学习 React

邵建伟

2019 年 4 月 25 日

目录

第一部分 React 文档

1 主要概念

1.1 Hello World

1.2 JSX 简介

JSX 用来声明 react 的元素，编译后变成 JavaScript 对象。

1.3 元素渲染

根 DOM 节点：

```
<div id="root"></div>
```

渲染元素到根 DOM 节点：

```
const element = <h1>Hello, World!</h1>;
ReactDOM.render(element, document.getElementById("root"));
```

1.4 组件 & props

使用函数或类来声明组件，它接受输入值 props，返回 React 元素，props 是只读的。

组件可以在它的输出中引用其它组件，所以尽可能的将组件切分成更小的组件。

1.5 state & 生命周期

state 与 props 类似，但是 state 是私有的，并且完全受控于当前组件。从该 state 派生的任何数据或 UI 只能影响组件构成的树中“低于”它们的组件，这通常会被叫做“自上而下”或是“单向”的数据流。

- setState：构造函数是唯一可以给 this.state 赋值的地方，其余使用setState ({...})
- 异步更新：出于性能考虑，React 可能会把多个setState()调用合并成一个调用，this.props 和 this.state 可能会异步更新，如果需要依赖上一个 state 和当前 props，使用setState ((prevState, props)=>({...}))
- 浅拷贝：调用setState()时，React 只会把提供的对象合并到当前的 state，不会改变其他的属性

1.6 事件处理

React 元素的事件处理：

- 事件的命名采用小驼峰式，而不是纯小些
- 使用 JSX 语法时，需要传入一个函数作为事件处理函数，而不是一个字符串
- 显示调用 e.preventDefault() 才能阻止默认行为
- 正确绑定类方法的 this
  1. 构造方法里手动绑定：this.handleX = this.handleX.bind(this)
  2. 使用实验性的 public class fields 语法：handleX = e => {...}
  3. 回调函数使用箭头函数：e => this.handleX(e)[1]
- 向事件处理程序传递参数：在循环中，通常我们会为事件处理函数传递额外的参数
  1. 箭头函数的事件对象必须显式传递：e => handleX(x, e)[2]
  2. bind 方式的事件对象将被隐式传递：handleX.bind(this, x)[3]
  3. 通过 data-属性传递参数：适用于优化大量元素或使用依赖于 React.PureComponent 相等性检查的渲染树

```
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 (
```

[1] 每次渲染都会创建不同的回调函数，如果该回调函数作为 prop 传入子组件时，这些组件可能会进行额外的重新渲染

[2] 类方法则是 this.handleX

[3] 类方法则是 this.handleX

```
      <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>
    )
  }
}
```

1.7 条件渲染

- if: if (condition) {...}
- 与运算符: condition && ...
- 三目运算符: condition ? ... : ...

组件返回 **null** 会在渲染时隐藏它自身，但并不会影响它的的生命周期。

如果条件过于复杂，考虑提取组件。

1.8 列表 & Keys

在 **map()** 方法中的元素需要设置 **key** 属性。**Key** 帮助 **React** 识别哪些元素改变了，比如被添加或删除。通常，我们使用来自数据 **id** 来作为元素的 **key**，不建议使用索引来用作键值。

数组元素中使用的 **key** 在其兄弟节点之间应该是独一无二的。然而，它们不需要是全局唯一的。

key 会传递信息给 **React**，但不会传递给你的组件。如果你的组件中需要使用 **key** 属性的值，请用其他属性名显式传递这个值。

如果 **map()** 中嵌套多层，考虑提取组件。

1.9 表单 & 受控组件

在 **HTML** 中，表单元素（如 <input>、<textarea> 和 <select>）之类的表单元素通常自己维护 **state**，并根据用户输入进行更新。而在 **React** 中，可变状态（**mutable state**）通常保存在组件的 **state** 属性中，并且只能通过使用 **setState()** 来更新。

我们可以把两者结合起来，使 React 的 state 成为“唯一数据源”。渲染表单的 React 组件还控制着用户输入过程中表单发生的操作。被 React 以这种方式控制取值的表单输入元素就叫做“受控组件”。

```
<input type="text" value={this.state.value} onChange={this.
    handleChange} />
```

1.9.1 常见受控组件

表单元素	value	change callback	new value in the callbak
<input type='text' />	value='string'	onChange	event.target.value
<input type='checkbox' />	checked={boolean}	onChange	event.target.checked
<input type='radio' />	checked={boolean}	onChange	event.target.checked
<textarea />	value='string'	onChange	event.target.value
<select />	value='option value'	onChange	event.target.value

在受控组件上指定 value 的值可以防止用户更改输入。如果指定了 value，但输入仍可编辑，则可能是意外地将 value 设置为 undefined 或 null。

1.10 状态提升

在 React 应用中，任何可变数据应当只有一个相对应的唯一“数据源”。通常，state 都是首先添加到需要渲染数据的组件中去。然后，如果其他组件也需要这个 state，那么你可以将它提升至这些组件的最近共同父组件中。

由于 state 是私有的，并且完成受控于当前组件，状态提升后，排查和隔离 bug 所需的工作量将会变少。此外，如果某些数据可以由 props 或 state 推导得出，那么它就不应该存在于 state 中。

1.11 组合 vs 继承

组合：

- 包含关系:有些组件无法提前知晓它们子组件的具体内容,这些组件使用 props.children 来将他们的子组件传递到渲染结果中，少数情况下，自行约定 props。
- 特例关系：一些组件看作是其他组件的特殊实例，比如 WelcomeDialog 可以说是 Dialog 的特殊实例。

如果想要在组件间复用非 UI 的功能，将其提取为一个单独的 JavaScript 模块，如函数、对象或者类，再由组件直接引入。

1.12 如何构建 React 应用

1. 根据单一功能原则将设计好的 UI 划分为组件层级：一个组件只负责一个功能
2. 自上而下或者自下而上构建应用的静态版本：创建的组件只通过 props 传入所需的数据

3. 确定 UI state 的最小（且完整）表示：
   - 该数据是否是由父组件通过 props 传递而来的？如果是，那它应该不是 state
   - 该数据是否随时间的推移而保持不变？如果是，那它应该也不是 state
   - 你能否根据其他 state 或 props 计算出该数据的值？如果是，那它也不是 state
4. 确定 state 放置的位置：找到根据这个 state 进行渲染的所有组件，找到在组件层级上高于所有需要该 state 的组件，如果没有则新建一个
5. 添加反向数据流：处于较低层级的表单组件更新较高层级的 state

2 高级指引

2.1 无障碍

2.2 代码分割

打包是一个将文件引入并合并到一个单独文件的过程，最终形成一个“bundle”。接着在页面上引入该 bundle，整个应用即可一次性加载。

代码分割是由诸如 Webpack（代码分割）和 Browserify（factor-bundle）这类打包器支持的一项技术，能够创建多个包并在运行时动态加载。通过“懒加载”当前用户所需要的内容，能够显著地提高应用性能。

引入代码分割：

- import()：import X, XX from './X'
- React.lazy()：const X = React.lazy(() => import('./X'))[4]

如果模块没有加载完成，使用 suspense 加载指示器为此组件做优雅降级：

```
<Suspense fallback={<div>loading...</div>}><X /></Suspense>
```

如果模块加载失败，使用异常捕获边界技术处理。

基于路由的代码分割：

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

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

const App = () => (
```

[4]React.lazy 目前只支持默认导出（default exports）

```
  <Router>
    <Suspense fallback={<div>Loading...</div>}>
      <Switch>
        <Route exact path="/" component={Home}/>
        <Route path="/about" component={About}/>
      </Switch>
    </Suspense>
  </Router>
);
```

2.3 Context

Context 提供了一个无需为每层组件手动添加 props，就能在组件树间进行数据传递的方法。主要应用场景在于很多不同层级的组件需要访问同样一些的数据，比如当前的 locale，theme，或者一些缓存数据。

- React.createContext: 创建一对 Provider, Consumer 。当 React 渲染 context 组件 Consumer 时，它将从组件树的上层中最接近的匹配的 Provider 读取当前的 context 值。如果上层的组件树没有一个匹配的 Provider，而此时需要渲染一个 Consumer 组件，那么使用 defaultValue 。这有助于在不封装它们的情况下对组件进行测试。

  ```
  const {Provider, Consumer} = React.createContext(defaultValue
      );
  ```

- Provider: React 组件允许 Consumers 订阅 context 的改变。接收一个 value 属性传递给 Provider 的后代 Consumers。一个 Provider 可以联系到多个 Consumers。Providers 可以被嵌套以覆盖组件树内更深层次的值。

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

- Consumer: 一个可以订阅 context 变化的 React 组件。接收一个函数作为子节点. 函数接收当前 context 的值并返回一个 React 节点。传递给函数的 value 将等于组件树中上层 context 的最近的 Provider 的 value 属性。如果 context 没有 Provider ，那么 value 参数将等于被传递给 createContext() 的 defaultValue 。

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

例子:

```
const ThemeContext = React.createContext('light');

function App(props) {
  return (
```

```
    <ThemeContext.Provider value="dark">
      <Toolbar />
    </ThemeContext.Provider>
  );
}

function Toolbar(props) {
  return (
    <div>
      <ThemedButton />
    </div>
  );

// Context.Consumer
<ThemeContext.Consumer>{value => /* 基于 context 值进行渲染*/}</
    ThemeContext.Consumer>

// public class fields语法
class ThemedButton extends React.Component {
  static contextType = ThemeContext;
  render() {
    return <Button theme={this.context} />;
  }
}

// 第二种写法
class ThemedButton extends React.Component {
  render() {
    return <Button theme={this.context} />;
  }
}

ThemeButton.contextType = ThemeContext;
```

动态 context & 父子耦合:

```
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      value: {something: 'something'},
      changeValue: something => this.setState({something})
    };
```

```
  }

  render() {
    return (
      <Provider value={this.state.value}>
        <Content />
      </Provider>
    );
  }
}
```

2.4 异常捕获边界

部分 UI 的异常不应该破坏了整个应用。错误边界是用于捕获其子组件树 JavaScript 异常，记录错误并展示一个回退的 UI 的 React 组件，而不是整个组件树的异常。错误边界在渲染期间、生命周期方法内、以及整个组件树构造函数内捕获错误。 [5]

一个类组件变成一个错误边界。如果它定义了生命周期方法 static getDerivedStateFromError() 或者 componentDidCatch() 中的任意一个或两个。当一个错误被扔出后，使用 static getDerivedStateFromError() 渲染一个退路 UI。使用 componentDidCatch() 去记录错误信息。

从 React 16 起，任何未被错误边界捕获的错误将导致卸载整个 React 组件树。

错误边界的粒度是由你决定。你可以将其包装在最顶层的路由组件显示给用户”有东西出错”消息，就像服务端框架经常处理崩溃一样。你也可以将单独的插件包装在错误边界内以保护应用其他部分不崩溃。

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

  static getDerivedStateFromError(error) {
    // Update state so the next render will show the fallback UI.
    return { hasError: true };
  }

  componentDidCatch(error, info) {
    // You can also log the error to an error reporting service
    logErrorToMyService(error, info);
  }
```

[5] 异常捕获边界不捕获事件处理、异步代码、服务端渲染和错误边界自身抛出来的错误。

```
  render() {
    if (this.state.hasError) {
      // You can render any custom fallback UI
      return <h1>Something went wrong.</h1>;
    }

    return this.props.children;
  }
}

<ErrorBoundary>
  <MyWidget />
</ErrorBoundary>
```

2.5 Refs 转发

Refs 转发就是暴露子组件的 ref。一般用于可以访问底层 DOM 节点的组件中。

```
const FancyButton = React.forwardRef((props, ref) => (
  <button ref={ref} className="FancyButton">
    {props.children}
  </button>
));

// You can now get a ref directly to the DOM button:
const ref = React.createRef();
<FancyButton ref={ref}>Click me!</FancyButton>;
```

2.6 Fragments

React 中一个常见模式是为一个组件返回多个元素。Fragments 可以让你聚合一个子元素列表，并且不在 DOM 中增加额外节点。

```
function Comp(props) {
  return (
    <React.Fragment>
      <ChildA />
      <ChildB />
      <ChildC />
    </React.Fragment>
  );
}
```

简写形式：<></>

2.7 高阶组件 HOC

在 React 中，组件是代码复用的主要单元，高阶组件则是复用组件横向逻辑的一种模式。本质上就是一个函数，该函数接受一个组件作为参数，并返回一个新的组件。

- 不要改变原始组件，使用组合

```
function logProps(WrappedComponent) {
  return class extends React.Component {
    componentWillReceiveProps(nextProps) {
    console.log('Current props: ', this.props);
    console.log('Next props: ', nextProps);
    }
    render() {
      // 用容器包裹输入组件，不要修改它，漂亮!
      return <WrappedComponent {...this.props} />;
    }
  }
}
```

- 约定：贯穿传递不相关 props 属性给被包裹的组件

```
render() {
  // 过滤掉专用于这个阶组件的props属性，
  // 不应该被贯穿传递
  const { extraProp, ...passThroughProps } = this.props;

  // 向被包裹的组件注入props属性，这些一般都是状态值或
  // 实例方法
  const injectedProp = someStateOrInstanceMethod;

  // 向被包裹的组件传递props属性
  return (
    <WrappedComponent
      injectedProp={injectedProp}
      {...passThroughProps}
    />
  );
}
```

- 约定：最大化的组合性，即返回高阶组件的函数

```
// React Redux's `connect`
const ConnectedComment = connect(commentSelector,
  commentActions)(Comment);
```

- 约定：包装显示名字以便于调试

```
function withSubscription(WrappedComponent) {
  class WithSubscription extends React.Component {/* ... */}
  WithSubscription.displayName = `WithSubscription(${
     getDisplayName(WrappedComponent)})`;
  return WithSubscription;
}

function getDisplayName(WrappedComponent) {
  return WrappedComponent.displayName || WrappedComponent.
     name || 'Component';
}
```

使用高阶组件的注意点：

- 应用高阶组件在组件定义的外面，不要在 render 方法内使用高阶组件[6]

```
render() {
  // 每一次渲染，都会创建一个新的EnhancedComponent版本
  // EnhancedComponent1 !== EnhancedComponent2
  const EnhancedComponent = enhance(MyComponent);
  // 那引起每一次都会使子对象树完全被卸载/重新加载
  return <EnhancedComponent />;
}
```

- 必须将静态方法做拷贝

```
// 手动拷贝
function enhance(WrappedComponent) {
  class Enhance extends React.Component {/*...*/}
  // 必须得知道要拷贝的方法 :(
  Enhance.staticMethod = WrappedComponent.staticMethod;
  return Enhance;
}

// 自动拷贝
import hoistNonReactStatic from 'hoist-non-react-statics';
function enhance(WrappedComponent) {
  class Enhance extends React.Component {/*...*/}
  hoistNonReactStatic(Enhance, WrappedComponent);
  return Enhance;
}
```

[6]React 的差分算法使用组件标识确定是否更新现有的子树或扔掉它并重新挂载一个新的。如果 render 方法返回的组件和前一次渲染返回的组件是完全相同的 (===)，React 就递归地更新子树，这是通过差分它和新的那个完成。如果它们不相等，前一个子树被完全卸载掉。

```
// 导出静态方法
// ...export the method separately...
export { someFunction };

// ...and in the consuming module, import both
import MyComponent, { someFunction } from './MyComponent.js';
```

- 一般来说，高阶组件可以传递所有的 props 属性给包裹的组件，但是不能传递 refs 引用。使用 React.forwardRef 的 API 来解决这一问题

2.8 与第三方库协同

2.8.1 集成带有 DOM 操作的插件

React 不会理会 React 自身之外的 DOM 操作。它根据内部虚拟 DOM 来决定是否需要更新，而且如果同一个 DOM 节点被另一个库操作了，React 会觉得困惑而且没有办法恢复。避免冲突的最简单方式就是防止 React 组件更新。你可以渲染无需更新的 React 元素，比如一个空的 。

集成 DOM 操作的插件在 componentDidMount 中绑定事件监听到 DOM 上，为了避免内存泄漏，在 componentWillUnmount 中注销监听。在 React 中，props 可以在不同的时间有不同的值，这意味着从集成的角度来看，我们应因 props 的更新而在 componentDidUpdate 中手动更新 DOM，因为我们已经不再使用 React 来帮我们管理 DOM 了。

```
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>
    );
  }
}
```

2.8.2 和其他视图库集成

虽然 React 通常被用来在启动的时候加载一个单独的根 React 组件到 DOM 上，ReactDOM.render() 同样可以在 UI 的独立部分上多次调用，这些部分可以小到一个按钮，也可以大到一个应用。

2.8.3 和 Model 层集成

React 应用推荐使用单向数据流动数据模型，例如 React state，Flux，或者 Redux。和其他框架的 Model 层集成时，建议每当 model 中的属性变化时都把它提取成简单数据，并且把这个逻辑放在一个独立的地方，例如高阶组件。

```
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} />;
    }
  }
}

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' });
ReactDOM.render(
  <Example model={model} />,
  document.getElementById('root')
);
```

2.9 深入 JSX

实际上，JSX 仅仅只是 React.createElement(component, props, ...children) 函数的语法糖。

2.9.1 指定 React 元素类型

- React 必须在作用域内：由于 JSX 会编译为 React.createElement 调用形式，所以 React 库也必须包含在 JSX 代码作用域内
- 在 JSX 类型中使用点表示法

```
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" />;
}
```

- 用户定义的组件必须以大写字母开头
- 在运行时选择类型：首先将类型赋值给一个大写字母开头的变量

```
import React from 'react';
import { PhotoStory, VideoStory } from './stories';

const components = {
```

```
  photo: PhotoStory,
  video: VideoStory
};

function Story(props) {
  // 正确! JSX 类型可以是大写字母开头的变量。
  const SpecificStory = components[props.storyType];
  return <SpecificStory story={props.story} />;
}
```

2.9.2 JSX 中的 props

- JavaScript 表达式
- 字符串字面量
- props 默认值为 true
- 属性扩散：<MyComponent {...props} />

2.9.3 JSX 中的子元素

- 字符串字面量：JSX 会移除行首尾的空格以及空行。与标签相邻的空行均会被删除，文本字符串之间的新行会被压缩为一个空格
- JSX 子元素：将字符串字面量与 JSX 子元素一起使用，这也是 JSX 类似 HTML 的一种表现；React 组件也能够返回存储在数组中的一组元素

```
render() {
  // 不需要用额外的元素包裹列表元素!
  return [
  // 不要忘记设置 key :)
    <li key="A">First item</li>,
    <li key="B">Second item</li>,
    <li key="C">Third item</li>,
  ];
}
```

- JavaScript 表达式
  - map 生成任意长度的列表
  - {...} 替代模版字符串
- true、false、null 以及 undefined 将会忽略
  - 直接作为子元素不会被渲染，除非转换为字符串
  - <>{true && <MyComponent />}</>

2.10 性能优化

- 开发应用时使用开发模式，部署应用时使用生产模式
- 虚拟化长列表：在有限的时间内仅渲染有限的内容
- 避免调停：在 shouldComponentUpdate 中返回 false 来跳过整个渲染过程。其包括该组件的 render 调用以及之后的操作[7]
  - shouldComponentUpdate(nextProps, nextState)
  - React.PureComponent：用当前与之前 props 和 state 的浅比较覆写了 shouldComponentUpdate() 的实现，当 props 或 state 是可变时不管用

实现不可变对象：

1. 使用 Array.concat()、Object.assign()、[...array, newValue]、{...object, newKey: newValue} 扩展运算
2. immutable.js

2.11 Portals

Portals 提供了一种很好的将子节点渲染到父组件以外的 DOM 节点的方式。对于 portals 的一个典型用例是当父组件有 overflow: hidden 或 z-index 样式，但你需要子组件能够在视觉上“跳出（break out）”其容器。例如，对话框、hovercards 以及提示框。

尽管 portals 可以被放置在 DOM 树的任何地方，但仍存在于 React 树中，其行为和普通的 React 子节点行为一致，一个从 portal 内部会触发的事件会一直冒泡至包含 React 树的祖先，例如在 #app-root 里的 Parent 组件能够捕获到未被捕获的从兄弟节点 #modal-root 冒泡上来的事件。

```
<html>
  <body>
    <div id="app-root"></div>
    <div id="modal-root"></div>
  </body>
</html>
```

```
// These two containers are siblings in the DOM
const appRoot = document.getElementById('app-root');
const modalRoot = document.getElementById('modal-root');

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

[7] React 构建并维护了一套内部的 UI 渲染描述。它包含了来自你的组件返回的 React 元素。该描述使得 React 避免创建 DOM 节点以及没有必要的节点访问，被称为虚拟 DOM。

```
    this.el = document.createElement('div');
  }

  componentDidMount() {
    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() {
    // This will fire when the button in Child is clicked,
    // updating Parent's state, even though button
    // is not direct descendant in the DOM.
    this.setState(prevState => ({
      clicks: prevState.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() {
  // The click event on this button will bubble up to parent,
  // because there is no 'onClick' attribute defined
  return (
    <div className="modal">
      <button>Click</button>
    </div>
  );
}

ReactDOM.render(<Parent />, appRoot);
```

2.12 不使用 ES6

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

var SetIntervalMixin = {
  componentWillMount: function() {
    this.intervals = [];
  },

  setInterval: function() {
    this.intervals.push(setInterval.apply(null, arguments));
  },

  componentWillUnmount: function() {
    this.intervals.forEach(clearInterval);
  }
};

var Greeting = createReactClass({
  // 使用mixin
```

```
  mixins: [SetIntervalMixin],

  // 函数和类组件的defaultProps
  getDefaultProps: function() {
    return {
      name: 'Mary'
    };
  },

  // 初始化state
  getInitialState: function() {
    return {message: 'Hello!'};
  },

  // 无需.bind
  handleClick: function() {
    alert(this.state.message);
  },

  // render方法
  render: function() {
    return (
      <Fragment>
        <h1>Hello, {this.props.name}</h1>
        <button onClick={this.handleClick}>say hello</button>
      </Fragment>
    );
  }
});
```

2.13 不使用 JSX

每一个 JSX 元素都只是 React.createElement(component, props, ...children) 的语法糖。

2.14 协调

在某一时间节点调用 React 的 render() 方法，会创建一棵由 React 元素组成的树。在下一次 state 或 props 更新时，相同的 render() 方法会返回一棵不同的树。React 需要基于这两棵树之间的差别来判断如何有效率的更新 UI 以保证当前 UI 与最新的树保持同步。

2.14.1 Diffing 算法

两个假设:

1. 两个不同类型的元素会产生出不同的树;
2. 开发者可以通过 key prop 来暗示哪些子元素在不同的渲染下能保持稳定;

当对比两颗树时，React 首先比较两棵树的根节点，在处理完当前节点之后，React 继续对子节点进行递归。

- 不同类型元素: 拆卸原有的树并且建立起新的树
- 相同类型元素: 保留 DOM 节点，仅比对及更新有改变的属性
- 对子节点进行递归: 在默认条件下，当递归 DOM 节点的子元素时，React 会同时遍历两个子元素的列表; 当产生差异时，生成一个 mutation。当子元素拥有 key 时，React 使用 key 来匹配原有树上的子元素以及最新树上的子元素。

2.15 Refs & DOM

在典型的 React 数据流中，props 是父组件与子组件交互的唯一方式。要修改一个子组件，你需要使用新的 props 来重新渲染它。Refs 提供了一种方式，允许我们访问 DOM 节点或在 render 方法中创建的 React 元素。

- 管理焦点，文本选择或媒体播放
- 触发强制动画
- 集成第三方 DOM 库

创建 Ref:

```
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.myRef = React.createRef();
  }

  render() {
    return <div ref={this.myRef} />;
  }
}

// 回调形式
class MyComponent extends React.Component {
  constructor(props) {
    super(props);
    this.myRef = null;
```

```
  }

  render() {
    return <div ref={element => this.myRef = element} />;
  }
}
```

访问 Ref:

```
const node = this.myRef.current;
```

ref 的值根据节点的类型而有所不同:

- 当 ref 属性用于 HTML 元素时，它是底层 DOM 元素
- 当 ref 属性用于自定义 class 组件时，它是组件的挂载实例[8]

通过 Refs 转发将子组件的 DOM Refs 暴露给父组件。

如果 ref 回调函数是以内联函数的方式定义的，在更新过程中它会被执行两次，第一次传入参数 null，然后第二次会传入参数 DOM 元素。这是因为在每次渲染时会创建一个新的函数实例，所以 React 清空旧的 ref 并且设置新的。通过将 ref 的回调函数定义成 class 的绑定函数的方式可以避免上述问题。

2.16 Render Props

在 React 中，组件是代码复用的主要单元，render prop 则是复用组件横向逻辑的一种模式。本质上就是一个用于告知组件需要渲染什么内容的函数 prop。

```
<DataProvider render={data => (
  <h1>Hello {data.target}</h1>
)}/>

<DataProvider>
  {data => <h1>Hello {data.target}</h1>}
</DataProvider>
```

Render Props 与 React.PureComponent 一起使用时，如果在 render 方法里创建函数，那么使用 render prop 会抵消使用 React.PureComponent 带来的优势，定义一个 prop 作为实例方法可以避免上述问题。

[8]不能在函数组件上使用 ref 属性，因为它没有实例，但可以在函数组件内部使用 ref 属性，只要它指向一个 DOM 元素或 class 组件

2.16.1 与 HOC 的关系

任一模式都可与 render prop 一起使用，例如 HOC：

```
function withMouse(Component) {
  return class extends React.Component {
    render() {
      return (
        <Mouse render={mouse => (
          <Component {...this.props} mouse={mouse} />
        )}/>
      );
    }
  }
}
```

2.17 静态类型检查

- Flow
- TypeScript
- Reason
- Kotlin

2.18 严格模式

StrictMode 用于检测以下问题：

- 不安全生命周期的组件
- 旧氏字符串 ref
- 已废弃的 findDomNode
- 意外的副作用
- 旧氏 context

```
import React from 'react';

function ExampleApplication() {
  return (
    <div>
      <Header />
      <React.StrictMode>
        <div>
          <ComponentOne />
```

```
          <ComponentTwo />
        </div>
      </React.StrictMode>
      <Footer />
    </div>
  );
}
```

2.19 使用 PropTypes 类型检查

要在组件的 props 上进行类型检查，只需配置特定的 propTypes 属性：

```
import PropTypes from 'prop-types';

MyComponent.propTypes = {
  // 你可以将属性声明为 JS 原生类型，默认情况下
  // 这些属性都是可选的。
  optionalArray: PropTypes.array,
  optionalBool: PropTypes.bool,
  optionalFunc: PropTypes.func,
  optionalNumber: PropTypes.number,
  optionalObject: PropTypes.object,
  optionalString: PropTypes.string,
  optionalSymbol: PropTypes.symbol,

  // 任何可被渲染的元素（包括数字、字符串、元素或数组）
  // (或 Fragment) 也包含这些类型。
  optionalNode: PropTypes.node,

  // 一个 React 元素。
  optionalElement: PropTypes.element,

  // 你也可以声明 prop 为类的实例，这里使用
  // JS 的 instanceof 操作符。
  optionalMessage: PropTypes.instanceOf(Message),

  // 你可以让你的 prop 只能是特定的值，指定它为
  // 枚举类型。
  optionalEnum: PropTypes.oneOf(['News', 'Photos']),

  // 一个对象可以是几种类型中的任意一个类型
  optionalUnion: PropTypes.oneOfType([
    PropTypes.string,
    PropTypes.number,
```

```
  PropTypes.instanceOf(Message)
]),

// 可以指定一个数组由某一类型的元素组成
optionalArrayOf: PropTypes.arrayOf(PropTypes.number),

// 可以指定一个对象由某一类型的值组成
optionalObjectOf: PropTypes.objectOf(PropTypes.number),

// 可以指定一个对象由特定的类型值组成
optionalObjectWithShape: PropTypes.shape({
  color: PropTypes.string,
  fontSize: PropTypes.number
}),

// 你可以在任何 PropTypes 属性后面加上 `isRequired` ，确保
// 这个 prop 没有被提供时，会打印警告信息。
requiredFunc: PropTypes.func.isRequired,

// 任意类型的数据
requiredAny: PropTypes.any.isRequired,

// 你可以指定一个自定义验证器。它在验证失败时应返回一个 Error
   对象。
// 请不要使用 `console.warn` 或抛出异常，因为这在 `onOfType` 中
   不会起作用。
customProp: function(props, propName, componentName) {
  if (!/matchme/.test(props[propName])) {
    return new Error(
      'Invalid prop `' + propName + '` supplied to' +
      ' `' + componentName + '`. Validation failed.'
    );
  }
},

// 你也可以提供一个自定义的 `arrayOf` 或 `objectOf` 验证器。
// 它应该在验证失败时返回一个 Error 对象。
// 验证器将验证数组或对象中的每个值。验证器的前两个参数
// 第一个是数组或对象本身
// 第二个是他们当前的键。
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.'
      );
    }
  })
};
```

通过配置特定的 defaultProps 属性来定义 props 的默认值： [9]

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

Greeting.defaultProps = {
  name: 'Stranger'
};

// Babel, 提案
class Greeting extends React.Component {
  static defaultProps = {
    name: 'stranger'
  }

  render() {
    return (
      <div>Hello, {this.props.name}</div>
    )
  }
}
```

2.20 非受控组件

在 HTML 中，表单元素（如 <input>、<textarea> 和 <select>）之类的表单元素通常自己维护 state，并根据用户输入进行更新，使用 ref 来从 DOM 节点中获取表单数据的方式控制取值的表单输入元素就叫做“非受控组件”。

[9] propTypes 类型检查发生在 defaultProps 赋值后，所以类型检查也适用于 defaultProps

在 React 渲染生命周期时，表单元素上的 value 将会覆盖 DOM 节点中的值，在非受控组件中，你经常希望 React 能赋予组件一个初始值，但是不去控制后续的更新。在这种情况下, 你可以指定一个 defaultValue 属性，而不是 value。

2.20.1 默认值

表单元素	默认值
<input type="checkbox">、<input type="radio">	defaultChecked
<select>、<textarea>	defaultValue

2.20.2 文件输入

在 React 中，<input type="file" /> 始终是一个非受控组件。

2.20.3 受控表单组件 vs 非受控表单组件

相较于受控表单组件，非受控组件将真实数据储存在 DOM 节点中，所以它缺少及时的数据校验、动态表单等一些特性。

2.21 Web Components

3 Hook

3.1 Hook 简介

Hook 是一些可以在函数组件里“钩入” React state 及生命周期等特性的函数。React 内置了一些像 useState、useEffect、useContext、useReducer 这样的 Hook。也可以创建自定义的 Hook 来复用不同组件之间的状态逻辑。Hook 为已知的 React 概念（props，state，context，refs 以及生命周期）提供了更直接的 API，它可以在不编写 class 的情况下使用 state 以及其他的 React 特性。

动机：

- 在组件之间复用状态逻辑很难：React 没有提供将可复用性行为“附加”到组件的途径，由 providers，consumers，高阶组件，render props 等其他抽象层组成的组件会形成“嵌套地狱”。
- 复杂组件变得难以理解：组件起初很简单，但是逐渐会被状态逻辑和副作用充斥。每个生命周期常常包含一些不相关的逻辑。在多数情况下，不可能将组件拆分为更小的粒度，因为状态逻辑无处不在。
- 难以理解的 class：this 绑定、class 某些写法不符合组件预编译的需求。

3.2 使用 State Hook

通过在函数组件里调用 useState 来给组件添加内部 state。React 会在重复渲染时保留这个 state。useState 会返回一对值：当前状态和一个让你更新它的函数，你可以在事件处理函数中或其他一些地方调用这个函数。

在一个组件中多次使用 State Hook 则声明多个 state 变量。

示例：

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

function Example() {
  // 声明一个叫 "count" 的 state 变量
  const [count, setCount] = useState(0);

  return (
    <div>
      <p>You clicked {count} times</p>
      <button onClick={() => setCount(count + 1)}>
        Click me
      </button>
    </div>
  );
}
```

3.2.1 单个 state vs 多个 state

不同于类组件的 this.setState()，State Hook 更新 state 变量总是替换它而不是合并它，如果把所有 state 放在同一个 setState 调用中，更新时需要手动合并。

把所有 state 放在同一个 setState 调用中，或是每一个字段都对应一个 setState 调用，这两方式都能跑通。当在这两个极端之间找到平衡，然后把相关 state 组合到几个独立的 state 变量时，组件就会更加的可读。如果 state 的逻辑开始变得复杂，使用 reducer 或自定义 Hook 来管理它。

3.3 使用 Effect Hook

通过在函数组件里调用 useEffect 来给组件添加数据获取、订阅或者手动修改 DOM 等副作用的能力。它跟类组件中的 componentDidMount、componentDidUpdate 和 componentWillUnmount 具有相同的用途，只不过被合并成了一个 API。默认情况下，React 会在每次渲染后调用副作用函数，包括第一次渲染的时候。useEffect 还可以通过返回一个函数来指定如何“清除”副作用。

在一个组件中多次使用 Effect Hook 则执行多个副作用。

无需清除的 effect 示例：

```
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>
  );
}
```

需要清除的 effect 示例：

```
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);
    // Specify how to clean up after this effect:
    return function cleanup() {
      ChatAPI.unsubscribeFromFriendStatus(props.friend.id,
          handleStatusChange);
    };
  });

  if (isOnline === null) {
    return 'Loading...';
  }
  return isOnline ? 'Online' : 'Offline';
}
```

3.3.1 为什么每次更新的时候都要运行 Effect?

在类组件中忘记正确地处理 componentDidUpdate 是 React 应用中常见的 bug 来源，useEffect 无需特定的代码默认处理，它会在调用一个新的 effect 之前对前一个 effect 进行清理。

3.3.2 通过跳过 Effect 进行性能优化

如果某些特定值在两次重渲染之间没有发生变化，传递数组作为 useEffect 的第二个可选参数，通知 React 跳过对 effect 的调用。 [10]

```
useEffect(() => {
  document.title = `You clicked ${count} times`;
}, [count]); // 仅在 count 更改时更新

useEffect(() => {
  function handleStatusChange(status) {
    setIsOnline(status.isOnline);
  }

  ChatAPI.subscribeToFriendStatus(props.friend.id,
     handleStatusChange);
  return () => {
    ChatAPI.unsubscribeFromFriendStatus(props.friend.id,
       handleStatusChange);
  };
}, [props.friend.id]); // 仅在 props.friend.id 发生变化时, 重新订
   阅
```

3.4 Hook 规则

- 只能在函数最顶层调用 Hook。不要在循环、条件判断或者子函数中调用。[11]
- 只能在 React 的函数组件或自定义 Hook 中调用 Hook。不要在其他 JavaScript 函数中调用。

3.4.1 只在函数最顶层使用 Hook

不要在循环，条件或嵌套函数中调用 Hook，因为 React 靠的是 Hook 调用的顺序知道哪个 useState 调用对应于哪个 state 变量以及匹配前后两次渲染中的每一个 effect 的。

```
// ————————————
// 首次渲染
// ————————————
```

[10] 传入 []，effect 只执行一次

[11] Hook 调用不能被放在循环中，但可以为列表项抽取一个单独的组件。

```
useState('Mary')            // 1. 使用 'Mary' 初始化变量名为 name
    的 state
useEffect(persistForm)      // 2. 添加 effect 以保存 form 操作
useState('Poppins')         // 3. 使用 'Poppins' 初始化变量名为
    surname 的 state
useEffect(updateTitle)      // 4. 添加 effect 以更新标题

// ---------------
// 二次渲染
// ---------------
useState('Mary')            // 1. 读取变量名为 name 的 state（参数
    被忽略）
useEffect(persistForm)      // 2. 替换保存 form 的 effect
useState('Poppins')         // 3. 读取变量名为 surname 的 state
    （参数被忽略）
useEffect(updateTitle)      // 4. 替换更新标题的 effect

// ...
```

如果我们想要有条件地执行一个 effect，可以将判断放到 Hook 的内部。

```
useEffect(function persistForm() {
  //   将条件判断放置在 effect 中
  if (name !== '') {
    localStorage.setItem('formData', name);
  }
});
```

3.4.2 只在 React 的函数组件或自定义 Hook 中调用 Hook

遵循此规则，确保组件的状态逻辑在代码中清晰可见。

3.5 自定义 Hook

通过自定义 Hook，可以将组件状态逻辑提取到可重用的函数中。我们约定如果函数的名字以“use”开头并调用其他 Hook，就说这是一个自定义 Hook。它的应用场景有表单处理、动画、订阅声明、计时器等等。

自定义 Hook 是一种复用状态逻辑的方式，它不复用 state 本身。事实上自定义 Hook 的每次调用都有一个完全独立的 state。 [12]

[12]组件之间重用状态逻辑的方案：高阶组件、render props 及自定义 Hook

示例：

```
function useReducer(reducer, initialState) {
  const [state, setState] = useState(initialState);

  function dispatch(action) {
    const nextState = reducer(state, action);
    setState(nextState);
  }

  return [state, dispatch];
}

function todosReducer(state, action) {
  switch (action.type) {
    case 'add':
      return [...state, {
        text: action.text,
        completed: false
      }];
    // ... other actions ...
    default:
      return state;
  }
}

function Todos() {
  const [todos, dispatch] = useReducer(todosReducer, []);

  function handleAddClick(text) {
    dispatch({ type: 'add', text });
  }

  // ...
}
```

3.6 Hook API 索引

3.6.1 基础 Hook

useState 返回一个 state，以及更新 state 的函数。在初始渲染期间，返回的状态 (state) 与传入的第一个参数 (initialState) 值相同，在后续的重新渲染中，useState 返回的第一个值将始终是更新后最新的 state。

- useState 只会替代而不是合并 state

```
// 展开运算符
setState(prevState => {
  return {...prevState, ...updatedValues};
});

// Object.assign

// useReducer: 适用于管理包含多个子值的 state 对象
```

- 调用 State Hook 的更新函数并传入当前的 state 时，React 将跳过子组件的渲染及 effect 的执行[13]

```
// 普通形式
const [state, setState] = useState(initialState);
setState(newState);

// 函数式更新: 新的state需要通过使用先前的state计算得出
setState(prevState => prevState + 1);

// 惰性初始state
const [state, setState] = useState(() => {
  const initialState = someExpensiveComputation(props);
  return initialState;
});
```

useEffect 接收一个包含命令式、且可能有副作用代码的函数，会在浏览器绘制后延迟执行，但会保证在任何新的渲染前执行。React 将在组件更新前刷新上一轮渲染的 effect，因此不应在函数中执行阻塞浏览器更新屏幕的操作。

```
// 不清除effect
useEffect(() => {});

// 清除effect
useEffect(() => {
  const subscription = props.source.subscribe();
  return () => {
    // 清除函数会在组件卸载前执行，如果组件多次渲染（通常如此），
       则在执行下一个 effect 之前，上一个 effect 就已被清除
    subscription.unsubscribe();
  };
});
```

[13] React 使用 Object.is 比较算法来比较 state

```
// effect条件执行
useEffect(
  () => {
    const subscription = props.source.subscribe();
    return () => {
      subscription.unsubscribe();
    };
  },
  //  只有当 props.source 改变后才会重新创建订阅
  [props.source],
);
```

useContext 接收一个 context 对象（React.createContext 的返回值）并返回该 context 的当前值。 [14]

```
const value = useContext(MyContext);

// 相当于
static contextType = MyContext or <MyContext.Consumer>
```

3.6.2 额外的 Hook

useReducer 是 useState 的替代方案。它接收一个形如 (state, action) => newState 的 reducer，并返回当前的 state 以及与其配套的 dispatch 方法，适用于 state 逻辑较复杂且包含多个子值，或者下一个 state 依赖于之前的 state 等场景。

使用 useReducer 还能给那些会触发深更新的组件做性能优化，因为你可以向子组件传递 dispatch 而不是回调函数。

- React 会确保 dispatch 函数的标识是稳定的，并且不会在组件重新渲染时改变。这就是为什么可以安全地从 useEffect 或 useCallback 的依赖列表中省略 dispatch。
- 如果 Reducer Hook 的返回值与当前 state 相同，React 将跳过子组件的渲染及副作用的执行。

```
const [state, dispatch] = useReducer(reducer, initialArg, init);

// 指定初始state
const [state, dispatch] = useReducer(
  reducer,
  {count: initialCount}
);

// 惰性初始化
```

[14] 当前的 context 值由上层组件中距离当前组件最近的 <MyContext.Provider> 的 value prop 决定。

```
const [state, dispatch] = useReducer(
  reducer,
  initialCount,
  initialCount => ({count: initialCount})
);

// 更新
dispatch({type: 'xxx'});
```

示例:

```
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({initialState}) {
  const [state, dispatch] = useReducer(reducer, initialState);
  return (
    <>
      Count: {state.count}
      <button onClick={() => dispatch({type: 'increment'})}>+</
          button>
      <button onClick={() => dispatch({type: 'decrement'})}>-</
          button>
    </>
  );
}
```

useRef 返回一个值可变的 ref 对象，类似于在类中使用实例字段的方式。其.current 属性被初始化为传入的参数（initialValue）。返回的 ref 对象在组件的整个生命周期内保持不变。当 ref 对象内容发生变化时，useRef 并不会通知你。变更.current 属性不会引发组件重新渲染。如果想要在 React 绑定或解绑 DOM 节点的 ref 时运行某些代码，则需要使用回调 ref 来实现。

```
const refContainer = useRef(initialValue);
```

useImperativeHandle 可以在使用 ref 时自定义暴露给父组件的实例值，其应当与 forwardRef 一起使用。

```
function FancyInput(props, ref) {
  const inputRef = useRef();
  useImperativeHandle(ref, () => ({
    focus: () => {
    inputRef.current.focus();
    }
  }));
  return <input ref={inputRef} ... />;
}
FancyInput = forwardRef(FancyInput);
```

useMemo 返回一个 memoized 值，接受“创建”函数和依赖项数组作为参数，它仅会在某个依赖项改变时才重新计算 memoized 值。如果没有提供依赖项数组，useMemo 在每次渲染时都会计算新的值。 [15]

```
const memoizedValue = useMemo(() => computeExpensiveValue(a, b),
   [a, b]);
```

useCallback 返回一个 memoized 回调函数，把内联回调函数及依赖项数组作为参数传入 useCallback，它将返回该回调函数的 memoized 版本，该回调函数仅在某个依赖项改变时才会更新。useCallback(fn, deps) 相当于 useMemo(() => fn, deps)。

```
const memoizedCallback = useCallback(
  () => {
    doSomething(a, b);
  },
  [a, b],
);
```

useLayoutEffect 的函数签名与 useEffect 相同，但它会在所有的 DOM 变更之后同步调用 effect。可以使用它来读取 DOM 布局并同步触发重渲染。在浏览器执行绘制之前，useLayoutEffect 内部的更新计划将被同步刷新。

3.7 Hooks FAQ

3.7.1 采纳策略

✏ 有什么是 Hook 能做而 class 做不到的?

Hook 通过「自定义 Hook」提供了强大而富有表现力的方式来在组件间复用功能。

[15] useMemo 允许记住一次昂贵的计算。但是，这仅作为一种提示，并不保证计算不会重新运行。

✏ Hook 会替代 render props 和高阶组件吗?

通常，render props 和高阶组件只渲染一个子节点，Hook 服务这个使用场景更加简单。

3.7.2 从 Class 迁移到 Hook

✏ 如何使用 Hook 进行数据获取?

```
const dataFetchReducer = (state, action) => {
  switch (action.type) {
    case 'FETCH_INIT':
      return {
        ...state,
        isLoading: true,
        isError: false
      };
    case 'FETCH_SUCCESS':
      return {
        ...state,
        isLoading: false,
        isError: false,
        data: action.payload,
      };
   case 'FETCH_FAILURE':
     return {
       ...state,
       isLoading: false,
       isError: true,
     };
   default:
     throw new Error();
  }
};

const useDataApi = (initialUrl, initialData) => {
  const [url, setUrl] = useState(initialUrl);

  const [state, dispatch] = useReducer(dataFetchReducer, {
    isLoading: false,
    isError: false,
    data: initialData,
  });

  useEffect(() => {
```

```
    const fetchData = async () => {
      dispatch({ type: 'FETCH_INIT' });

      try {
        const result = await axios(url);

        dispatch({ type: 'FETCH_SUCCESS', payload: result.
            data });
      } catch (error) {
        dispatch({ type: 'FETCH_FAILURE' });
      }
    };

    fetchData();
  }, [url]);

  const doFetch = url => {
    setUrl(url);
  };

  return { ...state, doFetch };
};

function App() {
  const [query, setQuery] = useState('redux');
  const { data, isLoading, isError, doFetch } = useDataApi(
    'http://hn.algolia.com/api/v1/search?query=redux',
    { hits: [] },
  );

  return (
    <Fragment>
      <form
        onSubmit={event => {
          event.preventDefault();
          doFetch(`http://hn.algolia.com/api/v1/search?query=
              ${query}`);
        }}
      >
        <input
          type="text"
          value={query}
```

```
          onChange={event => setQuery(event.target.value)}
        />
        <button type="submit">Search</button>
      </form>

      {isError && <div>Something went wrong ...</div>}

      {isLoading ? (
        <div>Loading ...</div>
      ) : (
        <ul>
          {data.hits.map(item => (
            <li key={item.objectID}>
              <a href={item.url}>{item.title}</a>
            </li>
          ))}
        </ul>
      )}
    </Fragment>
  );
}
```

✏ 有类似实例变量的东西吗?

useRef() Hook 不仅可以用于 DOM refs。「ref」对象是一个 current 属性可变且可以容纳任意值的通用容器，类似于一个 class 的实例属性。

```
function Timer() {
  const intervalRef = useRef();

  useEffect(() => {
    const id = setInterval(() => {
      // ...
    });
    intervalRef.current = id;
    return () => {
      clearInterval(intervalRef.current);
    };
  });

  function handleCancelClick() {
    clearInterval(intervalRef.current);
  }

  // ...
```

```
}
```

✏ 如何获取上一轮的 props 或 state?

通过「useRef」手动实现

```
function usePrevious(value) {
  const ref = useRef();
  useEffect(() => {
    ref.current = value;
  });
  return ref.current;
}

function Counter() {
  const [count, setCount] = useState(0);
  const prevCount = usePrevious(count);
  return <h1>Now: {count}, before: {prevCount}</h1>;
}
```

✏ 为什么会在函数中看到陈旧的 props 和 state?

- 异步代码[16]
- 使用了 useEffect、useMemo、useCallback 或 useImperativeHandle 的第二个参数，但没指定所有的依赖

✏ 如何测量 DOM 节点?

可以使用「callback ref」测量一个 DOM 节点的位置或是尺寸，每当 ref 被附加到一个另一个节点，React 就会调用 callback。使用「callback ref」可以确保即便子组件延迟显示被测量的节点 (比如为了响应一次点击)，我们依然能够在父组件接收到相关的信息，以便更新测量结果。[17]

```
function useClientRect() {
  const [rect, setRect] = useState(null);
  const ref = useCallback(node => {
    if (node !== null) {
    setRect(node.getBoundingClientRect());
    }
  }, []);
  return [rect, ref];
}

function MeasureExample() {
```

[16]组件内部的任何函数，包括事件处理函数和 effect，都是从它被创建的那次渲染中被「看到」的。解决方案：使用「useRef」保存最新的 state

[17]因为当 ref 是一个对象时，它并不会通知当前 ref 的值的变化。

```
  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>
      }
    </>
  );
}
```

3.7.3 性能优化

✏ 在依赖列表中省略函数是否安全?

一般来说，不安全。如果指定了一个依赖列表作为 useEffect、useMemo、useCallback 或 useImperativeHandle 的最后一个参数，它必须包含参与那次 React 数据流的所有值。这就包含了 props、state，以及任何由它们衍生而来的东西。只有当函数（以及它所调用的函数）不引用 props、state 以及由它们衍生而来的值时，你才能放心地把它们从依赖列表中省略。推荐方案：

- 把函数移动到 effect 内部，指定依赖列表
- 把函数移动到组件外部
- 对于纯计算函数，在 effect 之外调用它，并让 effect 依赖于它的返回值
- 把函数加入 effect 的依赖但把它的定义包裹进 useCallback Hook

```
function ProductPage({ productId }) {
  // 用 useCallback 包裹以避免随渲染发生改变
  const fetchProduct = useCallback(() => {
    // ... Does something with productId ...
  }, [productId]); // useCallback 的所有依赖都被指定了

  return <ProductDetails fetchProduct={fetchProduct} />;
}

function ProductDetails({ fetchProduct })
  useEffect(() => {
    fetchProduct();
  }, [fetchProduct]); // useEffect 的所有依赖都被指定了
  // ...
}
```

✏ 如果 effect 的依赖频繁变化该怎么办?

有时候 effect 会读取一些频繁变化的值，如果试图在依赖列表中省略那个 state 会引

起 Bug，指定 state 作为依赖列表就能修复这个 Bug，但会导致内部每次改变时都被重置。推荐方案：

- 使用 useState 的函数式更新形式

```
function Counter() {
  const [count, setCount] = useState(0);

  useEffect(() => {
    const id = setInterval(() => {
      setCount(c => c + 1); // 在这不依赖于外部的 ‘count‘ 变量
    }, 1000);
    return () => clearInterval(id);
  }, []); // 我们的 effect 不适用组件作用域中的任何变量

  return <h1>{count}</h1>;
}
```

- 使用 useReducer
- 使用 useRef 保存，类似于类中的 this 功能

```
function Example(props) {
  // 把最新的 props 保存在一个 ref 中
  let latestProps = useRef(props);
  useEffect(() => {
    latestProps.current = props;
  });

  useEffect(() => {
    function tick() {
      // 在任何时候读取最新的 props
      console.log(latestProps.current);
    }

    const id = setInterval(tick, 1000);
    return () => clearInterval(id);
  }, []); // 这个 effect 从不会重新执行
}
```

✏ 如何实现 shouldComponentUpdate?

用 React.memo[18]包裹一个组件来对它的 props 进行浅比较，通过第二个参数指定一个自定义的比较函数来比较新旧 props。如果函数返回 true，就会跳过更新。

[18]React.memo 等效于 PureComponent，但它不比较 state，只比较 props。

```
const Button = React.memo((props) => {
  // 你的组件
});
```

✏ 如何惰性创建昂贵的对象?

- useState

```
function Table(props) {
  // createRows() 只会被调用一次
  const [rows, setRows] = useState(() => createRows(props.
     count));
  // ...
}
```

- useRef

```
function Image(props) {
  const ref = useRef(null);

  // IntersectionObserver 只会被惰性创建一次
  function getObserver() {
    let observer = ref.current;
    if (observer !== null) {
      return observer;
    }
    let newObserver = new IntersectionObserver(onIntersect
       );
    ref.current = newObserver;
    return newObserver;
  }

  // 当你需要时, 调用 getObserver()
  // ...
}
```

✏ 如何避免向下传递回调?

在大型的组件树中, 在组件树的每一层手动传递回调很麻烦, 推荐的替代方案是通过 context 用 useReducer 往下传一个 dispatch 函数。

```
const TodosDispatch = React.createContext(null);

function TodosApp() {
  // 提示: `dispatch` 不会在重新渲染之间变化
  const [todos, dispatch] = useReducer(todosReducer);
```

```
  return (
    <TodosDispatch.Provider value={dispatch}>
      <DeepTree todos={todos} />
    </TodosDispatch.Provider>
  );
}

function DeepChild(props) {
  // 如果我们想要执行一个 action，我们可以从 context 中获取
     dispatch。
  const dispatch = useContext(TodosDispatch);

  function handleClick() {
    dispatch({ type: 'add', text: 'hello' });
  }

  return (
    <button onClick={handleClick}>Add todo</button>
  );
}
```

3.7.4 底层原理

✏ React 是如何把对 Hook 的调用和组件联系起来的?

React 保持对当先渲染中的组件的追踪，Hook 规范使得 Hook 只会在 React 组件或自定义 Hook 中被调用。每个组件内部都有一个「记忆单元格」列表，被用来存储一些数据的 JavaScript 对象。当你用 useState() 调用一个 Hook 的时候，它会读取当前的单元格（或在首次渲染时将其初始化），然后把指针移动到下一个。这就是多个 useState() 调用会得到各自独立的本地 state 的原因。

4 FAQ

4.1 AJAX 及 APIs

- axios
- jQuery AJAX
- window.fetch

4.2 Babel、JSX 及构建过程

4.3 在组件中使用事件处理函数

✏ 为什么绑定是必要的?

在 JavaScript 中，以下两种写法是不等价的，bind 方法确保了第二种写法与第一种写

法相同。

```
// 第一种
obj.method();

// 第二种
var method = obj.method;
method();
```

✏ 怎样阻止函数被调用太快或者太多次?

- 节流: 节流阻止函数在给定时间窗口内被调不能超过一次

```
import throttle from 'lodash.throttle';

class LoadMoreButton extends React.Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
    // 节流“click”事件处理器每秒钟的只能调用一次
    this.handleClickThrottled = throttle(this.handleClick,
        1000);
  }

  componentWillUnmount() {
    this.handleClickThrottled.cancel();
  }

  handleClick() {
    this.props.loadMore();
  }

  render() {
    return <button onClick={this.handleClickThrottled}>
       Load More</button>;
  }
}
```

- 防抖: 防抖确保函数不会在上一次被调用之后一定量的时间内被执行，适用于必须进行一些昂贵的计算来响应快速派发的事件（比如鼠标滚动或键盘事件）

```
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);
  }

  componentWillUnmount() {
    this.emitChangeDebounced.cancel();
  }

  emitChange(value) {
    this.props.onChange(value);
  }

  handleChange(e) {
    // React pools events, so we read the value before
        debounce.
    // Alternately we could call 'event.persist()' and
        pass the entire event.
    // For more info see reactjs.org/docs/events.html#
        event-pooling
    this.emitChangeDebounced(e.target.value);
  }

  render() {
    return (
      <input
        type="text"
        onChange={this.handleChange}
        placeholder="Search..."
        defaultValue={this.props.value}
      />
    );
  }
}
```

- requestAnimationFrame 节流：一个函数被 requestAnimationFrame 放入队列后将会在下一帧触发。浏览器会努力确保每秒 60 帧（60fps）。然而，如果浏览器无法确保，那么自然会限制每秒的帧数。例如，某个设备可能只能处理每秒 30 帧，所以每秒只能得到 30 帧。

```
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.
    // 使用这个方法时只能获取帧中最后发布的值
    this.scheduleUpdate = rafSchedule(
      point => this.props.onScroll(point)
    );
  }

  componentWillUnmount() {
    // Cancel any pending updates since we're unmounting.
    this.scheduleUpdate.cancel();
  }

  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 });
  }

  render() {
    return (
      <div
        style={{ overflow: 'scroll' }}
        onScroll={this.handleScroll}
      >
        <img src="/my-huge-image.jpg" />
      </div>
    );
  }
}
```

4.4 组件状态

4.5 样式及 CSS

✏ 在 React 中如何做动画?

- React Transition Group
- React Motion

4.6 项目文件结构

✏ 如何组织 React 项目?

避免太多嵌套，混合使用以下方式：

- 按照功能或者路由来分组
- 按照文件类型来分组

4.7 Virual DOM 及内核

✏ 什么是虚拟 DOM?

虚拟 DOM 是由 JavaScript 库在浏览器 API 之上实现的一种模式，通过 React 的 API 声明视图状态，由 ReactDOM 负责真实 DOM 和视图状态的匹配。

✏ 影子 DOM（Shadow DOM）和虚拟 DOM（Virtual DOM）是相同的概念吗?

影子 DOM 是一种浏览器技术，主要被设计用来为 Web 组件中的变量和 CSS 提供封装。虚拟 DOM 是由 JavaScript 库在浏览器 API 之上实现的一种模式。