handler，並且用你的代碼決定輸入是否有效（通過重新 render 更新後的數值）。如果你不重新 render，表格 element 將保持不變。

一個 *uncontrolled component* 就像表格 element 一樣在 React 以外工作。當用戶輸入數據到一個表格列（input box, dropdown 等）時，不需要 React 處理任何東西，更新的數據就會被反映出來。但同時這意味著你不能強迫表格列有特定的數值。

在大多數的情況下你應該使用 controlled components。

Keys {#keys}

`key` 是建立 arrays of elements 時，你需要包含在內的一個特殊的 string attribute。`key` 幫助 React 辨認哪些項目已更改、新增或移除。應該替 array 內部的 element 提供 key，以使 element 具有穩定的身份。

`key` 只需要在同一個 array 的 `sibling elements` 中是唯一的。它們不需要在整個應用程式或甚至單一 component 中是唯一的。

不要將 `Math.random()` 之類的值傳給 `key`。重要的是，`key` 在 re-render 時具有「穩定的身份」，以便 React 可以確定項目何時新增、移除或重新排序。理想情況下，`key` 應該對應到來自於你的資料中唯一且穩定的身份標識，例如 `post.id`。

Refs {#refs}

React 支持一個可以附加到任何 component 的特殊 attribute。Ref attribute 可以是由 `React.createRef()` function 返回的 object，或一個 callback function，或一個 string (在舊有 API 中)。當 `ref` attribute 是 callback function 時，該 function 接受底層的 DOM element 或 class instance (視乎 element 的類別) 作為參數。這使你可以直接訪問 DOM element 或 component instance。

不要過度使用 `ref`。假若你發現你在應用程式中經常使用 `ref` 來「實現某些事情」，請考慮對 [由上而下的數據流](#) 更熟悉。

Events {#events}

在 React element 中處理 event 有一些語法上的不同：

- React 的 event handler 以 camelCase 來命名，而不是小寫。
- 在 JSX 你需要傳入一個 function 到 event handler，而不是一個 string。

Reconciliation {#reconciliation}

當一個 component 的 props 或 stats 變更時，React 通過比較新返回的 element 和之前 render 的來決定是否需要實際的 DOM 更新。當它們不相等時，React 將更新 DOM。這個過程被稱為「reconciliation」。

JavaScript 環境要求

支援所有現代的瀏覽器 (Edge、Firefox、Chrome、Safari, 等等)。

如果你支援更舊的瀏覽器或是裝置像是 Internet Explorer, 它們沒有提供現代瀏覽器功能, 或是不合規範的實作, 考慮在你的應用程式 bundle 內引入一個全域的 polyfill。

以下是 React 18 使用的現代 feature 的列表：

- `Promise`
- `Symbol`
- `Object.assign`

這些功能的正確 polyfill 取決於你的環境。對於許多使用者, 你可以設定你的瀏覽器列表設定。對於其他人, 你可能需要直接 import 像 `core-js` 這樣的 polyfill。

Profiler API

The `Profiler` measures how often a React application renders and what the "cost" of rendering is. Its purpose is to help identify parts of an application that are slow and may benefit from [optimizations such as memoization](#).

Note:

Profiling adds some additional overhead, so **it is disabled in the production build**.

To opt into production profiling, React provides a special production build with profiling enabled. Read more about how to use this build at fb.me/react-profiling

Usage

A `Profiler` can be added anywhere in a React tree to measure the cost of rendering that part of the tree. It requires two props: an `id` (string) and an `onRender` callback (function) which React calls any time a component within the tree "commits" an update.

For example, to profile a `Navigation` component and its descendants:

```
render(
  <App>
    <Profiler id="Navigation" onRender={callback}>
      <Navigation {...props} />
    </Profiler>
    <Main {...props} />
  </App>
);
```

Multiple `Profiler` components can be used to measure different parts of an application:

```
render(
  <App>
    <Profiler id="Navigation" onRender={callback}>
      <Navigation {...props} />
    </Profiler>
    <Profiler id="Main" onRender={callback}>
      <Main {...props} />
    </Profiler>
  </App>
);
```

`Profiler` components can also be nested to measure different components within the same subtree:

```
render(
  <App>
    <Profiler id="Panel" onRender={callback}>
      <Panel {...props}>
```

```

        <Profiler id="Content" onRender={callback}>
          <Content {...props} />
        </Profiler>
        <Profiler id="PreviewPane" onRender={callback}>
          <PreviewPane {...props} />
        </Profiler>
      </Panel>
    </Profiler>
  </App>
);

```

Note

Although `Profiler` is a light-weight component, it should be used only when necessary; each use adds some CPU and memory overhead to an application.

onRender Callback

The `Profiler` requires an `onRender` function as a prop. React calls this function any time a component within the profiled tree "commits" an update. It receives parameters describing what was rendered and how long it took.

```

function onRenderCallback(
  id, // the "id" prop of the Profiler tree that has just committed
  phase, // either "mount" (if the tree just mounted) or "update" (if it re-
  rendered)
  actualDuration, // time spent rendering the committed update
  baseDuration, // estimated time to render the entire subtree without memoization
  startTime, // when React began rendering this update
  commitTime, // when React committed this update
  interactions // the Set of interactions belonging to this update
) {
  // Aggregate or log render timings...
}

```

Let's take a closer look at each of the props:

The `id` prop of the `Profiler` tree that has just committed. This can be used to identify which part of the tree was committed if you are using multiple profilers.

- `phase: "mount" | "update"` - Identifies whether the tree has just been mounted for the first time or re-rendered due to a change in props, state, or hooks.
- `actualDuration: number` - Time spent rendering the `Profiler` and its descendants for the current update. This indicates how well the subtree makes use of memoization (e.g. `React.memo`, `useMemo`, `shouldComponentUpdate`). Ideally this value should decrease significantly after the initial mount as many of the descendants will only need to re-render if their specific props change.
- `baseDuration: number` - Duration of the most recent `render` time for each individual component within the `Profiler` tree. This value estimates a worst-case cost of rendering (e.g. the initial mount or a tree with no memoization).
- `startTime: number` - Timestamp when React began rendering the current update.

- `commitTime: number` - Timestamp when React committed the current update. This value is shared between all profilers in a commit, enabling them to be grouped if desirable.
- `interactions: Set` - Set of "interactions" that were being traced when the update was scheduled (e.g. when `render` or `setState` were called).

Note

Interactions can be used to identify the cause of an update, although the API for tracing them is still experimental.

Learn more about it at fb.me/react-interaction-tracing

PureRenderMixin

Note

The `PureRenderMixin` mixin predates `React.PureComponent`. This reference doc is provided for legacy purposes, and you should consider using `React.PureComponent` instead.

If your React component's render function renders the same result given the same props and state, you can use this mixin for a performance boost in some cases.

Example:

```
var PureRenderMixin = require('react-addons-pure-render-mixin');
var createReactClass = require('create-react-class');
createReactClass({
  mixins: [PureRenderMixin],

  render: function() {
    return <div className={this.props.className}>foo</div>;
  }
});
```

Example using ES6 class syntax:

```
import PureRenderMixin from 'react-addons-pure-render-mixin';
class FooComponent extends React.Component {
  constructor(props) {
    super(props);
    this.shouldComponentUpdate = PureRenderMixin.shouldComponentUpdate.bind(this);
  }

  render() {
    return <div className={this.props.className}>foo</div>;
  }
}
```

Under the hood, the mixin implements `shouldComponentUpdate`, in which it compares the current props and state with the next ones and returns `false` if the equalities pass.

Note:

This only shallowly compares the objects. If these contain complex data structures, it may produce false-negatives for deeper differences. Only mix into components which have simple props and state, or use `forceUpdate()` when you know deep data structures have changed. Or, consider using [immutable objects](#) to facilitate fast comparisons of nested data.

Furthermore, `shouldComponentUpdate` skips updates for the whole component subtree. Make sure all the children components are also "pure".

React.Component

本章節包含了 React component class 的詳細 API 參考。我們假設你對 React 的基本概念已十分熟悉，例如 [Component](#) 和 [Prop](#) 以及 [State](#) 和 [生命週期](#)。如果你對這些概念還不清楚，請先閱讀相關文件。

概觀

在 React 中，你可以將 component 定義成 class 或 function。目前，被定義為 class 的 component 提供了更多功能，我們將會在本章節中逐一介紹。要定義一個 React component class，你需要繼承 (extend) `React.Component`：

```
class Welcome extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

在 `React.Component` 的 subclass 中唯一一個你必須定義的方法是 `render()`。本章節中所有其他的方法都並非絕對必要。

我們強烈建議你不要建立自己的 base component class。 在 React component 中，[程式的重複使用性主要是透過組合而非繼承來完成的](#)。

注意：

React 並不會強迫你使用 ES6 class 語法。如果你想避免它的話，你可以使用 `create-react-class` 或一個類似的自訂抽象。想了解更多詳情，請參考[如何在 React 中不使用 ES6](#)一文。

Component 生命週期

每一個 component 都有數個「生命週期方法」，你可以 [override](#) 這些方法，以便在開發過程中某些特定的時刻執行某些程式。你可以使用[這個生命週期表](#)作為速查表。以下，常用的生命週期方法將會以粗體表達。其餘的生命週期方法則相對較少。

Mounting

當一個 component 的 instance 被建立且加入 DOM 中時，其生命週期將會依照下列的順序呼叫這些方法：

- `constructor()`
- `static getDerivedStateFromProps()`
- `render()`
- `componentDidMount()`

注意：

這個方法已經被認為已過時，你在寫新程式應**避免使用**：

- `UNSAFE_componentWillMount()`

更新

當 prop 或 state 有變化時，就會產生更新。當一個 component 被重新 render 時，其生命週期將會依照下列的順序呼叫這些方法：

- `static getDerivedStateFromProps()`
- `shouldComponentUpdate()`
- `render()`
- `getSnapshotBeforeUpdate()`
- `componentDidUpdate()`

注意：

下列方法已過時，你在寫新程式應**避免使用**：

- `UNSAFE_componentWillUpdate()`
- `UNSAFE_componentWillReceiveProps()`

Unmounting

當一個 component 被從 DOM 中移除時，這個方法將會被呼叫：

- `componentWillUnmount()`

錯誤處理

當一個 component 在 render 的過程、生命週期、或在某個 child component 的 constructor 中發生錯誤時，這些方法會被呼叫：

- `static getDerivedStateFromError()`
- `componentDidCatch()`

其他 APIs

每個 component 也提供了其他 API：

- `setState()`
- `forceUpdate()`

Class 屬性

- `defaultProps`
- `displayName`

Instance 屬性

- `props`
- `state`

參考指南

常用的生命週期方法

此段落將會介紹你在建立 React component 時最可能會使用到的幾種方法。想更深入了解生命週期方法，請參考[生命週期表](#)。

`render()` {#render}

`render()`

`render()` 是 class component 中唯一必要的方法。

當 `render` 被呼叫時，它將會檢視 `this.props` 和 `this.state` 中的變化，並回傳以下類別之一：

- **React element**。通常是透過 `JSX` 建立的。例如，`<div />` 和 `<MyComponent />` 這兩個 React element 會告訴 React 要 render 一個 DOM node 和一個使用者定義的 component。
- **Array 和 fragment**。它們會從 `render` 中回傳數個 element。細節請參考 [fragment](#)。
- **Portal**。它們讓你將 children render 到不同的 DOM subtree 中。細節請參考 [portal](#)。
- **String 和 number**。這些在 DOM 中將會被 render 為文字 node。
- **Boolean 或 null**。什麼都不 render。（此類型主要是支援 回傳 `test && <Child />` 的模式，這裡的 `test` 是一個 boolean 值）。

`render()` function 應該是 pure 的，這表示：它並不會改變 component 的 state，它在每次呼叫時都會回傳同樣的結果，它並不會直接和瀏覽器有所互動。

如果你需要和瀏覽器互動，請在 `componentDidMount()` 或其他的生命週期方法內執行你的程式。將 `render()` 維持在 pure 的狀態有助於你對 component 的理解。

注意：

若 `shouldComponentUpdate()` 回傳的值為 `false` 的話，`render()` 將不會被呼叫。

`constructor()` {#constructor}

`constructor(props)`

如果你沒有初始化 `state` 也不綁定方法的話，你的 **React component** 就不需要 `constructor`。

一個 React component 的 constructor 會在其被 mount 之前被呼叫。當你為一個 `React.Component` subclass 建立 constructor 時，你應該在其他任何宣告之前呼叫 `super(props)`。否則，`this.props` 在 constructor 中的值會出現 undefined 的 bug。

通常在 React 中 constructor 只會有兩種用途：

- 透過指定一個 `this.state` 物件來初始化內部 state。
- 為 event handler 方法綁定 instance。

請不要在 `constructor()` 中呼叫 `setState()`。 如果你的 component 需要使用內部 state，請在 constructor 中將其最初的 state 指定為 `this.state`：

```
constructor(props) {
  super(props);
  // 不要在這裡呼叫 this.setState() !
  this.state = { counter: 0 };
  this.handleClick = this.handleClick.bind(this);
}
```

Constructor 是唯一一個你應該直接指定 `this.state` 的地方。在所有其他的方法中，你則需要使用 `this.setState()`。

請避免在 constructor 中產生任何 side effect 或 subscription。如果你需要它們的話，請使用 `componentDidMount()`。

注意：

請避免複製 prop 到 state 之中！這是一個很常見的錯誤：

```
constructor(props) {
  super(props);
  // 請不要這樣做！
  this.state = { color: props.color };
}
```

這樣做的問題是：一來這毫無必要（你可以直接用 `this.props.color`），二來這會產生 bug（任何改變對 color prop 所產生的更新都不會出現在 state 中）。

請在只有在你刻意要忽略 prop 更新的情況下才使用這個模式。 在這種情況下，比較合理的做法是將 prop 重新命名為 `initialColor` 或 `defaultColor`。如此一來，你可以在必要的情況下透過修改一個 component 的 key 來強迫它「重置」其初始的 state。

若想知道如何處理 state 依賴 prop 的情況，請參考我們 [關於避免 derived state 的部落格文章](#)。

componentDidMount() {#componentdidmount}

`componentDidMount()`

在一個 component 被 mount (加入 DOM tree 中) 後, `componentDidMount()` 會馬上被呼叫。需要 DOM node 的初始化應該寫在這個方法裡面。如果你需要從遠端終端點 (remote endpoint) 請求資料的話, 此處非常適合進行實例化網路請求 (network request)。

這個方法適合設立任何 subscription。設立完 subscription 後, 別忘了在 `componentWillUnmount()` 內取消 subscription。

你可以馬上在 `componentDidMount()` 內呼叫 `setState()`。這會觸發一次額外的 render, 但這會在瀏覽器更新螢幕之前發生。在這個情況下, 即使 `render()` 被呼叫兩次, 這確保使用者不會看見這兩次 render 中過渡時期的 state。請謹慎使用這個模式, 因為這經常會導致效能問題。在大多數情況下, 你應該能夠在 `constructor()` 內指定初始 state 的值。不過, 在某些情況下, 像是在使用 modal 和 tooltip 的時候, 你所 render 的 component 若是依賴某個 DOM node 的大小或位置時, 這種模式有時候可能是有必要的。

`componentDidUpdate()` {#componentdidupdate}

`componentDidUpdate(prevProps, prevState, snapshot)`

`componentDidUpdate()` 會在更新後馬上被呼叫。這個方法並不會在初次 render 時被呼叫。

在 component 更新之後, 可以在此處對 DOM 進行運作。此處也適合做網路請求, 如果你有比較目前的 prop 和之前的 prop 的話 (如果 prop 沒有改變的話, 網路請求可能並非必要)。

```
componentDidUpdate(prevProps) {
  // 常見用法 (別忘了比較 prop):
  if (this.props.userID !== prevProps.userID) {
    this.fetchData(this.props.userID);
  }
}
```

你可以馬上在 `componentDidUpdate()` 內呼叫 `setState()`, 但注意這必須要被包圍在一個類似上述範例的條件語句內, 否則你會進入一個無限迴圈。這也會導致額外的重新 render。雖然使用者看不見, 但這可能會影響 component 的效能。如果你想試著將某些 state 複製到由上往下傳的 prop 的話, 請考慮直接使用 prop。請參考[為何複製 prop 到 state 中會產生 bug](#)。

如果你的 component 裡面有 `getSnapshotBeforeUpdate()` 這個很少見的生命週期方法, 其回傳的值將會被當作第三個「snapshot」參數傳給 `componentDidUpdate()`。否則這個參數會是 `undefined`。

注意：

如果 `shouldComponentUpdate()` 回傳的值為 `false` 的話, `componentDidUpdate()` 將不會被呼叫。

`componentWillUnmount()` {#componentwillunmount}

`componentWillUnmount()`

`componentWillUnmount()` 會在一個 component 被 unmount 和 destroy 後馬上被呼叫。你可以在這個方法內進行任何清理，像是取消計時器和網路請求或是移除任何在 `componentDidMount()` 內建立的 subscription。

你不應該在 `componentWillUnmount()` 內呼叫 `setState()`，因為這個 component 永遠不會再重新 render。當一個 component instance 被 unmount 後，它就永遠不會再被 mount。

不常使用的生命週期方法

這個章節內介紹的方法是在那些不常見的情況中使用的。它們有時很方便，但是你大多數的 component 大概不會需要使用它們。你可以在這個[生命週期表](#)上方點選「顯示不常使用的生命週期方法」，並看到以下所介紹的方法。

`shouldComponentUpdate() {#shouldcomponentupdate}`

`shouldComponentUpdate(nextProps, nextState)`

你可以使用 `shouldComponentUpdate()` 來讓 React 知道一個 component 的 output 並不會被目前在 state 或 prop 內的改變所影響。React 的預設行為是每當 state 有所改變時就重新 render。在絕大多數的情況下，你應該依賴這個預設行為。

`shouldComponentUpdate()` 會在新的 prop 或 state 被接收之後並在該 component 被 render 之前被呼叫。其預設值是 `true`。這個方法並不會 component 初次 render 時或使用 `forceUpdate()` 時被呼叫。

這個方法的存在著要是為了[效能最佳化](#)。請不要依賴這個方法來「避免」render，因為這很有可能會導致 bug。[請考慮使用 React 內建的 `PureComponent`](#) 並避免手寫 `shouldComponentUpdate()`。`PureComponent` 會為 prop 和 state 做一個淺層比較 (shallow comparison) 並減低你錯過必要更新的機會。

如果你很確定你想要手寫這個方法的話，你可以將 `this.props` 和 `nextProps` 以及 `this.state` 和 `nextState` 做比較並回傳 `false` 以告知 React 這次的更新可以被略過。請注意，回傳 `false` 並不會避免 child component 在它們的 state 改變時重新 render。

我們並不建議你做深度比較 (deep equality check) 或在 `shouldComponentUpdate()` 內使用 `JSON.stringify()`。它們效率不佳且會造成功能問題。

目前，如果 `shouldComponentUpdate()` 回傳 `false` 的話，`UNSAFE_componentWillUpdate()`、`render()` 和 `componentDidUpdate()` 都不會被呼叫。在未來，React 可能會把 `shouldComponentUpdate()` 當作一個提示而非一個嚴格指令，而回傳 `false` 可能還是會造成 component 重新 render。

`static getDerivedStateFromProps() {#static-getderivedstatefromprops}`

`static getDerivedStateFromProps(props, state)`

`getDerivedStateFromProps` 會在一個 component 被 render 前被呼叫，不管是在首次 mount 時或後續的更新時。它應該回傳一個 object 以更新 state，或回傳 `null` 以表示不需要更新任何 state。

這個方法是為了某些[很少見的例子](#)而存在的，像是有時 state 會依賴 prop 在一段時間過後所產生的改變。例如，也許建立一個 `<Transition>` component 是很方便的，我們可以用它來比較其之前與之後的 children，並決定我們要 animate in and out 哪一個 child。

繼承 state 會導致冗長的程式碼並使你的 component 很難理解。[請確認你知道這些較為簡單的替代方案](#)。

- 如果你需要在某個 prop 改變時產生相對應的 **side effect**（例如，資料提取或使用動畫），請使用 `componentDidUpdate`。
- 如果你想要 **在某個 prop 改變時重新計算某些資料**，[請使用 memoization helper](#)。
- 如果你想要 **在某個 prop 改變時「重置」某個 state**，請考慮建立一個[完全被控制](#)的 component 或帶有 `key` 的[完全不可被控制](#) component。

這個方法無法觸及一個 component 的 instance。如果你想要這麼做的話，你可以把一個 component 的 prop 和 state 提取出來變成 pure function，並寫在該 class definition 之外，並透過這樣的方式在 `getDerivedStateFromProps()` 和其他 class 方法之間重複使用某些程式碼。

請注意這個方法在每一次 render 時都會被觸發，不論原因為何。這和 `UNSAFE_componentWillReceiveProps` 有所不同，這個方法只有在 parent 導致重新 render 時被觸發，而非在本地的 `setState` 導致重新 render 時被觸發。

`getSnapshotBeforeUpdate()` [\[#getsnapshotbeforeupdate\]](#)

`getSnapshotBeforeUpdate(prevProps, prevState)`

`getSnapshotBeforeUpdate()` 在提交最新 render 的 output 之前立即被調用。它讓你在 DOM 讀取之前先從其中抓取一些資訊（例如滾動軸的位置）。這個生命週期方法回傳的任何值會被當作一個參數傳遞給 `componentDidUpdate()`。

這個方法並不常見，但它可能會在像是對話串這類需要以某種特殊方始處理滾動軸位置的 UI 中出現。

一個 snapshot 的值（或 `null`）應該被回傳。

例如：

```
embed:react-component-reference/get-snapshot-before-update.js
```

在上面這個例子中，讀取 `getSnapshotBeforeUpdate` 內的 `scrollHeight` property 是很重要的，因為「render」階段的生命週期方法（如 `render`）和「commit」階段的生命週期方法（像是 `getSnapshotBeforeUpdate` 和 `componentDidUpdate`）兩者之間可能會有一些延遲。

錯誤邊界

錯誤邊界 是用於截取 child component tree 中 JavaScript 錯誤、記錄錯誤、並顯示一個 fallback UI 而非故障的 component tree 的一群 React component。錯誤邊界會在 render 期間、生命週期方法、以及其下整個 tree 群組所有的 constructor 內截取錯誤。

一個 class component 會變成錯誤邊界，如果其定義了 `static getDerivedStateFromError()` 和 `componentDidCatch()` 兩種或其中之一的生命週期方法。從這些生命週期方法中更新 state 讓你截取在其下的 tree 內未被處理的 JavaScript 錯誤，並顯示一個 fallback UI。

請只在從意料之外的異常中使用錯誤邊界。請不要用它來控制流程。

想了解更多，請參考 [React 16 中的錯誤邊界](#)一文。

注意：

錯誤邊界只會截取在 tree 中、自身以下的 component 中的錯誤。錯誤邊界無法截取自身內的錯誤。

`static getDerivedStateFromError() {#static-getderivedstatefromerror}`

`static getDerivedStateFromError(error)`

這個生命週期方法會在某個錯誤被一個 descendant component 拋出後被呼叫。它會接收該錯誤為其參數並回傳一個值以更新 state。

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    // 更新 state，如此下次 render 時 React 才能顯示 fallback UI
    return { hasError: true };
  }

  render() {
    if (this.state.hasError) {
      // 你可以 render 任何自訂的 fallback UI
      return <h1>Something went wrong.</h1>;
    }

    return this.props.children;
  }
}
```

注意：

`getDerivedStateFromError()` 會在「`render`」期間被呼叫，所以 side effect 是不被允許的。如果你想要使用 side effect 的話，請使用 `componentDidCatch()`。

componentDidCatch() {#componentdidcatch}

```
componentDidCatch(error, info)
```

這個生命週期方法會在某個錯誤被一個 descendant component 拋出後被呼叫。它接受兩個參數：

1. `error` - 被拋出的錯誤。
2. `info` - 一個有 `componentStack` key 的 object，這個 key 包含有那一個 component 拋出錯誤的資訊。

`componentDidCatch()` 會在「commit」期間被呼叫，所以 side effect 是被允許的。這個方法應該被用來做類似記錄錯誤這類的事情：

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }

  static getDerivedStateFromError(error) {
    // 更新 state，如此下次 render 時 React 才能顯示 fallback UI
    return { hasError: true };
  }

  componentDidCatch(error, info) {
    // ComponentStack 的範例：
    //   in ComponentThatThrows (created by App)
    //   in ErrorBoundary (created by App)
    //   in div (created by App)
    //   in App
    logComponentStackToMyService(info.componentStack);
  }

  render() {
    if (this.state.hasError) {
      // 你可以 render 任何自訂的 fallback UI
      return <h1>Something went wrong.</h1>;
    }

    return this.props.children;
  }
}
```

React 在 production 和 development 在處理 `componentDidCatch()` 錯誤的方式不同。

在 development 時，錯誤將會冒泡到 `window`，意思是任何的 `window.onerror` 或 `window.addEventListener('error', callback)` 將攔截透過 `componentDidCatch` 所捕捉到的錯誤。

反之，在 production 時不會冒泡，意思是任何 ancestor 錯誤處理將只會接收未由 `componentDidCatch()` 明確捕捉的錯誤。

注意：

在有錯誤的情況下，你可以透過呼叫 `setState` 來 `render` 一個含有 `componentDidCatch()` 的 fallback UI，但這個方法會在未來的版本中會被棄用。請使用 `static getDerivedStateFromError()` 來處理 fallback `render`。

過時的生命週期方法

以下介紹的幾個生命週期方法是「過時」的。它們仍然能運作，但我們並不建議你在新的程式碼內使用這些方法。你可以在[這篇文章中](#)了解如何從過時的方法 migrate 到我們建議的方法。

UNSAFE_componentWillMount() {#unsafe_componentwillmount}

UNSAFE_componentWillMount()

注意：

這個生命週期方法先前的命名是 `componentWillMount`。這個命名直到第 17 版仍然能繼續運作。請使用重新命名不安全的生命週期方法 [codemod](#) 來自動更新你的 component。

`UNSAFE_componentWillMount()` 會在 mounting 發生前被呼叫。它會在 `render()` 前被呼叫，因此在這個方法內同步呼叫 `setState()` 並不會觸發額外的 `render`。不過，一般情況來說，我們建議你使用 `constructor()` 來初始化 state。

請避免在這個方法中帶入任何的 side effect 或 subscription。如果你需要那樣做的話，請使用 `componentDidMount()`。

這是唯一一個在伺服器端 `render` 時被呼叫的生命週期方法。

UNSAFE_componentWillReceiveProps() {#unsafe_componentwillreceiveprops}

UNSAFE_componentWillReceiveProps(nextProps)

注意：

這個生命週期方法先前的命名是 `componentWillReceiveProps`。這個命名直到第 17 版仍然能繼續運作。請使用重新命名不安全的生命週期方法 [codemod](#) 來自動更新你的 component。

注意：

這個生命週期方法常常會導致 bug 和程式碼的不一致。

- 如果你需要在某個 prop 變更時 執行一個相對應的 **side effect** (例如資料提取或使用動畫)，請使用 `componentDidUpdate` 這個生命週期方法。
- 如果你想要 只在某個 prop 變更時重新計算某些資料，請使用 [memoization helper](#)。
- 如果你想要 在某個 prop 變更時「重置」某個 state，請考慮建立一個完全被控制 的 component 或帶有 key 的完全被控制 component。

針對以上這些使用情境，請遵照這篇文章內推薦的方法來處理繼承的 state。

`UNSAFE_componentWillReceiveProps()` 會在一個被 mount 的 component 接收新的 prop 前被呼叫。如果你需要在某個 prop 變更時更新 state 的話 (例如，重置 state)，你可以在這個生命週期方法裡面比較 `this.props` 和 `nextProps`，並使用 `this.setState()` 進行 state 的轉移。

請注意如果一個 parent component 導致你的 component 重新 render 的話，即使 prop 沒有改變，這個方法仍然會被呼叫。如果你不想要有這些改變的話，請確認你有比較目前和之後的 prop 的值。

React 並不會在初次 `mounting` 時使用初始化的 props 來呼叫 `UNSAFE_componentWillReceiveProps()`。它只會在 component 某些 prop 可能會更新時呼叫這個方法。一般來說，呼叫 `this.setState()` 並不會觸發 `UNSAFE_componentWillReceiveProps()`。

`UNSAFE_componentWillUpdate()` {#unsafe_componentwillupdate}

`UNSAFE_componentWillUpdate(nextProps, nextState)`

注意：

這個生命週期方法先前的命名是 `componentWillUpdate`。這個命名直到第 17 版仍然能繼續運作。請使用 重新命名不安全的生命週期方法 [codemod](#) 來自動更新你的 component。

`UNSAFE_componentWillUpdate()` 會在 render 發生之前、當新的 prop 或 state 正在被接收時被呼叫。請將這個方法作為更新發生之前做準備的一個機會。這個方法並不會在初次 render 時被呼叫。

請注意你不能在這裡呼叫 `this.setState()`，你也不應該在這裡進行其他任何在 `UNSAFE_componentWillUpdate()` 回傳之前會觸發 React component 更新的行為 (例如 dispatch 一個 Redux action)。

通常，這個方法可以被 `componentDidUpdate()` 取代。如果你在這個方法內從 DOM 中讀取資料 (例如儲存滾動軸的位置)，你可以將那部分的邏輯移到 `getSnapshotBeforeUpdate()` 裡面。

注意：

如果 `shouldComponentUpdate()` 回傳 `false` 的話，`UNSAFE_componentWillUpdate()` 將不會被呼叫。

其他的 API

和上述那些由 React 替你呼叫的生命週期方法不同，以下介紹的方法是你可以從你的 component 呼叫的。

只有兩個方法：`setState()` 和 `forceUpdate()`。

`setState()` {#setstate}

`setState(updater[, callback])`

`setState()` 會將改變排進一個 queue 中，並告知 React 這個 component 以及它的 children 需要用更新後的 state 重新 render。這是你會在事件處理和伺服器回應用來更新使用者介面最主要的方法。

請把 `setState()` 想成一個請求而非一個馬上對 component 進行更新的指令。為了達到更好的效能，React 也許會延遲這個請求，然後一次更新數個 component。在極少數情況下，你需要強制同步 DOM 更新，你可以將其包裝在 `flushSync` 中，但這可能會影響效能。

`setState()` 並不會總是馬上更新 component。它有可能會將更新分批處理更新或延遲到稍後才更新。這使得在呼叫 `setState()` 後讀取 `this.state` 成為一個潛在的問題。因此請不要這麼做。相反的，請使用 `componentDidUpdate` 或一個 `setState` callback (`setState(updater, callback)`)。不論你使用哪一個，React 都保證它會在更新後被觸發。如果你需要基於先前的 state 來設定 state 的話，請閱讀以下關於 `updater` 的參數。

除非 `shouldComponentUpdate()` 回傳 `false`，`setState()` 一定會導致重新 render。如果你有使用 mutable object，或者你無法在 `shouldComponentUpdate()` 裡面建立條件式 render 的邏輯的話，只在新的 state 和先前的 state 不同時呼叫 `setState()` 將會避免不必要的重新 render。

這個方法的第一個參數是一個帶有如下的形式的 `updater` function：

`(state, props) => stateChange`

`state` 是當某個改變正在被應用時對 component state 的一個參考。它不應該直接被 mutate。相反的，任何改變都應該用一個基於 `state` 和 `props` 的 input 所建立的新的 object 來表示。例如，假設我們想要使用 `props.step` 來增加 state 中的某個值的話：

```
this.setState((state, props) => {
  return {counter: state.counter + props.step};
});
```

被 `updater` function 所接受的 `state` 和 `props` 兩者都保證一定會被更新到最新的狀態。Updater 的 output 會被和 `state` 淺層合併。

`setState()` 的第二個參數是一個非必要、選擇性的 callback function。它會在 `setState` 完成且 component 被重新 render 後被執行。一般來說如果你要使用這樣的邏輯的話，我們比較推薦你使用 `componentDidUpdate()`。

你可以選擇將一個 object (而非 function) 作為第一個參數傳給 `setState()`：

`setState(stateChange[, callback])`

這會將 `stateChange` 淺層合併至新的 state 中。舉個例子，假設你想調整購物車中物品的數量：

```
this.setState({quantity: 2})
```

這種形式的 `setState()` 也是同步的，而同樣一個週期中的多次呼叫有可能會被結合成一批做處理。例如，假設你想在同一個週期中增加某個物品的數量超過一次的話，這樣做的結果會和以下程式碼相同：

```
Object.assign(
  previousState,
  {quantity: state.quantity + 1},
  {quantity: state.quantity + 1},
  ...
)
```

在同一個週期中，後續的呼叫會覆蓋之前的呼叫所產生的值，所以物品的數量只會被增加一次。如果下個 state 是根據目前的 state 而決定的話，我們比較建議你用 updater function 來更新 state：

```
this.setState((state) => {
  return {quantity: state.quantity + 1};
});
```

想了解更多細節，請參考：

- [State 和生命週期指南](#)
- [深入解析：為什麼 `setState\(\)` 的呼叫會批次處理？什麼時候會如此？](#)
- [深入解析：為什麼 `this.state` 不會馬上被更新？](#)

forceUpdate() {#forceupdate}

```
component.forceUpdate(callback)
```

當你的 component 的 state 或 prop 變更的時候，你的 component 的預設行為是會重新 render。如果你的 `render()` 方法還需要其他資料的話，你可以藉由呼叫 `forceUpdate()` 來告訴 React 這個 component 需要重新 render。

呼叫 `forceUpdate()` 會導致 `render()` 被呼叫於該 component 並跳過 `shouldComponentUpdate()`。這會觸發 children component 正常的生命週期方法，包含每個 child 的 `shouldComponentUpdate()` 方法。React 依然只會在 markup 變更時更新 DOM。

正常情況來說你應該避免使用 `forceUpdate()` 並只從 `render()` 中的 `this.props` 和 `this.state` 讀取。

Class 屬性

defaultProps {#defaultprops}

`defaultProps` 可以在一個 component class 自身被定義為一個屬性，它被用來設定該 class 的預設 props。它是為了 `undefined` (而非 `null`) 的 prop 使用的。例如：

```
class CustomButton extends React.Component {
  // ...
}

CustomButton.defaultProps = {
  color: 'blue'
};
```

如果 `props.color` 的值沒有被提供的話，它會被預設為 `'blue'`：

```
render() {
  return <CustomButton />; // props.color 被預設為 blue
}
```

如果 `props.color` 的值被設為 `null`，其值會繼續為 `null`：

```
render() {
  return <CustomButton color={null} />; // props.color 繼續為 null
}
```

displayName {#displayname}

`displayName` string 是用來 debug 的。通常，你不需要明確的設定這個屬性，因為它可以根據定義該 component 的 function 或 class 的名稱推斷這個值為何。當你為了 debug 或建立一個 higher-order component 而需要展示一個不同的名字時，你可能會想要明確的設定這個值，請參考[如何包覆 Display Name 並輕鬆 debug](#)。

Instance 屬性

props {#props}

`this.props` 包含了該 component 的呼叫者所定義的 prop。想了解 prop 的基礎概念，請參考[Components](#) 和[Props](#)文件。

值得注意的是，`this.props.children` 是一個特別的 prop，通常在 JSX 表達式內的 child tag 內所定義，而不是其自身的 tag。

state {#state}

State 包含了某個 component 內特定的、會隨時間改變的資料，這個 state 是由使用者定義的。它應是一個簡單的 JavaScript object。

如果某個值並沒有在 render 或資料流中被使用（例如計時器的 ID），你不需要將它放在 state 內。這類的值可以在 component 實例上被定義為 field。

想更了解 state 如何運作，請參考[State 和生命週期](#)。

請永遠不要直接 mutate `this.state`，因為後續的 `setState()` 會替換掉你的 mutation。請將 `this.state` 視為不可變的。

[Go to TOC](#)

ReactDOMClient

The `react-dom/client` package provides client-specific methods used for initializing an app on the client. Most of your components should not need to use this module.

```
import * as ReactDOM from 'react-dom/client';
```

If you use ES5 with npm, you can write:

```
var ReactDOM = require('react-dom/client');
```

Overview

The following methods can be used in client environments:

- `createRoot()`
- `hydrateRoot()`

Browser Support

React supports all modern browsers, although [some polyfills are required](#) for older versions.

Note

We do not support older browsers that don't support ES5 methods or microtasks such as Internet Explorer. You may find that your apps do work in older browsers if polyfills such as [es5-shim](#) and [es5-sham](#) are included in the page, but you're on your own if you choose to take this path.

Reference

`createRoot()` [\[#createroot\]](#)

```
createRoot(container[, options]);
```

Create a React root for the supplied `container` and return the root. The root can be used to render a React element into the DOM with `render`:

```
const root = createRoot(container);
root.render(element);
```

`createRoot` accepts two options:

- `onRecoverableError`: optional callback called when React automatically recovers from errors.
- `identifierPrefix`: optional prefix React uses for ids generated by `React.useId`. Useful to avoid conflicts when using multiple roots on the same page. Must be the same prefix used on the server.

The root can also be unmounted with `unmount`:

```
root.unmount();
```

Note:

`createRoot()` controls the contents of the container node you pass in. Any existing DOM elements inside are replaced when `render` is called. Later calls use React's DOM diffing algorithm for efficient updates.

`createRoot()` does not modify the container node (only modifies the children of the container). It may be possible to insert a component to an existing DOM node without overwriting the existing children.

Using `createRoot()` to hydrate a server-rendered container is not supported. Use `hydrateRoot()` instead.

hydrateRoot() {#hydrateroot}

```
hydrateRoot(container, element[, options])
```

Same as `createRoot()`, but is used to hydrate a container whose HTML contents were rendered by `ReactDOMServer`. React will attempt to attach event listeners to the existing markup.

`hydrateRoot` accepts two options:

- `onRecoverableError`: optional callback called when React automatically recovers from errors.
- `identifierPrefix`: optional prefix React uses for ids generated by `React.useId`. Useful to avoid conflicts when using multiple roots on the same page. Must be the same prefix used on the server.

Note

React expects that the rendered content is identical between the server and the client. It can patch up differences in text content, but you should treat mismatches as bugs and fix them. In development mode, React warns about mismatches during hydration. There are no guarantees that attribute differences will be patched up in case of mismatches. This is important for performance reasons because in most apps, mismatches are rare, and so validating all markup would be prohibitively expensive.

[Go to TOC](#)

ReactDOMServer

`ReactDOMServer` object 使你能將 component render 至靜態標記。它通常是用在 Node 伺服器上：

```
// ES modules
import * as ReactDOMServer from 'react-dom/server';
// CommonJS
var ReactDOMServer = require('react-dom/server');
```

概觀

These methods are only available in the **environments with Node.js Streams**:

- `renderToPipeableStream()`
- `renderToNodeStream()` (Deprecated)
- `renderToStaticNodeStream()`

These methods are only available in the **environments with Web Streams** (this includes browsers, Deno, and some modern edge runtimes):

- `renderToReadableStream()`

The following methods can be used in the environments that don't support streams:

- `renderToString()`
- `renderToStaticMarkup()`

Reference

`renderToPipeableStream()` {#rendertopipeablestream}

`ReactDOMServer.renderToPipeableStream(element, options)`

Render a React element to its initial HTML. Returns a stream with a `pipe(res)` method to pipe the output and `abort()` to abort the request. Fully supports Suspense and streaming of HTML with "delayed" content blocks "popping in" via inline `<script>` tags later. [Read more](#)

If you call `ReactDOM.hydrateRoot()` on a node that already has this server-rendered markup, React will preserve it and only attach event handlers, allowing you to have a very performant first-load experience.

```
let didError = false;
const stream = renderToPipeableStream(
  <App />,
  {
    onShellReady() {
      // The content above all Suspense boundaries is ready.
      // If something errored before we started streaming, we set the error code
      // appropriately.
      res.statusCode = didError ? 500 : 200;
    }
  }
);
```

```

    res.setHeader('Content-type', 'text/html');
    stream.pipe(res);
},
onShellError(error) {
  // Something errored before we could complete the shell so we emit an
  // alternative shell.
  res.statusCode = 500;
  res.send(
    '<!doctype html><p>Loading...</p><script src="clientrender.js"></script>'
  );
},
onAllReady() {
  // If you don't want streaming, use this instead of onShellReady.
  // This will fire after the entire page content is ready.
  // You can use this for crawlers or static generation.

  // res.statusCode = didError ? 500 : 200;
  // res.setHeader('Content-type', 'text/html');
  // stream.pipe(res);
},
onError(err) {
  didError = true;
  console.error(err);
},
}
);

```

See the [full list of options](#).

Note:

This is a Node.js-specific API. Environments with [Web Streams](#), like Deno and modern edge runtimes, should use `renderToReadableStream` instead.

`renderToReadableStream()` [#rendertoreadablestream](#)

`ReactDOMServer.renderToReadableStream(element, options);`

Streams a React element to its initial HTML. Returns a Promise that resolves to a [Readable Stream](#). Fully supports Suspense and streaming of HTML. [Read more](#)

If you call `ReactDOM.hydrateRoot()` on a node that already has this server-rendered markup, React will preserve it and only attach event handlers, allowing you to have a very performant first-load experience.

```

let controller = new AbortController();
let didError = false;
try {
  let stream = await renderToReadableStream(
    <html>
      <body>Success</body>
    </html>,
  {
    signal: controller.signal,
    onError(error) {

```

```

        didError = true;
        console.error(error);
    }
);

// This is to wait for all Suspense boundaries to be ready. You can uncomment
// this line if you want to buffer the entire HTML instead of streaming it.
// You can use this for crawlers or static generation:

// await stream.allReady;

return new Response(stream, {
    status: didError ? 500 : 200,
    headers: {'Content-Type': 'text/html'},
});
} catch (error) {
    return new Response(
        '<!doctype html><p>Loading...</p><script src="clientrender.js"></script>',
    {
        status: 500,
        headers: {'Content-Type': 'text/html'},
    }
);
}
}

```

See the [full list of options](#).

Note:

This API depends on [Web Streams](#). For Node.js, use `renderToPipeableStream` instead.

`renderToNodeStream()` (Deprecated)

 `ReactDOMServer.renderToNodeStream(element)`

這個方法會將一個 React element render 至其初始的 HTML。它會回傳一個 [Node.js Readable stream](#)並輸出為一個 HTML string。通過 stream 輸出的 HTML 和 `ReactDOMServer.renderToString` 回傳的 HTML 完全相同。你可以使用這個方法在伺服器端產生 HTML，並在初次請求時傳遞 markup，以加快頁面載入速度，並讓搜尋引擎爬取你的頁面以達到 SEO 最佳化的效果。

如果你在一個已經有伺服器端 render markup 的 node 上呼叫 `ReactDOM.hydrateRoot()`，React 將會保留這個 node 並只附上事件處理，這使你能有一個高效能的初次載入體驗。

注意：

這個 API 只在伺服器端有用。你無法在瀏覽器中使用此 API。

這個方法回傳的 stream 將會回傳一個由 utf-8 編碼的 byte stream。如果你需要另一種編碼的 stream，請參考像是 [iconv-lite](#) 這種為轉換文本提供轉換 stream 的專案。

`renderToStaticNodeStream()` {#rendertostaticnodestream}

ReactDOMServer.`renderToStaticNodeStream(element)`

這個方法和 `renderToNodeStream` 很相似，不過這個方法不會建立那些額外 React 內部使用的 DOM attribute，像是 `data-reactroot`。這個方法在你想要用 React 作為一個簡單的靜態頁面生成器時很有用，因為去除一些額外的 attribute 可以省去一些位元組。

通過這個 stream 輸出的 HTML 和 `ReactDOMServer.renderToStaticMarkup` 回傳的 HTML 完全相同。

如果你打算在前端使用 React 以使得 markup 有互動性的話，請不要使用這個方法。請在伺服器端使用 `renderToNodeStream` 並在前端使用 `ReactDOM.hydrateRoot()`。

注意：

這個 API 只在伺服器端使用。你無法在瀏覽器中使用此 API。

這個方法回傳的 stream 將會回傳一個由 utf-8 編碼的 byte stream。如果你需要另一種編碼的 stream，請參考像是 [iconv-lite](#) 這種為轉換文本提供轉換 stream 的專案。

`renderToString()` {#rendertostring}

ReactDOMServer.`renderToString(element)`

Render a React element to its initial HTML. React will return an HTML string. You can use this method to generate HTML on the server and send the markup down on the initial request for faster page loads and to allow search engines to crawl your pages for SEO purposes.

If you call `ReactDOM.hydrateRoot()` on a node that already has this server-rendered markup, React will preserve it and only attach event handlers, allowing you to have a very performant first-load experience.

Note

This API has limited Suspense support and does not support streaming.

On the server, it is recommended to use either `renderToPipeableStream` (for Node.js) or `renderToReadableStream` (for Web Streams) instead.

renderToStaticMarkup() {#rendertostaticmarkup}

ReactDOMServer.renderToStaticMarkup(element)

Similar to `renderToString`, except this doesn't create extra DOM attributes that React uses internally, such as `data-reactroot`. This is useful if you want to use React as a simple static page generator, as stripping away the extra attributes can save some bytes.

If you plan to use React on the client to make the markup interactive, do not use this method. Instead, use `renderToString` on the server and `ReactDOM.hydrateRoot()` on the client.

ReactDOM

`react-dom` 提供了 DOM 的特定方法讓你可以在你的應用程式頂層使用，如果有需要，也可以作為一個逃生窗口來脫離 React 模型。

```
import * as ReactDOM from 'react-dom';
```

如果你使用 ES5 與 npm，你可以這樣寫：

```
var ReactDOM = require('react-dom');
```

`react-dom` package 也提供了特定於 client 和 server 應用程式的 module：

- `react-dom/client`
- `react-dom/server`

概覽

`react-dom` package export 這些方法：

- `createPortal()`
- `flushSync()`

這些 `react-dom` 方法也被 export，但是被視為是 legacy：

- `render()`
- `hydrate()`
- `findDOMNode()`
- `unmountComponentAtNode()`

注意：

`render` 與 `hydrate` 兩者在 React 18 已經替換為新的 [client 方法](#)。如果你使用像是 React 17 方式，這些方法將警告你的應用程式的行為。（[從這裡了解更多](#)）。

瀏覽器支援

React 支援所有主流瀏覽器，雖然對於一些舊版瀏覽器需要一些 [polyfill](#)。

注意：

我們不支援那些較舊不支援 ES5 方法或是 microtasks 的瀏覽器，像是 Internet Explorer。但如果頁面上引入了像是 [es5-shim](#) 和 [es5-sham](#) 等 polyfill 你可能會發現你的應用程式在較舊的瀏覽器上仍可使用，但如果你選擇了這條路你就只能靠你自己了。

參考

`createPortal()` [\[#createportal\]](#)

`createPortal(child, container)`

提供了一個方式將 children `render` 存在於 DOM component 層次結構之外的 DOM 節點中。

`flushSync()` [\[#flushsync\]](#)

`flushSync(callback)`

強制 React 同步 flush 提供的 callback 中的任何更新。這可以確保 DOM 被立即的更新。

```
// Force this state update to be synchronous.
flushSync(() => {
  setCount(count + 1);
});
// By this point, DOM is updated.
```

注意：

`flushSync` 可能會對效能產生重大的影響。謹慎使用。

`flushSync` 可能會迫使 pending 的 Suspense boundary 顯示 `fallback` 狀態。

`flushSync` 也可以執行 pending effects，並在回傳之前同步 apply 它們包含的任何更新。

`flushSync` `flushSync` 在必要時也可以在 callback 外 flush 更新，以 flush callback 內的更新。例如，如果有來自 click 的 pending 更新，React 可能在 flush callback 中的更新之前 flush 這些內容。

Legacy Reference

`render()` [\[#render\]](#)

`render(element, container[, callback])`

注意：

`render` 在 React 18 已經被 `createRoot` 取代。更多資訊請參考 [createRoot](#)。

將一個 React element render 到提供提供的 `container` 的 DOM 中，並回傳一個該 component 的 `reference` (對於 `stateless components` 回傳 `null`)。

如果 React element 之前已經在 `container` 內被 render，它只會執行更新並 mutate 必要的 DOM，來呈現最新的 React element。

如果提供了可選的 callback，它將會在 component 被 render 或更新之後，才被執行。

注意：

`render()` 控制了你傳入到 `container` 內的 node 內容。當第一次被呼叫時，任何存在於 `container` 的 DOM element 都會被替換。之後的呼叫會使用 React 的 DOM diffing 演算法進行高效率的更新。

`render()` 不修改 `container` 的 node (只修改 `container` 的 children)。它可以將 component 插入一個現有的 DOM node 而不用覆蓋已經存在的 children。

`render()` 目前回傳一個 reference 到 root `ReactComponent` instance。然而，使用這個回傳值是被遺留的方式 並且應該被避免，因為未來版本的 React 在某些情況下可能會非同步地 render component。如果你需要 reference 到 root `ReactComponent` instance，首選的解決方式是附加一個 `callback ref` 在 root element 上。

使用 `render()` 來 hydrate 一個 server-render container 已經被棄用。使用 `hydrateRoot()` 作為代替。

hydrate() {#hydrate}

`hydrate(element, container[, callback])`

注意：

`hydrate` 在 React 18 已經被 `hydrateRoot` 取代。更多資訊請參考 [hydrateRoot](#)。

如同 `render()`，但它是被用來 hydrate 一個 container，其 HTML 內容是由 `ReactDOMServer` render。React 將嘗試將 event listener attach 到現有 markup 上。

React 預期在伺服器端和客戶端所 render 的內容是相同的。它可以修補 text content 的差異，但你應該把不匹配的部分視為 bug 並且修正。在開發模式中，React 會警告關於 hydration 過程中的不匹配。在不匹配的情況下，將無法保證 attribute 的差異會被修補。這對於效能來說很重要，因為在大部分的應用程式中，不匹配的情況很少見，也因此驗證要所有 markup 的成本非常高。

如果在伺服器端和客戶端某個 element 的 attribute 或 text content 無可避免的不相同 (例如，時間戳)，你可以透過加入 `suppressHydrationWarning={true}` 到 element 來關閉警告。這個只有在第一層時有效並且傾向於應急的做法。不要過度使用它。除非它是 text content 否則 React 仍然不會嘗試對其進行修補，所以在未來更新之前它可能會保持不一致。

如果你刻意要在服務端和客戶端上 render 不同的內容，你可以進行兩次的 render。在客戶端上呈現不同內容的 component 可以透過讀取一個 state 變數像是 `this.state.isClient` 之後在 `componentDidMount()` 內把它設定成 `true`。這樣初始 render 將跟伺服器端 render 的內容一樣，從而避免不匹配，但在 hydrate 之後將會立即同步額外的程序。請注意，此方法會使你的 component 變慢，因為它必須被 render 兩次，因此請謹慎使用。

請記得要留意連線緩慢的使用者體驗。JavaScript 載入顯然比 HTML 首次 render 要晚得多，因此，如果你只有在客戶端 render 一些不同的東西，則轉換可能會不穩定。然而，如果執行順利的話，在伺服器上 render 應用程式的「shell」可能是有幫助的，而且只顯示一些額外的插件在客戶端。要了解如何執行此操作而不會出現 markup 不匹配的問題，請參考上一個段落的說明。

`unmountComponentAtNode()` {#unmountcomponentatnode}

`unmountComponentAtNode(container)`

注意：

`unmountComponentAtNode` 在 React 18 已經被 `root.unmount()` 取代。更多資訊請參考 [createRoot](#)。

從 DOM 移除一個 mount React component 並清除它的 event handler 以及 state。如果沒有 component 被 mount 在 container 的話，呼叫這個 function 並不會做任何事。如果一個 component 被 unmount 回傳一個 `true`，反之如果沒有 component 被 unmount，回傳 `false`。

`findDOMNode()` {#finddomnode}

注意：

`findDOMNode` 是一個用來存取底層 DOM node 應急的做法。在大多數情況下，不鼓勵使用這個應急的做法因為它會穿透 component 抽象化。[它已經在 StrictMode 中被棄用了](#)。

`findDOMNode(component)`

如果這個 component 已經被 mount 到 DOM，則回傳對應原生瀏覽器的 DOM element。這個方法對於從 DOM 中讀取值是有用的，像是表單的欄位值和執行 DOM 的測量。**在大多數情況下，你可以附加一個 ref 給 DOM node 來避免使用 `findDOMNode`。**

當一個 component render 成 `null` 或 `false` 時，`findDOMNode` 回傳 `null`。當一個 component render 成字符串時，`findDOMNode` 回傳一個包含該值的文字 DOM node。從 React 16 開始，component 可以回傳包含很多 children 的 fragment，在這種情況下 `findDOMNode` 將會回傳和第一個非空的子節點相對應的 DOM node。

注意：

`findDOMNode` 只在已經 `mount` 的 `component` 上有用（即已放置在 DOM 中的 `component`）。如果你嘗試在尚未 `mount` 的 `component` 上呼叫它（比如在尚未建立的 `component` 的 `render()` 中調用 `findDOMNode()`），將會拋出異常。

`findDOMNode` 不能被用在 `function component`。

React 頂層 API

`React` 是 React 函式庫的進入點。如果你使用 `<script>` 標籤載入 React, 這些頂層 API 可以在 `React` 全域變數使用。如果你使用 ES6 撰寫並使用 npm, 你可以寫成 `import React from 'react'`。如果你使用 ES5 撰寫並使用 npm, 你可以寫成 `var React = require('react')`。

概觀

Component

React component 可以讓你把 UI 切分為獨立並可重複使用的單位，並且每個單位可以抽出來獨立思考。React component 可以透過繼承 `React.Component` 或是 `React.PureComponent` 定義。

- `React.Component`
- `React.PureComponent`

如果你沒有使用 ES6 class, 你可以使用 `create-react-class` 模組。請參閱[使用 React 但不使用 ES6](#) 取得更多資訊。

React component 也可以定義為 function 並可以被封裝：

- `React.memo`

建立 React Element

我們推薦[使用 JSX](#) 來描述你的 UI 應該長成什麼樣子。每個 JSX element 都只是呼叫 `React.createElement()` 的語法糖。當你使用 JSX 的時候你將不需要直接呼叫以下的 method：

- `createElement()`
- `createFactory()`

參閱[使用 React 但不使用 JSX](#) 取得更多資訊。

操作 Element

`React` 提供了多種 API 讓你可以操作 element：

- `cloneElement()`
- `isValidElement()`
- `React.Children`

Fragment

`React` 也提供了一個 component 讓你一次 render 多個 element 而不需要額外的 wrapper。

- `React.Fragment`

Refs

- `React.createRef`
- `React.forwardRef`

Suspense

`Suspense` 讓 components 在 `render` 之前可以「暫停」並等待其他事情。目前 `Suspense` 只支援一個情境：[使用 `React.lazy` 動態載入 component](#)。在未來，我們也會支援像是抓取資料等更多的使用情境。

- `React.lazy`
- `React.Suspense`

Transitions

`Transitions` 是 React 18 引入的新 concurrent 功能。它們允許你 mark 更新為 transitions，這是告訴 React 它們可以被中斷，並且對於已經可看見的內容，避免回到 Suspense fallback。

- `React.startTransition`
- `React.useTransition`

Hooks

`Hooks` 是 React 16.8 開始的新功能。這讓你可以使用 `state` 以及其他 React 功能而不需要撰寫一個 class。Hooks 有一個專屬的[文件專區](#) 和分開的 API 參考資料：

- [基本 Hooks](#)
 - `useState`
 - `useEffect`
 - `useContext`
 - [進階 Hooks](#)
 - `useReducer`
 - `useCallback`
 - `useMemo`
 - `useRef`
 - `useImperativeHandle`
 - `useLayoutEffect`
 - `useDebugValue`
 - `useDeferredValue`
 - `useTransition`
 - `useId`
 - [Library Hooks](#)
 - `useSyncExternalStore`
 - `useInsertionEffect`
-

參考資料

React.Component {#reactcomponent}

`React.Component` 是當你使用 [ES6 classes](#) 定義 React component 時所用的 base class：

```
class Greeting extends React.Component {
  render() {
    return <h1>Hello, {this.props.name}</h1>;
  }
}
```

參閱 [React.Component API 參考資料](#) 可查到跟 `React.Component` 相關的 method 及 properties。

React.PureComponent {#reactpurecomponent}

`React.PureComponent` 跟 `React.Component` 很相似。他們之間的差別是 `React.Component` 並沒有實作 `shouldComponentUpdate()`，但 `React.PureComponent` 提供了一個實作以對於 prop 及 state 進行 shallow compare。

備註

`React.PureComponent` 的 `shouldComponentUpdate()` 只對 object 進行 shallow compare。如果這些 object 包含複雜的資料結構，在深層的資料有所改變的時候將有可能回傳錯誤結果 (false-negative)。繼承 `PureComponent` 的時候，請確保你只有簡單的 prop 跟 state，或在當你知道深層的資料有所改變的時候使用 `forceUpdate()`。你也可以考慮改用 [immutable objects](#) 進行快速的深層資料比較。

此外，`React.PureComponent` 的 `shouldComponentUpdate()` 將會跳過整個 subtree 的 prop 更新。請確保所有 children component 也是「pure」的。

React.memo {#reactmemo}

```
const MyComponent = React.memo(function MyComponent(props) {
  /* render using props */
});
```

`React.memo` 是一個 [higher order component](#)。

如果你的 function component 每次得到相同 prop 的時候都會 render 相同結果，你可以將其包在 `React.memo` 之中，透過快取 render 結果來在某些情況下加速。這表示 React 會跳過 render 這個 component，並直接重用上次的 render 結果。

`React.memo` 只會確認 props 的改變。如果你的 function component 被 wrap 在 `React.memo` 內，實作中具有一個 `useState`、`useReducer` 或 `useContext` Hook，當 state 或 context 變更時，它仍然會持續 rerender。

這預設只會對 prop 進行 shallow compare。如果你需要控制比較的方法，你可以提供一個自訂的比較 function 作為第二個參數。

```
function MyComponent(props) {
  /* render using props */
}

function areEqual(prevProps, nextProps) {
  /*
    return true if passing nextProps to render would return
    the same result as passing prevProps to render,
    otherwise return false
  */
}

export default React.memo(MyComponent, areEqual);
```

這個 function 是用來作為[效能最佳化](#)所用。請勿依賴這個 function 來「避免」render，這可能會產生 bug。

備註

與 class component 的 `shouldComponentUpdate()` method 不同，`areEqual` function 當 prop 相等的時候回傳 `true`，不相等的時候回傳 `false`。這跟 `shouldComponentUpdate` 剛好相反。

`createElement()` [#createelement](#)

```
React.createElement(
  type,
  [props],
  [...children]
)
```

建立並回傳一個新的 `React element` 描述指定的 `type`。Type 參數可以是一個標籤名稱字串（像是 `'div'` 或是 `'span'`）、一個 `React component` type (class 或是 function) 或者是一個 `React fragment` type。

使用 `JSX` 寫的程式將會被轉換使用 `React.createElement()`。如果使用 `JSX`，你通常不需要自己呼叫 `React.createElement()`。請參閱[使用 React 但不使用 JSX](#)。

`cloneElement()` [#cloneelement](#)

```
React.cloneElement(
  element,
  [config],
  [...children]
)
```

使用 `element` 作為開始 clone 並回傳一個新 `React element`。`config` 應該包含所有新的 `props`、`key`、或 `ref`。產生的 `element` 將會有原始 `element` 的 `props` 以及新 `props` 的 shallow merge。新的 `children` 會取代現有的 `children`。如果 `config` 中不存在 `key` 和 `ref`，則原始的 `key` 和 `ref` 會被保留。

`React.cloneElement()` 幾乎等於：

```
<element.type {...element.props} {...props}>{children}</element.type>
```

但是，這同時也會保留 `ref`。如果你有一個 child 上有 `ref` 的時候，你將不會不小心從你的上層元件偷走 `ref`。你的新 element 會保留一樣的 `ref`。如果有存在 `ref` 或 `key` 的話，將會取代舊的。

這個 API 是用來取代目前已經過時的 `React.addons.cloneWithProps()`。

`createFactory()` [\(#createfactory\)](#)

```
React.createFactory(type)
```

回傳一個 function 可以產生指定 type 的 React element。跟 `React.createElement()` 一樣，這個 type 參數可以是一個標籤名稱字串（像是 `'div'` 或是 `'span'`）、一個 React component type (class 或是 function) 或者是一個 React fragment type。

這個 helper 已經被認定為過時，我們建議你使用 JSX 或是直接使用 `React.createElement()`。

如果使用 JSX，你通常不需要自己呼叫 `React.createFactory()`。請參閱[使用 React 但不使用 JSX](#)。

`isValidElement()` [\(#isvalidelement\)](#)

```
React.isValidElement(object)
```

檢查一個 object 是否為 React element。通常回傳 `true` 或是 `false`。

`React.Children` [\(#reactchildren\)](#)

`React.Children` 提供了一些工具可以將 `this.props.children` 作為不透明的資料結構處理。

`React.Children.map` [\(#reactchildrenmap\)](#)

```
React.Children.map(children, function[(thisArg)])
```

對每一個列為 `children` 之中的直接 child 呼叫 function 並將 `this` 設定為 `thisArg`。如果 `children` 是一個 array，這將會列舉整個 array 並對每一個 child 呫叫這個 function。如果 `children` 是 `null` 或是 `undefined`，這個 method 將會回傳 `null` 或是 `undefined` 而不是一個 array。

備註

如果 `children` 是一個 Fragment，它將會被視為只有一個 child 而不會繼續深入列舉。

`React.Children.forEach` [\(#reactchildrenforeach\)](#)

```
React.Children.forEach(children, function[(thisArg)])
```

跟 `React.Children.map()` 一樣，但不會回傳一個 array。

React.Children.count {#reactchildrencount}**React.Children.count(children)**

回傳 `children` 到底有幾個 child，跟傳入 `map` 或是 `forEach` 的 callback 會被呼叫的次數一致。

React.Children.only {#reactchildrenonly}**React.Children.only(children)**

確認 `children` 只有一個 child (一個 React element) 並回傳它，不然這個 method 將會拋出錯誤。

備註

`React.Children.only()` 無法接受 `React.Children.map()` 的回傳值，因為那個回傳值將會是一個 array 而不是一個 React element。

React.Children.toArray {#reactchildrentoarray}**React.Children.toArray(children)**

將 `children` 這個不透明的資料結構轉為一個扁平的 array 並對每個 child 指定一個 key。如果你想要在你的 render method 中操作 children 的集合時非常有用，特別是當你想要在傳遞它們之前調整順序或是擷取一部份 `this.props.children` 的時候。

備註

`React.Children.toArray()` 會改變 key 來保留巢狀 array 的語意。也就是說 `toArray` 將會在每個 key 前面加入前綴，確保每個 element 的 key 都在與它原本輸入的 array 相關。

React.Fragment {#reactfragment}

`React.Fragment` 讓你可以在一個 `render()` method 中一次回傳多個 element 而不需要建立一個額外的 DOM element：

```
render() {
  return (
    <React.Fragment>
      Some text.
      <h2>A heading</h2>
    </React.Fragment>
  );
}
```

你也可以使用 `<></>` 的精簡表示法。請參閱 [React v16.2.0：更好的 Fragment 支援](#)以獲得更多資訊。

React.createRef {#reactcreateref}

`React.createRef` 會建立一個 `ref` 以透過 `ref` attribute 夾帶在一個 React element 之上。 `embed:16-3-release-blog-post/create-ref-example.js`

React.forwardRef {#reactforwardref}

`React.forwardRef` 會建立一個 React component 並將 `ref` attribute 轉交給旗下的另外一個 component。這個技巧不是很常被使用，但在以下兩個情況很適合：

- 將 `ref` 轉交給 DOM component
- 在 higher-order-component 之中轉交 `ref`

`React.forwardRef` 接受一個 render function 作為參數。React 會呼叫這個 function 並傳入兩個參數：`props` 以及 `ref`。這個 function 應該要回傳一個 React node。

`embed:reference-react-forward-ref.js`

在上面這個範例，React 會把給予 `<FancyButton ref={ref}>` 的 `ref` 轉交給傳入 `React.forwardRef` 的 function 作為其第二個參數。這個 render function 接著將這個 `ref` 轉交給 `<button ref={ref}>` element。

結果，當 React 夾上這個 `ref` 的時候，`ref.current` 將會直接指到 `<button>` 這個 DOM element 實例。

請參閱[轉交 ref](#) 獲得更多資訊。

React.lazy {#reactlazy}

`React.lazy()` 讓你可以定義一個動態載入的 component。這可以在初始 render 期間延緩載入沒有被用到的 component 來減少 bundle size。

你可以閱讀我們的 [code splitting 文件](#) 來學習怎麼使用它。你可能也想要閱讀[這篇文章](#) 更深入了解如何使用這個 function。

```
// This component is loaded dynamically
const SomeComponent = React.lazy(() => import('./SomeComponent'));
```

請注意當你使用 `lazy` component 的時候，你的 render tree 上層中必須包含一個 `<React.Suspense>` 來指定 loading indicator。

React.Suspense {#reactsuspense}

`React.Suspense` 讓你指定 loading indicator，避免下面 tree 中的某些 component 尚未準備好 render。未來我們計畫讓 `Suspense` 處理更多場景，像是資料的取得。你可以閱讀關於這個在[我們的 roadmap](#)。

現在，`lazy` loading component 是 `<React.Suspense>` 唯一支援的使用場景：

```
// This component is loaded dynamically
const OtherComponent = React.lazy(() => import('./OtherComponent'));

function MyComponent() {
  return (
    // Displays <Spinner> until OtherComponent loads
```

```
<React.Suspense fallback={<Spinner />}>
  <div>
    <OtherComponent />
  </div>
</React.Suspense>
);
}
```

在我們的 [code splitting](#) 文件 有更多資訊。請注意 `lazy` component 可以在 `Suspense` tree 中底下很多層 — 你不需要把每一個 `lazy` 元素包起來。最好的方法是將 `<Suspense>` 放在你想看到 loading indicator 的地方，而在所有你想進行 code splitting 的地方使用 `lazy()`。

注意：

對於已經顯示給使用者的內容，切換回 loading indicator 可能會讓人迷惑。有時候在新的 UI 準備好之前，顯示「舊」的 UI 會來得更好。若要達成這個方式，你可以使用新的 `startTransition` 和 `useTransition` transition APIs，來 mark 更新為一個 transitions，並且避免不預期的 fallback。

`React.Suspense` 是 Server Side Rendering

在 server side rendering 期間，`Suspense` Boundaries 允許你透過 suspending 以更小的 chunks 來 flush 應用程式。當一個 component suspends 時，我們 schedule 一個低優先級的 task 來 render 最近的 `Suspense` boundary 的 fallback。如果 component 在我們 flush fallback 之前 unsuspends，那麼我們把實際的內容傳送下去，並丟棄 fallback。

`React.Suspense` during hydration

`Suspense` boundary 依賴於它們的 parent boundary 在它們被 hydrate 之前被 hydrate，但它們可以獨立於 sibling boundary hydrate。在被 hydrate 之前發生的 event 將會導致 boundary hydrate 的優先級高於相鄰的 boundary。[閱讀更多](#)。

`React.startTransition` {#starttransition}

`React.startTransition(callback)`

這個方法被設計的是被用在當 `React.useTransition` 不可使用時。

注意：

在 transition 中的更新會產生更緊急的更新，例如：點擊。

在 transition 中的更新不會顯示 re-suspended 的 fallback 內容，允許使用者可以在更新期間繼續互動。

`React.startTransition` 不提供一個 `isPending` 的 flag。若要追蹤 transition 的 pending 狀態，參考 `React.useTransition`。

Test Renderer

如何 Import

```
import TestRenderer from 'react-test-renderer'; // ES6
const TestRenderer = require('react-test-renderer'); // ES5 with npm
```

概覽

這個 package 提供了一個 React renderer，可以被用於 render React component 成 pure JavaScript object，無需依賴 DOM 或原生的行動裝置環境。

基本上，這個 package 提供的主要功能是在不依賴瀏覽器或 `jsdom` 的情況下，回傳某個時間點由 React DOM 或是 React Native component render 出的 view 結構（類似 DOM tree）snapshot。

範例：

```
import TestRenderer from 'react-test-renderer';

function Link(props) {
  return <a href={props.page}>{props.children}</a>;
}

const testRenderer = TestRenderer.create(
  <Link page="https://www.facebook.com/">Facebook</Link>
);

console.log(testRenderer.toJSON());
// { type: 'a',
//   props: { href: 'https://www.facebook.com/' },
//   children: [ 'Facebook' ] }
```

你可以使用 Jest 的 snapshot 測試功能來自動儲存目前 JSON tree 到一個文件中，並在測試中檢查它是否被修改：[了解更多](#)。

你也可以通過遍歷輸出來尋找特定的 node，並對它們進行 assert。

```
import TestRenderer from 'react-test-renderer';

function MyComponent() {
  return (
    <div>
      <SubComponent foo="bar" />
      <p className="my">Hello</p>
    </div>
  )
}

function SubComponent() {
  return (
    <p className="sub">Sub</p>
  );
}
```

```
const testRenderer = TestRenderer.create(<MyComponent />);
const testInstance = testRenderer.root;

expect(testInstance.findByType(SubComponent).props.foo).toBe('bar');
expect(testInstance.findByProps({className: "sub"}).children).toEqual(['Sub']);
```

TestRenderer

- `TestRenderer.create()`
- `TestRenderer.act()`

TestRenderer instance

- `testRenderer.toJSON()`
- `testRenderer.toTree()`
- `testRenderer.update()`
- `testRenderer.unmount()`
- `testRenderer.getInstance()`
- `testRenderer.root`

TestInstance

- `testInstance.find()`
- `testInstance.findByType()`
- `testInstance.findByProps()`
- `testInstance.findAll()`
- `testInstance.findAllByType()`
- `testInstance.findAllByProps()`
- `testInstance.instance`
- `testInstance.type`
- `testInstance.props`
- `testInstance.parent`
- `testInstance.children`

Reference

`TestRenderer.create()` {#testrenderercreate}

```
TestRenderer.create(element, options);
```

透過傳來的 React element 建立一個 `TestRenderer` instance。它不使用真實的 DOM，但是它依然將 component tree 完整地 render 到記憶體中，以便於你對它進行 assert。回傳為一個 `TestRenderer` instance。

`TestRenderer.act()` {#testrendereract}

```
TestRenderer.act(callback);
```

近似於 測試工具 中的 `act()` helper, `TestRenderer.act` 會為了 assertions 準備一個 component。使用這個版本的 `act()` 來統合 `TestRenderer.create` 和 `testRenderer.update` 的使用。

```
import {create, act} from 'react-test-renderer';
import App from './app.js'; // The component being tested

// render the component
let root;
act(() => {
  root = create(<App value={1}>/);
});

// make assertions on root
expect(root.toJSON()).toMatchSnapshot();

// update with some different props
act(() => {
  root.update(<App value={2}>);
})

// make assertions on root
expect(root.toJSON()).toMatchSnapshot();
```

`testRenderer.toJSON()` **{#testrenderertojson}**

`testRenderer.toJSON()`

回傳一個被 render 的 tree object。該 tree 僅包含特定平台的 node, 例如 `<div>` 或 `<view>` 和它們的 props, 但並不包含任何人員撰寫的 component。這對於 snapshot 測試非常方便。

`testRenderer.toTree()` **{#testrenderertotree}**

`testRenderer.toTree()`

回傳一個被 render 的 tree object。它表示的內容比 `toJSON()` 提供的內容更加詳細, 並且包含人員撰寫的 component。除非你要在 test renderer 之上撰寫自己的 assertion library, 否則你可能不需要這個方法。

`testRenderer.update()` **{#testrendererupdate}**

`testRenderer.update(element)`

透過新的 root element 去重新 render 記憶體中的 tree。它模擬在 root 的 React 更新。如果新的 element 和之前的 element 有相同的 type 和 key, 該 tree 將會被更新；否則, 它將重新 mount 一個新的 tree。

`testRenderer.unmount()` **{#testrendererunmount}**

`testRenderer.unmount()`

Unmount 記憶體中的 tree, 並觸發相對應的生命週期事件。

`testRenderer.getInstance()` **{#testrenderergetinstance}**

`testRenderer.getInstance()`

如果可以的話，回傳與 root element 相對應的 instance。如果 root element 是 function component，該方法無效，因為 function component 沒有 instance。

`testRenderer.root {#testrendererroot}`

`testRenderer.root`

回傳 root 「測試 instance」 object，它對於 assert tree 中的特定 node 十分有用。你可以利用它來尋找其他更深層的「測試 instance」。

`testInstance.find() {#testinstancefind}`

`testInstance.find(test)`

找到一個 descendant 測試 instance，其 `test(testInstance)` 回傳 true。如果找到不只一個測試 instance，將會拋出錯誤。

`testInstance.findByType() {#testinstancefindbytype}`

`testInstance.findByType(type)`

找到一個與指定 `type` 匹配的 descendant 測試 instance。如果不只有一個測試 instance 匹配指定的 `type`，將會拋出錯誤。

`testInstance.findByProps() {#testinstancefindbyprops}`

`testInstance.findByProps(props)`

找到一個與指定 `props` 匹配的 descendant 測試 instance。如果不只有一個測試 instance 匹配指定的 `props`，將會拋出錯誤。

`testInstance.findAll() {#testinstancefindall}`

`testInstance.findAll(test)`

找到所有的 descendant 測試 instance，其 `test(testInstance)` 回傳 true。

`testInstance.findAllByType() {#testinstancefindallbytype}`

`testInstance.findAllByType(type)`

找到所有與指定 `type` 匹配的 descendant 測試 instance。

`testInstance.findAllByProps() {#testinstancefindallbyprops}`

`testInstance.findAllByProps(props)`

找到所有與指定 `props` 匹配的 descendant 測試 instance。

testInstance.instance {#testinstanceinstance}

testInstance.instance

該測試 instance 相對應的 component instance。它只能用於 class component，因為 function component 沒有 instance。它與 component 內部的 `this` 值匹配。

testInstance.type {#testinstancetype}

testInstance.type

該測試 instance 相對應的 component 類型。例如，一個 `<Button />` component 的類型為 `Button`。

testInstance.props {#testinstanceprops}

testInstance.props

該測試 instance 相對應的 component props。例如，一個 `<Button size="small" />` component 的 props 為 `{size: 'small'}`。

testInstance.parent {#testinstanceparent}

testInstance.parent

該測試 instance 的 parent 測試 instance。

testInstance.children {#testinstancechildren}

testInstance.children

該測試 instance 的 children 測試 instance。

概念

你可以把 `createNodeMock` function 當作第二個參數傳給 `TestRenderer.create`，做為自訂 mock 的 refs。`createNodeMock` 接受目前 element 作為參數，並且回傳一個 mock 的 ref object。這十分有利於依賴 refs component 的測試。

```
import TestRenderer from 'react-test-renderer';

class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.input = null;
  }
  componentDidMount() {
    this.input.focus();
  }
  render() {
    return <input type="text" ref={el => this.input = el} />
  }
}
```

```
let focused = false;
TestRenderer.create(
  <MyComponent />,
{
  createNodeMock: (element) => {
    if (element.type === 'input') {
      // mock a focus function
      return {
        focus: () => {
          focused = true;
        }
      };
    }
    return null;
  }
);
expect(focused).toBe(true);
```

Refs 和 DOM

Ref 提供了一種可以取得 DOM 節點或在 render 方法內建立 React element 的方式。

在典型的 React 資料流裡，`props` 是 parent component 和 child component 唯一的互動方式。你會藉由使用新的 prop 重新 render 來改變你的 child。然而，有些情況下你需要在典型的資料流以外更改你的 child。這個被更改的 child 可能是 React component 的其中一個 instance，或他可能是個 DOM element。在這兩種情況下，React 提供了「逃生口」。

什麼時候該使用 Ref

有幾種適合使用 ref 的情況：

- 管理 focus、選擇文字、或影音播放。
- 觸發即時的動畫。
- 與第三方 DOM 函式庫整合。

避免在任何可以宣告性完成事情的地方使用 ref。

例如，不要把 `Dialog` component 上的 `open()` 和 `close()` 對外公開，應該將 `isOpen` 的 prop 傳進去。

不要過度使用 Ref

你一開始可能會傾向於在應用程式裡使用 ref 「讓事情發生」。如果這是你的情形，花點時間認真思考一下 state 應該在哪個 component 的層級被持有。通常你會清楚發現，在高層級的地方持有是比較合適的位置。請參閱[提升 State](#)這篇指南裡面的範例。

注意

以下的範例已經被更新為使用 React 16.3 所引入的 `React.createRef()` API。如果你是利用比較舊版本的 React，我們推薦使用 [callback refs](#)。

建立 Ref

Ref 是藉由使用 `React.createRef()` 所產生的，它藉由 `ref` 參數被依附在 React element。Ref 常常會在一個 component 被建立出來的時候，被賦值在某個 instance 屬性，這樣一來他們就可以在整個 component 裡面被參考。

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.myRef = React.createRef();
  }
  render() {
    return <div ref={this.myRef} />;
  }
}
```

存取 Ref

當 ref 在 render 裡被傳到一個 element 的時候，一個指向節點對 ref 的 current 參數的參考會變得可以取得。

```
const node = this.myRef.current;
```

Ref 的值會根據節點的類型而有所不同：

- 當在 HTML element 上使用 ref 參數時，使用 React.createRef() 建立 ref 會取得它底下的 DOM element 來做為它的 current 屬性。
- 當在客製化的 class component 使用 ref 參數時，ref 取得被 mount 的 component 上的 instance 來當作他的 current。
- **你不能在 function component 上使用 ref**，因為他們沒有 instance。

下面的範例示範了差異。

在 DOM Element 加上 Ref

這段程式碼利用 ref 來儲存對於 DOM 節點的參考：

```
class CustomTextInput extends React.Component {
  constructor(props) {
    super(props);
    // 產生一個可以儲存 textInput DOM element 的 ref
    this.textInput = React.createRef();
    this.focusTextInput = this.focusTextInput.bind(this);
  }

  focusTextInput() {
    // 特別利用原生的 DOM API 來關注文字的輸入
    // 注意：我們正利用「current」來取得 DOM 節點
    this.textInput.current.focus();
  }

  render() {
    // 告訴 React 我們想要將 <input> ref
    // 和我們在 constructor 產生的 `textInput` 連結
    return (
      <div>
        <input
          type="text"
          ref={this.textInput} />
        <input
          type="button"
          value="Focus the text input"
          onClick={this.focusTextInput}
        />
      </div>
    );
  }
}
```

React 會在 component mount 的時候將 DOM element 賦值到 current 屬性，並在 unmount 時將它清空回 null。ref 的更新發生在生命週期 componentDidMount 或 componentDidUpdate 之前。

在 Class Component 加上 Ref

如果我們想要把上面的 `CustomTextInput` 包起來然後模擬它在被 `mount` 之後馬上被點擊，我們可以使用 `ref` 來獲得客製化的 `input` 併手動呼叫他的 `focusTextInput` 函式：

```
class AutoFocusTextInput extends React.Component {
  constructor(props) {
    super(props);
    this.textInput = React.createRef();
  }

  componentDidMount() {
    this.textInput.current.focusTextInput();
  }

  render() {
    return (
      <CustomTextInput ref={this.textInput} />
    );
  }
}
```

注意這種情況只適用於利用 class 來宣告 `CustomTextInput` 的情形：

```
class CustomTextInput extends React.Component {
  // ...
}
```

Ref 和 Function Component

預設上，你不能在 **function component** 上使用 `ref`，因為它們沒有 `instance`：

```
function MyFunctionComponent() {
  return <input />;
}

class Parent extends React.Component {
  constructor(props) {
    super(props);
    this.textInput = React.createRef();
  }

  render() {
    // This will *not* work!
    return (
      <MyFunctionComponent ref={this.textInput} />
    );
  }
}
```

如果你想要讓其他人將 `ref` 帶到你的 function component，你可以使用 `forwardRef` (可能與 `useImperativeHandle`)，或者你可以轉換成 class component。

然而，只要你想要指到的是 DOM element 或 class component，你也可以在 **function component** 裡使用 `ref`：

```
function CustomTextInput(props) {
  // 必須在這裡宣告 textInput 使 ref 可以參考到它
  const textInput = useRef(null);
```

```

function handleClick() {
 textInput.current.focus();
}

return (
  <div>
    <input
      type="text"
      ref={textInput} />
    <input
      type="button"
      value="Focus the text input"
      onClick={handleClick}
    />
  </div>
);
}

```

對 Parent Component 公開 DOM 的 Ref

在很少的情況下，你可能會想要在 parent component 取得 child 的 DOM 節點。在一般的情形下，並不建議這麼做，因為這會破壞 component 的封裝，但有時候這麼做對於觸發 focus 或測量 child 的 DOM 節點的大小、位置是很有用的。

雖然你可以在 child component 新增一個 ref，但這並不是理想的解法，因為你只能拿到一個 component 的 instance 而不是一個 DOM 節點。另外，在 function component 裡並不適用。

如果你使用比 React 16.3 還新的版本，在這種情形下，我們推薦使用傳送 ref。傳送 Ref 使得 component 能夠選擇要不要把 child component 的 ref 當作自己的 ref。你可以在傳送 ref 的文件裡找到詳細關於怎麼把 child 的 DOM 節點公開給 parent component 的範例。

如果你使用比 React 16.2 還舊的版本，或你需要比傳送 ref 的方式有更多的彈性，你可以使用這個不同的方式並把 ref 當作另個有名字的 prop 來傳進去。

可能的話，我們會建議不要把 DOM 節點公開，但這可能是個有用的逃生艙。要注意的是，這個方式需要你在 child component 新增一些程式碼。如果你對 child component 的實作完全無法控制，你的最終選擇是使用 findDOMNode()，但我們並不建議使用，且在 StrictMode 裡他會被淘汰掉。

Callback Refs

React 也支援另一種設定 ref 的方式，這種方法叫做「callback refs」，它提供了對 ref 的設定上更細緻的控制。

不是將 createRef() 所產生的 ref 傳遞下去，而是把一個 function 往下傳。這個 function 會將 React component 的 instance 或 HTML DOM 作為他的參數，然後可以被儲存之後在別的地方使用。

下面的例子實作了一個常見的模式：利用 ref 的 callback 來儲存一個在 instance 屬性裡 DOM 節點的參考。

```

class CustomTextInput extends React.Component {
  constructor(props) {
    super(props);

    this.textInput = null;

    this.setTextInputRef = element => {
      this.textInput = element;
    };
}

```

```

this.focusTextInput = () => {
  // 利用原生的 DOM API 來 focus 文字輸入
  if (this.textInput) this.textInput.focus();
};

componentDidMount() {
  // 在 mount 的時候自動 focus 輸入
  this.focusTextInput();
}

render() {
  // 利用 `ref` callback 來儲存 instance 欄位裡文字輸入 DOM 的參考
  // (例如: this.textInput)
  return (
    <div>
      <input
        type="text"
        ref={this.setTextInputRef}
      />
      <input
        type="button"
        value="Focus the text input"
        onClick={this.focusTextInput}
      />
    </div>
  );
}
}

```

React 會在 component 安裝時用 DOM element 呼叫 `ref` callback，然後在 unmount 時用 `null` 呼叫他。Ref 被保證在 `componentDidMount` 或 `componentDidUpdate` 觸發時能夠維持在最新的狀態。

你可以將 callback ref 在 component 之間傳遞，就像你可以用一樣的方式在 `React.createRef()` 所產生的 object ref 一樣。

```

function CustomTextInput(props) {
  return (
    <div>
      <input ref={props.inputRef} />
    </div>
  );
}

class Parent extends React.Component {
  render() {
    return (
      <CustomTextInput
        inputRef={el => this.inputElement = el}
      />
    );
  }
}

```

在上面的範例裡，`Parent` 將他的 ref callback 做為一個 `inputRef` prop 傳到 `CustomTextInput`，然後 `CustomTextInput` 將一樣的 function 當作特別的 `ref` 屬性來傳給 `<input>`。因此，在 `Parent` 的 `this.inputElement` 會被設為與在 `CustomTextInput` 裡 `<input>` element 相關的 DOM 節點。

Legacy API: String Refs

如果你以前就使用過 React，你可能對一個舊的 API 有點熟悉，當 `ref` 屬性是字串的時候，例如 `"textInput"`，然後 DOM 節點被當作 `this.refs.textInput` 來取得。我們不建議使用它，因為 string ref 有一些問題，所以他被視為 legacy，且很有可能會在未來的版本被移除。

注意

如果你正在使用 `this.refs.textInput` 來取得 ref，我們建議使用 [callback 的方式](#)或 [createRef API](#)。

對 callback ref 的警告

如果 `ref` callback 是被 inline function 所定義的，他會在更新的時候被呼叫兩次，第一次用 `null` 然後再用 DOM element 呼叫一次。這是因為新的 function 的 instance 是在每次 render 的時候被產生，所以 React 需要將舊的 ref 清掉然後設定新的。你可以藉由定義 `ref` callback 為 class 上的一個 bound method 來避免這種情形，但在大多情況下他並沒有任何影響。

Release Channels

React relies on a thriving open source community to file bug reports, open pull requests, and [submit RFCs](#). To encourage feedback we sometimes share special builds of React that include unreleased features.

This document will be most relevant to developers who work on frameworks, libraries, or developer tooling. Developers who use React primarily to build user-facing applications should not need to worry about our prerelease channels.

Each of React's release channels is designed for a distinct use case:

- **Latest** is for stable, semver React releases. It's what you get when you install React from npm. This is the channel you're already using today. **Use this for all user-facing React applications.**
- **Next** tracks the main branch of the React source code repository. Think of these as release candidates for the next minor semver release. Use this for integration testing between React and third party projects.
- **Experimental** includes experimental APIs and features that aren't available in the stable releases. These also track the main branch, but with additional feature flags turned on. Use this to try out upcoming features before they are released.

All releases are published to npm, but only Latest uses [semantic versioning](#). Prereleases (those in the Next and Experimental channels) have versions generated from a hash of their contents and the commit date, e.g. `0.0.0-68053d940-20210623` for Next and `0.0.0-experimental-68053d940-20210623` for Experimental.

The only officially supported release channel for user-facing applications is Latest. Next and Experimental releases are provided for testing purposes only, and we provide no guarantees that behavior won't change between releases. They do not follow the semver protocol that we use for releases from Latest.

By publishing prereleases to the same registry that we use for stable releases, we are able to take advantage of the many tools that support the npm workflow, like [unpkg](#) and [CodeSandbox](#).

Latest Channel

Latest is the channel used for stable React releases. It corresponds to the `latest` tag on npm. It is the recommended channel for all React apps that are shipped to real users.

If you're not sure which channel you should use, it's Latest. If you're a React developer, this is what you're already using.

You can expect updates to Latest to be extremely stable. Versions follow the semantic versioning scheme. Learn more about our commitment to stability and incremental migration in our [versioning policy](#).

Next Channel

The Next channel is a prerelease channel that tracks the main branch of the React repository. We use prereleases in the Next channel as release candidates for the Latest channel. You can think of Next as a superset of Latest that is updated more frequently.

The degree of change between the most recent Next release and the most recent Latest release is approximately the same as you would find between two minor semver releases. However, **the Next channel does not conform to semantic versioning**. You should expect occasional breaking changes between successive releases in the Next channel.

Do not use prereleases in user-facing applications.

Releases in Next are published with the `next` tag on npm. Versions are generated from a hash of the build's contents and the commit date, e.g. `0.0.0-68053d940-20210623`.

Using the Next Channel for Integration Testing

The Next channel is designed to support integration testing between React and other projects.

All changes to React go through extensive internal testing before they are released to the public. However, there are a myriad of environments and configurations used throughout the React ecosystem, and it's not possible for us to test against every single one.

If you're the author of a third party React framework, library, developer tool, or similar infrastructure-type project, you can help us keep React stable for your users and the entire React community by periodically running your test suite against the most recent changes. If you're interested, follow these steps:

- Set up a cron job using your preferred continuous integration platform. Cron jobs are supported by both [CircleCI](#) and [Travis CI](#).
- In the cron job, update your React packages to the most recent React release in the Next channel, using `next` tag on npm. Using the npm cli:

```
npm update react@next react-dom@next
```

Or yarn:

```
yarn upgrade react@next react-dom@next
```

- Run your test suite against the updated packages.
- If everything passes, great! You can expect that your project will work with the next minor React release.
- If something breaks unexpectedly, please let us know by [filing an issue](#).

A project that uses this workflow is Next.js. (No pun intended! Seriously!) You can refer to their [CircleCI configuration](#) as an example.

Experimental Channel

Like Next, the Experimental channel is a prerelease channel that tracks the main branch of the React repository. Unlike Next, Experimental releases include additional features and APIs that are not ready for wider release.

Usually, an update to Next is accompanied by a corresponding update to Experimental. They are based on the same source revision, but are built using a different set of feature flags.

Experimental releases may be significantly different than releases to Next and Latest. **Do not use Experimental releases in user-facing applications.** You should expect frequent breaking changes between releases in the Experimental channel.

Releases in Experimental are published with the `experimental` tag on npm. Versions are generated from a hash of the build's contents and the commit date, e.g. `0.0.0-experimental-68053d940-20210623`.

What Goes Into an Experimental Release?

Experimental features are ones that are not ready to be released to the wider public, and may change drastically before they are finalized. Some experiments may never be finalized -- the reason we have experiments is to test the viability of proposed changes.

For example, if the Experimental channel had existed when we announced Hooks, we would have released Hooks to the Experimental channel weeks before they were available in Latest.

You may find it valuable to run integration tests against Experimental. This is up to you. However, be advised that Experimental is even less stable than Next. **We do not guarantee any stability between Experimental releases.**

How Can I Learn More About Experimental Features?

Experimental features may or may not be documented. Usually, experiments aren't documented until they are close to shipping in Next or Latest.

If a feature is not documented, they may be accompanied by an [RFC](#).

We will post to the [React blog](#) when we're ready to announce new experiments, but that doesn't mean we will publicize every experiment.

You can always refer to our public GitHub repository's [history](#) for a comprehensive list of changes.

Render Props

「render prop」這個詞指的是一種用一個其值為函式的 prop 來在 React component 之間共享程式碼的技巧。

一個帶有 render prop 的 component 不會執行自己的 render 邏輯，而會接受一個回傳 React element 的函式並呼叫。

```
<DataProvider render={data => (
  <h1>Hello {data.target}</h1>
)}>
```

使用 render props 的函式庫包含 [React Router](#)、[Downshift](#) 和 [Formik](#)。

以下我們會討論為什麼 render props 如此有用，以及怎麼實作自己的 render props。

為橫切關注點使用 Render Props

Component 是 React 中主要的程式碼重用單位，但如何將一個 component 所包含的 state 或行為共享給其他也同樣需要這些狀態或行為的 component 並不是那麼直觀。

例如，下面這個 component 負責在一個網頁應用中追蹤滑鼠游標的位置：

```
class MouseTracker extends React.Component {
  constructor(props) {
    super(props);
    this.handleMouseMove = this.handleMouseMove.bind(this);
    this.state = { x: 0, y: 0 };
  }

  handleMouseMove(event) {
    this.setState({
      x: event.clientX,
      y: event.clientY
    });
  }

  render() {
    return (
      <div style={{ height: '100vh' }} onMouseMove={this.handleMouseMove}>
        <h1>Move the mouse around!</h1>
        <p>The current mouse position is ({this.state.x}, {this.state.y})</p>
      </div>
    );
  }
}
```

當游標在螢幕中移動時，這個 component 會在一個 `<p>` 中顯示它的 (x,y) 座標。

現在問題來了：我們該如何在其他 component 中重用這個行為呢？換句話說，如果另一個 component 需要知道游標的位置，我們能不能將這個行為封裝起來，讓其他的 component 能輕鬆地共享呢？

既然 component 是 React 中程式碼基本重用單位，我們來試試看重構這段程式碼，改成用一個 `<Mouse>` component 來封裝這些需要在其他地方重用的行為。

```
// <Mouse> component 封裝我們所需的行為...
class Mouse extends React.Component {
  constructor(props) {
    super(props);
    this.handleMouseMove = this.handleMouseMove.bind(this);
    this.state = { x: 0, y: 0 };
  }

  handleMouseMove(event) {
    this.setState({
      x: event.clientX,
      y: event.clientY
    });
  }

  render() {
    return (
      <div style={{ height: '100vh' }} onMouseMove={this.handleMouseMove}>
        {/* ...但我們如何 render 除了 <p> 以外的東西？ */}
        <p>The current mouse position is ({this.state.x}, {this.state.y})</p>
      </div>
    );
  }
}

class MouseTracker extends React.Component {
  render() {
    return (
      <>
        <h1>Move the mouse around!</h1>
        <Mouse />
      </>
    );
  }
}
```

現在 `<Mouse>` component 封裝了全部跟監聽 `mousemove` 事件有關的行為，也儲存了游標的 (x,y) 位置，但它還不算真的可重用。

舉例來說，假設我們有一個 `<Cat>` component，它會在螢幕中 render 追著滑鼠跑的貓咪圖片。我們可能會用一個 `<Cat mouse={{ x, y }}>` 的 prop 來告訴這個 component 滑鼠的座標，它便知道該把這張圖片放在螢幕中何處。

作為第一次嘗試，你可能會試著把 `<Cat>` 從 `<Mouse>` 的 `render` 方法裡面 render 出來，像這樣：

```
class Cat extends React.Component {
  render() {
    const mouse = this.props.mouse;
    return (
      
    );
  }
}

class MouseWithCat extends React.Component {
```

```

constructor(props) {
  super(props);
  this.handleMouseMove = this.handleMouseMove.bind(this);
  this.state = { x: 0, y: 0 };
}

handleMouseMove(event) {
  this.setState({
    x: event.clientX,
    y: event.clientY
  });
}

render() {
  return (
    <div style={{ height: '100vh' }} onMouseMove={this.handleMouseMove}>
      /*
        我們大可以在這裡把 <p> 換成 <Cat> ...但這樣我們就必須在每次用到它時，  

        創建另外一個 <MouseWithSomethingElse> component，  

        所以 <MouseWithCat> 的可重用性還不夠。
      */
      <Cat mouse={this.state} />
    </div>
  );
}

class MouseTracker extends React.Component {
  render() {
    return (
      <div>
        <h1>Move the mouse around!</h1>
        <MouseWithCat />
      </div>
    );
  }
}

```

此方法在特定案例中可行，但我們還未能以可重用的方法把這個行為封裝起來。現在，每次在不同案例中，如果我們需要知道滑鼠位置，便必須另外創建一個特定為該案例 render 某些東西的 component（換句話說，就是另一個 `<MouseWithCat>`）

這就是 render prop 發揮功用的地方了：我們提供 `<Mouse>` 一個 function prop render prop，讓它能夠動態決定該 render 什麼，而不是把 `<Cat>` 寫死在 `<Mouse>` component 裡。

```

class Cat extends React.Component {
  render() {
    const mouse = this.props.mouse;
    return (
      
    );
  }
}

class Mouse extends React.Component {
  constructor(props) {
    super(props);
    this.handleMouseMove = this.handleMouseMove.bind(this);
  }
}

```

```

    this.state = { x: 0, y: 0 };
}

handleMouseMove(event) {
  this.setState({
    x: event.clientX,
    y: event.clientY
  });
}

render() {
  return (
    <div style={{ height: '100vh' }} onMouseMove={this.handleMouseMove}>
      /* 用 `render` prop 去動態決定該 render 什麼，而不是將 <Mouse> render 的東西靜態表示出來。
      */
      {this.props.render(this.state)}
    </div>
  );
}

class MouseTracker extends React.Component {
  render() {
    return (
      <div>
        <h1>Move the mouse around!</h1>
        <Mouse render={mouse => (
          <Cat mouse={mouse} />
        )}/>
      </div>
    );
  }
}

```

現在，我們提供一個 `render prop` 給 `<Mouse>`，讓它能夠動態決定它該 `render` 什麼，而不是為了一些特定案例，去複製 `<Mouse>` component 並在它的 `render` 方法中硬性寫入某些東西。

更具體地說，**render prop 是一個讓 component 知道該 render 什麼的 function prop**。

這個技巧讓我們想共享的行為變得極為可轉移。要使用這個行為時，就 `render` 一個有 `render prop` 的 `<Mouse>`，讓 `render prop` 來告訴 `<Mouse>` 該用現在游標的 (x,y) `render` 什麼。

關於 `render props` 一件有趣的事是，你可以用包含 `render prop` 的普通 `component` 來實作 `higher-order component`。舉例來說，如果你偏好用 `withMouse` HOC 而不是 `<Mouse>` component 的話，你可以輕易地用一個普通的 `<Mouse>` 加上 `render prop` 來建立：

```

// 如果你真的想用 HOC，你可以輕易地用一個
// 有 render prop 的普通 component 來建立！
function withMouse(Component) {
  return class extends React.Component {
    render() {
      return (
        <Mouse render={mouse => (
          <Component {...this.props} mouse={mouse} />
        )}/>
      );
    }
  };
}

```

```

        }
    }
}
```

因此 render prop 讓兩種模式皆可使用。

使用 Props 代替 render {#using-props-other-than-render}

記住，只因為這個模式稱為「render props」，不代表你一定要用一個名為 `render` 的 prop 來使用這個模式。事實上，任何是函式且被 component 用來認知該 render 什麼的 prop，都叫做「redner prop」。

雖然上述範例使用 `render`，我們可以同樣輕易地使用 `children` prop！

```

<Mouse children={mouse => (
  <p>The mouse position is {mouse.x}, {mouse.y}</p>
)} />
```

記得，`children` prop 不需要在 JSX element 的屬性列表中註明。你可以直接把它放在 element 裡面！

```

<Mouse>
  {mouse => (
    <p>The mouse position is {mouse.x}, {mouse.y}</p>
  )}
</Mouse>
```

你會看到這個技巧在 `react-motion` API 中被使用。

因為這個技巧有一點不常見，在設計像這樣的 API 時，你可能會想在 `propTypes` 明確地把 `children` 定義為一個函式。

```

Mouse.propTypes = {
  children: PropTypes.func.isRequired
};
```

警告

注意當 Render Props 使用在 React.PureComponent 時

如果你在 `render` 方法中建立函式，使用 render prop 會讓 `React.PureComponent` 帶來的好處作廢。這是因為對新的 props 而言，prop 的淺比較會永遠回傳 `false`，並且每次 `render` 都會為 render prop 產生新的值。

例如，繼續我們上述的 `<Mouse>` component，如果 `Mouse` 繼承 `React.PureComponent` 而不是 `React.Component` 的話，我們的範例會像這樣：

```

class Mouse extends React.PureComponent {
  // 如上述實作內容...
}

class MouseTracker extends React.Component {
  render() {
    return (
      <div>
        <Mouse />
      </div>
    );
  }
}
```

```

<div>
  <h1>Move the mouse around!</h1>

  /* 
    這很不好！render prop 的值在每次 render 都會不一樣。
  */
  <Mouse render={mouse => (
    <Cat mouse={mouse} />
  )}>
  </div>
);

}
}

```

在這個範例中，每次 `<MouseTracker>` render 時，它會產生一個新的函式，作為 `<Mouse render>` prop 的值，便使一開始 `<Mouse>` 繼承 `React.PureComponent` 的效果作廢！

為了避開這個問題，你有時候可以把這個 prop 定義為一個 instance 方法，像是：

```

class MouseTracker extends React.Component {
  // `this.renderTheCat` 被定義成一個 instance 方法
  // 當我們在 render 使用到時，會永遠指向*相同的*函式
  renderTheCat(mouse) {
    return <Cat mouse={mouse} />;
  }

  render() {
    return (
      <div>
        <h1>Move the mouse around!</h1>
        <Mouse render={this.renderTheCat} />
      </div>
    );
  }
}

```

在不能把 prop 定義成靜態的案例中（例：因為你需要封閉 component 的 props 和/或 state），`<Mouse>` 應改為繼承 `React.Component`。

Render Element

建立 React 應用程式最小的單位是 element。

一個 element 描述你想要在螢幕上所看到的：

```
const element = <h1>Hello, world</h1>;
```

與瀏覽器的 DOM element 不同，React element 是單純的 object，而且很容易被建立。React DOM 負責更新 DOM 來符合 React element。

注意：

大家可能會將 element 與更廣為人知的「component」概念混淆。我們將會在[下一個章節](#)介紹 component。Component 是由 element 所「組成」的，我們建議你在開始之前閱讀本章節。

Render Element 到 DOM 內

假設你的 HTML 檔案內有一個 `<div>`：

```
<div id="root"></div>
```

我們稱為這是一個「root」DOM node，因為所有在內的 element 都會透過 React DOM 做管理。

使用 React 建立應用程式時，通常會有一個單一的 root DOM node。如果你想要整合 React 到現有的應用程式時，你可以根據你的需求獨立出多個 root DOM node。

如果要 render 一個 React element 到 root DOM node，傳入兩者到 `ReactDOM.createRoot()`，接著傳入 React element 到 `root.render()`：

[embed:rendering-elements/render-an-element.js](#)

[在 CodePen 上試試看吧！](#)

在網頁上你會看見顯示「Hello, world」。

更新被 Render 的 Element

React element 是 [immutable](#) 的。一旦你建立一個 element，你不能改變它的 children 或是 attribute。Element 就像是電影中的一個幀：它代表特定時間點的 UI。

憑藉我們迄今為止對 React 的認識，更新 UI 唯一的方式是建立一個新的 element，並且將它傳入到 `root.render`。

思考以下這個 ticking clock 的範例：

[embed:rendering-elements/update-rendered-element.js](#)

[在 CodePen 上試試看吧！](#)

它從 `setInterval()` callback 每秒呼叫 `root.render()`。

注意：

在實踐中，大部分 React 應用程式只呼叫 `root.render()` 一次。在下一個章節中，我們將會學習如何將這些程式碼封裝到 [stateful component](#)。

我們建議你不要跳著主題看，因為它們彼此間是環環相扣的。

React 只更新必要的 Element

React DOM 會將 element 和它的 children 與先前的狀態做比較，並且只更新必要的 DOM 達到理想的狀態。

你可以透過瀏覽器工具來檢測最後一個範例做驗證：



即使我們在每秒建立一個 element 描述整個 UI tree，只有內容更改的 text node 才會被 React DOM 更新。

根據我們的經驗，應該思考 UI 在任何時候應該如何呈現，而不是隨著時間的推移改變它，就可以去所有消除錯誤。

[Go to TOC](#)

State 和生命週期

這個章節會介紹在 React component 中 state 以及生命週期的概念。你可以在[這裡找到 component API 詳細的參考](#)。

思考[前一章節](#)的 ticking clock 的範例。在 [Rendering Elements](#) 中，我們只學習到一種方式來更新 UI。我們呼叫 `root.render()` 來改變 `render` 的輸出：

```
const root = ReactDOM.createRoot(document.getElementById('root'));

function tick() {
  const element = (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {new Date().toLocaleTimeString()}.</h2>
    </div>
  );
  root.render(element);
}

setInterval(tick, 1000);
```

[在 CodePen 上試試看吧！](#)

在這個章節中，我們將會學習如何封裝 `Clock` component 讓它可以真正的被重複使用。它將會設定本身的 timer 並且每秒更新一次。

我們可以像這樣封裝 `Clock` 做為開始：

```
const root = ReactDOM.createRoot(document.getElementById('root'));

function Clock(props) {
  return (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {props.date.toLocaleTimeString()}.</h2>
    </div>
  );
}

function tick() {
  root.render(<Clock date={new Date()} />);
}

setInterval(tick, 1000);
```

[在 CodePen 上試試看吧！](#)

然而，它缺少了一個重要的需求：`Clock` 設定 timer 並在每秒更新 UI 應該是 `Clock` 實作的細節的事實。

理想情況下，我們想要撰寫一次 `Clock` 並且它會自己更新：

```
root.render(<Clock />);
```

如果要實現這個理想情況，我們需要加入「state」到 `Clock` component。

State 類似於 prop，但它是私有且由 component 完全控制的。

我們在先前提到過，component 被定義為 class 有一些額外的特性。Local state 就是 class 其中的一個特性。

轉換 Function 成 Class

你可以透過以下 5 個步驟轉換一個 function component 像是 `Clock` 成為 class：

1. 建立一個相同名稱並且繼承 `React.Component` 的 ES6 class。
2. 加入一個 `render()` 的空方法。
3. 將 function 的內容搬到 `render()` 方法。
4. 將 `render()` 內的 `props` 替換成 `this.props`。
5. 刪除剩下空的 function 壓告。

```
class Clock extends React.Component {
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.props.date.toLocaleTimeString()}</h2>
      </div>
    );
  }
}
```

[在 CodePen 上試試看吧！](#)

`Clock` 現在被定義成 class 而不是 function。

在每次發生更新時，`render` 方法都會被呼叫，但我們只要 render `<Clock />` 到相同的 DOM node 中，只有 `Clock` class 這個實例會被用到。這讓我們可以使用像是 local state 和生命週期方法這些額外的特性。

加入 Local State 到 Class

我們會透過以下 3 個步驟將 `date` 從搬移到 `state`：

1. 將 `render()` 方法內的 `this.props.date` 替換成 `this.state.date`：

```
class Clock extends React.Component {
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}</h2>
      </div>
    );
  }
}
```

2. 加入一個 `class constructor` 並分配初始的 `this.state`：

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

注意，我們將傳送 `props` 到基礎 `constructor`：

```
constructor(props) {
  super(props);
  this.state = {date: new Date()};
}
```

Class component 應該總是要呼叫基礎 `constructor` 和 `props`。

3. 從 `<Clock />` element 中移除 `date` prop：

```
root.render(<Clock />);
```

之後我們將會把 timer 的程式碼加入到 component 本身。

結果看起來會像是：

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Clock />);
```

[在 CodePen 上試試看吧！](#)

接下來，我們會讓 `Clock` 設定它本身的 timer 並且每秒更新一次。

加入生命週期方法到 Class

在具有許多 component 的應用程式中，當 component 被 destroy 時，釋放所佔用的資源是非常重要的。

每當 `Clock` render 到 DOM 的時候，我們想要設定一個 timer。在 React 中稱為「mount」。

每當產生的 `Clock` DOM 被移除時，我們想要清除 timer。在 React 中稱為「unmount」。

每當 component 在 mount 或是 unmount 的時候，我們可以在 component class 上宣告一些特別的方法來執行一些程式碼：

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  componentDidMount() {}

  componentWillUnmount() {}

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

這些方法被稱為「生命週期方法」。

`componentDidMount()` 方法會在 component 被 render 到 DOM 之後才會執行。這是設定 timer 的好地方：

```
componentDidMount() {
  this.timerID = setInterval(
    () => this.tick(),
    1000
  );
}
```

注意我們是如何正確的在 `this (this.timerID)` 儲存 timer ID。

雖然 `this.props` 是由 React 本身設定的，而且 `this.state` 具有特殊的意義，如果你需要儲存一些不相關於資料流的內容（像是 timer ID），你可以自由的手動加入。

我們將會在 `componentWillUnmount()` 生命週期方法內移除 timer：

```
componentWillUnmount() {
  clearInterval(this.timerID);
}
```

最後，我們將會實作一個 `tick()` 的方法，`Clock` component 將會在每秒執行它。

它將會使用 `this.setState()` 來安排 component local state 的更新：

```
class Clock extends React.Component {
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }

  componentDidMount() {
    this.timerID = setInterval(
      () => this.tick(),
      1000
    );
  }

  componentWillUnmount() {
    clearInterval(this.timerID);
  }

  tick() {
    this.setState({
      date: new Date()
    });
  }

  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.state.date.toLocaleTimeString()}</h2>
      </div>
    );
  }
}

const root = ReactDOM.createRoot(document.getElementById('root'));
root.render(<Clock />);
```

[在 CodePen 上試試看吧！](#)

現在我們的 `Clock` 每秒鐘都會滴答作響。

讓我們快速的回顧一下發生了哪些事情，以及呼叫這些方法的順序：

- 當 `<Clock />` 被傳入到 `root.render()` 時，React 會呼叫 `Clock` component 的 `constructor`。由於 `Clock` 需要顯示目前的時間，它使用包含目前時間的 object 初始化 `this.state`。我們會在之後更新這個 `state`。
- React 接著呼叫 `Clock` component 的 `render()` 方法。這就是 React 如何了解應該要在螢幕上顯示什麼內容。React 接著更新 DOM 來符合 `Clock` 的 `render` 輸出。
- 每當 `Clock` 輸出被插入到 DOM 時，React 會呼叫 `componentDidMount()` 生命週期方法。在 `Clock` component 生命週期方法內，會要求瀏覽器設定 timer 每秒去呼叫 component 的 `tick()` 方法。

4. 瀏覽器每秒呼叫 `tick()` 方法。其中，`Clock` component 透過包含目前時間的 object 呼叫 `setState()` 來調度 UI 更新。感謝 `setState()`，React 現在知道 state 有所改變，並且再一次呼叫 `render()` 方法來了解哪些內容該呈現在螢幕上。這時候，在 `render()` 方法內的 `this.state.date` 將會有所不同，因此 `render` 輸出將會是更新的時間。React 相應地更新 DOM。
5. 如果 `Clock` component 從 DOM 被移除了，React 會呼叫 `componentWillUnmount()` 生命週期方法，所以 timer 會被停止。

正確的使用 State

有三件關於 `setState()` 的事情你應該要知道。

請不要直接修改 State

例如，這將不會重新 render component：

```
// 錯誤
this.state.comment = 'Hello';
```

相反的，使用 `setState()`：

```
// 正確
this.setState({comment: 'Hello'});
```

你唯一可以指定 `this.state` 值的地方是在 constructor。

State 的更新可能是非同步的

React 可以將多個 `setState()` 呼叫批次處理為單一的更新，以提高效能。

因為 `this.props` 和 `this.state` 可能是非同步的被更新，你不應該依賴它們的值來計算新的 state。

例如，這個程式碼可能無法更新 counter：

```
// 錯誤
this.setState({
  counter: this.state.counter + this.props.increment,
});
```

要修正這個問題，使用第二種形式的 `setState()`，它接受一個 function 而不是一個 object。Function 將接收先前的 state 作為第一個參數，並且將更新的 props 作為第二個參數：

```
// 正確
this.setState((state, props) => ({
  counter: state.counter + props.increment
}));
```

在上面我們使用 arrow function，但它也可以適用於正常的 function：

```
// 正確
this.setState(function(state, props) {
  return {
```

```

        counter: state.counter + props.increment
    };
});

```

State 的更新將會被 Merge

當你呼叫 `setState()` 時，React 會 merge 你提供的 object 到目前的 state。

例如，你的 state 可能包含幾個單獨的變數：

```

constructor(props) {
  super(props);
  this.state = {
    posts: [],
    comments: []
  };
}

```

然後你可以單獨的呼叫 `setState()` 更新它們：

```

componentDidMount() {
  fetchPosts().then(response => {
    this.setState({
      posts: response.posts
    });
  });

  fetchComments().then(response => {
    this.setState({
      comments: response.comments
    });
  });
}

```

這個 merge 是 shallow 的，所以 `this.setState({comments})` 保持 `this.state.posts` 的完整，但它完全取代了 `this.state.comments`。

向下資料流

Parent 和 child component 不會知道某個 component 是 stateful 或 stateless 的 component，而且它們不在意它是透過 function 或是 class 被定義的。

這就是 state 通常被稱為 local state 或被封裝的原因。除了擁有和可以設定它之外的任何 component 都不能訪問它。

Component 可以選擇將它的 state 做為 props 往下傳遞到它的 child component：

```
<FormattedDate date={this.state.date} />
```

`FormattedDate` component 會在它的 props 接收到 `date`，但他不知道它是從 `Clock` 的 state 傳遞過來的，從 `Clock` 的 props 或者是透過手動輸入：

```

function FormattedDate(props) {
  return <h2>It is {props.date.toLocaleTimeString()}</h2>;
}

```

在 CodePen 上試試看吧！

這通常被稱作為「上至下」或「單向」的資料流。任何 state 總是由某個特定的 component 所擁有，任何從 state 得到的資料或 UI, state 只能影響在 tree 「以下」的 component。

如果你想像一個 component tree 是一個 props 的瀑布，每個 component 的 state 像是一個額外的水流源頭，它在任意的某個地方而且往下流。

為了表示所有 component 真的都是被獨立的，我們可以建立一個 App component 來 render 三個 <Clock>：

```
function App() {
  return (
    <div>
      <Clock />
      <Clock />
      <Clock />
    </div>
  );
}
```

在 CodePen 上試試看吧！

每個 Clock 設定它本身的 timer 並獨立的更新。

在 React 應用程式中，不論 component 是 stateful 或 stateless 都被視為是實作 component 的細節，它可能隨著時間而改變。你可以在 stateful component 內使用 stateless component，反之亦然。

[Go to TOC](#)

Static Type Checking

Static type checkers like [Flow](#) and [TypeScript](#) identify certain types of problems before you even run your code. They can also improve developer workflow by adding features like auto-completion. For this reason, we recommend using Flow or TypeScript instead of `PropTypes` for larger code bases.

Flow

[Flow](#) is a static type checker for your JavaScript code. It is developed at Facebook and is often used with React. It lets you annotate the variables, functions, and React components with a special type syntax, and catch mistakes early. You can read an [introduction to Flow](#) to learn its basics.

To use Flow, you need to:

- Add Flow to your project as a dependency.
- Ensure that Flow syntax is stripped from the compiled code.
- Add type annotations and run Flow to check them.

We will explain these steps below in detail.

Adding Flow to a Project

First, navigate to your project directory in the terminal. You will need to run the following command:

If you use [Yarn](#), run:

```
yarn add --dev flow-bin
```

If you use [npm](#), run:

```
npm install --save-dev flow-bin
```

This command installs the latest version of Flow into your project.

Now, add `flow` to the `"scripts"` section of your `package.json` to be able to use this from the terminal:

```
{
  // ...
  "scripts": {
    "flow": "flow",
    // ...
  },
  // ...
}
```

Finally, run one of the following commands:

If you use [Yarn](#), run:

```
yarn run flow init
```

If you use `npm`, run:

```
npm run flow init
```

This command will create a Flow configuration file that you will need to commit.

Stripping Flow Syntax from the Compiled Code

Flow extends the JavaScript language with a special syntax for type annotations. However, browsers aren't aware of this syntax, so we need to make sure it doesn't end up in the compiled JavaScript bundle that is sent to the browser.

The exact way to do this depends on the tools you use to compile JavaScript.

Create React App

If your project was set up using [Create React App](#), congratulations! The Flow annotations are already being stripped by default so you don't need to do anything else in this step.

Babel

Note:

These instructions are *not* for Create React App users. Even though Create React App uses Babel under the hood, it is already configured to understand Flow. Only follow this step if you *don't* use Create React App.

If you manually configured Babel for your project, you will need to install a special preset for Flow.

If you use Yarn, run:

```
yarn add --dev @babel/preset-flow
```

If you use npm, run:

```
npm install --save-dev @babel/preset-flow
```

Then add the `flow` preset to your [Babel configuration](#). For example, if you configure Babel through `.babelrc` file, it could look like this:

```
{
  "presets": [
    "@babel/preset-flow",
    "react"
  ]
}
```

This will let you use the Flow syntax in your code.

Note:

Flow does not require the `react` preset, but they are often used together. Flow itself understands JSX syntax out of the box.

Other Build Setups

If you don't use either Create React App or Babel, you can use [flow-remove-types](#) to strip the type annotations.

Running Flow

If you followed the instructions above, you should be able to run Flow for the first time.

```
yarn flow
```

If you use npm, run:

```
npm run flow
```

You should see a message like:

```
No errors!
✨ Done in 0.17s.
```

Adding Flow Type Annotations

By default, Flow only checks the files that include this annotation:

```
// @flow
```

Typically it is placed at the top of a file. Try adding it to some files in your project and run `yarn flow` or `npm run flow` to see if Flow already found any issues.

There is also [an option](#) to force Flow to check *all* files regardless of the annotation. This can be too noisy for existing projects, but is reasonable for a new project if you want to fully type it with Flow.

Now you're all set! We recommend to check out the following resources to learn more about Flow:

- [Flow Documentation: Type Annotations](#)
- [Flow Documentation: Editors](#)
- [Flow Documentation: React](#)
- [Linting in Flow](#)

TypeScript

TypeScript is a programming language developed by Microsoft. It is a typed superset of JavaScript, and includes its own compiler. Being a typed language, TypeScript can catch errors and bugs at build time, long before your app goes live. You can learn more about using TypeScript with React [here](#).

To use TypeScript, you need to:

- Add TypeScript as a dependency to your project
- Configure the TypeScript compiler options
- Use the right file extensions
- Add definitions for libraries you use

Let's go over these in detail.

Using TypeScript with Create React App

Create React App supports TypeScript out of the box.

To create a **new project** with TypeScript support, run:

```
npx create-react-app my-app --template typescript
```

You can also add it to an **existing Create React App project**, as documented [here](#).

Note:

If you use Create React App, you can **skip the rest of this page**. It describes the manual setup which doesn't apply to Create React App users.

Adding TypeScript to a Project

It all begins with running one command in your terminal.

If you use [Yarn](#), run:

```
yarn add --dev typescript
```

If you use [npm](#), run:

```
npm install --save-dev typescript
```

Congrats! You've installed the latest version of TypeScript into your project. Installing TypeScript gives us access to the `tsc` command. Before configuration, let's add `tsc` to the "scripts" section in our `package.json`:

```
{
  // ...
  "scripts": {
    "build": "tsc",
    // ...
  },
  // ...
}
```

Configuring the TypeScript Compiler

The compiler is of no help to us until we tell it what to do. In TypeScript, these rules are defined in a special file called `tsconfig.json`. To generate this file:

If you use [Yarn](#), run:

```
yarn run tsc --init
```

If you use [npm](#), run:

```
npx tsc --init
```

Looking at the now generated `tsconfig.json`, you can see that there are many options you can use to configure the compiler. For a detailed description of all the options, check [here](#).

Of the many options, we'll look at `rootDir` and `outDir`. In its true fashion, the compiler will take in typescript files and generate javascript files. However we don't want to get confused with our source files and the generated output.

We'll address this in two steps:

- Firstly, let's arrange our project structure like this. We'll place all our source code in the `src` directory.

```

└── package.json
└── src
  └── index.ts
└── tsconfig.json
```

- Next, we'll tell the compiler where our source code is and where the output should go.

```
// tsconfig.json

{
  "compilerOptions": {
    // ...
    "rootDir": "src",
    "outDir": "build"
    // ...
  },
}
```

Great! Now when we run our build script the compiler will output the generated javascript to the `build` folder. The [TypeScript React Starter](#) provides a `tsconfig.json` with a good set of rules to get you started.

Generally, you don't want to keep the generated javascript in your source control, so be sure to add the build folder to your `.gitignore`.

File extensions

In React, you most likely write your components in a `.js` file. In TypeScript we have 2 file extensions:

`.ts` is the default file extension while `.tsx` is a special extension used for files which contain `JSX`.

Running TypeScript

If you followed the instructions above, you should be able to run TypeScript for the first time.

```
yarn build
```

If you use npm, run:

```
npm run build
```

If you see no output, it means that it completed successfully.

Type Definitions

To be able to show errors and hints from other packages, the compiler relies on declaration files. A declaration file provides all the type information about a library. This enables us to use javascript libraries like those on npm in our project.

There are two main ways to get declarations for a library:

Bundled - The library bundles its own declaration file. This is great for us, since all we need to do is install the library, and we can use it right away. To check if a library has bundled types, look for an `index.d.ts` file in the project. Some libraries will have it specified in their `package.json` under the `typings` or `types` field.

DefinitelyTyped - DefinitelyTyped is a huge repository of declarations for libraries that don't bundle a declaration file. The declarations are crowd-sourced and managed by Microsoft and open source contributors. React for example doesn't bundle its own declaration file. Instead we can get it from DefinitelyTyped. To do so enter this command in your terminal.

```
# yarn
yarn add --dev @types/react

# npm
npm i --save-dev @types/react
```

Local Declarations Sometimes the package that you want to use doesn't bundle declarations nor is it available on DefinitelyTyped. In that case, we can have a local declaration file. To do this, create a `declarations.d.ts` file in the root of your source directory. A simple declaration could look like this:

```
declare module 'querystring' {
  export function stringify(val: object): string
  export function parse(val: string): object
}
```

You are now ready to code! We recommend to check out the following resources to learn more about TypeScript:

- [TypeScript Documentation: Everyday Types](#)
- [TypeScript Documentation: Migrating from JavaScript](#)
- [TypeScript Documentation: React and Webpack](#)

ReScript

[ReScript](#) is a typed language that compiles to JavaScript. Some of its core features are guaranteed 100% type coverage, first-class JSX support and [dedicated React bindings](#) to allow integration in existing JS / TS React codebases.

You can find more infos on integrating ReScript in your existing JS / React codebase [here](#).

Kotlin

[Kotlin](#) is a statically typed language developed by JetBrains. Its target platforms include the JVM, Android, LLVM, and [JavaScript](#).

JetBrains develops and maintains several tools specifically for the React community: [React bindings](#) as well as [Create React Kotlin App](#). The latter helps you start building React apps with Kotlin with no build configuration.

Other Languages

Note there are other statically typed languages that compile to JavaScript and are thus React compatible. For example, [F#/Fable](#) with [elmish-react](#). Check out their respective sites for more information, and feel free to add more statically typed languages that work with React to this page!

嚴格模式

`嚴格模式` 是一個用來突顯應用程式裡潛在問題的工具。如同 `Fragment` 一樣，`嚴格模式` 不會 render 任何可見的 UI。它為了底下的所有的依賴們啟動了額外的檢查和警告。

注意：

嚴格模式檢查只會在開發模式中執行；它們不應該影響正式環境。

你可以在應用程式的任何地方打開嚴格模式。例如：`embed:strict-mode/enabling-strict-mode.js`

在上面的範例裡，嚴格模式檢查將不會跑在 `Header` 和 `Footer` 元件上。然而 `ComponentOne` 和 `ComponentTwo`，以及它們底下的所有子依賴，都會被檢查。

`StrictMode` 目前可以幫助：

- 發現擁有不安全生命週期的 component
- 警告使用了 legacy string ref API
- 警告使用到了被棄用的 findDOMNode
- 偵測意想不到的副作用
- 偵測 legacy context API
- 確保可重用的 state

其他功能會在未來版本的 React 釋出時被加進去。

發現不安全的生命週期

如這篇文章所述，某些遺留的生命週期在非同步的 React 應用程式裡使用是不安全的。然而，如果你的應用程式使用到第三方套件，確認這些生命週期有沒有被使用到是很困難的。幸好嚴格模式可以在這點幫助我們！

當嚴格模式被打開的時候，React 編譯了一整串用到這些不安全生命週期的 class component，然後記錄了這些 component 的警告訊息，例如：

```
▶ Warning: Unsafe lifecycle methods were found within a strict-mode tree:  
  in div (created by ExampleApplication)  
    in ExampleApplication  
  
componentWillMount: Please update the following components to use componentDidMount instead: ThirdPartyComponent  
  
Learn more about this warning here:  
https://fb.me/react-strict-mode-warnings
```

現在就關注這些被嚴格模式所發現的問題，會幫助你在 React 將來的版本裡處理非同步 render 的時候更容易。

警告使用到了 legacy string ref API

以前 React 提供了兩種管理 ref 的方式：legacy string ref API 和 callback API。雖然 string ref API 在兩者之間是比較方便的，但它有很多缺點，所以我們的官方推薦是使用 callback form.

React 16.3 加上了第三種選擇，提供了 string ref 的便利性且免除了那些缺點：[embed:16-3-release-blog-post/create-ref-example.js](#)

因為 object ref 已經大量取代了 string ref，嚴格模式現在會在你使用 string ref 的時候警告你。

注意：

除了新的 `createRef` API 以外，callback ref 會持續被支援。

你不需要改掉你 component 裡的 callback ref。它們稍微更有彈性，所以它們會持續是一個進階性的功能。

學習更多關於新的 `createRef API`。

警告使用到了被棄用的 `findDOMNode`

React 過去支援了 `findDOMNode` 來用 class instance 搜尋 tree 裡面的 DOM 節點。通常你不需要這個，因為你可以直接把一個 ref 附到你的 DOM 節點。

`findDOMNode` 也可以被使用在 class component 上，但這是一個破壞抽象層的用法，它允許了 parent 來要求 render 某個特定的 children。它產生了重構的風險，因為 parent 可能會進入到某個 DOM 節點，所以你不能隨意改變 component 的實作細節。`findDOMNode` 只會回傳第一個 child，但如果使用了 Fragment，有可能某個 component 會 render 多個 DOM 節點。`findDOMNode` 是一個只能讀一次的 API。它只在你要求的時候告訴你答案。如果一個 child component render 了不同的節點，沒有任何方法可以處理這樣的改變。所以 `findDOMNode` 只在元件永遠回傳一個單一且永遠不改變的 DOM 節點時有用。

你可以藉由傳遞 ref 到你的客製化 component，且把它傳到使用 `ref forwarding` 的 DOM，使它變得明顯。

你也可以在你的 component 加上一個包裹的 DOM 節點，然後把 ref 直接附在它上面。

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.wrapper = React.createRef();
  }
  render() {
    return <div ref={this.wrapper}>{this.props.children}</div>;
  }
}
```

注意：

在 CSS 裡，如果你不想要某個節點成為 layout 的一部份，`display: contents` attribute 可以被使用。

偵測意想不到的副作用

概念上，React 在兩種面相上能夠運作：

- **render** 面相決定了必須做出什麼改變到例如 DOM 的地方。在這個面相上，React 呼叫 `render` 然後比較了它與上一次 `render` 的結果。
- **commit** 面相是每當 React 運用到任何改變的時候。（在 React DOM 的例子，當 React 插入、更新、或移除 DOM 節點。）React 也在這個面相上呼叫了像是 `componentDidMount` 和 `componentDidUpdate` 的生命週期。

Commit 面相通常非常快，但 `render` 可能會很慢。為了這個原因，將來的非同步模式（還沒被預設成開啟）會把 `render` 的工作切成小塊，暫停和恢復這些工作藉以避免阻擋瀏覽器。這表示 React 可能會在 `commit` 之前調用多次 `render` 面相的生命週期，或是他會不管有沒有 `commit` 就調用它們（因為錯誤或是更高優先性的中斷）。

Render 面相的生命週期包含了以下 class component 函式：

- `constructor`
- `componentWillMount` (或 `UNSAFE_componentWillMount`)
- `componentWillReceiveProps` (或 `UNSAFE_componentWillReceiveProps`)
- `componentWillUpdate` (或 `UNSAFE_componentWillUpdate`)
- `getDerivedStateFromProps`
- `shouldComponentUpdate`
- `render`
- `setState` updater functions (第一個參數)

因為以上函式可能會被呼叫不只一次，不包含副作用是很重要的。忽視這個規則可能會導致各種問題，包含記憶體滲漏，和不合法的應用程式 state。不幸的是，偵測這些問題很困難，因為它們通常是 [non-deterministic](#)。

嚴格模式無法自動檢測 side effect，但它可以藉由使這些 side effect 變得更有確定性，來幫助你發現它們。它藉由故意調用兩次下面的函式來完成這個功能：

- Class component `constructor`、`render` 和 `shouldComponentUpdate` 方法
- Class component 的靜態 `getDerivedStateFromProps` 方法
- Function component 的內容
- 狀態更新函式 (`setState` 的第一個參數)
- 函數傳遞至 `useState`、`useMemo` 或 `useReducer`

注意：

這個只在開發模式發生。生命週期不會被重複調用在正式環境。

例如，考慮以下程式碼：[embed:strict-mode/side-effects-in-constructor.js](#)

第一眼看這段程式碼，可能不會覺得它有問題。但如果 `SharedApplicationState.recordEvent` 不是 [idempotent](#) 的話，多次實體化這個 component 可能會導致不合法的應用程式 state。這種細微的錯誤可能在開發期間不會出現，或者會不一致地出現，因此被忽略了。

通過有意地雙重調用如 component constructor，嚴格模式使這種模式更容易被發現。

注意：

從 React 17 開始，React 會自動更改如 `console.log()` 的 `console` 方法，以在第二次調用生命週期函數時使 `log` 靜音，在某些情況下，它可能會導致無法預期的行為，你[可以使用替代方案](#)。

從 React 18 開始，React 不會抑制任何 `log`。然而，如果你有安裝 React DevTools，則第二次呼叫時會顯得有些黯淡。React DevTools 也提供了一個設定（預設關閉）來完全抑制它們。

檢測 legacy context API

Legacy context API 是容易出錯的，並將在以後的主要版本中刪除。它仍然適用於所有 16.x 版本，但將在嚴格模式下顯示以下警告訊息：

```
✖ ►Warning: Legacy context API has been detected within a strict-mode tree:  
  in div (at App.js:32)  
  in App (at index.js:7)  
  
Please update the following components: LegacyContextConsumer, LegacyContextProvider  
  
Learn more about this warning here:  
https://fb.me/react-strict-mode-warnings
```

閱讀[新的 context API 文件](#)，以助於遷移到新版本。

Ensuring reusable state

In the future, we'd like to add a feature that allows React to add and remove sections of the UI while preserving state. For example, when a user tabs away from a screen and back, React should be able to immediately show the previous screen. To do this, React will support remounting trees using the same component state used before unmounting.

This feature will give React better performance out-of-the-box, but requires components to be resilient to effects being mounted and destroyed multiple times. Most effects will work without any changes, but some effects do not properly clean up subscriptions in the destroy callback, or implicitly assume they are only mounted or destroyed once.

To help surface these issues, React 18 introduces a new development-only check to Strict Mode. This new check will automatically unmount and remount every component, whenever a component mounts for the first time, restoring the previous state on the second mount.

To demonstrate the development behavior you'll see in Strict Mode with this feature, consider what happens when React mounts a new component. Without this change, when a component mounts, React creates the effects:

- * React mounts the component.
- * Layout effects are created.
- * Effects are created.

With Strict Mode starting in React 18, whenever a component mounts in development, React will simulate immediately unmounting and remounting the component:

- * React mounts the component.
 - * Layout effects are created.
 - * Effect effects are created.
- * React simulates effects being destroyed on a mounted component.
 - * Layout effects are destroyed.
 - * Effects are destroyed.
- * React simulates effects being re-created on a mounted component.
 - * Layout effects are created
 - * Effect setup code runs

On the second mount, React will restore the state from the first mount. This feature simulates user behavior such as a user tabbing away from a screen and back, ensuring that code will properly handle state restoration.

When the component unmounts, effects are destroyed as normal:

- * React unmounts the component.
 - * Layout effects are destroyed.
 - * Effect effects are destroyed.

Unmounting and remounting includes:

- `componentDidMount`
- `componentWillUnmount`
- `useEffect`
- `useLayoutEffect`
- `useInsertionEffect`

Note:

This only applies to development mode, *production behavior is unchanged*.

For help supporting common issues, see:

- [How to support Reusable State in Effects](#)

[Go to TOC](#)

測試環境

本份文件介紹了可能影響環境的因素以及某些場景下的建議。

Test runner

像是 [Jest](#)、[mocha](#)、[ava](#) 的 Test runner 讓你以常規的 JavaScript 來撰寫測試，並在開發過程中執行它們。此外，test suite 也會作為持續整合中執行的一部份。

- Jest 廣泛的兼容 React 專案，支援像是 `module` 和 `timer`，以及 `jsdom` 的 mock 功能。**如果你使用 Create React App，內建已經包含了 Jest 與有用的預設設定。**
- 像是 [mocha](#) 之類的函式庫，在實際的瀏覽器環境中可以很好地工作，並且可以幫助明確需要它的測試在實際的瀏覽器環境中可以很好的運作，並且可以幫助明確需要它的測試。
- End-to-end 測試被用在較長需要橫跨多個頁面的流程，並且需要一個[不同的設定](#)。

Mock 一個 render 的 surface

測試通常不會在存取像是瀏覽器真實的 surface 的環境中執行。對於這些環境，我們建議使用 `jsdom` 模擬一個瀏覽器，它是一個可以在 Node.js 中執行的輕量瀏覽器實作。

在大多數情況下，`jsdom` 的行為像是一個正常的瀏覽器，但是沒有像是 `layout` 和 `navigation` 的功能。這對於大部分基於 web component 的測試仍然有用，因為它執行的速度比每次測試都啟動瀏覽器還要來得快。它也在相同的 process 執行你的測試，因此你可以撰寫程式來檢查和 assert 被 render 的 DOM。

就像在一個真實的瀏覽器一樣，`jsdom` 讓我們模擬使用者的互動；測試可以在 DOM node 上 dispatch event，並且觀察和 assert 這些操作（範例）的 side effect。

上述的設定可以撰寫大部分的 UI 測試：使用 Jest 作為 test runner，render 到 `jsdom`，透過 `act()` helper（範例）將使用者互動指定為瀏覽器 event 的序列。例如，React 本身許多的測試都是由此組合所撰寫的。

如果你正在撰寫一個函式庫，它主要測試於特定的瀏覽器行為，並要求原生的瀏覽器行為像是 `layout` 或是真實的輸入，你可以使用像是 [mocha](#) 的測試框架。

在無法模擬 DOM 的環境中（例如：在 Node.js 上測試 React Native 的 component），你可以使用 `event simulation helper` 來模擬互動的 element。或者，你可以使用 `@testing-library/react-native` 的 `fireEvent` helper。

像是 [Cypress](#)、[puppeteer](#) 和 [webdriver](#) 的框架對於執行 end-to-end tests 非常有用。

Mock 函式

當撰寫測試時，我們會想要 mock 在我們測試環境中沒有等效的程式碼部分（例如：在 Node.js 內判斷 `navigator.onLine` 狀態）。測試也可以監視（spy）某些函式，並觀察其他部分的測試如何與它交互。可以使用易於測試的版本，對於選擇性地來 mock 這些函式很有用。

這特別是對於資料的 `fetch` 非常有用。通常最好是使用「假」資料來測試，避免從真實 API endpoint 取得資料，造成緩慢和脆弱的測試（範例）。這增進了測試的可預期性。像是 [Jest](#) 和 [sinon](#) 的函式庫，都支援 mock 函式。對於 end-to-end 測試來說，mock network 可能會更困難，但你仍然可能也想要在測試中測試真實的 API endpoint。

Mocking module

有些 component 與其他 module 可能有依賴關係，在測試環境中可能會無法運作，或者是對於測試不可缺少的。選擇性的 mock 這些 module 並進行適當的替換會很有用（[範例](#)）。

在 Node.js 中，像是 Jest 的 runner [支援 mock module](#)。你也可以選擇像是 `mock-require` 的函式庫。

Mocking timer

Component 可能使用基於像是 `setTimeout`、`setInterval` 或是 `Date.now` 的時間函式。在測試環境中，可以透過 mock 這些函式來做替換，對於你想要手動「提前」時間非常有用。這可以確保你的測試很快地執行！依賴於 timer 的測試仍然可以按順序 `resolve`，但速度更快（[範例](#)）。大部分的測試框架，包含 [Jest](#)、[sinon](#) 以及 [lolex](#) 讓你可以在測試中 mock timer。

有時候你可能不想要 mock timer。例如，或許你想要測試一個 animation，或是與一個對時間較敏感的 endpoint 交互（像是 API 的 rate limiter）。函式庫的 timer mock 讓你在每個測試的基礎上啟用和關閉它們，所以你可以明確的選擇這些測試的執行方式。

End-to-end 測試

End-to-end 測試對於測試更長的 workflow 非常有用，特別是當它們對於你的商業邏輯至關重要時（像是付款或是註冊）。對於這些測試，你可能需要測試真實瀏覽器是如何 render 整個應用程式，從真實 API endpoint 來 fetch 資料，使用 session 和 cookie，在不同的連結之間導航。你可能不只想要 assert 在 DOM 上的 state，而且還要備份資料的 assert（例如：驗證更新是否已經被儲存到資料庫）。

在這個場景下，你會使用像是 [Cypress](#)、[Playwright](#)，或是像是 [Puppeteer](#) 的函式庫，所以你可以在不同的 route 之間切換，並 assert side effect，而不是只有在瀏覽器內，也有可能在後端。

Testing Recipes

Common testing patterns for React components.

Note:

This page assumes you're using [Jest](#) as a test runner. If you use a different test runner, you may need to adjust the API, but the overall shape of the solution will likely be the same. Read more details on setting up a testing environment on the [Testing Environments](#) page.

On this page, we will primarily use function components. However, these testing strategies don't depend on implementation details, and work just as well for class components too.

- [Setup/Tear down](#)
- [act\(\)](#)
- [Rendering](#)
- [Data Fetching](#)
- [Mocking Modules](#)
- [Events](#)
- [Timers](#)
- [Snapshot Testing](#)
- [Multiple Renderers](#)
- [Something Missing?](#)

Setup/Tear down

For each test, we usually want to render our React tree to a DOM element that's attached to `document`. This is important so that it can receive DOM events. When the test ends, we want to "clean up" and unmount the tree from the `document`.

A common way to do it is to use a pair of `beforeEach` and `afterEach` blocks so that they'll always run and isolate the effects of a test to itself:

```
import { unmountComponentAtNode } from "react-dom";

let container = null;
beforeEach(() => {
  // setup a DOM element as a render target
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // cleanup on exiting
  unmountComponentAtNode(container);
```

```
    container.remove();
    container = null;
});
```

You may use a different pattern, but keep in mind that we want to execute the cleanup *even if a test fails*. Otherwise, tests can become "leaky", and one test can change the behavior of another test. That makes them difficult to debug.

act() {#act}

When writing UI tests, tasks like rendering, user events, or data fetching can be considered as "units" of interaction with a user interface. `react-dom/test-utils` provides a helper called `act()` that makes sure all updates related to these "units" have been processed and applied to the DOM before you make any assertions:

```
act(() => {
  // render components
});
// make assertions
```

This helps make your tests run closer to what real users would experience when using your application. The rest of these examples use `act()` to make these guarantees.

You might find using `act()` directly a bit too verbose. To avoid some of the boilerplate, you could use a library like [React Testing Library](#), whose helpers are wrapped with `act()`.

Note:

The name `act` comes from the [Arrange-Act-Assert](#) pattern.

Rendering

Commonly, you might want to test whether a component renders correctly for given props. Consider a simple component that renders a message based on a prop:

```
// hello.js

import React from "react";

export default function Hello(props) {
  if (props.name) {
    return <h1>Hello, {props.name}!</h1>;
  } else {
    return <span>Hey, stranger</span>;
  }
}
```

We can write a test for this component:

```
// hello.test.js

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";

import Hello from "./hello";

let container = null;
beforeEach(() => {
  // setup a DOM element as a render target
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // cleanup on exiting
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});

it("renders with or without a name", () => {
  act(() => {
    render(<Hello />, container);
  });
  expect(container.textContent).toBe("Hey, stranger");

  act(() => {
    render(<Hello name="Jenny" />, container);
  });
  expect(container.textContent).toBe("Hello, Jenny!");

  act(() => {
    render(<Hello name="Margaret" />, container);
  });
  expect(container.textContent).toBe("Hello, Margaret!");
});
```

Data Fetching

Instead of calling real APIs in all your tests, you can mock requests with dummy data. Mocking data fetching with "fake" data prevents flaky tests due to an unavailable backend, and makes them run faster. Note: you may still want to run a subset of tests using an ["end-to-end"](#) framework that tells whether the whole app is working together.

```
// user.js

import React, { useState, useEffect } from "react";

export default function User(props) {
  const [user, setUser] = useState(null);

  async function fetchUserData(id) {
    const response = await fetch(`/${id}`);
    setUser(await response.json());
  }

  useEffect(() => {
    fetchUserData(props.id);
  });
}
```

```

}, [props.id]);

if (!user) {
  return "loading...";
}

return (
  <details>
    <summary>{user.name}</summary>
    <strong>{user.age}</strong> years old
    <br />
    lives in {user.address}
  </details>
);
}

```

We can write tests for it:

```

// user.test.js

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";
import User from "./user";

let container = null;
beforeEach(() => {
  // setup a DOM element as a render target
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // cleanup on exiting
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});

it("renders user data", async () => {
  const fakeUser = {
    name: "Joni Baez",
    age: "32",
    address: "123, Charming Avenue"
  };

  jest.spyOn(global, "fetch").mockImplementation(() =>
    Promise.resolve({
      json: () => Promise.resolve(fakeUser)
    })
  );

  // Use the asynchronous version of act to apply resolved promises
  await act(async () => {
    render(<User id="123" />, container);
  });

  expect(container.querySelector("summary").textContent).toBe(fakeUser.name);
  expect(container.querySelector("strong").textContent).toBe(fakeUser.age);
  expect(container.textContent).toContain(fakeUser.address);
})

```

```
// remove the mock to ensure tests are completely isolated
global.fetch.mockRestore();
});
```

Mocking Modules

Some modules might not work well inside a testing environment, or may not be as essential to the test itself. Mocking out these modules with dummy replacements can make it easier to write tests for your own code.

Consider a `Contact` component that embeds a third-party `GoogleMap` component:

```
// map.js

import React from "react";

import { LoadScript, GoogleMap } from "react-google-maps";
export default function Map(props) {
  return (
    <LoadScript id="script-loader" googleMapsApiKey="YOUR_API_KEY">
      <GoogleMap id="example-map" center={props.center} />
    </LoadScript>
  );
}

// contact.js

import React from "react";
import Map from "./map";

export default function Contact(props) {
  return (
    <div>
      <address>
        Contact {props.name} via{" "}
        <a data-testid="email" href={"mailto:" + props.email}>
          email
        </a>
        or on their <a data-testid="site" href={props.site}>
          website
        </a>.
      </address>
      <Map center={props.center} />
    </div>
  );
}
```

If we don't want to load this component in our tests, we can mock out the dependency itself to a dummy component, and run our tests:

```
// contact.test.js

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";

import Contact from "./contact";
import MockedMap from "./map";
```

```

jest.mock("./map", () => {
  return function DummyMap(props) {
    return (
      <div data-testid="map">
        {props.center.lat}:{props.center.long}
      </div>
    );
  };
});

let container = null;
beforeEach(() => {
  // setup a DOM element as a render target
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // cleanup on exiting
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});

it("should render contact information", () => {
  const center = { lat: 0, long: 0 };
  act(() => {
    render(
      <Contact
        name="Joni Baez"
        email="test@example.com"
        site="http://test.com"
        center={center}
      />,
      container
    );
  });
  expect(
    container.querySelector("[data-testid='email']").getAttribute("href")
  ).toEqual("mailto:test@example.com");

  expect(
    container.querySelector('[data-testid="site"]').getAttribute("href")
  ).toEqual("http://test.com");

  expect(container.querySelector('[data-testid="map"]').textContent).toEqual(
    "0:0"
  );
});
});

```

Events

We recommend dispatching real DOM events on DOM elements, and then asserting on the result. Consider a `Toggle` component:

```

// toggle.js

import React, { useState } from "react";

```

```

export default function Toggle(props) {
  const [state, setState] = useState(false);
  return (
    <button
      onClick={() => {
        setState(previousState => !previousState);
        props.onChange(!state);
      }}
      data-testid="toggle"
    >
      {state === true ? "Turn off" : "Turn on"}
    </button>
  );
}

```

We could write tests for it:

```

// toggle.test.js

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";

import Toggle from "./toggle";

let container = null;
beforeEach(() => {
  // setup a DOM element as a render target
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // cleanup on exiting
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});

it("changes value when clicked", () => {
  const onChange = jest.fn();
  act(() => {
    render(<Toggle onChange={onChange} />, container);
  });

  // get a hold of the button element, and trigger some clicks on it
  const button = document.querySelector("[data-testid=toggle]");
  expect(button.innerHTML).toBe("Turn on");

  act(() => {
    button.dispatchEvent(new MouseEvent("click", { bubbles: true }));
  });

  expect(onChange).toHaveBeenCalledTimes(1);
  expect(button.innerHTML).toBe("Turn off");

  act(() => {
    for (let i = 0; i < 5; i++) {
      button.dispatchEvent(new MouseEvent("click", { bubbles: true }));
    }
  });
})

```

```
    expect(onChange).toHaveBeenCalledTimes(6);
    expect(button.innerHTML).toBe("Turn on");
});
```

Different DOM events and their properties are described in [MDN](#). Note that you need to pass `{ bubbles: true }` in each event you create for it to reach the React listener because React automatically delegates events to the root.

Note:

React Testing Library offers a [more concise helper](#) for firing events.

Timers

Your code might use timer-based functions like `setTimeout` to schedule more work in the future. In this example, a multiple choice panel waits for a selection and advances, timing out if a selection isn't made in 5 seconds:

```
// card.js

import React, { useEffect } from "react";

export default function Card(props) {
  useEffect(() => {
    const timeoutID = setTimeout(() => {
      props.onSelect(null);
    }, 5000);
    return () => {
      clearTimeout(timeoutID);
    };
  }, [props.onSelect]);

  return [1, 2, 3, 4].map(choice => (
    <button
      key={choice}
      data-testid={choice}
      onClick={() => props.onSelect(choice)}
    >
      {choice}
    </button>
  ));
}
```

We can write tests for this component by leveraging [Jest's timer mocks](#), and testing the different states it can be in.

```
// card.test.js

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";

import Card from "./card";
```

```

let container = null;
beforeEach(() => {
  // setup a DOM element as a render target
  container = document.createElement("div");
  document.body.appendChild(container);
  jest.useFakeTimers();
});

afterEach(() => {
  // cleanup on exiting
  unmountComponentAtNode(container);
  container.remove();
  container = null;
  jest.useRealTimers();
});

it("should select null after timing out", () => {
  const onSelect = jest.fn();
  act(() => {
    render(<Card onSelect={onSelect} />, container);
  });

  // move ahead in time by 100ms
  act(() => {
    jest.advanceTimersByTime(100);
  });
  expect(onSelect).not.toHaveBeenCalled();

  // and then move ahead by 5 seconds
  act(() => {
    jest.advanceTimersByTime(5000);
  });
  expect(onSelect).toHaveBeenCalledWith(null);
});

it("should cleanup on being removed", () => {
  const onSelect = jest.fn();
  act(() => {
    render(<Card onSelect={onSelect} />, container);
  });

  act(() => {
    jest.advanceTimersByTime(100);
  });
  expect(onSelect).not.toHaveBeenCalled();

  // unmount the app
  act(() => {
    render(null, container);
  });

  act(() => {
    jest.advanceTimersByTime(5000);
  });
  expect(onSelect).not.toHaveBeenCalled();
});

it("should accept selections", () => {
  const onSelect = jest.fn();
  act(() => {
    render(<Card onSelect={onSelect} />, container);
  });
});

```

```

act(() => {
  container
    .querySelector("[data-testid='2']")
    .dispatchEvent(new MouseEvent("click", { bubbles: true }));
});

expect(onSelect).toHaveBeenCalledWith(2);
});

```

You can use fake timers only in some tests. Above, we enabled them by calling `jest.useFakeTimers()`. The main advantage they provide is that your test doesn't actually have to wait five seconds to execute, and you also didn't need to make the component code more convoluted just for testing.

Snapshot Testing

Frameworks like Jest also let you save "snapshots" of data with `toMatchSnapshot` / `toMatchInlineSnapshot`. With these, we can "save" the rendered component output and ensure that a change to it has to be explicitly committed as a change to the snapshot.

In this example, we render a component and format the rendered HTML with the `pretty` package, before saving it as an inline snapshot:

```

// hello.test.js, again

import React from "react";
import { render, unmountComponentAtNode } from "react-dom";
import { act } from "react-dom/test-utils";
import pretty from "pretty";

import Hello from "./hello";

let container = null;
beforeEach(() => {
  // setup a DOM element as a render target
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  // cleanup on exiting
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});

it("should render a greeting", () => {
  act(() => {
    render(<Hello />, container);
  });

  expect(
    pretty(container.innerHTML))
    .toMatchInlineSnapshot(); /* ... gets filled automatically by jest ... */

  act(() => {
    render(<Hello name="Jenny" />, container);
  });
}

```

```

expect(
  pretty(container.innerHTML)
).toMatchSnapshot(); /* ... gets filled automatically by jest ... */

act(() => {
  render(<Hello name="Margaret" />, container);
});

expect(
  pretty(container.innerHTML)
).toMatchSnapshot(); /* ... gets filled automatically by jest ... */
);

```

It's typically better to make more specific assertions than to use snapshots. These kinds of tests include implementation details so they break easily, and teams can get desensitized to snapshot breakages. Selectively [mocking some child components](#) can help reduce the size of snapshots and keep them readable for the code review.

Multiple Renderers

In rare cases, you may be running a test on a component that uses multiple renderers. For example, you may be running snapshot tests on a component with `react-test-renderer`, that internally uses `render` from `react-dom` inside a child component to render some content. In this scenario, you can wrap updates with `act()`'s corresponding to their renderers.

```

import { act as domAct } from "react-dom/test-utils";
import { act as testAct, create } from "react-test-renderer";
// ...
let root;
domAct(() => {
  testAct(() => {
    root = create(<App />);
  });
});
expect(root).toMatchSnapshot();

```

Something Missing?

If some common scenario is not covered, please let us know on the [issue tracker](#) for the documentation website.

測試概覽

- "community/testing.html"

你可以像測試其他 JavaScript 程式碼一樣來測試 React component。

有幾種方式可以來測試 React component。大致上來說，它們可以分為兩種類型：

- 在簡化的測試環境中 **render component tree** 並且 assert 它們的輸出。
- 在真實瀏覽器環境中執行一個完整的應用程式（也稱為「end-to-end」測試）。

本文件章節專注於第一種情況的測試策略。而完整的 end-to-end 測試對於預防重要的 workflow 不受影響非常有用，但此類測試不涉於 React component，因此不在本文件範圍內。

權衡

當選擇測試工具時，值得考慮一些權衡：

- **迭代速度 vs 實際環境**：有一些工具在做出更改和查看結果之間提供了非常快速的回饋，但沒有精確的模擬瀏覽器的行為。其他工具可能使用真實的瀏覽器環境，但降低了迭代速度而且在持續集成伺服器上較脆弱。
- **需要多少 Mock**：對於 component 的「單元」測試和「整合」測試之間的區別可能很模糊。如果你要測試一個表單，應該要測試表單內的按鈕嗎？或是 button component 應該有它自己的測試？重構按鈕會破壞表單的測試嗎？

對於不同的團隊和產品會有不同的答案。

推薦工具

Jest 是一個 JavaScript test runner 讓你可以透過 `jsdom` 存取 DOM。`jsdom` 只是一個很接近瀏覽器工作的方式，不過它已經足以測試 React component。Jest 有出色的迭代速度與像是 mock `module` 和 `timer` 的強大功能，因此你可以有更多的方法來控制你的程式碼執行。

React Testing Library 是一系列的 helper 讓你可以測試 React component 而不需要依賴它們的實作細節。這個方法讓重構變的輕而易舉，也讓你朝著可訪問性的最佳實踐的方向而前進。雖然它沒有提供一種「shallow」render 它的 child component 的方式，透過像是 Jest test runner 的 `mock` 可以讓你達到這個目的。

學習更多

本章節分成以下兩頁：

- **方法**：為 React component 撰寫測試時的常用模式。
- **環境**：為 React component 設定測試環境時應該考慮的事項。

用 React 思考

- 'blog/2013/11/05/thinking-in-react.html'
- 'docs/thinking-in-react-zh-CN.html'

在我們的意見中，React 是用 JavaScript 建立大型、快速的網路應用程式最首要的方式。它對於在 Facebook 和 Instagram 的我們來說能很有效的增加規模。

React 許多的優點之一是它讓你能在寫程式的同時去思考你的應用程式。在這個章節中，我們會帶領你走過一遍用 React 來建立一個可搜尋的產品資料表格的思考過程。

從視覺稿開始

想像一下我們已經有個 JSON API 和一個設計師給我們的產品視覺稿。這個視覺稿看起來像這樣：

<input type="text" value="Search..."/>
<input type="checkbox"/> Only show products in stock
Name Price
Sporting Goods
Football \$49.99
Baseball \$9.99
Basketball \$29.99
Electronics
iPod Touch \$99.99
iPhone 5 \$399.99
Nexus 7 \$199.99

視覺稿

我們的 JSON API 則會回傳一些看起來像這樣的資料：

```
[ ];
```

第一步：將 UI 拆解成 component 層級

首先，你要做的是將視覺稿中每一個 component（及 subcomponent）都圈起來，並幫它們命名。如果你在跟設計師合作的話，他們可能已經幫你做好這一步了，所以跟他們聊聊吧！他們在 Photoshop 中所用的圖層的名字可能可以作為你的 React component 的名字！

但是你要怎麼知道哪一個東西應該是自己獨立一個 component 呢？使用和你決定建立一個新的 function 或 object 一樣的準則即可。其中一個技巧是單一職責原則，它的意思是：在我們的理想中，一個 component 應該只負責做一件事情。如果這個 component 最後變大了，你就需要再將它分成數個更小的 subcomponent。

由於你常常會展示 JSON 的資料模型給使用者，你會發現，如果你的模式是正確地被建立的話，你的 UI（以及你的 component 結構）會很好的相互對應。這是因為 UI 和資料模型通常是遵守同樣的資訊架構，這意味著將你的 UI 拆成 component 通常是相當容易的。將 UI 分解成數個 component，每一個都明確代表著你的資料模型中的某一部份即可。



圖解巢狀的 component

你會看到在這裡我們應用程式中有 5 個 component。我們將每個 component 代表的資料用斜體表示。圖片中的數字對應到下面的數字。

1. `FilterableProductTable` (橘色)：包含整個範例
2. `SearchBar` (藍色)：接收所有使用者的輸入
3. `ProductTable` (綠色)：展示並過濾根據使用者輸入的資料集
4. `ProductCategoryRow` (土耳其藍色)：為每個列別展示標題
5. `ProductRow` (紅色)：為每個產品展示一列

如果你看看 `ProductTable`，你會發現表格的標題列（內含「Name」和「Price」標籤）並非獨立的 component。要不要把它們變成 component 這個議題完全是個人的喜好，正反意見都有。在這邊的例子裡面，我們把它當作 `ProductTable` 的一部分，因為它是 rendering 資料集的一部分，而這正是 `ProductTable` 這個 component 的責任。然而，如果標題欄之後變得越來越複雜（假如我們要加上可以分類的直觀功能的話），那麼建立一個獨立的 `ProductTableHeader` component 就非常合理。

既然我們已經找出視覺稿中的 component 了，讓我們來安排它們的層級。在視覺稿中，在另一個 component 中出現的 component 就應該是 child：

- `FilterableProductTable`
 - `SearchBar`
 - `ProductTable`
 - `ProductCategoryRow`
 - `ProductRow`

為你的應用程式建立一個 render 資料模型的版本，你會想要建立可以重複使用其他 component 的 component，並使用 *props* 傳遞資料。*Props* 是將資料從 parent 傳給 child 的方式。如果你對於 *state* 的概念很熟悉的話，請完全不要使用 **state** 來建立這個靜態版本。*State* 是保留給互動性的，也就是會隨時間改變的資料。既然我們目前要做的是這應用程式的靜態版本，你就不需要 *state*。

你可以從最上層開始，或從最下層開始。也就是說，你可以先從層級較高的 component 開始做起（也就是從 `FilterableProductTable` 開始），或者你也可以從比它低層級的（`ProductRow`）開始。在比較簡單的例子中，通常從上往下是比較簡單的。但在較為大型的專案中，從下往上、邊寫邊測試則比較容易。

在這一步的最後，你會有一個函式庫的可重複使用的 component 來 render 你的資料模型。這些 component 只會有 `render()` 方法，因為這是應用程式的靜態版本。最高層級的 component (`FilterableProductTable`) 會接收你的資料模型作為 prop。如果你改變底層的資料模型並再次呼叫 `root.render()` 的話，那麼 UI 就會被更新。你可以看到 UI 的更新方式以及更改的位置。React 的 **單向資料流**（也可稱為**單向綁定**）確保所有 component 都是模塊化且快速的。

如果你需要幫助來執行這一步的話，請參考這份 [React 文件](#)。

簡短的插曲：Props 和 State

React 中有兩種「模型」資料：`props` and `state`。理解兩者的差別至關重要。若你不確定兩者的差別，請瀏覽 [React 的官方文件](#) 以及 [FAQ : props 和 state 有何不同？](#)。

第三步：找出最少(但完整)的 UI State 的代表

為了將你的 UI 變成有互動性，你需要有辦法觸發底層的資料模型做出改變。React 使用 **state** 把這件事實現了。

為了正確地建立你的應用程式，你首先需要思考你的應用程式最少需要哪些可變的 *state*。這裡的關鍵是 [DRY：避免重複代碼原則](#)。請找出你的應用程式所需的最少的呈現方式，並在你遇到其他東西時再計算它們。例如，如果你在建立一個待辦清單，使用一個可以用來代表待辦事項的 array。不要另外用一個獨立的 *state* 變數來追蹤數量。當你要 render 代辦事項的數量時，讀取待辦事項 array 的長度即可。

思考我們範例中應用程式的所有資料。我們現在有：

- 原本的產品列表
- 使用者輸入的搜尋關鍵字
- checkbox 的值
- 篩選過後的產品列表

讓我們來看一下每一個資料，並找出哪一個是 *state*。對於每一個資料，問你自己這三個問題：

1. 這個資料是從 parent 透過 *props* 傳下來的嗎？如果是的話，那它很可能不是 *state*。
2. 這個資料是否一直保持不變呢？如果是的話，那它很可能不是 *state*。
3. 你是否可以根據你的 component 中其他的 *state* 或 *prop* 來計算這個資料呢？如果是的話，那它一定不是 *state*。

原本的產品列表是被當作 *prop* 往下傳的，所以它不是 *state*。搜尋關鍵字和 checkbox 看起來可能是 *state*，因為它們會隨時而改變，也不能從其他東西中被計算出來。最後，篩選過後的產品列表不是 *state*，因為它能透過結合原本的產品列表、搜尋關鍵字和checkbox 的值被計算出來。

所以，我們的 *state* 是：

- 使用者輸入的搜尋關鍵字
- checkbox 的值

第四步：找出你的 State 應該在哪裡

```

HTML      CSS      Babel      Result      EDIT ON
class ProductCategoryRow extends React.Component {
  render() {
    const category = this.props.category;
    return (
      <tr>
        <th colSpan="2">
          {category}
        </th>
      </tr>
    );
  }
}

class ProductRow extends React.Component {
  render() {
    const product = this.props.product;
    const name = product.stocked ?
      product.name :
      <span style={{color: 'red'}}>
        {product.name}
      </span>;
    return (
      <tr>
        <td>{name}</td>
        <td>{product.price}</td>
      </tr>
    );
  }
}

```

Resources View Compiled

OK，所以我們已經找出這個應用程式最少的 state 是哪些了。下一步，我們需要找出哪幾個 component 會 mutate，或者擁有，這個 state。

請記得，React 的核心精神是單向資料流，從 component 的層級從高往下流。也許哪個 component 該擁有 state 在一開始並不是很明顯。**對新手來說，這往往是最難理解的概念**，所以請跟著以下的步驟來思考：

在你的應用程式中的每個 state：

- 指出每個根據 state 來 render 某些東西的 component。
- 找出一個共同擁有者 component（在層級中單一一個需要 state 的、在所有的 component 之上的 component）。
- 應該擁有 state 的會是共同擁有者 component 或另一個更高層級的 component。
- 如果你找不出一個應該擁有 state 的 component 的話，那就建立一個新的 component 來保持 state，並把它加到層級中共同擁有者 component 之上的某處。

讓我們來討論一下我們應用程式的這個策略：

- `ProductTable` 需要根據 state 來篩選產品列表，而 `SearchBar` 需要展示搜尋關鍵字和 checkbox 的 state。
- 這兩個 component 的共同擁有者 component 是 `FilterableProductTable`。

如果你嘗試在上一個版本的範例（第四步）中輸入或勾選 checkbox，你會看到 React 忽略你的輸入。這是刻意的，因為我們把 `input` 的 `value` prop 設定為永遠和從 `FilterableProductTable` 傳下來的 `state` 一樣。

讓我們思考一下我們想要做些什麼。我們想確保當使用者改變這個表格時，我們會更新 `state` 以反映使用者的輸入。既然 component 只應該更新它自己本身的 `state`，`FilterableProductTable` 將會把 callback 傳給 `SearchBar`，而它們則會在 `state` 該被更新的時候被觸發。我們可以在輸入上使用 `onChange` 這個 event 來接收通知。被 `FilterableProductTable` 傳下來的 callback 則會呼叫 `setState()`，之後應用程式就會被更新。

完成

希望這幫助你理解如何用 React 建立 component 和應用程式。雖然這可能需要你比你習慣的多打一些程式碼，請記得閱讀程式碼比起寫程式碼更常發生，而閱讀這種模組化、清晰明確的程式碼是非常容易的。當你開始建立大型的 component 函式庫時，你會很感激有這樣的明確性和模組性，而當你開始重複使用程式碼時，你的程式的行數會開始減少。:)

Typechecking With PropTypes

Note:

`React.PropTypes` has moved into a different package since React v15.5. Please use [the prop-type library instead](#).

We provide [a codemod script](#) to automate the conversion.

As your app grows, you can catch a lot of bugs with typechecking. For some applications, you can use JavaScript extensions like [Flow](#) or [TypeScript](#) to typecheck your whole application. But even if you don't use those, React has some built-in typechecking abilities. To run typechecking on the props for a component, you can assign the special `propTypes` property:

```
import PropTypes from 'prop-types';

class Greeting extends React.Component {
  render() {
    return (
      <h1>Hello, {this.props.name}</h1>
    );
  }
}

Greeting.propTypes = {
  name: PropTypes.string
};
```

In this example, we are using a class component, but the same functionality could also be applied to function components, or components created by `React.memo` or `React.forwardRef`.

`PropTypes` exports a range of validators that can be used to make sure the data you receive is valid. In this example, we're using `PropTypes.string`. When an invalid value is provided for a prop, a warning will be shown in the JavaScript console. For performance reasons, `propTypes` is only checked in development mode.

PropTypes

Here is an example documenting the different validators provided:

```
import PropTypes from 'prop-types';

MyComponent.propTypes = {
  // You can declare that a prop is a specific JS type. By default, these
  // are all optional.
  optionalArray: PropTypes.array,
  optionalBool: PropTypes.bool,
  optionalFunc: PropTypes.func,
  optionalNumber: PropTypes.number,
```

```

optionalObject: PropTypes.object,
optionalString: PropTypes.string,
optionalSymbol: PropTypes.symbol,

// Anything that can be rendered: numbers, strings, elements or an array
// (or fragment) containing these types.
optionalNode: PropTypes.node,

// A React element.
optionalElement: PropTypes.element,

// A React element type (ie. MyComponent).
optionalElementType: PropTypes.elementType,

// You can also declare that a prop is an instance of a class. This uses
// JS's instanceof operator.
optionalMessage: PropTypes.instanceOf(Message),

// You can ensure that your prop is limited to specific values by treating
// it as an enum.
optionalEnum: PropTypes.oneOf(['News', 'Photos']),

// An object that could be one of many types
optionalUnion: PropTypes.oneOfType([
  PropTypes.string,
  PropTypes.number,
  PropTypes.instanceOf(Message)
]),

// An array of a certain type
optionalArrayOf: PropTypes.arrayOf(PropTypes.number),

// An object with property values of a certain type
optionalObjectOf: PropTypes.objectOf(PropTypes.number),

// An object taking on a particular shape
optionalObjectWithShape: PropTypes.shape({
  color: PropTypes.string,
  fontSize: PropTypes.number
}),

// An object with warnings on extra properties
optionalObjectWithStrictShape: PropTypes.exact({
  name: PropTypes.string,
  quantity: PropTypes.number
}),

// You can chain any of the above with `isRequired` to make sure a warning
// is shown if the prop isn't provided.
requiredFunc: PropTypes.func.isRequired,

// A required value of any data type
requiredAny: PropTypes.any.isRequired,

// You can also specify a custom validator. It should return an Error
// object if the validation fails. Don't `console.warn` or throw, as this
// won't work inside `oneOfType`.
customProp: function(props, propName, componentName) {
  if (!/matchme/.test(props[propName])) {
    return new Error(
      'Invalid prop `' + propName + '` supplied to' +
      ' `' + componentName + '`. Validation failed.'
    )
}

```

```

        );
    },

// You can also supply a custom validator to `arrayOf` and `objectOf`.
// It should return an Error object if the validation fails. The validator
// will be called for each key in the array or object. The first two
// arguments of the validator are the array or object itself, and the
// current item's key.
customArrayProp: PropTypes.arrayOf(function(propValue, key, componentName,
location, propFullName) {
  if (!/matchme/.test(propValue[key])) {
    return new Error(
      'Invalid prop `' + propFullName + '` supplied to' +
      ' `' + componentName + '`. Validation failed.'
    );
  }
});

```

Requiring Single Child

With `PropTypes.element` you can specify that only a single child can be passed to a component as children.

```

import PropTypes from 'prop-types';

class MyComponent extends React.Component {
  render() {
    // This must be exactly one element or it will warn.
    const children = this.props.children;
    return (
      <div>
        {children}
      </div>
    );
  }
}

MyComponent.propTypes = {
  children: PropTypes.element.isRequired
};

```

Default Prop Values

You can define default values for your `props` by assigning to the special `defaultProps` property:

```

class Greeting extends React.Component {
  render() {
    return (
      <h1>Hello, {this.props.name}</h1>
    );
  }
}

// Specifies the default values for props:
Greeting.defaultProps = {
  name: 'Stranger'
};

```

```
// Renders "Hello, Stranger":
const root = ReactDOM.createRoot(document.getElementById('example'));
root.render(<Greeting />);
```

Since ES2022 you can also declare `defaultProps` as static property within a React component class. For more information, see the [class public static fields](#). This modern syntax will require a compilation step to work within older browsers.

```
class Greeting extends React.Component {
  static defaultProps = {
    name: 'stranger'
  }

  render() {
    return (
      <div>Hello, {this.props.name}</div>
    )
  }
}
```

The `defaultProps` will be used to ensure that `this.props.name` will have a value if it was not specified by the parent component. The `propTypes` typechecking happens after `defaultProps` are resolved, so typechecking will also apply to the `defaultProps`.

Function Components

If you are using function components in your regular development, you may want to make some small changes to allow PropTypes to be properly applied.

Let's say you have a component like this:

```
export default function HelloWorldComponent({ name }) {
  return (
    <div>Hello, {name}</div>
  )
}
```

To add PropTypes, you may want to declare the component in a separate function before exporting, like this:

```
function HelloWorldComponent({ name }) {
  return (
    <div>Hello, {name}</div>
  )
}

export default HelloWorldComponent
```

Then, you can add PropTypes directly to the `HelloWorldComponent`:

```
import PropTypes from 'prop-types'

function HelloWorldComponent({ name }) {
  return (
    <div>Hello, {name}</div>
```

```
)  
}  
  
HelloWorldComponent.propTypes = {  
  name: PropTypes.string  
}  
  
export default HelloWorldComponent
```

[Go to TOC](#)

Uncontrolled Component

在大多數的情況下，我們推薦使用 [controlled component](#) 來實作表單。在控制元件裡，表單的資料是被 React component 所處理。另一個選擇是 uncontrolled component，表單的資料是由 DOM 本身所處理的。

如果要寫一個 uncontrolled component，你可以[使用 ref](#) 來從 DOM 取得表單的資料，而不是為了每個 state 的更新寫 event handler。

舉例來說，這段程式碼在 uncontrolled component 裡接受一個名字：

```
class NameForm extends React.Component {
  constructor(props) {
    super(props);
    this.handleSubmit = this.handleSubmit.bind(this);
    this.input = React.createRef();
  }

  handleSubmit(event) {
    alert('A name was submitted: ' + this.input.current.value);
    event.preventDefault();
  }

  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name:
          <input type="text" ref={this.input} />
        </label>
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

[在 CodePen 上試試看！](#)

由於 uncontrolled component 保持了 DOM 裡的唯一的真相來源，有的時候使用 uncontrolled component 時更容易整合 React 和非 React 的程式碼。如果你想有個又快又髒的方法，它也可以減少一些程式碼。否則，通常應使用 controlled component。

如果仍不清楚在特定情況下應使用哪種類型的 component，你可能會覺得[這篇關於控制與不可控制輸入的文章](#)有所幫助。

預設值

在 React 的 render 生命週期裡，表單上的 `value` attribute 會覆寫掉 DOM 的值。在 uncontrolled component 裡，你常常會希望 React 去指定初始值，但讓之後的更新保持不可控制的。為了處理這種情況，你可以指定 `defaultValue` attribute 而非 `value`。在 component mount 後改變 `defaultValue` 屬性不會造成任何在 DOM 裡面的值更新。

```
render() {
  return (
    <form onSubmit={this.handleSubmit}>
      <label>
```

```
Name:  
  <input  
    defaultValue="Bob"  
    type="text"  
    ref={this.input} />  
  </label>  
  <input type="submit" value="Submit" />  
</form>  
);  
}
```

相同地，`<input type="checkbox">` 和 `<input type="radio">` 支援 `defaultChecked`，而 `<select>` 和 `<textarea>` 支援 `defaultValue`。

檔案輸入標籤

在 HTML 裡，`<input type="file">` 讓使用者能夠選擇從他們的裝置儲存來上傳一個或多個檔案到伺服器，或由 JavaScript 透過 [File API](#) 來處理。

```
<input type="file" />
```

在 React 裡，`<input type="file" />` 永遠都是 uncontrolled component，因為它的值只能被使用者設定，而無法由程式碼來設定。

你應該使用 File API 來與檔案之間互動。以下範例顯示如何建立一個 [ref 到 DOM 節點上](#) 來取得在送出的 handler 的檔案：

```
embed:uncontrolled-components/input-type-file.js
```

[在 CodePen 上試試看！](#)

[Go to TOC](#)

Web Components

React 和 [Web Component](#) 是為了解決不同的問題所建立的。Web Component 為了可重複使用的 component 提供了強大的封裝，而 React 提供了一個宣告式函式庫，使 DOM 與你的資料保持同步。這兩個目標是相輔相成的。作為開發人員，你可以自由地在 Web Component 中使用 React，或在 React 中使用 Web Component，或兩者都是。

大部分使用 React 的人沒有使用 Web Component，但你可能會想要使用，尤其是你用到了使用 Web Component 所寫的第三方的 UI component。

在 React 裡使用 Web Component

```
class HelloMessage extends React.Component {
  render() {
    return <div>Hello <x-search>{this.props.name}</x-search>!</div>;
  }
}
```

注意：

Web Component 通常會公開命令式的 API。例如，一個 `video` Web Component 可能會公開 `play()` 和 `pause()` 函式。為了能夠使用 Web Component 的命令式 API，你會需要使用 `ref` 來跟 DOM 節點直接互動。如果你使用第三方的 Web Component，最好的解法是寫一個 React component 來包住你的 Web Component。

Web Component 所發出的事件可能不會被正確的傳遞到 React 的 render tree。你將會需要手動把 event handler 附加到你的 React component 內處理這些事件。

一個常見的疑惑是 Web Component 使用「`class`」而不是「`className`」。

```
function BrickFlipbox() {
  return (
    <brick-flipbox class="demo">
      <div>front</div>
      <div>back</div>
    </brick-flipbox>
  );
}
```

在你的 Web Component 裡使用 React

```
class XSearch extends HTMLElement {
  connectedCallback() {
    const mountPoint = document.createElement('span');
    this.attachShadow({ mode: 'open' }).appendChild(mountPoint);

    const name = this.getAttribute('name');
    const url = 'https://www.google.com/search?q=' + encodeURIComponent(name);
    const root = ReactDOM.createRoot(mountPoint);
    root.render(<a href={url}>{name}</a>);
  }
}
```

```
    }
}
customElements.define('x-search', XSearch);
```

注意：

如果你用 Babel 轉換了 class，這段程式碼將不會正常運作。查看[這個 issue](#) 裡面的討論。在載入 web component 之前引入 [custom-elements-es5-adapter](#) 來解決這個問題。

[Go to TOC](#)

Colophon

This book is created by using the following sources:

- React - 漢語
- GitHub source: reactjs/zh-hant.reactjs.org/content
- Created: 2022-11-27
- Bash v5.2.2
- Vivliostyle, <https://vivliostyle.org/>
- By: @shinokada
- GitHub repo: <https://github.com/shinokada/markdown-docs-as-pdf>