

```
let {a = [1, 2, 3]} = {};
a // [1, 2, 3]
let [x, y = 'hi'] = ["a"];
x // x='a', y='b'
```

3. 拓展运算符 (spread) ...

拓展运算符 (spread) 是 3 个点 (...)。可以将它比作 rest 参数的逆运算，将一个数组转为用逗号分隔的参数序列。下面来看看它有哪些作用。

(1) 合并数组。

在 ES 5 中要合并两个数组，写法是这样的：

```
var a = [1, 2];
var b = [3, 4];
a.concat(b); // [1, 2, 3, 4]
```

但在 ES 6 中拓展运算符提供了合并数组的新写法：

```
let a = [1, 2];
let b = [3, 4];
[...a, ...b]; // [1, 2, 3, 4]
```

如果想让一个数组添加到另一个数组后面，在 ES 5 中是这样写的：

```
var x = ["a", "b"];
var y = ["c", "d"];
Array.prototype.push.apply(arr1, arr2);
arr1 // ["a", "b", "c", "d"]
```

上述代码中由于 push()方法不能为数组，所以通过 apply()方法变相使用。但现在有了 ES 6 的拓展运算符后就可以直接使用 push()方法了：

```
let x = ["a", "b"];
let y = ["c", "d"];
x.push(...y); // ["a", "b", "c", "d"]
```

(2) 数组复制。

拓展运算符还可以用于数组复制，但要注意的是，复制的是指向底层数据结构的指针，并非复制一个全新的数组。

数组复制示例：

```
const x = ['a', 'b'];
const x1 = x;
x1[0]; // 'a'
x1[0] = 'c';
x // ['c', 'b']
```

(3) 与解构赋值结合。

拓展运算符可以和解构赋值相结合用于生成新数组：

```
const [arr1, ...arr2] = [1, 2, 3, 4];
arr1 // 1
arr2 // [2, 3, 4]
```

注意，使用拓展运算符给数组赋值时，必须放在参数最后的位置，不然会报错。例如：

```
const [...arr1, arr2] = [1, 2, 3, 4];           // 报错
const [1, ...arr1, arr2] = [1, 2, 3, 4];       // 报错
```

(4) 函数调用（替代 apply()方法）。

在 ES 5 中要合并两个数组，写法是这样的：

```
function add(a, b) {
  return a + b;
}
const num = [1, 10];
add.apply(null, num); // 11
```

在 ES 6 中可以这样写：

```
function add(a, b) {
  return a + b;
}
const num = [1, 10];
add(...num); // 11
```

上述代码使用拓展运算符将一个数组变为参数序列。当然，拓展运算符也可以和普通函数参数相结合使用，非常灵活。比如：

```
function add(a, b, c, d) {
  return a + b + c + d;
}
const num = [1, 10];
add(2, ...num, -2); // 11
```

拓展运算符中的表达式如下：

```
[...(true ? [1, 2] : [3]), 'a']; // [1, 2, 'a']
```

4. 箭头函数

ES 6 对于函数的拓展中增加了箭头函数`=>`，用于对函数的定义。

箭头函数语法很简单，先定义自变量，然后是箭头和函数主体。箭头函数相当于匿名函数并简化了函数定义。

不引入参数的箭头函数示例：

```
var sum = () => 1+2; // 圆括号代表参数部分
// 等同于
var sum = function() {
  return 1 + 2;
}
```

引入参数的箭头函数示例：

```
// 单个参数
var sum = value => value; // 可以不给参数 value 加小括号
// 等同于
var sum = function(value) {
  return value;
};
// 多个参数
var sum = (a, b) => a + b;
```

```
// 等同于
var sum = function(a, b) {
  return a + b;
};
```

花括号{}内的函数主体部分写法基本等同于传统函数写法。

 注意：如果箭头函数内要返回自定义对象，需要用小括号把对象括起来。例如：

```
var getInfo = id => ({
  id: id,
  title: 'Awesome React'
});
// 等同于
var getInfo = function(id) {
  return {
    id: id,
    title: 'Awesome React'
  }
}
```

箭头函数与传统的 JavaScript 函数主要区别如下：

- 箭头函数内置 this 不可改变；
- 箭头函数不能使用 new 关键字来实例化对象；
- 箭头函数没有 arguments 对象，无法通过 arguments 对象访问传入的参数。

这些差异的存在是有理可循的。首先，对 this 的绑定是 JavaScript 错误的常见来源之一，容易丢失函数内置数值，或得出意外结果；其次，将箭头函数限制为使用固定 this 引用，有利于 JavaScript 引擎优化处理。

箭头函数看似匿名函数的简写，但与匿名函数有明显区别，箭头函数内部的 this 是词法作用域，由上下文确定。如果使用了箭头函数，就不能对 this 进行修改，所以用 call() 或 apply() 调用箭头函数时都无法对 this 进行绑定，传入的第一个参数会被忽略。

更多详情，可参考阮一峰的《ECMAScript 6 入门》一书。

 注意：词法作用域是定义在词法阶段的作用域，它在代码书写的时候就已经确定。

1.2 React 简介

A JavaScript library for building user interfaces 这是 React 官网给 React 的一句话概括。

简单来说，React 就是一个使用 JavaScript 构建可组合的用户界面引擎，主要作用在于构建 UI。虽然有人说 React 属于 MVC 的 V（视图）层，但在 React 官方博客中阐明 React 不是一个 MVC 框架，而是一个用于构建组件化 UI 的库，是一个前端界面开发工具，他们并不认可 MVC 这种设计模式。

React 源于 Facebook 内部 PHP 框架 XHP 的一个分支，在每次有请求进入时会渲染整个页面。而 React 的出现就是为了把这种重新渲染整个页面的 PHP 式工作流，带入客户端应用，在使用 React 构建用户界面时，只需定义一次，就能将其复用在其他多个地方。当状态改变时，无须做出任何操作，它会自动、高效地更新界面。从此开发人员只需要关心维护应用内的状态，而不需要再关注 DOM 节点。这样开发人员就能从复杂的 DOM 操作中解脱出来，让工作重心回归状态本身。

由于 React 是一个专注于 UI 组件的类库，简单的理念和少量的 API 能和其他各种技术相融合，加之是由互联网“巨头”Facebook 维护，使 React 的生态圈在全球社区得以不断地良性发展。同时，基于 React 还诞生了 React Native，这无疑给当今移动互联网的蓬勃发展投下了的一枚重型“炸弹”。

得益于虚拟 DOM 的实现，React 可以实现跨平台开发：

- Web 应用；
- 原生 iOS 和 Android 应用；
- Canvas 应用和原生桌面应用；
- TV 应用。

可以说是“Learn once, Write Anywhere”。

1.3 React 的特征

本节将介绍有关 React 的三大突出特点：组件化、虚拟 DOM 和单向数据流，这有助于读者更好地认识和理解 React 的设计思想。

1.3.1 组件化

React 书写的一切用户界面都是基于组件的。这么做好处是什么呢？

最显而易见的就是组件具备良好的封装性，可以重复使用。想象一下，在一个应用中，假如每个页面顶部都有一个类似功能的搜索框，需要写多次重复的代码，如果把它单独抽象封装成一个单独的组件，在每个使用到的地方去引用，这样可以减少大量重复、多余的代码，并且方便迭代维护。

在 React 中，对于组件的理解可以比喻成古代的“封邦建国”。天子把自己直接管辖（父组件）以外的土地分封给诸侯，让他们各自管辖属于自己的领地（子组件）。只要天子（父组件）有什么需要，就吩咐（调用）诸侯（子组件）去做就行了。有什么旨意，就派信使传达（props 属性，2.12 节将详细讲解）。这样一来，天子一个人要管辖这么多的领土也不会觉得累了，同时又让自己的国家繁荣富强，实现自治，但又不脱离自己的掌控。

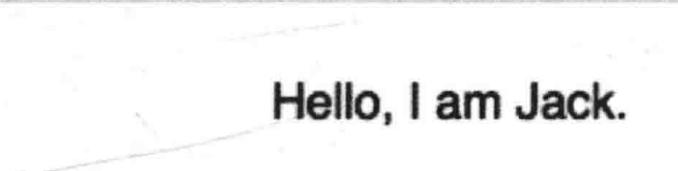
简单的组件示例：

```
import React, { Component } from 'react';
import { render } from 'react-dom';
export default Class MyComponent extends React.Component {
  render() {
    return (
      <div>
        Hello, I am {this.props.name}.
      </div>
    )
  }
}
```

自定义组件后，在其他需要使用这个组件的地方就可以像使用 HTML 标签一样去引用了，例如：

```
import React, { Component } from 'react';
import { render } from 'react-dom';
import MyComponent from './myComponent'
export default class extends React.Component {
  render() {
    return (
      <MyComponent name="Jack" />           // name 是自定义组件的属性
    )
  }
}
```

运行程序，输出结果如图 1.5 所示。



Hello, I am Jack.

1.3.2 虚拟 DOM

先来了解一下什么是 DOM，什么又是虚拟 DOM。

图 1.5 自定义组件

Document Object Model (DOM，文档对象模型)

是 W3C (World Wide Web Consortium, 万维网联盟) 的标准，定义了访问 HTML 和 XML 文档的标准方法：W3C 文档对象模型 (DOM) 是中立于平台和语言的接口，它允许程序和脚本动态地访问和更新文档的内容、结构和样式。简单来说，就是用于连接 document 和 JavaScript 的桥梁。

每个页面都有一个 DOM 树，用于对页面的表达。真实场景中用户的很多操作会导致浏览器的重排（引起 DOM 树重新计算的行为）。一般的写法都是手动操作 DOM，获取页面真实的 DOM 节点，然后修改数据。在复杂场景或需要频繁操作 DOM 的应用中，这样的写法非常消耗性能。当然也有许多方式可以避免页面重排，比如把某部分节点 position 设置为 absolute/fixed 定位，让其脱离文档流；或者在内存中处理 DOM 节点，完成之后推进文档等。这些方法维护成本昂贵，代码难以维护。

React 为了摆脱操作真实 DOM 的噩梦，开创性地把 DOM 树转换为 JavaScript 对象树，这就是虚拟 DOM (Virtual DOM)。

简单理解，虚拟 DOM 就是利用 JS 去构建真实 DOM 树，用于在浏览器中展示。每当有数据更新时，将重新计算整个虚拟 DOM 树，并和旧 DOM 树进行一次对比。对发生变化的部分进行最小程度的更新，从而避免了大范围的页面重排导致的性能问题。虚拟 DOM 树是内存中的数据，所以本身操作性能高很多。

可以说，React 赢就赢在了利用虚拟 DOM 超高性能地渲染页面，另辟蹊径地处理了这个对于开发者来说真实存在的痛点。除此之外，由于操作的对象是虚拟 DOM，与真实浏览器无关，与是否是浏览器环境都没有关系。只需要存在能将虚拟 DOM 转换为真实 DOM 的转换器，就能将其转为真实 DOM 在界面中展现，从而就达到了利用 React 实现跨平台的目的，比如 React Native 的实现。

1.3.3 单向数据流

在 React 中，数据流是单向的。

数据的流向是从父组件流向子组件，至上而下，如图 1.6 所示。这样能让组件之间的关系变得简单且可预测。

`props` 和 `state` 是 React 组件中两个非常重要的概念。`props` 是外来的数据，`state` 是组件内部的数据。一个组件内，可以接受父组件传递给它的数据，如果 `props` 改变了，React 会递归地向下遍历整棵组件树，在使用到这个属性的组件中重新渲染；同时组件本身还有属于自己的内部数据，只能在组件内部修改。可以将其与面向对象编程进行类比：`this.props` 就是传递给构造函数的参数，`this.state` 就是私有属性。

单向数据流的好处就是，所有状态变化都是可以被记录和跟踪的，源头容易追溯，没有“暗箱操作”，只有唯一入口和出口，使程序更直观易理解，利于维护。

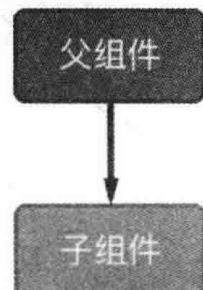


图 1.6 单向数据流

1.4 JSX 语法

前面一起了解了 React 的 3 个主要特征。本节将带领读者来学习 React 书写的“最佳姿势”——JSX 语法。它是由 React 官方推出的一种基于 JavaScript 的拓展语法。虽然不是使用 React 编写代码的必要条件，不过相信当读者了解到 JSX 的好处之后，便不会使用原生 JavaScript 开发 React 了。

1.4.1 JSX 简介

JSX（JavaScript XML），是 JavaScript 的一种拓展语法。可以在 JavaScript 代码中编写更像 XML 写法的代码。React 官方推荐使用 JSX 替代常规 JavaScript 写法，从而让代码

更直观，达到更高的可读性。但需要注意的一点是，它不能直接在浏览器中执行，需要经过转换器将 JSX 语法转为 JS 之后才可以。从本质上讲，JSX 也只是 React.createElement (component, props, ...children) 函数的语法糖，所以在 JSX 当中，依旧可以正常使用 JavaScript 表达式。

当然，使用 JSX 并不是 React 开发应用的必要条件，JSX 是独立于 React 的，也可以不使用。

下面通过两个示例来对比一下使用原生 JS 和使用 JSX 的区别，它们都是用来在页面中展示一个 HelloReact 的文案。

Before: 使用原生 JS 实现 Hello React。

```
class HelloMessage extends React.Component {
  render() {
    return React.createElement(
      "div",
      null,
      "Hello React"
    );
  }
}
ReactDOM.render(React.createElement(HelloMessage, null), mountNode);
```

After: 使用 JSX 实现 HelloReact。

```
class HelloMessage extends React.Component {
  render() {
    return (
      <div>Hello React</div>
    );
  }
}
ReactDOM.render(
  <HelloMessage />,
  mountNode
);
```

再来看一个略微“复杂”的示例，三层 div 嵌套，还是输出 HelloReact。

Before: 使用原生 JSX 实现。

```
class HelloMessage extends React.Component {
  render() {
    return React.createElement(
      "div",
      null,
      React.createElement(
        "div",
        null,
        React.createElement(
          "div",
          null,
          "Hello React"
        )
      )
    );
  }
}
```

```

    );
}
}
ReactDOM.render(React.createElement(HelloMessage, null), mountNode);

```

After: 使用 JSX 实现。

```

class HelloMessage extends React.Component {
  render() {
    return (
      <div>
        <div>
          <div>Hello React</div>
        </div>
      </div>
    );
  }
}
ReactDOM.render(
  <HelloMessage />,
  mountNode
);

```

从上面两个案例的对比中可以明显看出，JSX 语法更接近开发者平时的书写方式。

1.4.2 JSX 的转译

JSX 代码是不能被浏览器直接运行的，需要将其转译为 JavaScript 之后才能运行。转译之后的代码功能相同。由于前端发展速度较快，在很多老项目中依旧可以见到这类写法。这也是本节对 JSX 编译工具发展作一个简单介绍的初衷，初次学习 React 的读者暂时可以当成小故事去阅读。下面来看一看对 JSX 转译的一段小“历史”。

早期 Facebook 提供了一个简单的工具 JSXTransformer，这是一个浏览器端具有转译功能的脚本，将这个 JSXTransformer.js 文件直接引入 HTML 文档就能使用。例如：

```

<script src="./jsxtransformer.js"></script>
// type 为 text/jsx
<script type="text/jsx">
  // JSX 代码
</script>

```

这样写就需要在浏览器端进行转译工作，所以对性能有损耗，影响效率。当然 Facebook 也考虑到了这点，于是对应的也就有了服务端去渲染的工具，那就是 react-tools。这里暂不介绍，读者先大致了解下即可。

随后在 React v 0.14 之后官方发布：

Deprecation of react-tools

The react-tools package and JSXTransformer.js browser file have been deprecated. You can continue using version 0.13.3 of both, but we no longer support them and recommend migrating to Babel, which has built-in support for React and JSX.

也就是说，在 React v0.14 版本后将 JSXTransformer.js 弃用了。接下去可以使用 Babel，如图 1.7 所示，这也是当下主流使用的转译工具。



图 1.7 Babel 界面

Babel 原名是 6to5，是一个开源的转译工具。它的作用就是把当前项目中使用的 ES 6、ES 7 和 JSX 等语法，转译为当下可以执行的 JavaScript 版本，让浏览器能够识别。简单来说，它是一个转码工具集，包含各种各样的插件。

在 Babel 6.0 版本以前，使用了 browser.js，也就是最早替代 JSXTransformer.js 的转化器脚本。在 HTML 中引用如下：

```
<script src="./babel-core/browser.js"></script>
// type 为 text/babel
<script type="text/babel">
// JSX 代码
</script>
```

⚠ 注意：在 Babel 6.0 之后就不再提供 browser.js 了，当然老项目中依旧可以这样使用。不过这种写法在大型项目中通常不会使用，毕竟在浏览器中隔了一道转译层。如果只是想做一个本地的小 demo，还是可以直接引用在浏览器端做转译的。

Babel 还提供了在线编译的功能 (<http://babeljs.io/repl/>)，如图 1.8 所示，可以在上面进行测试或学习。

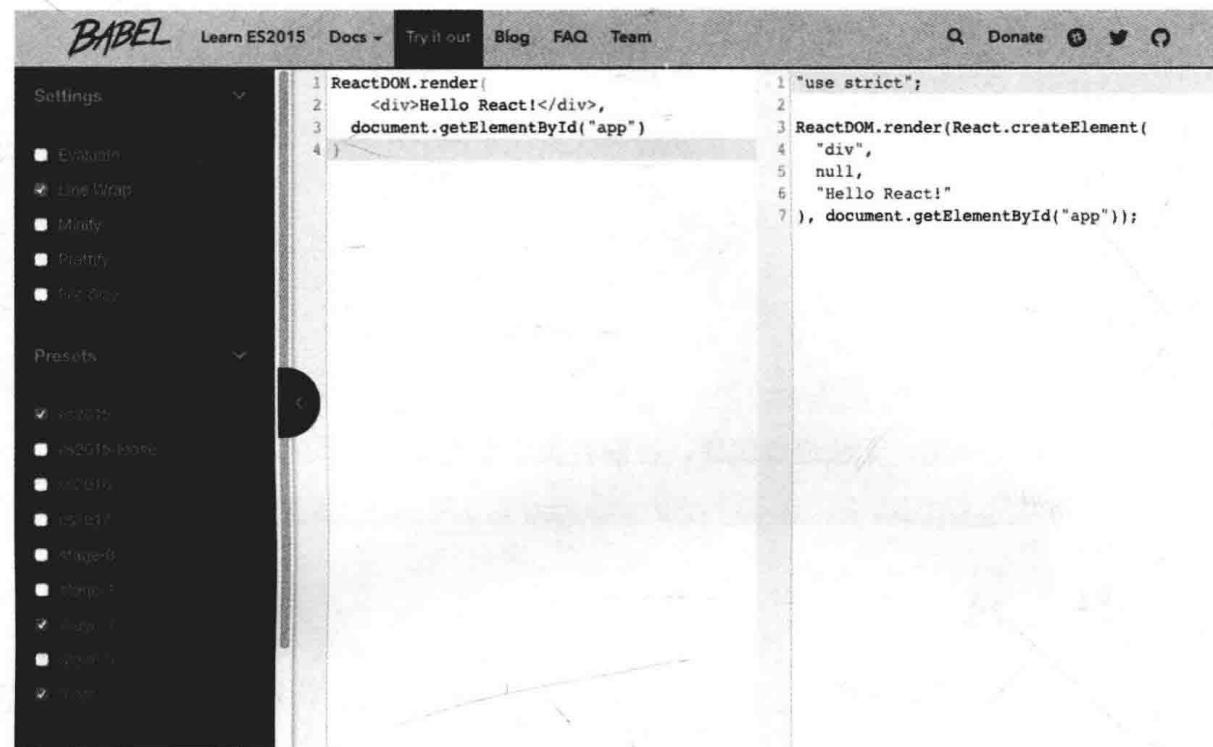


图 1.8 Babel 在线编译界面

以上就是 JSX 转译的大致历程。

本书之后的项目，将使用 Webpack 等构建工具配置 Babel，以实现对 JSX 语法的支持。

1.4.3 JSX 的用法

在 React 中，可以使用花括号 {} 把 JavaScript 表达式放入其中运行，放入 {} 中的代码会被当作 JavaScript 代码进行处理。

1. JSX 嵌套

定义标签时，最外层只允许有一个标签，例如：

```
const MessageList = () => (
  <div>
    <div>Hello React!</div>
    <ul>
      <li>List1</li>
      <li>List2</li>
      <li>List2</li>
    </ul>
  </div>
)
```

假如写成以下这样：

```

const MessageList = () => (
  <div>
    <div>Hello React!</div>
    <ul>
      <li>List1</li>
      <li>List2</li>
      <li>List2</li>
    </ul>
  </div>
  ...
</div>
)

```

进行程序后会报错，无法通过编译，控制台报错如图 1.9 所示。

```

repl: Adjacent JSX elements must be wrapped in an enclosing tag (10:
  3)
    8 |       </ul>
    9 |     </div>
  > 10 |     <div>
        |
  11 |     ...
  12 |   </div>
  13 | )

```

图 1.9 Babel 报错

 注意：由于 JSX 的特性接近 JavaScript 而非 HTML，所以 ReactDOM 使用小驼峰命名（camelCase）来定义属性名。例如，class 应该写为 className。

2. 属性表达式

在传统写法中，属性这样写：

```
<div id="test-id" class="test-class">...</div>
```

JSX 中也实现了同样的写法，比如：

```
const element = <div id="test-id" className="test-class"></div>
// 注意 class 写为 className
```

同时还支持 JavaScript 动态设置属性值，比如：

```
// 注意 class 写为 className
const element = <div id={this.state.testId} className={this.state.testClass}>
</div>;
const element = <img src={user.avatarUrl}></img>;
```

对于复杂情景，还可以直接在属性内运行一个 JavaScript 函数，把函数返回值赋值给属性，比如：

```
const element = <img src={this.getUrl()}></img>;
```

这样，每当状态改变时，返回值会被渲染到真实元素的属性中。

3. 注释

JSX 语法中的注释沿用 JavaScript 的注释方法，唯一需要注意的是，在一个组件的子元素位置使用注释需要用{}括起来。例如：

```
class App extends React.Component {
  render() {
    return (
      <div>
        {/* 注释内容 */}
        <p>Hi, man</p>
      </div>
    )
  }
}
```

4. Boolean属性

Boolean 的属性值，JSX 语法中默认为 true。要识别为 false 就需要使用{}，这类标签常出现在表单元素中，如 disable、checked 和 readOnly 等。例如：

<checkbox checked />等价于<checkbox checked={true} />，但要让 checked 为 false，必须这么写：<checkbox checked={false} />。

5. 条件判断

在 JavaScript 中可以使用判断条件的真假来判断对 DOM 节点进行动态显示或选择性显示。通常，在 HTML 中处理这样的问题比较麻烦，但对于 JavaScript 而言则轻而易举，这给开发者带来了极大的方便。本书主要介绍 3 种条件判断，分别是三目运算符、逻辑与(&&)运算符和函数表达式。

三目运算符示例：

```
class App extends React.Component {
  constructor() {
    super();
    this.state = {
      visible: false
    }
  }
  render() {
    return (
      <div>
        {
          this.state.visible ? <span>visible 为真</span> : <span>visible 为假</span>
        }
      </div>
    )
  }
}
ReactDOM.render(<App />, document.querySelector("#app"))
```

代码运行结果如图 1.10 所示。

逻辑与 (`&&`) 运算符示例：

```
class App extends React.Component {
  constructor() {
    super();
    this.state = {
      visible: false
    }
  }
  render() {
    return (
      <div>
        {
          !this.state.visible && <span>visible 为真</span>
        }
      </div>
    )
  }
}
ReactDOM.render(<App />, document.querySelector("#app"))
```

代码运行结果，如图 1.11 所示。

visible 为假

visible 为真

图 1.10 三目运算符示例显示文本

图 1.11 逻辑与(`&&`)运算符示例显示文本

函数表达式示例：

style 样式：

```
.red{
  color: red;
}
.blue{
  color: blue;
}
```

JSX：用 JSX 语法定义一个名为 App 的组件，用于在页面中渲染一个 div 节点。

```
class App extends React.Component {
  constructor() {
    super();
    this.state = {
      isRed: true
    }
  }
  getClassName() {
    return this.state.isRed?"red":"blue"
  }
}
```

```

}

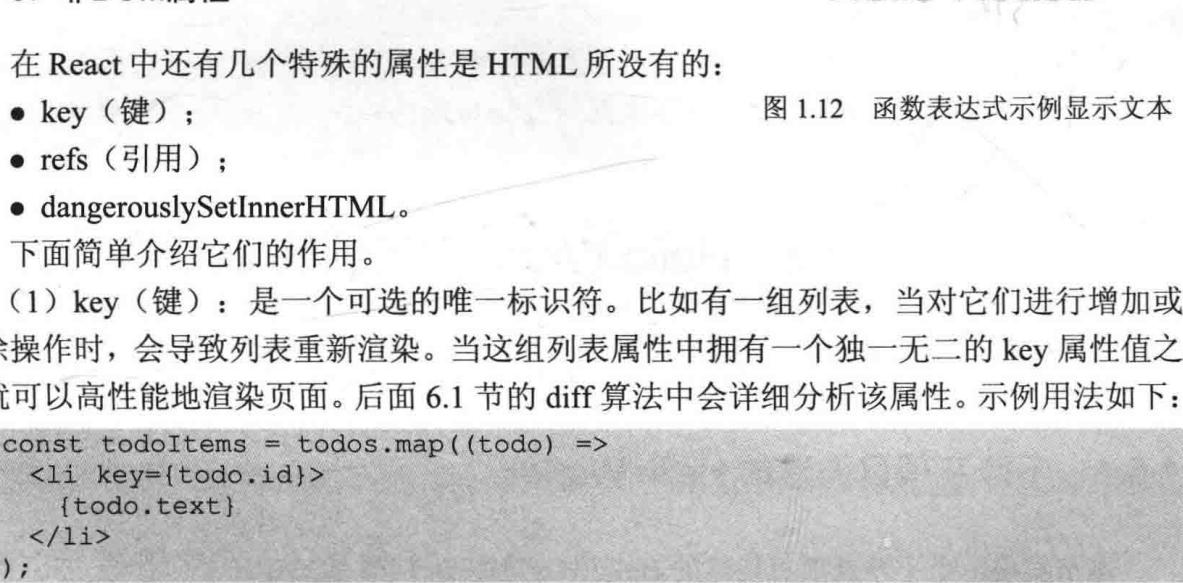
render() {
  return (
    <div className={this.getClassName()}>
      Hello React!
    </div>
  )
}

ReactDOM.render(<App />, document.querySelector("#app"))

```

运行代码，显示效果如图 1.12 所示。

6. 非DOM属性



Hello React!

在 React 中还有几个特殊的属性是 HTML 所没有的：

- key（键）；
- refs（引用）；
- dangerouslySetInnerHTML。

下面简单介绍它们的作用。

图 1.12 函数表达式示例显示文本

(1) **key**（键）：是一个可选的唯一标识符。比如有一组列表，当对它们进行增加或删除操作时，会导致列表重新渲染。当这组列表属性中拥有一个独一无二的 key 属性值之后就可以高性能地渲染页面。后面 6.1 节的 diff 算法中会详细分析该属性。示例用法如下：

```

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

```

(2) **refs**（引用 ref）：任何组件都可以附加这个属性，该属性可以是字符串或回调函数。当 refs 是一个回调函数时，函数接收底层 DOM 元素或实例作为参数。这样就可以直接访问这个 DOM 或组件的节点了。但此时获取到的不是真实的 DOM，而是 React 用来创建真实 DOM 的描述对象。写法如下：

```
<input ref="myInput" />
```

然后就可以通过 this.refs.myInput 去访问 DOM 或组件的节点了。

refs 适用的场景有处理焦点、文本选择、媒体控制、触发强制动画和集成第三方 DOM 库等。需要注意的是，官方并不推荐使用这个属性，除非“迫不得已”。

注意：无状态组件不支持 ref。在 React 调用无状态组件之前没有实例化的过程，因此就没有所谓的 ref。

(3) **dangerouslySetInnerHTML**：这算是 React 中的一个冷知识，它接收一个对象，可以通过字符串形式的 HTML 来正常显示。

```
<div dangerouslySetInnerHTML={{__html: '<span>First &middot; Second</span>'}} />
```

上面这段代码将在页面中显示 First · Second。

通过这种方式还可以避免 cross-site scripting (XSS) 攻击。这里不展开对 XSS 的介绍，读者可根据兴趣自行了解。

7. 样式属性

样式 (style) 属性接收一个 JavaScript 对象，用于改变 DOM 的样式。

JSX 中的样式示例：

```
let styles = {
  fontSize: "14px",
  color: "#red"
}
function AppComponent() {
  return <div style={styles}>Hello World!</div>
}
```

1.5 Hello World 实战训练

遵循传统，在学习 React 前先带领读者构建一个基于 Webpack 的 Hello World 应用。

1.5.1 不涉及项目构建的 Hello World

本节实现一个不涉及项目构建的 Hello World。

React 的第一个 Hello World 网页示例(源码地址是 <https://jsfiddle.net/allan91/2h1sf0ky/8/>)：

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Hello World</title>
  <script src="https://cdn.bootcss.com/react/15.4.2/react.min.js"></script>
  <script src="https://cdn.bootcss.com/react/15.4.2/react-dom.min.js"></script>
  <script src="https://cdn.bootcss.com/babel-standalone/6.22.1/babel.min.js"></script>
</head>
<body>
  <div id="root"></div>
  <script type="text/babel">
    ReactDOM.render(
      <h1>Hello World</h1>, //JSX 格式
      document.getElementById("root")
    )
  </script>
</body>
</html>
```

```

    );
</script>
</body>
</html>

```

上面的代码很简单，直接引用 CDN (Content-Delivery-Network) 上的 react.min.js、react-dom.min.js 和 babel.min.js 这 3 个脚本即可直接使用。唯一需要注意的是，script 的 type 属性需要写为 text/babel。在浏览器中打开这个 HTML 文件即可展示 Hello World 文案。

说明：CDN (Content Delivery Network) 是构建在网络之上的内容分发网络，依靠部署在各地的边缘服务器，通过中心平台的负载均衡、内容分发、调度等功能模块，使用户就近获取所需内容，降低网络拥塞，提高用户访问响应速度和命中率。react.main.js 是 React 的核心代码包；react-dom.min.js 是与 DOM 相关的包，主要用于把虚拟 DOM 渲染的文档变为真实 DOM，当然还有其他一些方法；babel.min.js 是用来编译还不被浏览器支持的代码的编译工具，其中 min 表示这是被压缩过的 JS 库。

也可以将 JavaScript 代码写在外面，比如在根目录下新建 main.js：

```

ReactDOM.render(
  <h1>Hello World</h1>, //JSX 格式
  document.getElementById("root")
);

```

然后在 HTML 文件内引入：

```
<script type="text/babel" src="./main.js"></script>
```

1.5.2 基于 Webpack 的 Hello World

真实项目中一般不会像 1.5.1 节介绍的这样来搭建项目，有了前面小节中的基础知识，接下来开始动手搭建一个基于 Webpack 的 Hello World 应用。这次搭建分为两部分：一部分是前期必要配置，另一部分是开发 React 代码。

基于 Webpack 的 React Hello World 项目示例：

1. 前期必要配置

(1) 首先要确保读者的开发设备上已经安装过 Node.js，新建一个项目：

```

mkdir react-hello-world
cd react-hello-world
npm init -y

```

(2) 项目中使用的是 Webpack 4.x，在项目根目录下执行如下命令：

```
npm i webpack webpack-cli -D
```

注意：上面命令代码中 npm install module_name -D 即 npm intsll module_name

`--save-dev`。表示写入 `package.json` 的 `devDependencies`。`devDependencies` 里面的插件用于开发环境，不用于生产环境。`npm install module_name --S` 即 `npm intsl module_name --save`。`dependencies` 是需要发布到生产环境的。

(3) 安装完 Webpack，需要有一个配置文件让 Webpack 知道要做什么事，这个文件取名为 `webpack.config.js`。

```
touch webpack.config.js
```

然后配置内容如下：

```
var webpack = require('webpack');
var path = require('path');
var APP_DIR = path.resolve(__dirname, 'src');
var BUILD_DIR = path.resolve(__dirname, 'build');
var config = {
  entry: APP_DIR + '/index.jsx', // 入口
  output: {
    path: BUILD_DIR, // 出口路径
    filename: 'bundle.js' // 出口文件名
  }
};
module.exports = config;
```

这是 Webpack 使用中最简单的配置，只包含了打包的入口和出口。`APP_DIR` 表示当前项目的入口路径，`BUILD_DIR` 表示当前项目打包后的输出路径。

(4) 上面配置的入口需要新建一个应用的入口文件 `./src/index.jsx`，我们让其打印 Hello World：

```
console.log('Hello World');
```

(5) 用终端在根目录下执行：

```
./node_modules/.bin/webpack -d
```

上面的命令在开发环境运行之后，会在根目录下生成一个新的 `build` 文件夹，里面包含了 Webpack 打包的 `bundle.js` 文件。

(6) 接下来创建 `index.html`，用于在浏览器中执行 `bundle.js`：

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Hello World</title>
</head>
<body>
  <div id="app"></div>
  <!--bundle.js 是 Webpack 打包后生成的文件-->
  <script src="build/bundle.js" type="text/javascript"></script>
</body>
</html>
```

在浏览器中打开 index.html 文件，在控制台就能看到./src/index.jsx 打印的内容：Hello World。

(7) 为了提高效率和使用最新的 ES 语法，通常使用 JSX 和 ES 6 进行开发。但 JSX 和 ES 6 语法在浏览器中还没有被完全支持，所以需要在 Webpack 中配置相应的 loader 模块来编译它们。只有这样，打包出来的 bundle.js 文件才能被浏览器识别和运行。

接下来安装 Babel：

```
npm i -D babel-core babel-loader@7 babel-preset-env babel-preset-react
```

 注意：babel-core 是调用 Babel 的 API 进行转码的包；babel-loader 是执行转义的核心包；babel-preset-env 是一个新的 preset，可以根据配置的目标运行环境自动启用需要的 Babel 插件；babel-preset-react 用于转译 React 的 JSX 语法。

(8) 在 webpack.config.js 中配置 loader：

```
var webpack = require("webpack");
var path = require("path");
var BUILD_DIR = path.resolve(__dirname, "build"); // 构建路径
var APP_DIR = path.resolve(__dirname, "src"); // 项目路径
var config = {
  entry: APP_DIR + "/index.jsx", // 项目入口
  output: {
    path: BUILD_DIR, // 路由
    filename: "bundle.js" // 文件命名
  },
  module: {
    rules: [
      {
        test: /\.js|jsx$/,
        exclude: /node_modules/,
        use: [
          loader: "babel-loader" // 使用 babel-loader 这个 loader 库
        ]
      }
    ]
  }
};
module.exports = config;
```

(9) 创建.babelrc 文件：

```
touch .babelrc
```

配置相应内容来告诉 babel-loader 使用 ES 6 和 JSX 插件：

```
{
  "presets": ["env", "react"]
}
```

至此为止，已经完成开发该项目的基础配置工作。

2. 使用React编码

下面正式开始使用 React 来编写前端代码。

(1) 用 NPM 命令安装 react 和 react-dom:

```
npm install react react-dom -S
```

(2) 用下面代码替换./src/index.jsx 中的 console:

```
import React from 'react';
import { render } from 'react-dom';
class App extends React.Component {
  render () {
    return <p> Hello React</p>;
  }
}
render(<App/>, document.getElementById('app'));
```

(3) 在根目录下执行:

```
./node_modules/.bin/webpack -d
```

在浏览器中打开 index.html，将会在页面展示 Hello World。当然真实开发中不能每一次修改前端代码就执行一次 Webpack 编译打包，可以执行如下命令来监听文件变化：

```
./node_modules/.bin/webpack -d --watch
```

终端将会显示：

```
myfirstapp Jack$ ./node_modules/.bin/webpack -d --watch
webpack is watching the files...
Hash: 6dbf97954b511aa86515
Version: webpack 4.22.0
Time: 839ms
Built at: 2018-10-23 19:05:01
          Asset      Size  Chunks             Chunk Names
bundle.js   1.87 MiB       main  [emitted]  main
Entrypoint main = bundle.js
[./src/index.jsx] 2.22 KiB {main}  [built]
+ 11 hidden modules
```

这就是 Webpack 的监听模式，一旦项目中的文件有改动，就会自动执行 Webpack 编译命令。不过浏览器上展示的 HTML 文件不会主动刷新，需要手动刷新浏览器。如果想实现浏览器自动刷新，可以使用 react-hot-loader (源码地址 <https://github.com/gaearon/react-hot-loader>)。

(4) 在真实的项目开发中，一般使用 NPM 执行 ./node_modules/.bin/webpack -d --watch 命令来开发。这需要在 package.json 中进行如下配置：

```
{
  ...
  "scripts": {
    "dev": "webpack -d --watch",
    "build": "webpack -p",
    "test": "echo \"Error: no test specified\" && exit 1"
  }
}
```

```
},
...
}
```

(5) 现在只需要在根目录下执行如下命令就能开发与构建:

```
npm run dev
npm run build
```

以上为真实项目中一个较为完整的项目结构,读者可以在此基础上根据项目需要自行拓展其他功能。本例源码地址为 <https://github.com/khno/react-hello-world>, 分支为 master。项目完整的结构如下:

```
.
├── build
│   └── bundle.js
├── index.html
├── package-lock.json
├── package.json
└── src
    ├── index.jsx
    ├── .gitignore
    ├── .babelrc
    └── webpack.config.js
```

1.5.3 Hello World 进阶

前面使用 Webpack 实现了最简单的项目构建,每次写完代码都需要手动刷新浏览器来查看实时效果。接下来继续完善上面的构建项目,实现如下功能:

- 项目启动,开启 HTTP 服务,自动打开浏览器;
- 实时监听代码变化,浏览器实时更新;
- 生产环境代码构建。

模块热替换(Hot Module Replacement, HMR)是 Webpack 中最令人兴奋的特性之一。当项目中的代码有修改并保存后,Webpack 能实时将代码重新打包,并将新的包同步到浏览器,让浏览器立即自动刷新。

在开始配置前,读者可以先思考一个问题,如何实现浏览器实时更新呢?也许会想到让浏览器每隔一段时间向本地服务器发送请求,采用轮询(Polling)方式来实现;或者让服务器端来通知客户端,服务器端接收到客户端有资源需要更新,就主动通知客户端;或者直接使用 WebSocket?

 **说明:** WebSocket 是一种在单个 TCP 连接上进行全双工通信的协议。WebSocket 通信协议于 2011 年被 IETF 定为标准 RFC6455,并由 RFC7936 补充规范。

WebSocket API 也被 W3C 定为标准。WebSocket 使客户端和服务器端之间的数据交换变得更加简单,允许服务器端主动向客户端推送数据。以上信息参考来源于维基百科。

其实 HMR 原理就是上面说的那样，早期 HMR 的原理是借助于 EventSource，后来使用了 WebSocket。可以在本地开发环境开启一个服务，将其作为本地项目的服务端，然后与本地浏览器之间进行通信即可。欲了解详情，请读者自行学习。

 注意：WebSocket 是基于 TCP 的全双工通信协议。

大致了解了 HMR 的原理之后，开始动手实践吧。

实时更新的 Hello World 示例：

(1) 首先，采用 webpack-dev-server 作为 HMR 的工具。在之前的项目基础上安装：

```
npm install webpack-dev-server -D
```

(2) 修改 webpack.config.js 文件：

```
+ devServer: {
+   port: 3000,
+   contentBase: "./dist"
+ },
```

devServer.port 是服务启动后的端口号，devServer.contentBase 是配置当前服务读取文件的目录。启动后可以通过 localhost:3000 端口访问当前项目。

(3) 修改 package.json 文件：

```
"scripts": {
+ "start": "webpack-dev-server --open --mode development",
  "test": "echo \"Error: no test specified\" && exit 1"
},
```

按照上面配置 start 命令后，执行 npm start 会直接找到 webpack.config.js 来启动项目。

(4) 最后安装 html-webpack-plugin 和 clean-webpack-plugin。

```
npm install html-webpack-plugin clean-webpack-plugin -D
```

html-webpack-plugin 插件用于简化创建 HTML 文件，它会在 body 中用 script 标签来包含我们生成的所有 bundles 文件。配置如下：

```
plugins: [
  new HtmlWebpackPlugin({
    template: "index.html",
    // favicon: 'favicon.ico',
    inject: true,
    sourceMap: true,
    chunksSortMode: "dependency"
  })
]
```

clean-webpack-plugin 插件在生产环境中编译文件时，会先删除 build 或 dist 目录文件，然后生成新的文件。

(5) 随着文件的增、删，打包的 dist 文件内可能会产生一些不再需要的静态资源，我们并不希望将这些静态资源部署到服务器上占用空间，所以最好在每次打包前清理 dist 目录。在 Webpack 中配置如下：

```
plugins: [
  new CleanWebpackPlugin(["dist"])
]
```

此时，Webpack 代码清单如下：

```
var webpack = require("webpack");
var path = require("path");
const CleanWebpackPlugin = require("clean-webpack-plugin");
var BUILD_DIR = path.resolve(__dirname, "dist"); // 输出路径
var APP_DIR = path.resolve(__dirname, "src"); // 项目路径
const HtmlWebpackPlugin = require("html-webpack-plugin");
var config = {
  entry: APP_DIR + "/index.jsx", // 项目入口
  output: {
    path: BUILD_DIR, // 输出路由
    filename: "bundle.js" // 输出文件命名
  },
  module: {
    rules: [
      {
        test: /\.js|jsx$/, // 编译后缀为 js 和 jsx 的格式文件
        exclude: /node_modules/,
        use: {
          loader: "babel-loader"
        }
      }
    ]
  },
  devServer: { // 在开发模式下，提供虚拟服务器用于项目开发和调试
    port: 3000, // 端口号
    contentBase: "./dist"
  },
  plugins: [ // 拓展 Webpack 功能
    new HtmlWebpackPlugin({ // 生成 HTML 文件
      template: "index.html",
      // favicon: 'theme/img/favicon.ico',
      inject: true,
      sourceMap: true, // 是否生成 sourceMap
      chunksSortMode: "dependency"
    }),
    new CleanWebpackPlugin(["dist"]) // 清除文件
  ]
};
module.exports = config;
```

(6) 在根目录终端执行 npm run start:

```
AllanMacBook-Pro:react-hello-world jack$ npm start
> myfirstapp@1.0.0 start /Users/allan/react-hello-world
> webpack-dev-server --open --mode development
// 删除 dist 文件
clean-webpack-plugin: /Users/alan/react-hello-world/dist has been removed.
[wds]: Project is running at http://localhost:3000/
[wds]: webpack output is served from /
```

```

「wds」: Content not from webpack is served from ./dist
「wdm」: wait until bundle finished: /
「wdm」: Hash: 4bfc0734100357258elf
Version: webpack 4.23.1
Time: 1358ms
Built at: 2018-11-06 20:11:12
    Asset      Size  Chunks             Chunk Names
  bundle.js   1.12 MiB  main  [emitted]  main
  index.html  342 bytes          [emitted]
Entrypoint main = bundle.js
[0] multi ./node_modules/_webpack-dev-server@3.1.10@webpack-dev-server/client?http://localhost:3000 ./src/index.jsx 40 bytes {main} [built]
  [./node_modules/_ansi-html@0.0.7@ansi-html/index.js] 4.16 KiB {main} [built]
  [./node_modules/_ansi-regex@2.1.1@ansi-regex/index.js] 135 bytes {main} [built]
  [./node_modules/_events@1.1.1@events/events.js] 8.13 KiB {main} [built]
  [./node_modules/_loglevel@1.6.1@loglevel/lib/loglevel.js] 7.68 KiB {main} [built]
  [./node_modules/_react-dom@16.6.0@react-dom/index.js] 1.33 KiB {main} [built]
  [./node_modules/_react@16.6.0@react/index.js] 190 bytes {main} [built]
  [./node_modules/_strip-ansi@3.0.1@strip-ansi/index.js] 161 bytes {main} [built]
  [./node_modules/_url@0.11.0@url/url.js] 22.8 KiB {main} [built]
  [./node_modules/_webpack-dev-server@3.1.10@webpack-dev-server/client/index.js?http://localhost:3000] ./node_modules/_webpack-dev-server@3.1.10@webpack-dev-server/client?http://localhost:3000 7.78 KiB {main} [built]
  [./node_modules/_webpack-dev-server@3.1.10@webpack-dev-server/client/overlay.js] 3.58 KiB {main} [built]
  [./node_modules/_webpack-dev-server@3.1.10@webpack-dev-server/client/socket.js] 1.05 KiB {main} [built]
  [./node_modules/_webpack@4.23.1@webpack/hot/emitter.js] (webpack)/hot/emitter.js 75 bytes {main} [built]
  [./node_modules/webpack/hot sync ^\.\./log$] ./node_modules/webpack/hot sync nonrecursive ^\.\./log$ 170 bytes {main} [built]
  [./src/index.jsx] 2.22 KiB {main} [built]
    + 22 hidden modules
Child html-webpack-plugin for "index.html":
    1 asset
Entrypoint undefined = index.html
  [./node_modules/_html-webpack-plugin@3.2.0@html-webpack-plugin/lib/loader.js!./index.html] 506 bytes {0} [built]
  [./node_modules/_lodash@4.17.11@lodash/lodash.js] 527 KiB {0} [built]
  [./node_modules/_webpack@4.23.1@webpack/buildin/global.js] (webpack)/buildin/global.js 489 bytes {0} [built]
  [./node_modules/_webpack@4.23.1@webpack/buildin/module.js] (webpack)/buildin/module.js 497 bytes {0} [built]
「wdm」: Compiled successfully.

```

此时，一旦项目内的文件有改动，就会自动执行编译，并通知浏览器自动刷新。

(7)至此开发配置已经完成，接下来配置生产命令，只需执行 `webpack -p` 即可。`webpack -p` 表示压缩打包代码。在 `package.json` 内配置：

```

"scripts": {
  "start": "webpack-dev-server --open --mode development",
  "build": "webpack -p",
  "test": "echo \"Error: no test specified\" && exit 1"
},

```

(8) 打包的时候只需执行 `npm run build` 即可，打包成功会出现：

```
AllandeMacBook-Pro:react-hello-world allan$ npm run build
> myfirstapp@1.0.0 build /Users/allan/react-hello-world
> webpack -p
clean-webpack-plugin: /Users/allan/react-hello-world/dist has been removed.
Hash: 97bb34a13d3fc8cbfccc
Version: webpack 4.23.1
Time: 2647ms
Built at: 2018-11-06 20:22:45
    Asset      Size  Chunks             Chunk Names
bundle.js    110 KiB     0  [emitted]  main
index.html   342 bytes          [emitted]
Entrypoint main = bundle.js
[2] ./src/index.jsx 2.22 KiB {0} [built]
+ 7 hidden modules
Child html-webpack-plugin for "index.html":
  1 asset
Entrypoint undefined = index.html
[0] ./node_modules/_html-webpack-plugin@3.2.0@html-webpack-plugin/lib/loader.js!./index.html 506 bytes {0} [built]
[2] (webpack)/buildin/global.js 489 bytes {0} [built]
[3] (webpack)/buildin/module.js 497 bytes {0} [built]
+ 1 hidden module
```

上面的打包结果展示了此次打包的哈希值、耗时，以及打包后每个包的大小等信息。此时项目的结构为：

```
.
├── README.md
├── dist
│   └── bundle.js
│   └── index.html
├── index.html
├── package-lock.json
├── package.json
└── src
    └── index.jsx
    └── webpack.config.js
```

以上项目的完整代码可以在 <https://github.com/khno/react-hello-world> 中下载，分支为 dev。本书中后面章节的代码案例默认将以此为基础运行，读者可以自行下载。

第2章 React 的组件

在 React 中，组件是应用程序的基石，页面中所有的界面和功能都是由组件堆积而成的。在前端组件化开发之前，一个页面可能会有成百上千行代码逻辑写在一个.js 文件中，这种代码可读性很差，如果增加功能，很容易出现一些意想不到的问题。合理的组件设计有利于降低系统各个功能的耦合性，并提高功能内部的聚合性。这对于前端工程化及降低代码维护成本来说，是非常必要的。

本章主要介绍 React 中组件的创建、成员、通信和生命周期，最后会通过一个实战案例——TodoList 演示组件的使用。

2.1 组件的声明方式

简单来说，在 React 中创建组件的方式有 3 种。

- ES 5 写法：React.createClass()（老版本用法，不建议使用）；
- ES 6 写法：React.Component；
- 无状态的函数式写法，又称为纯组件 SFC。

2.1.1 ES 5 写法：React.createClass()

React.createClass()是React刚出现时官方推荐的创建组件方式，它使用ES 5原生的JavaScript来实现 React 组件。React.createClass()这个方法构建一个组件“类”，它接受一个对象为参数，对象中必须声明一个 render()方法，render()方法将返回一个组件实例。

使用 React.createClass()创建组件示例：

```
var Input = React.createClass({
  // 定义传入 props 中的各种属性类型
  propTypes: {
    initialValue: React.PropTypes.string
  },
  // 组件默认的 props 对象
  defaultProps: {
    initialValue: ''
  }
})
```

```

},
// 设置 initial state
getInitialState: function() {
    return {
        text: this.props.initialValue || 'placeholder'
    };
},
handleChange: function(event) {
    this.setState({
        text: event.target.value
    });
},
render: function() {
    return (
        <div>
            Type something:
            <input onChange={this.handleChange} value={this.state.text} />
        </div>
    );
}
);
});

```

`createClass()`本质上是一个工厂函数。`createClass()`声明的组件方法的定义使用半角逗号隔开，因为 `createClass()`本质上是一个函数，传递给它的是一个 Object。通过 `propTypes` 对象和 `getDefaultProps()` 方法来设置 `props` 类型和获取 `props`。`createClass()` 内的方法会正确绑定 `this` 到 React 类的实例上，这也会导致一定的性能开销。React 早期版本使用该方法，而在新版本中该方法被废弃，因此不建议读者使用。

2.1.2 ES 6 写法：React.Component

`React.Component` 是以 ES 6 的形式来创建组件的，这是 React 目前极为推荐的创建有状态组件的方式。相对于 `React.createClass()`，此种方式可以更好地实现代码复用。本节将 2.1.1 节介绍的 `React.createClass()` 形式改为 `React.Component` 形式。

使用 `React.Component` 创建组件示例：

```

class Input extends React.Component {
    constructor(props) {
        super(props);
        // 设置 initial state
        this.state = {
            text: props.initialValue || 'placeholder'
        };
        // ES 6 类中的函数必须手动绑定
        this.handleChange = this.handleChange.bind(this);
    }
    handleChange(event) {
        this.setState({
            text: event.target.value
        });
    }
}

```

```

    }
    render() {
        return (
            <div>
                Type something:
                <input onChange={this.handleChange}
                    value={this.state.text} />
            </div>
        );
    }
}

```

React.Component 创建的组件，函数成员不会自动绑定 this，需要开发者手动绑定，否则 this 无法获取当前组件的实例对象。当然绑定 this 的方法有多种，除了上面的示例代码中在 constructor() 中绑定 this 外，最常见的还有通过箭头函数来绑定 this，以及在方法中直接使用 bind(this) 来绑定这两种。

通过箭头函数来绑定 this 示例：

```
// 使用 bind 来绑定
<div onClick={this.handleClick.bind(this)}></div>
```

在方法中直接使用 bind(this) 来绑定 this 示例：

```
// 使用 arrow function 来绑定
<div onClick={()=>this.handleClick()}></div>
```

2.1.3 无状态组件

下面来看看无状态组件，它是 React 0.14 之后推出的。如果一个组件不需要管理 state，只是单纯地展示，那么就可以定义成无状态组件。这种方式声明的组件可读性好，能大大减少代码量。无状态函数式组件可以搭配箭头函数来写，更简洁，它没有 React 的生命周期和内部 state。

无状态函数式组件示例：

```
const HelloComponent = (props) =>(
    <div>Hello {props.name}</div>
)
ReactDOM.render(<HelloComponent name="marlon" />, mountNode)
```

无状态函数式组件在需要生命周期时，可以搭配高阶组件（HOC）来实现。无状态组件作为高阶组件的参数，高阶组件内存放需要的生命周期和状态，其他只负责展示的组件都使用无状态函数式的组件来写。

有生命周期的函数式组件示例：

```
import React from 'react';
export const Table = (ComposedComponent) => {
    return class extends React.Component {
        constructor(props) {
            super(props)
```

```

    }
    componentDidMount() {
      console.log('componentDidMount');
    }
    render() {
      return (
        <ComposedComponent {...this.props}/>
      )
    }
  }
}

```

 注意：React 16.7.0-alpha（内测）中引入了 Hooks，这使得在函数式组件内可以使用 state 和其他 React 特性。

2.2 组件的主要成员

在 React 中，数据流是单向流动的，从父节点向子节点传递（自上而下）。子组件可以通过属性 props 接收来自父组件的状态，然后在 render()方法中渲染到页面。每个组件同时又拥有属于自己内部的状态 state，当父组件中的某个属性发生变化时，React 会将此改变了的状态向下递归遍历组件树，然后触发相应的子组件重新渲染（re-render）。

如果把组件视为一个函数，那么 props 就是从外部传入的参数，而 state 可以视为函数内部的参数，最后函数返回虚拟 DOM。

本节将学习组件中最重要的成员 state 和 props。

2.2.1 状态（state）

每个 React 组件都有自己的状态，相比于 props，state 只存在于组件自身内部，用来影响视图的展示。可以使用 React 内置的 setState()方法修改 state，每当使用 setState()时，React 会将需要更新的 state 合并后放入状态队列，触发调和过程（Reconciliation），而不是立即更新 state，然后根据新的状态结构重新渲染 UI 界面，最后 React 会根据差异对界面进行最小化重新渲染。

React 通过 this.state 访问状态，调用 this.setState()方法来修改状态。

React 访问状态示例：

（源码地址为 <https://jsfiddle.net/allan91/etbj6gsx/1/>）

```

class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      data: 'World'
    }
  }
}

```

```

        }
    }

    render() {
        return(
            <div>
                Hello, {this.state.data}
            </div>
        )
    }
}

ReactDOM.render(
    <App />,
    document.querySelector('#app') // App 组件挂载到 ID 为 app 的 DOM 元素上
)

```

上述代码中，App 组件在 UI 界面中展示了自身的状态 state。下面使用 setState()修改这个状态。

React 修改状态示例：

(源码地址为 <https://jsfiddle.net/allan91/etbj6gsx/3/>)

```

class App extends React.Component {
    constructor(props) {
        super(props);
        this.state = {
            data: 'World'
        }
    }

    handleClick = () => {
        this.setState({
            data: 'Redux'
        })
    }

    render() {
        return(
            <div>
                Hello, {this.state.data}
                <button onClick={this.handleClick}>更新</button>
            </div>
        )
    }
}

ReactDOM.render(
    <App />,
    document.querySelector("#app")
)

```

上述代码中通过单击“更新”按钮使用 setState()方法修改了 state 值，触发 UI 界面更新。本例状态更改前后的展示效果，如图 2.1 所示。

Hello, World 更新

Hello, Redux 更新

图 2.1 修改 state

2.2.2 属性 (props)

state 是组件内部的状态，那么组件之间如何“通信”呢？这就是 props 的职责所在了。通俗来说，props 就是连接各个组件信息互通的“桥梁”。React 本身是单向数据流，所以在 props 中数据的流向非常直观，并且 props 是不可改变的。props 的值只能从默认属性和父组件中传递过来，如果尝试修改 props，React 将会报出类型错误的提示。

props 示例应用：

(源码地址为 <https://jsfiddle.net/n5u2wwjg/35076/>)

```
function Welcome(props) {
  return <p>Hello, {props.name}</p>
}
function App() {
  return (
    <Welcome name='world' /> // 引用 Welcome 组件，name 为该组件的属性
  )
}
ReactDOM.render(
  <App />,
  document.querySelector("#app")
)
```

上述代码使用了函数定义组件。被渲染的 App 组件内引用了一个外部组件 Welcome，并在该组件内定义了一个名为 name 的属性，赋值为 world。Welcome 组件接收到来自父组件的 name 传递，在界面中展示 Hello, World。

当然，也可以使用 class 来定义一个组件：

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

这个 Welcome 组件与上面函数式声明的组件在 React 中的效果是一样的。

2.2.3 render()方法

render()方法用于渲染虚拟 DOM，返回 ReactElement 类型。

元素是 React 应用的最小单位，用于描述界面展示的内容。很多初学者会将元素与“组

件”混淆。其实，元素只是组件的构成，一个元素可以构成一个组件，多个元素也可以构成一个组件。`render()`方法是一个类组件必须拥有的特性，其返回一个 JSX 元素，并且外层一定要使用一个单独的元素将所有内容包裹起来。比如：

```
render() {
  return(
    <div>a</div>
    <div>b</div>
    <div>c</div>
  )
}
```

上面这样是错误的，外层必须有一个单独的元素去包裹：

```
render() {
  return(
    <div>
      <div>a</div>
      <div>b</div>
      <div>c</div>
    </div>
  )
}
```

1. `render()`返回元素数组

2017 年 9 月，React 发布的 React 16 版本中为 `render()`方法新增了一个“支持返回数组组件”的特性。在 React 16 版本之后无须将列表项包含在一个额外的元素中了，可以在 `render()`方法中返回元素数组。需要注意的是，返回的数组跟其他数组一样，需要给数组元素添加一个 `key` 来避免 `key warning`。

`render()`方法返回元素数组示例：

(源码地址为 <https://jsfiddle.net/n5u2wwjg/35080/>)

```
render() {
  return [
    <div key="a">a</div>,
    <div key="b">b</div>,
    <div key="c">c</div>,
  ];
}
```

除了使用数组包裹多个同级子元素外，还有另外一种写法如下：

```
import React from 'react';
export default function () {
  return (
    <>
      <div>a</div>
      <div>b</div>
      <div>c</div>
    </>
  );
}
```

简写的`<></>`其实是 React 16 中 `React.Fragment` 的简写形式，不过它对于部分前端工具的支持还不太好，建议使用完整写法，具体如下：

```
import React from 'react';
export default function () {
  return (
    <React.Fragment>
      <div>a</div>
      <div>b</div>
      <div>c</div>
    </React.Fragment>
  );
}
```

最后输出到页面的标签也能达到不可见的效果，也就是在同级元素外层实际上是没有包裹其他元素的，这样能减少 DOM 元素的嵌套。

2. render()返回字符串

当然，`render()`方法也可以返回字符串。

`render()`方法返回字符串示例：

(源码地址为 <https://jsfiddle.net/n5u2wwjg/35079/>)

```
render() {
  return 'Hello World';
}
```

运行程序，界面中将展示以上这段字符串。

3. render()方法中的变量与运算符`&&`

`render()`方法中可以使用变量有条件地渲染要展示的页面。常见做法是通过花括号{}包裹代码，在 JSX 中嵌入任何表达式，比如逻辑与`&&`。

`render()`方法中使用运算符示例：

```
const fruits = ['apple', 'orange', 'banana'];
function Basket(props) {
  const fruitsList = props.fruits;
  return (
    <div>
      <p>I have: </p>
      {fruitsList.length > 0 &&
        <span>{fruitsList.join(', ') }</span>
      }
    </div>
  )
}
ReactDOM.render(<Basket fruits={fruits}/>, document.querySelector("#app"))
```

上述代码表示，如果从外部传入 `Basket` 组件的数组不为空，也就是表达式左侧为真，`&&`右侧的元素就会被渲染。展示效果如图 2.2 所示。如果表达式左侧为 `false`，`&&`右侧元

素就会被 React 忽略渲染。

4. render()方法中的三目运算符

在 render() 方法中还能使用三目运算符
condition ? true : false。

在 render()方法中使用三目运算符示例：

(源码地址为 <https://jsfiddle.net/n5u2wwjg/35239/>)

```
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      isUserLogin: false
    }
  }
  render() {
    const { isUserLogin } = this.state;
    return (
      <div>
        { isUserLogin ? <p>已登录</p> : <p>未登录</p> }
      </div>
    )
  }
}
ReactDOM.render(<App/>, document.querySelector("#app"))
```

上述代码根据 isUserLogin 的真和假来动态显示 p 标签的内容，当然也可以动态展示封装好的组件，例如：

```
return (
  <div>
    { isUserLogin ? <ComponentA /> : <ComponentB /> }
  </div>
)
```

2.3 组件之间的通信

React 编写的应用是以组件的形式堆积而成的，组件之间虽相互独立，但相互之间还是可以通信的。本节将介绍组件中的几种通信方式。

2.3.1 父组件向子组件通信

前面章节已经提到过，React 的数据是单向流动的，只能从父级向子级流动，父级通过 props 属性向子级传递信息。

I have:
apple, orange, banana

图 2.2 render()方法中的逻辑与&&

父组件向子组件通信示例：

(源码地址为 <https://jsfiddle.net/n5u2wwjg/35403/>)

```
class Child extends React.Component {
  render () {
    return (
      <div>
        <h1>{ this.props.fatherToChild }</h1>
      </div>
    )
  }
}
class App extends React.Component {
  render() {
    let data = 'This message is from Dad!'
    return (
      <Child fatherToChild={ data } />
    )
  }
}
ReactDOM.render(
  <App/>,
  document.querySelector("#app")
)
```

上述代码中有两个组件：子组件 Child 和父组件 App。子组件在父组件中被引用，然后在父组件内给子组件定了一个 props: fatherToChild，并将父组件的 data 传递给子组件中展示。

 注意：父组件可以通过 props 向子组件传递任何类型。

2.3.2 子组件向父组件通信

虽然 React 数据流是单向的，但并不影响子组件向父组件通信。通过父组件可以向子组件传递函数这一特性，利用回调函数来实现子组件向父组件通信。当然也可以通过自定义事件机制来实现，但这种场景会显得过于复杂。所以为了简单方便，还是利用回调函数来实现。

子组件向父组件通信示例：

```
class Child extends React.Component {
  render () {
    return <input type="text" onChange={(e)=>this.props.handleChange(e.target.value)} />
  }
}
class App extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
```

```

        data: ''
    }
}

handleChange = text => {
  this.setState({
    data: text
  })
}

render() {
  return (
    <div>
      <p>This message is from Child:{this.state.data}</p>
      <Child handleChange={ this.handleChange } />
    </div>
  )
}
ReactDOM.render(
<App/>,
document.querySelector("#app")
)

```

上述代码中有两个组件：子组件 Child 和父组件 App。子组件被父组件引用，在父组件中定义了一个 handleChange 事件，并通过 props 传给子组件让子组件调用该方法。子组件接收到来自父组件的 handleChange 方法，当子组件 input 框内输入的值 Value 发生变化时，就会触发 handleChange 方法，将该值传递给父组件，从而达到子对父通信。

 注意：一般情况下，回调函数会与 setState() 成对出现。

2.3.3 跨级组件通信

当组件层层嵌套时，要实现跨组件通信，首先会想到利用 props 一层层去传递信息。虽然可以实现信息传递，但这种写法会显得有点“啰嗦”，也不优雅。这种场景在 React 中，一般使用 context 来实现跨级父子组件通信。

context 的设计目的就是为了共享对于一个组件树而言是“全局性”的数据，可以尽量减少逐层传递，但并不建议使用 context。因为当结构复杂的时候，这种全局变量不易追溯到源头，不知道它是从哪里传递过来的，会导致应用变得混乱，不易维护。

context 适用的场景最好是全局性的信息，且不可变的，比如用户信息、界面颜色和主题制定等。

context 实现的跨级组件通信示例（React 16.2.0）：

（源码地址为 <https://jsfiddle.net/allan91/Lbecjy18/2/>）

```
// 子(孙)组件
class Button extends React.Component {
```

```

render() {
  return (
    <button style={{background: this.context.color}}>
      {this.props.children}
    </button>
  );
}

// 声明 contextTypes 用于访问 MessageList 中定义的 context 数据
Button.contextTypes = {
  color: PropTypes.string
};

// 中间组件
class Message extends React.Component {
  render() {
    return (
      <div>
        <Button>Delete</Button>
      </div>
    );
  }
}

// 父组件
class MessageList extends React.Component {
  // 定义 context 需要实现的方法
  getChildContext() {
    return {
      color: "orange"
    };
  }

  render() {
    return <Message />;
  }
}

// 声明 context 类型
MessageList.childContextTypes = {
  color: PropTypes.string
};
ReactDOM.render(
  <MessageList />,
  document.getElementById('container')
);

```

上述代码中，MessageList 为 context 的提供者，通过在 MessageList 中添加 childContextTypes 和 getChildContext() 和 MessageList。React 会向下自动传递参数，任何组织只要在它的子组件中（这个例子中是 Button），就能通过定义 contextTypes 来获取参数。如果 contextTypes 没有定义，那么 context 将会是个空对象。

context 中有两个需要理解的概念：一个是 context 的生产者（provider）；另一个是 context 的消费者（consumer），通常消费者是一个或多个子节点。所以 context 的设计模式是属于生产-消费者模式。在上述示例代码中，生产者是父组件 MessageList，消费者是孙组件

Button。

在 React 中，context 被归为高级部分（Advanced），属于 React 的高级 API，因此官方不推荐在不稳定的版本中使用。值得注意的是，很多优秀的 React 第三方库都是基于 context 来完成它们的功能的，比如路由组件 react-route 通过 context 来管理路由，react-redux 的<Provider/>通过 context 提供全局 Store，拖曳组件 react-dnd 通过 context 分发 DOM 的 Drag 和 Drop 事件等。

 注意：不要仅仅为了避免在几个层级下的组件传递 props 而使用 context，context 可用于多个层级的多个组件需要访问相同数据的情景中。

2.3.4 非嵌套组件通信

非嵌套组件就是没有包含关系的组件。这类组件的通信可以考虑通过事件的发布-订阅模式或者采用 context 来实现。

如果采用 context，就是利用组件的共同父组件的 context 对象进行通信。利用父级实现中转传递在这里不是一个好的方案，会增加子组件和父组件之间的耦合度，如果组件层次嵌套较深的话，不易找到父组件。

那么发布-订阅模式是什么呢？发布-订阅模式又叫观察者模式。其实很简单，举个现实生活中的例子：

很多人手机上都有微信公众号，读者所关注的公众号会不定期推送信息。

这就是一个典型的发布-订阅模式。在这里，公众号就是发布者，而关注了公众号的微信用户就是订阅者。关注公众号后，一旦有新文章或广告发布，就会推送给订阅者。这是一种一对多的关系，多个观察者（关注公众号的微信用户）同时关注、监听一个主体对象（某个公众号），当主体对象发生变化时，所有依赖于它的对象都将被通知。

发布-订阅模式有以下优点：

- **耦合度低：**发布者与订阅者互不干扰，它们能够相互独立地运行。这样就不用担心开发过程中这两部分的直接关系。
- **易扩展：**发布-订阅模式可以让系统在无论什么时候都可进行扩展。
- **易测试：**能轻易地找出发布者或订阅者是否会得到错误的信息。
- **灵活性：**只要共同遵守一份协议，不需要担心不同的组件是如何组合在一起的。

React 在非嵌套组件中只需要某一个组件负责发布，其他组件负责监听，就能进行数据通信了。下面通过代码来演示这种实现。

非嵌套组件通信示例：

- (1) 安装一个现成的 events 包：

```
npm install events --save
```

(2) 新建一个公共文件 events.js，引入 events 包，并向外提供一个事件对象，供通信时各个组件使用：

```
import { EventEmitter } from "events";
export default new EventEmitter();
```

(3) 组件 App.js：

```
import React, { Component } from 'react';
import ComponentA from "./ComponentA";
import ComponentB from "./ComponentB";
import "./App.css";
export default class App extends Component{
    render(){
        return(
            <div>
                <ComponentA />
                <ComponentB />
            </div>
        );
    }
}
```

(4) 组件 ComponentA：

```
import React, { Component } from "react";
import emitter from "./events";
export default class ComponentA extends Component{
    constructor(props) {
        super(props);
        this.state = {
            data: React,
        };
    }
    componentDidMount() {
        // 组件加载完成以后声明一个自定义事件
        // 绑定 callMe 事件，处理函数为 addListener() 的第 2 个参数
        this.eventEmitter = emitter.addListener("callMe", (data)=>{
            this.setState({
                data
            })
        });
    }
    componentWillUnmount() {
        // 组件销毁前移除事件监听
        emitter.removeListener(this.eventEmitter);
    }
    render(){
        return(
            <div>
                Hello, { this.state.data }
            </div>
        );
    }
}
```

(5) 组件 ComponentB:

```

import React, { Component } from "react";
import emitter from "./events";
export default class ComponentB extends Component{
    render(){
        const cb = (data) => {
            return () => {
                // 触发自定义事件
                // 可传多个参数
                emitter.emit("callMe", "World")
            }
        }
        return(
            <div>
                <button onClick = { cb("Hey") }>点击</button>
            </div>
        );
    }
}

```

当在非嵌套组件 B 内单击按钮后，会触发 `emitter.emit()`，并且将字符串参数 `World` 传给 `callMe`。组件 A 展示的内容由 `Hello, React` 变为 `Hello, World`。这就是一个典型的非嵌套组件的通信。

注意：组件之间的通信要保持简单、干净，如果遇到了非嵌套组件通信，这时候读者需要仔细审查代码设计是否合理。要尽量避免使用跨组件通信和非嵌套组件通信等这类情况。

2.4 组件的生命周期

生命周期（Life Cycle）的概念应用很广泛，特别是在政治、经济、环境、技术、社会等诸多领域经常出现，其基本涵义可以通俗地理解为“从摇篮到坟墓”（Cradle-to-Grave）的整个过程。在 React 组件的整个生命周期中，`props` 和 `state` 的变化伴随着对应的 DOM 展示。每个组件提供了生命周期钩子函数去响应组件在不同时刻应该做和可以做的事情：创建时、存在时、销毁时。

本节将从 React 组件的“诞生”到“消亡”来介绍 React 的生命周期。由于 React 16 版本中对生命周期有所修改，所以本节只介绍最新版本的内容，React 15 版本的生命周期不推荐使用，如需了解请读者自行查阅。这里以 React 16.4 以上版本为例讲解。

2.4.1 组件的挂载

React 将组件渲染→构造 DOM 元素→展示到页面的过程称为组件的挂载。一个组件

的挂载会经历下面几个过程：

- constructor();
- static getDerivedStateFromProps();
- render();
- componentDidMount()。

组件的挂载示例：

(源码地址为 <https://jsfiddle.net/allan91/n5u2wwjg/225709/>)

```
class App extends React.Component {
  constructor(props) {
    super(props);
    console.log("constructor")
  }
  static getDerivedStateFromProps() {
    console.log("getDerivedStateFromProps")
    return null;
  }

  // React 17 中将会移除 componentWillMount()
  // componentWillMount() {
  //   console.log("componentWillMount")
  // }
  render() {
    console.log("render")
    return 'Test'
  }
  // render() 之后构造 DOM 元素插入页面
  componentDidMount() {
    console.log("componentDidMount")
  }
}
ReactDOM.render(
<App/>,
document.querySelector("#app")
)
```

打开控制台，上述代码执行后将依次打印：

```
constructor
getDerivedStateFromProps
render
componentDidMount
```

constructor()是ES6中类的默认方法，通过new命令生成对象实例时自动调用该方法。其中的super()是class方法中的继承，它是使用extends关键字来实现的。子类必须在constructor()中调用super()方法，否则新建实例会报错。如果没有用到constructor()，React会默认添加一个空的constructor()。

getDerivedStateFromProps()在组件装载时，以及每当props更改时被触发，用于在props(属性)更改时更新组件的状态，返回的对象将会与当前的状态合并。

`componentDidMount()`在组件挂载完成以后，也就是 DOM 元素已经插入页面后调用。而且这个生命周期在组件挂载过程中只会执行一次，通常会将页面初始数据的请求在此生命周期内执行。

 注意：其中被注释的 `componentWillMount()` 是 React 旧版本中的生命周期，官方不建议使用这个方法，以后会被移除，因此这里不做介绍。

2.4.2 数据的更新过程

组件在挂载到 DOM 树之后，当界面进行交互动作时，组件 `props` 或 `state` 改变就会触发组件的更新。假如父组件 `render()` 被调用，无论此时 `props` 是否有改变，在 `render()` 中被渲染的子组件就会经历更新过程。一个组件的数据更新会经历下面几个过程：

- `static getDerivedStateFromProps()`;
- `shouldComponentUpdate()`;
- `componentWillUpdate() / UNSAFE_componentWillUpdate()`;
- `render()`;
- `getSnapshotBeforeUpdate()`;
- `componentDidUpdate()`。

数据更新可以分为下面两种情况讨论：

1. 组件自身 state 更新

组件自身 `state` 更新会依次执行：

```
shouldComponentUpdate() —> render() —> getSnapshotBeforeUpdate() —>
componentDidUpdate()
```

2. 父组件 props 更新

父组件 `props` 更新会依次执行：

```
static getDerivedStateFromProps() —> shouldComponentUpdate() —> render()
—> getSnapshotBeforeUpdate() —> componentDidUpdate()
```

相对于自身 `state` 更新，这里多了一个 `getDerivedStateFromProps()` 方法，它的位置是组件在接收父组件 `props` 传入后和渲染前 `setState()` 的时期，当挂载的组件接收到新的 `props` 时被调用。此方法会比较 `this.props` 和 `nextProps` 并使用 `this.setState()` 执行状态转换。

上面两种更新的顺序情况基本相同，下面来看看它们分别有何作用和区别：

- `shouldComponentUpdate(nextProps, nextState)`：用于判断组件是否需要更新。它会接收更新的 `props` 和 `state`，开发者可以在这里增加判断条件。手动执行是否需要去更新，也是 React 性能优化的一种手法。默认情况下，该方法返回 `true`。当返回值为

false 时，则不再向下执行其他生命周期方法。

- `componentDidUpdate(object nextProps, object nextState)`: 很容易理解，从字面意思就知道它们分别代表组件 `render()` 渲染后的那个时刻。`componentDidUpdate()` 方法提供了渲染后的 `props` 和 `state`。

注意：无状态函数式组件没有生命周期，除了 React 16.7.0 的新特性 Hooks。

2.4.3 组件的卸载（unmounting）

React 提供了一个方法：`componentWillUnmount()`。当组件将要被卸载之前调用，可以在该方法内执行任何可能需要清理的工作。比如清除计时器、事件回收、取消网络请求，或清理在 `componentDidMount()` 中创建的任何监听事件等。

组件的卸载示例：

```
import React, { Component } from "react";
export default class Hello extends Component {

  componentDidMount() {
    this.timer = setTimeout(() => {
      console.log("挂在 this 上的定时器");
    }, 500);
  }
  componentWillUnmount() {
    this.timer && clearTimeout(this.timer);
  }
}
```

2.4.4 错误处理

在渲染期间，生命周期方法或构造函数 `constructor()` 中发生错误时将会调用 `componentDidCatch()` 方法。

React 错误处理示例：

```
import React from "react";
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }
  static getDerivedStateFromError(error) {
    return { hasError: true };
  }
  componentDidCatch(error, info) {
    this.setState({
      hasError: true
    });
  }
}
```

```

    }
    render() {
      if (this.state.hasError) {
        return <h1>这里可以自定义一些展示，这里的內容能正常渲染。</h1>;
      }
      return this.props.children;
    }
}

```

在 componentDidCatch()内部把 hasError 状态设置为 true，然后在渲染方法中检查这个状态，如果出错状态是 true，就渲染备用界面；如果状态是 false，就正常渲染应该渲染的界面。

错误边界不会捕获下面的错误：

- 错误边界本身错误，而非子组件抛出的错误。
- 服务端渲染（Server side rendering）。
- 事件处理(Event handlers)，因为事件处理不发生在 React 渲染时，报错不影响渲染)。
- 异步代码。

2.4.5 老版 React 中的生命周期

老版本的 React 中还有如下生命周期：

- componentWillMount();
- componentWillReceiveProps();
- componentWillUpdate();

老版本中的部分生命周期方法有多种方式可以完成一个任务，但很难弄清楚哪个才是最佳选项。有的错误处理行为会导致内存泄漏，还可能影响未来的异步渲染模式等。鉴于此，React 决定在未来废弃这些方法。

React 官方考虑到这些改动会影响之前一直在使用生命周期方法的组件，因此将尽量平缓过渡这些改动。在 React 16.3 版本中，为不安全生命周期引入别名：

- UNSAFE_componentWillMount;
- UNSAFE_componentWillReceiveProps;
- UNSAFE_componentWillUpdate。

旧的生命周期名称和新的别名都可以在 React16.3 版本中使用。将要废弃旧版本的生命周期会保留至 React 17 版本中删除。

同时，React 官方也提供了两个新的生命周期：

- getDerivedStateFromProps();
- getSnapshotBeforeUpdate()。

getDerivedStateFromProps()生命周期在组件实例化及接收新 props 后调用，会返回一个对象去更新 state，或返回 null 不去更新，用于确认当前组件是否需要重新渲染。这个生

命周期将可以作为 `componentWillReceiveProps()` 的安全替代者。

`getDerivedStateFromProps()` 生命周期示例：

```
class App extends React.Component {
  static getDerivedStateFromProps(nextProps, prevState) {
    ...
  }
}
```

`getSnapshotBeforeUpdate()` 生命周期方法将在更新之前被调用，比如 DOM 被更新之前。这个生命周期的返回值将作为第 3 个参数传递给 `componentDidUpdate()` 方法，虽然这个方法不经常使用，但是对于一些场景（比如保存滚动位置）非常有用。配合 `componentDidUpdate()` 方法使用，新的生命周期将覆盖旧版 `componentWillUpdate()` 的所有用例。

`getSnapshotBeforeUpdate()` 生命周期（官方示例）：

```
class ScrollingList extends React.Component {
  constructor(props) {
    super(props);
    this.listRef = React.createRef();
  }
  getSnapshotBeforeUpdate(prevProps, prevState) {
    // 是否添加新项目到列表
    // 捕获滚动定位用于之后调整滚动位置
    if (prevProps.list.length < this.props.list.length) {
      const list = this.listRef.current;
      return list.scrollHeight - list.scrollTop;
    }
    return null;
  }
  componentDidUpdate(prevProps, prevState, snapshot) {
    // 如果有新值，就添加进新项目
    // 调整滚动位置，新项目不会把老项目推到可视窗口外
    // (这里的 snapshot 来自于 getSnapshotBeforeUpdate() 这个生命周期的返回值)
    if (snapshot !== null) {
      const list = this.listRef.current;
      list.scrollTop = list.scrollHeight - snapshot;
    }
  }
  render() {
    return (
      <div ref={this.listRef}>{/* ...contents... */}</div>
    );
  }
}
```

2.4.6 生命周期整体流程总结

React 组件的整个生命周期流程图如图 2.3 所示来描述。

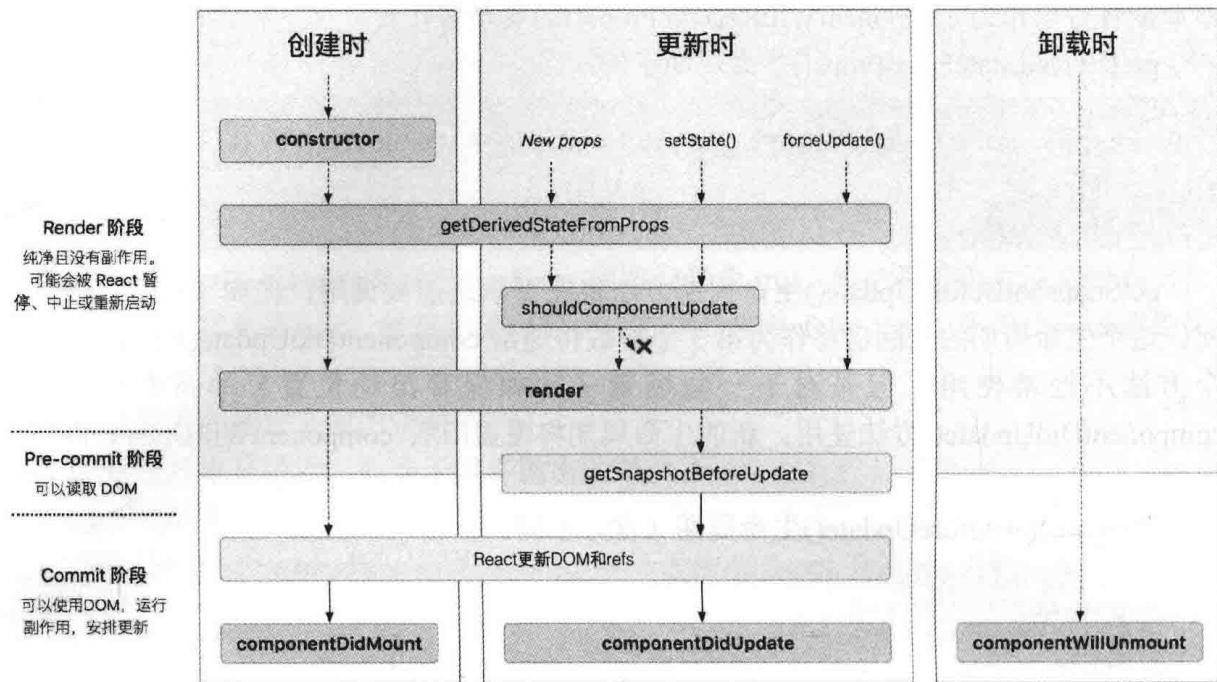


图 2.3 React 组件生命周期流程图

2.5 组件化实战训练——TodoList

前面章节中学习了如何配置 Webpack 来搭建 Hello World 项目，以及 React 的组件、组件通信和生命周期等。接下来继续基于前面的这个项目来实现一个简单的 TodoList，以此加深读者对组件化的了解。

在这个简单的 TodoList 项目中，需要实现：

- 通过 input 输入框输入 todo 内容；
- 单击 Submit 按钮将输入的内容展示在页面上。

在 1.5 节脚手架中，Webpack 的 loader 只对 JS 和 JSX 做了识别，现在需要在项目中加入 CSS 的相关 loader，目的是让 Webpack 识别和加载样式文件。

(1) 安装 CSS 的相关 loader：

```
npm install css-loader style-loader --save-dev
```

(2) 配置 Webpack 中的 loader：

```
var webpack = require("webpack");
var path = require("path");
const CleanWebpackPlugin = require("clean-webpack-plugin");
var BUILD_DIR = path.resolve(__dirname, "dist");
var APP_DIR = path.resolve(__dirname, "src");
```

```

const HtmlWebpackPlugin = require("html-webpack-plugin");
var config = {
  entry: APP_DIR + "/index.jsx",
  output: {
    path: BUILD_DIR,
    filename: "bundle.js"
  },
  module: {
    rules: [
      {
        test: /\.js|jsx$/,
        exclude: /node_modules/,
        use: {
          loader: "babel-loader"
        }
      },
      {
        test: /\.css$/,
        loader: 'style-loader!css-loader' // 只加载.css文件
      }
    ]
  },
  devServer: {
    port: 3000,
    contentBase: "./dist"
  },
  plugins: [
    new HtmlWebpackPlugin({
      template: "index.html",
      // favicon: 'theme/img/favicon.ico',
      inject: true,
      sourceMap: true,
      chunksSortMode: "dependency"
    }),
    new CleanWebpackPlugin(["dist"])
  ]
};
module.exports = config;

```

至此，TodoList 的项目脚手架配置结束。

(3) 接下来是相应组件的代码，入口页面 App.jsx 负责渲染组件头部 Header 和列表 ListItems，并在当前组件内部 state 维护列表的项目和输入的内容。

```

export default class App extends Component {
  constructor(props) {
    super(props);
    this.state = {
      todoItem: "",
      items: []
    }
  }
}

```

```

    );
}
render() {
  return (
    <div>
    </div>
  );
}
}

```

从上述代码可以看到 App 组件的 state 内有 todoItem 和 items。todoItem 用于存储输入框输入的值；items 用于存储输入框内提交的值，之后用于列表的渲染。

(4) 再来编写输入框输入内容时的 onChange 事件：

```

onChange(event) {
  this.setState({
    todoItem: event.target.value
  });
}
<input value={this.state.todoItem} onChange={this.onChange} />

```

从上述代码中可以看到，input 的值来自于 App 组件内的 state。用户每次输入后，onChange 事件监听其变化，然后调用 this.setState() 将改变的值实时写入 input 中展示。

(5) 表单提交：

```

onSubmit(event) {
  event.preventDefault();
  this.setState({
    todoItem: '',
    items: [
      ...this.state.items,
      this.state.todoItem
    ]
  });
}
<form className="form-wrap" onSubmit={this.onSubmit}>
  <input value={this.state.todoItem} onChange={this.onChange} />
  <button>Submit</button>
</form>

```

当单击 Submit 按钮时，输入框的值将通过表单提交的方式触发 onSubmit 事件，然后调用 this.setState() 添加输入框中的值到 items 数组，同时清空输入框。

(6) 将内容整理为 3 部分：头 Header、表单 form 和列表 ListItems。其中，Header 和 ListItems 各为一个组件。

./src/Header.js 内容如下：

```

import React from 'react';
const Header = props => (
  <h1>{props.title}</h1>
)

```

```
);
export default Header;
```

/src/ListItem.js 内容如下：

```
import React from 'react';
const ListItems = props => (
  <ul>
    {
      props.items.map(
        (item, index) => <li key={index}>{item}</li>
      )
    }
  </ul>
);
export default ListItems;
```

Header 和 ListItems 都是无状态函数式组件，接收父级./src/app.jsx 传入的 props 数据，用于各自的展示。

(7) 在入口./src/app.jsx 中引入组件：

```
import React, { Component } from "react";
import { render } from "react-dom";
+ import ListItems from "./ListItems";
+ import Header from "./Header";
```

(8) 引入样式：

```
import React, { Component } from "react";
import { render } from "react-dom";
import ListItems from "./ListItems";
import Header from "./Header";
+ import "./index.css";
```

至此，所有内容完成，此时这个项目的结构如下：

```
.
├── README.md
├── index.html
├── package-lock.json
└── package.json
└── src
  ├── Header.js
  ├── ListItems.js
  ├── app.jsx
  └── index.css
└── webpack.config.js
```

最终入口 app.jsx 文件的代码如下：

/src/app.jsx 内容如下：

```
import React, { Component } from "react";
import { render } from "react-dom";
```

```
import PropTypes from 'prop-types';           // 定义组件属性类型校验
import './index.css';
import ListItems from './ListItems';
import Header from './Header';
export default class App extends Component {
  constructor(props) {
    super(props);
    this.state = {
      todoItem: '',
      items: ["吃苹果", "吃香蕉", "喝奶茶"]
    };
    this.onChange = this.onChange.bind(this);
    this.onSubmit = this.onSubmit.bind(this);
  }
  // 输入框 onChange 事件
  onChange(event) {
    this.setState({
      todoItem: event.target.value
    });
  }
  // 表单提交按钮单击事件
  onSubmit(event) {
    event.preventDefault();
    this.setState({
      todoItem: '',
      items: [
        ...this.state.items,
        this.state.todoItem
      ]
    });
  }
  render() {
    return (
      <div className="container">
        <Header title="TodoList"/>
        <form className="form-wrap" onSubmit={this.onSubmit}>
          <input value={this.state.todoItem} onChange={this.onChange} />
          <button>Submit</button>
        </form>
        <ListItems items={this.state.items} />
      </div>
    );
  }
}
App.propTypes = {
  items: PropTypes.array,
  todoItem: PropTypes.string,
  onChange: PropTypes.func,
```

```
onSubmit: PropTypes.func  
};  
render(  
  <App />,  
  document.getElementById("app")  
)
```

本例最终的展示效果如图 2.4 所示。

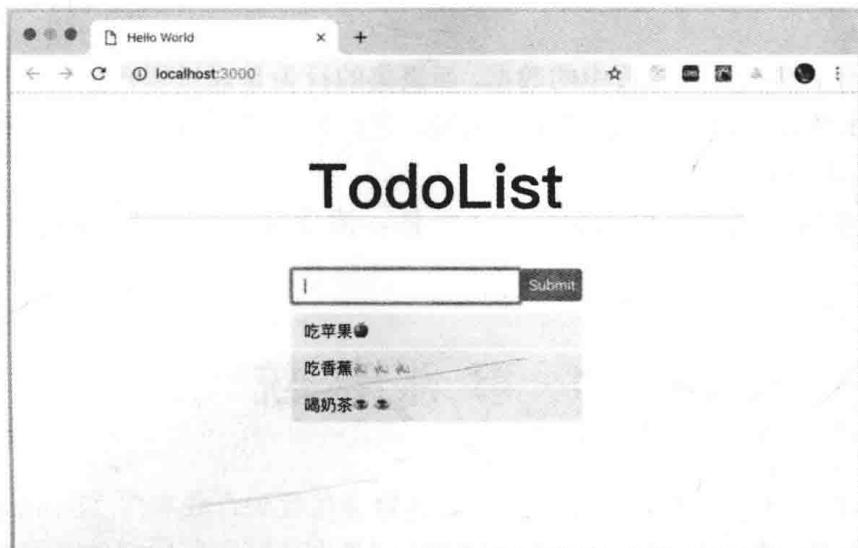


图 2.4 TodoList 展示效果

项目源码可在 GitHub 进行下载，地址是 <https://github.com/khno/react-comonent-todolist>。

第3章 React 的事件与表单

表单的作用主要是采集网页中的数据，而采集的行为是通过用户触发浏览器中的各种事件实现的。本章首先介绍 React 的事件系统，之后结合表单提交和样式处理，深入讲解 React 中表单处理的方式。

考虑到本章与其他章节没有很强的关联性，读者可以根据需要选择性地阅读本章内容。

3.1 事件系统

React 有自己的事件系统，定义的事件处理器会接收到合成事件（SyntheticEvent）的实例，并且所有事件都遵循 W3C 规范，这使得事件在不同浏览器中的表现一致。所有定义的事件都会被绑定到文档的根结点，当事件触发时，React 会将它映射到对应的组件元素，当组件卸载时会被自动移除。当然也可以通过使用 nativeEvent 访问浏览器原生事件对象。

React 事件系统和浏览器事件系统相比主要做了两件事：事件代理和事件自动绑定。这两个特性也正是合成事件的实现机制。

React 的事件书写方式与传统 HTML 事件监听器的书写基本相似。但 React 事件书写采用驼峰方式，如 onChange、onMouseMove、onKeyDown 等。比如给一个按钮添加单击事件：

```
// this.handleClick 表示当前组件中定义的事件
<button onClick={this.handleClick}>Click me</button>
```

3.1.1 合成事件的事件代理

跟传统的事件处理机制不同，React 把所有定义的事件都绑定到结构的最顶层。使用一个事件监听器 watch 所有事件，并且它内部包含一个映射表，记录了事件与组件事件处理函数的对应关系。当事件触发时，React 会根据映射关系找到真正的事件处理函数并调用。当组件被安装或被卸载时，对应的函数会被自动添加到事件监听器的内部映射表或者从表中删除。

3.1.2 事件的自动绑定

在 React 中，所有事件会被自动绑定到组件实例，并且会对该引用进行缓存，从而实现 CPU 和内存性能上的优化。

但如果使用 ES 6 Class 或无状态的函数式写法，默认不会绑定 this。因此在调用方法的时候需要手动绑定 this。

下面介绍几种绑定方式，以为 button 标签添加一个单击事件举例。

1. 在构造函数中使用 bind() 绑定 this

在构造函数 constructor() 内使用 bind() 绑定 this，等之后调用这个方法的时候，无须再次绑定。这是官方推荐的具有最佳性能的绑定方式。

在构造函数中使用 bind() 绑定 this 示例：

```
import React, { Component } from 'react';
import { render } from 'react-dom';
class Button extends Component {
  constructor(props) {
    super(props);
    // 在构造函数内完成 this 的绑定
    this.handleClick = this.handleClick.bind(this);
  }
  // 自定义的单击事件
  handleClick() {
    console.log('Clicked');
  }
  render() {
    return (
      <button onClick={this.handleClick}>           // 调用的时候不需要绑定 this
        Click me
      </button>
    );
  }
}
```

2. 使用箭头函数绑定 this

每次调用函数时去绑定 this，会生成一个新的方法实例，因此对性能会有一定影响。同时当这个函数传入低阶组件时，这些组件可能会再次 re-render。

使用箭头函数绑定 this 示例：

```
import React, { Component } from 'react';
import { render } from 'react-dom';
class Button extends Component {
  // 单击事件
  handleClick() {
```

```

        console.log('Clicked');
    }
    render() {
        return (
            <button onClick={()=>this.handleClick()}>      // 利用箭头函数绑定
                Click me
            </button>
        );
    }
}

```

3. 使用bind()方法绑定this

同方法 2 一样，每次调用函数的时候再去绑定 this，会对性能有一定影响。

使用 bind()方法绑定 this 示例：

```

import React, { Component } from 'react';
import { render } from 'react-dom';
class Button extends Component {
    // 单击事件
    handleClick(){
        console.log('Clicked');
    }
    render() {
        return (
            <button onClick={this.handleClick.bind(this)}>      // 用 bind()方法绑定
                Click me
            </button>
        );
    }
}

```

4. 使用属性初始化器语法绑定this

因为使用属性初始化器语法绑定 this 的方法在创建时使用了箭头函数，所以一创建就绑定了 this，因此在调用的时候无须再次绑定。

使用属性初始化器语法绑定 this 示例：

```

import React, { Component } from 'react';
import { render } from 'react-dom';
class Button extends Component {
    // 用箭头函数定义单击事件，直接绑定到当前组件
    handleClick=()=>{
        console.log('Clicked');
    }
    render() {
        return (
            <button onClick={this.handleClick}>      // 直接使用上面定义的方法
                Click me
            </button>
        );
    }
}

```

以上4种方式都能实现类定义组件this的绑定。

3.1.3 在React中使用原生事件

虽然React提供了完善的合成事件，但需要使用浏览器原生事件时，可以通过nativeEvent属性去获取和使用。由于原生事件需要绑定在真实的DOM中，所以一般在componentDidMount()生命周期阶段进行绑定操作。

 注意：在React中使用DOM原生事件记得必须要在组件卸载时手动移除，否则将可能出现内存泄漏问题。

这里以一个单击事件来举例。

使用原生事件示例：

```
import React, { Component } from 'react';
import { render } from 'react-dom';
class NativeEvent extends Component {
    // 真实DOM加载完成才能去绑定原生事件到节点
    componentDidMount() {
        this.refs.myDiv.addEventListener('click', handleClick, false)
    }
    // 单击事件
    handleClick = (e) => {
        console.log(e)
    }
    // 组件卸载时，需要移除
    componentWillUnmount() {
        this.refs.myDiv.removeEventListener('click')
    }
    render() {
        return (
            <div ref="myDiv">Click me</div>
        )
    }
}
ReactDOM.render(
    <NativeEvent />,
    document.getElementById('root')
);
```

3.1.4 合成事件与原生事件混用

即便合成事件系统已经如此完善，还是会有部分场景和需求需要使用原生事件。比如页面有个模态框，当单击模态框周围区域，需要将模态框隐藏。由于无法将模态框组件中的事件绑定到body上，因此只能通过原生事件在弹窗挂载到DOM完成后获取body，然

后通过原生事件绑定到 body 去实现。

合成事件与原生事件混用示例：

```

import React, { Component } from 'react';
import { render } from 'react-dom';
class Demo extends Component {
  constructor(props) {
    super(props);
    this.state = {
      isModalShow: true // 默认弹窗打开
    };
    // 将 this 绑定到单击事件
    this.handleClickModal = this.handleClickModal.bind(this);
  }
  componentDidMount() {
    document.body.addEventListener('click', e=>{
      // 判断是否在模态框内单击
      if(e.target && e.target.matches('div.modal')) {
        return;
      }
      // 单击模态框之外关闭
      this.setState({
        isModalShow: false
      });
    })
  }
  // 该生命周期表示组件将要卸载
  componentWillUnmount() {
    // 组件卸载之前先手动移除原生事件，避免出现内存泄漏
    document.body.removeEventListener('click');
  }
  render() {
    return (
      <div className="modal-wrapper">
        {
          this.state.isModalShow && // this.state.isModalShow 为真就显示 div
          <div className="modal">弹窗内容</div>
        }
      </div>
    )
  }
}
ReactDOM.render(
  <Demo />,
  document.getElementById('root')
);

```

在上述示例代码中，有一个细节需要特别注意。如果不使用 e.target 来判断单击的区域是否在弹窗内，就在示例代码中使用 React 的事件机制，如下：

```

clickInModal(e) {
  e.stopPropagation();
}
render() {
  return (

```

```

<div className="modal-wrapper">
{
  this.state.isModalShow &&           // this.state.isModalShow 为真就显
  <div className="modal" onClick={this.clickInModal}>弹窗内容</div>
}
</div>
}
)

```

以上代码是不能阻止弹窗关闭的。因为 React 的合成事件系统是事件代理，也就是事件并没有直接绑定在弹窗的 div 节点上。因此希望读者在 React 开发时尽量避免同时使用原生浏览器事件和 React 的合成事件。

本质上，React 的合成事件系统是属于原生浏览器事件的子集。所以原生事件中阻止冒泡可以阻止 React 的合成事件冒泡，反之在 React 合成事件中阻止冒泡无法阻止原生事件冒泡。

不管怎样，当有的场景 React 事件系统无法满足需求的时候，还是要使用原生事件。

3.2 表单 (Forms)

用户行为数据的采集是通过表单才得以实现的，用户的每一次输入都离不开表单。要管理好表单的状态并不容易，由于每次改变都会涉及缓存用户的录入状态，因此也给 React 表单的处理机制带来了一些特殊性。

React 的核心理念就是对状态的可控性，只要状态来自于 props 或 state，所有组件中被渲染出来的状态就会保持一致，对于表单状态的处理方式也正是如此。本节将介绍 React 是如何处理表单的。

3.2.1 受控组件

在 React 中强调的是对状态的可控管理，也就是对组件状态的可预知性和可测试性。表单状态是会随用户在表单中的输入、选择或勾选等操作不断发生变化的，每当发生变化时，将它们的状态都写入组件的 state 中，这种组件就被称为受控组件 (Controlled Component)。

这种类型的状态管理方式与 React 其他类型组件的模式一样，都是将状态交由 React 组件去控制。这也是 React 官方推荐的表单管理方式。

受控组件示例应用。

(源码地址为 <https://jsfiddle.net/n5u2wwjg/17710/>)

```

import React, { Component } from 'react';
import { render } from 'react-dom';

```

```

class MyForm extends Component {
  render() {
    return(
      <input type="text" value="Hello Form"/>
    )
  }
}
ReactDOM.render(
  < MyForm />, // 将该组件挂载到 id 为 root 的真实 DOM 去渲染
  document.getElementById('root')
);

```

本例效果如图 3.1 所示。

以上代码渲染了一个输入框，在 `input` 内 `value` 绑定了一个不可变的值。用户的任何输入都不会生效，那么在 React 中如何才能实时在界面反应用户的输入呢？

实时反应用户的输入示例：

(源码地址为 <https://jsfiddle.net/n5u2wwjg/23152/>)

```

import React, { Component } from 'react';
import { render } from 'react-dom';
class MyForm extends Component {
  constructor() {
    super();
    this.state = {
      inputValue: 'Hello Form!'
    };
  }
  // input 的 change 事件，每一次输入框的改变都会触发该事件
  handleChange(event) {
    this.setState({
      inputValue: event.target.value
    });
  }
  render() {
    return(
      <input type="text"
        value={this.state.inputValue}
        onChange={this.handleChange.bind(this)} // 这里用 bind() 方法绑定
        this (其他方法绑定 this 也可以)
      )
    )
  }
}
ReactDOM.render(
  < MyForm />,
  document.getElementById('root')
);

```



图 3.1 在 `input` 内 `value` 绑定了一个不可变值

从上面代码可以得知，只需要将该值绑定到组件的 `state`，然后通过 `onChange` 事件就能让该值随用户的输入实时发生变化。表单的数据都来源于 `state`。这种操作表单的方式看

似复杂，增加了代码量，但其实这样的处理方式能更好地控制数据流，这也正是 React 的状态（数据）驱动视图变化的设计思想。

注意：上面的 `onChange` 事件处理器正是 3.1.1 节所讲的 React 的事件合成机制，底层的操作就是拦截浏览器的原生 `change` 事件。在 `setState()` 被调用后，React 去计算差异，然后更新输入框的值。

视图展示的内容都是通过 JavaScript 控制的，比如当需要将用户输入的内容都转化为大写，可以这样写：

```
handleChange(event) {
  this.setState({
    inputValue: event.target.value.toUpperCase(),
  })
}
```

这样就能轻易控制用户输入内容的展示。除此之外，还可以对表单做诸如：输入字符长度限制、显示输入 HEX 值所代表的颜色，使用输入值去改变其他 UI 元素等行为的操作和控制。

3.2.2 非受控组件

非受控组件即无约束组件，React 强调的是对状态的可控管理，非受控组件是它的一种反模式。在大多数情况下，推荐使用受控组件来处理表单，当然这并不表明就不能使用非受控组件。比如有一个非常长的表单，希望用户先填写上面的表单域，填完了再去处理所有内容。

非受控组件示例：

(源码地址为 <https://jsfiddle.net/n5u2wwjg/23122/>)

```
import React, { Component } from 'react';
import { render } from 'react-dom';
class MyForm extends Component {
  constructor(props) {
    super(props);
    this.handleSubmit = this.handleSubmit.bind(this);
  }
  // 表单提交事件
  handleSubmit(event) {
    // 通过 event.target.name.value / event.target.email.value 获取已经输入的值
    console.log(event.target.name.value, event.target.email.value)
    event.preventDefault();
  }
  render() {
    return (
      <form onSubmit={this.handleSubmit}>
        <label>
          Name:
        </label>
        <input type="text" name="name" />
        <input type="text" name="email" />
        <input type="submit" value="Submit" />
      </form>
    );
  }
}
```

```

        <input type="text" />
    </label>
    <label>
        Email:
        <input type="mail" />
    </label>
    <button type="submit" value="Submit">Submit</button>
</form>
);
}
}
ReactDOM.render(
<MyForm />,
document.getElementById('root')
);

```

上面代码使用 onSubmit 来处理非受控组件表单，可以明显看出表单的值是不受 React 的 state 和 props 控制的。数据来源是 DOM 元素，这种方式比较适合非完全 React 集成的前端项目。

 注意：如果想为非受控组件添加默认初始值，可以使用 defaultValue 或 defaultChecked。

3.2.3 受控组件和非受控组件对比

非受控组件本身有自己的状态缓存，而受控组件由 React 的 state 状态进行缓存。在用户输入内容时，input 框中数值的每次改变都对应着调用一次 onChange 事件，调用的结果就是去触发一次 setState()。

非受控组件示例：

(源码地址为 <https://jsfiddle.net/n5u2wwjg/22876/>)

```

import React, { Component } from 'react';
import { render } from 'react-dom';
class MyForm extends Component {
    constructor(props) {
        super(props);
        this.state={
            value: '' // 在 state 中可以设置 input 的初始值，这里初始值为空
        }
    }
    render() {
        return (
            <input
                type='text'
                value={this.state.value}
                onChange={(e) => { // 方法也可以直接写在元素内，与写在外面用 this.fn 方法一样
                    this.setState({
                        value: e.target.value.toUpperCase(),
                    });
                }}
            />
        );
    }
}

```

```

    );
}

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

```

在 React 中，数据流是单向的，但是在表单的 `onChange` 事件中将数据回写到 `state` 的这种方式实现了双向数据绑定。当需要在表单输入的时候实时绑定到页面其他元素，使用可控性组件无疑是最佳选择，可参考下面的示例。

受控组件示例：

(源码地址为 <https://jsfiddle.net/n5u2wwjg/22872/>)

```

import React, { Component } from 'react';
import { render } from 'react-dom';
class MyForm extends Component {
  constructor(props) {
    super(props);
    this.state = {
      value: '' // 在 state 中可以设置 input 的初始值，这里初始值为空
    }
  }

  render() {
    return (
      <div>
        <input
          type='text'
          value={this.state.value}
          onChange={(e) => {
            this.setState({
              value: e.target.value.toUpperCase(),
            });
          }}
        />
        <p> 您刚刚输入了 : { this.state.value }</p> // 输入框输入的值被绑定到 p 标签
      </div>
    );
  }
}

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

```

本例效果如图 3.2 所示。

受控组件的优点是在用户输入和页面显示之间做了一道可控层，可以在用户输入之后和页面显示之前对输入值进行处理。上述案例中，用户输入的字母在显示前被转为大写字母，同时也在

HELLO REACT!

您刚刚输入了 : HELLO REACT!

图 3.2 表单输入实时绑定到页面其他元素

下面 p 标签内同步展示。

受控组件的缺点是需要为每个表单组件都绑定一个 `change` 事件，并且定义一个事件处理器去绑定表单值和组件的状态，而且每次表单值的改变都必定会调用一次 `onChange` 事件，带来了一些性能上的损耗。

即使如此，利大于弊，还是提倡在 React 中使用受控组件。

 注意：Model 层数据改变，View 层随之同步更新就是单向绑定；有了单向绑定的基础，反过来用户让 View 层代码改变，Model 层随之被更新就是双向数据绑定。

3.2.4 表单组件的几个重要属性

表单组件中含有几个重要属性，用于用户在页面交互时展示应用的状态：

- `value`: 用于`<input>`和`<textarea>`组件，类型为 `text`;
- `checked`: 用于`<checkbox>`和`<radio>`组件，类型为 `boolean`;
- `selected`: 用于 `select` 组件下面的`<option>`。

3.3 React 的样式处理

样式在应用中扮演着非常重要的角色，起到“美化”界面的作用。CSS 不算编程语言，它是对网页样式的一种描述。为了让其变得更像一门编程语言，从早先的 Less、Sass，到 PostCss 和 CSS in JS 都是为了解决这个问题。本节将带领读者了解如何在 React 中使用样式，以及 CSS Modules 相关概念。

3.3.1 基本样式设置

传统样式可以通过在标签内声明 `class` 或 `id` 名去定义，在 React 中也是一样。由于 `class` 在 JavaScript 属于保留字，为规避编译器的“误解”，JSX 语法中声明的标签属性中 `class` 必须使用 `className` 替换。

React 中的组件的样式设计示例：

`className` 支持常规 DOM 元素和 SVG 元素，例如：

```
<div className="button">Click me</div>
```

然后将定义的 css 文件在该组件顶部引用：

```
import './index.css'
class App extends React.Component {
  render() {
    return (
      <div className="button">Click me</div>
    )
  }
}
```

```

        <div className="btn">This is a demo</div>
    )
}
ReactDOM.render(
<App/>,
document.querySelector("#app")
)

```

之后通过 Webpack 之类的工具进行编译，就能将样式作用于该组件了。

当然读者也可以使用行内样式。但在 React 中，行内样式 style 属性并不是以字符串形式被接收的，而是以一个带有驼峰命名的对象形式出现的。这个样式对象的 key 为驼峰命名规则的样式描述，对应的值通常是一个字符串或数字。这种设计方式可以与 DOM 中样式的命名保持一致，也有助于弥补 XSS 安全漏洞。

React 中的行内样式示例：

(源码地址为 <https://jsfiddle.net/allan91/Lbecjy18/4/>)

```

class App extends React.Component {
  render() {
    const divStyle = {
      color: 'red',
      fontSize: 12,
      backgroundColor: 'yellow'
    }
    return (
      <div style={divStyle}>This is a demo</div>
    )
  }
}
ReactDOM.render(<App/>, document.querySelector("#app"))

```

或：

```

const divStyle = {
  color: 'blue',
  backgroundImage: 'url(' + imgUrl + ')',
};
function HelloWorldComponent() {
  return <div style={divStyle}>Hello World!</div>;
}

```

 注意：数字用于描述像素单位的值时不需要添加 px，React 支持自动将其转换为 px 单位，如 fontSize、height 和 lineHeight 等。

并不是所有的样式属性都能被转换为像素字符串，某些属性不行，例如 zoom、order 和 flex，更多这类属性参考如下：

```

/**
 * CSS 中接受数字但不以 px 为单位的属性
 */
var isUnitlessNumber = {
  animationIterationCount: true,

```

```
borderImageOutset: true,
borderImageSlice: true,
borderImageWidth: true,
boxFlex: true,
boxFlexGroup: true,
boxOrdinalGroup: true,
columnCount: true,
columns: true,
flex: true,
flexGrow: true,
flexPositive: true,
flexShrink: true,
flexNegative: true,
flexOrder: true,
gridRow: true,
gridRowEnd: true,
gridRowSpan: true,
gridRowStart: true,
gridColumn: true,
gridColumnEnd: true,
gridColumnSpan: true,
gridColumnStart: true,
fontWeight: true,
lineClamp: true,
lineHeight: true,
opacity: true,
order: true,
orphans: true,
tabSize: true,
widows: true,
zIndex: true,
zoom: true,

// SVG-related 属性
fillOpacity: true,
floodOpacity: true,
stopOpacity: true,
strokeDasharray: true,
strokeDashoffset: true,
strokeMiterlimit: true,
strokeOpacity: true,
strokeWidth: true,
};
```

3.3.2 CSS Modules 样式设置

近几年由于 ES 5/ ES 6 的快速普及和 Babel 和 Webpack 等工具的快速发展，CSS 的进化被远远抛到了后面，也因此 CSS 逐渐成为前端工程化的障碍点。1996 年 CSS 1 发布，规范的目的是将文档与文档内容分开，声明中的操作词是“文件”。当时的网络并不是主流，内容仍然基于文档，少量的样式表易于解耦和维护。

现代互联网应用中，很少考虑文档中的内容。现代网络以高度动态的 Web 应用程序为主。这种规模的建立和维护 CSS 给开发团队带来了独特的挑战。克服这些挑战需要平衡团队的纪律、工具和框架。在现代的前端培训和学习中会经常提到 JavaScript 全局变量的危险性，但很多人忽视了样式的全局性也带有危险性。这也是 CSS 模块化所要解决的问题。前端模块化发展要继续进化的前提就是解决 CSS 模块化方案。

目前 CSS 模块化的解决方案有很多，但主流的有两类：

一类是利用 JS 或 JSON 来写样式，Radium，jsxstyle，react-style 属于这一类。它们能为 CSS 提供 JS 一样强大的模块化能力，但不能使用当前成熟的 CSS 预处理器 Less、Sass 和 PostCss 等，像: hover，active 等伪类的处理也比较复杂。

另一类是利用 JS 来处理 CSS，这种 CSSinJS 的方案相当于完全抛弃传统的 CSS 写法，转而在 JS 中以对象的形式来书写 CSS，最具代表性的就是 CSS Modules。它可以结合当前成熟的 CSS 生态和利用 JS 的模块化能力。这种解决方案学习成本不高，有 Webpack 就能在项目中使用。

使用 CSS Modules，需要在 Webpack 中配置：

```
{
  test: /\.css$/,
  loader: "style!css?module&localIdentName=[hash:base64:5]-url"
}
```

如果使用 Sass 或 Less 只需要添加一个它们的 loader 就可以了。CSS Modules 内部通过 ICSS 来实现转换，但书写方式还是正常写法：

```
/* index.css */
.root {
  color: green;
}
.text {
  font-size: 16px;
}
```

将 CSSModules 导入 JS 模块内：

```
import styles from './index.css';
import React from 'react';
export default class Demo extends React.Component {
  render() {
    return (
      <div className={styles.root}>
        <p className={styles.text}>This is a demo!</p>
      </div>
    );
  }
};
```

然后渲染到页面，将看到类似如下代码：

```
<p class="p-text-abc4567"> ... </p>
```

其中生成的 class 名 p—text-abc4567 是 CSS Modules 按照 localIdentName 自动生成的。经过混淆处理可以避免代码重复，所以在使用 CSS Modules 项目中书写的 class 名称可以保证是唯一的。

这样写的好处有：

- 保留了很好的组件复用性；
- 消除了全局命名的问题，在组件的 index.css 中可以随意起名字，不用担心命名冲突；
- 和 React 结合得很好；
- 很方便地按需加载。

使用了 CSS Modules 就相当于默认给每个样式名外层包裹了一个:local, .className{...} 等价于:local(.className){...}，以此来实现样式的局部化。当需要让样式全局生效时，可以使用:global 包裹。例如：

```
/* 定义全局样式 */
:global(.className) {
  color: green;
}

/* 定义多个全局样式 */
:global {
  .className1 {
    color: green;
  }
  .className2 {
    color: green;
  }
}
```

在 CSS Modules 中，可以使用 compose 来处理样式的复用，这也是其唯一的处理方式：

```
/* 公共样式 */
.common {
  /* 公用样式 */
}

.textCenter {
  compose: common;
  /* .normal 的其他样式 */
}

import styles from './styles.css';
element.innerHTML = '<div class="' + styles.textCenter + '">This is a demo!</div>'
```

最终将被渲染为：

```
<div class="div-common-abc147 div-textCenter-abc147">This is a demo!</div>
```

正是由于在该 div 元素的.textCenter 中引入了公共样式，所以会在编译后生成两个 class。当需要将 compose 所需文件从外部导入时，可以这样做：

```
otherClassName {  
  compose: className from "./style.css";  
}
```

样式命名规范：对于本地类名，建议使用驼峰（camelCase）命名，但不强制执行。当使用短横线（kebab-case）命名时，将有可能在编译时导致一些意外情况发生。要使用短横线命名法可以使用括号表示法来替代，比如 style['class-name']，不过还是推荐读者使用驼峰命名，更简洁。

第 4 章 React+Redux 的数据流管理

Flux 是建立客户端 Web 应用的前端架构，它通过利用一个单向的数据流补充了 React 的组合视图组件。确切来说，Flux 是 2014 年开源的一套用于构建用户界面的应用程序架构（Application Architecture for Building User Interface）。它跟 React 本身没有必然联系，也可以作用于其他框架或组件上。本章就来重点介绍 Flux 的原理、使用，以及它的衍生——Redux。

4.1 Flux 架构

Flux 是一种应用架构，或者说是一种思想，本节通过介绍 MVC 和 MVVM 来让读者了解 Flux 的由来及其运作流程。

4.1.1 MVC 和 MVVM

1. MVC简介

MVC（Model View Controller），是 Model（模型）—View（视图）—Controller（控制器）的缩写。MVC 不是框架，不是设计模式，也不是软件架构，而是一种架构模式。它们彼此之间是有区别的：

- 框架（Framework）：是一个系统的可重用设计，表现为一组抽象的可交互方法。它就像若干类的构成，涉及若干构件，以及构件之间的相互依赖关系、责任分配和流程控制等。比如，C++语言的 QT、MFC、GTK，Java 语言的 SSH、SSI，PHP 语言的 Smarty（MVC 模式），Python 语言的 Django（MTV 模式）等。
- 设计模式（Design Pattern）：是一套被反复使用、多数人知晓的、经过分类的代码设计经验总结。其目的是为了代码的可重用性、让代码更容易被他人理解、保证代码的可靠性。比如，工厂模式、适配器模式和策略模式等。
- 软件架构（Software architecture）：是一系列相关的抽象模式，用于指导大型软件系统各个方面设计。软件架构是一个系统的草图，软件体系结构是构建计算机软件实践的基础。

- 架构模式（风格）：也可以说成框架模式，一个架构模式描述软件系统里基本的结构组织或纲要。架构模式提供一些事先定义好的子系统，指定它们的责任，并给出把它们组织在一起的法则和指南。一个架构模式常常可以分解成很多个设计模式的联合使用。MVC 模式就属于架构模式，还有 MTV、MVP、CBD 和 ORM 等。

框架与设计模式相似，但根本上是不同的。设计模式是对某种环境中反复出现的问题及解决该问题方案的描述，比框架更加抽象；框架可以用代码表示，也能直接执行或复用，而对模式而言只有实例才能用代码表示；设计模式是比框架更小的元素，一个框架中往往含有一个或多个设计模式，框架总是针对某一特定应用领域，但同一模式却可适用于各种应用。可以说，框架是软件，而设计模式是软件的知识。

介绍完框架、设计模式、软件架构和架构模式之后，再来看看属于架构模式的 MVC 示意图，如图 4.1 所示。

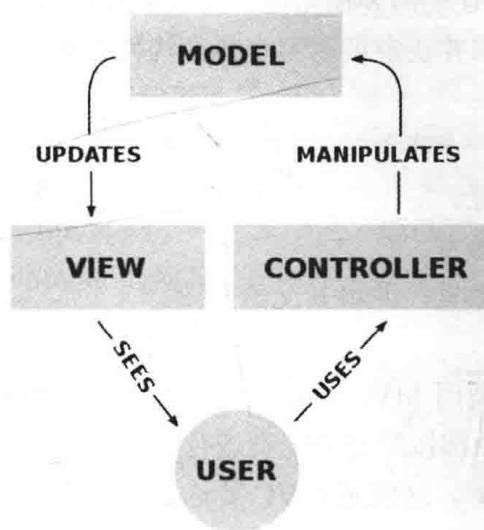


图 4.1 MVC 示意图

先来看一下维基百科中对 MVC 的解释：

Model-view-controller is commonly used for developing software that divides an application into three interconnected parts. This is done to separate internal representations of information from the ways information is presented to and accepted from the user. The MVC design pattern decouples these major components allowing for efficient code reuse and parallel development.

简单翻译成中文就是：

MVC 通常应用于软件开发中将应用程序划分为 3 个互连部分的软件。这是为了将信息的内部表示与信息呈现给用户并从用户接受的方式分开。MVC 设计模式将这些主要组件分离开，从而实现高效的代码重用和并行开发。

MVC 把软件工程分为三部分：模式、视图和控制器。这三者各有分工：

- 视图（View）：负责图形界面展示。前端 View 负责构建 DOM 元素，视图层是用户最直接接触到的，用户与之产生交互。View 一般都对应着一个 Model，可以读取或编辑 Model。View 接受 JSON 数据格式的数据。
- 模式（Model）：负责数据管理，保存着应用的数据和后端交互同步的数据。Model 不涉及表现层和用户界面。它的数据被 View 所使用，当 Model 发生改变时，它会通知视图作出对应的改变。
- 控制器（Controller）：作用是连接 View 和 Model。用户在 View 层的操作会通过 Controller 实施作用到 Model。前端 MVC 中对于 Controller 的划分界限不是特别清晰。Controller 的本质就是为了控制程序中 Model 与 View 的关联。

MVC 的出现是为了让应用的业务逻辑、数据和界面显示分离的方法来组织代码。其中，Controller 的存在是为了让视图和模型能同步。Controller 本身不输出任何内容，也不做任何处理，它只是接收请求并决定调用哪个模型构件去处理请求，然后确定用哪个视图来显示返回的数据。

在前端开发的应用中，一个事件的发生需要经历以下几步：

- (1) 用户与应用产生交互。
- (2) 事件处理器事件被触发。
- (3) 控制器从模型请求数据，并将其交给视图。
- (4) 视图呈现。

而现实中，这个流程不使用 MVC 也能实现，但为什么采用 MVC 去实现它呢？这是因为 MVC 能将它的每个部分都按照对应职责进行了分类、解耦，这样就能更好地对每个部分进行独立开发测试和维护。这就是 MVC 要达到的效果！

MVC 具有以下优点：

- (1) 独立开发

MVC 支持快速并行的开发。如果使用 MVC 模型开发任何特定的 Web 应用程序，那么有可能一个程序员在视图上工作，而另一个程序员在控制器上工作（创建 Web 应用程序的业务逻辑）。因此，使用 MVC 模型开发的应用程序可以比使用其他开发模式开发的应用程序更快。

- (2) 可重复性

在 MVC 模型中，我们可以为模型创建多个视图。面对当下应用程序的新需求日益增加，采用 MVC 开发无疑是一个很好的解决方案。而且在这种方法中，代码重复性小，因为它将数据和业务逻辑从显示中分离出来使用。

- (3) 低耦合

对于任何 Web 应用程序，用户界面往往会根据公司的业务规则频繁更改。很明显，开发人员可以频繁更改 Web 应用程序，例如更改颜色、字体、屏幕布局，以及为手机或

平板电脑添加新设备支持等。而且，在MVC模式中添加新类型的视图非常容易，因为模型部分不依赖于视图部分。因此，模型中的任何更改都不会影响整个体系结构。

诸如上述的各类优点，MVC这种架构模式在前端开发中可以高效、轻松完成各类复杂应用开发。最重要的一点是：它管理多个视图的能力能使MVC成为Web应用程序开发的最佳架构模式。

2. MVVM简介

MVVM（图4.2）的演化历史：

2004年，Martin Fowler发表了*Presentation Model*（以下简称PM）的文章，它实现了从视图层中分离行为和状态。2005年，John Gossman在其博客公布*Introduction to Model/View/ViewModel pattern for building WPF apps*（<https://blogs.msdn.microsoft.com/johngossman/2005/10/08/introduction-to-modelviewmodel-pattern-for-building-wpf-apps/>）一文。但MVVM与Martin Fowler所说的PM模式其实是完全相同的，PM模式是一种与平台无关的创建视图抽象的方法，而John Gossman的MVVM是专门用于WPF框架，简化用户界面的创建提出模式；可以认为MVVM是在WPF平台上对于PM模式的实现。

MVVM架构模式由微软于2005年提出，它从诞生就与WPF（微软用于处理GUI软件的框架）框架联系紧密，为其提供了一套优雅的、遵循PM模式的解决方案。从字面意思看它是Model-View-ViewModel的缩写，而本质上还是MVC的改进版。其设计思想是关注Model变化，让MVVM框架去自动更新DOM。

这也是AngularJS的核心，它实现了数据绑定（Data Binding），将原来MVC中的C（Controller）用VM（ViewModel）来取代，相当于对MVC做了拓展。目前MVVM的前端框架有Angular、Vue、Backbone.js和Ember等。虽然是Martin Fowler在2004年就提出的概念，但到今天PM依旧非常先进。

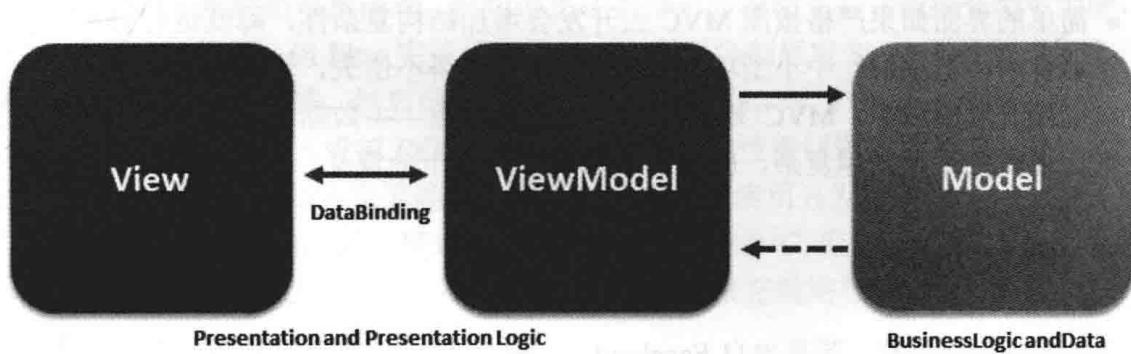


图4.2 MVVM示意图

除了熟悉的这3部分之外，其实在MVVM的实现中还引入了一个隐式的Binder层，如图4.3所示，声明式数据和命令绑定在MVVM模式中就是通过它实现的。

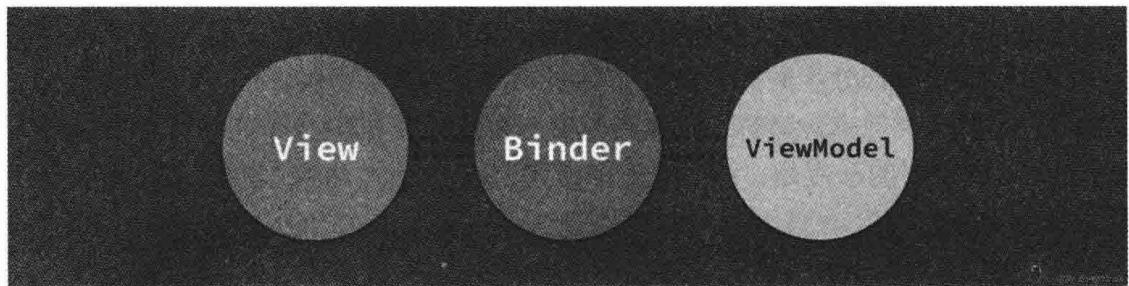


图 4.3 Binder 层

开发人员需要处理 View 层和 ViewModel（展示模型层）层之间的同步状态，于是使用隐式 Binder 和 XAML（视图层）文件来完成这两者之间的双向绑定。

不管是 MVVM 还是 PM，最重要的不是如何同步视图，以及同步展示模型和视图模型的状态，也不是使用观察者模式、双向数据绑定或是其他机制，最重要的是展示模型和视图模型创建了一个视图的抽象，将视图中的状态和行为抽离成了一个新的抽象。

3. 总结

经过几十年的发展和演变，从 MVC 架构模式到 MVVM 架构模式，出现了诸多的实现方式，但本质是不变的。其核心是关注点的分离，需要将不同的模块和功能分布到合适的位置中，减少依赖和耦合。

当然，MVC 并不是完美的，在前端领域 MVC 的缺点如下：

- 视图对模型数据的低效率访问；
- 视图与控制器之间过于紧密的连接；
- 没有明确的定义，它的内部原理比较复杂，所以要完全理解 MVC 需要花费很多时间；
- 简单的界面如果严格按照 MVC 去开发会增加结构复杂性，降低运行效率，并且不适合中小型项目。中小型项目中使用 MVC 会得不偿失，增加项目复杂度。

在前端开发领域中，MVC 还拥有一个致命的缺点——数据流管理混乱。随着项目越来越大、逻辑关系越来越复杂，就会明显发现数据流十分混乱，难以维护。

4.1.2 Flux 介绍

Flux 同 React 一样，都是出自 Facebook。Flux 的核心思想是利用单向数据流和逻辑单向流来应对 MVC 架构中出现状态混乱的问题。起初由于 Facebook 公司业务庞大，代码也随着业务的复杂和增多变得非常庞大，代码由此变得脆弱及不可预测，尤其对于刚接触新业务的开发者来说，这是非常严重的问题。于是 Facebook 的工程师认为 MVC 架构无法满足他们对业务拓展的需求，于是开发了 Flux。Flux 是基于 Dispatcher 的前端应用架构模式，其名字来自拉丁文的 Flow。Flux 模型如图 4.4 所示。

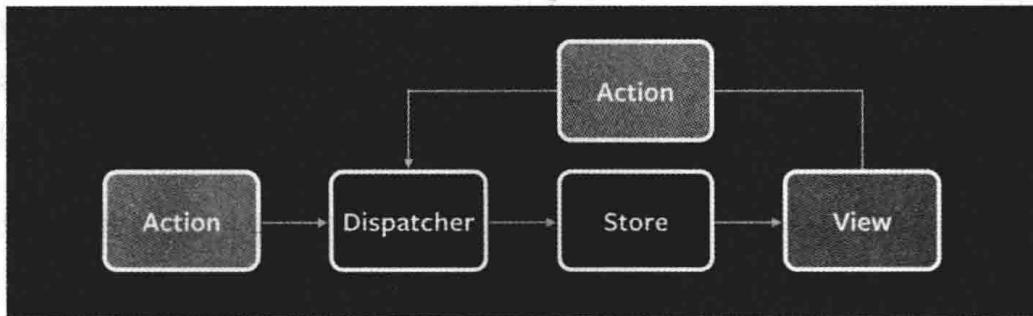


图 4.4 Flux 模型

Flux 由 3 部分组成：Dispatcher、Store 和 View。其中，Dispatcher（分发器）用于分发事件；Store 用于存储应用状态，同时响应事件并更新数据；View 表示视图层，订阅来自 Store 的数据，渲染到页面。

Flux 的核心是单向数据流，其运作方式是：

Action -> Dispatcher -> Store -> View

整个流程如下：

- (1) 创建 Action (提供给 Dispatcher)。
- (2) 用户在 View 层交互 (比如单击事件) 去触发 Action。
- (3) Dispatcher 收到 Action，要求 Store 进行相应的更新。
- (4) Store 更新，通知 View 去更新。
- (5) View 收到通知，更新页面。

从上面的流程可以得知，Flux 中的数据流向是单向的，不会发生双向流动的情况，从而保证了数据流向的清晰脉络。而 MVC 和 MVVM 中数据的流向是可以双向的，状态在 Model 和 View 之间来回“震荡”，很难追踪和预测数据的变化。

Flux 做到了数据的单向流动，要知道想让数据单向流动并且让数据变动可追溯，这是一件非常困难的问题。但 Flux 完美地解决了，它利用强制规定 Store 不可被直接修改从而保障了数据的纯粹和干净。然后结合 React 作为 View 层，每当 Store 改变时，页面就重新以最小的代价进行渲染 (虚拟 DOM 可以让页面渲染的性能问题不做考虑)。仅此，Flux 配合 React 的架构方式能够完美胜任各类复杂的中大型应用开发场景。

至此还无法断言 Flux 架构模式在任何场景都优于 MVC。但 Flux 这种全新的颠覆性设计思想目前还处于早期阶段，不得不让大家对它的未来充满期待。

4.1.3 深入 Flux

1. Dispatcher简介

Dispatcher 是 Flux 中的核心概念，它是一个调度中心，管理着所有的数据流，所有事

件通过它来分发。Dispatcher 处理 Action（动作）的分发，维护 Store 之间的依赖关系，负责处理 View 和 Store 之间建立 Action 的传递。

Dispatcher 用于将 Action 分发给 Store 注册的回调函数，它与普通的发布-订阅模式（pub-sub）有两点不同：

- 回调函数不是订阅到某一个特定的事件或频道，每个动作会分发给所有注册的回调函数；
- 回调函数可以指定在其他回调函数之后调用。

Dispatcher 通常是应用级单例，所以一个应用中只需要一个 Dispatcher 即可，示例如下：

```
var Dispatcher = require('flux').Dispatcher;
var AppDispatcher = new Dispatcher();
```

从代码可以看出，AppDispatcher 是来自 dispatcher.js 生成的实例。dispatcher.js 源码也很简单，总体结构如下：

```
'use strict';
var invariant = require('invariant');
export type DispatchToken = string;
var _prefix = 'ID_';
// Dispatcher 用于向注册的回调函数广播 payloads
class Dispatcher<TPayload> {
  ...
  constructor() {
    ...
  }
  register(callback: (payload: TPayload) => void): DispatchToken {
    ...
  }
  unregister(id: DispatchToken): void {
    ...
  }
  waitFor(ids: Array<DispatchToken>): void {
    ...
  }
  dispatch(payload: TPayload): void {
    ...
  }
  isDispatching(): boolean {
    ...
  }
  _invokeCallback(id: DispatchToken): void {
    ...
  }
  _startDispatching(payload: TPayload): void {
    ...
  }
  _stopDispatching(): void {
    ...
  }
}
module.exports = Dispatcher;
```

先看上述代码中的第3行：

```
export type DispatchToken = string;
```

这是一个类型定义，一个Flux回调代码块有一个唯一的Token，而这个Token的作用就是让别人调用你的代码。如何确保唯一性呢？来看下面的代码：

```
var id = _prefix + this._lastID++;
```

没错，就是这么简单，JS是单线程的，这种自增ID的做法很高效也很安全。

再来看Dispatcher这个class中包含了哪些信息：

```
_callbacks: {[key: DispatchToken]: (payload: TPayload) => void};
_isDispatching: boolean;
_isHandled: {[key: DispatchToken]: boolean};
_isPending: {[key: DispatchToken]: boolean};
_lastID: number;
_pendingPayload: TPayload;
```

以上代码中，Dispatcher包含的信息如下：

- callbacks: DispatchToken 和函数回调的字典；
- isDispatching: 展示当前 Dispatcher 是否处于 Dispatch 状态；
- isHandled: Token 检测一个函数是否被处理过；
- isPending: Token 检测一个函数是否被提交 Dispatcher 过；
- lastID: 最近一次被加入 Dispatcher 的函数体的唯一 ID，即 DispatchToken；
- pendingPayload: 需要传递给调用函数的数据。

接下来介绍Dispatcher类的函数。

(1) register()函数：用于注册一个回调函数进入 Dispatch，同时为这个 callback 生成 DispatchToken，并加入字典。

```
register(callback: (payload: TPayload) => void): DispatchToken {
  var id = _prefix + this._lastID++;
  this._callbacks[id] = callback;
  return id;
}
```

(2) unregister()函数：从字面意思就可以理解，是取消注册。它可以通过 DispatchToken 将 callback 从字典中删除。

```
unregister(id: DispatchToken): void {
  invariant(
    this._callbacks[id],
    'Dispatcher.unregister(...): "%s" does not map to a registered callback.',
    id
  );
  delete this._callbacks[id];
}
```

 注意：invariant是一个用于描述错误的Node包，接受两个参数，第1个参数是条件，第2个参数是报错信息描述。

(3) `waitFor()`函数：总体来说，它是一个等待函数，当执行到某些依赖的条件不满足时，就去等待它完成。

```

waitFor(ids: Array<DispatchToken>): void {
    // 判断当前是否处于 Dispatching，不处于 Dispatching 状态就不执行当前函数
    invariant(
        !this._isDispatching,
        'Dispatcher.waitFor(...): Must be invoked while dispatching.'
    );
    // 从 DispatchToken 的数组中进行遍历，如果遍历到的 DispatchToken 处于 Pending 状态，就暂时跳过
    for (var ii = 0; ii < ids.length; ii++) {
        var id = ids[ii];
        if (this._isPending[id]) {
            invariant(
                !this._handled[id],
                'Dispatcher.waitFor(...): Circular dependency detected while ' +
                'waiting for \'' + id + '\''
            );
            continue;
        }
        // 检查 Token 对应的 callback 是否存在
        invariant(
            this._callbacks[id],
            'Dispatcher.waitFor(...): \'' + id + '\' does not map to a registered callback.'
        );
        // 调用对应 Token 的 callback 函数
        this._invokeCallback(id);
    }
}

```

(4) `dispatch()`函数：Dispatcher 用于分发 payload 的函数。首先判断当前 Dispatcher 是否已经处于 Dispatching 状态中了。如果是，就不去打断。然后通过 `_startDispatching` 更新状态。更新状态结束以后，将非 Pending 状态的 callback 通过 `_invokeCallback` 执行（Pending 在这里的含义可以简单理解为还没准备好或者被卡住了）。所有任务执行完成以后，通过 `_stopDispatching` 恢复状态。

```

dispatch(payload: TPayload): void {
    invariant(
        !this._isDispatching,
        'Dispatcher.dispatch(...): Cannot dispatch in the middle of a dispatch.'
    );
    this._startDispatching(payload);
    try {
        for (var id in this._callbacks) {
            if (this._isPending[id]) {
                continue;
            }
            this._invokeCallback(id);
        }
    }
}

```

```

    } finally {
      this._stopDispatching();
    }
}

```

(5) `_startDispatching()`函数：该函数将所有注册的 `callback` 的状态都清空，并标记 `Dispatcher` 的状态进入 `Dispatching`。

```

_startDispatching(payload: TPayload): void {
  for (var id in this._callbacks) {
    this._isPending[id] = false;
    this._isHandled[id] = false;
  }
  this._pendingPayload = payload;
  this._isDispatching = true;
}

```

(6) `_stopDispatching()`函数：删除传递给 `callback` 的参数 `_pendingPayload`，让当前 `Dispatcher` 不再处于 `Dispatch` 状态。

```

_stopDispatching(): void {
  delete this._pendingPayload;
  this._isDispatching = false;
}

```

`Dispatcher` 要点小结：

- `register(function callback)`: 注册回调函数，返回一个可被 `waitFor()` 使用的 Token;
- `unregister(string id)`: 通过 Token 移除回调函数;
- `waitFor(array ids)`: 在指定的回调函数执行之后才执行当前回调，该方法只能在分发动作的回调函数中使用;
- `dispatch(object payload)`: 给所有注册的回调函数分发一个 payload;
- `isDispatching()`: boolean 值，返回 `Dispatcher` 当前是否处在分发的状态。

在实际开发应用中需要关注的是 `.register(callback)` 和 `.dispatcher(action)` 这两个方法。`register()` 方法用来注册监听器，`dispatch()` 方法用来分发 Action。

2. Action简介

Action 可以看作是一个交互动作，改变应用状态或 View 的更新，都需要通过触发 Action 来实现。Action 执行的结果就是调用了 Dispatcher 来处理相应的事情。Action 是所有交互的入口，改变应用的状态或者有 View 需要更新时就需要通过 Action 实现。

Action 是一个 JavaScript 对象，用来描述一个行为，里面包含了相关的信息。例如，要创建一篇文章的这个动作，它的 Action 写法如下：

```

{
  actionName: "create-post",
  data: {
    content: "new post"
  }
}

```

Action 这个对象主要由两部分构成：type（类型）和 payload（载荷）。type 是一个字符串常量，用于表示这个动作的标记。所谓的动作（Action）就是用来封装传递数据的，它仅仅是一个简单的对象。下面来看如何创建一个 Action：

```
import AppDispatcher from './AppDispatcher';
var Actions = {
  addTodo(text) {
    AppDispatcher.dispatch({
      type: 'ADD_TODO',
      text,
    });
  },
  // 其他 Actions
  ...
};
```

上述代码中，Actions 中的 addTodo()方法使用了 AppDispatcher，把动作 ADD_TODO 派发到 Store。

3. Store 和 View 简介

Store 包含应用状态和逻辑，不同的 Store 管理不同的应用状态。Store 负责保存数据和定义修改数据的逻辑，同时调用 Dispatcher 的 register()方法将自己设为监听器。每当发起一个 Action（动作）去触发 Dispatcher，Store 的监听器就会被调用，用于执行是否更新数据的操作。如果更新了，那么 View 中将获得最新的状态并更新。

Store 在 Flux 中的特性是，管理应用所有的数据；只对外暴露 getter 方法，用于获取 Store 的数据，而没有暴露 setter 方法，这就意味着不能通过 Store 去修改数据。如果要修改 Store 的数据，必须通过 Action 动作去触发 Dispatcher 实现。

只要 Store 发生变更，它就会使用 emit()方法通知 View 更新并展示新的数据。

以上是关于 Flux 的流程，来回顾总结一下：当用户在 View 上发起一个交互动作时，Dispatcher 会广播一个 Action（一个包含 Action 类型和数据的对象），然后 Store 对 Action 进行响应，数据如果改变，Store 就通知 View 界面进行重新渲染。在一个应用中，Dispatcher 这个总的调度台内注册了各种不同的 Action 以满足用户在界面中发起各种不同的功能和需求。

4.1.4 Flux 的缺点

Flux 的缺陷主要体现在增加了项目的代码量，使用 Flux 会让项目带入大量的概念和文件；单元测试难以进行，在 Flux 中，组件依赖 Store 等其他依赖，使得编写单元测试非常复杂。

虽然 Flux 的缺陷较明显，但换来的是相对于 MVC 更清晰的数据流和可预测的状态。

假如应用中没有涉及组件之间的数据共享，或者全是静态界面，那就无须使用 Flux。Flux 适合复杂项目，且组件数据相互共享的应用。

4.1.5 Flux 架构小结

Flux 是一种架构模式，它的出现可以说是 MVC 的“替代方案”，它是用于复杂应用中数据流管理的一种方案。

Flux 的核心思想是“单向数据流”，加之其“中心化控制”特点，得以让数据改变源头变得可控，Flux 架构追踪数据改变的复杂程度相对于 MVC 简单许多；Flux 让 View 保持高度整洁，无须关注太多逻辑，只需关注传入的数据；Flux 的 Action 提高了系统抽象的程度，对于用户来说，它仅仅就是一个动作。

虽然 Flux 的写法让应用产生了不少的冗余代码，但我们应该更加关注它的设计思想，毕竟它还“年轻”。近几年来，由 Flux 衍生的诸多“变种”已被社区开发者大量使用，其中最有名的就是 Redux，而且其知名度远远超越了 Flux，殊不知它的设计思想起源于 Flux。

4.2 Redux 状态管理工具

上一节了解了 Flux 单向数据流带来的好处，并且 Facebook 也给出了自己的实现方式。由于 Flux 还存在一些问题和不足，并未得到广泛推广和使用。但可喜的是，Flux 架构被广大开发者拓展并发扬光大，衍生出了 Refulx、Fluxxor 和 Redux 等优秀的 Flux 架构方案。其中，最受欢迎的当属 Redux，在 GitHub 上开源仅仅数月就揽获了上万 star。

本节将结合商城购物车实例来讲解 Redux 的运用，地址为 <https://github.com/khno/react-redux-example>。

4.2.1 Redux 简介

Redux 作者是 Dan Abramov。Dan Abramov 在 React Europe 2015 上进行了一次令人印象深刻的演示（<https://www.youtube.com/watch?v=xSsNQyntHs>），之后 Redux 迅速成为最受人关注的 Flux 实现之一。Redux 由 Facebook 的 Flux 演变而来，并受到了函数式编程语言 Elm 的启发。截至 2018 年 7 月 15 日，Redux 在 GitHub 上的加星数量为 42538 个，并有超过 601 个贡献者、10426 次 Fork。

Redux 是一个“可预测的状态容器”，而实质也是 Flux 里面“单向数据流”的思想，但它充分利用函数式的特性，让整个实现更加优雅纯粹，使用起来也更简单。Redux 是超越 Flux 的一次进化。

Redux 通过极少的接口实现了强大的功能，这与 React 有异曲同工之妙，它们都是主

张通过最少的接口实现其核心的功能。但有趣的是，Dan Abramov 并没有预料到 Redux 会变得如此受欢迎，当初他的本意只是为了利用 Flux 解决热加载和时间旅行问题。Dan Abramov 后来也加入了 Facebook，并在推特上发布了就职于 Facebook 的消息（图 4.5），这很有趣。



图 4.5 Dan Abramov 加入了 Facebook

4.2.2 Redux 的使用场景

在学习使用 Redux 之前，先来了解下它在 React 中所处的“位置”：React 是利用组件化思想进行的开发，将应用中各个模块细分为不同的组件，而组件之间有时候需要数据传递共享。一个组件的交互可能会影响另一个组件的状态展示，而 Redux 就是用来对这些组件进行状态管理的。Redux 可以比喻为买家和卖家之间的快递员，是一个“第三方”或“中介”。

需要注意的是，Redux 的存在仅仅是为了管理状态。

Dan Abramov 说过“只有遇到 React 实在解决不了的问题，你才需要 Redux。”如果应用满足以下场景：

- UI 层非常简单；
- 用户使用方式非常简单；
- 页面之间没有协作；
- 与服务器没有大量交互。

那么，应用没有必要使用 Redux，否则反而会增加项目的复杂程度，得不偿失。

当应用满足以下场景：

- 用户的使用方式复杂；
- 不同身份的用户有不同的使用方式（比如普通用户和管理员）；
- 多个用户之间可以协作；
- 与服务器大量交互，或者使用了 WebSocket；
- View 需要从多个来源获取数据。

那么，应用都是适合使用 Redux 的。上面场景可以总结为：这类应用是多交互、多数据源的！

4.2.3 Redux的动机

近几年前端开发应用变得越来越复杂，各种框架组件层出不穷，随之而来的就是这些应用的状态变得难以维护。这些状态可能来自服务端返回的数据、本地缓存的数据、本地生成没有持久化到服务器的数据、数据加载时候的加载状态等。

对于这些动态状态的管理令开发者苦恼不堪，改变了一个状态，可能会引起其他无法预知的副作用。有时候状态在何时、何地、何因而改变，无从知晓，造成状态变化而无法预测与追踪。

Redux 就是为了解决这些问题而出现的，它可预测和追踪状态的改变。那么它是如何做到呢？下一节来看看它的特性。

4.2.4 Redux 三大特性

1. 单一数据源

整个应用的 state 都存储在一个 JavaScript 对象——Store 中，可以将 Store 理解为全局的一个变量，且全局只有一个 Store。单一的状态树可以让调试变得简单，开发过程中可以将 state 保存在内存中，从而加快开发速度。

并且这种做法让同构应用开发也变得非常容易，来自服务器端的 state 可以在无须编写很多代码的情况下，被注入到客户端。

2. state是只读的

改变 state 的唯一方法就是触发 Action，Action 是一个普通的 JavaScript 对象，用于描述发生的事件。这样可以确保视图层和网络请求都无法直接修改状态，相反，它们仅仅是表达想要修改的意图。Store 中有一个 dispatch，dispatch 接收 Action 参数，然后通过 Store.dispatch(Action) 来改变 Store 中的 state。

```
store.dispatch({
  type: types.RECEIVE_PRODUCTS,           // Action 名称
  products      // products 表示该 Action 携带的状态，最后将存储在 Store 中
})
```

3. 使用纯函数执行修改

先来解释下什么是纯函数：一个函数的返回结果只依赖于它的参数，相同的输入，永远会得到相同的输出，而且没有任何可观察的副作用。

纯函数拥有以下特性：

- 可缓存性（Cacheable），总能根据输入来做缓存。

- 可移植性 / 自文档化（Portable / Self-Documenting），完全自给自足，纯函数需要的所有内容都能轻易获得。这种自给自足的好处是纯函数的依赖很明确，因此更容易观察和理解。
- 可测试性（Testable），纯函数在测试时只需要简单地输入一个输入值，然后断言输出即可。
- 引用透明性（Referential Transparency），如果一个表达式在程序中可以被它等价的值替换而不影响结果，那么就说这段代码是引用透明的。纯函数总能根据相同的输入返回相同的输出，所以能保证总是返回同一个结果。
- 并行代码，可以并行运行任意纯函数。因为纯函数根本不需要访问共享的内存，而且根据其定义，纯函数也不会因副作用而进入竞争态（Race Condition）。

通过以上阐述，相信读者已经了解了什么是纯函数，也得知了纯函数带来的好处。下面来看看 Redux 中如何利用纯函数修改状态。

这里所说的纯函数是指 Reducer，其作用是为了描述 Action 如何改变状态树。Reducer 接收之前的 state 和 Action，并返回新的 state。

Reducer 应用示例：

```
// 下面是一个 Reducer，它负责处理 Action，返回新的 state
const addedIds = (state = [], action) => {
  switch (action.type) {
    case ADD_TO_CART:
      return [...state, action.productId]
    default:
      return state
  }
}
```

纯函数的使用可以让 Reducer 内对状态的修改变得更纯粹及可测试，并且 Redux 利用每次返回的新状态生成时间旅行（Time Travel）调试方式，让每次修改变得可以被追踪。

通过以上内容介绍，读者应该已经对 Redux 有个大概的理解了。无论如何，请读者先记住以上特性，接下来看 Redux 的组成。

4.2.5 Redux 的组成——拆解商城购物车实例

Redux 由 3 部分组成：Action、Reducer 及 Store。

Action 用来表达动作，Reducer 根据 Action 更新 State。Redux 的设计思想是将应用看作是一个状态机，视图与状态是一一对应的；所有状态都存放在 Store 这个对象内。用户通过触发 View 层的 Action 去改变 Store 中的状态，而 View 层的状态来源于 Store，一个 state 对应一个 View，也就是“状态驱动视图变化”。

下面通过一个商城购物车实例来贯穿讲解 Redux 的组成。

1. Action简介

在 Redux 中，Action 的概念、使用方法和创建方法与 Flux 基本一致，是一个用于描述发生了什么事情 JavaScript 对象，它也是信息的载体。Action 是一个能够把 state 从应用传到 Store 的载体。

非常重要的一点，Action 是 Store 中数据的唯一来源。比如要获取商品列表的 Action，示例如下：

```
import * as types from '../actionTypes/ActionTypes'; // Action 类型定义
{
  type: types.RECEIVE_PRODUCTS, // Action 名称
  products // products 表示该 Action 携带的状态，最后将存储在 Store
}
```

上述代码中 Action 的名称是 ADD_ITEM，它携带的信息是 item。其中 type 是必须的属性，表示当前这个 Action 的名称。

由于 type 是一个字符串常量，表示要执行的动作名称，因此在大型项目中建议使用模块或文件夹单独存放 actionTypes。

actionType 源码：

```
export const ADD_TO_CART = 'ADD_TO_CART'
export const CHECKOUT_REQUEST = 'CHECKOUT_REQUEST'
export const CHECKOUT_SUCCESS = 'CHECKOUT_SUCCESS'
export const CHECKOUT_FAILURE = 'CHECKOUT_FAILURE'
export const RECEIVE_PRODUCTS = 'RECEIVE_PRODUCTS'
```

然后在 Action 中引用：

```
import * as types from '../actionTypes/';
```

由于应用中一般不会只有一个 Action，如果一个个手写会很麻烦。因此可以创建一个函数专门用于生成 Action，这个函数就是 Action Creator。

生成 Action 的示例：

```
/**
 * Action 创建函数
 * 创建一个被绑定的 Action 创建函数来自动 dispatch
 */
export const getAllProducts = () => dispatch => {
  shop.getProducts(products => {
    dispatch(getProducts(products))
  })
}
```

学习了前面的基础知识后，读者可以根据下面的代码注释来分析购物车 Action 的源码：

```
import shop from '../api/shop' // 模拟获取服务端数据
import * as types from '../actionTypes/' // actionTypes 定义
/**
```

```

    * Action 创建函数
    * 获取商品（书籍）
  */
const getProducts = products => ({
  type: types.RECEIVE_PRODUCTS,
  products
})
/***
    * Action 创建函数
    * 创建一个被绑定的 Action 创建函数来自动 dispatch
  */
export const getAllProducts = () => dispatch => {
  shop.getProducts(products => {
    dispatch(getProducts(products))
  })
}
/***
    * Action 创建函数
    * 加入购物车
  */
const addToCartUnsafe = productId => ({
  type: types.ADD_TO_CART,
  productId
})
/***
    * 加入购物车
  */
export const addToCart = productId => (dispatch, getState) => {
  if (getState().products.byId[productId].inventory > 0) {
    dispatch(addToCartUnsafe(productId))
  }
}
/***
    * 去结算
  */
export const checkout = products => (dispatch, getState) => {
  const { cart } = getState()
  dispatch({
    type: types.CHECKOUT_REQUEST
  })
  shop.buyProducts(products, () => {
    dispatch({
      type: types.CHECKOUT_SUCCESS,
      cart
    })
  })
}

```

2. Reducer简介

Action 定义了要执行的操作，但没有规定如何更改 state。而 Reducer 的职责就是定义整个应用的状态如何更改。Reducer 主要作用是根据 Action 执行去更新 Store 中的状态。

在 Redux 中，所有状态都被保存在一个单一的 JavaScript 对象中。这与 Flux 是不一样的，Flux 可以有多个 Store 来存储各种不同的状态。Reducer 没有什么特别的，可以理解为是一个纯函数，它接受之前的 state 和 Action 对象，并返回新的 state。如之前所说的，纯函数在这里的使用不会有副作用，只要传入的参数确定，返回的结果总是唯一的。以将商品放入购物车举例，Reducer 写法如下：

```
src/reducer/products.js
import { ADD_TO_CART } from '../actionTypes'
const products = (state, action) => {
  switch (action.type) {
    case ADD_TO_CART:
      return {
        ...state,
        inventory: state.inventory - 1
      }
    default:
      return state
  }
}
```

随着应用变得越来越复杂，可以将 Reducer 函数拆分成多个单独的函数，拆分后的每个函数负责独立管理 state 的一部分。这样能更好地管理代码，让代码结构更加清晰，最后再将这一个个小小的 Reducer 合并到一起。这个时候会用到 combineReducers()方法，这个方法由 Redux 提供。

combineReducers(Object)只接收一个对象参数，使用如下：

```
import { combineReducers } from 'redux';
rootReducer = combineReducers({
  cart: cartReducer,
  products: productsReducer
});
// rootReducer 将会返回如下 state 对象
{
  cart: {
    // cart 的 state 对象
  },
  products: {
    // products 的 state 对象
  }
}
```

当然，也可以通过更改 combineReducers()被传入的 Reducer 的 key，来修改返回的 statekey 的命名。例如：

```
import { combineReducers } from 'redux';
rootReducer = combineReducers({
  yourCart: cartReducer,
  yourProducts: productsReducer
});
```

也可以直接命名 Reducer，然后使用 ES 6 的简写：

```
import { combineReducers } from 'redux';
rootReducer = combineReducers({
  cartReducer,
  productsReducer
});
```

上述写法与下面写法是等价的：

```
rootReducer = combineReducers({
  cart: cartReducer,
  products: productsReducer
});
```

3. Store简介

Redux 中的 Store 概念和 Flux 中的 Store 基本相同，但 Redux 中全局只有一个 Store，用于存储整个应用的状态。它有以下 4 个 API：

- getState()方法用于获取 state；
- dispatch(action)方法用于执行一个 Action；
- subscribe(listener)用于注册回调，监听 state 变化；
- replaceReducer(nextReducer)更新当前 Store 内的 Reducer（一般只会在开发模式中使用）。

实际开发中最常用到的是 getState()和 dispatch()这两个方法。

Store 是通过 Redux 提供的 createStore()方法来创建的，这也是 Redux 最核心的方法。例如：

```
import { createStore } from 'redux'
import reducers from './reducers'
let store = createStore(reducers)           // Store 的创建使用 Reducer 作为参数
```

或者：

```
import { createStore, applyMiddleware } from 'redux'
import reducers from './reducers'
const store = createStore(
  reducer,
  applyMiddleware(...middleware)
)
```

上述代码中，createStore()的第 2 个参数是可选的，用于初始值的设置。这对于同构引用开发来说非常有用，当服务端 Redux 应用的状态结构与客户端一致时，客户端可以直接在这里传入服务数据用于初始化 state。

```
let store = createStore(reducer, window.STATE_FROM_SERVER)
```

createStore(reducer, [preloadedState], enhancer) 方法有 3 个参数，下面分别介绍。

(1) reducer(Function)

接收两个参数，分别是当前 state 和要执行的 Action，并返回新 state 树。

(2) [preloadedState](any)

可选参数。初始 state 状态，在同构应用中可以使用这个参数去初始化 state，或者从之前保存的用户会话中恢复并传给它。如果使用 `combineReducers()` 创建 Reducer，它必须是一个普通对象，与传入的 keys 保持同样的结构；否则，可以任意传入任何 Reducer 能理解并识别的内容。

(3) enhancer(Function)

可选参数。enhancer 就是指 store enhancer，顾名思义是增强 Store 的功能。它是一个高阶函数，其参数是创建 Store 的函数。与 middleware 相似（4.3 节介绍），允许通过复合函数改变 Store 接口。

`createStore()` 函数返回 Store，里面保存了所有 state 对象。例如：

```
import { createStore } from 'redux'
// Reducer 定义
const addToCart = (state = initialState.quantityById, action) => {
  switch (action.type) {
    case ADD_TO_CART:
      const { productId } = action
      return { ...state,
        [productId]: (state[productId] || 0) + 1
      }
    default:
      return state
  }
}
// store 创建
// 第1个参数：传入 Reducer；第2个参数：初始化 state
let store = createStore(addToCart, ['Hello Redux'])
```

需要注意的是，应用中不要创建多个 Store，可以通过 `combineReducers()` 把多个 Reducer 创建成一个根 Reducer，并且记住永远不要修改 state。比如 Reducer 内不要使用 `Object.assign(state, newData)`，应该使用 `Object.assign({}, state, newData)`。或者使用对象拓展操作符（object spread operator，参考 <https://github.com/tc39/proposal-object-rest-spread>）特性中的 `return{ ...state, ...newData }`。

当 Store 创建完之后，Redux 会 dispatch 一个 Action 到 Reducer，用于初始化 Store。但无须处理这个 Action。如果第 1 个参数（也就是传入的 state）是 `undefined`，则 Reducer 应该返回初始的 state。

4.2.6 Redux 搭配 React 使用

Redux 是不依赖于 React 而存在的，它本身能支持 React、Angular、Ember 和 jQuery 等。要让其在 React 上运行，就得让二者绑定起来去建立连接。于是就有了 `react-redux`，它能将 Redux 绑定到 React 上。

△注意：react-redux 是基于容器组件和展示组件相互分离的开发思想（更多详情，可参见笔者博客 https://medium.com/@dan_abramov/smart-and-dumb-components-7ca2f9a7c7d0）。

安装 react-redux：

```
npm install -save react-redux
```

react-redux 提供了两个重要对象：Provider 和 connect。

1. Provider简介

使用 react-redux 时需要先在最顶层创建一个 Provider 组件，用于将所有的 React 组件包裹起来，从而使 React 的所有组件都成为 Provider 的子组件。然后将创建好的 Store 作为 Provider 的属性传递给 Provider。

Provider 应用示例：

```
import React from "react";
import { render } from 'react-dom';
import { createStore } from 'redux';
import { Provider } from "react-redux";
// 导入 reducer
import reducer from './reducers';
// store 创建
const store = createStore(
  reducer
)

render(
  <Provider store={store}>          // Provider 需要包裹在整个应用组件的外部
    <App />                      // React 组件
  </Provider>,
  document.getElementById('root')
)
```

2. connect简介

connect 的主要作用是连接 React 组件与 Redux Store。当前组件可以通过 props 获取应用中的 state 和 Actions。

connect 接收 4 个参数：mapStateToProps()、mapDispatchToProps()、mergeProps()和 options。项目中经常使用的是前面 2 个参数，相关介绍如下：

(1) [mapStateToProps(state, ownProps): stateProps]: 该函数允许开发者将 Redux 的 Store 中的数据作为 props 绑定到组件上，返回对象的所有 key 都为当前组件的 props。此时该组件会监听 Store 的变化。一旦 Store 发生变化，mapStateToProps()函数就会被调用，并返回一个纯对象，这个对象将与当前组件的 props 合并。该函数的第 2 个参数 ownProps 表示当前组件自身的 props。只要组件接收到新的 props，mapStateToProps()就会被调用，计算出一个新的 stateProps 提供给组件使用。示例如下：

```

const mapStateToProps = (state) => {           // store 的第 1 个参数就是 Redux 的
  return {                                     Store
    list: state.list // 从 Redux 的 Store 中获取 list (只获取当前组件需要的 state)
  }
}

```

(2) [mapDispatchToProps(dispatch, [ownProps]): dispatchProps]: 它的功能是将 Action 作为 props 绑定到当前组件。返回当前组件的 actioncreator，并与 dispatch 绑定。如果指定了第 2 个参数 ownProps，该参数的值将会传递到该组件的 props 中，之后一旦组件收到新的 props，mapDispatchToProps() 也会被调用。

(3) [mergeProps(stateProps, dispatchProps, ownProps): props]: 如果指定了这个函数，可以将 mapStateToProps() 和 mapDispatchToProps() 返回值和当前组件的 props 作为参数，返回一个完整的 props。不管是 stateProps 还是 dispatchProps，都需要和 ownProps 合并之后才会被赋给组件使用。如果不传递这个参数，connect 将使用 Object.assign 代替。

(4) [option]: 可选的额外配置项，一般不太使用。如果指定这个参数，可以去制定 connector 的行为。

在演示项目中，目前只关注 mapStateToProps() 和 mapDispatchToProps() 函数。

mapStateToProps() 和 mapDispatchToProps() 示例应用：

```

import React from 'react';
import PropTypes from 'prop-types';
import { connect } from 'react-redux';
import * as cartActions from '../actions';
import ItemComponent from './ItemComponent';
import CartComponent from './CartComponent';
const App = React.createClass({
  render() {
    const lists = this.props;
    const cartActions = this.props.cartActions;
    return(
      <div>
        <ItemComponent {...lists} />
        <CartComponent {...cartActions} />
      </div>
    )
  }
})
// connect 包裹 App 组件
export default connect(state => ({
  lists: state.list
}), dispatch => ({
  cartActions: bindActionCreators(cartActions, dispatch)
}))
)(App)

```

上述例子中，connect 和 Provider 组件相互配合将 Actions 和 lists 以 props 的形式传入组件 App 中。在 mapStateToProps() 中，选取整棵 Store 树中的 list 分支作为当前组件的 props，并命名为 lists。然后在组件中使用 this.props.lists。在 mapDispatchToProps() 中，使用 Redux

提供的工具方法将 `cartActions` 与 `dispatch` 绑定，最后在组件中使用 `this.props.cartActions`。

`react-redux` 的 `connect` 属于高阶组件，它允许向一个现有组件添加新功能，同时不改变其结构，属于装饰器模式（Decorator Pattern）。

注意：ES 7 中添加了 `decorator` 属性，使用`@`符号表示，可以更精简地书写。

`decorator` 属性应用示例：

```
import React from 'react';
import { render } from 'react-dom';
import connect from './connect';
@connect
class App extends React.Component{
  render(){
    return <div>...</div>
  }
}
```

上述代码中，组件中通过 `connect` 将 `Store` 中的数据转换为组件可用的数据，并且生产 `Action` 的派发函数。`react-redux` 利用 `connect` 将当前“木偶组件”进行包裹，并为该组件传递数据。

木偶组件（Dumb components）也叫 UI 组件，是与 `Redux` 没有直接联系的组件。该组件不知道 `Store` 或 `Action` 的存在，需要通过 `props` 传入组件，让组件得到且使用它们。所谓“木偶组件”，就是该组件能独立运作，不依赖于这个应用的 `Actions` 或 `Stores` 存在而存在。它不必存在 `state`，也不能使用 `state`，允许接收数据和数据改动，但只能通过 `props` 来处理。原则上它只负责展示，像一个“木偶”，受到“外界”`props` 的控制。

木偶组件中的事件可以使用 `Actions` 方法的调用，`Actions` 中的函数是经过 `bindActionCreators` 处理过的，会直接派发，从而改变 `Store` 数据，触发视图重渲染。重渲染的过程其实就是由于 `props` 传入新的数据通过比较后对 DOM 的更新。

如果不使用 `react-redux` 的 `Provider` 和 `connect` 这种机制，也可以直接使用 `Redux`。但本质是一样的，还是在 `React` 组件最外层组件将其包裹，作为 `props` 属性向内层递进传递，但是不提倡这种做法的。

直接使用 `Redux` 示例：

```
class App extends Component {
  componentWillMount() {
    store.subscribe(state => this.setState(state))
  }
  render() {
    return <BooksContainer state={this.state}
      onClick={() => store.dispatch(actions.addToCart())}>
    />
  }
}
```

与 `Flux` 对比后就可以知道，使用 `react-redux` 的结构分离点更清晰，因此在 `React` 中还是推荐使用 `react-redux`。

4.3 middleware 中间件

之前 Action 的发起是同步的，如果现在需要发起异步的 Action，那么此时就是中间件 middleware 发挥作用的时候。本节先从问题分析说起，再介绍自己动手构建中间件来解决问题，通过这个过程让读者了解什么是中间件。

4.3.1 为何需要 middleware

Redux middleware provides a third-party extension point between dispatching an action, and the moment it reaches the reducer.

以上是官方对于 Redux 中间件的描述。Redux 中间件在发起一个 Action，Action 到达 Reducer 之间，提供了一个第三方拓展。也就是说，middleware 是架在 Action 和 Store 之间的一座桥梁。

如果读者之前使用过 Express 或 Koa 等服务端框架(7.2 节服务端渲染中将会有介绍)，那么对 middleware 应该不会陌生。在这类框架中，middleware 是嵌入在框架接收请求 req 和响应 res 之间的中间层，在这里能做很多事情，比如可以在 middleware 中完成日志记录、添加 CORS headers、内容压缩等，Redux 中 middleware 的作用也类似。

从前面知识点可以得知，Redux 是用来控制和管理所有数据的输入和输出，而 dispatch() 作为一个纯函数只是单纯地用于派发 Action 来修改数据，所以当碰到需要记录每次 dispatch() 等问题时就变得很不容易。而中间件可以对每一个流过的 Action 进行检阅，并进行相应的操作。也可以在原有 Action 的基础上创建一个新的 Action 和 dispatch() 并触发一些额外的行为，例如日志记录、创建崩溃报告、调用异步接口或路由等。而中间件的价值由此得以体现。

4.3.2 深入理解 middleware

middleware 本质上就是通过插件的形式，将原本 Action → Reducer 的流程改为 Action → middleware1 → middleware2 → middleware3... → Reducer。这也正是 Redux 中间件 middleware 最优秀的特性，可以被链式组合和自由拔插。下面以记录日志和创建崩溃报告为例，来引导读者体会从分析问题到构建 middleware 解决问题的思维过程。

1. 手动记录

Redux 的一个好处就是能让 state 的变化过程变得可预知和可追踪。每个 Action 发起后都会被计算并保存下来。假如有这样一个需求，需要记录应用中每一个 Action 发起的信

息和 state 的状态。这样，当程序出现问题时，就能通过查阅日志找出哪个 Action 导致了 state 不正确。

最简单的做法就是，在每次调用 `store.dispatch(action)` 前后手动记录。

记录日志示例：

```
let action = actionTodo('Use Redux');
console.log('dispatching', action);
store.dispatch(action);
console.log('next state', store.getState());
```

这样就能满足记录应用中每一个 Action 发起的信息和 State 的状态这个需求，但是这样就意味着每当 `dispatch` 一个 Action 的时候，就需要重复性书写以上代码。这样会造成重复性代码，并不是开发者想要的。

2. 封装`dispatch()`

由于手动记录代码是重复性的，因此可以将以上的代码封装成一个函数。

记录日志的函数示例：

```
function dispatchAndLog(store, action) {
  console.log('dispatching', action)
  store.dispatch(action)
  console.log('next state', store.getState())
}
```

这样每当需要发起一个 Action 的时候就会导入这个方法，写法上有所简化，但实际代码量还是没有改变。

3. 替换`dispatch()`

由于 Redux 的 Store 是一个包含一些方法的普通对象，因此可以直接替换 Redux 的 Store 方法：

```
const next = store.dispatch; // 获取 Redux 中的 dispatch
store.dispatch = function dispatchAndLog(action) { // 重新覆盖原有的方法
  console.log('dispatching', action);
  let result = next(action);
  console.log('next state', store.getState());
  return result;
}
```

这样一来，就非常接近满足开发者去记录每次发起 Action 时想要获取 action 信息和 state 这个需求的理想方式了。

4. 添加多个middleware

现在来思考如何添加多个中间件。中间件的功能各不相同，发起 Action 的时候要按顺序依次执行，所以添加多个中间件并不是一件简单的事情。比如再来增加一个报错记录的需求。

首先，需要将日志记录和报错记录这两个函数区分开，独立封装。

日志记录和报错记录示例：

```
function patchStoreToAddLogging(store) {
  const next = store.dispatch
  store.dispatch = function dispatchAndLog(action) {
    console.log('dispatching', action)
    let result = next(action)
    console.log('next state', store.getState())
    return result
  }
}

function patchStoreToAddCrashReporting(store) {
  const next = store.dispatch
  store.dispatch = function dispatchAndReportErrors(action) {
    try {
      return next(action)
    } catch (err) {
      console.error('捕获一个异常!', err)
      Raven.captureException(err, {
        extra: {
          action,
          state: store.getState()
        }
      })
      throw err
    }
  }
}
```

然后依次调用上面两个函数就可以覆盖原来的 `store.dispatch()` 函数，最后得到增强的 `store.dispatch()`。

```
patchStoreToAddLogging(store);
patchStoreToAddCrashReporting(store);
store.dispatch(addTodo('Use Redux'));
```

此时再去调用 `store.dispatch()` 方法，就具备日志记录和报错记录的功能了。但这并非最佳方案。

之前采用直接覆盖 API 的方式来实现日志记录和报错记录，这种写法可以拓展和增强这个 API 本身的功能。每次给 `store.dispatch()` 赋值的时候就意味着已经获取了前一个对 middleware 修改的方法。这就是一种链式调用，新增的功能将会使 `store.dispatch()` 变得越来越强大。

除此之外，还有一种方法也可以实现前面的需求，那就是把 `store.dispatch` 的引用当作参数传递到另一个函数中，而不是直接改变它的值。示例如下：

```
function logger(store) {
  const next = store.dispatch
  // 之前的做法
  // store.dispatch = function dispatchAndLog(action) {
  return function dispatchAndLog(action) {
    console.log('dispatching', action)
```

```

        let result = next(action)
        console.log('next state', store.getState())
        return result
    }
}

```

上面代码中的 `next()` 就是 `dispatch()`，但这个 `dispatch()` 函数每次执行时会保留上一个 `middleware` 传递的 `dispatch()` 函数的引用。接下来写一个方法将多个 `middleware` 连起来，该方法的主要作用是将上一次返回的函数赋值给 `store.dispatch`：

```

function applyMiddlewareByMonkeypatching(store, middlewares) {
    middlewares = middlewares.slice()
    middlewares.reverse()
    // 在每一个 middleware 中变换 dispatch 方法
    middlewares.forEach(middleware =>
        // 每次 middleware 会直接返回函数，然后赋值给 store.dispatch
        store.dispatch = middleware(store)
    )
}

```

然后可以在其他地方像下面这样使用多个 `middleware`：

```
applyMiddlewareByMonkeypatching(store, [logger, crashReporter])
```

在 `applyMiddle` 内需要给 `store.dispatch` 赋值，否则在下一个中间件中就拿不到最新的 `dispatch`。

5. 柯里化 middleware

其实，还有另一种方式也可以实现之前的链式调用，示例如下：

```

function logger(store) {
    return function wrapDispatchToAddLogging(next) {
        return function dispatchAndLog(action) {
            console.log('dispatching', action)
            let result = next(action)
            console.log('next state', store.getState())
            return result
        }
    }
}

```

看到这里，相信读者已经明白了。没错，这种做法就是让中间件以方法参数的形式接收一个 `next()` 方法，而不是通过 `Store` 的实例去获取。上面的写法看起来比较奇怪，下面再换一种写法，用 ES 6 的箭头函数来表达函数柯里化（currying）：

```

logger:
const logger = store => next => action => {
    console.log('dispatching', action)
    let result = next(action)
    console.log('next state', store.getState())
    return result
}

```

从上面代码可以得知，`middleware` 接收一个 `next()` 的 `dispatch` 函数，并返回一个

dispatch，返回的函数又会被当做下一个中间件的 next()，依次传递。又由于考虑到 Store 中的 getState()方法可能会被使用，因此将其作为顶层参数向内传递，从而使得在所有 middleware 中可以被使用到。

接下来，再用箭头函数完善之前的错误报告。

crashReporter:

```
const crashReporter = store => next => action => {
  try {
    return next(action)
  } catch (err) {
    console.error('Caught an exception!', err)
    Raven.captureException(err, {
      extra: {
        action,
        state: store.getState()
      }
    })
    throw err
  }
}
```

6. 最终版本

将上面介绍的两个中间件 logger 和 crashReporter 引用到 Redux Store：

```
import { createStore, combineReducers, applyMiddleware } from 'redux'
let todoApp = combineReducers(reducers)
let store = createStore(
  todoApp,
  // applyMiddleware() 告诉 createStore() 如何处理中间件
  applyMiddleware(logger, crashReporter)
)
```

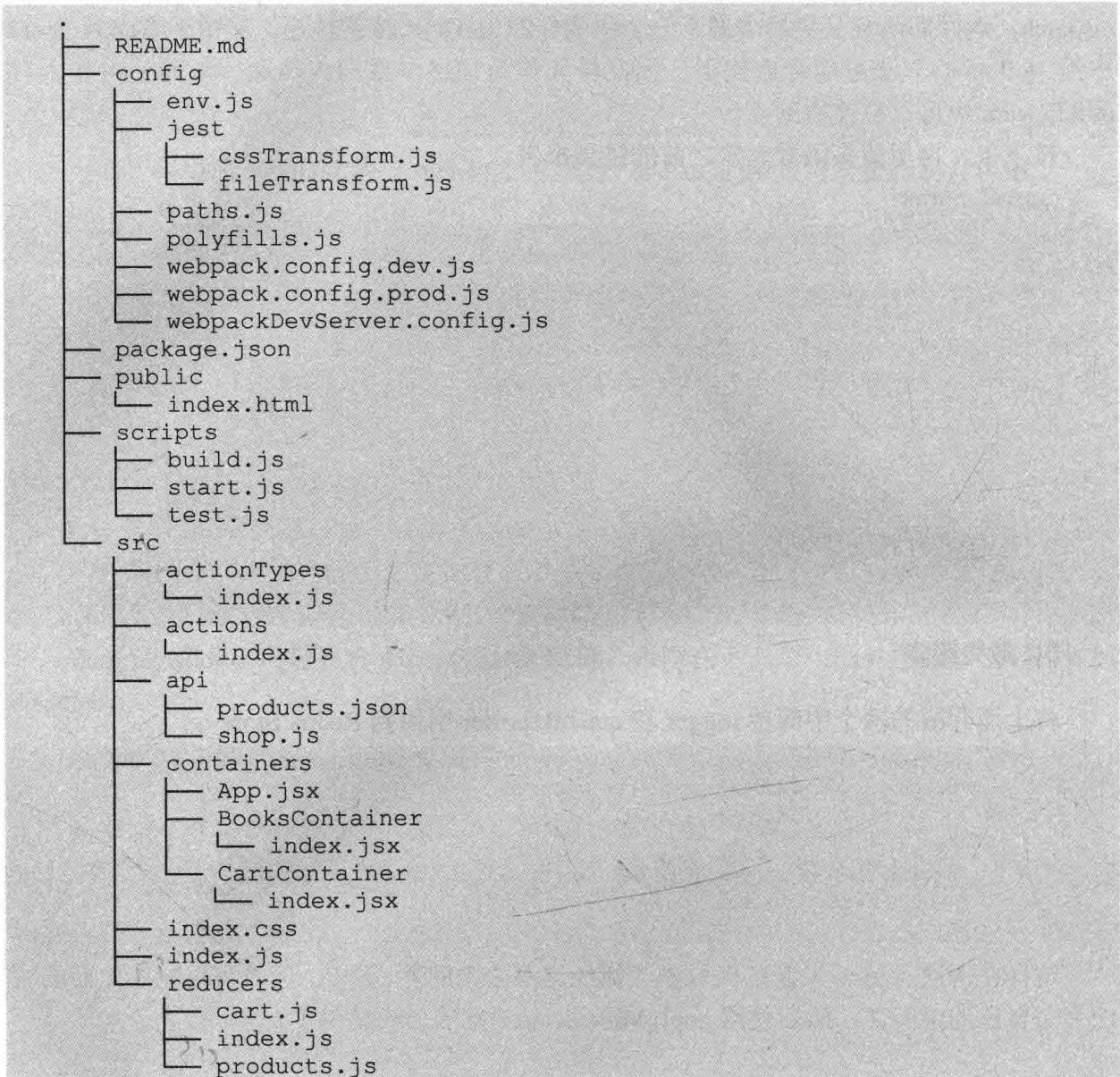
这样在每次 Action 发起的时候都会流经这两个中间件。现在，读者可以参照上面的写法书写自己的中间件，然后使用 applyMiddleware()方法加载使用了。

4.4 Redux 实战训练——网上书店

本节通过项目实战来加深和巩固之前学习的内容。项目脚手架采用 create-react-app 来搭建，在此省略项目脚手架的搭建过程。

4.4.1 目录结构

项目中，非 UI 部分有 Action（actions 和 actionTypes）、Store（Reducer）和模拟接口请求的 API。从目录结构中，可以得知应用的大致组成和逻辑关系，具体如下：



app/目录下的其他文件多为项目脚手架的配置文件。应用的具体业务逻辑基本位于/src 目录中。因此主要关注 src/目录中的内容。

不难发现，其中的 actions、actionTypes 和 reducers 便是 Redux 在项目中的组成部分，而 containers 则是应用中 UI 相关的部分。

4.4.2 应用入口 src/index.js

当执行 npm start 命令启动项目后，先确认当前的开发环境，然后执行 config/webpack.config.dev.js 命令。此时 Webpack 会找到 public/index.html 和 src/index.js 这两个文件，并在浏览器中打开 index.html 和加载被 Webpack 打包处理过的静态文件。当然以上这些是 create-

react-app项目脚手架中已经配置好的，无须初学者修改。重点来关注Webpack的entry入口文件./src/index.js。代码如下：

```

import React from "react";
import { render } from "react-dom";
import { createStore, applyMiddleware } from "redux";
import { composeWithDevTools } from 'redux-devtools-extension';
// 浏览器 Redux 工具
import { Provider } from "react-redux"; // Redux 顶层组件，用于包裹原有的组件
import { createLogger } from "redux-logger"; // 中间件，日志打印
import thunk from "redux-thunk"; // 中间件
import reducer from "./reducers"; // 应用的 Reducer
import { getAllProducts } from "./actions"; // 应用的 Action
import App from "./containers/App.jsx"; // 应用的 UI 组件
import "./index.css"; // 样式
const middleware = [thunk];
if (process.env.NODE_ENV !== "production") { // 如果是开发环境，就添加中间件
  middleware.push(createLogger()); // 日志打印
}
// 创建 Store，整个应用中有且仅有一个 Store，用于存放整个应用中所有的 state
const store = createStore(reducer, composeWithDevTools(
  applyMiddleware(...middleware),
));
// 请求接口获取数据
store.dispatch(getAllProducts());
render(
  <Provider store={store}> // 使原来整个应用成为 Provider 的子组件
    <App />
  </Provider>,
  document.getElementById("root") // 渲染的节点，index.html 中有一个 ID 为
                                // root 的 div
);

```

简而言之，上述代码是这个应用的入口文件，它主要做了以下工作：

- 引入应用相关的UI组件和样式；
- 在开发环境里加入中间件；
- 请求接口，获取应用初始数据；
- 利用Provider组件将应用<APP>包裹；
- 创建Store并通过Provider的Store属性传入应用。

注意：为了更好地调试，在项目中额外增加了redux-devtools-extension，用于在浏览器中使用ReduxDevTools。并且它可以在各种浏览器中安装使用，非常优雅且方便地供开发者在使用Redux过程中进行代码的调试。详情见<https://github.com/zalmoxisus/redux-devtools-extension>，这里不再赘述。

安装如下：

```
npm install --save-dev redux-devtools-extension
```

使用：

```
import { createStore, applyMiddleware } from 'redux';
import { composeWithDevTools } from 'redux-devtools-extension';
const store = createStore(reducer, composeWithDevTools(
  applyMiddleware(...middleware),
  // other store enhancers if any
));

```

然后在浏览器中可以看到如图 4.6 所示的面板。

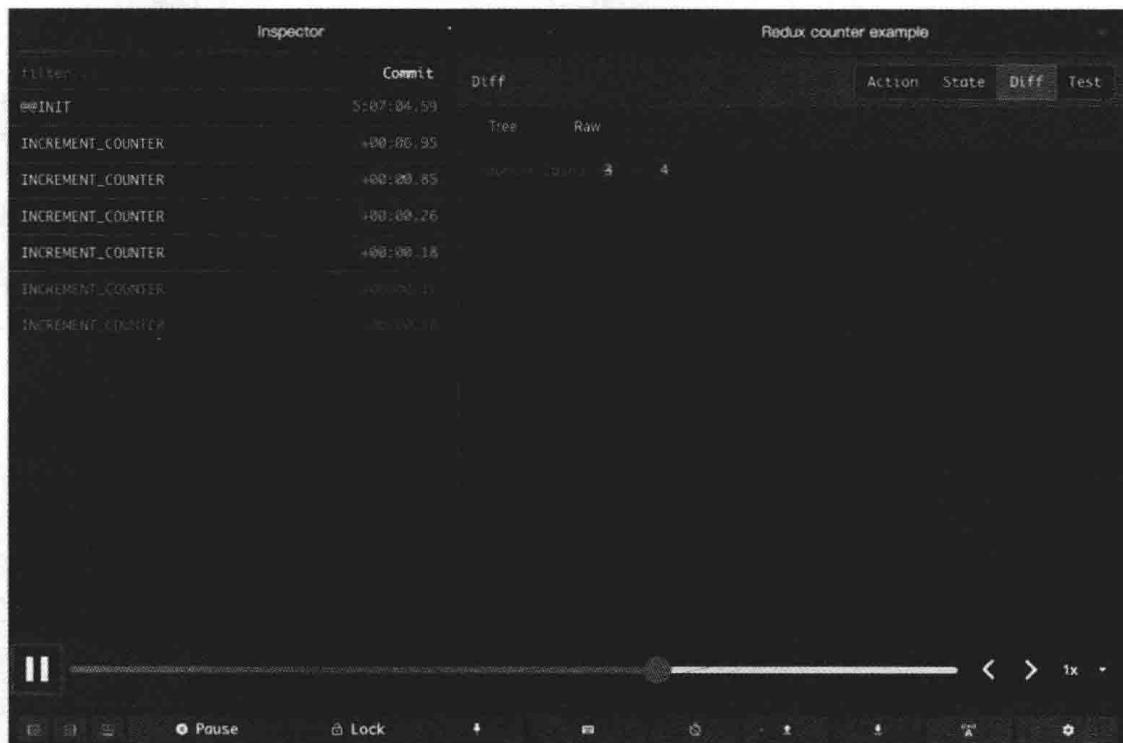


图 4.6 Redux DevTools 面板

4.4.3 Action 的创建和触发

Action 用于描述整个应用中的所有行为。在入口文件 src/index.js 中，请求了一个初始数据，笔者从这里开始讲解。

```
// src/index.js 中，请求接口获取数据
store.dispatch(getAllProducts());
```

其中，`getAllProducts()` 函数是一个 Action Creators，用来生成 Action。下面来看 `./src/actions/index.js`：

```
import shop from '../api/shop'          // 模拟获取服务端数据
import * as types from '../actionTypes' // actionTypes 定义
/**
```

```

 * Action 创建函数
 * 获取商品（书籍）
 */
const getProductsAction = products => ({
  type: types.RECEIVE_PRODUCTS,           // type 是一个常量，用来标识动作类型
  products      // products 用于这个 Action 动作携带的数据，这里写法相当于
                // products:products
})
/***
 * Action 创建函数
 * 创建一个被绑定的 Action 创建函数来自动 dispatch
 */
export const getAllProducts = () => dispatch => {
  shop.getProducts(products => {
    // 请求到内容后再发起另一个 dispatch action 将返回结果存于 store
    dispatch(getProductsAction(products))
  })
}
...

```

getAllProducts()最终返回一个 Action。在这个动作中，它请求了一个接口，并将获取的数据在此发起一个 Action，将数据放到 Store。因此在 src/index.js 中才能使用 dispatch 去触发，并将请求到的数据存于 Store 中。

./src/actions/index.js 完整代码如下：

```

import shop from '../api/shop'                      // 模拟获取服务端数据
import * as types from '../actionTypes/'           // actionTypes 定义
/**
 * Action 创建函数
 * 获取商品（书籍）
 */
const getProductsAction = products => ({
  type: types.RECEIVE_PRODUCTS,
  products
})
/***
 * Action 创建函数
 * 创建一个被绑定的 Action 创建函数来自动 dispatch
 */
export const getAllProducts = () => dispatch => {
  shop.getProducts(products => {
    // 请求到内容后再发起另一个 dispatch action 将返回结果存于 store
    dispatch(getProductsAction(products))
  })
}
/***
 * Action 创建函数
 * 加入购物车
 */
const addToCartAction = productId => ({
  type: types.ADD_TO_CART,
  productId
})

```

```

    })
    /**
     * 加入购物车
     */
    export const addToCart = productId => (dispatch, getState) => {
      if (getState().products.byId[productId].inventory > 0) {
        dispatch(addToCartAction(productId))
      }
    }
    /**
     * 去结算
     */
    export const checkout = products => (dispatch, getState) => {
      const { cart } = getState()
      dispatch({
        type: types.CHECKOUT_REQUEST
      })
      shop.buyProducts(products, () => {
        dispatch({
          type: types.CHECKOUT_SUCCESS,
          cart
        })
        // Replace the line above with line below to rollback on failure:
        // dispatch({ type: types.CHECKOUT_FAILURE, cart })
      })
    }
  }
}

```

其中，Action 的 type 通常定义为常量，放置于 actionTypes 文件的./src/actionTypes/index.js 下。

```

export const ADD_TO_CART = 'ADD_TO_CART'
export const CHECKOUT_REQUEST = 'CHECKOUT_REQUEST'
export const CHECKOUT_SUCCESS = 'CHECKOUT_SUCCESS'
export const CHECKOUT_FAILURE = 'CHECKOUT_FAILURE'
export const RECEIVE_PRODUCTS = 'RECEIVE_PRODUCTS'

```

上面的 actionTypes 都用字符串字面量定义，然后 export 出去。

简单来说，一个 Action 的创建有 3 点：定义类型（type）、定义内容（payload）和定义 Action Creator（Action 生成函数）。

4.4.4 Reducer 的创建

Reducer 用于处理 Action 触发后对 Store 的修改。在 Redux 中，不允许用户直接修改应用中的状态树，必须通过 dispatch()发起一个 Action 触发去修改 Store。这部分内容已经在上一章中讲解过，可以翻阅前面章节再次了解。

在 4.4.3 节中，定义完 Action 之后，接下来需要创建对应的 Store。Store 的创建可以理解为 Reducer 的创建，Store 的设计实际上也就是对 Reducer 的设计。在 src/index.js 中可以看到这么一行代码：

```
const store = createStore(reducer, applyMiddleware(...middleware));
```

这就是 Store 的创建，它通过 createStore()方法创建。createStore()方法将 Reducer 作为它的必要参数生成应用的 Store。在 src/reducers 文件中创建了购物车和商品列表两个 Reducer，然后在 src/reducers/index.js 中汇总。代码如下：

./src/reducers/products.js:

```

import { combineReducers } from 'redux'
import { RECEIVE_PRODUCTS, ADD_TO_CART } from '../actionTypes'
const products = (state, action) => {
  switch (action.type) {
    case ADD_TO_CART:
      return {
        ...state,
        inventory: state.inventory - 1
      }
    default:
      return state
  }
}
const byId = (state = {}, action) => {
  switch (action.type) {
    case RECEIVE_PRODUCTS:
      return {
        ...state,
        ...action.products.reduce((obj, product) => {
          obj[product.id] = product
          return obj
        }, {})
      }
    default:
      const { productId } = action
      if (productId) {
        return {
          ...state,
          [productId]: products(state[productId], action)
        }
      }
      return state
  }
}
const visibleIds = (state = [], action) => {
  switch (action.type) {
    case RECEIVE_PRODUCTS:
      return action.products.map(product => product.id)
    default:
      return state
  }
}
export default combineReducers({
  byId,
  visibleIds
})
export const getProduct = (state, id) =>
  state.byId[id]
export const getVisibleProducts = state =>

```

```
state.visibleIds.map(id => getProduct(state, id))
```

./src/reducers/cart.js:

```
import {
  ADD_TO_CART,
  CHECKOUT_REQUEST,
  CHECKOUT_FAILURE
} from '../actionTypes'
const initialState = {
  addedIds: [],
  quantityById: {}
}
const addedIds = (state = initialState.addedIds, action) => {
  switch (action.type) {
    case ADD_TO_CART:
      if (state.indexOf(action.productId) !== -1) {
        return state
      }
      return [...state, action.productId]
    default:
      return state
  }
}
const quantityById = (state = initialState.quantityById, action) => {
  switch (action.type) {
    case ADD_TO_CART:
      const { productId } = action
      return { ...state,
        [productId]: (state[productId] || 0) + 1
      }
    default:
      return state
  }
}
export const getQuantity = (state, productId) =>
  state.quantityById[productId] || 0
/**
 * 从中过滤出所有所选商品的 ID
 * 当结算时，会作为参数传递给后台
 */
export const getAddedIds = state => state.addedIds
/**
 * 购物车结算
 */
const cart = (state = initialState, action) => {
  switch (action.type) {
    case CHECKOUT_REQUEST:
      return initialState
    case CHECKOUT_FAILURE:
      return action.cart
    default:
      return {
        addedIds: addedIds(state.addedIds, action),
        quantityById: quantityById(state.quantityById, action)
      }
  }
}
```

```

    }
}

export default cart

```

./src/reducers/index.js:

```

import { combineReducers } from 'redux'
import cart, * as fromCart from './cart'
import products, * as fromProducts from './products'
/***
 * 用于 Reducer 的拆分，定义各个子 Reducer 函数，然后用这个方法，  

 * 将它们合成一个大的 Reducer。
 */
export default combineReducers({
  cart,
  products
})
const getAddedIds = state => fromCart.getAddedIds(state.cart)
const getQuantity = (state, id) => fromCart.getQuantity(state.cart, id)
const getProduct = (state, id) => fromProducts.getProduct(state.products, id)
/***
 * 获取购物车内商品的价格
 */
export const getTotal = state =>
  getAddedIds(state)
    .reduce((total, id) =>           // 价格累加
      total + getProduct(state, id).price * getQuantity(state, id),
    0
  )
    .toFixed(2)                      // 保留 2 位小数
/***
 * 向后台请求获取到的商品列表
 */
export const getCartProducts = state =>
  getAddedIds(state).map(id => ({
    ...getProduct(state, id),
    quantity: getQuantity(state, id)
  }))

```

上述代码中，用 Redux 的 `combineReducers()` 方法将两个 Reducer 合并，然后 export 到 `src/index.js` 中使用。

./src/index.js:

```

import reducer from "./reducers"
...
const store = createStore(reducer)

```

其中，`getTotal()`方法用于计算在购物车中商品的价格，也就是说，这里购物车内商品的总价格也被保存在 Store 中，最后将在应用的 UI 界面上被展示。

./src/containers/CartContainer/index.jsx:

```

...
// 页面展示

```

```

<p>总价是: {total}元</p>
...
// mapstatetoprops 用于建立一个从 store 的状态对象到当前组件 props 对象的映射关系
const mapStateToProps = (state) => ({
  total: getTotal(state)
})
...

```

getCartProducts()方法也是类似，这里不再赘述。

4.4.5 UI 展示组件的创建

在 src/index.js 中引入<App/>，并挂载到了 HTML 文件 ID 为 root 的 div 上。

public/index.html

```

<!doctype html>
<html lang="en">
  <head>
    <meta charset="utf-8">
    <meta name="viewport" content="width=device-width, initial-scale=1">
    <title>Redux</title>
  </head>
  <body>
    <div id="root"></div>
    <!--
      This HTML file is a template.
      If you open it directly in the browser, you will see an empty page.
      You can add webfonts, meta tags, or analytics to this file.
      The build step will place the bundled scripts into the <body> tag.
      To begin the development, run 'npm start' in this folder.
      To create a production bundle, use 'npm run build'.
    -->
  </body>
</html>

```

src/containers/App.jsx 就是页面中 UI 的展示文件。代码如下：

```

import React from 'react'
import BooksContainer from './BooksContainer/index.jsx'
import CartContainer from './CartContainer/index.jsx'
const App = () => (
  <div>
    <BooksContainer />
    <CartContainer />
  </div>
)
export default App

```

从以上代码可以看出，整个项目中的 UI 分为两部分：商品货架 BooksContainer 和购物车 CartContainer。下面分别给出具体代码。

src/containers/BooksContainer/index.jsx 商品货架：

```

import React from 'react'
import PropTypes from 'prop-types'
import { connect } from 'react-redux'
import { addToCart } from '../actions'
import { getVisibleProducts } from '../reducers/products'
const BooksContainer = ({ products, addToCart }) => (
  <div>
    <h2>网上书店</h2>
    <div className="products-container">
      {products.map(product =>
        <div key={product.id}>
          {product.title} - {product.price} 元 库存数量: {product.inventory ? `${product.inventory}` : null}
          <button
            className="add-btn"
            onClick={() => addToCart(product.id)}
            disabled={product.inventory > 0 ? '' : 'disabled'}>
            {product.inventory > 0 ? '添加到购物车' : '售罄'}
          </button>
        </div>
      )}
    </div>
  </div>
)
BooksContainer.propTypes = {
  products: PropTypes.arrayOf(PropTypes.shape({
    id: PropTypes.number.isRequired,
    title: PropTypes.string.isRequired,
    price: PropTypes.number.isRequired,
    inventory: PropTypes.number.isRequired
  })).isRequired,
  addToCart: PropTypes.func.isRequired
}
// mapStateToProps 用于建立组件和 store 中 state 的映射关系
const mapStateToProps = state => ({
  products: getVisibleProducts(state.products)
})
// connect 用于连接 React 组件与 Redux store
export default connect(
  mapStateToProps,
  { addToCart }
)(BooksContainer)

```

src/containers/CartContainer/index.jsx 商品购物车:

```

import React from 'react'
import PropTypes from 'prop-types'
import { connect } from 'react-redux'
import { checkout } from '../actions'
import { getTotal, getCartProducts } from '../reducers'
const CartContainer = ({ products, total, checkout }) => {
  const hasProducts = products.length > 0;
  return (
    <div>
      <h3>您的购物车</h3><em>请选择以上商品</em>

```

```

<div className="cart-container">
  <div>
    {
      products.map(product =>
        <div key={product.id}>{product.title} - {product.price}元
        {product.quantity ? ' x ${product.quantity} 本' : null}</div>
      )
    }
  </div>
  <p>总价是: {total}元</p>
</div>
<button className="submit-btn" onClick={() => checkout(products)}
       disabled={hasProducts ? '' : 'disabled'}>
  去结算
</button>
</div>
)
}
CartContainer.propTypes = {
  products: PropTypes.arrayOf(PropTypes.shape({
    id: PropTypes.number.isRequired,
    title: PropTypes.string.isRequired,
    price: PropTypes.number.isRequired,
    quantity: PropTypes.number.isRequired
  })).isRequired,
  total: PropTypes.string,
  checkout: PropTypes.func.isRequired
}
// mapStateToProps 用于建立组件和 store 中 state 的映射关系
const mapStateToProps = (state) => ({
  products: getCartProducts(state),
  total: getTotal(state)
})
// connect 用于连接 React 组件与 Redux store
export default connect(
  mapStateToProps,
  { checkout }
)(CartContainer)

```

4.4.6 发起一个动作 Action (添加商品到购物车)

以添加商品到购物车举例，用户单击页面中商品列表中的“添加到购物车”按钮。按钮代码如下：

```

<button
  className="add-btn"
  onClick={() => addToCart(product.id)}
  disabled={product.inventory > 0 ? '' : 'disabled'}>
  {product.inventory > 0 ? '添加到购物车' : '售罄'}
</button>

```

以上代码中，当 product.inventory 商品库存为 0 时，页面按钮展示文案为“售罄”，

并将按钮置为 disabled 不可点击；否则展示按钮为“添加到购物车”。单击“添加到购物车”这个事件时将触发一个 Action 动作 addToCart：

```
/** 
 * Action 创建函数
 * 加入购物车
 */
const addToCartAction = productId => ({
  type: types.ADD_TO_CART,
  productId
})
/** 
 * 加入购物车
 */
export const addToCart = productId => (dispatch, getState) => {
  if (getState().products.byId[productId].inventory > 0) {
    dispatch(addToCartAction(productId))
  }
}
```

以上述代码中，单击按钮时将商品 ID 传入 addToCart 这个 Action Creator，然后在 addToCart 内部再 dispatch 一个 Action 将该商品 ID 存到 Store，同时将库存减 1。代码如下：

src/reducers/products.js

```
const products = (state, action) => {
  switch (action.type) {
    case ADD_TO_CART:
      return {
        ...state,
        inventory: state.inventory - 1
      }
    default:
      return state
  }
}
```

添加商品到购物车的页面展示，如图 4.7 所示。

网上书店

The screenshot shows a user interface for a bookstore. At the top, there's a header '网上书店'. Below it, a section titled '您的购物车' (Your Shopping Cart) displays three items:

- 《JavaScript 权威指南》 - 136.2元 库存数量: 2
- 《Python核心编程》 - 44.5元 库存数量: 7
- 《C程序设计》 - 45元 库存数量: 2

Below the cart summary, a section says '请选择以上商品' (Select the above products). It lists the same three books with their quantities and unit prices. At the bottom, it shows the total price '总价是: 180.70元' (Total price: 180.70 yuan) and a button '去结算' (Go to Settlement).

图 4.7 添加商品至购物车