React

王知衡 无08 wangzhih20@mails.tsinghua.edu.cn

React是用于构建用户界面的JavaScript库，起源于Facebook的内部项目，用来架设 Instagram 的网站，并于 2013 年 5 月开源。

框架特点：

- 声明式设计：React 使创建交互式 UI 变得轻而易举。为你应用的每一个状态设计简洁的视图，当数据变动时 React能高效更新并渲染合适的组件。
- 组件化: 构建管理自身状态的封装组件，然后对其组合以构成复杂的 UI。、
- 高效：React通过对DOM的模拟，最大限度地减少与DOM的交互。
- 灵活：无论你现在使用什么技术栈，在无需重写现有代码的前提下，通过引入React来开发新功能。

使用 create-react-app 初始化项目

创建React应用

npm：

```
npx create-react-app react-tutorial --typescript
```

进入create-react-app创建的文件夹并安装typescript

npm：

```
cd react-tutorial
npm install typescript @types/node @types/react @types/react-dom @types/jest --
save-dev
```

yarn：

```
cd react-tutorial
yarn add typescript @types/node @types/react @types/react-dom @types/jest --dev
```

创建的文件夹有如下内容：

```
node_modules/
    ...
public/
    ...
src/
    App.tsx
    index.tsx
    ...
.gitignore
package.json
README.md
tsconfig.json
yarn.lock
```

我们的结构代码主要放在src文件夹内。

启动React

npm：

```
npm run start
```

yarn：

```
yarn start
```

在浏览器打开`localhost:3000`查看应用

JSX简介

考虑如下变量声明：

```
const element = <h1>Hello, world!</h1>;
```

这个有趣的标签语法既不是字符串也不是 HTML。它被称为 JSX，是一个 JavaScript 的语法扩展。JSX 可以生成 React “元素”。

在 JSX 中嵌入表达式

```
const name = 'Josh Perez';
const element = <h1>Hello, {name}</h1>;

ReactDOM.render(
  element,
  document.getElementById('root')
);
```

在 JSX 语法中，你可以在大括号内放置任何有效的 JavaScript 表达式。

JSX 也是一个表达式

在编译之后，JSX 表达式会被转为普通 JavaScript 函数调用，并且对其取值后得到 JavaScript 对象。

也就是说，你可以在 `if` 语句和 `for` 循环的代码块中使用 JSX，将 JSX 赋值给变量，把 JSX 当作参数传入，以及从函数中返回 JSX：

```
function getGreeting(user) {
  if (user) {
    return <h1>Hello, {formatName(user)}!</h1>;
  }
  return <h1>Hello, Stranger.</h1>;
}
```

JSX特定属性

你可以通过使用引号，来将属性值指定为字符串字面量：

```
const element = <div tabIndex="0"></div>;
```

也可以使用大括号，来在属性值中插入一个 JavaScript 表达式：

```
const element = <img src={user.avatarUrl}></img>;
```

在属性中嵌入 JavaScript 表达式时，不要在大括号外面加上引号。你应该仅使用引号（对于字符串值）或大括号（对于表达式）中的一个，对于同一属性不能同时使用这两种符号。

> 因为 JSX 语法上更接近 JavaScript 而不是 HTML，所以 React DOM 使用 `camelCase`（小驼峰命名）来定义属性的名称，而不使用 HTML 属性名称的命名约定。
>
> 例如，JSX 里的 `class` 变成了 `className`，而 `tabindex` 则变为 `tabIndex`。

JSX表示对象

Babel 会把 JSX 转译成一个名为 `React.createElement()` 函数调用。

以下两种示例代码完全等效：

```
const element = (
  <h1 className="greeting">
    Hello, world!
  </h1>
);
```

```
const element = React.createElement(
  'h1',
  {className: 'greeting'},
  'Hello, world!'
);
```

`React.createElement()` 会预先执行一些检查，以帮助你编写无错代码，但实际上它创建了一个这样的对象：

```
const element = {
  type: 'h1',
  props: {
    className: 'greeting',
    children: 'Hello, world!'
  }
};
```

这些对象被称为 “React 元素”。它们描述了你希望在屏幕上看到的内容。React 通过读取这些对象，然后使用它们来构建 DOM 以及保持随时更新。

元素渲染

元素是构成 React 应用的最小砖块，描述了你在屏幕上想看到的内容。

与浏览器的 DOM 元素不同，React 元素是创建开销极小的普通对象。React DOM 会负责更新 DOM 来与 React 元素保持一致。

将一个元素渲染为DOM

假设你的 HTML 文件某处有一个 `<div>`：

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

我们将其称为“根” DOM 节点，因为该节点内的所有内容都将由 React DOM 管理。

仅使用 React 构建的应用通常只有单一的根 DOM 节点。如果你在将 React 集成进一个已有应用，那么你可以在应用中包含任意多的独立根 DOM 节点。

想要将一个 React 元素渲染到根 DOM 节点中，只需把它们一起传入 `ReactDOM.render()`：

```
const element = <h1>Hello, world</h1>;
ReactDOM.render(element, document.getElementById('root'));
```

页面上会展示出 “Hello, world”。

更新已渲染的元素

React 元素是不可变对象。一旦被创建，你就无法更改它的子元素或者属性。一个元素就像电影的单帧：它代表了某个特定时刻的 UI。

根据我们已有的知识，更新 UI 唯一的方式是创建一个全新的元素，并将其传入 `ReactDOM.render()`。

考虑一个计时器的例子：

```
function tick() {
  const element = (
    <div>
      <h1>Hello, world!</h1>
      <h2>It is {new Date().toLocaleTimeString()}.</h2>
    </div>
  );
  ReactDOM.render(element, document.getElementById('root'));
}

setInterval(tick, 1000);
```

这个例子会在 `setInterval()` 回调函数，每秒都调用 `ReactDOM.render()`。也就是说，它每秒都渲染一个新的React元素，并将其传入 `ReactDOM.render()` 中。

React 只更新它需要更新的部分

React DOM 会将元素和它的子元素与它们之前的状态进行比较，并只会进行必要的更新来使 DOM 达到预期的状态。

在上面这个例子中，尽管每一秒我们都会新建一个描述整个 UI 树的元素，React DOM 只会更新实际改变了的内容，也就是例子中的文本节点。

组件 & Props

组件，从概念上类似于 JavaScript 函数。它接受任意的入参（即 “props”），并返回用于描述页面展示内容的 React 元素。

组件允许你将 UI 拆分为独立可复用的代码片段，并对每个片段进行独立构思。

函数组件与 class 组件

定义组件最简单的方式就是编写 JavaScript 函数：

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

该函数是一个有效的 React 组件，因为它接收唯一带有数据的 “props”（代表属性）对象与并返回一个 React 元素。这类组件被称为“函数组件”，因为它本质上就是 JavaScript 函数。

你同时还可以使用 ES6 的 `class` 来定义组件：

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

渲染组件

之前，我们遇到的 React 元素都只是 DOM 标签：

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

不过，React 元素也可以是用户自定义的组件：

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

当 React 元素为用户自定义组件时，它会将 JSX 所接收的属性（attributes）以及子组件（children）转换为单个对象传递给组件，这个对象被称之为 “props”。

例如，这段代码会在页面上渲染 “Hello, Sara”：

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

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

> **注意：** 组件名称必须以大写字母开头。
>
> React 会将以小写字母开头的组件视为原生 DOM 标签。例如，`<div />` 代表 HTML 的 div 标签，而 `<Welcome />` 则代表一个组件，并且需在作用域内使用 `Welcome`。

组合组件

组件可以在其输出中引用其他组件。这就可以让我们用同一组件来抽象出任意层次的细节。按钮，表单，对话框，甚至整个屏幕的内容：在 React 应用程序中，这些通常都会以组件的形式表示。

例如，我们可以创建一个可以多次渲染 `Welcome` 组件的 `App` 组件：

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

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

ReactDOM.render(
  <App />,
  document.getElementById('root')
);
```

通常来说，每个新的 React 应用程序的顶层组件都是 `App` 组件。但是，如果你将 React 集成到现有的应用程序中，你可能需要使用像 `Button` 这样的小组件，并自下而上地将这类组件逐步应用到视图层的每一处。

提取组件

将组件拆分为更小的组件。

例如，参考如下 `Comment` 组件：

```
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>
  );
```

```
}
```

该组件用于描述一个社交媒体网站上的评论功能，它接收 `author`（对象），`text`（字符串）以及 `date`（日期）作为 props。

该组件由于嵌套的关系，变得难以维护，且很难复用它的各个部分。因此，让我们从中提取一些组件出来。

首先，我们将提取 `Avatar` 组件：

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

`Avatar` 不需知道它在 `Comment` 组件内部是如何渲染的。因此，我们给它的 props 起了一个更通用的名字：`user`，而不是 `author`。

> 建议从组件自身的角度命名 props，而不是依赖于调用组件的上下文命名。

我们现在针对 `Comment` 做些微小调整：

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

接下来，我们将提取 `UserInfo` 组件，该组件在用户名旁渲染 `Avatar` 组件：

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

进一步简化 `Comment` 组件：

```
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>
  );
}
```

Props的只读性

组件无论是使用函数声明还是通过 class 声明，都决不能修改自身的 props。

所有 React 组件都必须像纯函数一样保护它们的 props 不被更改。

当然，应用程序的 UI 是动态的，并会伴随着时间的推移而变化。后面我们将介绍一种新的概念，称之为“state”。在不违反上述规则的情况下，state 允许 React 组件随用户操作、网络响应或者其他变化而动态更改输出内容。

State & 声明周期

在本小节我们将学习如何封装真正可复用的 `Clock` 组件。它将设置自己的计时器并每秒更新一次。

理想情况下，我们希望只编写一次代码，便可以让 `Clock` 组件自我更新：

```
ReactDOM.render(
  <Clock />,
  document.getElementById('root')
);
```

我们需要在 `Clock` 组件中添加“state”来实现这个功能。

State 与 props 类似，但是 state 是私有的，并且完全受控于当前组件。

将函数组件转换成 class 组件

通过以下五步将 `Clock` 的函数组件转成 class 组件：

1. 创建一个同名的 ES6 class，并且继承于 `React.Component`。
2. 添加一个空的 `render()` 方法。
3. 将函数体移动到 `render()` 方法之中。
4. 在 `render()` 方法中使用 `this.props` 替换 `props`。
5. 删除剩余的空函数声明。

```
class Clock extends React.Component {
  render() {
    return (
      <div>
        <h1>Hello, world!</h1>
        <h2>It is {this.props.date.toLocaleTimeString()}.</h2>
      </div>
    );
  }
}
```

现在 `Clock` 组件被定义为 class，而不是函数。

每次组件更新时 `render` 方法都会被调用，但只要在相同的 DOM 节点中渲染 `<Clock />` ，就仅有一个 `Clock` 组件的 class 实例被创建使用。这就使得我们可以使用如 state 或生命周期方法等很多其他特性。

向 class 组件中添加局部的 state

我们通过以下三步将 `date` 从 props 移动到 state 中：

1. 把 `render()` 方法中的 `this.props.date` 替换成 `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. 添加一个 class 构造函数，然后在该函数中为 `this.state` 赋初值：

```
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>
    );
  }
}
```

通过以下方式将 `props` 传递到父类的构造函数中：

```
  constructor(props) {
    super(props);
    this.state = {date: new Date()};
  }
```

Class 组件应该始终使用 `props` 参数来调用父类的构造函数。

3. 移除 `<Clock />` 元素中的 `date` 属性：

```
ReactDOM.render(
  <Clock />,
  document.getElementById('root')
);
```

我们之后会将计时器相关的代码添加到组件中。

代码如下：

```
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>
    );
  }
}

ReactDOM.render(
  <Clock />,
  document.getElementById('root')
);
```

接下来，我们会设置 `Clock` 的计时器并每秒更新它。

将生命周期方法添加到 Class 中

在具有许多组件的应用程序中，当组件被销毁时释放所占用的资源是非常重要的。

当 `Clock` 组件第一次被渲染到 DOM 中的时候，就为其设置一个计时器。这在 React 中被称为“挂载（mount）”。

同时，当 DOM 中 `Clock` 组件被删除的时候，应该清除计时器。这在 React 中被称为“卸载（unmount）”。

我们可以为 class 组件声明一些特殊的方法，当组件挂载或卸载时就会去执行这些方法：

```
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>
    );
  }
}
```

这些方法叫做“生命周期方法”。

`componentDidMount()` 方法会在组件已经被渲染到 DOM 中后运行，所以，最好在这里设置计时器：

```
  componentDidMount() {
    this.timerID = setInterval(
        () => this.tick(),
        1000
    );
  }
```

接下来把计时器的 ID 保存在 `this` 之中（`this.timerID`）。

尽管 `this.props` 和 `this.state` 是 React 本身设置的，且都拥有特殊的含义，但是其实你可以向 class 中随意添加不参与数据流（比如计时器 ID）的额外字段。

我们会在 `componentWillUnmount()` 生命周期方法中清除计时器：

```
  componentWillUnmount() {
    clearInterval(this.timerID);
  }
```

最后，我们会实现一个叫 `tick()` 的方法，`Clock` 组件每秒都会调用它。

使用 `this.setState()` 来时刻更新组件 state：

```
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>
    );
  }
}

ReactDOM.render(
  <Clock />,
  document.getElementById('root')
);
```

正确地使用State

关于 `setState()` 你应该了解三件事：

不要直接修改State

例如，此代码不会重新渲染组件：

```
// Wrong
this.state.comment = 'Hello';
```

而是应该使用 `setState()`：

```
// Correct
this.setState({comment: 'Hello'});
```

构造函数是唯一可以给 `this.state` 赋值的地方。

State的更新可能是异步的

出于性能考虑，React 可能会把多个 `setState()` 调用合并成一个调用。

因为 `this.props` 和 `this.state` 可能会异步更新，所以你不要依赖他们的值来更新下一个状态。

例如，此代码可能会无法更新计数器：

```
// Wrong
this.setState({
  counter: this.state.counter + this.props.increment,
});
```

要解决这个问题，可以让 `setState()` 接收一个函数而不是一个对象。这个函数用上一个 state 作为第一个参数，将此次更新被应用时的 props 做为第二个参数：

```
// Correct
this.setState((state, props) => ({
  counter: state.counter + props.increment
}));
```

State的更新会被合并

当你调用 `setState()` 的时候，React 会把你提供的对象合并到当前的 state。

例如，你的 state 包含几个独立的变量：

```
  constructor(props) {
    super(props);
    this.state = {
      posts: [],
      comments: []
    };
  }
```

然后你可以分别调用 `setState()` 来单独地更新它们：

```
  componentDidMount() {
    fetchPosts().then(response => {
      this.setState({
        posts: response.posts
      });
    });

    fetchComments().then(response => {
      this.setState({
        comments: response.comments
      });
    });
  }
```

这里的合并是浅合并，所以 `this.setState({comments})` 完整保留了 `this.state.posts`， 但是完全替换了 `this.state.comments`。

数据是向下流动的

不管是父组件或是子组件都无法知道某个组件是有状态的还是无状态的，并且它们也并不关心它是函数组件还是 class 组件。

这就是为什么称 state 为局部的或是封装的的原因。除了拥有并设置了它的组件，其他组件都无法访问。

组件可以选择把它的 state 作为 props 向下传递到它的子组件中：

```
<h2>It is {this.state.date.toLocaleTimeString()}.</h2>
```

这对于自定义组件同样适用：

```
<FormattedDate date={this.state.date} />
```

`FormattedDate` 组件会在其 props 中接收参数 `date`，但是组件本身无法知道它是来自于 `Clock` 的 state，或是 `Clock` 的 props，还是手动输入的：

```
function FormattedDate(props) {
  return <h2>It is {props.date.toLocaleTimeString()}.</h2>;
}
```

这通常会被叫做“自上而下”或是“单向”的数据流。任何的 state 总是所属于特定的组件，而且从该 state 派生的任何数据或 UI 只能影响树中“低于”它们的组件。

如果你把一个以组件构成的树想象成一个 props 的数据瀑布的话，那么每一个组件的 state 就像是在任意一点上给瀑布增加额外的水源，但是它只能向下流动。

事件处理

React 元素的事件处理和 DOM 元素的很相似，但是有一点语法上的不同：

- React 事件的命名采用小驼峰式（camelCase），而不是纯小写。
- 使用 JSX 语法时你需要传入一个函数作为事件处理函数，而不是一个字符串。

例如，传统的 HTML：

```
<button onclick="activateLasers()">
    Activate Lasers
</button>
```

在 React 中略微不同：

```
<button onClick={activateLasers}>
    Activate Lasers
</button>
```

在 React 中另一个不同点是你不能通过返回 `false` 的方式阻止默认行为。你必须显式的使用 `preventDefault` 。例如，传统的 HTML 中阻止链接默认打开一个新页面，你可以这样写：

```
<a href="#" onclick="console.log('The link was clicked.'); return false">
  Click me
</a>
```

在 React 中，可能是这样的：

```
function ActionLink() {
  function handleClick(e) {
      e.preventDefault();
      console.log('The link was clicked.');
  }
  return (
    <a href="#" onClick={handleClick}>
        Click me
    </a>
  );
}
```

使用 React 时，你一般不需要使用 `addEventListener` 为已创建的 DOM 元素添加监听器。事实上，你只需要在该元素初始渲染的时候添加监听器即可。

当你使用 ES6 class语法定义一个组件的时候，通常的做法是将事件处理函数声明为 class 中的方法。例如，下面的 `Toggle` 组件会渲染一个让用户切换开关状态的按钮：

```
class Toggle extends React.Component {
  constructor(props) {
    super(props);
    this.state = {isToggleOn: true};

    // 为了在回调中使用 `this`，这个绑定是必不可少的
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
      this.setState(state => ({
          isToggleOn: !state.isToggleOn
      }));
  }
  render() {
    return (
      <button onClick={this.handleClick}>
        {this.state.isToggleOn ? 'ON' : 'OFF'}
      </button>
    );
  }
}

ReactDOM.render(
  <Toggle />,
  document.getElementById('root')
);
```

你必须谨慎对待 JSX 回调函数中的 `this`，在 JavaScript 中，class 的方法默认不会绑定 `this`。如果你忘记绑定 `this.handleClick` 并把它传入了 `onClick`，当你调用这个函数的时候 `this` 的值为 `undefined`。

这并不是 React 特有的行为；这其实与 JavaScript 函数工作原理有关。通常情况下，如果你没有在方法后面添加 `()`，例如 `onClick={this.handleClick}`，你应该为这个方法绑定 `this`。

如果觉得使用 `bind` 很麻烦，这里有两种方式可以解决。如果你正在使用实验性的 public class fields 语法，你可以使用 class fields 正确的绑定回调函数：

```
class LoggingButton extends React.Component {
  // 此语法确保 `handleClick` 内的 `this` 已被绑定。
  // 注意: 这是 *实验性* 语法。用箭头函数定义。
  handleClick = () => {
    console.log('this is:', this);
  }
  render() {
    return (
      <button onClick={this.handleClick}>
        Click me
      </button>
    );
  }
}
```

如果你没有使用 class fields 语法，你可以在回调中使用箭头函数：

```
class LoggingButton extends React.Component {
  handleClick() {
    console.log('this is:', this);
  }

  render() {
    // 此语法确保 `handleClick` 内的 `this` 已被绑定。
    return (
      <button onClick={() => this.handleClick()}>
        Click me
      </button>
    );
  }
}
```

此语法问题在于每次渲染 `LoggingButton` 时都会创建不同的回调函数。在大多数情况下，这没什么问题，但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。我们通常建议在构造器中绑定或使用 class fields 语法来避免这类性能问题。

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。例如，若 `id` 是你要删除那一行的 ID，以下两种方式都可以向事件处理函数传递参数：

```
<button onClick={(e) => this.deleteRow(id, e)}>Delete Row</button>
<button onClick={this.deleteRow.bind(this, id)}>Delete Row</button>
```

上述两种方式是等价的，分别通过箭头函数和 `Function.prototype.bind` 来实现。

在这两种情况下，React 的事件对象 `e` 会被作为第二个参数传递。如果通过箭头函数的方式，事件对象必须显式的进行传递，而通过 `bind` 的方式，事件对象以及更多的参数将会被隐式的进行传递。

> `bind()` 方法创建一个新的函数，在 `bind()` 被调用时，这个新函数的 `this` 被指定为 `bind()` 的第一个参数，而其余参数将作为新函数的参数，供调用时使用。

条件渲染

在 React 中，你可以创建不同的组件来封装各种你需要的行为。然后，依据应用的不同状态，你可以只渲染对应状态下的部分内容。

React 中的条件渲染和 JavaScript 中的一样，使用 JavaScript 运算符 `if` 或者条件运算符去创建元素来表现当前的状态，然后让 React 根据它们来更新 UI。

观察这两个组件:

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

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

再创建一个 `Greeting` 组件，它会根据用户是否登录来决定显示上面的哪一个组件。

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

ReactDOM.render(
  // Try changing to isLoggedIn={true}:
  <Greeting isLoggedIn={false} />,
  document.getElementById('root')
);
```

元素变量

你可以使用变量来储存元素。 它可以帮助你有条件地渲染组件的一部分，而其他的渲染部分并不会因此而改变。

观察这两个组件，它们分别代表了注销和登录按钮：

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

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

在下面的示例中，我们将创建一个名叫 `LoginControl` 的有状态的组件。

它将根据当前的状态来渲染 `<LoginButton />` 或者 `<LogoutButton />`。同时它还会渲染上一个示例中的 `<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>
    );
  }
}

ReactDOM.render(
  <LoginControl />,
  document.getElementById('root')
);
```

与运算符&&

```
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'];
ReactDOM.render(
  <Mailbox unreadMessages={messages} />,
  document.getElementById('root')
);
```

之所以能这样做，是因为在 JavaScript 中，`true && expression` 总是会返回 `expression`，而 `false && expression` 总是会返回 `false`。

因此，如果条件是 `true`，`&&` 右侧的元素就会被渲染，如果是 `false`，React 会忽略并跳过它。

三目运算符

另一种内联条件渲染的方法是使用 JavaScript 中的三目运算符 `condition ? true : false`。

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

组止组件渲染

在极少数情况下，你可能希望能隐藏组件，即使它已经被其他组件渲染。若要完成此操作，你可以让 `render` 方法直接返回 `null`，而不进行任何渲染。

列表 & Key

首先，让我们看下在 Javascript 中如何转化列表。

如下代码，我们使用 `map()` 函数让数组中的每一项变双倍，然后我们得到了一个新的列表 `doubled` 并打印出来：

```
const numbers = [1, 2, 3, 4, 5];
const doubled = numbers.map((number) => number * 2);console.log(doubled);
```

代码打印出 `[2, 4, 6, 8, 10]`。

在 React 中，把数组转化为元素列表的过程是相似的。

渲染多个组件

你可以通过使用 `{}` 在 JSX 内构建一个元素集合。

下面，我们使用 Javascript 中的 `map()` 方法来遍历 `numbers` 数组。将数组中的每个元素变成 `<li>` 标签，最后我们将得到的数组赋值给 `listItems`：

```
const numbers = [1, 2, 3, 4, 5];
const listItems = numbers.map((number) =>  <li>{number}</li>);
```

我们把整个 `listItems` 插入到 `<ul>` 元素中，然后渲染进 DOM：

```
ReactDOM.render(
  <ul>{listItems}</ul>,  document.getElementById('root')
);
```

基础列表组件

通常你需要在一个组件中渲染列表。

我们可以把前面的例子重构成一个组件，这个组件接收 `numbers` 数组作为参数并输出一个元素列表。

```
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];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('root')
);
```

当我们运行这段代码，将会看到一个警告 `a key should be provided for list items`，意思是当你创建一个元素时，必须包括一个特殊的 `key` 属性。

让我们来给每个列表元素分配一个 `key` 属性来解决上面的那个警告：

```
function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    <li key={number.toString()}>
      {number}
    </li>
  );
  return (
    <ul>{listItems}</ul>
  );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('root')
);
```

key

key 帮助 React 识别哪些元素改变了，比如被添加或删除。因此你应当给数组中的每一个元素赋予一个确定的标识。

一个元素的 key 最好是这个元素在列表中拥有的一个独一无二的字符串。通常，我们使用数据中的 id 来作为元素的 key：

```
const todoItems = todos.map((todo) =>
  <li key={todo.id}>
      {todo.text}
  </li>
);
```

当元素没有确定 id 的时候，万不得已你可以使用元素索引 index 作为 key：

```
const todoItems = todos.map((todo, index) =>
  // Only do this if items have no stable IDs
  <li key={index}>
    {todo.text}
  </li>
);
```

如果列表项目的顺序可能会变化，我们不建议使用索引来用作 key 值，因为这样做会导致性能变差，还可能引起组件状态的问题。如果你选择不指定显式的 key 值，那么 React 将默认使用索引用作为列表项目的 key 值。

可以参考这篇文章了解为什么key是必须的

用 key 提取组件

元素的 key 只有放在就近的数组上下文中才有意义。

比方说，如果你提取出一个 `ListItem` 组件，你应该把 key 保留在数组中的这个 `<ListItem />` 元素上，而不是放在 `ListItem` 组件中的 `<li>` 元素上。

例子：不正确的使用 key 的方式

```
function ListItem(props) {
  const value = props.value;
  return (
    // 错误！你不需要在这里指定 key：
    <li key={value.toString()}>
      {value}
    </li>
  );
}

function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    // 错误！元素的 key 应该在这里指定：
    <ListItem value={number} />
  );
  return (
    <ul>
```

```
      {listItems}
    </ul>
  );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('root')
);
```

例子：正确的使用 key 的方式

```
function ListItem(props) {
  // 正确！这里不需要指定 key：
  return <li>{props.value}</li>;
}

function NumberList(props) {
  const numbers = props.numbers;
  const listItems = numbers.map((number) =>
    // 正确！key 应该在数组的上下文中被指定
    <ListItem key={number.toString()}              value={number} />

  );
  return (
    <ul>
      {listItems}
    </ul>
  );
}

const numbers = [1, 2, 3, 4, 5];
ReactDOM.render(
  <NumberList numbers={numbers} />,
  document.getElementById('root')
);
```

一个好的经验法则是：在 `map()` 方法中的元素需要设置 key 属性。

key 只是在兄弟节点之间必须唯一

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。当我们生成两个不同的数组时，我们可以使用相同的 key 值。

在 JSX 中嵌入 map()

JSX 允许在大括号中嵌入任何表达式，所以我们可以内联 `map()` 返回的结果：

```
function NumberList(props) {
  const numbers = props.numbers;
  return (
    <ul>
      {numbers.map((number) =>
        <ListItem key={number.toString()}
                  value={number} />
      )}
    </ul>
  );
}
```

Hook

Hook简介

Hook 是 React 16.8 的新增特性。它可以让你在不编写 class 的情况下使用 state 以及其他的 React 特性。

State Hook

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

function Example() {
  // 声明一个新的叫做 “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>
  );
}
```

等价的class示例

```
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 初始值为 `{ count: 0 }`，当用户点击按钮后，我们通过调用 `this.setState()` 来增加 `state.count`。

那么，什么是Hook?

Hook 是一些可以让你在函数组件里“钩入” React state 及生命周期等特性的函数。Hook 不能在 class 组件中使用 —— 这使得你不使用 class 也能使用 React。

什么时候我会用 Hook?

如果你在编写函数组件并意识到需要向其添加一些 state，以前的做法是必须将其转化为 class。现在你可以在现有的函数组件中使用 Hook。

声明State变量

在函数组件中，我们没有 `this`，所以我们不能分配或读取 `this.state`。我们直接在组件中调用 `useState` Hook：

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

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

`调用 useState` **方法的时候做了什么?** 它定义一个 “state 变量”。我们的变量叫 `count`， 但是我们可以叫他任何名字，比如 `banana`。这是一种在函数调用时保存变量的方式 —— `useState` 是一种新方法，它与 class 里面的 `this.state` 提供的功能完全相同。一般来说，在函数退出后变量就会”消失”，而 state 中的变量会被 React 保留。

`useState` **需要哪些参数?** `useState()` 方法里面唯一的参数就是初始 state。不同于 class 的是，我们可以按照需要使用数字或字符串对其进行赋值，而不一定是对象。在示例中，只需使用数字来记录用户点击次数，所以我们传了 `0` 作为变量的初始 state。（如果我们想要在 state 中存储两个不同的变量，只需调用 `useState()` 两次即可。）

`useState` **方法的返回值是什么?** 返回值为：当前 state 以及更新 state 的函数。这就是我们写 `const [count, setCount] = useState()` 的原因。这与 class 里面 `this.state.count` 和 `this.setState` 类似，唯一区别就是你需要成对的获取它们（数组解构）。

读取State

在函数中，我们可以直接用 `count` 读取State:

```
  <p>You clicked {count} times</p>
```

更新State

在函数中，我们已经有了 `setCount` 和 `count` 变量，所以我们不需要 `this`：

```
  <button onClick={() => setCount(count + 1)}>
    Click me
  </button>
```

Effect Hook

Effect Hook 可以让你在函数组件中执行副作用操作

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

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

  // Similar to componentDidMount and componentDidUpdate:
  useEffect(() => {
    // Update the document title using the browser API
    document.title = `You clicked ${count} times`;
  });

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

在上一节的计数器示例的基础上，我们为计数器增加了一个小功能：将 document 的 title 设置为包含了点击次数的消息。

数据获取，设置订阅以及手动更改 React 组件中的 DOM 都属于副作用。不管你知不知道这些操作，或是“副作用”这个名字，应该都在组件中使用过它们。

> 你可以把 `useEffect` Hook 看做 `componentDidMount`，`componentDidUpdate` 和 `componentWillUnmount` 这三个函数的组合。

在 React 组件中有两种常见副作用操作：需要清除的和不需要清除的。

无需清除的effect

有时候，我们只想**在 React 更新 DOM 之后运行一些额外的代码。**比如发送网络请求，手动变更 DOM，记录日志，这些都是常见的无需清除的操作。因为我们在执行完这些操作之后，就可以忽略他们了。

`useEffect` **做了什么？** 通过使用这个 Hook，你可以告诉 React 组件需要在渲染后执行某些操作。React 会保存你传递的函数（我们将它称之为 “effect”），并且在执行 DOM 更新之后调用它。在这个 effect 中，我们设置了 document 的 title 属性，不过我们也可以执行数据获取或调用其他命令式的 API。

为什么在组件内部调用 `useEffect`**？** 将 `useEffect` 放在组件内部让我们可以在 effect 中直接访问 `count` state 变量（或其他 props）。我们不需要特殊的 API 来读取它 —— 它已经保存在函数作用域中。Hook 使用了 JavaScript 的闭包机制，而不用在 JavaScript 已经提供了解决方案的情况下，还引入特定的 React API。

`useEffect` **会在每次渲染后都执行吗？** 是的，默认情况下，它在第一次渲染之后和每次更新之后都会执行。你可能会更容易接受 effect 发生在“渲染之后”这种概念，不用再去考虑“挂载”还是“更新”。React 保证了每次运行 effect 的同时，DOM 都已经更新完毕。

> 某种意义上讲，effect 更像是渲染结果的一部分 —— 每个 effect “属于”一次特定的渲染。

需要清除的effect

之前，我们研究了如何使用不需要清除的副作用，还有一些副作用是需要清除的。例如**订阅外部数据源**。这种情况下，清除工作是非常重要的，可以防止引起内存泄露！现在让我们来比较一下如何用 Class 和 Hook 来实现。

使用 Class 的示例

在 React class 中，你通常会在 `componentDidMount` 中设置订阅，并在 `componentWillUnmount` 中清除它。例如，假设我们有一个 `ChatAPI` 模块，它允许我们订阅好友的在线状态。以下是我们如何使用 class 订阅和显示该状态：

```
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';
  }
}
```

使用Hook的示例

如何使用 Hook 编写这个组件。

你可能认为需要单独的 effect 来执行清除操作。但由于添加和删除订阅的代码的紧密性，所以 `useEffect` 的设计是在同一个地方执行。如果你的 effect 返回一个函数，React 将会在执行清除操作时调用它：

```
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);
    // 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';
}
```

为什么要在 effect 中返回一个函数? 这是 effect 可选的清除机制。每个 effect 都可以返回一个清除函数。如此可以将添加和移除订阅的逻辑放在一起。它们都属于 effect 的一部分。

React 何时清除 effect? React 会在组件卸载的时候执行清除操作。正如之前学到的，effect 在每次渲染的时候都会执行。这就是为什么 React *会*在执行当前 effect 之前对上一个 effect 进行清除。

> 并不是必须为 effect 中返回的函数命名。这里我们将其命名为 `cleanup` 是为了表明此函数的目的，但其实也可以返回一个箭头函数或者给起一个别的名字。

就像你可以使用多个 *state* 的 Hook 一样，你也可以使用多个 effect。这会将不相关逻辑分离到不同的 effect 中。

通过跳过 Effect 进行性能优化

在某些情况下，每次渲染后都执行清理或者执行 effect 可能会导致性能问题。

如果某些特定值在两次重渲染之间没有发生变化，你可以通知 React **跳过**对 effect 的调用，只要传递数组作为 `useEffect` 的第二个可选参数即可：

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

如果想执行只运行一次的 effect（仅在组件挂载和卸载时执行），可以传递一个空数组（`[]`）作为第二个参数。这就告诉 React 你的 effect 不依赖于 props 或 state 中的任何值，所以它永远都不需要重复执行。这并不属于特殊情况 —— 它依然遵循依赖数组的工作方式。

Hook规则

- **只在最顶层使用 Hook**：不要在循环，条件或嵌套函数中调用 Hook
- **只在 React 函数中调用 Hook**：不要在普通的 JavaScript 函数中调用 Hook

Ant Design组件库

Ant Design - 一套企业级 UI 设计语言和 React 组件库

导入antd

```
$ yarn add antd
```

具体使用方法见录屏