第 4 章

深入理解组件

在第 2 章中已经介绍了 React 组件的基本用法，本章将继续探究 React 组件，从组件的 state、组件与服务器通信、组件通信、组件的 ref 属性 4 个方面深入讲解组件，帮助读者在项目中正确地使用组件。

4.1 组件 state

4.1.1 设计合适的 state

组件 state 必须能代表一个组件 UI 呈现的完整状态集，即组件的任何 UI 改变都可以从 state 的变化中反映出来；同时，state 还必须代表一个组件 UI 呈现的最小状态集，即 state 中的所有状态都用于反映组件 UI 的变化，没有任何多余的状态，也不应该存在通过其他状态计算而来的中间状态。

我们通过一个例子来解释上面的定义。假设需要开发一个购物车组件，需要展示的信息有购买的物品列表以及物品的总金额。设计一个错误的 state：

```
// 错误的 state 示例
{
  purchaseList:[],
  totalCost: 0
}
```

这里的 state 是初始状态，因此 purchaseList 初始化为一个空数组，totalCost 初始化为 0，这个

state 的设计确实可以满足组件 UI 呈现的完整状态集这一条件，但是它包含一个无用的状态 totalCost，因为 totalCost 可以根据购买的每一项物品的价格和数量计算得出，所以有了 purchaseList，就可以计算出 totalCost，totalCost 属于中间状态，可以省略。

state 所代表的一个组件 UI 呈现的完整状态集又可以分成两类数据：用作渲染组件时使用到的数据的来源以及用作组件 UI 展现形式的判断依据。例如，下面的 Hello 组件定义了 user 和 display 作为组件 state，user 是组件最终要在界面上呈现的数据，而 display 决定了<h1>标签是否需要渲染，是组件 UI 呈现形式的判断依据。

```
class Hello extends React.Component {
  constructor(props) {
    super(props);
    this.state = {
      user : 'React',
      display: true
    }
  }

  render() {
    return (
      <div>
        {
          this.state.display ?
          <h1>Hello, {this.state.user}</h1> : null
        }
      </div>
    );
  }
}
```

state 还容易和 props 以及组件的普通属性混淆。这是我们第一次提到组件的普通属性，所以先明确一下组件普通属性的定义。我们的组件都是使用 ES6 的 class 定义的，所以组件的属性其实也就是 class 的属性（更确切的说法是 class 实例化对象的属性，但因为 JavaScript 本质上是没有类的定义的，class 只不过是 ES6 提供的语法糖，所以这里模糊化类和对象的区别）。在 ES6 中，可以使用 this.{属性名}定义一个 class 的属性，也可以说属性是直接挂载到 this 下的变量。因此，state、props 实际上也是组件的属性，只不过它们是 React 为我们在 Component class 中预定义好的属性。除了 state、props 以外的其他组件属性称为组件的普通属性。

假设一个组件需要显示当前时间，并且这个时间每秒都会自动更新，这个组件内就需要定义一个计时器，在这个计时器中每隔 1 秒更新一次组件的 state。这个计时器变量并不适合定义到组件的 state 中，因为它并不代表组件 UI 呈现状态，它只是用来更改组件的 state，这时就到了组件的普通属性发挥作用的时候了。例如，下面的代码为 Hello 组件定义了 timer 属性，用来定时更新组件状态：

```
class Hello extends React.Component {
  constructor(props) {
    super(props);
    this.timer = null;   //普通属性
    this.state = {
      date : new Date()
    }
    this.updateDate = this.updateDate.bind(this);
  }

  componentDidMount() {
    this.timer = setInterval(this.updateDate, 1000)
  }

  componentWillUnmount() {
    clearInterval(this.timer);
  }

  updateDate(){
    this.setState({
      date: new Date()
    });
  }

  render() {
    return (
      <div>
        <h1>Hello</h1>
        <h1>{this.state.date.toString()}</h1>
      </div>
    );
  }
}
```

因此，当我们在组件中需要用到一个变量，并且它与组件的渲染无关时，就应该把这个变量定义为组件的普通属性，直接挂载到 this 下，而不是作为组件的 state。还有一个更加直观的判断方法，就是看组件 render 方法中有没有使用到这个变量，如果没有，它就是一个普通属性。

state 和 props 又有什么区别呢？state 和 props 都直接和组件的 UI 渲染有关，它们的变化都会触发组件重新渲染，但 props 对于使用它的组件来说是只读的，是通过父组件传递过来的，要想修改 props，只能在父组件中修改；而 state 是组件内部自己维护的状态，是可变的。

总结一下，组件中用到的一个变量是不是应该作为 state 可以通过下面的 4 条依据进行判断：

（1）这个变量是否通过 props 从父组件中获取？如果是，那么它不是一个状态。

（2）这个变量是否在组件的整个生命周期中都保持不变？如果是，那么它不是一个状态。

（3）这个变量是否可以通过其他状态（state）或者属性（props）计算得到？如果是，那么它不是一个状态。

（4）这个变量是否在组件的 render 方法中使用？如果**不是**，那么它不是一个状态。这种情况下，这个变量更适合定义为组件的一个普通属性。

4.1.2　正确修改 state

state 可以通过 this.state.{属性}的方式直接获取，但当修改 state 时，往往有很多陷阱需要注意。下面介绍常见的三种陷阱：

1. 不能直接修改 state

直接修改 state，组件并不会重新触发 render。例如：

```
// 错误
this.state.title = 'React';
```

正确的修改方式是使用 setState()：

```
// 正确
this.setState({title: 'React'});
```

2. state 的更新是异步的

调用 setState 时，组件的 state 并不会立即改变，setState 只是把要修改的状态放入一个队列中，React 会优化真正的执行时机，并且出于性能原因，可能会将多次 setState 的状态修改合并成一次状态修改。所以不要依赖当前的 state，计算下一个 state。当真正执行状态修改时，依赖的 this.state 并不能保证是最新的 state，因为 React 会把多次 state 的修改合并成一次，这时 this.state 还是这几次 state 修改前的 state。另外，需要注意的是，同样不能依赖当前的 props 计算下一个状态，因为 props 的更新也是异步的。

举个例子，对于一个电商类应用，在购物车中，点击一次购买数量按钮，购买的数量就会加 1，如果连续点击两次按钮，就会连续调用两次 this.setState({quantity: this.state.quantity + 1})，在 React 合并多次修改为一次的情况下，相当于等价执行了如下代码：

```
Object.assign(
  previousState,
  {quantity: this.state.quantity + 1},
  {quantity: this.state.quantity + 1}
)
```

于是，后面的操作覆盖前面的操作，最终购买的数量只增加 1。

如果有这样的需求，可以使用另一个接收一个函数作为参数的 setState，这个函数有两个参数，第一个是当前最新状态（本次组件状态修改生效后的状态）的前一个状态 preState（本次组件状态修改前的状态），第二个参数是当前最新的属性 props。代码如下：

```
// 正确
this.setState((preState, props) => ({
  counter: preState.quantity + 1;
}))
```

3. state 的更新是一个合并的过程

当调用 setState 修改组件状态时，只需要传入发生改变的 state，而不是组件完整的 state，因为组件 state 的更新是一个合并的过程。例如，一个组件的状态为：

```
this.state = {
  title : 'React',
  content : 'React is an wonderful JS library!'
}
```

当只需要修改状态 title 时，将修改后的 title 传给 setState 即可：

```
this.setState({title: 'Reactjs'});
```

React 会合并新的 title 到原来的组件状态中，同时保留原有的状态 content，合并后的 state 为：

```
{
  title : 'Reactjs',
  content : 'React is an wonderful JS library!'
}
```

4.1.3 state 与不可变对象

React 官方建议把 state 当作不可变对象，一方面，直接修改 this.state，组件并不会重新 render；另一方面，state 中包含的所有状态都应该是不可变对象。当 state 中的某个状态发生变化时，应该重新创建这个状态对象，而不是直接修改原来的状态。那么，当状态发生变化时，如何创建新的状态呢？根据状态的类型可以分成以下三种情况：

1. 状态的类型是不可变类型（数字、字符串、布尔值、null、undefined）

这种情况最简单，因为状态是不可变类型，所以直接给要修改的状态赋一个新值即可。例如要修改 count（数字类型）、title（字符串类型）、success（布尔类型）三个状态：

```
this.setState({
  count: 1,
  title: 'React',
  success: true
})
```

2. 状态的类型是数组

例如有一个数组类型的状态 books，当向 books 中增加一本书时，可使用数组的 concat 方法或 ES6 的数组扩展语法（spread syntax）：

```
// 方法一：使用 preState、concat 创建新数组
this.setState(preState => ({
  books: preState.books.concat(['React Guide']);
}))

// 方法二：ES6 spread syntax
this.setState(preState => ({
  books: [...preState.books, 'React Guide'];
}))
```

当从 books 中截取部分元素作为新状态时，可使用数组的 slice 方法：

```
this.setState(preState => ({
  books: preState.books.slice(1,3);
}))
```

当从 books 中过滤部分元素后，作为新状态时，可使用数组的 filter 方法：

```
this.setState(preState => ({
  books: preState.books.filter(item => {
    return item !== 'React';
  });
}))
```

注意，不要使用 push、pop、shift、unshift、splice 等方法修改数组类型的状态，因为这些方法都是在原数组的基础上修改的，而 concat、slice、filter 会返回一个新的数组。

3. 状态的类型是普通对象（不包含字符串、数组）

（1）使用 ES6 的 Object.assgin 方法：

```
this.setState(preState => ({
  owner: Object.assign({}, preState.owner, {name: 'Jason'});
}))
```

（2）使用对象扩展语法（object spread properties）：

```
this.setState(preState => ({
  owner: {...preState.owner, name: 'Jason'};
}))
```

总结一下，创建新的状态对象的关键是，避免使用会直接修改原对象的方法，而是使用可以返回一个新对象的方法。当然，也可以使用一些 Immutable 的 JS 库（如 Immutable.js）实现类似的效果。

为什么 React 推荐组件的状态是不可变对象呢？一方面是因为对不可变对象的修改会返回一个新对象，不需要担心原有对象在不小心的情况下被修改导致的错误，方便程序的管理和调试；另一方面是出于性能考虑，当对象组件状态都是不可变对象时，在组件的 shouldComponentUpdate 方法中仅需要比较前后两次状态对象的引用就可以判断状态是否真的改变，从而避免不必要的 render 调用。在第 5 章会详细介绍这部分内容。

4.2 组件与服务器通信

React 关注的是 UI 的分离、视图的组件化，对于组件如何与服务器端 API 通信，React 官方并没有给出太多指导。但是，几乎所有应用都避免不了和服务器端 API 通信。这就给很多 React 的使用者带来了困惑，React 中的组件到底应该如何优雅地和服务器通信呢？本节将结合实践对这个问题的解决方法给出建议。

首先需要明确一点，本节讨论的组件与服务器通信特指组件从服务器上获取数据，不包含组件向服务器提交数据的情况。组件向服务器提交数据一定是由组件 UI 的某一事件触发的，比如提交了一个表单、点击了一个元素等，所以只要在监听相应事件的回调函数中执行向服务器提交数据的逻辑即可，一般不会有疑问。但组件从服务器上获取数据，情况就要复杂得多。

4.2.1 组件挂载阶段通信

React 组件的正常运转本质上是组件不同生命周期方法的有序执行，因此组件与服务器的通信也必定依赖组件的生命周期方法。我们先来看一下组件在挂载阶段如何与服务器通信。

定义一个 UserListContainer 组件，需要从服务器获取用户列表：

```
class UserListContainer extends React.Component{

 /** 省略无关代码 **/

  componentDidMount() {
    var that = this;
    fetch('/path/to/user-api').then(function(response) {
      response.json().then(function(data) {
        that.setState({users: data})
      });
    });
  }
}
```

UserListContainer 是在 componentDidMount 中与服务器进行通信的，这时候组件已经挂载，真实 DOM 也已经渲染完成，是调用服务器 API 最安全的地方，也是 React 官方推荐的进行服务器通信的地方。

除了 componentDidMount 外，在 componentWillMount 中进行服务器通信也是比较常见的一种方式。代码如下：

```
class UserListContainer extends React.Component{

 /** 省略无关代码 **/
```

```
    componentWillMount() {
      var that = this;
      fetch('/path/to/user-api').then(function(response) {
        response.json().then(function(data) {
          that.setState({users: data})
        });
      });
    }
  }
```

componentWillMount 会在组件被挂载前调用，因此从时间上来讲，在 componentWillMount 中执行服务器通信要早于在 componentDidMount 中执行，执行得越早意味着服务器数据越能更快地返回组件。这也是很多人青睐在 componentWillMount 中执行服务器通信的重要原因。但实际上，componentWillMount 与 componentDidMount 执行的时间差微乎其微，完全可以忽略不计。

componentDidMount 是执行组件与服务器通信的最佳地方，原因主要有两个：

（1）在 componentDidMount 中执行服务器通信可以保证获取到数据时，组件已经处于挂载状态，这时即使要直接操作 DOM 也是安全的，而 componentWillMount 无法保证这一点。

（2）当组件在服务器端渲染时（本书不涉及服务器渲染内容），componentWillMount 会被调用两次，一次是在服务器端，另一次是在浏览器端，而 componentDidMount 能保证在任何情况下只会被调用一次，从而不会发送多余的数据请求。

有些开发人员会在组件的构造函数中执行服务器通信，一般情况下，这种方式也可以正常工作。但是，构造函数的意义是执行组件的初始化工作，如设置组件的初始状态，并不适合做数据请求这类有“副作用”的工作。因此，不推荐在构造函数中执行服务器通信。

4.2.2　组件更新阶段通信

组件在更新阶段常常需要再次与服务器通信，获取服务器上的最新数据。例如，组件需要以 props 中的某个属性作为与服务器通信时的请求参数，当这个属性值发生更新时，组件自然需要重新与服务器通信。回想 2.3 节中对组件生命周期的介绍，不难发现 componentWillReceiveProps 非常适合做这个工作。假设 UserListContainer 在获取用户列表时还需要一个参数 category，用来根据用户的职业做筛选，category 这个参数是从 props 中获取的，实现代码如下：

```
class UserListContainer extends React.Component{

  /** 省略无关代码 **/

  componentWillReceiveProps(nextProps) {
    if(nextProps.category !== this.props.category) {
      fetch('/path/to/user-api?category='+ nextProps.category).
then(function(response) {
        response.json().then(function(data) {
```

```
          that.setState({users: data})
        });
      });
    }
  }

}
```

这里还有一个地方要注意，在执行 fetch 请求时，要先对新老 props 中的 category 做比较，只有不一致才说明 category 有了更新，才需要重新进行服务器通信。componentWillReceiveProps 的执行并不能保证 props 一定发生了修改。

4.3 组件通信

一个 React 应用是由许多个组件像搭积木一样搭建而成的，只要应用不是完全由展示组件组成的，组件之间就难免需要进行通信。其实，前面一些内容已经涉及组件的通信，只是我们并没有刻意强调这个概念。下面系统地介绍组件是如何进行通信的。

4.3.1 父子组件通信

父子组件通信是最常见的通信形式，例如 4.2 节的 UserListContainer 组件获取到的用户数据需要通过 UserList 组件展示，这时 UserListContainer 和 UserList 就存在父子组件通信。UserListContainer 作为父组件，将获取到的用户信息通过子组件 UserList 的 props 传递给 UserList。所以父组件向子组件通信是通过父组件向子组件的 props 传递数据完成的。代码如下：

```
class UserList extends React.Component{

  render() {
    return (
      <div>
        <ul className="user-list">
          {this.props.users.map(function(user) {
            return (
              <li key={user.id}>
                <span>{user.name}</span>
              </li>
            );
          })}
        </ul>
      </div>
    )
  }
```

```
}

import UserList from './UserList'

class UserListContainer extends React.Component{

  constructor(props){
    super(props);
    this.state = {
      users: []
    }
  }

  componentDidMount() {
    var that = this;
    fetch('/path/to/user-api').then(function(response) {
      response.json().then(function(data) {
        that.setState({users: data})
      });
    });
  }

  render() {
    return (
      {/* 通过 props 传递 users */}
      <UserList users={this.state.users} />
    )
  }
}
```

当子组件需要向父组件通信时，又该怎么做呢？答案依然是 props。父组件可以通过子组件的 props 传递给子组件一个回调函数，子组件在需要改变父组件数据时，调用这个回调函数即可。下面为 UserList 再增加一个添加新用户的功能：

```
class UserList extends React.Component{

  constructor(props){
    super(props);
    this.state = {
      newUser : ''
    };
    this.handleChange = this.handleChange.bind(this);
    this.handleClick = this.handleClick.bind(this);
  }
```

```
  handleChange(e) {
    this.setState({newUser: e.target.value});
  }
  // 通过 props 调用父组件的方法新增用户
  handleClick() {
    if(this.state.newUser && this.state.newUser.length > 0) {
      this.props.onAddUser(this.state.newUser);
    }
  }

  render() {
    return (
      <div>
        <ul className="user-list">
          {this.props.users.map(function(user) {
            return (
              <li key={user.id}>
                <span>{user.name}</span>
              </li>
            );
          })}
        </ul>
        <input onChange={this.handleChange} value={this.state.newUser} />
        <button onClick={this.handleClick}>新增</button>
      </div>
    )
  }
}
```

在 input 内输入新用户的名称，然后点击新增按钮，调用 handleClick 方法，在 handleClick 内部，调用通过 props 传递过来的 onAddUser 执行保存用户的逻辑。下面来看一下 onAddUser 在 UserListContainer 中的实现：

```
import UserList from './UserList'

class UserListContainer extends React.Component{

  constructor(props){
    super(props);
    this.state = {
      users: []
    }
    this.handleAddUser = this.handleAddUser.bind(this);
  }

  componentDidMount() {
```

```
    var that = this;
    fetch('/path/to/user-api').then(function(response) {
      response.json().then(function(data) {
        that.setState({users: data})
      });
    });
  }
  // 新增用户
  handleAddUser(user) {
    var that = this;
    fetch('/path/to/save-user-api',{
      method: 'POST',
      body: JSON.stringify({'username':user})
    }).then(function(response) {
      response.json().then(function(newUser) {
        // 将服务器端返回的新用户添加到 state 中
        that.setState((preState) => ({users: preState.users.concat
([newUser])}))
      });
    });
  }

  render() {
    return (
      {/* 通过 props 传递 users 和 handleAddUser 方法 */}
      <UserList users={this.state.users}
         onAddUser={this.handleAddUser}
      />
    )
  }
}
```

子组件 UserList 通过调用 props.onAddUser 方法成功地将待新增的用户传递给父组件 UserListContainer 的 handleAddUser 方法执行保存操作，保存成功后，UserListContainer 会更新状态 users，从而又将最新的用户列表传递给 UserList。这一过程既包含子组件到父组件的通信，又包含父组件到子组件的通信，而通信的桥梁就是通过 props 传递的数据和回调方法。

4.3.2　兄弟组件通信

当两个组件不是父子关系但有相同的父组件时，称为兄弟组件。注意，这里的兄弟组件在整个组件树上并不一定处于同一层级，如图 4-1 所示的两种情况，B 和 C 都是兄弟组件，因为他们都有相同的父组件 A。

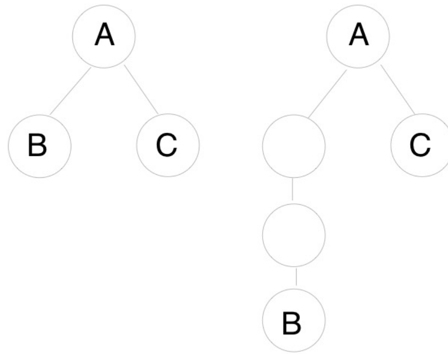

图 4-1

兄弟组件不能直接相互传送数据，需要通过状态提升的方式实现兄弟组件的通信，即把组件之间需要共享的状态保存到距离它们最近的共同父组件内，任意一个兄弟组件都可以通过父组件传递的回调函数来修改共享状态，父组件中共享状态的变化也会通过 props 向下传递给所有兄弟组件，从而完成兄弟组件之间的通信。

我们在 UserListContainer 中新增一个子组件 UserDetail，用于显示当前选中用户的详细信息，比如用户的年龄、联系方式、家庭地址等。这时，UserList 和 UserDetail 就成了兄弟组件，UserListContainer 是它们的共同父组件。当用户在 UserList 中点击一条用户信息时，UserDetail 需要同步显示该用户的详细信息，因此，可以把当前选中的用户 currentUser 保存到 UserListContainer 的状态中。

先来修改 UserList 组件：

```
class UserList extends React.Component{

  constructor(props){
    super(props);
    this.state = {
      newUser : ''
    };
    this.handleChange = this.handleChange.bind(this);
    this.handleClick = this.handleClick.bind(this);
  }

  handleChange(e) {
    this.setState({newUser: e.target.value});
  }
  // 通过 props 调用父组件的方法新增用户
  handleClick() {
    if(this.state.newUser && this.state.newUser.length > 0) {
      this.props.onAddUser(this.state.newUser);
    }
  }
  // 通过 props 调用父组件的方法，设置当前选中的用户
```

```
  handleUserClick(userId) {
    this.props.onSetCurrentUser(userId);
  }

  render() {
    return (
      <div>
        <ul className="user-list">
          {this.props.users.map((user) => {
            return (
              <li key={user.id}
           {/*使用不同样式显示当前用户 */}
                className={(this.props.currentUserId === user.id) ? 'current' : ''}
                onClick={this.handleUserClick.bind(this, user.id)}>
                <span>{user.name}</span>
              </li>
            );
          })}
        </ul>
        <input onChange={this.handleChange} value={this.state.newUser} />
        <button onClick={this.handleClick}>新增</button>
      </div>
    )
  }
}
```

我们为 UserList 添加了处理点击用户项的回调函数 handleUserClick，还为当前处于选中状态的用户项添加了名为 current 的样式。

再来创建 UserDetail 组件：

```
function UserDetail(props) {
  return (
      <div>
        {props.currentUser ?
          (<div>用户姓名：{props.currentUser.name}</div>
          <div>用户年龄：{props.currentUser.age}</div>
          <div>用户联系方式：{props.currentUser.phone}</div>
          <div>家庭地址：{props.currentUser.address}</div>)
          : ''
        }
      </div>
      )
}
```

UserDetail 不需要维护自己的状态，因此最适合用函数组件来实现。

最后修改 UserListContainer 组件：

```
import UserList from './UserList'
import UserDetail from './UserDetail'

class UserListContainer extends React.Component{

  constructor(props){
    super(props);
    this.state = {
      users: [],
      currentUserId: null
    }
    this.handleAddUser = this.handleAddUser.bind(this);
    this.handleSetCurrentUser = this.handleSetCurrentUser.bind(this);
  }

  componentDidMount() {
    var that = this;
    fetch('/path/to/user-api').then(function(response) {
      response.json().then(function(data) {
        that.setState({users: data})
      });
    });
  }
  // 新增用户
  handleAddUser(user) {
    var that = this;
    fetch('/path/to/save-user-api',{
      method: 'POST',
      body: JSON.stringify({'username':user})
    }).then(function(response) {
      response.json().then(function(newUser) {
        that.setState((preState) => ({users: preState.users.concat
([newUser])}))
      });
    });
  }
  // 设置当前选中的用户
  handleSetCurrentUser(userId) {
    this.setState({
      currentUserId : userId
    });
  }
```

```
    render() {
    // 根据 currentUserId，筛选出当前用户对象
      const filterUsers = this.state.users.filter((user) => {user.id ===
this.state.currentUserId});
      const currentUser = filterUsers.length > 0 ? filterUsers[0] : null;
      return (
        <UserList users={this.state.users}
          currentUserId = {this.state.currentUserId}
          onAddUser = {this.handleAddUser}
          onSetCurrentUser = {this.handleSetCurrentUser}
        />
        <UserDetail currentUser = {currentUser} />
      )
    }
  }
```

UserListContainer 新增状态 currentUserId 用来标识当前选中的用户，这个状态正是 UserList 和 UserDetail 两个组件都要用到的状态，通过状态提升保存到它们共同的父组件 UserListContainer 中。同时，UserListContainer 通过 UserList 的 props 将修改 currentUserId 的回调函数传递给 UserList，使 UserList 可以在自身内部修改 currentUserId。

4.3.3　Context

当组件所处层级太深时，往往需要经过很多层的 props 传递才能将所需的数据或者回调函数传递给使用组件。这时，以 props 作为桥梁的组件通信方式便会显得很烦琐。例如，我们把 UserList 中新增用户的工作单独拆分到一个新的组件 UserAdd 中：

```
  class UserAdd extends React.Component{

    constructor(props){
      super(props);
      this.state = {
        newUser : ''
      };
      this.handleChange = this.handleChange.bind(this);
      this.handleClick = this.handleClick.bind(this);
    }

    handleChange(e) {
      this.setState({newUser: e.target.value});
    }
    // 通过 props 调用父组件的方法新增用户
    handleClick() {
      if(this.state.newUser && this.state.newUser.length > 0) {
        this.props.onAddUser(this.state.newUser);
```

```
    }
  }

  render() {
    return (
      <div>
        <input onChange={this.handleChange} value={this.state.newUser} />
        <button onClick={this.handleClick}>新增</button>
      </div>
    )
  }
}
```

同时，修改原有的 UserList 组件：

```
import UserAdd from './UserAdd'

class UserList extends React.Component{
  // 通过 props 调用父组件的方法，设置当前用户
  handleUserClick(userId) {
    this.props.onSetCurrentUser(userId);
  }

  render() {
    return (
      <div>
        <ul className="user-list">
          {this.props.users.map((user) => {
            return (
              <li key={user.id}
                className={(this.props.currentUserId === user.id) ? 'current' : ''}
                onClick={this.handleUserClick.bind(this, user.id)}>
                <span>{user.name}</span>
              </li>
            );
          })}
        </ul>
        {/* 传递 UserListContainer 的 handleAddUser 方法 */}
        <UserAdd onAddUser = {this.props.onAddUser} />
      </div>
    )
  }
}
```

可以发现，UserListContainer 中处理添加用户的函数 handleAddUser 经过 UserList 和 UserAdd 两个层级的 props 传递才到达 UserAdd 组件中。当应用更加复杂时，组件的层级会更多，组件通信

就需要经过更多层级的传递，组件通信会变得非常麻烦。幸好，React 提供了一个 context 上下文，让任意层级的子组件都可以获取父组件中的状态和方法。创建 context 的方式是：在提供 context 的组件内新增一个 getChildContext 方法，返回 context 对象，然后在组件的 childContextTypes 属性上定义 context 对象的属性的类型信息。UserListContainer 是提供 context 的组件，改写如下：

```
class UserListContainer extends React.Component{

  /** 省略其余代码 **/

  // 创建 context 对象，包含 onAddUser 方法
  getChildContext() {
    return {onAddUser: this.handleAddUser};
  }
  // 新增用户
  handleAddUser(user) {
    this.setState((preState) => ({users: preState.users.concat([{'id':'c',
'name': 'cc' }])}))
  }

  render() {
    const filterUsers = this.state.users.filter((user) => {user.id =
this.state.currentUserId});
    const currentUser = filterUsers.length > 0 ? filterUsers[0] : null;
    return (
      <UserList users={this.state.users}
        currentUserId = {this.state.currentUserId}
        onSetCurrentUser = {this.handleSetCurrentUser}
      />
      <UserDetail currentUser = {currentUser} />
    )
  }
}

// 声明 context 的属性的类型信息
UserListContainer.childContextTypes = {
  onAddUser: PropTypes.func
};
```

UserListContainer 通过增加 getChildContext 和 childContextTypes 将 onAddUser 在组件树中自动向下传递，当任意层级的子组件需要使用时，只需要在该组件的 contextTypes 中声明使用的 context 属性即可。例如，UserAdd 需要使用 context 中的 onAddUser，代码如下：

```
class UserAdd extends React.Component{
```

```
  /**省略其余代码**/

  handleChange(e) {
    this.setState({newUser: e.target.value});
  }

  handleClick() {
    if(this.state.newUser && this.state.newUser.length > 0) {
      this.context.onAddUser(this.state.newUser);
    }
  }

  render() {
    return (
      <div>
        <input onChange={this.handleChange} value={this.state.newUser} />
        <button onClick={this.handleClick}>Add</button>
      </div>
    )
  }
}

// 声明要使用的 context 对象的属性
UserAdd.contextTypes = {
  onAddUser: PropTypes.func
};
```

增加 contextTypes 后，在 UserAdd 内部就可以通过 this.context.onAddUser 的方式访问 context 中的 onAddUser 方法。注意，这里的示例传递的是组件的方法，组件中的任意数据也可以通过 context 自动向下传递。另外，当 context 中包含数据时，如果要修改 context 中的数据，一定不能直接修改，而是要通过 setState 修改，组件 state 的变化会创建一个新的 context，然后重新传递给子组件。

虽然 context 给组件通信带来了便利，但过多使用 context 会让应用中的数据流变得混乱，而且 context 是一个实验性的 API，在未来的 React 版本中是可能被修改或者废弃的。所以，使用 context 一定要慎重。

4.3.4 延伸

前面介绍的三种组件通信方式都是依赖 React 组件自身的语法特性。其实，还有更多的方式可以来实现组件通信。我们可以使用消息队列来实现组件通信：改变数据的组件发起一个消息，使用数据的组件监听这个消息，并在响应函数中触发 setState 来改变组件状态。本质上，这是观察者模式的实现，我们可以通过引入 EventEmitter 或 Postal.js 等消息队列库完成这一过程。当应用更加复杂时，还可以引入专门的状态管理库实现组件通信和组件状态的管理，例如 Redux 和 MobX 是当前非常受欢迎的两种状态管理库。后面的章节中会对这两种状态解决方案做详细介绍。

4.4　特殊的 ref

在 2.6.2 节非受控组件中，已经使用过 ref 来获取表单元素。ref 不仅可以用来获取表单元素，还可以用来获取其他任意 DOM 元素，甚至可以用来获取 React 组件实例。在一些场景下，ref 的使用可以带来便利，例如控制元素的焦点、文本的选择或者和第三方操作 DOM 的库集成。但绝大多数场景下，应该避免使用 ref，因为它破坏了 React 中以 props 为数据传递介质的典型数据流。本节将介绍 ref 常用的使用场景。

4.4.1　在 DOM 元素上使用 ref

在 DOM 元素上使用 ref 是最常见的使用场景。ref 接收一个回调函数作为值，在组件被挂载或卸载时，回调函数会被调用，在组件被挂载时，回调函数会接收当前 DOM 元素作为参数；在组件被卸载时，回调函数会接收 null 作为参数。例如：

```
class AutoFocusTextInput extends React.Component {
  componentDidMount() {
    // 通过 ref 让 input 自动获取焦点
    this.textInput.focus();
  }

  render() {
    return (
      <div>
        <input
          type="text"
          ref={(input) => { this.textInput = input; }} />
      </div>
    );
  }
}
```

AutoFocusTextInput 中为 input 元素定义 ref，在组件挂载后，通过 ref 获取该 input 元素，让 input 自动获取焦点。如果不使用 ref，就难以实现这个功能。

4.4.2　在组件上使用 ref

React 组件也可以定义 ref，此时 ref 的回调函数接收的参数是当前组件的实例，这提供了一种在组件外部操作组件的方式。例如，在使用 AutoFocusTextInput 组件的外部组件 Container 中控制 AutoFocusTextInput：

```
class AutoFocusTextInput extends React.Component {
  constructor(props) {
```

```
    super(props);
    this.blur = this.blur.bind(this);
  }

  componentDidMount() {
    // 通过 ref 让 input 自动获取焦点
    this.textInput.focus();
  }
  // 让 input 失去焦点
  blur() {
    this.textInput.blur();
  }

  render() {
    return (
      <div>
        <input
          type="text"
          ref={(input) => { this.textInput = input; }} />
      </div>
    );
  }
}

class Container extends React.Component {
  constructor(props) {
    super(props);
    this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    // 通过 ref 调用 AutoFocusTextInput 组件的方法
    this.inputInstance.blur();
  }

  render() {
    return (
      <div>
       <AutoFocusTextInput ref={(input) => {this.inputInstance = input}}/>
       <button onClick={this.handleClick}>失去焦点</button>
      </div>
    );
  }
}
```

在 Container 组件中，我们通过 ref 获取到了 AutoFocusTextInput 组件的实例对象，并把它赋值给 Container 的 inputInstance 属性，这样就可以通过 inputInstance 调用 AutoFocusTextInput 中的 blur 方法，让已经处于获取焦点状态的 input 元素失去焦点。

注意，只能为类组件定义 ref 属性，而不能为函数组件定义 ref 属性，例如下面的写法是不起作用的：

```
function MyFunctionalComponent() {
  return <input />;
}

class Parent extends React.Component {
  render() {
    // ref 不生效
    return (
      <MyFunctionalComponent
        ref={(input) => { this.textInput = input; }} />
    );
  }
}
```

函数组件虽然不能定义 ref 属性，但这并不影响在函数组件内部使用 ref 来引用其他 DOM 元素或组件，例如下面的例子是可以正常工作的：

```
function MyFunctionalComponent() {
  let textInput = null;

  function handleClick() {
    textInput.focus();
  }

  return  (
    <div>
      <input
        type="text"
        ref={(input) => { textInput = input; }} />
      <button onClick={handleClick}>获取焦点</button>
    </div>
  )
}
```

4.4.3　父组件访问子组件的 DOM 节点

在一些场景下，我们可能需要在父组件中获取子组件的某个 DOM 元素，例如父组件需要知道这个 DOM 元素的尺寸或位置信息，这时候直接使用 ref 是无法实现的，因为 ref 只能获取子组件

的实例对象，而不能获取子组件中的某个 DOM 元素。不过，我们可以采用一种间接的方式获取子组件的 DOM 元素：在子组件的 DOM 元素上定义 ref，ref 的值是父组件传递给子组件的一个回调函数，回调函数可以通过一个自定义的属性传递，例如 inputRef，这样父组件的回调函数中就能获取到这个 DOM 元素。下面的例子中，父组件 Parent 的 inputElement 指向的就是子组件的 input 元素。

```
function Children(props) {
  // 子组件使用父组件传递的 inputRef，为 input 的 ref 赋值
  return (
    <div>
      <input ref={props.inputRef} />
    </div>
  );
}

class Parent extends React.Component {
  render() {
    // 自定义一个属性 inputRef，值是一个函数
    return (
      <Children
        inputRef={el => this.inputElement = el}
      />
    );
  }
}
```

从这个例子中还可以发现，即使子组件是函数组件，这种方式同样有效。

4.5 本章小结

本章再次讨论 React 组件。首先，我们详细介绍了组件 state，包括 state 的设计、state 的修改以及 state 和不可变对象之间的关系；接着，我们介绍了组件与服务器通信，这是初学者常常产生困惑的地方，关键是要清楚应该在组件的哪些生命周期方法中进行服务器请求，组件之间的通信桥梁是 props，要注意父子组件通信时状态提升的情况，context 虽然能简化组件的通信，但它破坏了 React 组件的数据流，使用时要慎重；最后，我们介绍了 ref 的 3 种常见的使用场景，ref 也需要避免过度使用。