React 基础

本章将指导你了解 React 的基础知识。由于静态组件会有些枯燥，所以这章的内容会包含组件的状态与交互。此外，你将学习使用不同方式声明组件以及如何保持组件的可组合性和可复用性。准备好创造你自己的组件。

组件内部状态

组件内部状态也被称为局部状态，允许你保存、修改和删除存储在组件内部的属性。使用 ES6 类组件可以在构造函数中初始化组件的状态。 构造函数只会在组件初始化时调用一次。

让我们引入类构造函数。

{title="src/App.js",lang=javascript}

```
class App extends Component {

# leanpub-start-insert
  constructor(props) {
    super(props);
  }
# leanpub-end-insert

  ...

}
```

当你使用 ES6 编写的组件有一个构造函数时，它需要强制地调用 `super();` 方法，因为这个 App 组件是 `Component` 的子类。因此在你的APP组件要声明 `extends Component` 。 你会在后续内容中更详细地了解使用 ES6 编写的组件。

你也可以调用 `super(props);` ，它会在你的构造函数中设置 `this.props` 以供在构造函数中访问它们。 否则当在构造函数中访问 `this.props` ，会得到 `undefined` 。稍后你将了解更多关于 React 组件的 props。

现在，在你的示例中，组件中的初始状态应该是一个列表。

{title="src/App.js",lang=javascript}

```
const list = [
  {
    title: 'React',
    url: 'https://facebook.github.io/react/',
    author: 'Jordan Walke',
    num_comments: 3,
    points: 4,
    objectID: 0,
  },
  ...
];

class App extends Component {

  constructor(props) {
    super(props);

# leanpub-start-insert
    this.state = {
      list: list,
    };
# leanpub-end-insert
  }

  ...

}
```

state 通过使用 `this` 绑定在类上。因此，你可以在整个组件中访问到 state。例如它可以用在 `render()` 方法中。此前你已经在 `render()` 方法中映射一个在组件外定义静态列表。现在你可以在组件中使用 state 里的 list 了。

{title="src/App.js",lang=javascript}

```
class App extends Component {

  ...

  render() {
    return (
      <div className="App">
# leanpub-start-insert
        {this.state.list.map(item =>
```

```
# leanpub-end-insert
          <div key={item.objectID}>
            <span>
              <a href={item.url}>{item.title}</a>
            </span>
            <span>{item.author}</span>
            <span>{item.num_comments}</span>
            <span>{item.points}</span>
          </div>
        )}
      </div>
    );
  }
}
```

现在 list 是组件的一部分。它驻留在组件的 state 中。你可以从 list 中添加、修改或者删除列表项。每次你修改组件的内部状态，组件的 `render` 方法会再次运行。这样你可以简单地修改组件内部状态，确保组件重新渲染并且展示从内部状态获取到的正确数据。

但是需要注意，不要直接修改 state。你必须使用 `setState()` 方法来修改它。你将在接下来的章节了解到它。

练习：

- 练习使用 state
- 在构造函数中定义更多的初始化 state
- 在 `render()` 函数中访问使用 state
- 阅读更多关于 ES6类构造函数

ES6 对象初始化

在 ES6 中，你可以通过简写属性更加简洁地初始化对象。想象下面的对象初始化：

{title="Code Playground",lang="javascript"}

```
const name = 'Robin';

const user = {
  name: name,
};
```

当你的对象中的属性名与变量名相同时，您可以执行以下的操作：

{title="Code Playground",lang="javascript"}

```
const name = 'Robin';

const user = {
  name,
};
```

在应用程序中，你也可以这样做。列表变量名和状态属性名称共享同一名称。

{title="Code Playground",lang="javascript"}

```
// ES5
this.state = {
  list: list,
};

// ES6
this.state = {
  list,
};
```

另一个简洁的辅助办法是简写方法名。在 ES6 中，你能更简洁地初始化一个对象的方法。

{title="Code Playground",lang="javascript"}

```
// ES5
var userService = {
  getUserName: function (user) {
    return user.firstname + ' ' + user.lastname;
  },
};

// ES6
const userService = {
  getUserName(user) {
    return user.firstname + ' ' + user.lastname;
  },
};
```

最后值得一提的是你可以在 ES6 中使用计算属性名。

{title="Code Playground",lang="javascript"}

```
// ES5
var user = {
  name: 'Robin',
};

// ES6
const key = 'name';
const user = {
  [key]: 'Robin',
};
```

或许你目前还觉得计算属性名没有意义。为什么需要他们呢？在后续的章节中，当你为一个对象动态地根据 key 分配值时便会涉及到。在 JavaScript 中生成查找表是很简单的。

练习：

- ES6 对象初始化练习
- 阅读更多关于 ES6 对象初始化

单向数据流

现在你的组件中有一些内部的 state。但是你还没有操纵它们，因此 state 是静态的。一个练习 state 操作好方法是增加一些组件的交互。

让我们为列表中的每一项增加一个按钮。按钮的文案为 “Dismiss” ，意味着将从列表中删除该项。这个按钮在你希望保留未读列表和删除不感兴趣的项时会非常有用。

{title="src/App.js",lang=javascript}

```
class App extends Component {

  ...

  render() {
    return (
      <div className="App">
        {this.state.list.map(item =>
```

```
          <div key={item.objectID}>
            <span>
              <a href={item.url}>{item.title}</a>
            </span>
            <span>{item.author}</span>
            <span>{item.num_comments}</span>
            <span>{item.points}</span>
# leanpub-start-insert
            <span>
              <button
                onClick={() => this.onDismiss(item.objectID)}
                type="button"
              >
                Dismiss
              </button>
            </span>
# leanpub-end-insert
          </div>
        )}
      </div>
    );
  }
}
```

这个类方法 `onDismiss()` 还没有被定义， 我们稍后再来做这件事。目前先把重点放在按钮元素的 `onClick` 事件处理器上。正如你看见的， `onDismiss()` 方法被另外一个函数包裹在 `onClick` 事件处理器中，它是一个箭头函数。这样你可以拿到 `item` 对象中的 `objectID` 属性来确定那一项会被删除掉。另外一种方法是在 `onClick` 处理器之外定义函数，并只将已定义的函数传到处理器。在后续的章节中会解释更多关于元素处理器的细节。

你有没有注意到按钮元素是多行代码的？元素中一行有多个属性会看起来比较混乱。所以这个按钮使用多行格式来书写以保持它的可读性。这虽然不是强制的，但这是我的极力推荐的代码风格。

现在你需要来完成 `onDismiss()` 的功能，它通过 id 来标示那一项需要被删除。此函数绑定到类，因此成为类方法。这就是为什么你访问它使用 `this.onDismiss()` 而不是 `onDismiss()`。 `this` 对象是类的实例，为了将 `onDismiss()` 定义为类方法，你需要在构造函数中绑定它。绑定稍后将在另一章中详细解释。

{title="src/App.js",lang=javascript}

```
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      list,
    };

# leanpub-start-insert
    this.onDismiss = this.onDismiss.bind(this);
# leanpub-end-insert
  }

  render() {
    ...
  }
}
```

下一步，你需要在类中定义它的功能和业务逻辑。类方法可以用以下方式定义。

{title="src/App.js",lang=javascript}

```
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      list,
    };

    this.onDismiss = this.onDismiss.bind(this);
  }

# leanpub-start-insert
  onDismiss(id) {
    ...
  }
# leanpub-end-insert

  render() {
    ...
  }
}
```

现在你可以定义方法内部的功能。总的来说你希望从列表中删除由 id 标识的项，并且保存更新后的列表到 state 中。随后这个更新后列表被使用到再次运行的 `render()` 方法中并渲染，最后这个被删除项就不再显示了。

你可以通过 JavaScript 内置的 filter 方法来删除列表中的一项。fitler 方法以一个函数作为输入。这个函数可以访问列表中的每一项，因为它会遍历整个列表。通过这种方式，你可以基于过滤条件来判断列表的每一项。如果该项判断结果为 true，则该项保留在列表中。否则将从列表中过滤掉。另外，好的一点是这个方法会返回一个新的列表而不是改变旧列表。它遵循了 React 中不可变数据的约定。

{title="src/App.js",lang=javascript}

```
onDismiss(id) {
# leanpub-start-insert
  const updatedList = this.state.list.filter(function isNotId(it
    return item.objectID !== id;
  });
# leanpub-end-insert
}
```

在下一步中，你可以抽取函数并将其传递给 filter 函数。

{title="src/App.js",lang=javascript}

```
onDismiss(id) {
# leanpub-start-insert
  function isNotId(item) {
    return item.objectID !== id;
  }

  const updatedList = this.state.list.filter(isNotId);
# leanpub-end-insert
}
```

另外，可以通过使用 ES6 的箭头函数让代码更简洁。

{title="src/App.js",lang=javascript}

```
onDismiss(id) {
# leanpub-start-insert
  const isNotId = item => item.objectID !== id;
  const updatedList = this.state.list.filter(isNotId);
```

```
# leanpub-end-insert
}
```

你甚至可以内联到一行内完成，就像在按钮的 `onClick` 事件处理器做的一样，但如此会损失一些可读性。

{title="src/App.js",lang=javascript}

```
onDismiss(id) {
# leanpub-start-insert
  const updatedList = this.state.list.filter(item => item.object
# leanpub-end-insert
}
```

现在已经从列表中删除了点击项，但是 state 还并没有更新。因此你需要最后使用类方法 `setState()` 来更新组件 satate 中的列表了。

{title="src/App.js",lang=javascript}

```
onDismiss(id) {
  const isNotId = item => item.objectID !== id;
  const updatedList = this.state.list.filter(isNotId);
# leanpub-start-insert
  this.setState({ list: updatedList });
# leanpub-end-insert
}
```

现在重新运行你的程序并尝试点击 “Dismiss” 按钮，它应该是工作的。你现在所练习的就是 React 中的**单向数据流**。你在界面通过 `onClick` 触发一个动作，再通过函数或类方法修改组件的 state，最后组件的 `render()` 方法再次运行并更新界面。

Internal state update with unidirectional data flow

练习:

- 阅读更多关于 React的状态与生命周期

绑定

当使用 ES6 编写的React组件时，了解在 JavaScript 类的绑定会非常重要。在前面章节，你已经在构造函数中绑定了 `onDismiss()` 方法

{title="src/App.js",lang=javascript}

```
class App extends Component {
  constructor(props) {
    super(props);

    this.state = {
      list,
    };

    this.onDismiss = this.onDismiss.bind(this);
  }

  ...
}
```

为什么一开始就需要这么做呢？绑定的步骤是非常重要的，因为类方法不会自动绑定 `this` 到实例上。让我们通过下面的代码来做验证。

{title="Code Playground",lang=javascript}

```
class ExplainBindingsComponent extends Component {
  onClickMe() {
    console.log(this);
  }

  render() {
    return (
      <button
        onClick={this.onClickMe}
        type="button"
      >
        Click Me
      </button>
    );
  }
}
```

组件正确的渲染，但是当你点击按钮时候，你会在开发调试控制台中得到 `undefined` 。这是使用 React 主要的 bug 来源，因为当你想在类方法中访问 `this.state` 时，由于 `this` 是 `undefined` 所以并不能被检索到。所以为了确保 `this` 在类方法中是可访问的，你需要将 `this` 绑定到类方法上。

在下面的组件中，类方法在构造函数中正确绑定。

{title="Code Playground",lang=javascript}

```
class ExplainBindingsComponent extends Component {
# leanpub-start-insert
  constructor() {
    super();

    this.onClickMe = this.onClickMe.bind(this);
  }
# leanpub-end-insert

  onClickMe() {
    console.log(this);
  }

  render() {
    return (
      <button
        onClick={this.onClickMe}
        type="button"
      >
        Click Me
      </button>
    );
  }
}
```

再次尝试点击按钮，这个 `this` 对象就指向了类的实例。你现在就可以访问到 `this.state` 或者是后面会学习到的 `this.props` 。

类方法的绑定也可以写起其他地方，比如写在 `render()` 函数中。

{title="Code Playground",lang=javascript}

```
class ExplainBindingsComponent extends Component {
  onClickMe() {
    console.log(this);
  }
```

```
  render() {
    return (
      <button
# leanpub-start-insert
        onClick={this.onClickMe.bind(this)}
# leanpub-end-insert
        type="button"
      >
        Click Me
      </button>
    );
  }
}
```

但是你应该避免这样做，因为它会在每次 `render()` 方法执行时绑定类方法。总结来说组件每次运行更新时都会导致性能消耗。当在构造函数中绑定时，绑定只会在组件实例化时运行一次，这样做是一个更好的方式。

另外有一些人们提出在构造函数中定义业务逻辑类方法。

{title="Code Playground",lang=javascript}

```
class ExplainBindingsComponent extends Component {
  constructor() {
    super();

# leanpub-start-insert
    this.onClickMe = () => {
      console.log(this);
    }
# leanpub-end-insert
  }

  render() {
    return (
      <button
        onClick={this.onClickMe}
        type="button"
      >
        Click Me
      </button>
    );
  }
}
```

你同样也应该避免这样，因为随着时间的推移它会让你的构造函数变得混乱。构造函数目的只是实例化你的类以及所有的属性。这就是为什么我们应该把业务逻辑应该定义在构造函数之外。

{title="Code Playground",lang=javascript}

```
class ExplainBindingsComponent extends Component {
  constructor() {
    super();

    this.doSomething = this.doSomething.bind(this);
    this.doSomethingElse = this.doSomethingElse.bind(this);
  }

  doSomething() {
    // do something
  }

  doSomethingElse() {
    // do something else
  }

  ...
}
```

最后值得一提的是类方法可以通过 ES6 的箭头函数做到自动地绑定。

{title="Code Playground",lang=javascript}

```
class ExplainBindingsComponent extends Component {
  onClickMe = () => {
    console.log(this);
  }

  render() {
    return (
      <button
        onClick={this.onClickMe}
        type="button"
      >
        Click Me
      </button>
    );
  }
}
```

如果在构造函数中的重复绑定对你有所困扰，你可以使用这种方式代替。React 的官方文档中坚持在构造函数中绑定类方法，所以本书也会采用同样的方式。

练习:

- 尝试绑定不同的方法并在控制台中打印 `this` 对象

事件处理

本章节会让你对元素的事件处理有更深入的了解，在你的应用程序中，你将使用下面的按钮来从列表中忽略一项内容。

{title="src/App.js",lang=javascript}

```
...

<button
  onClick={() => this.onDismiss(item.objectID)}
  type="button"
>
  Dismiss
</button>

...
```

这已经是一个复杂的例子了，因为你必须传递一个参数到类的方法，因此你需要将它封装到另一个（箭头）函数中，基本上，由于要传递给事件处理器使用，因此它必须是一个函数。下面的代码不会工作，因为类方法会在浏览器中打开程序时立即执行。

{title="src/App.js",lang=javascript}

```
...

<button
  onClick={this.onDismiss(item.objectID)}
  type="button"
>
  Dismiss
</button>

...
```

当使用 `onClick={doSomething()}` 时，`doSomething()` 函数会在浏览器打开程序时立即执行，由于监听表达式是函数执行的返回值而不再是函数，所以点击按钮时不会有任何事发生。但当使用 `onClick={doSomething}` 时，因为 `doSomething` 是一个函数，所以它会在点击按钮时执行。同样的规则也适用于在程序中使用的 `onDismiss()` 类方法。

然而，使用 `onClick={this.onDismiss}` 并不够，因为这个类方法需要接收 item.objectID 属性来识别那个将要被忽略的项，这就是为什么它需要被封装到另一个函数中来传递这个属性。这个概念在 JavaScript 中被称为高阶函数，稍后会做简要解释。

{title="src/App.js",lang=javascript}

```
...

<button
  onClick={() => this.onDismiss(item.objectID)}
  type="button"
>
  Dismiss
</button>

...
```

其中一个解决方案是在外部定义一个包装函数，并且只将定义的函数传递给处理程序。因为需要访问特定的列表项，所以它必须位于 map 函数块的内部。

{title="src/App.js",lang=javascript}

```
class App extends Component {

  ...

  render() {
    return (
      <div className="App">
        {this.state.list.map(item => {
# leanpub-start-insert
          const onHandleDismiss = () =>
            this.onDismiss(item.objectID);
# leanpub-end-insert

          return (
```

```
            <div key={item.objectID}>
              <span>
                <a href={item.url}>{item.title}</a>
              </span>
              <span>{item.author}</span>
              <span>{item.num_comments}</span>
              <span>{item.points}</span>
              <span>
                <button
# leanpub-start-insert
                  onClick={onHandleDismiss}
# leanpub-end-insert
                  type="button"
                >
                  Dismiss
                </button>
              </span>
            </div>
          );
        }
        )}
      </div>
    );
  }
}
```

毕竟，传给元素事件处理器的内容必须是函数。作为一个示例，请尝试以下代码：

{title="src/App.js",lang=javascript}

```
class App extends Component {

  ...

  render() {
    return (
      <div className="App">
        {this.state.list.map(item =>
            ...
            <span>
              <button
# leanpub-start-insert
                onClick={console.log(item.objectID)}
# leanpub-end-insert
                type="button"
              >
```

```
                Dismiss
              </button>
            </span>
          </div>
        )}
      </div>
    );
  }
}
```

它会在浏览器加载该程序时执行，但点击按钮时并不会。而下面的代码只会在点击按钮时执行。它是一个在触发事件时才会执行的函数。

{title="src/App.js",lang=javascript}

```
...

<button
# leanpub-start-insert
  onClick={function () {
    console.log(item.objectID)
  }}
# leanpub-end-insert
  type="button"
>
  Dismiss
</button>

...
```

为了保持简洁，你可以把它转成一个 JavaScript ES6 的箭头函数，和我们在 `onDismiss()` 类方法时做的一样。

{title="src/App.js",lang=javascript}

```
...

<button
# leanpub-start-insert
  onClick={() => console.log(item.objectID)}
# leanpub-end-insert
  type="button"
>
  Dismiss
</button>
```

```
...
```

经常会有 React 初学者在事件处理中使用函数遇到困难，这就是为什么我要在这里更详细的解释它。最后，你应该使用下面的代码来拥有一个可以访问 `item` 对象的 `objectID` 属性简洁的内联 JavaScript ES6 箭头函数。

{title="src/App.js",lang=javascript}

```
class App extends Component {
  ...

  render() {
    return (
      <div className="App">
        {this.state.list.map(item =>
          <div key={item.objectID}>
            ...
            <span>
# leanpub-start-insert
              <button
                onClick={() => this.onDismiss(item.objectID)}
                type="button"
              >
                Dismiss
              </button>
# leanpub-end-insert
            </span>
          </div>
        )}
      </div>
    );
  }
}
```

另一个经常会被提到的性能相关话题是在事件处理程序中使用箭头函数的影响。例如，`onClick` 事件处理中的 `onDismiss()` 方法被封装在另一个箭头函数中以便能传递项标识。因此每次 `render()` 执行时，事件处理程序就会实例化一个高阶箭头函数，它*可能*会对你的程序性能产生影响，但在大多数情况下你都不会注意到这个问题。假设你有一个包含1000个项目的巨大数据表，每一行或者列在事件处理程序中都有这样一个箭头函数，这个时候就需要考虑性能影响，因此你可以实现一个专用的按钮组件来在构造函数中绑定方法，但这是一个不成熟的优化。所以现在，专注到学习 React 会更有价值。

练习：

- 尝试在按钮的 `onClick` 处理程序中使用函数的不同方法。

和表单交互

让我们在程序中加入表单来体验 React 和表单事件的交互，我们将在程序中加入搜索功能，列表会根据输入框的内容对标题进行过滤。

第一步，你需要在 JSX 中定义一个带有输入框的表单。

{title="src/App.js",lang=javascript}

```
class App extends Component {

  ...

  render() {
    return (
      <div className="App">
# leanpub-start-insert
        <form>
          <input type="text" />
        </form>
# leanpub-end-insert
        {this.state.list.map(item =>
          ...
        )}
      </div>
    );
  }
}
```

在下面的场景中，将会使用在输入框中的内容作为搜索字段来临时过滤列表。为了能根据输入框的值过滤列表，你需要将输入框的值储存在你的本地状态中，但是如何访问这个值呢？你可以使用 React 的**合成事件**来访问事件返回值。

让我们为输入框定义一个 `onChange` 处理程序。

{title="src/App.js",lang=javascript}

```
class App extends Component {

  ...

  render() {
    return (
      <div className="App">
        <form>
# leanpub-start-insert
          <input
            type="text"
            onChange={this.onSearchChange}
          />
# leanpub-end-insert
        </form>
        ...
      </div>
    );
  }
}
```

这个函数被绑定到组件上，因此再次成为一个类方法，你必定义方法并 bind 它。

{title="src/App.js",lang=javascript}

```
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      list,
    };

# leanpub-start-insert
    this.onSearchChange = this.onSearchChange.bind(this);
# leanpub-end-insert
    this.onDismiss = this.onDismiss.bind(this);
  }

# leanpub-start-insert
  onSearchChange() {
    ...
  }
# leanpub-end-insert
```

```
  ...
}
```

在元素中使用监听时，你可以在回调函数的签名中访问到 React 的合成事件。

{title="src/App.js",lang=javascript}

```
class App extends Component {

  ...

# leanpub-start-insert
  onSearchChange(event) {
# leanpub-end-insert
    ...
  }

  ...
}
```

event 对象的 target 属性中带有输入框的值，因此你可以使用 `this.setState()` 来更新本地的搜索词的状态了。

{title="src/App.js",lang=javascript}

```
class App extends Component {

  ...

  onSearchChange(event) {
# leanpub-start-insert
    this.setState({ searchTerm: event.target.value });
# leanpub-end-insert
  }

  ...
}
```

此外，你应该记住在构造函数中为 `searchTerm` 定义初始状态，输入框在开始时应该是空的，因此初始值应该是空字符串。

{title="src/App.js",lang=javascript}

```
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      list,
# leanpub-start-insert
      searchTerm: '',
# leanpub-end-insert
    };

    this.onSearchChange = this.onSearchChange.bind(this);
    this.onDismiss = this.onDismiss.bind(this);
  }

  ...
}
```

现在你将会把输入框每次变化的输入值储存到组件的内部状态中。

关于一个在 React 组件中更新状态的简要说明，在使用 `this.setState()` 更新 `searchTerm` 时应该把这个列表也传递进去来保留它才是公平的，但事实并非如此，React 的 `this.setState()` 是一个浅合并，在更新一个唯一的属性时，他会保留状态对象中的其他属性，因此即使你已经在列表状态中排除了一个项，在更新 `searchTerm` 属性时也会保持不变。

让我们回到你的程序中，现在列表还没有根据储存在本地状态中的输入字段进行过滤。基本上，你已经具有了根据 `searchTerm` 临时过滤列表的所有东西。那么怎么暂时的过滤它呢？你可以在 `render()` 方法中，在 map 映射列表之前，插入一个过滤的方法。这个过滤方法将只会匹配标题属性中有 `searchTerm` 内容的列表项。你已经使用过了 JavaScript 内置的 filter 功能，让我们再用一次，你可以在 map 函数之前加入 filter 函数，因为 filter 函数返回一个新的数组，所以 map 函数可以这样方便的使用。

{title="src/App.js",lang=javascript}

```
class App extends Component {

  ...

  render() {
    return (
      <div className="App">
```

```
        <form>
          <input
            type="text"
            onChange={this.onSearchChange}
          />
        </form>
# leanpub-start-insert
        {this.state.list.filter(...).map(item =>
# leanpub-end-insert
          ...
        )}
      </div>
    );
  }
}
```

让我们用一种不同的方式来处理过滤函数，我们想在 ES6 组件类之外定义一个传递给过滤函数的函数参数，在这里我们不能访问到组件内的状态，所以无法访问 `searchTerm` 属性来作为筛选条件求值，我们需要传递 `searchTerm` 到过滤函数并返回一个新函数来根据条件求值，这叫做高阶函数。

一般来说，我不会提到高阶函数，但在 React 中了解高阶函数是有意义的，因为在 React 中有一个高阶组件的概念，你将在这本书的后面了解到这个概念。但现在，让我们关注到过滤器的功能。

首先，你需要在 App 组件外定义一个高阶函数。

{title="src/App.js",lang=javascript}

```
# leanpub-start-insert
function isSearched(searchTerm) {
  return function(item) {
    // some condition which returns true or false
  }
}
# leanpub-end-insert

class App extends Component {

  ...

}
```

该函数接受 `searchTerm` 并返回另一个函数，因为所有的 filter 函数都接受一个函数作为它的输入，返回的函数可以访问列表项目对象，因为它是传给

filter 函数的函数。此外，返回的函数将会根据函数中定义的条件对列表进行过滤。让我们来定义条件。

{title="src/App.js",lang=javascript}

```
function isSearched(searchTerm) {
  return function(item) {
# leanpub-start-insert
    return item.title.toLowerCase().includes(searchTerm.toLowerC
# leanpub-end-insert
  }
}

class App extends Component {

  ...

}
```

条件是列表中项目的标题属性和输入的 `searchTerm` 参数相匹配，你可以使用 JavaScript 内置的 `includes` 功能来实现这一点。只有满足匹配时才会返回 true 并将项目保留在列表中。当不匹配时，项目会从列表中移除。但需要注意的是，你需要把输入内容和待匹配的内容都转换成小写，否则，搜索词 'redux' 和列表中标题叫 'Redux' 的项目无法匹配。由于我们使用的是一个不可变的列表，并使用 filter 函数返回一个新列表，所以本地状态中的原始列表根本就没有被修改过。

还有一点需要注意，我们使用了 Javascript 内置的 includes 功能，它已经是一个 ES6 的特性了。这在 ES5 中该如何实现呢？你将使用 `indexOf()` 函数来获取列表中项的索引，当项目在列表中时，`indexOf()` 将会返回它的索引。

{title="Code Playground",lang="javascript"}

```
// ES5
string.indexOf(pattern) !== -1

// ES6
string.includes(pattern)
```

另一个优雅的重构可以用 ES6 箭头函数完成，它可以让函数更加整洁:

{title="Code Playground",lang="javascript"}

```
// ES5
function isSearched(searchTerm) {
  return function(item) {
    return item.title.toLowerCase().includes(searchTerm.toLowerC
  }
}

// ES6
const isSearched = searchTerm => item =>
  item.title.toLowerCase().includes(searchTerm.toLowerCase());
```

人们会争论哪个函数更易读，就我个人而言，我更习惯第二个。React 的生态使用了大量的函数式编程概念。通常情况下，你会使用一个函数返回另一个函数（高阶函数）。在 JavaScript ES6 中，可以使用箭头函数更简洁的表达这些。

最后，你需要使用定义的 `isSearched()` 函数来过滤你的列表，你从本地状态中传递 `searchTerm` 属性返回一个根据条件过滤列表的输入过滤函数。之后它会映射过滤后的列表用于显示每个列表项的元素。

{title="src/App.js",lang=javascript}

```
class App extends Component {

  ...

  render() {
    return (
      <div className="App">
        <form>
          <input
            type="text"
            onChange={this.onSearchChange}
          />
        </form>
# leanpub-start-insert
        {this.state.list.filter(isSearched(this.state.searchTerm
# leanpub-end-insert
          ...
        )}
      </div>
    );
  }
}
```

搜索功能现在应该起作用了，在浏览器中自己尝试一下。

练习：

- 阅读更多 React 事件 相关内容
- 阅读更多 高阶函数 相关内容

ES6 解构

在 JavaScript ES6 中有一种更方便的方法来访问对象和数组的属性，叫做解构。比较下面 JavaScript ES5 和 ES6 的代码片段。

{title="Code Playground",lang="javascript"}

```
const user = {
  firstname: 'Robin',
  lastname: 'Wieruch',
};

// ES5
var firstname = user.firstname;
var lastname = user.lastname;

console.log(firstname + ' ' + lastname);
// output: Robin Wieruch

// ES6
const { firstname, lastname } = user;

console.log(firstname + ' ' + lastname);
// output: Robin Wieruch
```

在 JavaScript ES5 中每次访问对象的属性都需要额外添加一行代码，但在 JavaScript ES6 中可以在一行中进行。可读性最好的方法是在将对象解构成多个属性时使用多行。

{title="Code Playground",lang="javascript"}

```
const {
  firstname,
  lastname
} = user;
```

对于数组一样可以使用解构，同样，多行代码会使你的代码保持可读性。

{title="Code Playground",lang="javascript"}

```
const users = ['Robin', 'Andrew', 'Dan'];
const [
  userOne,
  userTwo,
  userThree
] = users;

console.log(userOne, userTwo, userThree);
// output: Robin Andrew Dan
```

也许你已经注意到，程序组件内的状态对象也可以使用同样的方式解构，你可以让 map 和 filter 部分的代码更简短。

{title="src/App.js",lang=javascript}

```
  render() {
# leanpub-start-insert
    const { searchTerm, list } = this.state;
# leanpub-end-insert
    return (
      <div className="App">
        ...
# leanpub-start-insert
        {list.filter(isSearched(searchTerm)).map(item =>
# leanpub-end-insert
          ...
        )}
      </div>
    );
```

你也可以使用 ES5 或者 ES6 的方式来做：

{title="Code Playground",lang="javascript"}

```
// ES5
var searchTerm = this.state.searchTerm;
var list = this.state.list;

// ES6
const { searchTerm, list } = this.state;
```

但由于这本书大部分时候都使用了 JavaScript ES6，所以你也可以坚持使用它。

练习：

- 阅读更多ES6 解构的相关内容

受控组件

你已经了解了 React 中的单向数据流，同样的规则适用于更新本地状态 `searchTerm` 来过滤列表的输入框。当状态变化时，`render()` 方法将再次运行，并使用最新状态中的 `searchTerm` 值来作为过滤条件。

但是我们是否忘记了输入元素的一些东西？一个 HTML 输入标签带有一个 `value` 属性，这个属性通常有一个值作为输入框的显示，在本例中，它是 `searchTerm` 属性。然而，看起来我们在 React 好像并不需要它。

这是错误的，表单元素比如 `<input>`，`<textarea>` 和 `<select>` 会以原生 HTML 的形式保存他们自己的状态。一旦有人从外部做了一些修改，它们就会修改内部的值，在 React 中这被称为**不受控组件**，因为它们自己处理状态。在 React 中，你应该确保这些元素变为**受控组件**。

你应该怎么做呢？你只需要设置输入框的值属性，这个值已经在 `searchTerm` 状态属性中保存了，那么为什么不从这里访问呢？

{title="src/App.js",lang=javascript}

```
class App extends Component {

  ...

  render() {
    const { searchTerm, list } = this.state;
    return (
      <div className="App">
        <form>
          <input
            type="text"
# leanpub-start-insert
            value={searchTerm}
# leanpub-end-insert
            onChange={this.onSearchChange}
          />
```

```
        </form>
        ...
      </div>
    );
  }
}
```

就是这样。现在输入框的单项数据流循环是自包含的，组件内部状态是输入框的唯一数据来源。

整个内部状态管理和单向数据流可能对你来说比较新，但你一旦习惯了它，你就会自然而然的在 React 中实现它。一般来说，React 带来一种新的模式，将单向数据流引入到单页面应用的生态中，到目前为止，它已经被几个框架和库所采用。

练习

- 阅读更多React 表单相关内容

拆分组件

现在，你有一个大型的 App 组件。它在不停地扩展，最终可能会变得混乱。你可以开始将它拆分成若干个更小的组件。

让我们开始使用一个用于搜索的输入组件和一个用于展示的列表组件。

{title="src/App.js",lang=javascript}

```
class App extends Component {

  ...

  render() {
    const { searchTerm, list } = this.state;
    return (
      <div className="App">
# leanpub-start-insert
        <Search />
        <Table />
# leanpub-end-insert
      </div>
    );
  }
}
```

你可以给组件传递属性并在组件中使用它们。至于 App 组件，它需要传递由本地状态 (state) 托管的属性和它自己的类方法。

{title="src/App.js",lang=javascript}

```
class App extends Component {

  ...

  render() {
    const { searchTerm, list } = this.state;
    return (
      <div className="App">
# leanpub-start-insert
        <Search
          value={searchTerm}
          onChange={this.onSearchChange}
        />
        <Table
          list={list}
          pattern={searchTerm}
          onDismiss={this.onDismiss}
        />
# leanpub-end-insert
      </div>
    );
  }
}
```

现在你可以接着 App 组件定义这些组件。这些组件仍然是 ES6 类组件，它们会渲染和之前相同的元素。

第一个是 Search 组件。

{title="src/App.js",lang=javascript}

```
class App extends Component {
  ...
}

# leanpub-start-insert
class Search extends Component {
  render() {
    const { value, onChange } = this.props;
    return (
      <form>
```

```
        <input
          type="text"
          value={value}
          onChange={onChange}
        />
      </form>
    );
  }
}
# leanpub-end-insert
```

第二个是 Table 组件。

{title="src/App.js",lang=javascript}

```
...

# leanpub-start-insert
class Table extends Component {
  render() {
    const { list, pattern, onDismiss } = this.props;
    return (
      <div>
        {list.filter(isSearched(pattern)).map(item =>
          <div key={item.objectID}>
            <span>
              <a href={item.url}>{item.title}</a>
            </span>
            <span>{item.author}</span>
            <span>{item.num_comments}</span>
            <span>{item.points}</span>
            <span>
              <button
                onClick={() => onDismiss(item.objectID)}
                type="button"
              >
                Dismiss
              </button>
            </span>
          </div>
        )}
      </div>
    );
  }
}
# leanpub-end-insert
```

现在你有了三个 ES6 类组件。你可能已经注意到，`props` 对象可以通过这个类实例的 `this` 来访问。props 是 properties 的简写，当你在 App 组件里面使用它时，它有你传递给这些组件的所有值。这样，组件可以沿着组件树向下传递属性。

从 App 组件中提取这些组件之后，你就可以在别的地方去重用它们了。因为组件是通过 props 对象来获取它们的值，所以当你在别的地方重用它时，你可以每一次都传递不同的 props，这些组件就变得可复用了。

练习：

- 从已经完成的 Search 和 Table 组件中找出可以进一步提取的组件。
- 但是不要现在就去做，否则在接下来的几个章节你会遇到冲突。

可组合组件

在 props 对象中还有一个小小的属性可供使用: `children` 属性。通过它你可以将元素从上层传递到你的组件中，这些元素对你的组件来说是未知的，但是却为组件相互组合提供了可能性。让我们来看一看，当你只将一个文本（字符串）作为子元素传递到 Search 组件中会怎样。

{title="src/App.js",lang=javascript}

```
class App extends Component {

  ...

  render() {
    const { searchTerm, list } = this.state;
    return (
      <div className="App">
# leanpub-start-insert
        <Search
          value={searchTerm}
          onChange={this.onSearchChange}
        >
          Search
        </Search>
# leanpub-end-insert
        <Table
          list={list}
          pattern={searchTerm}
          onDismiss={this.onDismiss}
```

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

现在 Search 组件可以从 props 对象中解构出 children 属性。然后它就可以指定这个 children 应该显示在哪里。

{title="src/App.js",lang=javascript}

```
class Search extends Component {
  render() {
# leanpub-start-insert
    const { value, onChange, children } = this.props;
# leanpub-end-insert
    return (
      <form>
# leanpub-start-insert
        {children} <input
# leanpub-end-insert
          type="text"
          value={value}
          onChange={onChange}
        />
      </form>
    );
  }
}
```

现在，你应该可以在输入框旁边看到这个 "Search" 文本了。当你在别的地方使用 Search 组件时，如果你喜欢，你可以选择一个不同的文本。总之，它不仅可以把文本作为子元素传递，还可以将一个元素或者元素树（它还可以再次封装成组件）作为子元素传递。children 属性让组件相互组合到一起成为可能。

练习：

- 阅读更多关于 React 组件模型 的内容

可复用组件

可复用和可组合组件让你能够思考合理的组件分层，它们是 React 视图层的基础。前面几章提到了可重用性的术语。现在你可以复用 Search 和 Table 组件了。甚至 App 组件都是可复用的了，因为你可以在别的地方重新实例化它。

让我们再来定义一个可复用组件 Button，最终会被更频繁地复用。

{title="src/App.js",lang=javascript}

```
class Button extends Component {
  render() {
    const {
      onClick,
      className,
      children,
    } = this.props;

    return (
      <button
        onClick={onClick}
        className={className}
        type="button"
      >
        {children}
      </button>
    );
  }
}
```

声明这样一个组件可能看起来有点多余。你将会用 `Button` 组件来替代 `button` 元素。它只省去了 `type="button"`。当你想使用 Button 组件的时候，你还得去定义除了 type 之外的所有属性。但这里你必须要考虑到长期投资。想象在你的应用中有若干个 button，但是你想改变它们的一个属性、样式或者行为。如果没有这个组件的话，你就必须重构每个 button。相反，Button 组件拥有单一可信数据源。一个 Button 组件可以立即重构所有 button。一个 Button 组件统治所有的 button。

既然你已经有了 button 元素，你可以用 Button 组件代替。它省略了 type 属性，因为 Button 组件已经指定了。

{title="src/App.js",lang=javascript}

```
class Table extends Component {
  render() {
    const { list, pattern, onDismiss } = this.props;
    return (
      <div>
        {list.filter(isSearched(pattern)).map(item =>
          <div key={item.objectID}>
            <span>
              <a href={item.url}>{item.title}</a>
            </span>
            <span>{item.author}</span>
            <span>{item.num_comments}</span>
            <span>{item.points}</span>
            <span>
# leanpub-start-insert
              <Button onClick={() => onDismiss(item.objectID)}>
                Dismiss
              </Button>
# leanpub-end-insert
            </span>
          </div>
        )}
      </div>
    );
  }
}
```

Button 组件期望在 props 里面有一个 `className` 属性. `className` 属性是 React 基于 HTML 属性 `class` 的另一个衍生物。但是当使用 Button 组件时，我们并没有传递任何 `className` 属性，所以在 Button 组件的代码中，我们更应该明确地标明 `className` 是可选的。

因此，你可以使用默认参数，它是一个 JavaScript ES6 的特性。

{title="src/App.js",lang=javascript}

```
class Button extends Component {
  render() {
    const {
      onClick,
# leanpub-start-insert
      className = '',
# leanpub-end-insert
      children,
```

```
    } = this.props;

    ...
  }
}
```

这样当使用 Button 组件时，若没有指定 `className` 属性，它的值就是一个空字符串，而非 `undefined`。

练习：

- 阅读更多关于 ES6 默认参数 的内容

Component Declarations 组件声明

现在你已经有四个 ES6 类组件了，但是你可以做得更好。让我来介绍一下函数式无状态组件 (functional stateless components)，作为除了 ES6 类组件的另一个选择。在重构你的组件之前，让我来介绍一下 React 不同的组件类型。

- **函数式无状态组件:** 这类组件就是函数，它们接收一个输入并返回一个输出。输入是 props，输出就是一个普通的 JSX 组件实例。到这里，它和 ES6 类组件非常的相似。然而，函数式无状态组件是函数（函数式的），并且它们没有本地状态（无状态的）。你不能通过 `this.state` 或者 `this.setState()` 来访问或者更新状态，因为这里没有 `this` 对象。此外，它也没有生命周期方法。虽然你还没有学过生命周期方法，但是你已经用到了其中两个：`constructor()` and `render()`。constructor 在一个组件的生命周期中只执行一次，而 `render()` 方法会在最开始执行一次，并且每次组件更新时都会执行。当你阅读到后面关于生命周期方法的章节时，要记得函数式无状态组件是没有生命周期方法的。

- **ES6 类组件:** 在你的四个组件中，你已经使用过这类组件了。在类的定义中，它们继承自 React 组件。`extend` 会注册所有的生命周期方法，只要在 React component API 中，都可以在你的组件中使用。通过这种方式你可以使用 `render()` 类方法。此外，通过使用 `this.state` 和 `this.setState()`，你可以在 ES6 类组件中储存和操控 state。

- **React.createClass:** 这类组件声明曾经在老版本的 React 中使用，仍然存在于很多 ES5 React 应用中。但是为了支持 JavaScript ES6，Faceb

> ook 声明它已经不推荐使用了。他们还在 React 15.5 中加入了不推荐使用的警告。你不会在本书使用它。

因此这里基本只剩下两种组件声明了。但是什么时候更适合使用函数式无状态组件而非 ES6 类组件？一个经验法则就是当你不需要本地状态或者组件生命周期方法时，你就应该使用函数式无状态组件。最开始一般使用函数式无状态组件来实现你的组件，一旦你需要访问 state 或者生命周期方法时，你就必须要将它重构成一个 ES6 类组件。在我们的应用中，为了学习 React，我们采取了相反的方式。

让我们回到你的应用中。App 组件使用内部状态，这就是为什么它必须作为 ES6 类组件存在的原因。但是你的其他三个 ES6 类组件都是无状态的，它们不需要使用 `this.state` 或者 `this.setState()`，甚至都不需要使用生命周期函数。让我们一起把 Search 组件重构成一个函数式无状态组件。Table 和 Button 组件的重构会留做你的练习。

{title="src/App.js",lang=javascript}

```
# leanpub-start-insert
function Search(props) {
  const { value, onChange, children } = props;
  return (
    <form>
      {children} <input
        type="text"
        value={value}
        onChange={onChange}
      />
    </form>
  );
}
# leanpub-end-insert
```

基本上就是这样了。props 可以在函数签名（译者注：这里应指函数入参）中访问，返回值是 JSX。你已经知道 ES6 解构了，所以在函数式无状态组件中，你可以优化之前的写法。最佳实践就是在函数签名中通过解构 props 来使用它。

{title="src/App.js",lang=javascript}

```
# leanpub-start-insert
function Search({ value, onChange, children }) {
# leanpub-end-insert
```

```
  return (
    <form>
      {children} <input
        type="text"
        value={value}
        onChange={onChange}
      />
    </form>
  );
}
```

但是它还可以变得更好。你已经知道，ES6 箭头函数允许让你保持你的函数简洁。你可以移除函数的块声明（译者注：即花括号`{}`）。在简化的函数体中，表达式会自动作为返回值，因此你可以将 return 语句移除。因为你的函数式无状态组件是一个函数，你同样可以用这种方式来简化它。

{title="src/App.js",lang=javascript}

```
# leanpub-start-insert
const Search = ({ value, onChange, children }) =>
  <form>
    {children} <input
      type="text"
      value={value}
      onChange={onChange}
    />
  </form>
# leanpub-end-insert
```

最后一步对于强制只用 props 作为输入和 JSX 作为输出非常有用。这之间没有任何别的东西。但是你仍然可以在 ES6 箭头函数块声明中*做一些事情*。

{title="Code Playground",lang=javascript}

```
const Search = ({ value, onChange, children }) => {

  // do something

  return (
    <form>
      {children} <input
        type="text"
        value={value}
        onChange={onChange}
      />
```

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

但是你现在并不需要这样做，这也是为什么你可以让之前的版本没有块声明。当使用块声明时，人们往往容易在这个函数里面做过多的事情。通过移除块声明，你可以专注在函数的输入和输出上。

现在你已经有一个轻量的函数式无状态组件了。一旦你需要访问它的内部组件状态或者生命周期方法，你最好将它重构成一个 ES6 类组件。另外，你也已经看到，JavaScript ES6 是如何被用到 React 组件中并让它们变得更加的简洁和优雅。

练习：

- 将 Table 和 Button 组件重构成函数式无状态组件
- 阅读更多关于 ES6 类组件和函数式无状态组件 的内容

给组件声明样式

让我们给你的应用和组件添加一些基本的样式。你可以复用 *src/App.css* 和 *src/index.css* 文件。因为你是用 *create-react-app* 来创建的，所以这些文件应该已经在你的项目中了。它们应该也被引入到你的 *src/App.js* 和 *src/index.js* 文件中了。我准备了一些 CSS，你可以直接复制粘贴到这些文件中，你也可以随意使用你自己的样式。

首先，给你的整个应用声明样式。

{title="src/index.css",lang="css"}

```
body {
  color: #222;
  background: #f4f4f4;
  font: 400 14px CoreSans, Arial,sans-serif;
}

a {
  color: #222;
}

a:hover {
  text-decoration: underline;
```

```
}

ul, li {
  list-style: none;
  padding: 0;
  margin: 0;
}

input {
  padding: 10px;
  border-radius: 5px;
  outline: none;
  margin-right: 10px;
  border: 1px solid #dddddd;
}

button {
  padding: 10px;
  border-radius: 5px;
  border: 1px solid #dddddd;
  background: transparent;
  color: #808080;
  cursor: pointer;
}

button:hover {
  color: #222;
}

*:focus {
  outline: none;
}
```

其次，在 App 文件中给你的组件声明样式。

{title="src/App.css",lang="css"}

```
.page {
  margin: 20px;
}

.interactions {
  text-align: center;
}

.table {
  margin: 20px 0;
```

```
}

.table-header {
  display: flex;
  line-height: 24px;
  font-size: 16px;
  padding: 0 10px;
  justify-content: space-between;
}

.table-empty {
  margin: 200px;
  text-align: center;
  font-size: 16px;
}

.table-row {
  display: flex;
  line-height: 24px;
  white-space: nowrap;
  margin: 10px 0;
  padding: 10px;
  background: #ffffff;
  border: 1px solid #e3e3e3;
}

.table-header > span {
  overflow: hidden;
  text-overflow: ellipsis;
  padding: 0 5px;
}

.table-row > span {
  overflow: hidden;
  text-overflow: ellipsis;
  padding: 0 5px;
}

.button-inline {
  border-width: 0;
  background: transparent;
  color: inherit;
  text-align: inherit;
  -webkit-font-smoothing: inherit;
  padding: 0;
  font-size: inherit;
  cursor: pointer;
}
```

```
.button-active {
  border-radius: 0;
  border-bottom: 1px solid #38BB6C;
}
```

现在你可以在一些组件中使用这些样式。但是别忘了使用 React 的 `className` ，而不是 HTML 的 `class` 属性。

首先，将它应用到你的 App ES6 类组件中。

{title="src/App.js",lang=javascript}

```
class App extends Component {

  ...

  render() {
    const { searchTerm, list } = this.state;
    return (
# leanpub-start-insert
      <div className="page">
        <div className="interactions">
# leanpub-end-insert
          <Search
            value={searchTerm}
            onChange={this.onSearchChange}
          >
            Search
          </Search>
# leanpub-start-insert
        </div>
# leanpub-end-insert
        <Table
          list={list}
          pattern={searchTerm}
          onDismiss={this.onDismiss}
        />
# leanpub-start-insert
      </div>
# leanpub-end-insert
    );
  }
}
```

其次，将它应用到你的 Table 函数式无状态组件中。

{title="src/App.js",lang=javascript}

```
const Table = ({ list, pattern, onDismiss }) =>
# leanpub-start-insert
  <div className="table">
# leanpub-end-insert
    {list.filter(isSearched(pattern)).map(item =>
# leanpub-start-insert
      <div key={item.objectID} className="table-row">
# leanpub-end-insert
        <span>
          <a href={item.url}>{item.title}</a>
        </span>
        <span>{item.author}</span>
        <span>{item.num_comments}</span>
        <span>{item.points}</span>
        <span>
          <Button
            onClick={() => onDismiss(item.objectID)}
# leanpub-start-insert
            className="button-inline"
# leanpub-end-insert
          >
            Dismiss
          </Button>
        </span>
# leanpub-start-insert
      </div>
# leanpub-end-insert
    )}
# leanpub-start-insert
  </div>
# leanpub-end-insert
```

现在你已经给你的应用和组件添加了基本的 CSS 样式，看起来应该非常不错。如你所知，JSX 混合了 HTML 和 JavaScript。现在有人呼吁将 CSS 也加入进去，这就叫作内联样式 (inline style)。你可以定义 JavaScript 对象，并传给一个元素的 style 属性。

让我们通过使用内联样式来使 Table 的列宽自适应。

{title="src/App.js",lang=javascript}

```
const Table = ({ list, pattern, onDismiss }) =>
  <div className="table">
    {list.filter(isSearched(pattern)).map(item =>
```

```
      <div key={item.objectID} className="table-row">
# leanpub-start-insert
        <span style={{ width: '40%' }}>
          <a href={item.url}>{item.title}</a>
        </span>
        <span style={{ width: '30%' }}>
          {item.author}
        </span>
        <span style={{ width: '10%' }}>
          {item.num_comments}
        </span>
        <span style={{ width: '10%' }}>
          {item.points}
        </span>
        <span style={{ width: '10%' }}>
          <Button
            onClick={() => onDismiss(item.objectID)}
            className="button-inline"
          >
            Dismiss
          </Button>
        </span>
# leanpub-end-insert
      </div>
    )}
  </div>
```

现在样式已经内联了。你可以在你的元素之外定义一个 style 对象，这样可以让它变得更整洁。

{title="Code Playground",lang="javascript"}

```
const largeColumn = {
  width: '40%',
};

const midColumn = {
  width: '30%',
};

const smallColumn = {
  width: '10%',
};
```

随后你可以将它们用于你的 columns ：`<span style={smallColumn}>`。

总而言之，关于 React 中的样式，你会找到不同的意见和解决方案。现在你已经用过纯 CSS 和 内联样式了。这足以开始。

在这里我不想下定论，但是想给你一些更多的选择。你可以自行阅读并应用它们。但是如果你刚开始使用 React，目前我会推荐你坚持纯 CSS 和内联样式。

- styled-components
- CSS Modules

{pagebreak}

你已经学习了编写一个 React 应用所需要的基础知识了！让我们来回顾一下前面几个章节:

- React
- 使用 `this.state` 和 `setState()` 来管理你的内部组件状态
- 将函数或者类方法传递到你的元素处理器
- 在 React 中使用表单或者事件来添加交互
- 在 React 中单向数据流是一个非常重要的概念
- 拥抱 controlled components
- 通过 children 和可复用组件来组合组件
- ES6 类组件和函数式无状态组件的使用方法和实现
- 给你的组件声明样式的方法
- ES6
- 绑定到一个类的函数叫作类方法
- 解构对象和数组
- 默认参数
- General
- 高阶函数

该休息一下了，吸收这些知识然后转化成你自己的东西。你可以用你已有的代码来做个实验。另外，你可以进一步阅读官方文档

你可以在官方代码仓库找到源码。