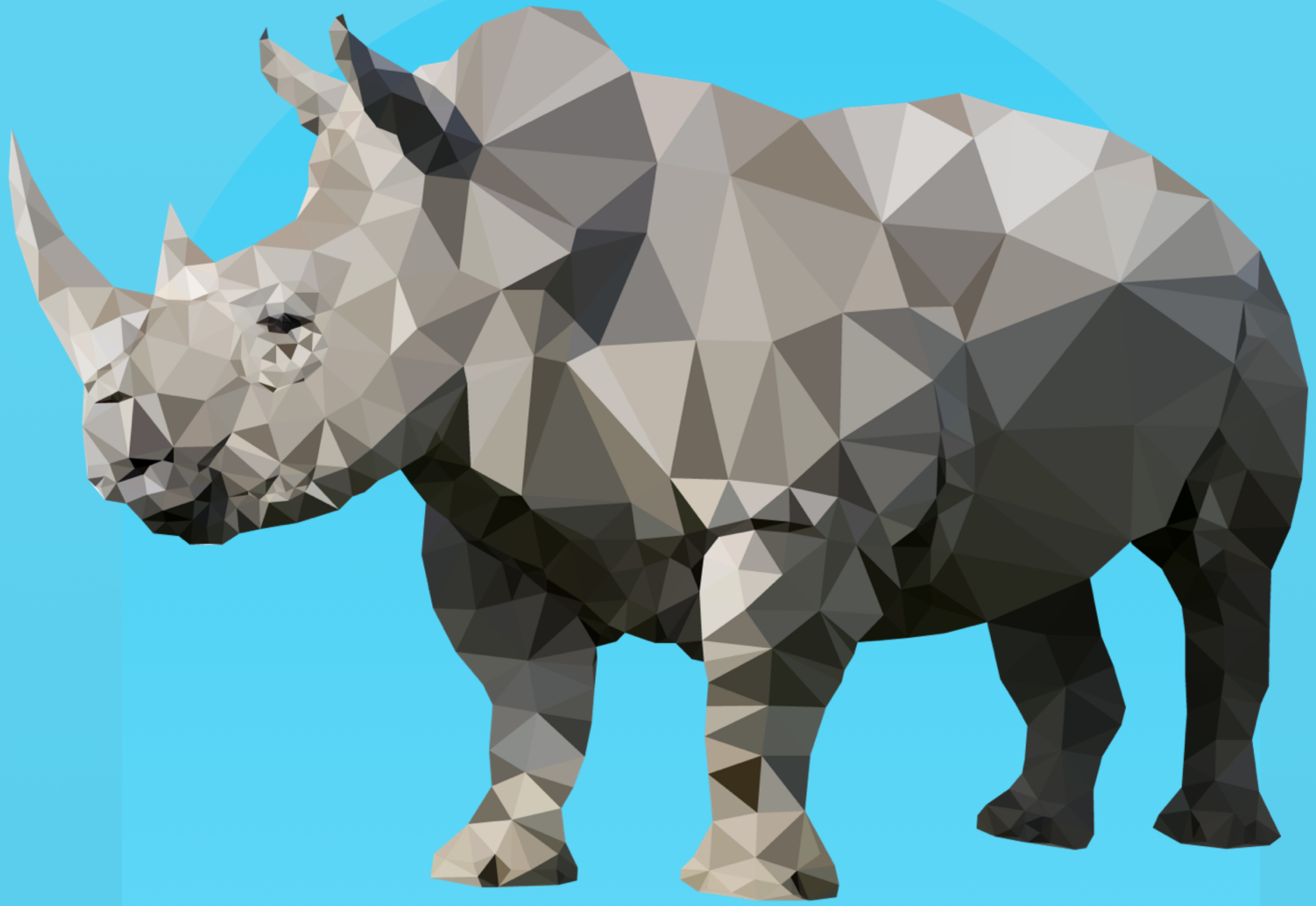
掘金小册



React 实战：设计模式和最佳实践

程墨 著

JJID-0034

React 的设计思想

React 是目前最火的用户界面开发库之一。它获得今天这样的业界地位并不是一个偶然，最根本的原因，是 React 汇聚了几十年用户界面开发的经验思想，贯彻了众多被证明行之有效的模式和原则。

这本小册，就是向大家讲解 React 应用中的模式和原则。

内容范围

读者应该已经对 React 有一个初步了解，熟悉 React 的基本用法，这本小册的目的是帮助读者进一步认识 React。读者在看完这本小册之后，应该不只是利用 React 来开发网页应用，还能够应用各种设计模式，开发出高效易于维护的代码。

如果读者对 React 还并不了解，可以先去看作者写的《深入浅出React和Redux》，本小册的目的是提高开发者水平，所以不会花太多篇幅讲解 React 的基础入门知识。

当然，React 技术的应用并不只限于网页，可以利用 React Native 来开发原生应用，但是受篇幅限制，本小册不探讨 React Native 的用法，而是集中介绍 React 技术在网页中的应用。

React 的基础原则

要真正理解 React，开发者必须要明白这几点：

1. React 界面完全由数据驱动；
2. React 中一切都是组件；
3. props 是 React 组件之间通讯的基本方式。

好的，让我们开始吧！

界面完全由数据驱动

初学 React 的开发者，尤其是用惯了 jQuery 的开发者，往往对 React 的工作方式很难接受，但是，请试着接受这一点，React 的哲学，简单说来可以用下面这条公式来表示：

```
UI = f(data)
```

等号左边的 UI 代表最终画出来的界面；等号右边的 f 是一个函数，也就是我们写的 React 相关代码；data 就是数据，在 React 中，data 可以是 state 或者 props。

UI 就是把 data 作为参数传递给 f 运算出来的结果。这个公式的含义就是，如果要渲染界面，不要直接去操纵 DOM 元素，而是修改数据，由数据去驱动 React 来修改界面。

我们开发者要做的，就是设计出合理的数据模型，让我们的代码完全根据数据来描述界面应该画成什么样子，而不必纠结如何去操作浏览器中的 DOM 树结构。

这样一种程序结构，是声明式编程（Declarative Programming）的方式，代码结构会更加容易理解和维护。

组件：React 世界的一等公民

React 是一个用 JavaScript 语言开发的库，而我们知道，在 JavaScript 的世界里，一切皆是对象，甚至连一个函数都是一个对象。

我们可以把一个函数当做对象来赋值给一个变量，可以把对象作为参数传递给一个函数，也可以访问一个函数的属性，函数完全就是一个对象。

比如，你是否知道，你可以访问一个函数的 length 来获知这个函数声明的参数？

```
function foo(a, b) {
  return a + b;
}

console.log(foo.length); // 输出为2
```

因为函数本身就是对象，所以，在 JavaScript 中，函数拥有一等公民的地位。

那么，在 React 的世界中，什么是一等公民呢？

答案就是**组件（Component）**。

可以这么说，在 React 中一切皆为组件。这是因为：

1. 用户界面就是组件；
2. 组件可以嵌套包装组成复杂功能；
3. 组件可以用来实现副作用。

接下来，我们逐个来看组件的这三个方面。

第一点用户界面就是组件，很好理解，在界面上看到的任何一个“块”，都需要代码来实现，而这部分代码最好是独立存在的，与其他代码之间的纠葛越少越好，所以要把这个“块”的相关代码封装在一个代码单元里。这样的代码单元，在 React 里就是一个“组件”。

在上面的图中，一个 Button 是一个界面元素，对应的就是一个 React 组件。在 React 中，一个组件可以是一个类，也可以是一个函数，这取决于这个组件是否有自己的状态。

第二点，组件可以嵌套包装组成复杂功能。现实中的应用是很复杂的，界面设计中包含很多元素，一个“块”套着另一个“块”，React 中的组件可以重复嵌套，就是为了支持现实中的用户界面需要。

第三点，组件可以用来实现副作用。并不是说组件必须要在界面画一些东西，一个组件可以什么都不画，或者把画界面的事情交给其他组件去做，自己做一些和界面无关的事情，比如获取数据。

下面是一个 Beacon 组件，它的 render 函数返回为空，所以它实际上并不渲染任何东西。

```
class Beacon extends React.Component {
  render() {
    return null;
  }

  componentDidMount() {
    const beacon = new Image();
    beacon.src = 'https://domain.name/beacon.gif';
  }
}
```

不过，Beacon 的 componentDidMount 函数中创造了一个 `Image` 对象，访问了一个特定的图片资源，这样就可以对应服务器上留下日志记录，用于记录这一次网页访问。

Beacon 组件的使用方式和普通组件别无二致，但是却能够轻松实现对网页访问的跟踪。

```
<div>
  <Beacon />
</div>
```

组件之间的语言：props

如果说组件是 React 世界的一等公民，这些公民之间也肯定是需要交流的，他们通过什么语言交流呢？答案就是 props。

如果一个父组件有话要对子组件说，也就是，想要传递数据给子组件，则应该通过 props。

当然，你可以给子组件增加一个新的函数，然后让父组件去调用这个函数，但是，这种方法很拙劣。如果直接调用子组件的函数，执行过程也处于 React 生命周期之外，所以，不应该使用这种方法。

同样，如果子组件有话要同父组件说，那应该支持函数类型的 props。身为 JavaScript 里一等公民的函数可以作为参数传递，当然也可以作为 props 传递。让父组件传递一个函数类型的 props 进来，当子组件要传递数据给父组件时，调用这个函数类型 props，就把信息传递给了父组件。

如果两个完全没有关系的组件之间有话说，情况就复杂了一点，比如下图中，两个橙色组件之间如果有话说，就没法直接通过 props 来传递信息。

一个比较土的方法，就是通过 props 之间的逐步传递，来把这两个组件关联起来。如果之间跨越两三层的关系，这种方法还凑合，但是，如果这两个组件隔了十几层，或者说所处位置多变，那让 props 跨越千山万水来相会，实在是得不偿失。

另一个简单的方式，就是建立一个全局的对象，两个组件把想要说的话都挂在这个全局对象上。这种方法当然简单可行，但是，我们都知道全局变量的危害罄竹难书，如果不想将来被难以维护的代码折磨，我们最好对这种方法敬而远之。

一般，业界对于这种场景，往往会采用第三方数据管理工具来解决，在本小册的第 12 和第 13 小节会详细介绍用 Redux 和 Mobx 解决这些问题的方法。

其实，不依赖于第三方工具，React 也提供了自己的跨组件通讯方式，这种方式叫 Context，在第 8 小节会详细介绍。

小结

本节介绍了使用 React 的基本思想，希望读者能够学习到：

1. 在 React 中，界面完全由数据驱动；
2. 在 React 中，一切都是组件；
3. props 是 React 组件之间通讯的基本方式。

在下一小节中，我们会投入实践，用 React 的基本原则来设计一个比较复杂的应用。

组件实践（1）：如何定义清晰可维护的接口

在第一节，我们已经知道，React 世界由组件构成，所以，如何设计组件的接口就成了组件设计最重要的事情。

设计原则

React 的组件其实就就是软件设计中的模块，所以其设计原则也遵从通用的组件设计原则，简单说来，就是要减少组件之间的耦合性（Coupling），让组件的界面简单，这样才能让整体系统易于理解、易于维护。

更具体一点，在设计 React 组件时，要注意以下原则：

1. 保持接口小，props 数量要少；
2. 根据数据边界来划分组件，充分利用组合（composition）；
3. 把 state 往上层组件提取，让下层组件只需要实现为纯函数。

说太多理论没意思，让我们用一个实际例子来解读这些原则，我们选择“秒表”这个程序来讲。

秒表曾经是一个体育专业人士才配备的工具，不过，现在大家的手机上肯定都有这样的应用，比如，在 iPhone 上的“时钟”里就如下图所示的秒表。

按下右侧“启动”按钮，这个按钮就会变成“停止”，同时上面的数字时钟开始计时；按下“停止”按钮，数字时钟停止计时。请注意左侧还有一个按钮，初始状态显示“复位”，点击该铵钮会清空时钟；开始计时之后，这个左侧按钮会变成“计次”，按一下“计次”，秒表底部就会增加一列时间，记录下按下“计次”这一瞬间的时刻。

秒表可以用来测量运动员的训练时间，比如运动员起跑的时候按“启动”，每跑一圈回到起点，按一下“计次”。这样跑十圈下来，可以知道每一圈都分别花了多少时间，运动员和教练可以做对应调整。

秒表是一个很实用的应用，复杂度也适当，这本小册中会以秒表为例展示 React 应用的开发，不过，我们无需急着写代码，首先来规划一下秒表的 React 组件接口如何设计。

组件的划分

我们会做一个 React 组件来渲染整个秒表，这个组件可以叫做 StopWatch，目前看来这个组件不需要从外部获得什么输入，本着“props 数量要少”的原则，我们也不需要费心未来会用上什么 props，目前就当 StopWatch 不支持 props 好了。

此外，这个组件需要记录当前计时，还要记录每一次按下“计次”的时间，所以需要维持状态（state），可以肯定，StopWatch 是一个有状态的组件，不能只是一个纯函数，而是一个继承自 Component 的类。

```
class StopWatch extends React.Component {
  render() {
     //TODO: 返回所有JSX
  }
}
```

任何一个复杂组件都是从简单组件开始的，一开始我们在 render 函数里写的代码不多，但是随着逻辑的复杂，JSX 代码越来越多，于是，就需要拆分函数中的内容。

在 React 中，有一个误区，就是把 render 中的代码分拆到多个 renderXXXX 函数中去，比如下面这样：

```
class StopWatch extends React.Component {
  render() {
    const majorClock = this.renderMajorClock();
    const controlButtons = this.renderControlButtons();
    const splitTimes = this.renderSplitTimes();

    return (
       <div>
          {majorClock}
          {controlButtons}
          {splitTimes}
       </div>
    );
  }

  renderMajorClock() {
     //TODO: 返回数字时钟的JSX
  }

  renderControlButtons() {
     //TODO: 返回两个按钮的JSX
  }

  renderSplitTimes() {
     //TODO: 返回所有计次时间的JSX
  }
}
```

用上面的方法组织代码，当然比写一个巨大的 render 函数要强，但是，实现这么多 renderXXXX 函数并不是一个明智之举，因为这些 renderXXXX 函数访问的是同样的 props 和 state，这样代码依然耦合在了一起。更好的方法，是把这些 renderXXXX 重构成各自独立的 React 组件，像下面这样：

```
class StopWatch extends React.Component {
  render() {
    return (
       <div>
          <MajorClock>
          <ControlButtons>
          <SplitTimes>
       </div>
    );
  }
}

const MajorClock = (props) => {
  //TODO: 返回数字时钟的JSX
};

const ControlButtons = (props) => {
  //TODO: 返回两个按钮的JSX
};
```

```
const SplitTimes = (props) => {
  //TODO: 返回所有计次时间的JSX
}
```

我们创造了 `MajorClock`、`ControlButtons` 和 `SplitTimes` 这三个组件，目前，我们并不知道它们是否应该有自己的 state，但是从简单开始，首先假设它们没有自己的 state，定义为函数形式的无状态组件。

按照数据边界来分割组件

现在，我们来检视一下，这样的组件划分，是否符合“按照数据边界划分”的原则。

渲染 `MajorClock`，需要的是当前展示的时间，在点击“启动”按钮之后，这个时间是不断增长的。

渲染 `ControlButtons`，两个按钮显示什么内容，完全由当前是否是“启动”的激活状态决定。此外，Buttons 是秒表中唯一有用户输入的组件，对于按钮的按键会改变秒表的状态。

最后，计次时间 `SplitTimes`，需要渲染多个时间，可以想象，需要有一个数组来记录所有计次时间。

总结一下所有需要的数据和对应标识符，以及影响的组件：

数据

标识符

影响的组件

当前时间

timeElapsed

MajorClock

是否启动

activated

MajorClock, ControlButtons

计次时间

splits

SplitTimes

从上面的表格可以看出，每个数据影响的组件都不多，唯一影响两个组件的数据是 activated，这个 activated 基本上就是一个布尔值，数据量很小，影响两个组件问题也不大。

这样的组件划分是符合以数据为边界原则的，很好。

state 的位置

接下来，我们要确定 state 的存储位置。

当秒表处于启动状态，MajorClock 会不断更新时间，似乎让 MajorClock 来存储时间相关的 state 很合理，但是仔细考虑一下，就会发现这样并不合适。

设想一下，MajorClock 包含一个 state 记录时间，因为 state 是组件的内部状态，只能通过组件自己来更新，所以要 MajorClock 用一个 `setTimeout` 或者 `setInterval` 来持续更新这个 state，可是，另一个组件 ControlButtons 将会决定什么时候暂停 MajorClock 的 state 更新，而且，当用户按下“计次”按钮的时候，MajorClock 还需要一个方法把当前的时间通知给 SplitTimes 组件。

这样一个数据传递过程，想一想都觉得很麻烦，明显不合适。

这时候就需要考虑这样的原则，**尽量把数据状态往上层组件提取。**在秒表这个应用中，上层组件就是 StopWatch，如果我们让 StopWatch 来存储时间状态，那一切就会简单很多。

StopWatch 中利用 setTimeout 或者 setInterval 来更新 state，每一次更新会引发一次重新渲染，在重新渲染的时候，直接把当前时间值传递给 MajorClock 就完事了。

至于 ControlButtons 对状态的控制，让 StopWatch 传递函数类型 props 给 ControlButtons，当特定按钮时间点击的时候回调这些函数，StopWatch 就知道何时停止更新或者启动 setTimeout 或者 setInterval，因为这一切逻辑都封装在 StopWatch 中，非常直观自然。

对了，还有 SplitTimes，它需要一个数组记录所有计次时间，这些数据也很自然应该放在 StopWatch 中维护，然后通过 props 传递给 SplitTimes，这样 SplitTimes 只单纯做渲染就足够。

组件 props 的设计

当我们确定了组件结构和 state 之后，剩下来要做的就是 props 的设计了。

先来看 MajorClock，因为它依赖的数据只有当前时间，所以只需要一个 props。

```
const MajorClock = ({milliseconds}) => {
  //TODO: 返回数字时钟的JSX
};

MajorClock.propTypes = {
  milliseconds: PropTypes.number.isRequired
};
```

和函数参数的命名一样，props的命名一定力求简洁而且清晰。对于MajorClock，如果把这个props命名为 `time`，很容易引起歧义，这个 time 的单位是什么？是毫秒？还是秒？还是一个 Date 对象？

所以，我们明确传入的 props 是一个代表毫秒的数字，所以命名为 `milliseconds`，如果你的开发团队可以接受，也可以简写为 `ms`。

然后是 ControlButtons，这个组件需要根据当前是否是“启动”状态显示不同的按钮，所以需要一个 props 来表示是否“启动”，我们把它命名为 `activated`。

此外，StopWatch 还需要传递回调函数给 ControlButtons，所以还需要支持函数类型的 props，分别代表 ControlButtons 可以做的几个动作：

- 启动（start）
- 停止（pause）
- 计次（split）
- 复位（reset）

一般来说，为了让开发者能够一眼认出回调函数类型的 props，这类 props 最好有一个统一的前缀，比如 `on` 或者 `handle`，我个人倾向于用 on，所以，ControlButtons 的接口就是下面这样：

```
const ControlButtons = (props) => {
  //TODO: 返回两个按钮的JSX
};

ControlButtons.propTypes = {
  activated: PropTypes.bool,
  onStart: PropTypes.func.isRquired,
  onPause: PropTypes.func.isRquired,
  onSplit: PropTypes.func.isRquired,
  onReset: PropTypes.func.isRquired,
};
```

最后是 SplitTimes，很简单，它需要接受一个数组类型的 props。

你知道吗？PropTypes 也可以支持数组类型的定义哦。

```
const SplitTimes = (props) => {
  //TODO: 返回所有计次时间的JSX
}

SplitTimes.propTypes = {
  splits: PropTypes.arrayOf(PropTypes.number)
};
```

至此，完成了秒表的组件接口设计，我们还完全没有涉及组件内部的具体代码编写，这在后面的小节中会详细讲解，不过，一个好的设计就是要在写代码之前就应用被证明最佳的原则，这样写代码的过程就会少走弯路。

小结

在这一节中，我们通过设计秒表，展示了组件接口设计的三个原则：

1. 保持接口小，props 数量要少
2. 根据数据边界来划分组件，利用组合（composition）
3. 把 state 尽量往上层组件提取

同时，我们也接触了这样一些最佳实践：

1. 避免 renderXXXX 函数
2. 给回调函数类型的 props 加统一前缀
3. 使用 propTypes 来定义组件的 props

组件实践（2）：组件的内部实现

上一小节中，通过设计一个“秒表”应用，我们学习了组件的接口定义的原则和方法，但那只是搭建了一个骨架，在这一小节中，我们就给这个骨架填充血肉，制造出能够运转的“秒表”。

我们不大可能一次就写出完美的代码，软件开发本来就是一个逐渐精进的过程，但是我们应该努力让代码达到这样的要求：

1. 功能正常；
2. 代码整洁；
3. 高性能。

初始化应用框架

我们使用 Facebook 提供的 create-react-app 来创建我们的 React 应用。严格说来，如果要开发一款真正大型的应用，用 create-react-app 并不合适，或者至少 create-react-app 的默认配置并不合适，真正的应用需要对 webpack 等做更精细的配置，但是，create-react-app 足够简单易用，从学习 React 的角度来看，真的是非常合适了。

如果你的机器上还没有安装 create-react-app，那么就使用下面的命令来全局安装：

```
npm install -g create-react-app
```

然后，找一个合适的目录，使用下面的命令来创建我们的应用框架，在这里，我们的应用名字叫 `basic_stop_watch`。

```
create-react-app basic_stop_watch --use-npm
```

create-react-app 会优先使用 yarn 来安装依赖的 npm 包，但是不知道读者机器上有没有安装 yarn，所以用 `--use-npm` 参数强制 create-react-app 使用传统的 npm 工具来安装依赖的包，这样能够保持一致。如果你更喜欢 yarn，不使用 `--use-npm` 这个参数就可以了。

创建应用框架需要花费一些时间，因为现在依赖的包实在太多太细碎了，在完成之后，会创建一个 `basic_stop_watch` 目录，进入这个目录，运行下面给的命令，就可以启动给一个基本的 React 应用。

```
npm start
```

我们不会花费太多时间介绍 create-react-app 的其他功能和代码结构，因为我们有更重要的事情要做，那就是根据上一小节的组件设计来制造“秒表”相关组件。

构建 StopWatch

在上一小节中，我们已经确定了要用四个组件组合来实现“秒表”，这四个组件分别是 StopWatch、MajorClock、ControlButtons 和 SplitTimes。

我们在写代码之前要做的第一个决定就是，要把这些组件放在哪里？是放在不同的文件中？还是放在一个文件中就好？

表面上看，把所有组件放在一个文件中也行得通，但是，将来维护代码的朋友可能会很抓狂，想要修改 ControlButtons 这个组件，但是从文件目录里找不到对应文件，这样很不方便。

所以，从达到"代码整洁"的目的来说，应该每个组件都有一个独立的文件，然后这个文件用 `export default` 的方式导出单个组件。

比如，我们会在 `src` 目录下为 ControlButtons 创建一个 `ControlButtons.js` 文件，最初内容像下面这样：

```
import React from 'react';

const ControlButtons = () => {
  //TODO: 实现ControlButtons
};

export default ControlButtons;
```

第一行导入 React，虽然目前没有派上什么用场，但是任何 JSX 都需要 React，很快我们在实现 ControlButtons 这个控件的内容时，就要写 JSX，所以导入 React 是必需的。

最后一行用 `export default` 的方式导出 ControlButtons，这样，在其他组件中就可以用下面的方式导入：

```
import ControlButtons from './ControlButtons';
```

在上一节中我们已经设计好了 ControlButtons 可以接受的 props，我们重试 ControlButtons 的实现代码，如下：

```
const ControlButtons = (props) => {
  const {activated, onStart, onPause, onReset, onSplit} = props;
  if (activated) {
    return (
      <div>
        <button onClick={onSplit}>计次</button>
        <button onClick={onPause}>停止</button>
      </div>
    );
  } else {
    return (
      <div>
        <button onClick={onReset}>复位</button>
        <button onClick={onStart}>启动</button>
      </div>
    );
  }
};
```

在这里用了一个 ES6 功能，叫做**解构赋值**（Destructuring Assignment）。因为 ControlButtons 是一个函数类型的组件，所以 props 以参数形式传递进来，props 中的属性包含 `activated` 这样的值，利用大括号，就可以完成对 props 的“解构”，把 props.activated 赋值给同名的变量 `activated`。

如果没有解构赋值，就只能用下面的代码，很明显，与使用了解构赋值的代码相比，真是啰嗦得让人难以忍受。

```
  const activated = props.activated;
  const onStart = props.onStart;
  const onPause = props.onPause;
  const onReset = props.onReset;
  const onSplit = props.onSplit;
```

我们可以更进一步，把解构赋值提到参数中，这样连 props 的对象都看不见，就像下面这样：

```
const ControlButtons = ({activated, onStart, onPause, onReset, onSplit}) => {

}
```

在 ControlButtons 的实现部分，我们根据 activated 的值返回不同的 JSX，当 activated 为 `true` 时，返回的是“计次”和“停止”；当 activated 为 `false` 时，返回的是“复位”和“启动”，对应分别使用了传入的 `on` 开头的函数类型 props。

可以看到，ControlButtons 除了显示内容和分配 props，没有做什么实质的工作，实质工作会在 StopWatch 中介绍。

接下来我们做 MajorClock，根据传入 props 的 milliseconds 来显示数字时钟一样的时分秒。在 MajorClock.js 文件中，我们这样定义 MajorClock：

```
const MajorClock = ({milliseconds=0}) => {
  return <h1>{ms2Time(milliseconds)}</h1>
};
```

在这里，我们不光直接解构了参数，而且使用了默认值。如果使用 MajorClock 时没有传入 `milliseconds` 这个 props，那么 `milliseconds` 的值就是 `0`。

因为把毫秒数转为 `HH:mm:ss:mmm` 这样的格式和 JSX 没什么关系，所以，我们不在组件中直接编写，而是放在 `ms2Time` 函数中，`ms2Time` 就是 `ms-to-Time`，代码如下：

```
import padStart from 'lodash/padStart';

const ms2Time = (milliseconds) => {
  let time = milliseconds;
  const ms = milliseconds % 1000;
  time = (milliseconds - ms) / 1000;
  const seconds = time % 60;
  time = (time - seconds) / 60;
  const minutes = time % 60;
  const hours = (time - minutes) / 60;
```

```
  const result = padStart(hours, 2, '0') + ":" + padStart(minutes, 2, '0') + ":"
+ padStart(seconds, 2, '0') + "." + padStart(ms, 3, '0');
  return result;
}
```

通过逐步从 milliseconds 中抽取毫秒、秒、分、时的信息，最终拼出人类容易理解的时间。不过，为了和数字时钟显示一致，需要补齐，比如 2 秒 23 毫秒，显示成 `2:23` 可不好看，不够的位数要补上 0，显示成 `00:00:02:023` 。这个补齐的工作和 React 无关，我们也不深究，直接使用 `lodash` 中的 `padStart` 实现。

为了在项目中使用 `lodash`，请先用 npm 完成对应的库安装。

最后是 SplitTimes 这个组件，在 `SplitTimes.js` 这个文件中，我们需要这样定义 SplitTimes：

```
import MajorClock from './MajorClock';

const SplitTimes = ({value=[]}) => {
  return value.map((v, k) => (
    <MajorClock key={k} milliseconds={v} />
  ));
};
```

因为根据毫秒数显示数字时钟的功能在 MajorClock 中已经做到了，所以我们直接导入 MajorClock 使用就好，这符合“重用代码”的原则。

值得一提的是，利用循环或者数组 map 而产生的动态数量的 JSX 元件，必须要有 `key` 属性。这个 `key` 属性帮助 React 搞清楚组件的顺序，如果不用 `key`，那 React 会在开发模式下在 console 上输出红色警告。

一般来说，key 不应该取数组的序号，因为 key 要唯一而且稳定，也即是每一次渲染过程中，key 都能唯一标识一个内容。对于 StopWatch 这个例子，倒是可以直接使用数组序号，因为计次时间的数组顺序不会改变，使用数组序号足够唯一标识内容。

StopWatch 状态管理

在实现了 MajorClock、ControlButtons 和 SplitTimes 之后，我们需要把这些子组件串起来，这就是 StopWatch。

StopWatch 是一个有状态的组件，所以，不能只用一个函数实现，而是做成一个继承自 `React.Component` 的类，如下：

```
class StopWatch extends React.Component {
  render() {
    return (
      <Fragment>
        <MajorClock />
        <ControlButtons />
        <SplitTimes />
      </Fragment>
    );
  }
}
```

对于一个 React 组件类，最少要有一个 `render` 函数实现，不过，上面的 `render` 只是一个大概的代码框架，引用了相关子组件，但是没有传入 props。

传入什么 props 呢？当然是 StopWatch 记录的 state。

StopWatch 的 state 需要有这些信息：

1. isStarted，是否开始计时；
2. startTime，计时开始时间，Date 对象；
3. currentTime，当前时间，也是 Date 对象；
4. splits，所有计次时间的数组，每个元素是一个毫秒数。

React 组件的 state 需要初始化，一般来说，初始化 state 是在构造函数中，代码如下：

```
  constructor() {
    super(...arguments);

    this.state = {
      isStarted: false,
      startTime: null,
      currentTime: null,
      splits: [],
    };
  }
```

如果定义构造函数 constructor，一定要记得通过 super 调用父类 `React.Component` 的构造函数，不然，功能会不正常。

React官方网站上的代码示例是这样调用super函数：

```
  constructor(props) {
    super(props); //目前可行，但有更好的方法
  }
```

在早期版本中，React.Component 的构造函数参数有两个，第一个是 `props`，第二个是 `context`，如果忽略掉 `context` 参数，那么这个组件的 context 功能就不能正常工作，不过，现在React的行为已经变了，第二个参数传递不传递都能让context正常工作，看起来React.Component 的构造函数只有第一个参数被用到，但是，没准未来还会增加新的参数呢，所以，以不变应万变的方法，就是使用扩展

操作符（spread operator）来展开 arguments，这样不管 React 将来怎么变，这样的代码都正确。

```
  constructor() {
    super(...arguments); //永远正确!
  }
```

扩展操作符的作用，在 React 开发中会经常用到，在 JSX 中展开 props 的时候会用到。

属性初始化方法

不过，其实我们也可以完全避免编写 constructor 函数，而直接使用属性初始化（Property Initializer），也就是在 class 定义中直接初始化类的成员变量。

不用 constructor，可以这样初始化 state，效果是完全一样的：

```
class StopWatch extends React.Component {
  state = {
    isStarted: false,
    startTime: null,
    currentTime: null,
    splits: [],
  }
}
```

接下来，我们要考虑如何实现传递给 ControlButtons 的一系列函数。

首先，明确一点，尽量不要在 JSX 中写内联函数（inline function），比如这样写，是很不恰当的：

```
        <ControlButtons
          activated={this.state.isStarted}
          onStart={() => { /* TODO */}}
          onPause={() => { /* TODO */}}
          onReset={() => { /* TODO */}}
          onSplit={() => { /* TODO */}}
        />
```

当然，按照上面那种写法，也可以完成程序的功能，但是，会带来性能的代价。首先，每一次渲染这段 JSX，都会产生全新的函数对象，这是一种浪费；其次，因为每一次传给 ControlButtons 的都是新的 props，这样 ControlButtons 也无法通过 `shouldComponentUpdate` 对 props 的检查来避免重复渲染。

在本小册中后续章节中，为了代码简洁会使用内联函数，但只是为了示例方便，在实际工作中，在 JSX 中应用的函数 props 应该尽量使用类成员函数，不要用内联函数。

以最容易实现的 `onSplit` 为例，这个函数响应用户点击“计次”按钮的事件，代码如下：

```
  onSplit() {
    this.setState({
      splits: [...this.state.splits, this.state.currentTime -
this.state.startTime]
    });
  }
```

在 onSplit 中，利用 `this.setState` 来修改组件的状态。不过问题来了，这个函数执行时，`this` 是什么呢?

很可惜，对于 ES6 的类成员函数，并不自动绑定 `this`，也就是说，onSplit 中的 `this`，可不保证就是当前组件对象。

至于 `render` 这些生命周期函数，里面访问的 `this` 就是当前组件本身，完全是因为这些函数是 React 调用的，React 对它们进行了特殊处理，对于其他普通的成员函数，特殊处理就要靠我们自己了。

通常的处理方法，就是在构造函数中对函数进行绑定，然后把新产生的函数覆盖原有的函数，就像这样:

```
  constructor() {
    super(...arguments);

    this.onSplit = this.onSplit.bind(this);
  }
```

如果可以使用 bind operator，也可以这样写:

```
    this.onSplit = ::this.onSplit;
```

只可惜 bind operator 并不是稳定的标准语法，而 create-react-app 又不想依赖不稳定的语法，所以在我们的应用中还不能这么写。

我们的 StopWatch 需要给 ControlButtons 传递四个函数类型的 props，分别是 `onStart`、`onPause`、`onReset` 和 `onSplit`，对每一个函数都在构造函数里加一个 bind，也是够累的，还容易出错，所以，我们肯定会寻求更好的方法。

更好的方法依然是使用属性初始化，就和初始化 state 一样，利用等号直接初始化 onSplit，代码如下:

```
  onSplit = () => {
    this.setState({
      splits: [...this.state.splits, this.state.currentTime -
this.state.startTime]
    });
  }
```

像上面这样写，就不需要 constructor，函数体内的 `this` 绝对就是当前组件对象。

用同样的方法，我们一起实现其他函数成员。

```
  onStart = () => {
    this.setState({
      isStarted: true,
      startTime: new Date(),
      currentTime: new Date(),
    });

    this.intervalHandle = setInterval(() => {
      this.setState({currentTime: new Date()});
    }, 1000 / 60);
  }

  onPause = () => {
    clearInterval(this.intervalHandle);
    this.setState({
      isStarted: false,
    });
  }

  onReset = () => {
    this.setState({
      startTime: null,
      currentTime: null,
      splits: [],
    });
  }
```

至此，一个“秒表”的功能就完成了，在 `App.js` 文件中导入 `StopWatch`，在浏览器中我们可以看到这样的界面。

点击“启动”按钮，可以看见数字时钟开始运转；点击“计次”按钮，在按钮下方可以看到点击瞬间的时间；点击“停止”，时钟停止运转。

当然，现在这个“秒表”的界面还非常粗糙，和 iPhone 上的秒表应用差远了。但是，它该有的功能一个都不缺，只有功能完整而且正确，样式才有意义。

在后面的章节中，我们会用 React 的方式来来美化界面。

小结

在这一小节中，我们完成 StopWatch 秒表组件的实现，在这个过程中，读者应该学习到这些技巧：

1. 尽量每个组件都有自己专属的源代码文件；
2. 用**解构赋值**（destructuring assignment）的方法获取参数 props 的每个属性值；
3. 利用**属性初始化**（property initializer）来定义 state 和成员函数。

组件实践（3）：组件化样式

在上一节中，我们创造了“秒表”组件，但是那个“秒表”的样式很粗糙，因为我们只实现了功能，而不关注样式。并不是说样式不重要，实际上，样式美观是前端开发的一个重要部分，只是做事有先有后，用渐进式开发的原则，先搞定功能，然后再来处理样式。

在这一节中，我们要探讨如何给 React 组件增加样式，让“秒表”这个应用看起来更美观。

React 带来的对样式管理革命

在我刚入行的时候，业界对前端开发普遍有这样的认知，把网页应用分为三层，分别是用 HTML 实现的“内容”，用 CSS 实现的“样式”，还有用 JavaScript 实现的“动态行为”。

就像上面这样，构建一个网页应用，首先用 HTML 展示内容。比如，我们展示一个新闻，首先用 HTML 的 `p` 标签展示文字，用 `img` 标签展示图片，这样，即使没有 CSS 和 JavaScript，至少用户还可以看见新闻内容。

当然，单纯只有 HTML，那网页内容也太枯燥了，所以还是需要 CSS 来增加一些多彩的样式，修改一下字体、颜色、阴影之类，这是第二层的功能。

最后，进一步提高用户体验，就要 JavaScript 上场了，在 HTML 和 CSS 的基础上赋予网页动态交互的功能，比如给新闻网页增加一个“点赞”的功能，这是 HTML 和 CSS 无法做到的。

按照这样“内容”->“样式”->“动态功能”的方式渐进增强（Progressive Enhancement），是非常正确的思想。但是，长期以来，实现方式存在问题，问题就是 HTML、CSS 和 JavaScript 被分开管理了。

如果你在 React 诞生之前从事过网页开发，肯定有这样的体会。为了修改一个功能，需要牵扯到 HTML、CSS 和 JavaScript 的修改，但是这三部分分别属于不同的文件，明明是一个功能，你却要去修改至少三个文件。

在软件开发中，同一个功能相关的代码最好放在一个地方，这就是高内聚性（High Cohesiveness）。把网页功能分在 HTML、CSS 和 JavaScript 中，明显背离了高内聚性的原则，但是我们也忍受了这个做法这么多年，直到 React 出现。

在实现“秒表”的时候，我们已经可以看到，“内容”和“动态功能”已经混合在一起，换句话说，长得很像 HTML 的 JSX 负责产生“内容”，和各种响应用户输入的 JavaScript 代码共同存在于 React 组件之中。

在 React 中，当你要修改一个功能的内容和行为时，在一个文件中就能完成，这样就达到了高内聚的要求。

那么，在 React 中又是如何处理样式的呢？

我们先从组件的 style 属性开始，最后过渡到组件式的样式。

style 属性

在上一小节实现的“秒表”中，虽然功能齐备，但是展示上有一个大问题，就是当时钟开始运转之后，因为各个数字的宽度不同，比如 `1` 就没有 `0` 宽，导致时间宽度忽大忽小，产生闪烁效果，这样看起来很不专业。

为了解决这个问题，我们就需要定制 MajorClock 的样式。

最简单也是最直接的方法，就是给对应的 React 元素增加 style 属性，属性值为一个普通的 JavaScript 对象，如下所示：

```
const MajorClock = ({milliseconds=0}) => {
  const style = {
    'font-family': 'monospace'
  };
  return <h1 style={style}>{ms2Time(milliseconds)}</h1>
}
```

在上面的例子中，我们把 MajorClock 中的 `h1` 元素的 `font-family` 设为 `monospace`,`monospace` 是等宽字体，这样所有数字所占宽度相同，数字变化起来的时候宽度也就不会发生变化了，效果图如下：

你可能也见过有人像下面这么写：

```
const MajorClock = ({milliseconds=0}) => {
  return <h1 style={{
    'font-family': 'monospace'
  }}>{ms2Time(milliseconds)}</h1>
}
```

上面这种写法，并不好。因为每次渲染 MajorClock 组件都会创建一个新的 style 对象，纯粹就是浪费。

如果 style 对象每次都是一样的，最好把它提取到组件之外，这样就可以重用一个对象，像下面这样：

```
const clockStyle = {
  'font-family': 'monospace'
};

const MajorClock = ({milliseconds=0}) => {
  return <h1 style={clockStyle}>{ms2Time(milliseconds)}</h1>
}
```

导入 CSS 文件

长期来，前端开发者都习惯了使用 CSS 来定制样式，React 也支持这种做法。

我们以 ControlButtons 为例，改进控制按钮的样式。

为了配合 CSS，我们要在 ControlButtons 的 JSX 中让渲染出来的 DOM 元素包含 class。

```
const ControlButtons = ({activated, onStart, onPause, onReset, onSplit}) => {
  if (activated) {
    return (
      <div>
        <button className="left-btn" onClick={onSplit}>计次</button>
        <button className="right-btn" onClick={onPause}>停止</button>
      </div>
    );
  } else {
    return (
```

```
      <div>
        <button className="left-btn" onClick={onReset}>复位</button>
        <button className="right-btn" onClick={onStart}>启动</button>
      </div>
    );
  }
};
```

值得一提的是，虽然最终产生的 DOM 或者 HTML 中属性为 class，在 JSX 中不能用 class，要用 className 来指定元素的类名。

然后，我们在 `ControlButtons.js` 中增加下面这样，导入一个同目录下的 `ControlButtons.css` 文件：

```
import "./ControlButtons.css";
```

create-react-app 会用 webpack 完成打包过程，只要 JavaScript 文件中应用的资源，都会被打包进最终的文件，所以，`ControlButtons.css` 中的样式规则就会被应用。

`ControlButtons.css` 中的内容如下：

```
.left-btn, .right-btn {
  border-radius: 50%;
  width: 70px;
  height: 70px;
}

.left-btn {
  margin: 0 35px 0 0;
}

.right-btn {
  margin: 0 0 0 35px;
}
```

最终的效果图如下：

可以看到，按钮之间有了合适的距离，而且边缘也和 iPhone 上的“秒表”一样显示为圆形。

你已经做到接近 iPhone 外观的“秒表”应用了，给自己鼓鼓掌吧！

组件式的样式

对比使用 style 属性和导入 CSS 两种方法，可以看出各有优缺点。

使用 style 属性，好处是可以将样式应用到每个元素，互相不干扰；缺点就是非常不简洁，如果你想要定制一个元素的样式，必须给这个元素加 style 属性。

比如，我们想让 MajorClock 中的 `h1` 元素字体为 `monospace`，使用 style 属性来实现，就要给 `h1` 加上 style，如果只有一个 `h1` 元素还好应付，如果很多 `h1` 元素，就非常麻烦。

```
const style={
  'font-family': 'monospace'
};

  <h1 style={style}>...</h1>
```

相反，用 CSS 表达复杂的样式规则很容易，比如，上一段提到的样式，用 CSS 轻松可以实现，而且不用给每个 `h1` 加什么 style 属性。

```
h1 {
  font-family: monospace;
}
```

不过，CSS 也有它的缺点，CSS 定义的样式是全局的，这样很容易失控，比如上面的 CSS 规则，一旦导入，那么所有的 `h1` 都具备这样的样式，即使不在 MajorClock 中的`h1` 元素，一样被 MajorClock 导入的 CSS 文件影响。

为了解决不同模块之间 CSS 互相干扰的问题，前端开发者想出了好多种解决方法，基本原则就是给 CSS 规则增加更加特定的限制。比如，要限定上面的 CSS 规则只作用于 MajorClock 中的 `h1` 元素，就要这样来写一个 `MajorClock.css`：

```
.clock h1 {
  font-family: monospace;
}
```

但是，你也需要修改 MajorClock 的 JSX，让 `h1` 包含在一个类名为 `clock` 的元素中。

```
import "./MajorClock.css";

const MajorClock = ({milliseconds=0}) => {
  return (
    <div class="clock">
     <h1 style={clockStyle}>{ms2Time(milliseconds)}</h1>
    </div>
  );
}
```

这样当然可行，但是，开发者不好处理 JSX 和 CSS 之间的关系，而且，就像在上面说过的，这样违背高内聚的原则。当你需要修改一个组件时，要被迫去分别修改 JavaScript 文件 和 CSS 文件，明显不是最优的方法。

我曾说过，在 React 的世界中，一切都是组件，所以很自然诞生了组件化的样式(Component Style）。

组件化样式的实现方式很多，这里我们介绍最容易理解的一个库，叫做 `styled-jsx`。

添加 styled-jsx 支持

要使用 styled-jsx，必须要修改 webpack 配置，一般来说，对于用 create-react-app 创建的应用，需要用 `eject` 方法来"弹射"出配置文件，只是，`eject` 指令是不可逆的，不到万不得已，我们还是不要轻易"弹射"。

一个更简单的方式，是使用 `react-app-rewired`，不需要 `eject`，轻轻松松就能够修改 create-react-app 产生应用的配置方法。

首先，我们在项目中安装 react-app-rewired 和 styled-jsx。如果读者使用的是npm v5之前的版本，最好添加 `--save` 参数用于修改 `package.json`，如果使用npm v5之后版本，则无需添加 `--save` 参数。

```
npm install react-app-rewired styled-jsx
```

然后，打开 `package.json` 文件，找到 `scripts` 这个部分，应该是下面这样：

```
   "scripts": {
     "start": "react-scripts start",
     "build": "react-scripts build",
     "test": "react-scripts test --env=jsdom",
     "eject": "react-scripts eject"
   }
```

当我们在命令行执行 `npm start` 时，执行的就是 `scripts` 部分定义的指令，可以看到都是执行 `react-scripts`。

在这里还可以看到 `eject` 指令的定义，我们做这个修改，恰恰就是为避免使用 `eject`。

我们修改 `scripts` 部分的代码如下:

```
   "scripts": {
    "start": "react-app-rewired start",
    "build": "react-app-rewired build",
    "test": "react-app-rewired test --env=jsdom",
    "eject": "react-scripts eject"
```

修改的方法其实就是把 `start`、`build` 和 `test` 对应脚本中的 `react-scripts` 替换为 `react-app-rewired`，之后，当用 npm 执行这些指令的时候，就会使用 react-app-rewired。

react-app-rewired 扩展了 react-scripts 的功能，可以从当前目录的 `config-overrides.js` 文件中读取配置，扩充 react-scripts 的功能。

我们需要让 react-scripts 支持 styled-jsx，对应只需要在项目根目录增加一个 `config-overrides.js` 文件，内容如下：

```
const { injectBabelPlugin } = require('react-app-rewired');

module.exports = function override(config, env) {
  config = injectBabelPlugin(['styled-jsx/babel'], config);

  return config;
};
```

上面 config-overrides.js 文件中的逻辑很简单，就是把 `styled-jsx/babel` 注入到 react-scripts 的基本配置中去，然后，我们的应用就支持 styled-jsx 了。

使用 styled-jsx 定制样式

有了 styled-jsx 中，我们就可以在 JSX 中用 `style jsx` 标签直接添加 CSS 规则。

比如，我们要给 MajorClock 中的 `h1` 增加 CSS 规则，可以这样使用：

```
const MajorClock = ({milliseconds=0}) => {
  return (
    <React.Fragment>
      <style jsx>{`
        h1 {
          font-family: monospace;
        }
      `}</style>
      <h1>
        {ms2Time(milliseconds)}
      </h1>
    </React.Fragment>
  );
};
```

注意紧贴 style jsx 内部的是一对大括号，大括号代表里面是一段 JavaScript 的表达式，再往里，是一对符号，代表中间是一段多行的字符串，也就是说，style jsx 包裹的是一个字符串表达式，而这个字符串就是 CSS 规则。

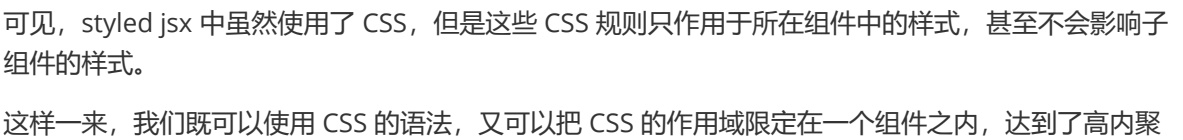
在 MajorClock 中用 style jsx 添加的 CSS 规则，只作用于 MajorClock 的 JSX 中出现的元素，不会影响其他的组件。

你可以尝试在其他组件中添加 `h1` 元素，也可以尝试在其他组件中添加 `style jsx` 标签来定制 `h1` 的样式，会发现和 MajorClock 完全是井水不犯河水，互不影响。

我在 StopWatch 中添加一个 `h1` 元素，内容就是“秒表”，然后用 `style jsx` 把 `h1` 的颜色设为绿色，代码如下：

```
  render() {
    return (
      <Fragment>
        <style jsx>{`
          h1 {
            color: green;
          }
        `}</style>
        <h1>秒表</h1>
        <MajorClock
          milliseconds={this.state.currentTime - this.state.startTime}
          activated={this.state.isStarted}
        />
   ...
```

界面效果如下，可以看到，StopWatch 中的 `h1` 字体不是 monospace，MajorClock 中的 color 也不是绿色。



可见，styled jsx 中虽然使用了 CSS，但是这些 CSS 规则只作用于所在组件中的样式，甚至不会影响子组件的样式。

这样一来，我们既可以使用 CSS 的语法，又可以把 CSS 的作用域限定在一个组件之内，达到了高内聚的要求。

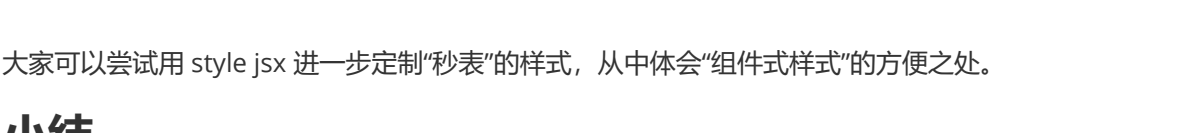
动态 styled jsx

更妙的是，我们还可以动态修改 styled jsx 中的值，因为 styled jsx 的内容就是字符串，我们只要修改其中的字符串，就修改了样式效果。

比如，我们让 MajorClock 在开始计时状态显示红色，否则显示黑色，修改代码如下：

```
const MajorClock = ({milliseconds=0, activated=false}) => {
  return (
    <React.Fragment>
      <style jsx>{`
        h1 {
          color: ${activated? 'red' : 'black'};
          font-family: monospace;
        }
      `}</style>
      <h1>
        {ms2Time(milliseconds)}
      </h1>
    </React.Fragment>
  );
};
```

在 style jsx 中，`color` 后面的值不是固定的，利用 ES6 的字符串模板功能，我们可以根据 activated 的值动态决定是 red 还是 black，最终效果图如下：



大家可以尝试用 style jsx 进一步定制“秒表”的样式，从中体会“组件式样式”的方便之处。

小结

本小节介绍了 React 中的样式实现方法，读者应该掌握：

1. React 将内容、样式和动态功能聚集在一个模块中，是高聚合的表现；
2. React 原生 style 属性的用法；
3. 组件化样式 styled jsx 的用法。

组件设计模式（1）：聪明组件和傻瓜组件

从这一节开始，我们来介绍 React 中的模式。

在 React 应用中，最简单也是最常用的一种组件模式，就是“聪明组件和傻瓜组件”。

其实，这个模式的名称很多，就我所知，除了“聪明组件和傻瓜组件”，还有这些称呼：

1. 容器组件和展示组件（Container and Presentational Components）；
2. 胖组件和瘦组件；
3. 有状态组件和无状态组件。

名字只是一个代号，关键还是要看本质，这种模式的本质，就是把一个功能分配到两个组件中，形成父子关系，外层的父组件负责管理数据状态，内层的子组件只负责展示。

在本小册中，都会以“聪明组件”和“傻瓜组件”称呼这种模式。

为什么要分割聪明组件和傻瓜组件

软件设计中有一个原则，叫做“责任分离”（Separation of Responsibility），简单说就是让一个模块的责任尽量少，如果发现一个模块功能过多，就应该拆分为多个模块，让一个模块都专注于一个功能，这样更利于代码的维护。

还记得我么说过 React 其实就是这样一个公式吗？

```
UI = f(data)
```

使用 React 来做界面，无外乎就是获得驱动界面的数据，然后利用这些数据来渲染界面。当然，你可以在一个组件中就搞定，但是，最好把获取和管理数据这件事和界面渲染这件事分开。做法就是，把获取和管理数据的逻辑放在父组件，也就是聪明组件；把渲染界面的逻辑放在子组件，也就是傻瓜组件。

这么做的好处，是可以灵活地修改数据状态管理方式，比如，最初你可能用 Redux 来管理数据，然后你想要修改为用 Mobx，如果按照这种模式分割组件，那么，你需要改的只有聪明组件，傻瓜组件可以保持原状。

随机笑话样例

我们利用一个显示“随机笑话”的功能来演示这种模式，所谓“随机笑话”，就是需要从服务器获取随机的一个笑话，展示在页面上。

功能可以分为两部分，第一部分是展示，也就是傻瓜组件，代码如下：

```
import SmileFace from './yaoming_simile.png';

const Joke = ({value}) => {
  return (
    <div>
      <img src={SmileFace} />
      {value || 'loading...' }
    </div>
  );
}
```

傻瓜组件 `Joke` 的功能很简单，显示一个笑脸，然后显示名为 value 的 props，也就是笑话的内容，如果没有 value 值，就显示一个“loading...”。

至于怎么获得笑话内容，不是 Joke 要操心的事，它只专注于显示笑话，所谓傻人有傻福，傻瓜组件虽然“傻”了一点，但是免去了数据管理的烦恼。

然后是聪明组件，这个组件不用管渲染的逻辑，只负责拿到数据，然后把数据传递给傻瓜组件，由傻瓜组件来完成渲染。

我们把聪明组件命名为 `RandomJoke`，代码如下：

```
export default class RandomJoke extends React.Component {
  state = {
    joke: null
  }

  render() {
    return <Joke value={this.state.joke} />
  }

  componentDidMount() {
    fetch('https://icanhazdadjoke.com/',
      {headers: {'Accept': 'application/json'}}
    ).then(response => {
      return response.json();
    }).then(json => {
      this.setState({joke: json.joke});
    });
  }
}
```

可以看到，RandomJoke 的 `render` 函数只做一件事，就是渲染 Joke，并把 `this.state` 中的值作为 props 传进去。聪明组件的 `render` 函数一般都这样简单，因为渲染不是他们操心的业务，他们的主业是获取数据。

RandomJoke 获取数据的方法是在 `componentDidMount` 函数中调用一个 API (https://icanhazdadjoke.com/)，这个 API 随即返回一个英文笑话。实话说，这些英文笑话都很冷，也不大容易看懂，但是足够展示通过 API 获取数据的过程。

当 RandomJoke 被第一次渲染的时候，它的 state 中的 joke 值为 `null`，所以它传给 Joke 的 value 也是 `null`，这时候，Joke 会渲染一 “loading...”。但是，在第一次渲染完毕的时候，`componentDidMount` 被调用，一个 API 请求发出去，拿到一个随机笑话，更新 state 中的 joke 值。因为对一个组件 state 的更新会引发一个新的渲染过程，所以 RandomJoke 的 `render` 再一次被调用，

所以 Joke 也会再一次被渲染，这一次，传入的 value 值是一个真正的笑话，所以，笑话也就出现了。

最终界面类似这个样子：

当然，这个界面非常简陋，获取数据的方法也十分粗糙，但是，最妙的是，应用了这种方法之后，如果你要优化界面，只需要去修改傻瓜组件 Joke，如果你想改进数据管理和获取，只需要去修改聪明组件 RandomJoke。

如此一来，维护工作就简单多了，你甚至可以把两个组件分配各两个不同的开发者去维护开发。

如果应用 Redux 和 Mobx，也会应用到这种模式，我们在后面的小节中会详细介绍更多这种模式的应用。

PureComponent

因为傻瓜组件一般没有自己的状态，所以，可以像上面的 Joke 一样实现为函数形式，其实，我们可以进一步改进，利用 `PureComponent` 来提高傻瓜组件的性能。

函数形式的 React 组件，好处是不需要管理 state，占用资源少，但是，函数形式的组件无法利用 `shouldComponentUpdate`。

看上面的例子，当 RandomJoke 要渲染 Joke 时，即使传入的 props 是一模一样的，Joke 也要走一遍完整的渲染过程，这就显得浪费了。

好一点的方法，是把 Joke 实现为一个类，而且定义 shouldComponentUpdate 函数，每次渲染过程中，在 render 函数执行之前 shouldComponentUpdate 会被调用，如果返回 `true`，那就继续，如果返回 `false`，那么渲染过程立刻停止，因为这代表不需要重画了。

对于傻瓜组件，因为逻辑很简单，界面完全由 props 决定，所以 shouldComponentUpdate 的实现方式就是比较这次渲染的 props 是否和上一次 props 相同。当然，让每一个组件都实现一遍这样简单的 shouldComponentUpdate 也很浪费，所以，React 提供了一个简单的实现工具 `PureComponent`，可以满足绝大部分需求。

改进后的 Joke 组件如下：

```
class Joke extends React.PureComponent {
  render() {
    return (
      <div>
        <img src={SmileFace} />
        {this.props.value || 'loading...' }
      </div>
    );
  }
}
```

值得一提的是，`PureComponent` 中 `shouldComponentUpdate` 对 props 做得只是浅层比较，不是深层比较，如果 props 是一个深层对象，就容易产生问题。

比如，两次渲染传入的某个 props 都是同一个对象，但是对象中某个属性的值不同，这在 PureComponent 眼里，props 没有变化，不会重新渲染，但是这明显不是我们想要的结果。

React.memo

虽然 PureComponent 可以提高组件渲染性能，但是它也不是没有代价的，它逼迫我们必须把组件实现为 class，不能用纯函数来实现组件。

如果你使用 React v16.6.0 之后的版本，可以使用一个新功能 `React.memo` 来完美实现 React 组件，上面的 Joke 组件可以这么写：

```
const Joke = React.memo(() => (
    <div>
        <img src={SmileFace} />
        {this.props.value || 'loading...' }
    </div>
));
```

React.memo 既利用了 shouldComponentUpdate，又不要求我们写一个 class，这也体现出 React 逐步向完全函数式编程前进。

小结

在这一小节中，我们介绍了“聪明组件和傻瓜组件”这种做法简单的 React 设计模式，读者应该能够理解为什么有时候要把组件分为两个部分，在众多框架中，都可以应用“聪明组件和傻瓜组件”模式。

组件设计模式（2）：高阶组件

在开发 React 组件过程中，很容易发现这样一种现象，某些功能是多个组件通用的，如果每个组件都重复实现这样的逻辑，肯定十分浪费，而且违反了“不要重复自己”（DRY，Don't Repeat Yourself)的编码原则，我们肯定想要把这部分共用逻辑提取出来重用。

我们说过，在 React 的世界里，组件是第一公民，首先想到的是当然是把共用逻辑提取为一个 React 组件。不过，有些情况下，这些共用逻辑还没法成为一个独立组件，换句话说，这些共用逻辑单独无法使用，它们只是对其他组件的功能加强。

举个例子，对于很多网站应用，有些模块都需要在用户已经登录的情况下才显示。比如，对于一个电商类网站，“退出登录”按钮、“购物车”这些模块，就只有用户登录之后才显示，对应这些模块的 React 组件如果连“只有在登录时才显示”的功能都重复实现，那就浪费了。

这时候，我们就可以利用“高阶组件（HoC）”这种模式来解决问题。

高阶组件的基本形式

“高阶组件”名为“组件”，其实并不是一个组件，而是一个函数，只不过这个函数比较特殊，它接受至少一个 React 组件为参数，并且能够返回一个全新的 React 组件作为结果，当然，这个新产生的 React 组件是对作为参数的组件的包装，所以，有机会赋予新组件一些增强的“神力”。

一个最简单的高阶组件是这样的形式：

```
const withDoNothing = (Component) => {
  const NewComponent = (props) => {
    return <Component {...props} />;
  };
  return NewComponent;
};
```

上面的函数 `withDoNothing` 就是一个高阶组件，作为一项业界通用的代码规范，高阶组件的命名一般都带 `with` 前缀，命名中后面的部分代表这个高阶组件的功能。

就如同 withDoNothing 这个名字所说的一样，这个高阶组件什么都没做，但是从中可以看出高阶组件的基本代码套路。

1. 高阶组件不能去修改作为参数的组件，高阶组件必须是一个纯函数，不应该有任何副作用。
2. 高阶组件返回的结果必须是一个新的 React 组件，这个新的组件的 JSX 部分肯定会包含作为参数的组件。
3. 高阶组件一般需要把传给自己的 props 转手传递给作为参数的组件。

用高阶组件抽取共同逻辑

接下来，我们对 withDoNothing 进行一些改进，让它实现“只有在登录时才显示”这个功能。

假设我们已经有一个函数 `getUserId` 能够从 cookies 中读取登录用户的 ID，如果用户未登录，这个 `getUserId` 就返回空，那么“退出登录按钮”就需要这么写：

```
const LogoutButton = () => {
  if (getUserId()) {
    return ...; // 显示”退出登录“的JSX
  } else {
    return null;
  }
};
```

同样，购物车的代码就是这样：

```
const ShoppintCart = () => {
  if (getUserId()) {
    return ...; // 显示”购物车“的JSX
  } else {
    return null;
  }
};
```

上面两个组件明显有重复的代码，我们可以把重复代码抽取出来，形成 `withLogin` 这个高阶组件，代码如下：

```
const withLogin = (Component) => {
  const NewComponent = (props) => {
    if (getUserId()) {
      return <Component {...props} />;
    } else {
      return null;
    }
  }

  return NewComponent;
};
```

如此一来，我们就只需要这样定义 `LogoutButton` 和 `ShoppintCart`：

```
const LogoutButton = withLogin((props) => {
  return ...; // 显示”退出登录“的JSX
});

const ShoppingCart = withLogin(() => {
  return ...; // 显示”购物车“的JSX
});
```

你看，我们避免了重复代码，以后如果要修改对用户是否登录的判断逻辑，也只需要修改 withLogin，而不用修改每个 React 组件。

高阶组件的高级用法

高阶组件只需要返回一个 React 组件即可，没人规定高阶组件只能接受一个 React 组件作为参数，完全可以传入多个 React 组件给高阶组件。

比如，我们可以改进上面的 withLogin，让它接受两个 React 组件，根据用户是否登录选择渲染合适的组件。

```
const withLoginAndLogout = (ComponentForLogin, ComponentForLogout) => {
  const NewComponent = (props) => {
    if (getUserId()) {
      return <ComponentForLogin {...props} />;
    } else {
      return <ComponentForLogout{...props} />;
    }
  }
  return NewComponent;
};
```

有了上面的 `withLoginAndLogout`，就可以产生根据用户登录状态显示不同的内容。

```
const TopButtons = withLoginAndLogout(
  LogoutButton,
  LoginButton
);
```

链式调用高阶组件

高阶组件最巧妙的一点，是可以链式调用。

假设，你有三个高阶组件分别是 `withOne`、`withTwo` 和 `withThree`，那么，如果要赋予一个组件 X 某个高阶组件的超能力，那么，你要做的就是挨个使用高阶组件包装，代码如下：

```
const X1 = withOne(X);
const X2 = withTwo(X1);
const X3 = withThree(X2);
const SuperX = X3; //最终的SuperX具备三个高阶组件的超能力
```

很自然，我们可以避免使用中间变量 `X1` 和 `X2`，直接连续调用高阶组件，如下：

```
const SuperX = withThree(withTwo(withOne(X)));
```

对于 `X` 而言，它被高阶组件包装了，至于被一个高阶组件包装，还是被 N 个高阶组件包装，没有什么差别。而高阶组件本身就是一个纯函数，纯函数是可以组合使用的，所以，我们其实可以把多个高阶组件组合为一个高阶组件，然后用这一个高阶组件去包装`X`，代码如下：

```
const hoc = compose(withThree, withTwo, withOne);
const SuperX = hoc(X);
```

在上面代码中使用的 `compose`，是函数式编程中很基础的一种方法，作用就是把多个函数组合为一个函数，在很多开源的代码库中都可以看到，下面是一个参考实现：

```
export default function compose(...funcs) {
  if (funcs.length === 0) {
    return arg => arg
  }

  if (funcs.length === 1) {
    return funcs[0]
  }

  return funcs.reduce((a, b) => (...args) => a(b(...args)))
}
```

React 组件可以当做积木一样组合使用，现在有了 compose，我们就可以把高阶组件也当做积木一样组合，进一步重用代码。

假如一个应用中多个组件都需要同样的多个高阶组件包装，那就可以用 compose 组合这些高阶组件为一个高阶组件，这样在使用多个高阶组件的地方实际上就只需要使用一个高阶组件了。

不要滥用高阶组件

高阶组件虽然可以用一种可重用的方式扩充现有 React 组件的功能，但高阶组件并不是绝对完美的。

首先，高阶组件不得不处理 `displayName`，不然 debug 会很痛苦。当 React 渲染出错的时候，靠组件的 displayName 静态属性来判断出错的组件类，而高阶组件总是创造一个新的 React 组件类，所以，每个高阶组件都需要处理一下 displayName。

如果要做一个最简单的什么增强功能都没有的高阶组件，也必须要写下面这样的代码：

```
const withExample = (Component) => {
  const NewComponent = (props) => {
    return <Component {...props} />;
  }

  NewComponent.displayName = `withExample(${Component.displayName ||
Component.name || 'Component'})`;

  return NewCompoennt;
};
```

每个高阶组件都这么写，就会非常的麻烦。

对于 React 生命周期函数，高阶组件不用怎么特殊处理，但是，如果内层组件包含定制的静态函数，这些静态函数的调用在 React 生命周期之外，那么高阶组件就必须要在新产生的组件中增加这些静态函数的支持，这更加麻烦。

其次，高阶组件支持嵌套调用，这是它的优势。但是如果真的一大长串高阶组件被应用的话，当组件出错，你看到的会是一个超深的 stack trace，十分痛苦。

最后，使用高阶组件，一定要非常小心，要避免重复产生 React 组件，比如，下面的代码是有问题的：

```
const Example = () => {
  const EnhancedFoo = withExample(Foo);
  return <EnhancedFoo />
}
```

像上面这样写，每一次渲染 Example，都会用高阶组件产生一个新的组件，虽然都叫做 `EnhancedFoo`，但是对 React 来说是一个全新的东西，在重新渲染的时候不会重用之前的虚拟 DOM，会造成极大的浪费。

正确的写法是下面这样，自始至终只有一个 EnhancedFoo 组件类被创建：

```
const EnhancedFoo = withExample(Foo);

const Example = () => {
  return <EnhancedFoo />
}
```

总之，高阶组件是重用代码的一种方式，但并不是唯一方式，在下一小节，我们会介绍一种更加精妙的方式。

小结

在这一小节中，我们介绍了高阶组件（HoC）这种重用逻辑的模式，读者应该能够学会：

1. 高阶组件的形式；
2. 高阶组件的链式调用方法；
3. 高阶组件的不足。

组件设计模式（3）：render props 模式

在上一小节中，我们介绍了高阶组件，高阶组件并不是 React 中唯一的重用组件逻辑的方式，在这一小节中，我们会介绍另一种方式 render props。

render props

所谓 render props，指的是让 React 组件的 props 支持函数这种模式。因为作为 props 传入的函数往往被用来渲染一部分界面，所以这种模式被称为 render props。

一个最简单的 render props 组件 `RenderAll`，代码如下：

```
const RenderAll = (props) => {
  return(
     <React.Fragment>
        {props.children(props)}
     </React.Fragment>
  );
};
```

这个 `RenderAll` 预期子组件是一个函数，它所做的事情就是把子组件当做函数调用，调用参数就是传入的 props，然后把返回结果渲染出来，除此之外什么事情都没有做。

使用 RenderAll 的代码如下：

```
      <RenderAll>
        {() => <h1>hello world</h1>}
      </RenderAll>
```

可以看到，RenderAll 的子组件，也就是夹在 RenderAll 标签之间的部分，其实是一个函数。这个函数渲染出 `<h1>hello world</h1>`，这就是上面使用 RenderAll 渲染出来的结果。

当然，这个 RenderAll 没做任何实际工作，接下来我们看 render props 真正强悍的使用方法。

传递 props

和高阶组件一样，render props 可以做很多的定制功能，我们还是以根据是否登录状态来显示一些界面元素为例，来实现一个 render props。

下面是实现 render props 的 `Login` 组件，可以看到，render props 和高阶组件的第一个区别，就是 render props 是真正的 React 组件，而不是一个返回 React 组件的函数。

```
const Login = (props) => {
  const userName = getUserName();

  if (userName) {
    const allProps = {userName, ...props};
    return (
      <React.Fragment>
```

```
        {props.children(allProps)}
      </React.Fragment>
    );
  } else {
    return null;
  }
};
```

当用户处于登录状态，`getUserName` 返回当前用户名，否则返回空，然后我们根据这个结果决定是否渲染 `props.children` 返回的结果。

当然，render props 完全可以决定哪些 props 可以传递给 props.children，在 Login 中，我们把 `userName` 作为增加的 props 传递给下去，这样就是 Login 的增强功能。

一个使用上面 Login 的 JSX 代码示例如下：

```
  <Login>
    {({userName}) => <h1>Hello {userName}</h1>}
  </Login>
```

对于名为“程墨Morgan”的用户登录，上面的 JSX 会产生 `<h1>Hello 程墨Morgan</h1>`。

不局限于 children

在上面的例子中，作为 render 方法的 props 就是 `children`，在我写的[《深入浅出React和Redux》]中，将这种模式称为“以函数为子组件（function as child）”的模式，这可以算是 render props 的一种具体形式，也就利用 `children` 这个 props 来作为函数传递。

实际上，render props 这个模式不必局限于 children 这一个 props，任何一个 props 都可以作为函数，也可以利用多个 props 来作为函数。

我们来扩展 Login，不光在用户登录时显示一些东西，也可以定制用户没有登录时显示的东西，我们把这个组件叫做 `Auth`，对应代码如下：

```
const Auth= (props) => {
  const userName = getUserName();

  if (userName) {
    const allProps = {userName, ...props};
    return (
      <React.Fragment>
        {props.login(allProps)}
      </React.Fragment>
    );
  } else {
    <React.Fragment>
      {props.nologin(props)}
    </React.Fragment>
  }
};
```

使用 Auth 的话，可以分别通过 `login` 和 `nologin` 两个 props 来指定用户登录或者没登录时显示什么，用法如下：

```
  <Auth
    login={({userName}) => <h1>Hello {userName}</h1>}
    nologin={() => <h1>Please login</h1>}
  />
```

依赖注入

render props 其实就是 React 世界中的“依赖注入”（Dependency Injection)。

所谓依赖注入，指的是解决这样一个问题：逻辑 A 依赖于逻辑 B，如果让 A 直接依赖于 B，当然可行，但是 A 就没法做得通用了。依赖注入就是把 B 的逻辑以函数形式传递给 A，A 和 B 之间只需要对这个函数接口达成一致就行，如此一来，再来一个逻辑 C，也可以用一样的方法重用逻辑 A。

在上面的代码示例中，`Login` 和 `Auth` 组件就是上面所说的逻辑 A，而传递给组件的函数类型 props，就是逻辑 B 和 C。

render props 和高阶组件的比较

我们来比对一下这两种重用 React 组件逻辑的模式。

首先，render props 模式的应用，就是做一个 React 组件，而高阶组件，虽然名为“组件”，其实只是一个产生 React 组件的函数。

render props 不像上一小节中介绍的高阶组件有那么多毛病，如果说 render props 有什么缺点，那就是 render props 不能像高阶组件那样链式调用，当然，这并不是一个致命缺点。

render props 相对于高阶组件还有一个显著优势，就是对于新增的 props 更加灵活。还是以登录状态为例，假如我们扩展 withLogin 的功能，让它给被包裹的组件传递用户名这个 props，代码如下：

```
const withLogin = (Component) => {
  const NewComponent = (props) => {
    const userName= getUserName();
    if (userName) {
      return <Component {...props} userName={userName}/>;
    } else {
      return null;
    }
  }

  return NewComponent;
};
```

这就要求被 withLogin 包住的组件要接受 `userName` 这个props。可是，假如有一个现成的 React 组件不接受 userName，却接受名为 `name` 的 props 作为用户名，这就麻烦了。我们就不能直接用 withLogin 包住这个 React 组件，还要再造一个组件来做 `userName` 到 `name` 的映射，十分费事。

对于应用 render props 的 Login，就不存在这个问题，接受 `name` 不接受 `userName` 是吗？这样写就好了：

```
<Login>
  {
    (props) => {
      const {userName} = props;
      return <TheComponent {...props} name={userName} />
    }
  }
</Login>
```

所以，当需要重用 React 组件的逻辑时，建议首先看这个功能是否可以抽象为一个简单的组件；如果行不通的话，考虑是否可以应用 render props 模式；再不行的话，才考虑应用高阶组件模式。

这并不表示高阶组件无用武之地，在后续章节，我们会对 render props 和高阶组件分别讲解具体的实例。

小结

在这一小节中，我们介绍了 render props 这种模式，也将 render props 和高阶组件两种模式进行了比较。

读者应该要明白:

1. render props 的形式；
2. render props 其实就是“依赖注入”；
3. 如何利用 render props 实现共享组件之间的逻辑。

组件设计模式（4）：提供者模式

这一节我们来介绍 React 中的“提供者模式”（Provider Pattern）。

问题场景

在 React 中，props 是组件之间通讯的主要手段，但是，有一种场景单纯靠 props 来通讯是不恰当的，那就是两个组件之间间隔着多层其他组件，下面是一个简单的组件树示例图图：

在上图中，组件 A 需要传递信息给组件 X，如果通过 props 的话，那么从顶部的组件 A 开始，要把 props 传递给组件 B，然后组件 B 传递给组件 D，最后组件 D 再传递给组件 X。

其实组件 B 和组件 D 完全用不上这些 props，但是又被迫传递这些 props，这明显不合理，要知道组件树的结构会变化的，将来如果组件 B 和组件 D 之间再插入一层新的组件，这个组件也需要传递这个 props，这就麻烦无比。

可见，对于跨级的信息传递，我们需要一个更好的方法。

在 React 中，解决这个问题应用的就是“提供者模式”。

提供者模式

虽然这个模式叫做“提供者模式”，但是其实有两个角色，一个叫“提供者”（Provider），另一个叫“消费者”（Consumer），这两个角色都是 React 组件。其中“提供者”在组件树上居于比较靠上的位置，“消费者”处于靠下的位置。在上面的组件树中，组件 A 可以作为提供者，组件 X 就是消费者。

既然名为“提供者”，它可以提供一些信息，而且这些信息在它之下的所有组件，无论隔了多少层，都可以直接访问到，而不需要通过 props 层层传递。

避免 props 逐级传递，即是提供者的用途。

如何实现提供者模式

实现提供者模式，需要 React 的 Context 功能，可以说，提供者模式只不过是让 Context 功能更好用一些而已。

所谓 Context 功能，就是能够创造一个“上下文”，在这个上下文笼罩之下的所有组件都可以访问同样的数据。

在 React v16.3.0 之前，React 虽然提供了 Context 功能，但是官方文档上都建议尽量不要使用，因为对应的 API 他们并不满意，觉得迟早要废弃掉。即使如此，依然有很多库和应用使用 Context 功能，可见对这个需求的呼声有多大。

当 React 发布 v16.3.0 时，终于提供了“正式版本”的 Context 功能 API，和之前的有很大不同，当然，这也带来一些问题，我在后面会介绍。

提供者模式的一个典型用例就是实现“样式主题”（Theme），由顶层的提供者确定一个主题，下面的样式就可以直接使用对应主题里的样式。这样，当需要切换样式时，只需要修改提供者就行，其他组件不用修改。

为了方便比对，这里我会介绍提供者模式用不同 Context API 的实现方法。不过，你如果完全不在意老版本 React 如何实现的，可以略过下面一段。

React v16.3.0 之前的提供者模式

在 React v16.3.0 之前，要实现提供者，就要实现一个 React 组件，不过这个组件要做两个特殊处理。

1. 需要实现 `getChildContext` 方法，用于返回“上下文”的数据；
2. 需要定义 `childContextTypes` 属性，声明“上下文”的结构。

下面就是一个实现“提供者”的例子，组件名为 `ThemeProvider`：

```
class ThemeProvider extends React.Component {
  getChildContext() {
    return {
      theme: this.props.value
    };
  }

  render() {
    return (
      <React.Fragment>
        {this.props.children}
      </React.Fragment>
    );
  }
}

ThemeProvider.childContextTypes = {
  theme: PropTypes.object
};
```

在上面的例子中，getChildContext 只是简单返回名为 `value` 的 props 值，但是，因为 getChildContext 是一个函数，它可以有更加复杂的操作，比如可以从 state 或者其他数据源获得数据。

对于 ThemeProvider，我们创造了一个上下文，这个上下文就是一个对象，结构是这样：

```
{
  theme: {
    //一个对象
  }
}
```

接下来，我们来做两个消费（也就是使用）这个“上下文”的组件，第一个是 `Subject`，代表标题；第二个是 `Paragraph`，代表章节。

我们把 Subject 实现为一个类，代码如下：

```
class Subject extends React.Component {
  render() {
    const {mainColor} = this.context.theme;
    return (
      <h1 style={{color: mainColor}}>
        {this.props.children}
      </h1>
    );
```

```
  }
}

Subject.contextTypes = {
  theme: PropTypes.object
}
```

在 Subject 的 `render` 函数中，可以通过 `this.context` 访问到“上下文”数据，因为 ThemeProvider 提供的“上下文”包含 `theme` 字段，所以可以直接访问 `this.context.theme`。

千万不要忘了 Subject 必须增加 `contextTypes` 属性，必须和 ThemeProvider 的 `childContextTypes` 属性一致，不然，`this.context` 就不会得到任何值。

读者可能会问了，为什么这么麻烦呢？为什么要求“提供者”用 `childContextTypes` 定义一次上下文结构，又要求“消费者”再用 `contextTypes` 再重复定义一次呢？这不是很浪费吗？

React 这么要求，是考虑到“上下文”可能会嵌套，就是一个“提供者”套着另一个“提供者”，这时候，底层的消费者组件到底消费哪一个“提供者”呢？通过这种显示的方式指定。

不过，实话实说，这样的 API 设计的确麻烦了一点，难怪 React 官方在最初就不建议使用。

上面的 Subject 是一个类，其实也可以把消费者实现为一个纯函数组件，只不过访问“上下文”的方式有些不同，我们用纯函数的方式实现另一个消费者 `Paragraph`，代码如下：

```
const Paragraph = (props, context) => {
  const {textColor} = context.theme;
  return (
    <p style={{color: textColor}}>
      {props.children}
    </p>
  );
};

Paragraph.contextTypes = {
  theme: PropTypes.object
};
```

从上面的代码可以看到，因为 Paragraph 是一个函数形式，所以不可能访问 `this.context`，但是函数的第二个参数其实就是 `context`。

当然，也不要忘了设定 Paragraph 的 `contextTypes`，不然参数 `context` 也不会是上下文。

最后，我们看如何结合”提供者“和”消费者“。

我们做一个组件来使用 Subject 和 Paragraph，这个组件不需要帮助传递任何 props，代码如下：

```
const Page = () => (
  <div>
    <Subject>这是标题</Subject>
    <Paragraph>
      这是正文
    </Paragraph>
  </div>
);
```

上面的组件 `Page` 使用了 Subject 和 Paragraph，现在我们想要定制样式主题，只需要在 Page 或者任何需要应用这个主题的组件外面包上 ThemeProvider，对应的 JSX 代码如下：

```
  <ThemeProvider value={{mainColor: 'green', textColor: 'red'}} >
    <Page />
  </ThemeProvider>
```

最后，看到的效果如下：

当我们需要改变一个样式主题的时候，改变传给 ThemeProvider的 value 值就搞定了。

React v16.3.0 之后的提供者模式

到了 React v16.3.0 的时候，新的 Context API 出来了，这套 API 毫不掩饰自己就是“提供者模式”的实现，命名上就带 “Provider” 和 “Consumer”。

还是上面的样式主题的例子，首先，要用新提供的 `createContext` 函数创造一个“上下文”对象。

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

这个“上下文”对象 `ThemeContext` 有两个属性，分别就是一一对，你没猜错——`Provider` 和 `Consumer`。

```
const ThemeProvider = ThemeContext.Provider;
const ThemeConsumer = ThemeContext.Consumer;
```

创造“提供者”极大简化了，都不需要我们创造一个 React 组件类。

使用“消费者”也同样简单，而且应用了上一节我们介绍的 render props 模式，比如，Subject 的代码如下:

```
class Subject extends React.Component {
  render() {
    return (
      <ThemeConsumer>
        {
          (theme) => (
            <h1 style={{color: theme.mainColor}}>
              {this.props.children}
            </h1>
          )
        }
      </ThemeConsumer>
    );
  }
}
```

上面的 `ThemeConsumer` 其实就是一个应用了 render props 模式的组件，它要求子组件是一个函数，会把“上下文”的数据作为参数传递给这个函数，而这个函数里就可以通过参数访问“上下文”对象。

在新的 API 里，不需要设定组件的 `childContextTypes` 或者 `contextTypes` 属性，这省了不少事。

可以注意到，Subject 没有自己的状态，没必要实现为类，我们用纯函数的形式实现 `Paragraph`，代码如下：

```
const Paragraph = (props, context) => {
  return (
    <ThemeConsumer>
      {
        (theme) => (
          <p style={{color: theme.textColor}}>
            {props.children}
          </p>
          )
      }
    </ThemeConsumer>
  );
};
```

实现 `Page` 的方式并没有变化，而应用 `ThemeProvider` 的代码和之前也完全一样:

```
  <ThemeProvider value={{mainColor: 'green', textColor: 'red'}} >
    <Page />
  </ThemeProvider>
```

两种提供者模式实现方式的比较

通过上面的代码，可以很清楚地看到，新的 Context API 更简洁，但是，也并不是十全十美。

在老版 Context API 中，“上下文”只是一个概念，并不对应一个代码，两个组件之间达成一个协议，就诞生了“上下文”。

在新版 Context API 中，需要一个“上下文”对象（上面的例子中就是 `ThemeContext`），使用“提供者”的代码和“消费者”的代码往往分布在不同的代码文件中，那么，这个 `ThemeContext` 对象放在哪个代码文件中呢?

最好是放在一个独立的文件中，这么一来，就多出一个代码文件，而且所有和这个“上下文”相关的代码，都要依赖于这个“上下文”代码文件，虽然这没什么大不了的，但是的确多了一层依赖关系。

为了避免依赖关系复杂，每个应用都不要滥用“上下文”，应该限制“上下文”的使用个数。

不管怎么说，新版本的 Context API 才是未来，在 React v17 中，可能就会删除对老版 Context API 的支持，所以，现在大家都应该使用第二种实现方式。

小结

这一小节我们介绍了“提供者模式”，读者应该能够理解：

1. 提供者模式解决的问题；
2. React 的 Context 功能对这种模式有很直接的支持；
3. 提供者模式中 render props 的应用。

在接下来关于 Redux 和 Mobx 的介绍中，可以看到“提供者模式”更广泛的应用。

组件设计模式（5）：组合组件

这一小节我们介绍“组合组件”（Compound Component）这种模式。网上介绍这种模式的文章也不少，但是都是一上来直接讲实现细节，却很少说应用于什么场景，所以往往都讲得让读者云里雾里。

这里我要强调，所谓模式，就是特定于一种问题场景的解决办法。

```
模式(Pattern) = 问题场景(Context) + 解决办法(Solution)
```

如果不搞清楚场景，单纯知道有这么一个办法，就好比拿到了一杆枪却不知道这杆枪用于打什么目标，是没有任何意义的。并不是所有的枪都是一样的，有的枪擅长狙击，有的枪适合近战，有的枪只是发个信号。

模式就是我们的武器，我们一定要搞清楚一件武器应用的场合，才能真正发挥这件武器的威力。

组合组件模式要解决的是这样一类问题：父组件想要传递一些信息给子组件，但是，如果用 props 传递又显得十分麻烦。

一看到这个问题描述，读者应该能立刻想到上一节我们介绍过的 Context API，利用 Context，可以让组件之间不用 props 来传递信息。

不过，使用 Context 也不是完美解法，上一节我们介绍过，使用 React 在 v16.3.0 之后提供的新的 Context API，需要让“提供者”和“消费者”共同依赖于一个 Context 对象，而且消费者也要使用 render props 模式。

如果不嫌麻烦，用 Context 来解决问题当然好，但是我们肯定会想有没有更简洁的方式。

问题描述

为了让问题更加具体，我们来解决一个实例。

很多界面都有 Tab 这样的元件，我们需要一个 `Tabs` 组件和 `TabItem` 组件，Tabs 是容器，TabItem 是一个一个单独的 Tab，因为一个时刻只有一个 TabItem 被选中，很自然希望被选中的 TabItem 样式会和其他 TabItem 不同。

这并不是一个很难的功能，首先我们想到的就是，用 Tabs 中一个 state 记录当前被选中的 Tabitem 序号，然后根据这个 state 传递 props 给 TabItem，当然，还要传递一个 `onClick` 事件进去，捕获点击选择事件。

按照这样的设计，Tabs 中如果要显示 One、Two、Three 三个 TabItem，JSX 代码大致这么写：

```
      <TabItem active={true} onClick={this.onClick}>One</TabItem>
      <TabItem active={false} onClick={this.onClick}>Two</TabItem>
      <TabItem active={false} onClick={this.onClick}>Three</TabItem>
```

上面的 TabItem 组件接受 `active` 这个 props，如果 `true` 代表当前是选中状态，当然可以工作，但是，也存在大问题：

1. 每次使用 TabItem 都要传递一堆 props，好麻烦；
2. 每增加一个新的 TabItem，都要增加对应的 props，更麻烦；
3. 如果要增加 TabItem，就要去修改 Tabs 的 JSX 代码，超麻烦。

我们不想要这么麻烦，理想情况下，我们希望可以随意增加减少 TabItem 实例，不用传递一堆 props，也不用去修改 Tabs 的代码，最好代码就这样：

```
    <Tabs>
      <TabItem>One</TabItem>
      <TabItem>Two</TabItem>
      <TabItem>Three</TabItem>
    </Tabs>
```

如果能像上面一样写代码，那就达到目的了。

像上面这样，Tabs 和 TabItem 不通过表面的 props 传递也能心有灵犀，二者之间有某种神秘的“组合”，就是我们所说的“组合组件”。

实现方式

上面我们说过，利用 Context API，可以实现组合组件，但是那样 TabItem 需要应用 render props，至于如何实现，读者可以参照上一节的介绍自己尝试。

在这里，我们用一种更巧妙的方式来实现组合组件，可以避免 TabItem 的复杂化。

我们先写出 TabItem 的代码，如下：

```
const TabItem = (props) => {
  const {active, onClick} = props;
  const tabStyle = {
    'max-width': '150px',
    color: active ? 'red' : 'green',
    border: active ? '1px red solid' : '0px',
  };
  return (
    <h1 style={tabStyle} onClick={onClick}>
      {props.children}
    </h1>
  );
};
```

TabItem 有两个重要的 props：`active` 代表自己是否被激活，`onClick` 是自己被点击时应该调用的回调函数，这就足够了。TabItem 所做的就是根据这两个 props 渲染出 `props.children`，没有任何复杂逻辑，是一个活脱脱的“傻瓜组件”，所以，用一个纯函数实现就可以了。

接下来要做的，就看 Tabs 如何把 `active` 和 `onClick` 传递给 TabItem。

我们再来看一下使用组合组件的 JSX 代码：

```
    <Tabs>
      <TabItem>One</TabItem>
      <TabItem>Two</TabItem>
      <TabItem>Three</TabItem>
    </Tabs>
```

没有 props 的传递啊，怎么悄无声息地把 `active` 和 `onClick` 传递给 TabItem 呢？

Tabs 虽然可以访问到作为 props 的 `children`，但是到手的 `children` 已经是创造好的元素，而且是不可改变的，Tabs 是不可能把创造好的元素再强塞给 `children` 的。

怎么办?

办法还是有的，如果 Tabs 并不去渲染 `children`，而是把 `children` 拷贝一份，就有机会去篡改这份拷贝，最后渲染这份拷贝就好了。

我们来看 Tabs 的实现代码：

```
class Tabs extends React.Component {
  state = {
    activeIndex:  0
  }

  render() {
    const newChildren = React.Children.map(this.props.children, (child, index)
=> {
      if (child.type) {
        return React.cloneElement(child, {
          active: this.state.activeIndex === index,
          onClick: () => this.setState({activeIndex: index})
        });
      } else {
        return child;
      }
    });

    return (
      <Fragment>
        {newChildren}
      </Fragment>
    );
  }
}
```

在 render 函数中，我们用了 React 中不常用的两个 API：

1. React.Children.map
2. React.cloneElement

使用 `React.Children.map`，可以遍历 `children` 中所有的元素，因为 `children` 可能是一个数组嘛。

使用 `React.cloneElement` 可以复制某个元素。这个函数第一个参数就是被复制的元素，第二个参数可以增加新产生元素的 props，我们就是利用这个机会，把 `active` 和 `onClick` 添加了进去。

这两个 API 双剑合璧，就能实现不通过表面的 props 传递，完成两个组件的“组合”。

最终的效果如下：

点击任何一个 TabItem，其样式就会立刻改变。而维护哪个 TabItem 是当前选中的状态，则是 Tabs 的责任。

实际应用

从上面的代码可以看出来，对于组合组件这种实现方式，TabItem 非常简化；Tabs 稍微麻烦了一点，但是好处就是把复杂度都封装起来了，从使用者角度，连 props 都看不见。

所以，应用组合组件的往往是共享组件库，把一些常用的功能封装在组件里，让应用层直接用就行。在 antd 和 bootstrap 这样的共享库中，都使用了组合组件这种模式。

如果你的某两个组件并不需要重用，那么就要谨慎使用组合组件模式，毕竟这让代码复杂了一些。

小结

这一小节介绍了组合组件（Compound Component）这种模式，这是一种比较高级的模式，如果要开发需要关联的成对组件，可以采用这个方案。

React 单元测试

这一小节来讲 React 中的测试，虽然不涉及 React 的使用，但是却关系到我们开发的 React 的质量。

毫无疑问，对代码进行测试是最佳实践，可以保证代码质量，不过，对于软件代码质量毫不关心的读者可以略过这一节。

测试的目的

测试对于软件开发非常重要，简单来说，测试就是尽力发现软件中的缺陷（俗称 bug），当我们发现不了更多的 bug 时，说明这个软件质量可以接受了。

然而，没有 bug 的软件我还没见过呢。

在互联网时代，我们更是不可能等到所有 bug 都修复了才上线，那样黄花菜都凉了，稍微有一些工作经验的人都会有这样的体会。

所以，事实上，测试是**尽力**发现软件中的 bug。当我们发现 bug 数量和严重程度呈稳定的下降趋势，直到低于一个门槛（无须降低为 0，只需要降低到可接受的程度），没有更多更严重的 bug 出现，就说明这个软件的质量可以接受，可以上线了。

这样当然要比达到“零 bug 软件”要容易多了，但是，不要因此以为这就是一件没有困难的任务。为了让 bug 的数量和严重程度足够低，我们开发者必须严格要求自己，只有保证我们写的每一小块代码都经受住测试的考验，这些小块代码集合在一起的时候才可能（只是有可能）不会出很多 bug，如果我们写的小块代码质量都无法保证，那大项目的代码根本无法保证。

这一小节重点讲对”小块代码“的测试，也就是单元测试。

单元测试

单元测试的内容足够讲一本书了，所以，我只针对性地讲一讲 React 组件的单元测试，并且只有三个要点：

1. 用 Jest；
2. 用 Enzyme；
3. 保持 100% 的代码覆盖率。

Jest

在 JavaScript 的世界里，单元测试的框架很多，品牌最老名气最响的是 Mocha，不过，不要纠结于名气，请使用 Jest。你不会后悔的，接下来我告诉你为什么。

我们先假设，作为开发者，你是在团队中工作。所谓团队，就是有很多人一起工作，而且随着业务和团队的发展，人会越来越多，潜台词就是——不确定因素越来越多。

人和人之间交流会出现偏差，人的水平有高低之分，人也会犯错，总之，你不能指望所有人都把事情做得尽善尽美。

具体到单元测试这件事上来，“测试驱动”是开发喊了这么多年，为什么真正做到这一点的团队依然不多呢？因为，当团队变大之后，很多问题也就出现了。

1. **单元测试用例庞大，执行时间过长。**

想象一下，一个代码库里假设有一千个单元测试用例，即使每个单元测试用例平均只需要 10 毫秒，那总时间也就需要 10 秒钟。好，假设代码库进一步扩大，有了一万个单元测试用例，那就跑一遍就需要 100 秒，已经超过了一分钟。这还只是保守估计，实际上单元测试用例的运行时间只会比这长。开发者如果每次修改都需要等待这么漫长的单元测试运行时间，肯定会三心二意上网去看其他东西。

2. **单元测试用例之间相互影响。**

你可能也有这样的体验，代码库中的单元测试突然失败了，但是你修改的代码根本不会取影响失败的那个单元测试用例，怎么回事？这往往是因为某个成员以前的代码写得不好，影响了一个全局变量。当然，谁都知道单元测试应该在 setup 时创建环境，在 teardown 时恢复环境，可是，总会有人有马虎大意的情况，这时候你怎么办？要么你只好去修复一个本不是你改坏的代码，要么你干脆删掉那段不可靠的单元测试代码，不管怎样，这都会打击你支持"测试驱动开发"的决心。

Jest 较好地解决了上面说的问题，因为 Jest 最重要的一个特性，就是支持并行执行。

Mocha 之类老牌单元测试框架，把所有的单元测试都放在一个环境中执行，这就使所有单元测试访问的是同样一个全局变量空间，所以只要测试代码没写好，就会互相影响。而且，为了保证执行正常，所有的单元测试必须一个接一个地执行，这是体系架构决定的，没有办法。

Jest 不同，Jest 为每一个单元测试文件创造一个独立的运行环境，换句话说，Jest 会启动一个进程执行一个单元测试文件，运行结束之后，就把这个执行进程废弃了，这个单元测试文件即使写得比较差，把全局变量污染得一团糟，也不会影响其他单元测试文件，因为其他单元测试文件是用另一个进程来执行。

更妙的是，因为每个单元测试文件之间再无纠葛，Jest 可以启动多个进程同时运行不同的文件，这样就充分利用了电脑的多 CPU 多核，单进程 100 秒才完成的测试执行过程，8 核只需要 12.5 秒，速度快了很多。

Jest 还有很多其他友好的特性，大家可以自己去发掘，这里废话不多说，只想安利各位，**测试 React 或者 JavaScript 代码，用 Jest!**

使用 create-react-app 产生的项目自带 Jest 作为测试框架，不奇怪，因为 Jest 和 React 一样都是出自 Facebook。

运行下面的命令，就可以进入交互式的"测试驱动开发"模式：

```
npm test
```

Enzyme

虽然最好的 React 测试框架出自 Facebook 家，最受欢迎的 React 测试工具库却出自 Airbnb，这个工具库叫做 Enzyme。Enzyme 这个单词的含义是"酶"，至于命名原因已经无法考据，可能寓意着快速分解。

不过因为 Enzyme 不是 Facebook 家出品，所以使用 Enzyme 还真稍微有些麻烦——在 create-react-app 产生的应用中并不包含 Enzyme，需要我们自己来添加。

在项目目录下，通过下面的命令来安装 `enzyme`：

```
npm i --save-dev enzyme enzyme-adapter-react-16
```

可以注意到，我们不光要安装 enzyme，还要安装 `enzyme-adapter-react-16`，这个库是用来作为适配器的。因为不同 React 版本有各自特点，所用的适配器也会不同，我们的项目中使用的是 16.4 之后的版本，所以用 `enzyme-adapter-react-16`；如果用 16.3 版本，需要用 `enzyme-adapter-react-16.3`；如果用 16.2 版本，需要用 `enzyme-adapter-react-16.2`；如果用更老的版本 15.5，需要用

`enzyme-adapter-react-15`。具体各个 React 版本对应什么样的 Adapter，请参考 enzyme官方文档。

现在，可以在测试代码中使用 enzyme 了。我们以之前秒表应用中的 ControlButtons 组件为例，来说明如何做单元测试。

我们创造一个 `ControlButtons.test.js`，来容纳对应的测试用例，因为所有后缀为 `.test.js` 的文件都会被 Jest 认作是测试用例文件。

在代码中，需要使用 Adapter，代码如下：

```
import {configure} from 'enzyme';
import Adapter from 'enzyme-adapter-react-16';
configure({adapter: new Adapter()});
```

我们对 ControlButtons 组件的测试，就是要渲染它一次，看看渲染结果如何，enzyme 就能帮助我们做这件事。

比如，我们想要保证渲染出来的内容必须包含两个按钮，其中一个按钮的 class 名是 `left-btn`，另一个是 `right-btn`，那么我们就需要下面的单元测试用例：

```
import {shallow} from 'enzyme';

it('renders without crashing', () => {
  const wrapper = shallow(<ControlButtons />);
  expect(wrapper.find('.left-btn')).toHaveLength(1);
  expect(wrapper.find('.right-btn')).toHaveLength(1);
});
```

在这里我们使用了 `shallow`，其实也可以使用 `mount`。

shallow 和 mount 的区别，就是 shallow 只会渲染被测试的 React 组件这一层，不会渲染子组件；而 mount 则是完整地渲染 React 组件包括其所有子组件，包括触发 `componentDidMount` 生命周期函数。

原则上，能用 shallow 就尽量用 shallow，首先是为了测试性能考虑，其次是可以减少组件之间的影响，比如，一个组件 `Foo` 有子组件 `Bar`，如下：

```
const Foo = () => ()
    <div>
       {/* other logic */
       <Bar />
    </div>
)
```

如果用 mount 去渲染 Foo，会连带 Bar 一起完全渲染，如果 Bar 出了什么毛病，那 Foo 的单元测试也过不了；如果用 shallow，只知道 Bar 曾经被用，即使 Bar 哪里出了问题，也不影响 Foo 的单元测试。

这并不是说我们就不管 Bar，Bar 的质量会由它自己的单元测试来检验，这就引出下一个话题——代码覆盖率。

代码覆盖率

你不能给自己的程序随便写几个单元测试，就说自己的代码已经测试好了，就像上面我只给 ControlButtons 组件写了一个测试用例，我并不能说整个秒表应用已经通过了测试。

你的代码测试覆盖率只有达到一定程度，才好说自己的代码已经被测试了。

剩下来就是一个纠结的问题：代码测试的覆盖率应该达到多少才算够？

以我个人的经验，代码覆盖率必须达到 100%，也就是说，一个应用不光所有的单元测试都要通过，而且所有单元测试都必须覆盖到代码 100% 的角落。

如果对覆盖率的要求低于 100%，时间一长，质量必定会越来越下滑。

遇到一个不好测试的代码，开发者倾向于不去考虑如何重构代码提高可测试性，而是直接忽略这部分代码不去测试，反正不要求 100% 嘛；遇到工期比较紧的时候，甚至会进一步降低代码覆盖率要求，用牺牲质量来加快开发速度，反正不要求 100% 嘛。

所以，如果你真的对代码质量认真负责的话，请坚守 100% 代码覆盖率的底线！

在 create-react-app 创造的应用中，已经自带了代码覆盖率的支持，运行下面的命令，不光会运行所有单元测试，也会得到覆盖率汇报。

```
npm test -- --coverage
```

代码覆盖率包含四个方面：

1. 语句覆盖率
2. 逻辑分支覆盖率
3. 函数覆盖率
4. 代码行覆盖率

只有四个方面都是 100%，才算真的 100%。

小结

这一小节相对较为简略，这是有原因的。虽然我可以讲解很详细的单元测试工具使用方法，但是这并不是最重要的，最重要也最困难的是具有质量意识。

如果你具备质量意识，只需要强调上面说的 3 点，你就知道该怎么做，不需要我多说你也会找到对应的使用方法；相反，如果你不具备质量意识，说遍所有测试技巧，你也不会去实践，多说无益。

就像我在本节开始所说的，如果你根本不在意软件质量，完全可以忽略掉这一小节。

React 状态管理（1）：组件状态

在前面的章节中，我们反复声明过 React 其实就是这样一个公式：

```
UI = f(data)
```

f 的参数 data，除了 props，就是 state。props 是组件外传递进来的数据，state 代表的就是 React 组件的内部状态。

为什么要了解 React 组件自身状态管理

可能读者也知道在 React 开发社区中 Redux 和 Mobx 这样的状态管理工具，不过，我们首先不要管这些第三方工具，先从了解 React 组件自身的管理开始。

为什么呢?

第一个原因，因为 React 组件自身的状态管理是基础，其他第三方工具都是在这个基础上构筑的，连基础都不了解，无法真正理解第三方工具。

另一个重要原因，对于很多应用场景，React 组件自身的状态管理就足够解决问题，犯不上动用 Redux 和 MobX 这样的大杀器，简单问题简单处理，可以让代码更容易维护。

组件自身状态 state

什么数据放在 state 中

对于 React 组件而言，数据分为两种：

1. props
2. state

二者的区别显而易见，简单说就是，props 是外部传给组件的数据，而 state 是组件自己维护的数据，对外部是不可见的。

所以，判断某个数据以 props 方式存在，还是以 state 方式存在，并不难，只需要判断这个状态是否是组件内部状态。

一个经常被问到的问题，就是为什么不把组件的数据直接存放在组件类的成员变量中？比如像下面这样：

```
class Foo extends React.Component {
  foo = 'foo'

  render() {
    return (
      <React.Fragment>{this.foo}</React.Fragment>
    );
  }
}
```

像上面，数据存在 `this.foo` 中，而不是存在 `this.state.foo` 中，当这个组件渲染的时候，当然 `this.foo` 的值也就被渲染出来了，问题是，更新 `this.foo` 并不会引发组件的重新渲染，这很可能不是我们想要的。

所以，判断一个数据应该放在哪里，用下面的原则：

1. 如果数据由外部传入，放在 props 中；
2. 如果是组件内部状态，是否这个状态更改应该立刻引发一次组件重新渲染？如果是，放在 state 中；不是，放在成员变量中。

修改 state 的正确方式

组件自身的状态可以通过 `this.state` 读到，`this.state` 本身就是一个对象，但是修改状态不应该通过直接修改 `this.state` 对象来完成。因为，我们修改 state，当然不只是想修改这个对象的值，而是想引发 React 组件的重新渲染。

```
this.state.foo = 'bar'; //错误的方式

this.setState({foo:'bar'}); //正确的方式
```

如上面代码所示，如果只是修改 `this.state`，那改了也就只是改了这个对象，其他的什么都不会发生；如果使用 `setState` 函数，那不光修改 `state`，还能引发组件的重新渲染，在重新渲染中就会使用修改后的 `state`，这也就是达到根据 `state` 改变公式左侧 UI 的目的。

```
UI = f(state)
```

state 改变引发重新渲染的时机

现在我们知道应该用 `setState` 函数来修改组件 state，而且可以引发组件重新渲染，有意思的是，并不是一次 `setState` 调用肯定会引发一次重新渲染。

这是 React 的一种性能优化策略，如果 React 对每一次 `setState` 都立刻做一次组件重新渲染，那代价有点大，比如下面的代码：

```
this.setState({count: 1});
this.setState({caption: 'foo'});
this.setState({count: 2});
```

连续的同步调用 `setState`，第三次还覆盖了第一次调用的效果，但是效果只相当于调用了下面这样一次：

```
this.setState({count: 2, caption: 'foo'});
```

虽然明智的开发者不会故意连续写三个 setState 调用，但是代码一旦写得复杂，可能有多个 setState 分布在一次执行的不同代码片段中，还是会同步连续调用 setState，这时候，如果真的每个 setState 都引发一次重新渲染，实在太浪费了。

React 非常巧妙地用任务队列解决了这个问题，可以理解为每次 setState 函数调用都会往 React 的任务队列里放一个任务，多次 setState 调用自然会往队列里放多个任务。React 会选择时机去批量处理队列里执行任务，当批量处理开始时，React 会合并多个 setState 的操作，比如上面的三个 setState 就被合并为只更新 state 一次，也只引发一次重新渲染。

因为这个任务队列的存在，React 并不会同步更新 state，所以，在 React 中，setState 也不保证同步更新 state 中的数据。

state 不会被同步修改

简单说来，调用 setState 之后的下一行代码，读取 this.state 并不是修改之后的结果。

```
console.log(this.state.count);// 修改之前this.state.count为0
this.setState({count: 1})
console.log(this.state.count);// 在这里this.state.count依然为0
```

这乍看是很让人费解的结果，但是如果你理解了上面 React 任务队列的设计，一切也不难理解。

setState 只是给任务队列里增加了一个修改 this.state 的任务，这个任务并没有立即执行，所以 this.state 并不会立刻改变。

好吧，其实问题也没有那么简单，上面我所举的例子中，都假设 setState 是由 React 的生命周期函数或者事件处理函数中同步调用，这种情况下 setState 不会立即同步更新 state 和重新渲染，但是，如果调用 setState 由其他条件引发，就不是这样了。

看下面的代码，结果可能会出乎你的所料：

```
setTimeout(() => {
  this.setState({count: 2}); //这会立刻引发重新渲染
  console.log(this.state.count); //这里读取的count就是2
}, 0);
```

为什么 setTimeout 能够强迫 setState 同步更新 state 呢?

可以这么理解，当 React 调用某个组件的生命周期函数或者事件处理函数时，React 会想：“嗯，这一次函数可能调用多次 setState，我会先打开一个标记，只要这个标记是打开的，所有的 setState 调用都是往任务队列里放任务，当这一次函数调用结束的时候，我再去批量处理任务队列，然后把这个标记关闭。”

因为 setTimeout 是一个 JavaScript 函数，和 React 无关，对于 setTimeout 的第一个函数参数，这个函数参数的执行时机，已经不是 React 能够控制的了，换句话说，React 不知道什么时候这个函数参数会被执行，所以那个“标记”也没有打开。

当那个“标记”没有打开时，setState 就不会给任务列表里增加任务，而是强行立刻更新 state 和引发重新渲染。这种情况下，React 认为：“这个 setState 发生在自己控制能力之外，也许开发者就是想要强行同步更新呢，宁滥勿缺，那就同步更新了吧。”

知道这个“技巧”之后，可能会有开发者说：好啊，那么以后我就用 setTimeout 来调用 setState 吧，能够立刻更新 state，多好!

我劝你不要这么做。

就像上面所说，React 选择不同步更新 state，是一种性能优化，如果你用上 setTimeout，就没机会让 React 优化了。

而且，每当你觉得需要同步更新 state 的时候，往往说明你的代码设计存在问题，绝大部分情况下，你所需要的，并不是“state 立刻更新”，而是，“确定 state 更新之后我要做什么”，这就引出了 setState 另一个功能。

setState 的第二个参数

setState 的第二个参数可以是一个回调函数，当 state 真的被修改时，这个回调函数会被调用。

```
  console.log(this.state.count); // 0
  this.setState({count: 1}, () => {
    console.log(this.state.count); // 这里就是1了
  })
  console.log(this.state.count); // 依然为0
```

当 setState 的第二个参数被调用时，React 已经处理完了任务列表，所以 this.state 就是更新后的数据。

如果需要在 state 更新之后做点什么，请利用第二个参数。

函数式 setState

不管怎么说，setState 不能同步更新的确会带来一些麻烦，尤其是多个 setState 调用之间有依赖关系的时候，很容易写错代码。

一个很典型的例子，当我们不断增加一个 state 的值时：

```
  this.setState({count: this.state.count + 1});
  this.setState({count: this.state.count + 1});
  this.setState({count: this.state.count + 1});
```

上面的代码表面上看会让 this.state.count 增加 3，实际上只增加了 1，因为 setState 没有同步更新 this.state 啊，所以给任务队列加的三个任务都是给 this.state.count 同一个值而已。

面对这种情况，我们很自然地想到，如果任务列表中的任务不只是给 state 一个固定数据，如果任务列表里的“任务”是一个函数，能够根据当前 state 计算新的状态，那该多好！

实际上，setState 已经支持这种功能，到现在为止我们给 setState 的第一个参数都是对象，其实也可以传入一个函数。

当 setState 的第一个参数为函数时，任务列表上增加的就是一个可执行的任务函数了，React 每处理完一个任务，都会更新 this.state，然后把新的 state 传递给这个任务函数。

setState 第一个参数的形式如下：

```
function increment(state, props) {
  return {count: state.count + 1};
}
```

可以看到，这是一个纯函数，不光接受当前的 state，还接受组件的 props，在这个函数中可以根据 state 和 props 任意计算，返回的结果会用于修改 this.state。

如此一来，我们就可以这样连续调用 setState：

```
  this.setState(increment);
  this.setState(increment);
  this.setState(increment);
```

用这种函数式方式连续调用 setState，就真的能够让 this.state.count 增加 3，而不只是增加 1。

小结

通过这一小节，读者应该能够明白：

1. 如何确定数据以 props 还是以 state 形式存在；
2. 更新 state 的正确方法；
3. setState 通常并不会立刻更新 state；
4. 函数参数形式的 setState 才是推荐的用法。

React 状态管理（2）：Redux 使用模式

这一节我们来介绍 React 社区里名望很高的一个状态管理工具 Redux。不过，首先要明白，Redux 和 React 在技术上并没有什么直接关系。

虽然 Redux 的两个创造者 Dan Abramov 和 Andrew Clark 目前都是 React 的核心开发人员，而且 Redux 得到最广泛应用的场景是和 React 配合，但是，我们还是要明确，Redux 和 React 没有任何直接关系，Redux 可以用来管理 React 的状态，也可以用来管理其他应用的状态，只是由于历史原因 Redux 在 React 社区被应用最多罢了。

在出版《深入浅出React和Redux》一书后，我收到读者反馈，有不少读者反映看到 Redux 部分就看不大懂了，很大一部分原因，就是这些读者在学习 React 时已经习惯了用户界面的概念，突然出现一个和用户界面无关的 Redux，就觉得不好理解了。所以，我们还是从理解 Redux 开始。

理解 Redux

要理解 Redux，首先要明白我们为什么需要 Redux，或者说，Redux 适用于什么样的场景。

应用的状态往往十分复杂，如果应用状态就是一个普通 JavaScript 对象，而任何能够访问到这个对象的代码都可以修改这个状态，就很容易乱了套。当 bug 发生的时候，我们发现是状态错了，但是也很难理清到底谁把状态改错了，到底是如何走到出 bug 这一步。

Redux 的主要贡献，就是限制了对状态的修改方式，让所有改变都可以被追踪。

虽然 Redux 和 React 没有直接关系，但是我们依然以 React 应用为例，来说明 Redux 扮演什么角色。

在真实应用中，React 组件树会很庞大很复杂，两个没有父子关系的 React 组件之间要共享信息，怎么办呢？

最直观的方法，就是创建一个独立于这两个组件的对象，在这个对象中存放共享的数据，没错，这个对象，相当于一个 Store。

如果只是一个简单对象，那么任何人都可以修改 Store，这不大合适。所以我们做出一些限制，让 Store 只接受某些『事件』，如果要修改 Store 上的数据，就往 Store 上发送这些『事件』，Store 对这些『事件』的响应，就是修改状态。

这里所说的『事件』，就是 action，而对应修改状态的函数，就是 reducer。

Redux 中的 Store 其实实现的就是上述过程和概念，只不过实现很巧妙，让人没有办法绕过上面过程来修改状态，这种限制，是 Redux 成功的要素之一。

Redux 的内容并不少，具体使用方法超出了本小册的范围，因为本小册着重讲解高阶的技巧，读者如果对 Redux 还不熟悉，可以去看我写的书《深入浅出React和Redux》，了解 Redux 怎么用之后，更有利于学习后续内容。

适合 Redux 的场景

当一个 React 应用采用 Redux 之后，开发者往往就会陷入这样的纠结：对于某个状态，到底是放在 Redux 的 Store 中呢，还是放在 React 组件自身的状态中呢？

如果所有状态全都放在 Redux 的 Store 上，那就要对应增加 reducer 和 action 的代码，虽然拥有了可以跟踪的好处，但是对一些很细小的状态也要增加 reducer 和 action，会感觉很啰嗦（真的，Redux 本身就是一个啰嗦的技术，利用"啰嗦"来实现可维护性），开发者又会觉得得不偿失。

如果状态放在 React 组件中，感觉又白白放弃了 Redux 的优势，回到了 React 原生管理状态的老路上去，令人很不甘心。

面对这种左右为难的纠结状况，我这里有一套步骤，可以帮助开发者决定如何防止应用状态。

第一步，看这个状态是否会被多个 React 组件共享。

所谓共享，就是多个组件需要读取或者修改这个状态，如果是，那不用多想，应该放在 Store 上，因为 Store 上状态方便被多个组件共用，避免组件之间传递数据；如果不是，继续看第二步。

第二步，看这个组件被 unmount 之后重新被 mount，之前的状态是否需要保留。

举个简单例子，一个对话框组件。用户在对话框打开的时候输入了一些内容，不做提交直接关闭这个对话框，这时候对话框就被 unmount 了，然后重新打开这个对话框（也就是重新 mount），需求是否要求刚才输入的内容依然显示？如果是，那么应该把状态放在 Store 上，因为 React 组件在 unmount 之后其中的状态也随之消失了，要想在重新 mount 时重获之前的状态，只能把状态放在组件之外，Store 当然是一个好的选择；如果需求不要求重新 mount 时保持 unmount 之前的状态，继续看第三步。

第三步，到这一步，基本上可以确定，这个状态可以放在 React 组件中了。

不过，如果你觉得这个状态很复杂，需要跟踪修改过程，那看你个人喜好，可以选择放在 Store 上；如果你想简单处理，可以心安理得地让这个状态由 React 组件自己管理。

我想说明的是，React 组件的状态管理已经很强大了（在第 11 小节中有介绍），对于简单状态，尽量用 React 自己来搞定，只有那些适用场合不限于一个组件的，才有足够理由让 Redux 来管理。

代码组织方式

在应用中引入 Redux 之后，就会引入 action 和 reducer。从方便管理的角度出发，和 React 组件一样，action 和 reducer 都有自己独立的源代码文件，很自然，我们需要决定如何组织这些代码。

最傻的一种方法，就是把所有源代码文件放在一个目录下，代码文件少的时候还凑合着能看，一旦多起来，一个目录下各种类型文件会看花眼，所以这种方式不可取。

更好的方法，是把源代码文件分类放在不同的目录中，根据分类方式，可以分为两种：

1. 基于角色的分类（role based）
2. 基于功能的分类（feature based）

如果你曾经开发过 MVC 类应用，对基于角色的分类不会陌生。MVC 应用中在一个目录下放所有的 controller，在另一个目录下放所有的 view，在第三个目录下放所有的 model，每个目录下的文件都是同样的“角色”，这就是基于角色的分类。对应到使用 React 和 Redux 的应用，做法就是把所有 reducer 放在一个目录（通常就叫做 reducers），把所有 action 放在另一个目录（通常叫 actions），最后，把所有的纯 React 组件放在另一个目录。

另一种基于功能的分类方式，是把一个模块相关的所有源代码放在一个目录。例如，对于博客系统，有 Post（博客文章）和 Comment（注释）两个基本模块，建立两个目录 Post 和 Comment，每个目录下都有各自的 `action.js` 和 `reducer.js` 文件，如下所示，每个目录都代表一个模块功能，这就是基于功能的分类方式。

```
Post -- action.js
     |_ reucer.js
     |_ view.js
Comment -- action.js
        |_ reucer.js
        |_ view.js
```

一般说来，基于功能的分类方式更优。因为每个目录是一个功能的封装，方便共享，不过，我们也看到很多应用依然采用基于角色的方式组织代码，连 Facebook 开源的一些应用也采用这种方法，这很大程度上是因为这些应用开发一个模块的时候，没想过有朝一日要分享这些模块，换句话说这些模块开发出来就只被指望在这个应用中使用，这样一来，基于功能的组织方式也就没有必要了。

具体用哪种方式来组织代码，主要就看你是否预期这些模块会被共享，如果会，那采用基于功能的方式就是首选。

react-redux 中的模式

因为 Redux 是一个中立的状态管理工具，和 React 没有直接联系，所以，如果在 React 应用中使用 Redux，我们除了要引入 Redux，还需要导入 react-redux 这个安装包，安装方法如下：

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

在第 8 小节，我们介绍了『提供者模式』，react-redux 就是『提供者模式』的实践。在组件树的一个比较靠近根节点的位置，我们通过 `Provider` 来引入一个 store，代码如下：

```
import {createStore} from 'redux';
import {Provider} from 'react-redux';

const store = createStore(...);

// JSX
  <Provider store={store}>
    { // Provider之下的所有组件都可以connect到给定的store }
  </Provider>
```

这个 `Provider` 当然也是利用了 React 的 Context 功能。在这个 Provider 之下的所有组件，如果使用 connect，那么『链接』的就是 Provider 的 state。

以最简单的 Counter 为例来介绍一下 connect 的用法，首先，我们需要一个『傻瓜组件』，可以由纯函数实现，如下：

```
const CounterView = ({count, onIncrement}) => {
  return (
    <div>
      <div>{count}</div>
      <button onClick={onIncrement}>+</button>
    </div>
  );
};
```

上面的 `CounterView` 没有自己的 state，完全依赖于外部存储计数值，那么计数值存在哪呢？存在 store 上。我们要做的就是把 CounterView 和 store 连接起来，代码如下：

```
import {connect} from 'react-redux';

const mapStateToProps = (state) => {
  return {
```

```
    count: state.count
  };
}

const mapDispatchToProps = (dispatch) => ({
  onIncrement: () => dispatch({type: 'INCREMENT'})
});

const Counter = connect(mapStateToProps, mapDispatchToProps)(CounterView);
```

这里的 `connect` 函数接受两个参数，一个 `mapStateToProps` 是把 Store 上的 state 映射为 props；另一个 `mapDispatchToProps` 则是把回调函数类型的 props 映射为派发 action 的动作，connect 函数调用会产生一个『高阶组件』。

在第 6 小节我们介绍过『高阶组件』模式，一个高阶组件就是一个函数，它接受 React 组件为参数，返回一个新的 React 组件为结果。在上面的例子中，connect 产生的高阶组件产生了一个新的 React 组件 Counter，这个 Counter 其实就是一个『聪明组件』，它负责管理状态，而 CounterView 是一个『傻瓜组件』，只负责渲染。

从上面可以看出，在 react-redux 中，应用了三个 React 模式：

1. 提供者模式
2. 高阶组件
3. 聪明组件和傻瓜组件的分离

Redux 和 React 结合的最佳实践

应用 Redux 的时候，有这些业界已经证明的最佳实践：

1. Store 上的数据应该范式化。

所谓范式化，就是尽量减少冗余信息，像设计 MySQL 这样的关系型数据库一样设计数据结构。

2. 使用 selector。

对于 React 组件，需要的是『反范式化』的数据，当从 Store 上读取数据得到的是范式化的数据时，需要通过计算来得到反范式化的数据。你可能会因此担心出现问题，这种担心不是没有道理，毕竟，如果每次渲染都要重复计算，这种浪费积少成多可能真会产生性能影响，所以，我们需要使用 seletor。业界应用最广的 selector 就是 reslector 。

reselector 的好处，是把反范式化分为两个步骤，第一个步骤是简单映射，第二个步骤是真正的重量级运算，如果第一个步骤发现产生的结果和上一次调用一样，那么第二个步骤也不用计算了，可以直接复用缓存的上次计算结果。

绝大部分实际场景中，总是只有少部分数据会频繁发生变化，所以 reselector 可以避免大量重复计算。

3. 只 connect 关键点的 React 组件

当 Store 上状态发生改变的时候，所有 connect 上这个 Store 的 React 组件会被通知：『状态改变了！』

然后，这些组件会进行计算。connect 的实现方式包含 `shouldComponentUpdate` 的实现，可以阻挡住大部分不必要的重新渲染，但是，毕竟处理通知也需要消耗 CPU，所以，尽量让关键的 React 组件 connect 到 store 就行。

一个实际的例子就是，一个列表种可能包含几百个项，让每一个项都去 connect 到 Store 上不是一个明智的设计，最好是只让列表去 connect，然后把数据通过 props 传递给各个项。

一个还是多个 Store

虽然理论上一个应用可以有任意多个 Store，但是按照官方的推荐，一个应用只应该有一个 Store。实际上，一切用了多个 Store 的应用，都可以改为用单个 Store 解决。

不过，我在一次技术咨询中，见过使用多个 Store 的应用，这个应用十分庞大，不只是组件多，参与的团队也多，而且地域和管理结构上都很分散，每个团队都在一个网页上贡献组件，为了避免互相踩到对方的脚，他们干脆就各自创建和管理各自的 Store，各自开发的组件也只把状态放在自己的 Store 上。

上面这种多个 Store 的方式当然行得通，不同团队之间需要共享数据。如果一个 React 组件需要访问多个 Store，情况就会比较复杂。

使用 react-redux 的话，虽然 Provider 可以嵌套，但是，最里层的 Provider 提供的 store 才生效。

在下面的代码示例中，Foo 能够 connect 到的 store 是s tore1，而 Bar 能够 connect 到的是 store2，因为内层的 Provider 会屏蔽掉外层的 Provider。

```
<Provider store={store1}>
  <React.Fragment>
    <Foo />
    <Provider store={store2} >
      <React.Fragment>
        <Bar />
      </React.Fragment>
    </Provider>
 </React.Fragment>
</Provider>
```

如果真的需要让 Bar 来访问到 store1，那么就不能通过 Provider 来传递，只能通过 props 等方式传递，如此一来，引入了新的复杂度。

所以，建议还是尽量使用一个 Store，如果真的需要多个 Store，除非认定只有很少组件会访问多个 Store。

如何实现异步操作

使用 Redux 对于同步状态更新非常顺手，但是，遇到需要异步更新状态的场景，例如调用 AJAX 从服务器获得数据，这时候单用 Redux 就不够了，需要其他方式来辅助。

至今为止，还无法推荐一个杀手级的方法，各种方法都在吹嘘自己多厉害，但是任何一种方法都是易用性和复杂性的平衡。

最简单的 redux-thunk，代码量少，只有几行，用起来也很直观，但是开发者要写很多代码；而比较复杂的 redux-observable 相当强大，可以只用少量代码就实现复杂功能，但是前提是你要学会 RxJS，RxJS 本身学习曲线很陡，内容需要 一本书 的篇幅来介绍，这就是代价。

读者在自己的项目中，无论选择什么方式，一定要考虑这个方式的复杂度和学习成本。

在这里我不想过多介绍任何一种 Redux 扩展，因为任何一种都比不上 React 将要支持的 Suspense，Suspense 才是 React 中做异步操作的未来，在第 19 小节会详细介绍 Suspense。

小结

这一小节我们介绍了 Redux，读者应该掌握：

1. Redux 中的基本概念 action、reducer 和 store；
2. 使用 react-redux 会应用哪些设计模式；
3. 如何设计 Redux 的 Store。

React 状态管理（3）：Mobx 使用模式

既然说了 Redux，没有理由不聊一聊 Mobx，这两个状态管理工具在业界都被广泛应用，走得却是完全不同的两条路。

理解 Mobx

虽然 Mobx 和 Redux 有很大不同，但是至少还有一个共同点——这两个工具都和 React 没有任何直接关系，只不过凑巧 React 社区大量使用它们罢了。从技术上说，Mobx 和 Redux 都是中立的状态管理工具，他们能够应用于 React，也可以用于其他需要状态管理的场景。

还是用 create-react-app 产生的应用来演示 Mobx 吧，要使用 Mobx，首先我们要在项目中安装对应的 npm 包。

```
npm install mobx
```

我们用 Mobx 来实现一个很简单的计数工具，首先，需要有一个对象来记录计数值，代码如下：

```
import {observable} from 'mobx';

const counter = observable({
  count: 0
});
```

在上面的代码中，`counter` 是一个对象，其实就是用 `observable` 函数包住一个普通 JavaScript 对象，但是 `observable` 的介入，让 `counter` 对象拥有了神力。

我们用最简单的代码来展示这种“神力”，代码如下：

```
import {autorun} from 'mobx';

window.counter = counter;

autorun(() => {
  console.log('#count', counter.count);
});
```

把 `counter` 赋值给 `window.counter`，是为了让我们在 Chrome 的开发者界面可以访问。用 `autorun` 包住了一个函数，这个函数输出 `counter.count` 的值，这段代码的作用，我们很快就能看到。

在 Chrome 的开发者界面，我们可以直接访问 `window.counter.count`，神奇之处是，如果我们直接修改 `window.counter.count` 的值，可以直接触发 `autorun` 的函数参数！

这个现象说明，`mobx` 的 `observable` 拥有某种“神力”，任何对这个对象的修改，都会立刻引发某些函数被调用。和 observable 这个名字一样，被包装的对象变成了“被观察者”，而被调用的函数就是“观察者”，在上面的例子中，autorun 的函数参数就是“观察者”。

Mobx 这样的功能，等于实现了设计模式中的“观察者模式”（Observer Pattern），通过建立 observer 和 observable 之间的关联，达到数据联动。不过，传统的“观察者模式”要求我们写代码建立两者的关联，也就是写类似下面的代码：

```
observable.register(observer);
```

Mobx 最了不起之处，在于不需要开发者写上面的关联代码，Mobx自己通过解析代码就能够自动发现 observer 和 observable 之间的关系。

我们很自然想到，如果让我们的数据拥有这样的“神力”，那我们就不用在修改完数据之后，再费心去调用某些函数使用这些数据了，数据管理会变得十分轻松。

decorator

因为 Mobx 的作用就是把简单的对象赋予神力，总要有一种方法能够在不改变对象代码的前提，去改变对象的行为，这就用得上“装饰者模式”（Decorator Pattern）。

单独说“装饰者模式”，这只是面向对象编程思想下的一种模式，不过对 JavaScript 语言而言，就不只是一种模式，而是一种语言特性，它在语法上对这种模式提供了强大的支持，所谓强大，就是指使用起来代码极其简洁。

根据 JavaScript 语法，我们可以这样创造一个 decorator，叫做 `log`：

```
function log(target, name, descriptor) {
  console.log('#target', target);
  console.log('#name', name);
  console.log('#descriptor', descriptor);
  return descriptor;
}
```

当然，很明显这个 decorator 什么实质的事情都没做，只是用 `console.log` 输出了三个参数秀了一下存在感，最后返回的 `descriptor`，就是被这个『装饰者』所『装饰』的对象。

下面是使用这个 decorator 的代码示例：

```
@log
class Bar {
  @log
  bar() {
    console.log('bar');
  }
}
```

可以看到，`@` 符号就是使用 decorator 的标志，将 `@log` 作用于一个类 `Bar`，那么最后得到的 `Bar` 其实是调用 `log` 函数返回的结果；将 `@log` 作用于一个类成员 `@bar`，最后得到的 `bar` 同样是调用 `log` 函数之后得到的结果。可见，如果我们巧妙地编写 `log` 函数，控制返回的结果，就可以操纵被『装饰』的类或者成员。

编写 decorator 是一个复杂的过程，也超出了这本小册的范围，有兴趣的读者可以自行研究。在这里，读者只需要知道，虽然使用 Mobx 并不是必须使用 decorator，但是使用 decorator 会让 Mobx 的应用代码简洁易读很多。

在 create-react-app 中增加 decorator 支持

很不幸，create-react-app 产生的应用并不支持 decorator，官方解释 是：decorator 并没有成为稳定的标准，为了防止今天写的代码没多久就不好使，create-react-app 不会支持这样的不稳定的功能。

不过，这并不表示完全没有办法，事情可以解决，只是有些麻烦，我们要做的只是在应用中添加支持 decorator 的 babel plugin。

首先，我们动用 create-react-app 的 `eject` 功能，这表示我们和 create-react-app 缺省照顾一切的 `react-scripts` 一刀两断，从此之后，webpack 和 babel 配置就完全由我们自己控制。要注意，eject 是不可逆的，做了就收不回来了，所以，请谨慎使用 eject，不过，为了支持 decorator，我们也别无选择。

```
npm run eject
```

在 eject 之后，我们手动安装支持 decorator 的 babel 插件:

```
npm install --save-dev babel-plugin-transform-decorators-legacy
```

然后，我们找到 `package.json` 里面 `babel` 这一部分，添加 `plugins` 部分，让这一部分变成这个样子:

```
  "babel": {
    "plugins": [
      "transform-decorators-legacy"
    ],
    "presets": [
      "react-app"
    ]
  },
```

现在，我们离在 create-react-app 产生的应用中使用 decorator 只差一步了，记得我们说过 Mobx 和 React 并无直接关系吗？为了建立二者的关系，还需要安装 `mobx-react`:

```
npm install mobx-react
```

现在，我们可以使用 decorator 来操作 Mobx了。

用 decorator 来使用 Mobx

还是以 Counter 为例，看如何用 decorator 使用 Mobx，我们先看代码:

```
import {observable} from 'mobx';
```

```
import {observer} from 'mobx-react';

@observer
class Counter extends React.Component {
  @observable count = 0;

  onIncrement = () => {
    this.count ++;
  }

  onDecrement = () => {
    this.count --;
  }

  componentWillUpdate() {
    console.log('#enter componentWillUpdate');
  }

  render() {
    return(
      <CounterView
        caption="With decorator"
        count={this.count}
        onIncrement={this.onIncrement}
        onDecrement={this.onDecrement}
        />
    );
  }
}
```

在上面的代码中，`Counter` 这个 React 组件自身是一个 `observer`，而 `observable` 是 `Counter` 的一个成员变量 `count`。

注意 `observer` 这个 `decorator` 来自于 `mobx-react`，它是 Mobx 世界和 React 的桥梁，被它“装饰”的组件，只要用到某个被 Mobx 的 `observable` “装饰”过的数据，自然会对这样的数据产生反应。所以，只要 `Counter` 的 `count` 成员变量一变化，就会引发 `Counter` 组件的重新渲染。

可以注意到，`Counter` 的代码中并没有自己的 state，其实，被 `observer` 修饰过之后，`Counter` 被强行"注入”了 state，只不过我们看不见而已。

独立的 Store

虽然把 observer 和 observable 集中在一个 React 组件中可行，但是，这也让 observable 的状态被封存在了 React 组件内，那我们直接用 React 自身的 state 管理也能解决问题，所以，这样使用 Mobx 意义不大。

更多适用于 Mobx 的场合，就和适用于 Redux 的场合一样，是一个状态需要多个组件共享，所以 observable 一般是在 React 组件之外。

我们重写一遍 Counter 组件，代码如下：

```
const store = observable({
  count: 0
});
store.increment = function() {
  this.count ++;
```

```
};
store.decrement = function() {
  this.count --;
}

@observer // this decorator is must
class Counter extends React.Component {
  onIncrement = () => {
    store.increment();
  }

  onDecrement = () => {
    store.decrement();
  }

  render() {
    return(
      <CounterView
        caption="With external state"
        count={store.count}
        onIncrement={this.onIncrement}
        onDecrement={this.onDecrement}
        />
    );
  }
}
```

可以看到，我们把 `count` 提到组件之外，甚至就把它叫做 store，这延续的是 Redux 的命名方法。

小结

在这一小节中，我们介绍了 Mobx，读者应该能学到：

1. Mobx 的基本功能就是“观察者模式”
2. decorator 是“装饰者模式”在 JavaScript 语言中的实现
3. Mobx 通常利用 decorator 来使用
4. 如何利用 mobx-react 来管理 React组件的状态

React 状态管理（4）：不同方式对比

现在我们了解了 React 的状态管理，也学习了 Redux 和 Mobx 这两个第三方状态管理工具，是时候来对比一下这几种方式的优缺点了。

Mobx 和 Redux 的比较

Mobx 和 Redux 的目标都是管理好应用状态，但是最根本的区别在于对数据的处理方式不同。

Redux 认为，数据的一致性很重要，为了保持数据的一致性，要求Store 中的数据尽量范式化，也就是减少一切不必要的冗余，为了限制对数据的修改，要求 Store 中数据是不可改的（Immutable），只能通过 action 触发 reducer 来更新 Store。

Mobx 也认为数据的一致性很重要，但是它认为解决问题的根本方法不是让数据范式化，而是不要给机会让数据变得不一致。所以，Mobx 鼓励数据干脆就“反范式化”，有冗余没问题，只要所有数据之间保持联动，改了一处，对应依赖这处的数据自动更新，那就不会发生数据不一致的问题。

值得一提的是，虽然 Mobx 最初的一个卖点就是直接修改数据，但是实践中大家还是发现这样无组织无纪律不好，所以后来 Mobx 还是提供了 action 的概念。和 Redux 的 action 有点不同，Mobx 中的 action 其实就是一个函数，不需要做 `dispatch`，调用就修改对应数据，在上面的代码中，`increment` 和 `decrement` 就是 action。

如果想强制要求使用 action，禁止直接修改 observable 数据，使用 Mobx 的 `configure`，如下：

```
import {configure} from 'mobx';

configure({enforceActions: true});
```

总结一下 Redux 和 Mobx 的区别，包括这些方面：

1. Redux 鼓励一个应用只用一个 Store，Mobx 鼓励使用多个 Store；
2. Redux 使用“拉”的方式使用数据，这一点和 React是一致的，但 Mobx 使用“推”的方式使用数据，和 RxJS 这样的工具走得更近；
3. Redux 鼓励数据范式化，减少冗余，Mobx 容许数据冗余，但同样能保持数据一致。

然后，被问起最多的问题就来了：我应该选用 Mobx 还是 Redux 呢？

问：你的应用是小而且简单，还是大而且复杂？

如果是前者，选择 Mobx；如果是后者，选择 Redux。

问：你想要快速开发应用，还是想要长期维护这个应用？

如果是前者，选择 Mobx；如果是后者，选择 Redux。

我们真的必须使用 Mobx 和 Redux 吗

当然不是！

首先我们要明白，Redux 和 Mobx 都是一个特定时期的产物，在 React 没有提供更好的状态管理方法之前，Redux 能够帮助使用 React 的开发者一把，Mobx 也能提供一种全新的状态管理理念。

不过，React 是在持续发展的，光是 Context API 的改进，几乎就可以取代 react-redux 和 mobx-react 的作用。实际上，react-redux 和 mobx-react 两者的实现都依赖于 React 的 Context 功能。

以 Redux 为例，它相较于 React Context 还有哪些特点呢？

Redux 有更清晰的数据流转过程，配合 Redux Devtools 的确方便 Debug，代价就是我们必须要写啰嗦的 action 和 reducer。如果我们觉得应用并不需要 Redux 这样的增强功能，那完全就可以直接使用 React 的 Context。

当然，React 的 Context 还是过于简单了一点，我建议开发者不要只关注 Redux，可以尝试一些更加轻量级的第三方管理工具，其中的佼佼者，我认为就是 unstated 。

最后，还是苦口婆心地对开发者们说出这句我曾说过不下一万遍的话：**不要因为某个工具或者技术炫酷或者热门而去用它，要根据自己的工作需要去选择工具和技术。**

小结

本小节比较了多种管理 React 状态的方式，读者应该明白：

1. Redux 和 Mobx的不同；
2. 很多场合 React 自身的状态管理就足够用，并不是必须要使用 Redux 或者 Mobx 这样的第三方管理工具；
3. 随着 React 的发展，开发者对第三方工具的依赖会越来越少。

路由的魔法：React Router

随着 AJAX 技术的成熟，现在单页应用（Single Page Application）已经是前端网页界的标配，名为“单页”，其实在设计概念上依然是多页的界面，只不过从技术层面上页之间的切换是没有整体网页刷新的，只需要做局部更新。

要实现“单页应用”，一个最要紧的问题就是做好“路由”（Routing），也就是处理好下面两件事：

1. 把 URL 映射到对应的页面来处理；
2. 页面之间切换做到只需局部更新。

感谢业界前人给我们开发者铺平了道路，在 React 的世界里，上面说的问题都有成熟解法，其中最热门的工具，就是 react-router，这一节我们就来介绍这个工具。

react router v4 的动态路由

我们现在说到 react-router，基本上都是在说 react-router 的第 4 版，也就是 v4。这个 v4 很有意思，它完全推翻了之前 v3 的做法。可以说，react-router 的 v3 和 v4 版完完全全是不同的两个工具，两者差距实在太大。

其实当初 v3 也已经很优秀很热门了，但是 react-router 的开发者不满意，他们认为 v3 还是落入了“静态路由”的窠臼，所以在 v4 中 react-router 做到了“动态路由”的功能。

所谓“静态路由”，就是说路由规则是固定的，无论 express、Angular 还是 Rails 等业界响当当的框架，都用的是静态路由。以 express 为例，路由规则差不多是这么写的：

```
app.get('/', Home);
app.get('/product/:id', Product);
app.get('/about', About);
```

对于大部分应用，支持这样的路由规则真的是足够了，但是，react-router 的开发者觉得这样还不够好，要支持“动态路由”才是最好。

所谓动态路由，指的是路由规则不是预先确定的，而是在渲染过程中确定的。因为 react-router 的定位就是专供 React 应用服务，而 React 的世界中一切皆为组件，所以 react-router v4 就完全用 React 组件来实现路由功能。

不得不承认，虽然 react-router 的开发者是挺折腾的，但是他们的确是领悟了 React 的精髓，而且在 react-router 中把 React 的哲学发挥到了极致。

接下来，我们通过一个很简单的例子来说明 react-router v4 如何工作的，然后在这个例子的基础上介绍“动态路由”。

React Router 实例

安装包 react-router-dom

create-react-app 产生的应用默认为不支持多个页面，但还是在 README 文件中友情推荐了一下 react-router 来增强功能，可见 react-router 影响力之大。

不过，我们并不需要安装 react-router 这个 npm 包，因为 react-router 为了支持 Web 和 React Native 出了两个包—— react-router-dom 和 react-router-native ，我们只关心 Web，所以只需要安装 react-router-dom 。这个 react-router-dom 依赖于 react-router ， 所以 react-router 也会被自动安装上。

```
npm install react-router-dom
```

HashRouter 还是 BrowserRouter

react-router 的工作方式，是在组件树顶层放一个 Router 组件，然后在组件树中散落着很多 Route 组件（注意比 Router 少一个“r”），顶层的 Router 组件负责分析监听 URL 的变化，在它保护伞之下的 Route 组件可以直接读取这些信息。

很明显，Router 和 Route 的配合，就是之前我们介绍过的“提供者模式”，Router 是“提供者”，Route 是“消费者”。

更进一步，Router 其实也是一层抽象，让下面的 Route 无需各种不同 URL 设计的细节，不要以为 URL 就一种设计方法，至少可以分为两种。

第一种很自然，比如 `/` 对应 Home 页，`/about` 对应 About 页，但是这样的设计需要服务器端渲染，因为用户可能直接访问任何一个 URL，服务器端必须能对 `/` 的访问返回 HTML，也要对 `/about` 的访问返回 HTML。

第二种看起来不自然，但是实现更简单。只有一个路径 `/`，通过 URL 后面的 `#` 部分来决定路由，`/#/` 对应 Home 页，`/#/about` 对应 About 页。因为 URL 中 `#` 之后的部分是不会发送给服务器的，所以，无论哪个 URL，最后都是访问服务器的 `/` 路径，服务器也只需要返回同样一份 HTML 就可以，然后由浏览器端解析 `#` 后的部分，完成浏览器端渲染。

在 react-router，有 BrowserRouter 支持第一种 URL，有 HashRouter 支持第二种 URL。

因为 create-react-app 产生的应用默认不支持服务器端渲染，为了简单起见，我们在下面的例子中使用 HashRouter，在实际产品中，其实最好还是用 BrowserRouter，这样用户体验更好。

修改 `index.js` 文件，增加下面的代码：

```
import {HashRouter} from 'react-router-dom';

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

把 Router 用在 React 组件树的最顶层，这是最佳实践。因为将来我们如果想把 HashRouter 换成 BrowserRouter，组件 App 以下几乎不用任何改变。

使用 Link

对于单页应用，需要在不同“页面”之间切换，往往需要一个“导航栏”，我们在这里也实现一个简单的导航栏。

在 `App.js` 中，我们让网页由两个组件 Navigation 和 Content 组成， Navigation 就是导航栏，而 Content 是具体内容。

```
class App extends Component {
  render() {
    return (
      <div className="App">
        <Navigation />
        <Content />
      </div>
    );
  }
}
```

我们计划只增加两个页面，在 Navigation 中就应该有两个链接，但是，如果我们简单使用 HTML 的 `<a>` 标签那就错了，用户点击 `<a>` 标签缺省行为是网页跳转，这违背了“单页应用”的原则。虽然对于 HashRouter 使用的是没有网页跳转的 `#`，但是为了将来可以无缝切换为 BrowserRouter，我们也不能使用 `<a href="#about">` 这样的标签。

正确的解法是用 react-router 提供的 Link 组件，虽然 Link 最终还是渲染为 `<a>` 标签，但这是有神力的 `<a>` 标签，用户点击时，react-router 可以知晓这是一个单页应用的链接，不用网页跳转只做局部页面更新。

```
const ulStyle = {
  'list-style-type': 'none',
  margin: 0,
  padding: 0,
};

const liStyle = {
  display: 'inline-block',
  width: '60px',
};

const Navigation = () => (
  <header>
    <nav>
      <ul style={ulStyle}>
        <li style={liStyle}><Link to='/'>Home</Link></li>
        <li style={liStyle}><Link to='/about'>About</Link></li>
      </ul>
    </nav>
  </header>
)
```

使用 Route 和 Switch

我们来看 Content 这个组件，这里会用到 react-router 最常用的两个组件 Route 和 Switch。

```
const Content = () => (
  <main>
    <Switch>
      <Route exact path='/' component={Home}/>
      <Route path='/about' component={About}/>
    </Switch>
  </main>
)
```

Route 组件的 path 属性用于匹配路径，因为我们需要匹配 `/` 到 `Home`，匹配 `/about` 到 `About`，所以肯定需要两个 Route，但是，我们不能这么写。

```
      <Route path='/' component={Home}/>
      <Route path='/about' component={About}/>
```

如果按照上面这么写，当访问 `/about` 页面时，不光匹配 `/about`，也配中 `/`，界面上会把 Home 和 About 都渲染出来的。

解决方法，可以在想要精确匹配的 Route 上加一个属性 `exact`，或者使用 Switch 组件。

可以把 Switch 组件看做是 JavaScript 的 switch 语句，像这样：

```
switch (条件) {
  case 1: 渲染1; break;
  case 2: 渲染2; break;
}
```

从上往下找第一个匹配的 Route，匹配中了之后，立刻就 break，不继续这个 Switch 下其他的 Route 匹配了。

可以看到，react-router 巧妙地用 React 组件实现了路由的所有逻辑，印证了那句话：React 世界里一切都是组件。

动态路由

在了解了 react-router的基本路由功能之后，再来理解"动态路由"就容易了。

假设，我们增加一个新的页面叫 `Product`，对应路径为 `/product`，但是只有用户登录了之后才显示。如果用静态路由，我们在渲染之前就确定这条路由规则，这样即使用户没有登录，也可以访问 `product`，我们还不得不在 Product 组件中做用户是否登录的检查。

如果用动态路由，则只需要在代码中的一处涉及这个逻辑：

```
    <Switch>
      <Route exact path='/' component={Home}/>
      {
        isUserLogin() &&
        <Route exact path='/product' component={Product}/>,
      }
      <Route path='/about' component={About}/>
    </Switch>
```

可以用任何条件决定 Route 组件实例是否渲染，比如，可以根据页面宽度、设备类型决定路由规则，动态路由有了最大的自由度。

小结

这一小节我们介绍了 React 世界中最热门的路由工具 react-router，读者应该能够理解：

1. 单页应用中路由功能的必要；
2. 如何使用 react-router v4 来实现路由；
3. 动态路由的意义。

服务器端渲染（1）：基本套路

这一节，我们来介绍如何利用 React 做服务器端渲染，我们会先了解一下服务器端渲染的作用，然后在下一节我们会看一看业界公认最好的服务器端渲染框架 next.js 是怎么做的。

服务器端渲染

React 的功能就是把数据转化成用户界面，我们还是要祭出这个公式：

```
UI = f(data)
```

既然对 React 而言，“吃”进去的是 data，“吐”出来的是 UI，那么，这个转化过程在浏览器端可以做，当然在服务器端也可以做，不同的是浏览端的效果是直接操作 DOM，服务器端没有 DOM 可言，所以是直接吐出 HTML 字符串。

为什么要服务器端渲染

最近几年浏览器端框架很繁荣，以至于很多新入行的开发者只知道浏览器端渲染框架，都不知道存在服务器端渲染这回事，其实，网站应用最初全都是服务器端渲染，由服务器端用 PHP、Java 或者 Python 等其他语言产生 HTML 来给浏览器端解析。

相比于浏览器端渲染，服务器端渲染的好处是：

1. 可以缩短“第一有意义渲染时间”（First-Meaningful-Paint-Time）。

如果完全依赖于浏览器端渲染，那么服务器端返回的 HTML 就是一个空荡荡的框架和对 JavaScript 的应用，然后浏览器下载 JavaScript，再根据 JavaScript 中的 AJAX 调用获取服务器端数据，再渲染出 DOM 来填充网页内容，总共需要三个 HTTP 或 HTTPS 请求。

如果使用服务器端渲染，第一个 HTTP/HTTPS 请求返回的 HTML 里就包含可以渲染的内容了，这样用户第一时间就会感觉到“有东西画出来了”，这样的感知性能更好。

2. 更好的搜索引擎优化（Search-Engine-Optimization，SEO）。

大部分网站都希望自己能够出现在搜索引擎的搜索页前列，这个前提就是网页内容要能够被搜索引擎的爬虫正确抓取到。虽然 Google 这样的搜索引擎已经可以检索浏览器端渲染的网页，但毕竟不是全部搜索引擎都能做到，如果搜索引擎的爬虫只能拿到服务器端渲染的内容，完全浏览器端渲染就行不通了。

即使对于 Google，网页性能也是搜索排名的重要指标，如果通过服务器端渲染提高网页性能，网页的排名更可能靠前。

上面两点，就是服务器端渲染的主要意义。

React 对服务器端渲染的支持

因为 React 是声明式框架，所以，在渲染上对服务器端渲染非常友好。

假设我们我们要渲染一个以 `App` 为最根节点的组件树，浏览器端渲染的代码如下：

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

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

现在我们想要在服务器端渲染 `App`，如果使用 React v16 之前的版本，代码是这样：

```
import React from 'react';
import ReactDOMServer from 'react-dom/server';

// 把产生html返回给浏览器端
const html = ReactDOMServer.renderToString(<Hello />);
```

从 React v16 开始，上面的服务器端代码依然可以使用，但是也可以把 `renderToString` 替换为 `renderToNodeStream`，代码如下：

```
import React from 'react';
import ReactDOMServer from 'react-dom/server';

// 把渲染内容以流的形式塞给response
ReactDOMServer.renderToNodeStream(<Hello />).pipe(response);
```

此外，浏览器端代码也有一点变化，`ReactDOM.render` 依然可以使用，但是官方建议替换为 `ReactDOM.hydrate`，原来的 `ReactDOM.render` 将来会被废弃掉。

`renderToString` 的功能是一口气同步产生最终 HTML，如果 React 组件树很庞大，这样一个同步过程可能比较耗时。假设渲染完整 HTML 需要 500 毫秒，那么一个 HTTP/HTTPS 请求过来，500 毫秒之后才返回 HTML，显得不大合适，这也是为什么 v16 提供了 `renderToNodeStream` 这个新 API 的原因。

`renderToNodeStream` 把渲染结果以“流”的形式塞给 response 对象（这里的 response 是 express 或者 koa 的概念），这意味着不用等到所有 HTML 都渲染出来了才给浏览器端返回结果，也许 10 毫秒内就渲染出来了网页头部，那就没必要等到 500 毫秒全部网页都出来了才推给浏览器，“流”的作用就是有多少内容给多少内容，这样用户只需要 10 毫秒多一点的延迟就可以看到网页内容，进一步改进了“第一有意义渲染时间”。

服务器端渲染的难点

看到这里，你可能觉得服务器端渲染也太简单了，的确，因为 React 组件可以不必关心自己是在哪个端渲染，可以做到代码一次编写，到处都可以执行。但是，真的这么简单吗？

为了简化问题，上面的代码示例有意忽略了一个事实，那就是，应用往往需要外部服务器获取数据啊！

除非你的网页应用根本没有动态内容，不然你必须要考虑在服务器端怎么给 React 组件获取数据。

比如，你现在看到的掘金小册，为了渲染你所看到的页面，需要调用掘金小册的服务器 API 来获取这篇文章的内容。对于浏览器端渲染，在 `componentDidMount` 里调用 AJAX 就好了；对于服务器端渲染，要想产生 HTML 的包含内容，必须事先把数据准备好，也就是说，代码要是这样才行：

```
import React from 'react';
import ReactDOMServer from 'react-dom/server';

callAPI().then(result => {
  const props = result;
  ReactDOMServer.renderToNodeStream(<Hello {...props}/>).pipe(response);
});
```

最大的问题来了，如何给组件获取和提供数据呢？

解决了这个问题，才算真的解决了服务器端渲染的问题。

“脱水”和“注水”

React 有一个特点，就是把内容展示和动态功能集中在一个组件中。比如，一个 Counter 组件既负责怎么画出内容，也要负责怎么响应按键点击，这当然符合软件高内聚性的原则，但是也给服务器端渲染带来更多的工作。

设想一下，如果只使用服务器端渲染，那么产生的只有 HTML，虽然能够让浏览器端画出内容，但是，没有 JavaScript 的辅助是无法响应用户交互事件的。对应 Counter 的例子，一个 Counter 组件在浏览器中也就渲染出一个数字两个按钮，用户点击 `+` 按钮或者 `-` 按钮，什么都不会发生。

很显然我们必须要在浏览器端赋予 Counter 组件一些“神力”，让它能够响应事件。那么怎么赋予 Counter 组件“神力”呢？其实我们已经做过这件事了，Counter 组件里面已经有对按钮事件的处理，我们所要做的只是让 Counter 组件在浏览器端重新执行一遍，也就是 mount 一遍就可以了。

也就是说，**如果想要动态交互效果，使用 React 服务器端渲染，必须也配合使用浏览器端渲染。**

现在问题变得更加有趣了，在服务器端我们给 Counter 一个初始值（这个值可以不是缺省的 0），让 Counter 渲染产生 HTML，这些 HTML 要传递给浏览器端，为了让 Counter 的 HTML“活”起来点击相应事件，必须要在浏览器端重新渲染一遍 Counter 组件。在浏览器端渲染 Counter 之前，用户就可以看见 Counter 组件的内容，但是无法点击交互，要想点击交互，就必须要等到浏览器端也渲染一次 Counter 之后。

接下来的一个问题，如果服务器端塞给 Counter 的数据和浏览器端塞给 Counter 的数据不一样呢？

在 React v16 之前，React 在浏览器端渲染之后，会把内容和服务器端给的 HTML 做一个比对。如果完全一样，那最好，接着用服务器端 HTML 就好了；如果有一丁点不一样，就会立刻丢掉服务器端的 HTML，重新渲染浏览器端产生的内容，结果就是用户可以看到界面闪烁。因为 React 抛弃的是整个服务器端渲染内容，组件树越大，这个闪烁效果越明显。

React 在 v16 之后，做了一些改进，不再要求整个组件树两端渲染结果分毫不差，但是如果发生不一致，依然会抛弃局部服务器端渲染结果。

总之，**如果用服务器端渲染，一定要让服务器端塞给 React 组件的数据和浏览器端一致。**

为了达到这一目的，必须把传给 React 组件的数据给保留住，随着 HTML 一起传递给浏览器网页，这个过程，叫做“脱水”（Dehydrate）；在浏览器端，就直接拿这个“脱水”数据来初始化 React 组件，这个过程叫“注水”（Hydrate）。

前面提到过 React v16 之后用 `React.hydrate` 替换 `React.render`，这个 `hydrate` 就是“注水”。

总之，为了实现React的服务器端渲染，必须要处理好这两个问题：

1. 脱水
2. 注水

Facebook 未使用服务器端渲染

值得一提的是，虽然 React 从最初版本就支持“服务器端渲染”，并且 React 的创建者 Facebook 也全力在自己的网站产品中使用 React，但他们自己却没有使用 React 的服务器端渲染功能。理由是，Facebook 已经在 PHP 上投入了很多资源，不打算放弃这些投入。

这里我当然不是批评 Facebook，实际上，Facebook 对 React 的支持是真心的，它在自己的网站上大范围使用 React，而不只是做出来后让外部使用者当小白鼠，这种全力投入也给了 React 使用者很大信心。但另一方面，因为 Facebook 自己不用 React 的服务器端渲染，如何利用这个功能，就缺乏一个官方参考标准了。

也许就是因为缺乏 Facebook 的官方标准，业界对服务器端渲染的解决方法层出不穷，不过，到目前看来，`next.js` 还是最佳方案。

小结

本小节介绍了 React 的服务器端渲染功能，读者应该理解下列要点：

1. 服务器端渲染的作用；
2. React 的服务器端渲染支持方法；
3. 什么叫“脱水”和“注水”。

服务器端渲染（2）：理解 Next.js

我们已经知道了服务器端渲染的原理，你只需要搭建一个 Express 服务器，在服务器端手工打造『脱水』，在浏览器端做『注水』，完成某个页面的服务器端渲染并不难。

不过，服务器端渲染的问题并不这么简单，一个最直接的问题，就是怎么处理多个页面的『单页应用』（Single-Page-Application）？

所以单页应用，就是虽然用户感觉有多个页面，但是实现上只有一个页面，用户感觉到页面可以来回切换，但其实只是一个页面并没有完全刷新，只是局部界面更新而已。

假设一个单页应用有三个页面 Home、Prodcut 和 About，分别对应的的路径是 `/home`、`/product` 和 `/about`，而且三个页面都依赖于 API 调用来获取外部数据。

现在我们要做服务器端渲染，如果只考虑用户直接在地址栏输入 `/home`、`/product` 和 `/about` 的场景，很容易满足，按照上面说的套路做就是了。但是，这是一个单页应用，用户可以在 Home 页面点击链接无缝切换到 Product，这时候 Product 要做完全的浏览器端渲染。换句话说，每个页面都需要既支持服务器端渲染，又支持完全的浏览器端渲染，更重要的是，对于开发者来说，肯定不希望为了这个页面实现两套程序，所以必须有同时满足服务器端渲染和浏览器端渲染的代码表示方式。

读者可以思考一下什么样的代码表示合适，也可以直接往下，看看业界公认最科学的实现方式 Next.js 是如何做的。

快速创建 Next.js 项目

在说明 Next.js 的工作原理之前，我们先看怎么快速创建 Next.js 项目，这个问题用代码来说明会更顺畅。

我们也可以手工创建 Next.js 项目，不过更简单的方式是用自动化工具 create-next-app，这个 create-next-app 类似于 create-react-app，一个命令就创建一个可以运行的应用。

首先安装 create-next-app。

```
npm install -g create-next-app
```

然后，就可以在你专门存放项目的目录下执行 create-next-app，产生一个使用 Next.js 的 React 应用，下面的命令创建一个叫 next_demo 的应用：

```
create-next-app next_demo
```

进入新生成的项目目录 next_demo 里检查一下，可以看到文件结构非常简洁，`pages` 目录下是页面文件，`package.json` 中差不是下面这样，没有繁冗的 webpack 和 babel 依赖包，因为一切都被 Next.js 封装起来了。

```
{
  "name": "create-next-example-app",
  "scripts": {
    "dev": "next",
    "build": "next build",
    "start": "next start"
```

```
  },
  "dependencies": {
    "next": "^6.0.3",
    "react": "^16.5.2",
    "react-dom": "^16.5.2"
  }
}
```

虽然有不少框架都表示自己的功能很强大，但其中有很多框架的设计并不中立，用这些框架去开发某些特定应用或许还行，如果放到一个更大范围的应用类型中，就会发现无法满足要求，这样的框架通用性不足，开发者一定要谨慎使用。

讲良心话，Next.js 真的是一个通用性非常高的框架，因为 Next.js 完全遵从了 React 的技术哲学：一切皆为组件。

在 Next.js 中，创造一个页面，其实就是创造一个 React 组件，接下来我们看看如何创建一个页面。

编写页面

使用下面的命令启动 Next.js 应用，进入的是开发者模式，这时候对代码的改变，会立刻体现在网页上。

```
npm run dev
```

请注意，这一点上 Next.js 的习惯用法和 create-react-app 产生的应用不一样。在 create-react-app 产生的应用中， npm run start 启动是开发者模式，但在 Next.js 应用中，习惯上 npm run start 以产品模式启动，所以要先运行 npm run build 然后才能运行 npm run start。

Next.js 遵从『协定优于配置』（convention over configuration）的设计原则，根据『协定』，在 `pages` 中每个文件对应一个网页文件，文件名对应的就是网页的路径名，比如 `pages/home.js` 文件对应的就是 `/home` 路径的页面，当然 `pages/index.js` 比较特殊，对应的是默认根路径 `/` 的页面。

我们修改 `pages/index.js`，让它更简单一些，如下：

```
import React from 'react'

const Home = (props) => (
  <h1>
    Hello World
  </h1>
)

export default Home
```

这样会在页面上显示出一个 `Hello World`，而这个页面代码就是一个普通的 React 组件而已。

页面都是 React 组件，这就是 Next.js 的哲学。

getInitialProps

我们还是要回到本来的话题，如何优雅地实现服务器端渲染，上面的 `Home` 页面虽然能够渲染出完整包含 `Hello World` 的 HTML，但是并没有调用任何外部 API 资源，所以也没有异步操作，并不能体现服务器端渲染的难度。

我们用一个函数来实现异步操作，以此模拟调用 API 的延迟效果，如下：

```
const timeout = (ms, result) => {
  return new Promise(resolve => setTimeout(() => resolve(result), ms));
};
```

然后，我们利用这个 `timeout` 来获得展示网页所需的数据。比如说，获取用户名，那么我们的 Home 组件就要换一个写法，像下面那样，增加 `getInitialProps` 的定义：

```
const Home = (props) => (
  <h1>
    Hello {props.userName}
  </h1>
)

Home.getInitialProps = async () => {
  return await timeout(200, {userName: 'Morgan'});
};
```

这个 getiInitialProps 是 Next.js 最伟大的发明，它确定了一个规范，一个页面组件只要把访问 API 外部资源的代码放在 getInitialProps 中就足够，其余的不用管，Next.js 自然会在服务器端或者浏览器端调用 getInitialProps 来获取外部资源，并把外部资源以 props 的方式传递给页面组件。

注意 getInitialProps 是页面组件的静态成员函数，可以用下面的方法定义：

```
Home.getInitialProps = async () = {...};
```

也可以在组件类中加上 `static` 关键字定义：

```
class Home extends React.Component {
  static async getInitialProps() {
     ...
  }
}
```

通过上面的代码，我么也可以注意到，getInitialProps 是一个 async 函数，所以，在 getInitialProps 函数中可以使用 `await` 关键字，用同步的方式编写异步逻辑。

我们可以这样来看待 getInitialProps，它就是 Next.js 对代表页面的 React 组件生命周期的扩充。React 组件的生命周期函数缺乏对异步操作的支持，所以 Next.js 干脆定义出一个新的生命周期函数 getInitialProps，在调用 React 原生的所有生命周期函数之前，Next.js 会调用 getInitialProps 来获取数据，然后把获得数据作为 props 来启动 React 组件的原生生命周期过程。

这个生命周期函数的扩充十分巧妙，因为：

1. 没有侵入 React 原生生命周期函数，以前的 React 组件该怎么写还是怎么写；

2. getInitialProps 只负责获取数据的过程，开发者不用操心什么时候调用 getInitialProps，依然是 React 哲学的声明式编程方式；
3. getInitialProps 是 async 函数，可以利用 JavaScript 语言的新特性，用同步的方式实现异步功能。

Next.js 的“脱水”和“注水”

我们说过服务器端渲染的关键是如何“脱水”和“注水”，如果你对 Next.js 如何实现这两个关键点好奇（实际上你确实应该感到好奇），那么在浏览器中使用“显示网页源代码”就可以让你一目了然。

在网页的 HTML 中，可以看到类似下面的内容：

```
<script>
  __NEXT_DATA__ = {
    "props":{
      "pageProps": {"userName":"Morgan"}},
      "page":"/","pathname":"/","query":{},"buildId":"-
","assetPrefix":"","nextExport":false,"err":null,"chunks":[]}
</script>
```

Next.js 在做服务器端渲染的时候，页面对应的 React 组件的 getInitialProps 函数被调用，异步结果就是“脱水”数据的重要部分，除了传给页面 React 组件完成渲染，还放在内嵌 script 的 `__NEXT_DATA__` 中，这样，在浏览器端渲染的时候，是不会去调用 getInitialProps 的，直接通过 `__NEXT_DATA__` 中的“脱水”数据来启动页面 React 组件的渲染。

这样一来，如果 getInitialProps 中有调用 API 的异步操作，只在服务器端做一次，浏览器端就不用做了。

那么，getInitialProps 什么时候会在浏览器端调用呢？

当在单页应用中做页面切换的时候，比如从 Home 页切换到 Product 页，这时候完全和服务器端没关系，只能靠浏览器端自己了，Product页面的 getInitialProps 函数就会在浏览器端被调用，得到的数据用来开启页面的 React 原生生命周期过程。

关键点是，浏览器可能会直接访问 `/home` 或者 `/product`，也可能通过网页切换访问这两个页面，也就是说 Home 或者 Product 都可能被服务器端渲染，也可能完全只有浏览器端渲染，不过，这对应用开发者来说无所谓，应用开发者只要写好 getInitialProps，至于调用 getInitialProps 的时机，交给 Next.js 处理就好了。

你可以发明自己的服务器端框架，但很可能最后你发现，如果要做得通用性好，最后都会做到和 Next.js 一样的模式上来。

值得一提的是，getInitialProps 返回的应该是“纯数据”，也就是不要返回一个定制类的实例。比如，有一个类 Foo 有一个成员函数 bar，不要在 getInitialProps 返回一个 Foo 实例。不然，经过“脱水”和“注水”过程，网页组件获得的那个“Foo 实例”不再是你想的那个 Foo 实例了，它变成了一个纯粹的数据，不会包含成员函数 bar的。

小结

这一小节我们介绍了 React 服务器端渲染最优秀的框架 Next.js。Next.js 的内容很多，读者可以在官网深入了解，这一小节主要讲了以下内容：

1. getInitialProps 是 Next.js 的核心，也是 Next.js 最伟大的发明；
2. 在 getInitialProps 中实现 AJAX 等异步操作，其他 React 组件就无需关心自己是在服务器端还是浏览器端被渲染；

3. Next.js 如何实现“脱水”和“注水”。

React 的未来（1）： 拥抱异步渲染

在这一节中，我们会展望一下 React 的未来，不过，在向前看之前，我们要回顾一下 React 的历史。

React 简史

React 最初是 Facebook 的一个内部项目，投入实际产品是在 2011 年，用于 Facebook 的时间线界面，随后，Facebook 在其收购的 Instagram 中也使用了这种技术，2013 年 5 月，Facebook 决定将 React 开源。

有很长一段时间，React 一直是以 0.x.x 的模式来增长版本，大版本是 0，只有小版本和补丁版本迭代。直到 2016 年 4 月，React 经过这么长时间产品环境的磨练，已经足够成熟，所以版本一下子从 v0.14.8 跳到了 v15.0.0。其实这种版本改变的模式对开发者并没有什么直接影响，但是反映了 Facebook 致力于 React 开发的决心。

到了 v16.0.0 的时候，React 有一个重大改变——核心代码被重写，引入了叫 Fiber 这个全新的架构。这个架构使得 React 用异步渲染成为可能，但要注意，这个改变只是让异步渲染（async rendering）成为“可能”，React 却并没有在 v16 发布的时候立刻开启这种“可能”，也就是说，React 在 v16 发布之后依然使用的是同步渲染。不过，虽然异步渲染没有立刻采用，Fiber 架构还是打开了通向新世界的大门，React v16 一系列新功能几乎都是基于 Fiber 架构。

要面向 React 未来，我们首先要理解这个异步渲染的概念。

同步渲染的问题

长期以来，React 一直用的是同步渲染，这样对 React 实现非常直观方便，但是会带来性能问题。

假设有一个超大的 React 组件树结构，有 1000 个组件，每个组件平均使用 1 毫秒，那么，要做一次完整的渲染就要花费 1000 毫秒也就是 1 秒钟，然而 JavaScript 运行环境是单线程的，也就是说，React 用同步渲染方式，渲染最根部组件的时候，会同步引发渲染子组件，再同步渲染子组件的子组件……最后完成整个组件树。在这 1 秒钟内，同步渲染霸占 JavaScript 唯一的线程，其他的操作什么都做不了，在这 1 秒钟内，如果用户要点击什么按钮，或者在某个输入框里面按键，都不会看到立即的界面反应，这也就是俗话说的“卡顿”。

在同步渲染下，要解决“卡顿”的问题，只能是尽量缩小组件树的大小，以此缩短渲染时间，但是，应用的规模总是在增大的，不是说缩小就能缩小的，虽然我们利用定义 shouldComponentUpdate 的方法可以减少不必要的渲染，但是这也无法从根本上解决大量同步渲染带来的“卡顿”问题。

异步渲染：两阶段渲染

React Fiber 引入了异步渲染，有了异步渲染之后，React 组件的渲染过程是分时间片的，不是一口气从头到尾把子组件全部渲染完，而是每个时间片渲染一点，然后每个时间片的间隔都可去看看有没有更紧急的任务（比如用户按键），如果有，就去处理紧急任务，如果没有那就继续照常渲染。

根据 React Fiber 的设计，一个组件的渲染被分为两个阶段：第一个阶段（也叫做 render 阶段）是可以被 React 打断的，一旦被打断，这阶段所做的所有事情都被废弃，当 React 处理完紧急的事情回来，依然会重新渲染这个组件，这时候第一阶段的工作会重做一遍；第二个阶段叫做 commit 阶段，一旦开始就不能中断，也就是说第二个阶段的工作会稳稳当当地做到这个组件的渲染结束。

两个阶段的分界点，就是 `render` 函数。render 函数之前的所有生命周期函数（包括 render)都属于第一阶段，之后的都属于第二阶段。

开启异步渲染，虽然我们获得了更好的感知性能，但是考虑到第一阶段的的生命周期函数可能会被重复调用，不得不对历史代码做一些调整。

在 React v16.3 之前，render 之前的生命周期函数（也就是第一阶段生命周期函数）包括这些：

- componentWillReceiveProps
- shouldComponentUpdate
- componentWillUpdate
- componentWillMount
- render

下图是 React v16.3 之前的完整的生命周期函数图：

React 官方告诫开发者，虽然目前所有的代码都可以照常使用，但是未来版本中会废弃掉，为了将来，使用 React 的程序应该快点去掉这些在第一阶段生命函数中有副作用的功能。不得不说 React 真的很够意思，提前这么久告诉大家这个事情，让大家有足够的时间去修改自己的代码。

一个典型的错误用例，也是我被问到做多的问题之一：为什么不在 componentWillMount 里去做 AJAX？componentWillMount 可是比 componentDidMount 更早调用啊，更早调用意味着更早返回结果，那样性能不是更高吗？

首先，一个组件的 componentWillMount 比 componentDidMount 也早调用不了几微秒，性能没啥提高；而且，等到异步渲染开启的时候，componentWillMount 就可能被中途打断，中断之后渲染又要重做一遍，想一想，在 componentWillMount 中做 AJAX 调用，代码里看到只有调用一次，但是实际上可能调用 N 多次，这明显不合适。相反，若把 AJAX 放在 componentDidMount，因为 componentDidMount 在第二阶段，所以绝对不会多次重复调用，这才是 AJAX 合适的位置（当然，React 未来有更好的办法，在下一小节 Suspense 中可以讲到）。

getDerivedStateFromProps

到了 React v16.3，React 干脆引入了一个新的生命周期函数 getDerivedStateFromProps，这个生命周期函数是一个 static 函数，在里面根本不能通过 `this` 访问到当前组件，输入只能通过参数，对组件渲染的影响只能通过返回值。没错，getDerivedStateFromProps 应该是一个纯函数，React 就是通过要求这种纯函数，强制开发者们必须适应异步渲染。

```
static getDerivedStateFromProps(nextProps, prevState) {
  //根据nextProps和prevState计算出预期的状态改变，返回结果会被送给setState
}
```

到了 React v16.3，React 生命周期函数全图如下：

注意，上图中并包含全部React生命周期函数，在React v16发布时，还增加了一个 `componentDidCatch`，当异常发生时，一个可以捕捉到异常的 `componentDidCatch` 就排上用场了。不过，很快React觉着这还不够，在v16.6.0又推出了一个新的捕捉异常的生命周期函数 `getDerivedStateFromError`。

如果异常发生在第一阶段（render阶段），React就会调用 `getDerivedStateFromError`，如果异常发生在第二阶段（commit阶段），React会调用 `componentDidCatch`。这个区别也体现出两个阶段的区分对待。

适应异步渲染的组件原则

明白了异步渲染的来龙去脉之后，开发者就应该明白，现在写代码必须要为未来的某一次 React 版本升级做好准备，当 React 开启异步渲染的时候，你的代码应该做到在 render 之前最多只能这些函数被调用：

- 构造函数
- getDerivedStateFromProps
- shouldComponentUpdate

幸存的这些第一阶段函数，除了构造函数，其余两个全都必须是纯函数，也就是不应该做任何有副作用的操作。

实际上，如果之前你的用法规范，除了 shouldComponentUpdate 不怎么使用第一阶段生命周期函数，你还会发现不怎么需要改动代码，比如 componentWillMount 中的代码移到构造函数中就可以了。但是如果用法错乱，比如滥用componentWillReceiveProps，那就不得不具体情况具体分析，从而决定这些代码移到什么位置。

开发者中一个普遍的误区，就是总想把任务往前提，提到靠前的生命周期函数去，就像我前面说过的在 componentWillMount 中做 AJAX。正确的做法是根据各函数的语义来放置代码，并不是越往前越好。

小结

这一小节我们介绍了 React v16 引入的“异步渲染”，读者通过本节阅读应该能够理解：

1. 异步渲染的意义；
2. 理解两阶段渲染；
3. 为了应对异步渲染，我们开发 React 组件需要对生命周期函数做哪些调整。

React 的未来（2）：Suspense 带来的异步操作革命

上一节我们介绍了 Fiber 架构下的异步渲染机制，我们知道生命周期函数的修改是势在必行，那么，接下来呢？接下来 React 会有什么“大事”呢？

这个答案估计连 React 的核心开发者也在讨论中，不过从各种渠道信息看来，至少有两件“大事”在会在看得见的未来发生，那就是：

- Suspense
- Hooks

当然 React 增加的功能肯定远不止这点，将这两件“大事”在这里提出来，是因为它们对我们使用开发者的影响最大，会彻底改变我们的代码模式。

在写这本小册时，React 正式版是 v16.6.0，还只是 alpha 阶段，也许当你读到这本小册时，React 已经走得更远，但是你依然应该阅读这一小节，因为作为开发者你应该要明白技术演化的来龙去脉。

我们首先来了解 Suspense。Suspense 应用的场合就是异步数据处理，最常见的例子，就是通过 AJAX 从服务器获取数据，每一个 React 开发者都曾为这个问题纠结。

如果用一句话概括 Suspense 的功用，那就是：**用同步的代码来实现异步操作。**

而要理解 Suspense，我们先来体会一下 React 中做 AJAX 之类异步操作的痛苦。

React 同步操作的不足

上一节介绍过，React 最初的设计，整个渲染过程都是同步的。同步的意思是，当一个组件开始渲染之后，就必须一口气渲染完，不能中断，对于特别庞大的组件树，这个渲染过程会很耗时，而且，这种同步处理，也会导致我们的代码比较麻烦。

当我们开始渲染某个组件的时候，假设这个组件需要从服务器获取数据，那么，要么由这个组件的父组件想办法拿到服务器的数据，然后通过 props 传递进来，要么就要靠这个组件自力更生来获取数据，但是，没有办法通过一次渲染完成这个过程，因为渲染过程是同步的，不可能让 React 等待这个组件调用 AJAX 获取数据之后再继续渲染。

常用的做法，需要组件的 render 和 componentDidMount 函数配合。

1. 在 componentDidMount 中使用 AJAX，在 AJAX 成功之后，通过 setState 修改自身状态，这会引发一次新的渲染过程。
2. 在 render 函数中，如果 state 中没有需要的数据，就什么都不渲染或者渲染一个“正在装载”之类提示；如果 state 中已经有需要的数据，就可以正常渲染了，但这也必定是在 componentDidMount 修改了 state 之后，也就是只有在第二次渲染过程中才可以。

下面是代码实例：

```
class Foo extends React.Component {
  state = {
     data: null
  }

  render() {
     if (!this.state.data) {
        return null;
```

```
    } else {
       return <div>this.state.data</div>;
    }
  }

  componentDidMount() {
    callAPI().then(result => {
      this.setState({data: result});
    });
  }
}
```

这种方式虽然可行，我们也照这种套路写过不少代码，但它的缺点也是很明显的。

1. 组件必须要有自己的 state 和 componentDidMount 函数实现，也就不可能做成纯函数形式的组件。
2. 需要两次渲染过程，第一次是 mount 引发的渲染，由 componentDidMount 触发 AJAX 然后修改 state，然后第二次渲染才真的渲染出内容。
3. 代码啰嗦，十分啰嗦。

理想中的代码形式

而 Suspense 就是为了克服上述 React 的缺点。

在了解 Suspense 怎么解决这些问题之前，我们不妨自己想象一下，如果要利用 AJAX 获取数据，代码怎样写最简洁高效?

我先来说一说自己设想的最佳代码形式。首先，我不想写一个有状态的组件，因为通过 AJAX 获取的数据往往也就在渲染用一次，没必要存在 state 里；其次，想要使数据拿来就用，不需要经过 componentDidMount 走一圈。所以，代码最好是下面这样：

```
const Foo = () => {
  const data = callAPI();
  return <div>{data}</div>;
}
```

够简洁吧，可是目前的 React 版本做不到啊！

因为 `callAPI` 肯定是一个异步操作，不可能获得同步数据，无法在同步的 React 渲染过程中立足。

不过，现在做不到，不代表将来做不到，将来 React 会支持这样的代码形式，这也就是 Suspense。

Suspense

在 JsConf Iceland 2018 技术大会 上，React 的开发者展示了未来 React 会支持的新特性 Suspense，有了 Suspense，就可以在 React 中以同步的形式来写异步代码，代码形式类似下面：

```
const Foo = () => {
  const data = createFetcher(callAJAX).read();
  return <div>{data}</div>;
}
```

看到这里，你的第一反应很可能就是：怎么可能?

真的可能。

你还会想：这怎么做到的？怎么在同步的渲染过程中强行加入异步操作？

接下来，我们就介绍一下 Suspense 的原理。

在 React 推出 v16 的时候，就增加了一个新生命周期函数 componentDidCatch。如果某个组件定义了 componentDidCatch，那么这个组件中所有的子组件在渲染过程中抛出异常时，这个 componentDidCatch 函数就会被调用。

可以这么设想，componentDidCatch 就是 JavaScript 语法中的 `catch`，而对应的 `try` 覆盖所有的子组件，就像下面这样:

```
try {
  //渲染子组件
} catch (error) {
  // componentDidCatch被调用
}
```

Suspense 就是巧妙利用 componentDidCatch 来实现同步形式的异步处理。

Suspense 提供的 createFetcher 函数会封装异步操作，当尝试从 createFetcher 返回的结果读取数据时，有两种可能：一种是数据已经就绪，那就直接返回结果；还有一种可能是异步操作还没有结束，数据没有就绪，这时候 createFetcher 会抛出一个“异常”。

你可能会说，抛出异常，渲染过程不就中断了吗？

的确会中断，不过，createFetcher 抛出的这个“异常”比较特殊，这个“异常”实际上是一个 Promise 对象，这个 Promise 对象代表的就是异步操作，操作结束时，也是数据准备好的时候。当 componentDidCatch 捕获这个 Promise 类型的“异常”时，就可以根据这个 Promise 对象的状态改变来重新渲染对应组件，第二次渲染，肯定就能够成功。

下面是 createFetcher 的一个简单实现方式：

```
var NO_RESULT = {};

export const createFetcher = (task) => {
  let result = NO_RESULT;

  return () => {
    const p = task();

    p.then(res => {
      result = res;
    });


    if (result === NO_RESULT) {
      throw p;
    }

    return result;
  }
}
```

在上面的代码中，createFetcher 的参数 `task` 被调用应该返回一个 Promise 对象，这个对象在第一次调用时会被 throw 出去，但是，只要这个对象完结，那么 `result` 就有实际的值，不会再被 throw。

还需要一个和 createFetcher 配合的 Suspense，代码如下：

```
class Suspense extends React.Component {
  state = {
    pending: false
  }

  componentDidCatch(error) {
    // easy way to detect Promise type
    if (typeof error.then === 'function') {
      this.setState({pending: true});

      error.then(() => this.setState({
        pending: false
      }));
    }
  }

  render() {
    return this.state.pending ? null : this.props.children;
  }
}
```

上面的 Suspense 组件实现了 componentDidCatch，如果捕获的 `error` 是 Promise 类型，那就说明子组件用 createFetcher 获取异步数据了，就会等到它完结之后重设 state，引发一次新的渲染过程，因为 createFetcher 中会记录异步返回的结果，新的渲染就不会抛出异常了。

使用 createFetcher 和 Suspense 的示例代码如下：

```
const getName = () => new Promise((resolve) => {
  setTimeout(() => {
    resolve('Morgan');
  }, 1000);
})

const fetcher = createFetcher(getName);

const Greeting = () => {
  return <div>Hello {fetcher()}</div>
};

const SuspenseDemo = () => {
  return (
    <Suspense>
      <Greeting />
    </Suspense>
  );
};
```

上面的 `getName` 利用 `setTimeout` 模拟了异步 AJAX 获取数据，第一次渲染 `Greeting` 组件时，会有 Promise 类型的异常抛出，被 `Suspense` 捕获。1 秒钟之后，当 `getName` 返回实际结果的时候，`Suspense` 会引发重新渲染，这一次 `Greeting` 会显示出 `hello Morgan`。

上面的 createFetcher 和 Suspense 是一个非常简陋的实现，主要用来让读者了解 Suspense 的工作原理，正式发布的 Suspense 肯定会具备更强大的功能。

React v16.6.0 对 Suspense 的支持

React 发布 v16.6.0 的时候，提供了 Suspense 组件，直接支持 Suspense 功能，但是还没有正式提供 createFetcher 的功能，只发布了一个独立但不稳定的 react-cache 包。这个包里的 unstable_createResource 相当于上面描述的 createFetcher。照这个命名来看，正式发布的时候这个 API 可能会叫做 createResource 而不是叫 createFetcher。

我们利用 React v16.6.0 和不稳定的 react-cache 来实现上述功能，代码如下：

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

import {unstable_createResource as createResource} from 'react-cache';

const getName = () => new Promise((resolve) => {
  setTimeout(() => {
    resolve('Morgan');
  }, 1000);
})

const resource = createResource(getName);

const Greeting = () => {
  return <div>hello {resource.read()}</div>
};

const SuspenseDemo = () => {
  return (
    <Suspense fallback={<div>loading...</div>} >
      <Greeting />
    </Suspense>
  );
};
```

在上面的代码中，我们使用 React 提供的 Suspense 组件，支持一个 `fallback` 属性，这个属性可以用于显示“装载中”界面。在上面的例子中，要等待 1 秒钟时间才得到模拟 API 的结果，这时候显示一个空白页面是肯定不合适的，在等待的这 1 秒钟里，显得就是一个“Loading...”字样。

很显然，需要一个最佳实践来控制 Suspense 的范围。如果我们只在组件树最顶层放一个 Suspense 组件，那么在 API 返回之前，整个页面只显示“装载中”，这样的用户体验并不好。正确的做法，是将每一个独立依赖某个 API 调用的组件用一个 Suspense 包住。

例如，一个页面中包括头部的 `Header`、左侧的导航栏 `LeftPanel` 和右侧的内容 `Content`，其中只有 `Header` 的渲染不依赖于 API，那么，JSX 可以这样写：

```
<div>
    <Header />
    <Suspense fallback={<LoadingSpin />}>
        <LeftPanel />
    </Suspense>
    <Suspense fallback={<LoadingSpin />}>
        <Content />
    </Suspense>
</div>
```

这样，网页首先显示 `Header`，然后无论 `LeftPanel` 还是 `Content` 中谁的 AJAX 首先返回结果，都可以立刻显示对应模块，而不用等待所有 AJAX 都返回才让用户看到更新。

Suspense 带来的 React 使用模式改变

Suspense 被推出之后，可以极大地减少异步操作代码的复杂度。

之前，只要有 AJAX 这样的异步操作，就必须要用两次渲染来显示 AJAX 结果，这就需要用组件的 state 来存储 AJAX 的结果，用 state 又意味着要把组件实现为一个 class。总之，我们需要做这些：

1. 实现一个 class；
2. class 中需要有 state；
3. 需要实现 componentDidMount 函数；
4. render 必须要根据 `this.state` 来渲染不同内容。

有了 Suspense 之后，不需要做上面这些杂事，只要一个函数形式组件就足够了。

在介绍 Redux 时，我们提到过在 Suspense 面前，Redux 的一切异步操作方案都显得繁琐，读者现在应该能够通过代码理解这一点了。

很可惜，目前 Suspense 还不支持服务器端渲染，当 Suspense 支持服务器端渲染的时候，那就真的会对 React 社区带来革命性影响。

小结

在这一小节中我们介绍了 Suspense 功能，读者应该可以了解到：

1. Suspense 解决异步操作的问题；
2. 有了 Supsense 之后，依赖 AJAX 的组件也可以是函数形式，不需要是 class。

React 的未来（3）：函数化的 Hooks

这一节我们来介绍 Hooks，React v16.7.0-alpha 中第一次引入了 Hooks 的概念，因为这是一个 alpha 版本，不算正式发布，所以，将来正式发布时 API 可能会有变化。

Hooks 的目的，简而言之就是让开发者不需要再用 class 来实现组件。

还记得之前我们介绍的经典 Counter 组件吗？不考虑用 Redux 或者 Mobx 来管理状态的话，Counter 组件就需要把计数数据放在 state 里，要用 state，就意味着需要定义一个 class。

很多时候，一个简单组件也需要实现一个 class，的确是一件很烦的事，有了 Hooks 之后，事情就简单多了，我们用几个已经公开的 Hooks API 来看看如何避免写 class。

useState

Hooks 会提供一个叫 `useState` 的方法，它开启了一扇新的定义 state 的门，对应 Counter 的代码可以这么写：

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

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

  return (
    <div>
       <div>{count}</div>
       <button onClick={() => setCount(count + 1)}>+</button>
       <button onClick={() => setCount(count - 1)}>-</button>
    </div>
  );
};
```

注意看，Counter 拥有自己的“状态”，但它只是一个函数，不是 class。

`useState` 只接受一个参数，也就是 state 的初始值，它返回一个只有两个元素的数组，第一个元素就是 state 的值，第二个元素是更新 state 的函数。

我们利用解构赋值（destructuring assignment）把两个元素分别赋值给 count 和 setCount，相当于这样的代码：

```
  // 下面代码等同于： const [count, setCount] = useState(0);
  const result = useState(0);
  const count = result[0];
  const setCount = result[1];
```

利用 `count` 可以读取到这个 state，利用 `setCount` 可以更新这个 state，而且我们完全可以控制这两个变量的命名，只要高兴，你完全可以这么写：

```
  const [theCount, updateCount] = useState(0);
```

因为 `useState` 在 `Counter` 这个函数体中，每次 Counter 被渲染的时候，这个 `useState` 调用都会被执行，`useState` 自己肯定不是一个纯函数，因为它要区分第一次调用（组件被 mount 时）和后续调用（重复渲染时），只有第一次才用得上参数的初始值，而后续的调用就返回“记住”的 state 值。

读者看到这里，心里可能会有这样的疑问：如果组件中多次使用 `useState` 怎么办？React 如何“记住”哪个状态对应哪个变量？

React 是完全根据 `useState` 的调用顺序来“记住”状态归属的，假设组件代码如下：

```
const Counter = () => {
  const [count, setCount] = useState(0);
  const [foo, updateFoo] = useState('foo');

  ...
}
```

每一次 Counter 被渲染，都是第一次 `useState` 调用获得 `count` 和 `setCount`，第二次 `useState` 调用获得 `foo` 和 `updateFoo`（这里我故意让命名不用 `set` 前缀，可见函数名可以随意）。React 是渲染过程中的“上帝”，每一次渲染 Counter 都要由 React 发起，所以它有机会准备好一个内存记录，当开始执行的时候，每一次 useState 调用对应内存记录上一个位置，而且是按照顺序来记录的。React 不知道你把 `useState` 等 Hooks API 返回的结果赋值给什么变量，但是它也不需要知道，它只需要按照 `useState` 调用顺序记录就好了。

正因为这个原因，**Hooks，千万不要在 if 语句或者 for 循环语句中使用！**

像下面的代码，肯定会出乱子的：

```
const Counter = () => {
    const [count, setCount] = useState(0);
    if (count % 2 === 0) {
        const [foo, updateFoo] = useState('foo');
    }
    const [bar, updateBar] = useState('bar');
  ...
}
```

因为条件判断，让每次渲染中 `useState` 的调用次序不一致了，于是 React 就错乱了。

useEffect

除了 `useState`，React 还提供 `useEffect`，用于支持组件中增加副作用的支持。

在 React 组件生命周期中如果要做有副作用的操作，代码放在哪里？

当然是放在 componentDidMount 或者 componentDidUpdate 里，但是这意味着组件必须是一个 class。

在 Counter 组件，如果我们想要在用户点击“+”或者“-”按钮之后把计数值体现在网页标题上，这就是一个修改 DOM 的副作用操作，所以必须把 Counter 写成 class，而且添加下面的代码：

```
componentDidMount() {
  document.title = `Count: ${this.state.count}`;
}

componentDidUpdate() {
  document.title = `Count: ${this.state.count}`;
}
```

而有了 `useEffect`，我们就不用写一个 class 了，对应代码如下：

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

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

  useEffect(() => {
    document.title = `Count: ${count}`;
  });

  return (
    <div>
       <div>{count}</div>
       <button onClick={() => setCount(count + 1)}>+</button>
       <button onClick={() => setCount(count - 1)}>-</button>
    </div>
  );
};
```

`useEffect` 的参数是一个函数，组件每次渲染之后，都会调用这个函数参数，这样就达到了 componentDidMount 和 componentDidUpdate 一样的效果。

虽然本质上，依然是 componentDidMount 和 componentDidUpdate 两个生命周期被调用，但是现在我们关心的不是 mount 或者 update 过程，而是“after render”事件，`useEffect` 就是告诉组件在“渲染完”之后做点什么事。

读者可能会问，现在把 componentDidMount 和 componentDidUpdate 混在了一起，那假如某个场景下我只在 mount 时做事但 update 不做事，用 useEffect 不就不行了吗？

其实，用一点小技巧就可以解决。`useEffect` 还支持第二个可选参数，只有同一 `useEffect` 的两次调用第二个参数不同时，第一个函数参数才会被调用，所以，如果想模拟 `componentDidMount`，只需要这样写：

```
  useEffect(() => {
    // 这里只有mount时才被调用，相当于componentDidMount
  }, [123]);
```

在上面的代码中，`useEffect` 的第二个参数是 `[123]`，其实也可以是任何一个常数，因为它永远不变，所以 `useEffect` 只在 mount 时调用第一个函数参数一次，达到了 componentDidMount 一样的效果。

useContext

在前面介绍“提供者模式”章节我们介绍过 React 新的 Context API，这个 API 不是完美的，在多个 Context 嵌套的时候尤其麻烦。

比如，一段 JSX 如果既依赖于 ThemeContext 又依赖于 LanguageContext，那么按照 React Context API 应该这么写：

```
<ThemeContext.Consumer>
    {
        theme => (
            <LanguageContext.Cosumer>
                language => {
                    //可以使用theme和lanugage了
                }
            </LanguageContext.Cosumer>
        )
    }
</ThemeContext.Consumer>
```

因为 Context API 要用 render props，所以用两个 Context 就要用两次 render props，也就用了两个函数嵌套，这样的缩格看起来也的确过分了一点点。

使用 Hooks 的 `useContext`，上面的代码可以缩略为下面这样：

```
const theme = useContext(ThemeContext);
const language = useContext(LanguageContext);
// 这里就可以用theme和language了
```

这个`useContext`把一个需要很费劲才能理解的 Context API 使用大大简化，不需要理解render props，直接一个函数调用就搞定。

但是，`useContext`也并不是完美的，它会造成意想不到的重新渲染，我们看一个完整的使用`useContext`的组件。

```
const ThemedPage = () => {
    const theme = useContext(ThemeContext);

    return (
       <div>
            <Header color={theme.color} />
            <Content color={theme.color}/>
            <Footer color={theme.color}/>
       </div>
    );
};
```

因为这个组件`ThemedPage`使用了`useContext`，它很自然成为了Context的一个消费者，所以，只要Context的值发生了变化，`ThemedPage`就会被重新渲染，这很自然，因为不重新渲染也就没办法重新获得`theme`值，但现在有一个大问题，对于ThemedPage来说，实际上只依赖于`theme`中的`color`属性，如果只是`theme`中的`size`发生了变化但是`color`属性没有变化，`ThemedPage`依然会被重新渲染，当然，我们通过给`Header`、`Content`和`Footer`这些组件添加`shouldComponentUpdate`实现可以减少没有必要的重新渲染，但是上一层的`ThemedPage`中的JSX重新渲染是躲不过去了。

说到底，`useContext`需要一种表达方式告诉React："我没有改变，重用上次内容好了。"

希望Hooks正式发布的时候能够弥补这一缺陷。

Hooks 带来的代码模式改变

上面我们介绍了 `useState`、`useEffect` 和 `useContext` 三个最基本的 Hooks，可以感受到，Hooks 将大大简化使用 React 的代码。

首先我们可能不再需要 class了，虽然 React 官方表示 class 类型的组件将继续支持，但是，业界已经普遍表示会迁移到 Hooks 写法上，也就是放弃 class，只用函数形式来编写组件。

对于 `useContext`，它并没有为消除 class 做贡献，却为消除 render props 模式做了贡献。很长一段时间，高阶组件和 render props 是组件之间共享逻辑的两个武器，但如同我前面章节介绍的那样，这两个武器都不是十全十美的，现在 Hooks 的出现，也预示着高阶组件和 render props 可能要被逐步取代。

但读者朋友，不要觉得之前学习高阶组件和 render props 是浪费时间，相反，你只有明白 React 的使用历史，才能更好地理解 Hooks 的意义。

可以预测，在 Hooks 兴起之后，共享代码之间逻辑会用函数形式，而且这些函数会以 `use-` 前缀为约定，重用这些逻辑的方式，就是在函数形式组件中调用这些 `useXXX` 函数。

例如，我们可以写这样一个共享 Hook `useMountLog`，用于在 mount 时记录一个日志，代码如下：

```
const useMountLog = (name) => {
    useEffect(() => {
        console.log(`${name} mounted`);
    }, [123]);
}
```

任何一个函数形式组件都可以直接调用这个 `useMountLog` 获得这个功能，如下：

```
const Counter = () => {
    useMountLog('Counter');

    ...
}
```

对了，所有的 Hooks API 都只能在函数类型组件中调用，class 类型的组件不能用，从这点看，很显然，class 类型组件将会走向消亡。

小结

这一节我们介绍了 React Hooks，读者应该能够理解：

1. Hooks 的意义就是可以淘汰 class 类型的组件；
2. Hooks 将改变重用组件逻辑的模式；
3. 在未来，Hooks 将是 React 使用的主流。

结语

这本小册介绍的是 React 设计模式和最佳实践，所以在最后我们还是要讲一讲“设计模式”和“最佳实践”。

模式的进化

React 社区的重要贡献者、同时也是 react-router 的作者之一 的 Ryan Florence 说过 这么一句话。

翻译过来就是：“如果有很多所谓‘模式’出现，那意味着应该有很多第一公民地位的 API。”

这话什么意思呢？也许读者你看完这本小册心里已经有了答案。

平台开发者提供通用的方案，但应用开发者往往更接地气，更了解有哪些问题需要被解决，所以会超前于平台发明各种模式。

在 React 发展的过程中，很多场景并没有提供完善的解决方法，于是出现高阶组件、render props、组合组件等等各种“设计模式”，但是，React 也在不断完善自身，你看，当 Hooks 出现之后，原有的很多模式就无用武之地。

这是因为平台开发者会接受应用开发者的反馈，把需求抽象出来，在“设计模式”的基础上提供更好的解法，Hooks 这种第一公民地位的 API 就是一个很好的例子。

平台开发者和应用开发者，相辅相成，React 社区就是最好的例证。

最佳实践的先决条件

虽然我们经常说“最佳实践”，但是我们要明白，这世界上其实没有绝对的“最佳”，任何“最佳实践”，都是针对于某一个特定场景，目前的技术发展水平下是最佳，过一段时间未必是最佳，在这个应用中是最佳，在另一个类型的应用中未必是最佳。

不只是 React 相关技术，所有的“最佳实践”都是这样的。比如，在某些公司里要求代码要做严格审核，可以让代码质量百尺竿头更进一步，但是，在有些公司里开发周期很短，严格的代码审核会制造很多噪音，延误开发流程，可能只能是添乱。

看这本小册的朋友，有的将来会走上管理岗位，更应该明白，这世界上没有绝对的“最佳实践”，一切都要因地制宜，不同问题用不同方法对待。

回到 React 技术上来，最佳实践也是变化的，曾几何时高阶组件和 render props 被认为是“最佳实践”，但是随着 Suspense 和 Hooks 等技术的推广，很可能以前的“最佳实践”会退出历史舞台。这并不表示以前的“最佳实践”没有价值，恰恰相反，正是开发者们不断总结的“最佳实践”推动了行业和技术的发展。

最后的话

技术在不断进步，时代在不断进步，设计模式和最佳实践也在不断进步，我们开发者自身也应该不断进步。

作为一个 React 开发者，你很幸运，因为 React 技术和社区也在与时俱进不断发展，最后祝大家工作顺利一路进步！

再见！