React 状态管理与进阶

在前面的章节中，你已经学习了 React 中状态管理的基础知识，本章将会深入这个话题。你将学习到状态管理的最佳实践，如何去应用它们以及为什么可以考虑使用第三方的状态管理库。

状态提取

在你的应用中，只有 App 是具有状态的 ES6 组件。在该组件的方法中，包含了许多应用的状态和业务的处理逻辑。可能你已经注意到了，我们给 Table 组件传入了大量属性。而这些参数中的绝大部分只有在 Table 组件中才被用到。所以有人可能会得出“App 组件不需要知道这些参数”的结论。

整个排序功能只有在 Table 组件中用到了，你可以将其移动到 Table 组件中，因为 App 组件根本不需要了解这些信息。将子状态（substate）从一个组件移动到其他组件中的重构过程被称为*状态提取*。在这里，你想要将 App 组件中用不到的状态移动到 Table 组件中。这里的状态是从父组件到子组件向下移动。

为了能够在 Table 组件中管理状态和添加方法，需要将其改写成 ES6 类的形式。从函数式无状态组件（functional stateless component）到 ES6 类形式组件的重构非常简单明了。

函数式无状态组件形式的 Table 组件：

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

```
const Table = ({
  list,
  sortKey,
  isSortReverse,
  onSort,
  onDismiss
}) => {
  const sortedList = SORTS[sortKey](list);
  const reverseSortedList = isSortReverse
    ? sortedList.reverse()
    : sortedList;
```

```
  return(
    ...
  );
}
```

ES6 类形式的 Table 组件：

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

```
# leanpub-start-insert
class Table extends Component {
  render() {
    const {
      list,
      sortKey,
      isSortReverse,
      onSort,
      onDismiss
    } = this.props;

    const sortedList = SORTS[sortKey](list);
    const reverseSortedList = isSortReverse
      ? sortedList.reverse()
      : sortedList;

    return(
      ...
    );
  }
}
# leanpub-end-insert
```

由于想要在 Table 组件中管理状态，你需要添加构造函数和初始状态。

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

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

    this.state = {};
  }
# leanpub-end-insert

  render() {
```

```
    ...
  }
}
```

现在你可以将状态和有关排序的方法从 App 组件向下移动到 Table 组件中。

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

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

# leanpub-start-insert
    this.state = {
      sortKey: 'NONE',
      isSortReverse: false,
    };

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

# leanpub-start-insert
  onSort(sortKey) {
    const isSortReverse = this.state.sortKey === sortKey && !thi
    this.setState({ sortKey, isSortReverse });
  }
# leanpub-end-insert

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

别忘了将挪走的状态和 `onSort()` 方法从 App 组件中移除。

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

```
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      results: null,
      searchKey: '',
```

```
      searchTerm: DEFAULT_QUERY,
      error: null,
      isLoading: false,
    };

    this.setSearchTopStories = this.setSearchTopStories.bind(thi
    this.fetchSearchTopStories = this.fetchSearchTopStories.bind
    this.onDismiss = this.onDismiss.bind(this);
    this.onSearchSubmit = this.onSearchSubmit.bind(this);
    this.onSearchChange = this.onSearchChange.bind(this);
    this.needsToSearchTopStories = this.needsToSearchTopStories.
  }

  ...

}
```

除此之外，你还可以让 Table 组件更加轻量。你还可以去掉从 App 组件传入的属性，因为现在这些属性可以由 Table 组件的内部状态控制。

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

```
class App extends Component {

  ...

  render() {
# leanpub-start-insert
    const {
      searchTerm,
      results,
      searchKey,
      error,
      isLoading
    } = this.state;
# leanpub-end-insert

    ...

    return (
      <div className="page">
        ...
# leanpub-start-insert
        <Table
          list={list}
          onDismiss={this.onDismiss}
        />
```

```
# leanpub-end-insert
        ...
      </div>
    );
  }
}
```

现在你就可以使用 Table 组件内的 `onSort()` 方法和状态了。

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

```
class Table extends Component {

  ...

  render() {
# leanpub-start-insert
    const {
      list,
      onDismiss
    } = this.props;

    const {
      sortKey,
      isSortReverse,
    } = this.state;
# leanpub-end-insert

    const sortedList = SORTS[sortKey](list);
    const reverseSortedList = isSortReverse
      ? sortedList.reverse()
      : sortedList;

    return(
      <div className="table">
        <div className="table-header">
          <span style={{ width: '40%' }}>
            <Sort
              sortKey={'TITLE'}
# leanpub-start-insert
              onSort={this.onSort}
# leanpub-end-insert
              activeSortKey={sortKey}
            >
              Title
            </Sort>
          </span>
```

```
          <span style={{ width: '30%' }}>
            <Sort
              sortKey={'AUTHOR'}
# leanpub-start-insert
              onSort={this.onSort}
# leanpub-end-insert
              activeSortKey={sortKey}
            >
              Author
            </Sort>
          </span>
          <span style={{ width: '10%' }}>
            <Sort
              sortKey={'COMMENTS'}
# leanpub-start-insert
              onSort={this.onSort}
# leanpub-end-insert
              activeSortKey={sortKey}
            >
              Comments
            </Sort>
          </span>
          <span style={{ width: '10%' }}>
            <Sort
              sortKey={'POINTS'}
# leanpub-start-insert
              onSort={this.onSort}
# leanpub-end-insert
              activeSortKey={sortKey}
            >
              Points
            </Sort>
          </span>
          <span style={{ width: '10%' }}>
            Archive
          </span>
        </div>
        { reverseSortedList.map((item) =>
          ...
        )}
      </div>
    );
  }
}
```

应用应该还是可以像之前一样正常运行，但是你已经做了非常重要的重构工作。相关的逻辑代码和状态信息从 App 组件移动到了 Table 组件中，这使得

App 组件更加轻量。此外因为 Table 的排序逻辑放在了组件内部，所以它的接口也更加轻量了。

状态提取的过程也可以反过来：从子组件到父组件，这种情形被称为状态提升。想象一下，你在子组件中处理了内部的状态信息。现在为了满足新的需求，在其父组件中也显示该组件的状态信息，你需要将状态提升到父组件中。但情况还不止这些，假如你需要在子组件的兄弟组件上显示该组件的状态，你还是需要将状态提升到父组件中。在父组件中处理内部状态，同时将状态信息暴露给相关的子组件。

练习：

- 了解更多关于 React 的状态提升 的内容
- 在使用 Redux 之前学习 React这篇文章中了解更多关于状态提升的信息

再探：setState()

至此，你已经使用过 React 的 `setState()` 方法来管理组件的内部状态。你可以给该函数传入一个对象来改变部分的内部状态。

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

```
this.setState({ foo: bar });
```

但是 `setState()` 方法不仅可以接收对象。在它的第二种形式中，你还可以传入一个函数来更新状态信息。

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

```
this.setState((prevState, props) => {
  ...
});
```

为什么你会需要第二种形式呢？使用函数作为参数而不是对象，有一个非常重要的应用场景，就是当更新状态需要取决于之前的状态或者属性的时候。如果不使用函数参数的形式，组件的内部状态管理可能会引起 bug。

当更新状态需要取决于之前的状态或者属性时，为什么使用对象而不是函数会引起 bug 呢？这是因为 React 的 `setState()` 方法是异步的。React 依次执行 `setState()` 方法，最终会全部执行完毕。如果你的 `setState()`

方法依赖于之前的状态或者属性的话，有可能在按批次执行的期间，状态或者属性的值就已经被改变了。

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

```
const { fooCount } = this.state;
const { barCount } = this.props;
this.setState({ count: fooCount + barCount });
```

想象一下像 `fooCount` 和 `barCount` 这样的状态或属性，在你调用 `setState()` 方法的时候在其他地方被异步地改变了。在不断膨胀的应用中，你会有多个 `setState()` 调用。因为 `setState()` 是异步执行的，你可能像上面的例子一样，依赖了一个已经过期的值。

使用函数参数形式的话，传入 `setState()` 方法的参数是一个回调，该回调会在被执行时传入状态和属性。尽管 `setState()` 方法是异步的，但是通过回调函数，它使用的是执行那一刻的状态和属性。

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

```
this.setState((prevState, props) => {
  const { fooCount } = prevState;
  const { barCount } = props;
  return { count: fooCount + barCount };
});
```

现在让我们回到代码中来修复这个问题。我们会一起修复一个 `setState()` 依赖于状态和属性的地方，之后你就可以按照同样的方式修复代码中的其他地方。

`setSearchTopStories()` 方法依赖于之前的状态，因此它是个使用函数而不是对象作为 `setState()` 参数的绝佳例子。目前的代码片段如下。

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

```
setSearchTopStories(result) {
  const { hits, page } = result;
  const { searchKey, results } = this.state;

  const oldHits = results && results[searchKey]
    ? results[searchKey].hits
    : [];
```

```
  const updatedHits = [
    ...oldHits,
    ...hits
  ];

  this.setState({
    results: {
      ...results,
      [searchKey]: { hits: updatedHits, page }
    },
    isLoading: false
  });
}
```

你从 state 变量中提取了一些值，但是更新状态时异步地依赖于之前的状态。现在你可以使用函数参数的形式来防止脏状态信息造成的 bug。

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

```
setSearchTopStories(result) {
  const { hits, page } = result;

# leanpub-start-insert
  this.setState(prevState => {
    ...
  });
# leanpub-end-insert
}
```

你可以将已经实现的逻辑移动到函数内部，只需将在 `this.state` 上的操作改为 `prevState` 。

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

```
setSearchTopStories(result) {
  const { hits, page } = result;

  this.setState(prevState => {
# leanpub-start-insert
    const { searchKey, results } = prevState;

    const oldHits = results && results[searchKey]
      ? results[searchKey].hits
      : [];
```

```
    const updatedHits = [
      ...oldHits,
      ...hits
    ];

    return {
      results: {
        ...results,
        [searchKey]: { hits: updatedHits, page }
      },
      isLoading: false
    };
# leanpub-end-insert
  });
}
```

如此可以修复脏状态所导致的问题。还有一个可以改进的地方，由于它是一个函数，你可以将该函数提取出来从而改善代码的可读性。这是使用函数参数形式相对于对象形式的另一个好处，该函数可以独立于组件。但是你需要使用一个高阶函数并将 `result` 传给它。毕竟，你是想根据 API 的获取结果来更新状态。

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

```
setSearchTopStories(result) {
  const { hits, page } = result;
  this.setState(updateSearchTopStoriesState(hits, page));
}
```

`updateSearchTopStoriesState()` 是一个高阶函数，因为它返回一个函数。你可以在 App 组件之外定义这个高阶函数。请注意现在函数的签名有了一些变化。

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

```
# leanpub-start-insert
const updateSearchTopStoriesState = (hits, page) => (prevState)
  const { searchKey, results } = prevState;

  const oldHits = results && results[searchKey]
    ? results[searchKey].hits
    : [];
```

```
    const updatedHits = [
      ...oldHits,
      ...hits
    ];

    return {
      results: {
        ...results,
        [searchKey]: { hits: updatedHits, page }
      },
      isLoading: false
    };
  };
# leanpub-end-insert

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

搞定！`setState()` 中函数参数形式相比于对象参数来说，在预防潜在 bug 的同时，还可以提高代码的可读性和可维护性。此外，它可以在 App 组件之外进行测试。你可以将其导出并写个测试来当作练习。

练习：

- 了解更多关于在 React 中正确使用 state 的内容
- 将所有使用 `setState()` 方法的地方重构为函数参数形式
- 只重构那些需要的地方，即依赖于之前的属性或者状态
- 重新跑一遍测试，确保一切正常工作

驾驭 State

前面的章节已经说明，状态管理在大型的应用中是一个至关重要的话题。总体来说，不光是 React ，很多单页面应用（SPA）框架都面临这个问题。近些年来应用变得越来越复杂。当今的 web 应用面临的一个重大挑战就是如何驾驭和控制状态。

与其他的解决方案相比，React 已经向前迈进了一大步。单向数据流和简单的组件状态管理 API 非常必要 。这些概念使得推断状态和其改变更加容易，在组件级别以及一定程度上的应用级别的状态推断也更加容易。

在不断膨胀的应用中，推断状态的变化随之变得困难。`setState()` 方法使用对象形式而不是函数形式的话，如果在脏状态上进行操作，则可能会引入 bug。为了能够共享状态或者在兄弟组件之间隐藏不必要的状态，你需要将状态进行提升或者降低。有些状况下，组件需要将其状态提升，因为它的兄弟组件依赖于这些状态。也有可能你需要和相隔甚远的组件共享状态，所以你可能需要在其整个组件树中共享该状态。这样做的结果会使得在状态管理中涉及的组件范围很广。但是毕竟组件的主要职责只是描绘 UI，不是吗？

由于这些原因，存在一些独立的解决方案来解决状态管理问题。这些方案不仅仅可以在 React 中使用，但是却使得 React 的生态更加繁荣。你可以使用不同的解决方案来解决你的问题。为了解决规模化的状态管理问题，你可能已经听说过 Redux 或者 MobX。你可以在 React 应用中使用这两者其一。它们还有一些扩展，如 react-redux 和 mobx-react 来将其连接到 React 的视图层。

Redux 和 MobX 超出了本书的讨论范围。当读读完本书的时候，你将获得关于如何继续学习 React 及其生态系统的指导。其中一个学习路线是 Redux。在你深入外部状态管理主题之前，我推荐你阅读这篇文章。它旨在给你一个关于如何学习外部状态管理的更好理解。

练习：

- 阅读更多关于 外部状态管理以及如何学习 的内容
- 看看我的第二本电子书关于 React 中的状态管理

{pagebreak}

你已经学习了 React 的高级状态管理！让我们回顾一下前面几章的内容。

- React
- 将状态提升或者下降到合适的组件中
- setState 方法可以使用函数参数形式来阻止脏状态的 bug
- 存在外部的解决方案帮助你掌握驾驭 state

你可以从 官方代码仓库 中找到源代码。