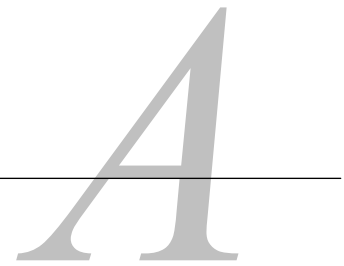
附录 A

PropTypes



PropTypes 是验证通过组件 props 传递的值的一种方法。定义良好的接口可以在应用程序运行时为我们提供一层安全保障。它们还为组件的使用者提供了一层文档。

要使用 PropTypes，请使用 npm 安装该包：

```
$ npm i --save prop-types
```

然后将该库导入所需的文件中：

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

在 15.5.0 之前的 React 版本中，可以通过主 React 库访问 PropType。不过此行为已被弃用。

我们通过将 PropTypes 定义为组件类的静态属性来定义它们。可以将 propTypes 设置为类的静态属性，也可以在定义好类之后设置它们。

code/appendix/proptypes/Component.js

```
class Component extends React.Component {
  static propTypes = {
    name: PropTypes.string
  }
  // ...
  render() {
    return (<div>{this.props.name}</div>)
  }
}
```

还可以在类定义好之后将 propTypes 设置为组件的静态属性。

code/appendix/proptypes/Component.js

```
class Component extends React.Component {
  // ...
  render() {
    return (<div>{this.props.name}</div>)
  }
  // ...
Component.propTypes = {
  name: PropTypes.string
}
```

使用 createReactClass()定义 propType

在使用 createReactClass()方法定义组件时，我们将 propTypes 作为键传递，其值为具体的属性类型：

```
const Component = createReactClass({
  propTypes: {
    // 在这里定义 propType
  },
  render: function() {}

});
```

propTypes 对象的键定义了我们要验证的属性的名称，而值是具体的类型，由 PropTypes 对象（下面会讨论）或通过自定义函数来定义的。

PropTypes 对象导出了许多验证器，涵盖了我们将遇到的大多数情况。对于不太常见的情况，React 也允许我们定义自己的 PropType 验证器。

A.1　验证器

PropTypes 对象包含了一个公共验证器列表（但我们也可以定义自己的验证器，稍后会详细介绍）。

当使用无效类型传入一个属性或该属性类型验证失败时，则会向 JavaScript 控制台传递一个警告。这些警告只会在开发模式下显示，因此，如果我们在没有正确使用组件的情况下却意外地将应用程序部署到生产环境中时，用户不会看到该警告。

内置的验证器如下所示：

- string
- number
- boolean
- function
- object
- shape
- oneOf
- instanceOf
- array
- arrayOf
- node
- element
- any
- required

A.2 string

要将属性定义为字符串，可以使用 PropTypes.string。

code/appendix/proptypes/string.js

```
class Component extends React.Component {
  static propTypes = {
    name: PropTypes.string
  }
  // ...
  render() {
    return (<div>{this.props.name}</div>)
  }
}
```

要将字符串作为属性传递，我们可以将字符串本身作为属性传递，也可以使用大括号（{}）来定义字符串变量。以下方式在**功能上**是等效的：

```
<Component name={"Ari"} />
<Component name="Ari" />
```

A.3 number

要指定一个属性是一个数字，可以使用 PropTypes.number。

code/appendix/proptypes/number.js

```
class Component extends React.Component {
  static propTypes = {
    totalCount: PropTypes.number
  }
  // ...
  render() {
    return (<div>{this.props.totalCount}</div>)
  }
}
```

传递数字时，必须将它作为 JavaScript 值或用大括号包裹的变量的值传递：

```
var x = 20;

<Component totalCount={20} />
<Component totalCount={x} />
```

A.4 boolean

要指定一个布尔值（true 或 false），可以使用 PropTypes.bool。

code/appendix/proptypes/bool.js

```
class Component extends React.Component {
  static propTypes = {
    on: PropTypes.bool
  }
  // ...
  render() {
    return (
      <div>
        {this.props.on ? 'On' : 'Off'}
      </div>
    )
  }
}
```

要在 JSX 表达式中使用布尔值，可以将其作为 JavaScript 值传递。

```
var isOn = true;

<Component on={true} />
<Component on={false} />
<Component on={isOn} />
```

一个使用布尔值显示或隐藏内容的技巧是使用&&表达式。

例如，在 Component.render()函数中，如果我们只想在 on 为真时才显示内容，可以这样做：

```
{lang=js, line-numbers=off
render() {
  return (
    <div> {this.props.on && <p>This component is on</p>} </div>)}
}
```

A.5 function

也可以传递一个函数作为属性。要将一个属性定义为一个函数，可以使用 PropTypes.func。通常在编写 Form 组件时，我们会传入一个函数作为提交表单时（即 onSubmit()）调用的属性。通常会将一个属性定义为组件上需要的函数：

code/appendix/proptypes/func.js

```
class Component extends React.Component {
  static propTypes = {
    onPress: PropTypes.func
  }
  // ...
  render() {
    return (
      <div onClick={this.props.onPress}>
        Press me
```

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

可以使用 JavaScript 表达式语法将函数作为属性传递，如下所示：

```
const x = function(name) {};
const fn = value => alert("Value: " + value);

<Component onComplete={x} />
<Component onComplete={fn} />
```

A.6 object

可以通过 PropTypes.object 来要求一个属性是一个 JavaScript 对象：

code/appendix/proptypes/object.js

```
class Component extends React.Component {
  static propTypes = {
    user: PropTypes.object
  }
  // ...
  render() {
    const { user } = this.props
    return (
      <div>
        <h1>{user.name}</h1>
        <h5>{user.profile}</h5>
      </div>
    )
  }
}
```

发送对象作为属性，我们需要使用 JavaScript 表达式{}语法：

```
const user = {
  name: 'Ari'
}

<Component user={user} />
<Component user={{name: 'Anthony'}} />
```

A.7 对象的 shape 属性

React 允许我们使用 PropTypes.shape()来定义希望接收的对象的形状。PropTypes.shape()函数接受一个具有键/值对列表的对象，该列表指定一个对象应该具有的键和值的类型：

code/appendix/proptypes/objectOfShape.js

```
class Component extends React.Component {
  static propTypes = {
    user: PropTypes.shape({
      name: PropTypes.string,
      profile: PropTypes.string
    })
  }
  // ...
  render() {
    const { user } = this.props
    return (
      <div>
        <h1>{user.name}</h1>
        <h5>{user.profile}</h5>
      </div>
    )
  }
}
```

A.8　多种类型

我们有时事先并不知道属性具体是什么类型的，但可以接受多种类型中的其中一种。React 为我们提供了 oneOf()和 oneOfType()的 propTypes 来处理这些情况。

使用 oneOf()要求 propType 是值的离散值，例如要求组件指定日志级别的值：

code/appendix/proptypes/oneOf.js

```
class Component extends React.Component {
  static propTypes = {
    level: PropTypes.oneOf([
      'debug', 'info', 'warning', 'error'
    ])
  }
  // ...
  render() {
    return (
      <div>
        <p>{this.props.level}</p>
      </div>
    )
  }
}
```

使用 oneOfType()表示一个属性可以是多种类型之一。例如，电话号码可以作为字符串或整数传递给组件：

code/appendix/proptypes/oneOfType.js

```
class Component extends React.Component {
  static propTypes = {
```

```
    phoneNumber: PropTypes.oneOfType([
      PropTypes.number,
      PropTypes.string
    ])
  }
  // ...
  render() {
    return (
      <div>
        <p>{this.props.phoneNumber}</p>
      </div>
    )
  }
}
```

A.9 instanceOf

可以使用PropTypes.instanceOf()作为propType的值来规定组件**必须**是一个JavaScript类的实例：

code/appendix/proptypes/instanceOf.js

```
class Component extends React.Component {
  static propTypes = {
    user: PropTypes.instanceOf(User)
  }
  // ...
  render() {
    const { user } = this.props

    return (
      <div>
        <h3>{user.name}</h3>
      </div>
    )
  }
}
```

我们将使用 JavaScript 表达式语法来传递特定的属性。

code/appendix/proptypes/instanceOf.js

```
class User {
  constructor(name) {
    this.name = name
  }
}


const ari = new User('Ari');

<Component user={ari} />
```

A.10 array

有时需要传递一个数组作为属性。要设置数组，我们将使用 PropTypes.array 作为值：

code/appendix/proptypes/array.js

```
class Component extends React.Component {
  static propTypes = {
    authors: PropTypes.array
  }
  // ...
  render() {
    const { authors } = this.props
    return (
      <div>
        {authors && authors.map(author => {
          <AuthorCard author={author} />
        })}
      </div>
    )
  }
}
```

要发送一个对象作为属性，我们需要使用 JavaScript 表达式{}语法：

```
const users = [
  {name: 'Ari'}
  {name: 'Anthony'}
];

<Component authors={[{name: 'Anthony'}]} />
<Component authors={users} />
```

A.11 数组的类型

React 允许我们使用 PropTypes.arrayOf()来指定数组中每个成员应该使用的值类型：

code/appendix/proptypes/arrayOfType.js

```
class Component extends React.Component {
  static propTypes = {
    authors: PropTypes.arrayOf(PropTypes.object)
  }
  // ...
  render() {
    const { authors } = this.props
    return (
      <div>
        {authors && authors.map(author => {
          <AuthorCard author={author} />
        })}
      </div>
```

```
    )
  }
}
```

我们将使用 JavaScript 表达式{}语法来传递一个数组：

```
const users = [
  {name: 'Ari'}
  {name: 'Anthony'}
];

<Component authors={[{name: 'Anthony'}]} />
<Component authors={users} />
```

A.12　node

我们还可以传递任何可以渲染的内容，比如数字、字符串、DOM 元素、数组或者片段，可以使用 PropTypes.node 来包含它们：

code/appendix/proptypes/node.js

```
class Component extends React.Component {
  static propTypes = {
    icon: PropTypes.node
  }
  // ...
  render() {
    const { icon } = this.props
    return (
      <div>
        {icon}
      </div>
    )
  }
}
```

将**节点**作为属性传递也很简单。当要求组件具有子组件或设置自定义元素时，将节点作为值传递通常很有用。如果我们希望允许用户传递图标名称或自定义组件的名称，那么可以使用节点 propType。

```
const icon = <FontAwesomeIcon name="user" />

<Component icon={icon} />
<Component icon={"fa fa-cog"} />
```

A.13　element

React 的灵活性允许我们使用 PropTypes.element 来传递另一个 React 元素作为属性。

可以构建自己的组件，以允许用户能够通过组件的接口指定自定义组件。例如，我们可能有一个负责输出元素列表的<List />组件。如果没有自定义组件，我们将不得不为要渲染的每种类型的列表

构建一个单独的<List />React 组件（这可能是合适的，取决于元素的行为）。通过传递组件类型，我们可以重用<List />组件。

例如，列表组件可能看起来如下所示：

code/appendix/proptypes/element.js

```
class Component extends React.Component {
  static propTypes = {
    listComponent: PropTypes.element,
    list: PropTypes.array
  }
  // ...
  render() {
    const { list } = this.props
    return (
      <ul>
        {list.map(this.renderListItem)}
      </ul>
    )
  }
}
```

无论是否指定自定义组件，我们都可以使用此列表组件：

code/appendix/proptypes/element.js

```
const Item = function(props) {
  return (
    <div>{props.children}</div>
  )
}
```

```
<List list={[1, 2, 3]} />
<List list={[1, 2, 3]} listComponent={Item} />
```

A.14　any 类型

React 还允许我们指定一个属性必须存在，不管它是什么类型。可以使用 PropTypes.any 验证器来做到这一点：

code/appendix/proptypes/any.js

```
class Component extends React.Component {
  static propTypes = {
    mustBePresent: PropTypes.any
  }
  // ...
  render() {
    return (
      <div>
        Is here: {this.props.mustBePresent}
```

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

A.15　可选的 props 和必需的 props

除非另有说明，否则所有的属性都是可选的。要将一个属性传递给组件并进行验证，我们可以在每个 propType 验证后附加.isRequired。

如果有一个函数必须在组件加载完成后并且执行完一些操作后被调用，那么我们可以这样指定：

code/appendix/proptypes/optional.js

```
class Component extends React.Component {
  static propTypes = {
    // 可选属性
    onStart: PropTypes.func,
    // 必需属性
    onComplete: PropTypes.func.isRequired,
    name: PropTypes.string.isRequired
  }
  // ...
  startTimer = (seconds=5) => {
    const { onStart, onComplete } = this.props
    onStart()
    setTimeout(() => onComplete(), seconds)
  }
  // ...
  render() {
    const { name } = this.props
    return (
      <div onClick={this.startTimer}>
        {name}
      </div>
    )
  }
}
```

A.16　自定义验证器

React 允许我们为默认验证函数无法涵盖的所有其他情况指定自定义验证函数。为了编写自定义验证，我们将指定一个接收三个参数的函数：

(1) 传递给组件的 props；

(2) 被验证的 propName；

(3) 要验证的 componentName。

如果验证通过，那么我们就可以运行该函数并返回想要的内容。只有在 Error 对象被触发时验证

函数才会失败（例如 new Error()）。

如果我们有一个接受通过验证的用户的加载器，则可以对该属性运行一个自定义函数：

code/appendix/proptypes/custom.js

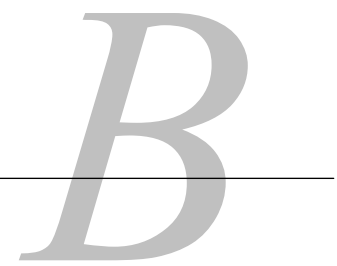
```
class Component extends React.Component {
  static propTypes = {
    user: function(props, propName, componentName) {
      const user = props[propName];
      if (!user.isValid()) {
        return new Error('Invalid user');
      }
    }
  }
  // ...
  render() {
    const { user } = this.props
    return (
      <div>
        {user.name}
      </div>
    )
  }
}
```

User 类可能看起来如下所示：

code/appendix/proptypes/custom.js

```
class User {
  constructor(name) {
    this.name = name
  }
  isValid() {
    // 必须包含名称
    return !!this.name && new Error('Name must be present')
  }
}
```

附录 B ES6



本附录是 ES6 添加到 JavaScript 的新语法特性和方法的非详尽列表。这些特性是非常常用且非常有帮助的。

虽然本附录没有涵盖 ES6 类，但我们在学习本书中的组件时会介绍这些基础知识。此外，本附录也不包括对一些更大的新特性（如 promise 和生成器）的描述。如果想要更多有关这些主题或以下任何主题的信息，建议参考 MDN Web Docs 网站。

B.1 首选 const 和 let 而不是 var

如果你以前使用过 ES5 JavaScript，那么可能会经常看到用 var 声明的变量：

appendix/es6/const_let.js

```
var myVariable = 5;
```

const 语句和 let 语句也都用来声明变量。它们是在 ES6 中引入的。

const 是在变量不会被重新分配值的情况下使用的。使用 const 可以让代码更加清晰。它表示变量在其定义的上下文中是“常量”状态。

如果变量的值要重新分配，则可以使用 let。

我们鼓励使用 const 和 let 来代替 var。除了 const 引入的限制外，const 和 let 都是**块作用域**而不是**函数作用域**。这个作用域可以帮助避免意外的错误。

B.2 箭头函数

编写箭头函数体有三种方法。假设我们有一个城市对象数组，如下所示：

appendix/es6/arrow_funcs.js

```
const cities = [
  { name: 'Cairo', pop: 7764700 },
  { name: 'Lagos', pop: 8029200 },
];
```

如果编写跨越多行的箭头函数，则必须使用大括号来分隔函数体，如下所示：

appendix/es6/arrow_funcs.js

```
const formattedPopulations = cities.map((city) => {
  const popMM = (city.pop / 1000000).toFixed(2);
  return popMM + ' million';
});
console.log(formattedPopulations);
// -> [ "7.76 million", "8.03 million" ]
```

请注意，我们还必须显式为函数指定一个返回值。

但是，如果我们编写的函数体只有一行（或一个表达式），则可以使用括号来分隔它：

appendix/es6/arrow_funcs.js

```
const formattedPopulations2 = cities.map((city) => (
  (city.pop / 1000000).toFixed(2) + ' million'
));
```

值得注意的是，我们没有使用 return，因为它是隐含着的。

此外，如果函数体很简洁，则可以这样写：

appendix/es6/arrow_funcs.js

```
const pops = cities.map(city => city.pop);
console.log(pops);
// [ 7764700, 8029200 ]
```

箭头函数的简洁性是我们使用它的两个原因之一。将上面的单行代码与下面的代码进行比较：

appendix/es6/arrow_funcs.js

```
const popsNoArrow = cities.map(function(city) { return city.pop });
```

不过，更大的好处是箭头函数绑定 this 对象的方式。

传统的 JavaScript 函数声明语法（function(){}）会在匿名函数中将 this 绑定到全局对象。为了说明这种情况引起的混淆，请考虑以下示例：

appendix/es6/arrow_funcs_jukebox_1.js

```
function printSong() {
  console.log("Oops - The Global Object");
}

const jukebox = {
  songs: [
    {
      title: "Wanna Be Startin' Somethin'",
      artist: "Michael Jackson",
    },
    {
      title: "Superstar",
      artist: "Madonna",
    },
```

```
  ],
  printSong: function (song) {
    console.log(song.title + " - " + song.artist);
  },
  printSongs: function () {
    // 'this'可以绑定到对象
    this.songs.forEach(function(song) {
      // 'this'无法绑定到全局对象
      this.printSong(song);
    });
  },
}

jukebox.printSongs();
// > "Oops - The Global Object"
// > "Oops - The Global Object"
```

printSongs()方法使用 forEach()对 this.songs 进行遍历。在这个上下文中，this 能按预期绑定到对象（jukebox），但传递给 forEach()的匿名函数会将其内部的 this 绑定到全局对象。因此，this.printSong(song)会调用在示例顶部声明的函数，而**不是**在 jukebox 上的方法。

传统上，JavaScript 开发人员在碰到此行为时会使用变通办法，但是箭头函数可以通过**捕获封闭上下文的 this 值**来解决这个问题。因此使用 printSongs()的箭头函数可以得到预期的结果，如下所示：

appendix/es6/arrow_funcs_jukebox_2.js

```
  printSongs: function () {
    this.songs.forEach((song) => {
      // 'this'会绑定到和 printSongs() (jukebox)相同的 this
      this.printSong(song);
    });
  },
}

jukebox.printSongs();
// > "Wanna Be Starting' Something' - Michael Jackson"
// > "Superstar - Madonna"
```

因此，在整本书中，我们对所有匿名函数都使用了箭头函数。

B.3　模块

ES6 正式支持使用 import/export 语法的模块。

命名 export

在任何文件中，你都可以使用 export 指定模块应公开的变量。下面是一个导出两个函数的文件示例：

```
// greetings.js

export const sayHi = () => (console.log('Hi!'));
export const sayBye = () => (console.log('Bye!'));

const saySomething = () => (console.log('Something!'));
```

现在可以在任何地方使用 import 来使用这些函数。我们需要指定要导入哪些函数。一种常见的方法是使用 ES6 的解构赋值语法来列出它们，如下所示：

```
// app.js

import { sayHi, sayBye } from './greetings';

sayHi(); // -> Hi!
sayBye(); // => Bye!
```

重要的是，未导出的函数（saySomething）在模块外部不可使用。

还需注意的是，我们给 from 提供了一个**相对路径**，表明该 ES6 模块是一个本地文件，而不是一个 npm 包。

可以使用以下语法在一个区域中列出所有公开的变量，而无须在要导出的每个变量之前插入 export 语句：

```
// greetings.js

const sayHi = () => (console.log('Hi!'));
const sayBye = () => (console.log('Bye!'));

const saySomething = () => (console.log('Something!'));

export { sayHi, sayBye };
```

还可以使用 import * as <Namespace>语法来指定在给定命名空间下要导入模块的所有功能：

```
// app.js

import * as Greetings from './greetings';

Greetings.sayHi();
  // -> Hi!
Greetings.sayBye();
  // => Bye!
Greetings.saySomething();
  // => TypeError: Greetings.saySomething is not a function
```

默认导出

另一种导出类型是默认导出，且一个模块只能包含一个默认导出：

```
// greetings.js

const sayHi = () => (console.log('Hi!'));
const sayBye = () => (console.log('Bye!'));
```

```
const saySomething = () => (console.log('Something!'));

const Greetings = { sayHi, sayBye };

export default Greetings;
```

这是一些库使用的常见模式。这意味着你可以轻松地批量导入库，而无须指定所需的每个函数：

```
// app.js

import Greetings from './greetings';

Greetings.sayHi(); // -> Hi!
Greetings.sayBye(); // => Bye!
```

模块同时使用命名导出和默认导出并不少见。例如，可以使用 react-dom 像下面这样导入 ReactDOM（默认导出）：

```
import ReactDOM from 'react-dom';

ReactDOM.render(
  // ...
);
```

或者，如果你只打算使用 render()函数，则可以像下面这样导入指定的 render()函数：

```
import { render } from 'react-dom';

render(
  // ...
);
```

为了实现这种灵活性，react-dom 的导出实现如下所示：

```
// 模仿的react-dom.js

export const render = (component, target) => {
  // ...
};

const ReactDOM = {
  render,
  // ……其他函数
};

export default ReactDOM;
```

如果你想玩一玩模块语法，请查看 code/webpack/es6-modules 文件夹。

有关 ES6 模块的更多信息，请参见 Mozilla 网站上的文章“ES6 In Depth—Modules”。

B.4　Object.assign()

在本书中我们经常使用 Object.assign()，并会在创建现有对象的修改版本的地方使用它。

Object.assign()接收任意数量的对象作为参数。当函数接收到两个参数时，它会将第二个对象的属性**复制**到第一个对象上，如下所示：

appendix/es6/object_assign.js

```
const coffee = { };
const noCream = { cream: false };
const noMilk = { milk: false };
Object.assign(coffee, noCream);
// coffee 的值现在是{ cream: false }
```

将三个参数传递给 Object.assign()是惯用的方式。第一个参数是一个新的 JavaScript 对象，且 Object.assign()最终将返回该对象；第二个是我们想要构建其属性的对象；最后一个是我们想要应用的变化：

appendix/es6/object_assign.js

```
const coffeeWithMilk = Object.assign({}, coffee, { milk: true });
// coffeeWithMilk 的值是{ cream: false, milk: true }
// coffee 的值没有变化：{ cream: false, milk: false }
```

Object.assign()是处理“不可变性”的 JavaScript 对象的便捷方法。

B.5　模板字面量

在 ES5 JavaScript 中，可以像下面这样在字符串中插入变量：

appendix/es6/template_literals_1.js

```
var greeting = 'Hello, ' + user + '! It is ' + degF + ' degrees outside.';
```

使用了 ES6 模板文字后，可以像下面这样创建相同的字符串：

appendix/es6/template_literals_2.js

```
const greeting = `Hello, ${user}! It is ${degF} degrees outside.`;
```

B.6　扩展操作符（...）

在数组中，省略号（...）操作符会将随后的数组**展开**为父数组。扩展操作符使我们能够简洁地将新数组构造为现有数组的组合。

下面是一个例子：

appendix/es6/spread_operator_arrays.js

```
const a = [ 1, 2, 3 ];
const b = [ 4, 5, 6 ];
const c = [ ...a, ...b, 7, 8, 9 ];

console.log(c); // -> [ 1, 2, 3, 4, 5, 6, 7, 8, 9 ]
```

注意看这和我们下面这样写有何不同：

appendix/es6/spread_operator_arrays.js

```
const d = [ a, b, 7, 8, 9 ];
console.log(d); // -> [ [ 1, 2, 3 ], [ 4, 5, 6 ], 7, 8, 9 ]
```

B.7　对象字面量增强

在 ES5 中，所有对象都需要有显式的键和值声明：

appendix/es6/enhanced_object_literals.js

```
const explicit = {
  getState: getState,
  dispatch: dispatch,
};
```

在 ES6 中，只要属性名和变量名相同，就可以使用下面这种更简洁的语法：

appendix/es6/enhanced_object_literals.js

```
const implicit = {
  getState,
  dispatch,
};
```

许多开源库都使用了这种语法，因此熟悉它是很有好处的。不过是否选择在自己的代码中使用它只是个人风格偏好的问题。

B.8　默认参数

使用 ES6，我们可以为参数指定一个默认值，以防在调用函数时它还没有被定义。

如下所示：

appendix/es6/default_args.js

```
function divide(a, b) {
  // divisor 默认值设为'1'
  const divisor = typeof b === 'undefined' ? 1 : b;

  return a / divisor;
}
```

可以写成这样：

appendix/es6/default_args.js

```
function divide(a, b = 1) {
  return a / b;
}
```

在这两种情况下，可以像下面这样使用该函数：

appendix/es6/default_args.js

```
divide(14, 2);
// => 7
divide(14, undefined);
// => 14
divide(14);
// => 14
```

如果上面示例中的参数 b 未定义，则会使用默认参数。注意，null 不会使用默认参数：

appendix/es6/default_args.js

```
divide(14, null); // divisor 会使用'null'值
// => 无穷大       // 14 / null
```

B.9　解构赋值

B.9.1　数组的解构赋值

在 ES5 中，从数组中提取和分配多个元素的过程如下所示：

appendix/es6/destructuring_assignments.js

```
var fruits = [ 'apples', 'bananas', 'oranges' ];
var fruit1 = fruits[0];
var fruit2 = fruits[1];
```

在 ES6 中，我们可以使用解构语法来完成相同的任务，如下所示：

appendix/es6/destructuring_assignments.js

```
const [ veg1, veg2 ] = [ 'asparagus', 'broccoli', 'onion' ];
console.log(veg1); // -> 'asparagus'
console.log(veg2); // -> 'broccoli'
```

左侧数组中的变量被“匹配”并分配给右侧数组中的相应元素。请注意，'onion'会被忽略，且不会绑定任何变量。

B.9.2　对象的解构赋值

可以执行类似的操作将对象属性提取到变量中：

appendix/es6/destructuring_assignments.js

```
const smoothie = {
  fats: [ 'avocado', 'peanut butter', 'greek yogurt' ],
  liquids: [ 'almond milk' ],
  greens: [ 'spinach' ],
  fruits: [ 'blueberry', 'banana' ],
```

```
};

const { liquids, fruits } = smoothie;

console.log(liquids); // -> [ 'almond milk' ]
console.log(fruits); // -> [ 'blueberry', 'banana' ]
```

B.9.3　参数上下文匹配

可以使用相同的原理将函数内的参数绑定到作为参数提供的对象的属性：

appendix/es6/destructuring_assignments.js

```
const containsSpinach = ({ greens }) => {
  if (greens.find(g => g === 'spinach')) {
    return true;
  } else {
    return false;
  }
};

containsSpinach(smoothie); // -> true
```

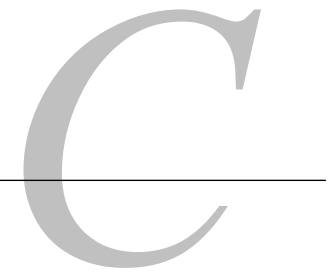
我们经常使用函数式 React 组件来执行此操作：

appendix/es6/destructuring_assignments.js

```
const IngredientList = ({ ingredients, onClick }) => (
  <ul className='IngredientList'>
    {
      ingredients.map(i => (
        <li
          key={i.id}
          onClick={() => onClick(i.id)}
          className='item'
        >
          {i.name}
        </li>
      ))
    }
  </ul>
)
```

在这里，我们使用解构赋值将属性提取到变量（`ingredients` 和 `onClick`）中，然后在组件的函数体内使用。

附录 C React Hook



本附录由 Yomi Eluwande 贡献。

如果你一直在阅读 Twitter，可能知道 Hook 是 React 的一个新特性，但你可能会问实际上要**如何使用它们**呢？本附录将向你展示一些关于如何使用 Hook 的例子。

要理解的一个关键思想是，**Hook 允许你在不编写类的情况下使用状态和其他 React 特性**。

C.1 警告：Hook 还不完善

在深入讨论之前，有一点要着重提一下，那就是 Hook API 还没有完成。

另外，它的官方文档非常好，尤其是因为它们扩展 Hook 的动机，我们建议你去阅读一下。

C.2 Hook 背后的动机

虽然基于组件的设计允许我们跨应用程序重用**视图**，但是 React 开发人员面临的最大问题之一是**如何重用组件之间的状态逻辑**。当我们拥有共享相似状态逻辑的组件时，如果没有好的重用解决方案，那么有时就会导致构造函数和生命周期方法中的逻辑重复。

传统上处理这种情况的典型方法如下所示：

- 使用高阶组件；
- 渲染复杂的属性。

但是这两种模式都有缺点，都可能导致代码变得复杂。

Hook 旨在解决所有这些问题，它使你能够编写可以访问诸如状态、上下文、生命周期方法、引用等特性的函数式组件，而不需要编写类组件。

C.3 Hook 如何映射到组件类

如果你熟悉 React，那么理解 Hook 的最佳方法之一就是通过使用 Hook 来查看我们重现“组件类”中惯用的行为的方式。

回想一下，在编写组件类时，我们经常需要做以下事情：

- 维护 state；
- 使用生命周期方法，如 componentDidMount()和 componentDidUpdate()；
- 访问上下文（通过设置 contextType）。

通过 React Hook，我们可以在函数式组件中复制类似或相同的行为。

- 组件状态使用 useState() Hook。
- 如 componentDidMount()和 componentDidUpdate()的生命周期方法会使用 useEffect() Hook。
- 静态 contextType 使用 useContext() Hook。

C.4　使用 Hook 需要 react "next"包

现在，你可以通过将 package.json 文件中的 react 和 react-dom 设置为 next 来开始使用 Hook。

```
// package.json
"react": "next",
"react-dom": "next"
```

C.5　useState() Hook 示例

状态是 React 的重要组成部分。它允许我们声明保存数据的状态变量，这些数据会在应用程序中使用。对于类组件，状态通常是这样定义的：

```
class Example extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      count: 0
    };
  }
```

在 Hook 之前，状态通常仅用于类组件中，但如上所述，Hook 允许我们将状态添加到函数式组件中。

让我们看下面的例子。这里将为灯泡 SVG 构建一个开关，它将根据状态的值改变颜色（见图 C-1）。为此，我们将使用 useState Hook。

下面是完整的代码（且是可运行的示例），我们将详细介绍下面的内容。

hooks/01-react-hooks-usestate/src/index.js

```
import React, { useState } from "react";
import ReactDOM from "react-dom";

import "./styles.css";

function LightBulb() {
  let [light, setLight] = useState(0);
```

```
  const setOff = () => setLight(0);
  const setOn = () => setLight(1);

  let fillColor = light === 1 ? "#ffbb73" : "#000000";

  return (
    <div className="App">
      <div>
        <LightbulbSvg fillColor={fillColor} />
      </div>

      <button onClick={setOff}>Off</button>
      <button onClick={setOn}>On</button>
    </div>
  );
}

function LightbulbSvg(props) {
  return (
    /*
      下面是一个灯泡形状的SVG 的标记代码。
      重要的是“填充”部分，这里会根据 props 动态设置颜色
    */
    <svg width="56px" height="90px" viewBox="0 0 56 90" version="1.1">
      <defs />
      <g
        id="Page-1"
        stroke="none"
        stroke-width="1"
        fill="none"
        fill-rule="evenodd"
      >
        <g id="noun_bulb_1912567" fill="#000000" fill-rule="nonzero">
          <path
            d="M38.985,68.873 L17.015,68.873 C15.615,68.873 14.48,70.009 14.48,71.40\
9 C14.48,72.809 15.615,73.944 17.015,73.944 L38.986,73.944 C40.386,73.944 41.521,72.\
809 41.521,71.409 C41.521,70.009 40.386,68.873 38.985,68.873 Z"
            id="Shape"
          />
          <path
            d="M41.521,78.592 C41.521,77.192 40.386,76.057 38.986,76.057 L17.015,76.\
057 C15.615,76.057 14.48,77.192 14.48,78.592 C14.48,79.993 15.615,81.128 17.015,81.1\
28 L38.986,81.128 C40.386,81.127 41.521,79.993 41.521,78.592 Z"
            id="Shape"
          />
          <path
            d="M18.282,83.24 C17.114,83.24 16.793,83.952 17.559,84.83 L21.806,89.682\
C21.961,89.858 22.273,90 22.508,90 L33.492,90 C33.726,90 34.039,89.858 34.193,89.68\
2 L38.44,84.83 C39.207,83.952 38.885,83.24 37.717,83.24 L18.282,83.24 Z"
            id="Shape"
          />
          <path
            d="M16.857,66.322 L39.142,66.322 C40.541,66.322 41.784,65.19 42.04,63.81\
```

```
4 C44.63,49.959 55.886,41.575 55.886,27.887 C55.887,12.485 43.401,0 28,0 C12.599,0 0\
.113,12.485 0.113,27.887 C0.113,41.575 11.369,49.958 13.959,63.814 C14.216,65.19 15.\
458,66.322 16.857,66.322 Z"
            id="Shape"
            fill={props.fillColor}
          />
        </g>
      </g>
    </svg>
  );
}

const rootElement = document.getElementById("root");
ReactDOM.render(<LightBulb />, rootElement);
```

图 C-1　灯泡按钮

C.5.1　我们的组件是一个函数

在上面的代码块中，首先从 react 导入 useState。useState 是一个使用 this.state 提供的功能的新方法。

接下来，请注意该组件**是函数而非类**。有趣吧！

C.5.2　状态读写

在这个函数中，我们调用 useState 来创建一个状态变量：

```
let [light, setLight] = useState(0);
```

useState 用于声明状态变量，并可以使用**任何**类型的值进行初始化（不像类中的状态，类型必须是一个对象）。

如上所示，我们对 useState 的返回值使用解构赋值。

- 第一个值（在本例中是 light）是当前状态（有点像 this.state）。
- 第二个值是**一个用于更新状态（第一个值）的函数**（类似于传统的 this.setState）。

接下来创建两个函数，每个函数会将状态设置为不同的值（0 或 1）：

```
const setOff = () => setLight(0);
const setOn = () => setLight(1);
```

然后将这些函数用作视图中按钮的事件处理程序：

```
<button onClick={setOff}>Off</button>
<button onClick={setOn}>On</button>
```

C.5.3　React 状态跟踪

当“On”按钮被按下时，setOn 函数会被调用，接着 setOn 会调用 setLight(1)。对 setLight(1) 的调用将**在下一次渲染时更新 light 的值**。这听起来有些不可思议，但实际的情况是 **React 正在跟踪该变量的值**，并且在重新渲染该组件时它会传入新值。

然后使用当前状态（light）来确定灯泡是否应该“打开”。也就是说，根据 light 的值设置 SVG 的填充颜色。如果 light 值为 0（关闭状态），那么 fillColor 值设置为#000000；如果 light 值为 1（打开状态），那么 fillColor 值设置为#ffbb73。

C.5.4　多个状态

虽然上面的示例中没有这样做，但你可以通过多次调用 useState 创建多个状态。如下所示：

```
let [light, setLight] = useState(0);
let [count, setCount] = useState(10);
let [name, setName] = useState("Yomi");
```

注意：在使用 Hook 时应注意一些限制。最重要的一点是，你**只能在函数的顶层调用 Hook**。更多信息，请参见 React 网站上的文章“Rules of Hooks”。

C.6　useEffect() Hook 示例

useEffect() Hook 允许你在函数式组件中执行副作用。副作用可以是 API 调用、更新 DOM 和订阅事件监听器等——任何你想要执行“命令”操作的地方。

通过使用 useEffect() Hook，React 知道你希望在渲染完成后执行某个特定的操作。

让我们看下面的例子。我们将使用 useEffect() Hook 来进行 API 调用并获取响应（效果见图 C-2）。

hooks/02-react-hooks-useeffect/src/index.js

```
import React, { useState, useEffect } from "react";
import ReactDOM from "react-dom";

import "./styles.css";

function App() {
  let [names, setNames] = useState([]);

  useEffect(() => {
    fetch("https://uinames.com/api/?amount=25&region=nigeria")
      .then(response => response.json())
      .then(data => {
        setNames(data);
      });
```

```
  }, []);

  return (
    <div className="App">
      <div>
        {names.map((item, i) => (
          <div key={i}>
            {item.name} {item.surname}
          </div>
        ))}
      </div>
    </div>
  );
}

const rootElement = document.getElementById("root");
ReactDOM.render(<App />, rootElement);
```

图 C-2　使用 useEffect 获取数据

在这个示例代码中，useState 和 useEffect 均被导入，这是因为我们希望**将 API 调用的结果设置为一个状态**。

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

C.6.1　获取数据并更新状态

为了“使用副作用”，我们需要将动作放到 useEffect 函数中。也就是说，我们需要将副作用的“动作”作为一个匿名函数传递给 useEffect 的第一个参数。

在上面的示例中，我们对返回名称列表的端点进行了 API 调用。当 response 返回时，我们会将其转换为 JSON，然后使用 setNames(data)设置状态。

```
let [names, setNames] = useState([]);

useEffect(() => {
  fetch("https://uinames.com/api/?amount=25&region=nigeria")
    .then(response => response.json())
    .then(data => {
      setNames(data);
    });
}, []);
```

C.6.2　使用副作用时的性能问题

关于使用 useEffect，有一些事情需要注意。

首先要考虑的是，默认情况下 useEffect 会在每次渲染时被调用！好消息是无须担心过时的数据，但坏消息是我们可能不希望在每次渲染时都发出 HTTP 请求（在本例中）。

可以通过**使用 useEffect 的第二个参数**来跳过副作用，就像我们在本例中所做的那样。useEffect 的第二个参数是我们要“观察”的变量列表，然后仅当其中一个值发生变化时才会重新运行副作用。

在上面的示例代码中，注意我们传递了**一个空数组**作为第二个参数。这是用于告诉 React，我们只想在组件挂载时调用此副作用。

要了解更多关于副作用性能的信息，请在 React 网站官方文档“Using the Effect Hook”中查看此部分。

另外，就像上面的 useState 函数一样，useEffect 允许多个实例，这意味着你可以拥有多个 useEffect 函数。

C.7　useContext() Hook 示例

C.7.1　上下文的意义

React 中的上下文是子组件访问父组件中的值的一种方式。

为了理解上下文的使用场景，在构建 React 应用程序时，你通常需要从 React 树的顶部到底部获取值。如果没有上下文，你最终会通过不需要使用 props 的组件来传递它们。将 props 传递给不需要它们的组件不仅很麻烦，而且如果操作不当还会无意中引入耦合。

将 props 从“不相关”的组件树中向下传递，被亲切地称为 **props 钻取**。

React 上下文允许你通过组件树与任何请求这些值的组件来共享值，从而解决了 props 钻取的问题。

C.7.2　useContext()使上下文更易于使用

通过使用 useContext Hook，使用上下文将比以往任何时候都更容易。

useContext()函数接收一个**上下文对象**，该对象最初从 React.createContext()返回，接着它会

返回当前上下文的值。让我们看下面的例子（效果见图 C-3）。

hooks/03-react-hooks-usecontext/src/index.js

```
import React, { useContext } from "react";
import ReactDOM from "react-dom";
import "./styles.css";

const JediContext = React.createContext();

function Display() {
  const value = useContext(JediContext);
  return <div>{value}, I am your Father.</div>;
}

function App() {
  return (
    <JediContext.Provider value={"Luke"}>
      <Display />
    </JediContext.Provider>
  );
}

const rootElement = document.getElementById("root");
ReactDOM.render(<App />, rootElement);
```

图 C-3　上例的效果

在上面的代码中，上下文 JediContext 是使用 React.createContext()创建的。

我们在 App 组件中使用了 JediContext.Provider，并将其中的 value 设置为"Luke"。这意味着现在树中的任何读取上下文的对象都可以读取该值。

要在 Display()函数中读取这个值，我们需要将 JediContext 作为参数传入来调用 useContext。

然后传入从 React.createContext 获得的上下文对象，它会自动输出值。当 provider 的值更新时，此 Hook 将使用最新上下文的值来触发重新渲染。

C.7.3　在更大的应用程序中获取对上下文的引用

上面，我们在两个组件的作用域内创建了 JediContext，但在更大的应用程序中，Display 和 App 组件将位于不同的文件中。因此，如果你跟我们一样，可能也会想知道："怎样才能跨文件引用 JediContext？"

答案是你需要**创建一个导出 JediContext 的新文件**。

例如，你可能有一个 context.js 文件，其内容如下：

```
const JediContext = React.createContext();
export { JediContext };
```

然后在 App.js（和 Display.js）中编写如下代码：

```
import { JediContext } from "./context.js";
```

C.8　useRef() Hook 示例

引用提供了一种访问在 render()方法中创建的 React 元素的方法。

如果你是 React 引用的新手，可以阅读 Fullstack React 网站的文章“Fullstack React's Guide to Using Refs in React Components”，了解关于 React 引用的介绍。

useRef()函数会返回一个引用对象。

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

useRef()和带有 input 的表单

让我们来看一个使用 useRef Hook 的例子（效果见图 C-4）。

hooks/04-react-hooks-useref/src/index.js

```
import React, { useState, useRef } from "react";
import ReactDOM from "react-dom";

import "./styles.css";

function App() {
  let [name, setName] = useState("Nate");

  let nameRef = useRef();

  const submitButton = () => {
    setName(nameRef.current.value);
  };

  return (
    <div className="App">
      <p>{name}</p>

      <div>
        <input ref={nameRef} type="text" />
        <button type="button" onClick={submitButton}>
          Submit
        </button>
      </div>
    </div>
```

```
  );
}

const rootElement = document.getElementById("root");
ReactDOM.render(<App />, rootElement);
```

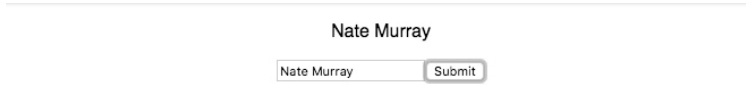


图 C-4　useRef

在上面的例子中，我们将 useRef() Hook 与 useState()结合使用，以将 input 标签的值渲染到 p 标签中。

该引用被实例化为 nameRef 变量。然后可以将 nameRef 变量设置为 ref，从而在输入字段中使用它。本质上，这意味着现在可以通过引用访问输入字段的内容。

代码中的提交按钮有一个 onClick 事件处理程序，名为 submitButton。submitButton 函数将调用 setName（它是通过 useState 创建的）。

正如我们之前对 useState Hook 所做的那样，setName 将用于设置 name 状态。要从 input 标签中提取名称，我们需要读取 nameRef.current.value 的值。

关于 useRef 需要注意的另一件事是，它可用于 ref 属性之外的其他属性。

C.9　使用自定义 Hook

Hook 最酷的特性之一是我们可以**通过创建自定义的 Hook 来轻松地在多个组件之间共享逻辑**。

在下面的例子中，我们将创建一个自定义的 setCounter() Hook，它允许我们跟踪状态并提供自定义的状态更新函数（效果见图 C-5）!

另外，请参阅 react-use 的 useCounter Hook 和 Kent 的 useCounter Hook。

hooks/05-react-hooks-custom-hooks/src/index.js

```
import React, { useState } from "react";
import ReactDOM from "react-dom";
import "./styles.css";

function useCounter({ initialState }) {
  const [count, setCount] = useState(initialState);
  const increment = () => setCount(count + 1);
  const decrement = () => setCount(count - 1);
  return [count, { increment, decrement, setCount }];
}

function App() {
```

```
  const [myCount, { increment, decrement }] = useCounter({ initialState: 0 });
  return (
    <div>
      <p>{myCount}</p>
      <button onClick={increment}>Increment</button>
      <button onClick={decrement}>Decrement</button>
    </div>
  );
}

const rootElement = document.getElementById("root");
ReactDOM.render(<App />, rootElement);
```

图 C-5　自定义递增 Hook

在上面的代码块中，我们创建了一个 useCounter 函数，它存储了自定义 Hook 的逻辑。

请注意 useCounter 可以使用其他 Hook！我们首先通过 useState 创建一个新的状态 Hook。

接下来定义两个辅助函数——increment 和 decrement，它们会调用 setCount 并相应地调整当前计数。

最后返回与 Hook 交互所必需的引用。

问：返回值是数组的对象么？

答：就像 Hook 中的大多数东西一样，API 约定还没有最终确定。不过我们在这里做的是返回一个数组：

- 第一项是 Hook 的**当前值**；
- 第二项是一个对象，包含用于与 Hook 交互的函数。

这个约定允许我们轻松地“重命名”Hook 的当前值，就像在上面对 myCount 所做的那样。

也就是说，可以从自定义 Hook 中返回任何想要的东西。

在上面的例子中，我们使用 increment 和 decrement 作为视图中的 onClick 处理程序。当用户按下按钮时，计数器将更新并在视图中作为 myCount 重新显示。

C.10　为 React Hook 编写测试

为了编写 Hook 的测试，我们将使用 react-testing-library 对其进行测试。

react-testing-library 是一个非常轻量级的测试 React 组件的解决方案。它扩展了 react-dom 和 react-dom/test-utils，以提供轻量级的实用程序功能。使用 react-testing-library 可确保你的测试直接在 DOM 节点上进行。

在撰写本文时，关于 Hook 的测试还有些不成熟。你目前还不能单独测试一个 Hook，而是需要将 Hook 附加到组件并测试该组件。

因此，在下面我们将通过**组件**而不是直接与 Hook 交互来编写测试。好消息是，我们的测试看起来会像普通的 React 测试一样。

C.10.1　为 useState() Hook 编写测试

让我们来看一个为 useState() Hook 编写测试的例子。在上面的课程中，我们测试了上面使用的 useState 示例的更多变体。我们将编写测试，以确保点击“Off”按钮时将该状态设置为 0，而点击“On”按钮将该状态设置为 1。

hooks/06-tests-for-usestate/src/__tests__testhooks.js

```
import React from "react";
import { render, fireEvent, getByTestId } from "react-testing-library";
// 导入 LightBulb 组件
import LightBulb from "../index";

test("bulb is on", () => {
  // 获取已渲染的 React 元素包含的 DOM 节点
  const { container } = render(<LightBulb />);
  // p 标签包含 LightBulb 组件的当前状态值
  const lightState = getByTestId(container, "lightState");
  // 这引用了 On 按钮
  const onButton = getByTestId(container, "onButton");
  // 这引用了 Off 按钮
  const offButton = getByTestId(container, "offButton");

  // 模拟点击 On 按钮
  fireEvent.click(onButton);
  // 期望测试的状态值为 1
  expect(lightState.textContent).toBe("1");
  // 模拟点击 Off 按钮
  fireEvent.click(offButton);
  // 期望测试的状态值为 0
  expect(lightState.textContent).toBe("0");
});
```

在上面的代码块中，我们首先从 react-testing-library 和要测试的组件中导入一些帮助程序。

- render：它将帮助我们渲染组件，并将其渲染到附加于 document.body 的容器中。

- getByTestId：可以通过 data-testid 获取 DOM 元素。
- fireEvent：用于“触发”DOM 事件。它在 document 元素上附加了一个事件处理程序，并通过事件委托（例如点击一个按钮）来处理一些 DOM 事件。

接下来，在测试断言函数中，我们为 data-testid 元素创建常量变量，并在测试中使用它们的值。通过引用 DOM 上的元素，我们可以使用 fireEvent 方法来模拟点击按钮。

该测试会检查如果点击了 onButton，则状态设置为 1，当点击 offButton 时，状态则设置为 0。

C.10.2　为 useEffect() Hook 编写测试

在此示例中，我们将编写测试来使用 useEffect() Hook 将一个商品添加到购物车中。该商品的数量也会存储在 localStorage 中。在下面 CodeSandbox 中的 index.js 文件包含了用于向购物车添加商品的实际逻辑。

我们将进行测试，以确保更新了购物车中的商品数量也会反映在 localStorage 中，即使重新加载页面，购物车中的商品数量仍然保持不变。

hooks/07-tests-for-useeffect/src/tests/testhooks.js

```
import React from "react";
import { render, fireEvent, getByTestId } from "react-testing-library";
// 导入 App 组件
import App from "../index";

test("cart item is updated", () => {
  // 将 localStorage 计数设置为 0
  window.localStorage.setItem("cartItem", 0);
  // 获取已渲染的 React 元素包含的 DOM 节点
  const { container, rerender } = render(<App />);
  // 引用增加购物车商品数量的增加按钮
  const addButton = getByTestId(container, "addButton");
  // 引用重置按钮，重置购物车商品数量
  const resetButton = getByTestId(container, "resetButton");
  // 引用显示购物车商品数量的 p 标签
  const countTitle = getByTestId(container, "countTitle");
  // 模拟点击增加按钮
  fireEvent.click(addButton);
  // 测试期望购物车商品数量为 1
  expect(countTitle.textContent).toBe("Cart Item - 1");
  // 模拟重新加载应用程序
  rerender(<App />);
  // 测试仍然期望购物车商品计数为 1
  expect(window.localStorage.getItem("cartItem")).toBe("1");
  // 模拟点击重置按钮
  fireEvent.click(resetButton);
  // 测试期望购物车商品数量为 0
  expect(countTitle.textContent).toBe("Cart Item - 0");
});
```

在测试断言函数中，我们首先将 localStorage 中的 cartItem 设置为 0，这意味着购物车商品数

量为 0。然后通过解构赋值从 App 组件获取 container 和 rerender。rerender 允许我们模拟重新加载页面。

接下来获得对按钮和 p 标签的引用，用于显示当前购物车商品数量，并将它们设置为常量变量。

完成后，该测试将模拟点击 addButton 并检查当前购物车商品数量是否为 1；然后重新加载页面，并检查 localStorage 中的计数（cartItem）是否也设置为 1。最后模拟点击 resetButton 并检查当前购物车商品数量是否设置为 0。

C.10.3　为 useRef() Hook 编写测试

在此示例中，我们将测试 useRef() Hook，并使用上面的原始 useRef 示例作为测试的基础。useRef Hook 用于从 input 字段获取值，然后将其设置为状态值。在下面 CodeSandbox 中的 index.js 文件包含了输入一个值并提交它的逻辑。

hooks/08-tests-for-useref/src/tests/testhooks.js

```
import React from "react";
import { render, fireEvent, getByTestId } from "react-testing-library";
// 导入 App 组件
import App from "../index";

test("input field is updated", () => {
  // 获取已渲染的 React 元素包含的 DOM 节点
  const { container } = render(<App />);
  // 引用 input 字段
  const inputName = getByTestId(container, "nameinput");
  // 引用 p 标记，用于显示从 ref 中获得的值
  const name = getByTestId(container, "name");
  // 引用提交按钮，用于将状态值设置为 ref 值
  const submitButton = getByTestId(container, "submitButton");
  // 要在 input 字段中输入的值
  const newName = "Yomi";

  // 模拟在 input 字段中输入"Yomi"值
  fireEvent.change(inputName, { target: { value: newName } });
  // 模拟点击提交按钮
  fireEvent.click(submitButton);
  // 测试期望 name 引用的值显示等于 input 字段中输入的值
  expect(name.textContent).toEqual(newName);
});
```

在测试断言函数中，我们将输入字段、显示当前 ref 值的 p 标签和提交按钮设置为常量变量。我们还将希望在输入字段中输入的值设置为 newName 常量变量，用于在测试中进行检查。

```
fireEvent.change(inputName, { target: { value: newName } });
```

fireEvent.change()方法用于在输入字段中输入值，在本例中，它使用了存储在 newName 常量变量中的名称。在这之后再点击提交按钮。

然后该测试会检查在按钮被点击后引用的值是否等于 newName。

最后，你应该会在控制台中看到一条 `There are no failing tests, congratulations!`（没有失败的测试，恭喜你！）的消息。

C.11　社区对 Hook 的反应

自从 React Hook 的消息发布以来，社区对这个特性很感兴趣，我们已看到了很多关于 React Hook 的例子和用例。以下是一些亮点。

- React 网站文章“Collection of React Hooks”展示了 React Hook 的系列产品。
- `react-use` 是一个带有大量 React Hook 的库。
- 这个 CodeSandbox 示例[①]说明了如何使用 `useEffect` Hook 和 `react-spring` 来创建动画。
- 一个 `useMutableReducer` Hook 的示例[②]，使你只需要在 `reducer` 中修改状态即可对它进行更新。
- 这个 CodeSandbox 示例[③]展示了子级和父级通信的复杂用法以及 reducer 的用法。
- 一个使用 React Hook 构建的切换组件[④]。
- React Hook 的另一个集合[⑤]，具有用于输入值、设备方向和文档可见性的 Hook。

C.12　不同 Hook 类型的引用

可以在 React 代码中开始使用各种类型的 Hook，如下所示。

- `useState`——允许我们编写带有状态的纯函数。
- `useEffect`——让我们可以执行副作用。副作用可能是 API 调用、更新 DOM、订阅事件监听器等。
- `useContext`——允许我们编写带有上下文的纯函数。
- `useReducer`——为我们提供了一个类 Redux reducer 的引用。
- `useRef`——允许我们编写返回可变 `ref` 对象的纯函数。
- `useMemo`——用于返回一个已存储的值。
- `useCallback`——用于返回一个已存储的回调。
- `useImperativeMethods`——用于定制在使用 `ref` 时向父组件暴露的实例值。
- `useMutationEffects`——与 `useEffect` 类似，因为它允许执行 DOM 变更。
- `useLayoutEffect`——用于从 DOM 读取布局并同步重新渲染。
- 自定义 `Hook`——允许我们将组件逻辑写入可重用的函数中。

---

① 参见 CodeSandbox 网站的 ppxnl191zx 页面。
② 参见 GitHub Gist 网站的 aweary/App.js 页面。
③ 参见 CodeSandbox 网站的 y570vn3v9 页面。
④ 参见 CodeSandbox 网站的 m449vyk65x 页面。
⑤ 参见 React 网站。

C.13　Hook 的未来

Hook 的伟大之处在于，它可以与现有代码并行工作，这样你就可以慢慢地进行采用 Hook 的更改。你所要做的就是将 React 的依赖项升级到具有 Hook 特性的版本。

尽管如此，**Hook 仍然是一个实验性的特性**，React 团队已多次警告说可能会对 API 进行修改。这一点需要你自己考虑清楚。

Hook 的出现对类意味着什么？根据 React 团队的说法，类仍会存在，因为它们是 React 代码库的重要组成部分，并且很可能还会存在一段时间。

> 我们没有废弃类的计划。在 Facebook，我们有成千上万的类组件，并且像你一样，我们不打算重写它们。但是，如果 React 社区支持 Hook，那么采用两种不同的推荐方式来编写组件是没有意义的。——Dan Abramov

虽然特定的 Hook API 如今仍处于试验阶段，但社区喜欢 Hook 的思想，因此我们预计它不会很快消失。

很高兴能在应用程序中使用 Hook！

C.14　更多资源

- React 团队在 React Hook 的文档方面做得非常出色，你可以 React 网站了解更多内容。
- 有关官方 API 参考资料，参见 React 网站文档“Hooks API Reference”。
- GitHub 网站 reactjs/rfcs 页面有一个正在进行的 RFC，你可以直接去那里提问或发表评论。

更新日志

修订 39 - 2019-01-10

- 增加了一个关于 Hook 的（实验性的）新章节。

修订 38 - 2018-12-20

- 更新图书以支持 React 16.7.0。

修订 37 - 2018-12-19

- 更新图书以支持 React 16.6.3。
- 恢复“Relay”一章为第 15 章。

修订 36 - 2018-10-01

- 内部发布。

修订 35 - 2018-04-02

- 更新以支持 React 16.3.1。
- 较小的代码修复。
- 移除“Relay”一章。

修订 34 - 2017-10-17

- 更新以支持 React 16。
- 更正输入错误。
- 较小的代码修复。

修订 33 - 2017-08-31

- 更正输入错误。
- 较小的代码修复。

修订 32 - 2017-06-14

- 修复了 Spotify API 更改在第 9 章中引入的错误。
- 更新所有终端命令，使它们与 Windows PowerShell 兼容。
- 语言和格式修复。

修订 31 - 2017-05-18

- 更新图书以支持 15.5.4。

- 使用 `prop-types` 包中的 `PropTypes`。
- 将特定于 ES6 的材料移至附录。
- 语言和 bug 修复。

修订 30 - 2017-04-20

- 添加“如何充分利用这本书”。
- 使用 Create React App 更新 Redux 应用程序。
- 更正输入错误。

修订 29 - 2017-04-13

- 第 6 章修复了对 `this.state` 的修改和其他错别字的问题。
- 将第 5 章更新到 ES6 类。
- 将“JSX”更新到 ES6 类。

修订 28 - 2017-04-10

- 更新第 9 章以支持 `react-router v4.0.0`。
- 小的 bug 修复和拼写错误更正。

修订 27 - 2017-03-16

- 修复第 5 章中的依赖关系问题。
- 修复“Relay”一章中的格式校验错误。

修订 26 - 2017-02-22

- 增加了 Christopher Chedeau（@vjeux）的序。
- 更正输入错误。
- 更新关于第 2 章中的属性初始化器的讨论。

修订 25 - 2017-02-17

- 增加“Relay”一章。

修订 24 - 2017-02-08

- 更新 Redux 相关章节以使用 ES6 类组件。

修订 23 - 2017-02-06

- 更新“Webpack”和“单元测试”。
  - 使用 ES6 类组件。
  - `.babelrc` 讨论。
- 修复第 9 章的错误。

修订 22 - 2017-02-01

- 更新第 1 章。
  - 使用属性初始化器。

- 更深入的讨论 Babel 和 Babel 插件。

- 更新第 2、3 章。
  - 使用 ES6 类组件。
- 更新 Redux 一章。
  - 图像大小。

修订 21 - 2017-01-27

- 更新第 1 章。
  - 使用 ES6 类组件。
  - 改进了关于不变性的讨论。

修订 20 - 2017-01-10

- 增加了“React Native”一章。

修订 19 - 2016-12-20

- 增加了第 9 章。

修订 18 - 2016-11-22

- 改进了 Windows 支持。
- 更新代码到 `react-15.4.1`。
- bug 修复。

修订 17 - 2016-11-04

- 更新代码，使其可以在没有 Cygwin（PowerShell）的 Windows 中使用。
- 在书中添加了 Windows 安装说明。
- 替换 Babel 版本。

修订 16 - 2016-10-12

- 增加了第 8 章。

修订 15 - 2016-10-05

- 增加了第 7 章。

修订 14 - 2016-08-26

- 增加了第 6 章。

修订 13 - 2016-08-02

- 更新代码到 `react-15.3.0`。

修订 12 - 2016-07-26

- 更新代码到 `react-15.2.1`。

修订 11 - 2016-07-08

- 更新代码到 `react-15.2.0`。

修订 10 - 2016-06-24

- 更新代码到 `react-15.1.0`。

修订 9 - 2016-06-21

- 增加了“编写 GraphQL 服务器”一章。

修订 8 - 2016-06-02

- 增加了第 11 章。
- 增加“使用 Redux 表示和容器组件”一章。

修订 7 - 2016-05-13

- 增加了第 13 章。
- 更新代码到 `react-15.0.2`。

修订 6 - 2016-05-13

- 增加了“配置组件”一章。
- 增加了附录 A。

修订 5 - 2016-04-25

- 增加了第 10 章。

修订 4 - 2016-04-22

- 增加了第 4 章。

修订 3 - 2016-04-08

- 更新代码到 `react-15.0.1`。

修订 2 - 2016-03-16

- 对第 1 章进行了重要的重写，以使其思路更清晰/更好。
- 各种修正。
- 改进代码风格。
- 更新图表。

修订 1 - 2016-02-14

本书的最初版本包含：

- 第 1 章，第一个 React Web 应用程序；
- 第 2 章，组件；
- 第 3 章，组件与服务器。