高级React组件

本章将重点介绍高级 React 组件的实现。我们将了解什么是高阶组件以及如何实现它们。此外，我们还将深入探讨 React 中更高级的主题，并用它实现复杂的交互功能。

引用DOM元素

有时我们需要在 React 中与 DOM 节点进行交互。`ref`属性可以让我们访问元素中的一个节点。通常，访问 DOM 节点是 React 中的一种反模式，因为我们应该遵循它的声明式编程和单向数据流。当我们引入第一个搜索输入组件时，就已经了解这些了。但是在某些情况下，我们仍然需要访问 DOM 节点。官方文档提到了三种情况：

- 使用 DOM API（focus事件，媒体播放等）
- 调用命令式 DOM 节点动画
- 与需要 DOM 节点的第三方库集成（例如 D3.JavaScript）

让我们通过 Search 组件这个例子看一下。当应用程序第一次渲染时，input 字段应该被聚焦。这是需要访问 DOM API 的一种用例。本章将展示渲染时聚焦 input 字段是如何工作的，但由于这个功能对于应用程序并不是很有用，所以我们将在本章之后省略这些更改。尽管如此，你仍然可以为自己的应用程序保留它。

通常，无状态组件和 ES6 类组件中都可以使用 `ref` 属性。在聚焦 input 字段的用例中，我们就需要一个生命周期方法。这就是为什么接下来会先在 ES6 类组件中展示如何使用 `ref` 属性。

第一步是将无状态组件重构为 ES6 类组件。

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

```
# leanpub-start-insert
class Search extends Component {
  render() {
    const {
      value,
      onChange,
```

```
      onSubmit,
      children
    } = this.props;

    return (
# leanpub-end-insert
      <form onSubmit={onSubmit}>
        <input
          type="text"
          value={value}
          onChange={onChange}
        />
        <button type="submit">
          {children}
        </button>
      </form>
# leanpub-start-insert
    );
  }
}
# leanpub-end-insert
```

ES6 类组件的 `this` 对象可以帮助我们通过 `ref` 属性引用 DOM 节点。

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

```
class Search extends Component {
  render() {
    const {
      value,
      onChange,
      onSubmit,
      children
    } = this.props;

    return (
      <form onSubmit={onSubmit}>
        <input
          type="text"
          value={value}
          onChange={onChange}
# leanpub-start-insert
          ref={(node) => { this.input = node; }}
# leanpub-end-insert
        />
        <button type="submit">
          {children}
```

```
        </button>
      </form>
    );
  }
}
```

现在，你可以通过使用 this 对象、适当的生命周期方法和 DOM API 在组件挂载的时候来聚焦 input 字段。

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

```
class Search extends Component {
# leanpub-start-insert
  componentDidMount() {
    if(this.input) {
      this.input.focus();
    }
  }
# leanpub-end-insert

  render() {
    const {
      value,
      onChange,
      onSubmit,
      children
    } = this.props;

    return (
      <form onSubmit={onSubmit}>
        <input
          type="text"
          value={value}
          onChange={onChange}
          ref={(node) => { this.input = node; }}
        />
        <button type="submit">
          {children}
        </button>
      </form>
    );
  }
}
```

当应用程序渲染时，input 字段应该被聚焦。这就是 `ref` 属性的基本用法。

但是我们怎样在没有`this`对象的无状态组件中访问`ref`属性呢？接下来我们在无状态组件中演示。

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

```
const Search = ({
  value,
  onChange,
  onSubmit,
  children
}) => {
# leanpub-start-insert
  let input;
# leanpub-end-insert
  return (
    <form onSubmit={onSubmit}>
      <input
        type="text"
        value={value}
        onChange={onChange}
# leanpub-start-insert
        ref={(node) => input = node}
# leanpub-end-insert
      />
      <button type="submit">
        {children}
      </button>
    </form>
  );
}
```

现在我们能够访问 input DOM 元素。由于在无状态组件中，没有生命周期方法去触发聚焦事件，这个功能对于聚焦 input 字段这个用例而言没什么用。但是在将来，你可能会遇到其他一些合适的需要在无状态组件中使用`ref`属性的情况。

练习

- 阅读 React中的ref属性概述

- 阅读 在React中使用ref属性

加载 ……

现在让我们回到应用程序。当向 Hacker News API 发起搜索请求时，我们想要显示一个加载指示符。

请求是异步的，此时应该向用户展示某些事情即将发生的某种反馈。让我们在 `src/App.js` 中定义一个可重用的 Loading 组件。

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

```
const Loading = () =>
  <div>Loading ...</div>
```

现在我们需要存储加载状态 (isLoading)。根据加载状态 (isLoading)，决定是否显示 Loading 组件。

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

```
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      results: null,
      searchKey: '',
      searchTerm: DEFAULT_QUERY,
      error: null,
# leanpub-start-insert
      isLoading: false,
# leanpub-end-insert
    };

    ...
  }

  ...

}
```

`isLoading` 的初始值是 false。在 App 组件挂载完成之前，无需加载任何东西。

当发起请求时，将加载状态 (isLoading) 设置为 true。最终，请求会成功，那时可以将加载状态 (isLoading) 设置为 false。

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

```
class App extends Component {

  ...

  setSearchTopStories(result) {
    ...

    this.setState({
      results: {
        ...results,
        [searchKey]: { hits: updatedHits, page }
      },
# leanpub-start-insert
      isLoading: false
# leanpub-end-insert
    });
  }

  fetchSearchTopStories(searchTerm, page = 0) {
# leanpub-start-insert
    this.setState({ isLoading: true });
# leanpub-end-insert

    fetch(`${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTer
      .then(response => response.json())
      .then(result => this.setSearchTopStories(result))
      .catch(e => this.setState({ error: e }));
  }

  ...

}
```

最后一步，我们将在应用程序中使用 Loading 组件。基于加载状态 (isLoading) 的条件来决定渲染 Loading 组件或 Button 组件。后者为一个用于获取更多数据的按钮。

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

```
class App extends Component {

  ...

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

    ...

    return (
      <div className="page">
        ...
        <div className="interactions">
# leanpub-start-insert
          { isLoading
            ? <Loading />
            : <Button
              onClick={() => this.fetchSearchTopStories(searchKe
              More
            </Button>
          }
# leanpub-end-insert
        </div>
      </div>
    );
  }
}
```

由于我们在`componentDidMount()`中发起请求，Loading 组件会在应用程序启动的时候显示。此时，因为列表是空的，所以不显示 Table 组件。当响应数据从 Hacker News API 返回时，返回的数据会通过 Table 组件显示出来，加载状态 (isLoading) 设置为 false，然后 Loading 组件消失。同时，出

现了可以获取更多的数据的“More”按钮。一旦点击按钮，获取更多的数据，该按钮将消失，加载组件会重新出现。

练习:

- 使用第三方库，比如Font Awesome，来显示加载图标，而不是“Loading ...”文本

高阶组件

高阶组件（HOC）是 React 中的一个高级概念。HOC 与高阶函数是等价的。它接受任何输入 - 多数时候是一个组件，也可以是可选参数 - 并返回一个组件作为输出。返回的组件是输入组件的增强版本，并且可以在JSX中使用。

HOC可用于不同的情况，比如：准备属性，管理状态或更改组件的表示形式。其中一种情况是将 HOC 用于帮助实现条件渲染。想象一下现在有一个 List 组件，由于列表可以为空或无，那么它可以渲染一个列表或者什么也不渲染。当没有列表的时候，HOC 可以屏蔽掉这个不显示任何内容的列表。另一方面，这个简单的 List 组件不再需要关心列表存不存在，它只关心渲染列表。

我们接下来创建一个简单的 HOC，它将一个组件作为输入并返回一个组件。我们可以把它放在 `src / App.js` 文件中。

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

```
function withFoo(Component) {
  return function(props) {
    return <Component { ...props } />;
  }
}
```

有一个惯例是用 “with” 前缀来命名 HOC。由于我们现在使用的是 ES6，因此可以使用 ES6 箭头函数更简洁地表达 HOC。

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

```
const withFoo = (Component) => (props) =>
  <Component { ...props } />
```

在这个例子中，没有做任何改变，输入组件将和输出组件一样。它渲染与输入组件相同的实例，并将所有的属性(props)传递给输出组件，但是这个 HOC 没意义。我们来增强输出组件功能：当加载状态 (isLoading) 为 true 时，组件显示 Loading 组件，否则显示输入的组件。条件渲染是 HOC 的一种绝佳用例。

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

```
# leanpub-start-insert
const withLoading = (Component) => (props) =>
  props.isLoading
    ? <Loading />
    : <Component { ...props } />
# leanpub-end-insert
```

基于加载属性 (isLoading)，我们可以实现条件渲染。该函数将返回 Loading 组件或输入的组件。

一般来说，将对象展开然后作为一个组件的输入是非常高效的（比如说前面那个例子中的 props 对象）。请参阅下面的代码片段中的区别。

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

```
// before you would have to destructure the props before passing
const { foo, bar } = props;
<SomeComponent foo={foo} bar={bar} />

// but you can use the object spread operator to pass all object
<SomeComponent { ...props } />
```

有一点应该避免。我们把包括`isLoading`属性在内的所有 props 通过展开对象传递给输入的组件。

然而，输入的组件可能不关心`isLoading`属性。我们可以使用 ES6 中的 rest 解构来避免它。

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

```
# leanpub-start-insert
const withLoading = (Component) => ({ isLoading, ...rest }) =>
  isLoading
    ? <Loading />
```

```
    : <Component { ...rest } />
# leanpub-end-insert
```

这段代码从 `props` 对象中取出一个属性，并保留剩下的属性。这也适用于多个属性。你可能已经在 解构赋值中了解过它。

现在，我们已在 JSX 中使用 HOC。应用程序中的用例可能是显示 “More” 按钮或 Loading 组件。

Loading 组件已经封装在 HOC 中，缺失了输入组件。在显示 Button 组件或 Loading 组件的用例中，Button 是 HOC 的输入组件。增强的输出组件是一个 ButtonWithLoading 的组件。

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

```
const Button = ({ onClick, className = '', children }) =>
  <button
    onClick={onClick}
    className={className}
    type="button"
  >
    {children}
  </button>

const Loading = () =>
  <div>Loading ...</div>

const withLoading = (Component) => ({ isLoading, ...rest }) =>
  isLoading
    ? <Loading />
    : <Component { ...rest } />

# leanpub-start-insert
const ButtonWithLoading = withLoading(Button);
# leanpub-end-insert
```

现在所有的东西已经被定义好了。最后一步，就是使用 ButtonWithLoading 组件，它接收加载状态 (isLoading) 作为附加属性。当 HOC 消费加载属性 (isLoading) 时，再将所有其他 props 传递给 Button 组件。

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

```
class App extends Component {
```

```
  ...

  render() {
    ...
    return (
      <div className="page">
        ...
        <div className="interactions">
# leanpub-start-insert
          <ButtonWithLoading
            isLoading={isLoading}
            onClick={() => this.fetchSearchTopStories(searchKey,
            More
          </ButtonWithLoading>
# leanpub-end-insert
        </div>
      </div>
    );
  }
}
```

当再次运行测试时，App 组件的快照测试会失败。执行 diff 在命令行可能显示如下：

{title="Command Line",lang="text"}

```
-    <button
-      className=""
-      onClick={[Function]}
-      type="button"
-    >
-      More
-    </button>
+    <div>
+      Loading ...
+    </div>
```

如果你认为是 App 组件有问题，现在可以选择修复该组件，或者选择接受 App 组件的新快照。因为本章介绍了 Loading 组件，我们可以在交互测试的命令行中接受已经更改的快照测试。

高阶组件是 React 中的高级技术。它可以使组件具有更高的重用性，更好的抽象性，更强的组合性，以及提升对 props，state 和视图的可操作性。如果不能马上理解，别担心。我们需要时间去熟悉它。

我们推荐阅读高阶组件的简单介绍。这篇文章介绍了另一种学习高阶组件的方法，展示了如何用函数式的方式定义高阶组件并优雅的使用它，以及使用高阶组件解决条件渲染的问题。

练习:

- 阅读 高阶组件的简单介绍
- 使用创建的高阶组件
- 思考一个适合使用高阶组件的场景
- 如果想到了使用场景，请实现这个高阶组件

高级排序

我们已经实现了客户端和服务器端搜索交互。因为我们已经拥了 Table 组件，所以增强 Table 组件的交互性是有意义的。那接下来，我们为 Table 组件加入根据列标题进行排序的功能如何？

你自己写一个排序函数，但是一般这种情况，我个人更喜欢使用第三方工具库。lodash就是这些工具库之一，当然你也可以选择适用于你的任何第三方库。让我们安装 lodash 并使用。

{title="Command Line",lang="text"}

```
npm install lodash
```

现在我们可以在 `src/App` 文件中导入lodash的sort方法。

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

```
import React, { Component } from 'react';
import fetch from 'isomorphic-fetch';
# leanpub-start-insert
import { sortBy } from 'lodash';
# leanpub-end-insert
import './App.css';
```

Table 组件中有好几列，分别是标题，作者，评论和评分。你可以定义排序函数，而每个函数接受一个列表并返回按照指定属性排序过的列表。此外，我们还需要一个默认的排序函数，该函数不做排序而只是用于返回未排序的列表。这将作为组件的初始状态。

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

```
...

# leanpub-start-insert
const SORTS = {
  NONE: list => list,
  TITLE: list => sortBy(list, 'title'),
  AUTHOR: list => sortBy(list, 'author'),
  COMMENTS: list => sortBy(list, 'num_comments').reverse(),
  POINTS: list => sortBy(list, 'points').reverse(),
};
# leanpub-end-insert

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

可以看到有两个排序函数返回一个反向列表。这是因为当用户首次点击排序的时候，希望查看评论和评分最高的项目，而不是最低的。

现在，SORTS 对象允许你引用任何排序函数。

我们的 App 组件负责存储排序函数的状态。组件的初始状态存储的是默认排序函数，它不对列表排序而只是将输入的list作为输出。

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

```
this.state = {
  results: null,
  searchKey: '',
  searchTerm: DEFAULT_QUERY,
  error: null,
  isLoading: false,
# leanpub-start-insert
  sortKey: 'NONE',
# leanpub-end-insert
};
```

一旦用户选择了一个不同的`sortKey`，比如说`AUTHOR`，App组件将从`SORTS`对象中选取合适的排序函数对列表进行排序。

现在，我们要在App组件中定义一个新的类方法，用来将`sortKey`设置为App组件的状态。然后，`sortKey`可以被用来选取对应的排序函数并对其列表进行排序。

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

```
class App extends Component {

  constructor(props) {

    ...

    this.needsToSearchTopStories = this.needsToSearchTopStories.
    this.setSearchTopStories = this.setSearchTopStories.bind(thi
    this.fetchSearchTopStories = this.fetchSearchTopStories.bind
    this.onSearchSubmit = this.onSearchSubmit.bind(this);
    this.onSearchChange = this.onSearchChange.bind(this);
    this.onDismiss = this.onDismiss.bind(this);
# leanpub-start-insert
    this.onSort = this.onSort.bind(this);
# leanpub-end-insert
  }

# leanpub-start-insert
  onSort(sortKey) {
    this.setState({ sortKey });
  }
# leanpub-end-insert

  ...

}
```

下一步是将类方法和`sortKey`传递给 Table 组件。

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

```
class App extends Component {

  ...

  render() {
    const {
      searchTerm,
      results,
      searchKey,
```

```
      error,
      isLoading,
# leanpub-start-insert
      sortKey
# leanpub-end-insert
    } = this.state;

    ...

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

Table 组件负责对列表排序。它通过 `sortKey` 选取 `SORT` 对象中对应的排序函数，并列表作为该函数的输入。之后，Table 组件将在已排序的列表上继续 mapping。

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

```
# leanpub-start-insert
const Table = ({
  list,
  sortKey,
  onSort,
  onDismiss
}) =>
# leanpub-end-insert
  <div className="table">
# leanpub-start-insert
    {SORTS[sortKey](list).map(item =>
# leanpub-end-insert
      <div key={item.objectID} className="table-row">
        ...
      </div>
```

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

理论上，列表可以按照其中的任意排序函数进行排序，但是默认的排序 (sortKey) 是 `NONE` ，所以列表不进行排序。至此，还没有人执行 `onSort()` 方法来改变 `sortKey` 。让我们接下来用一行列标题来扩展表格，每个列标题会使用列中的 Sort 组件对每列进行排序。

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

```
const Table = ({
  list,
  sortKey,
  onSort,
  onDismiss
}) =>
  <div className="table">
# leanpub-start-insert
    <div className="table-header">
      <span style={{ width: '40%' }}>
        <Sort
          sortKey={'TITLE'}
          onSort={onSort}
        >
          Title
        </Sort>
      </span>
      <span style={{ width: '30%' }}>
        <Sort
          sortKey={'AUTHOR'}
          onSort={onSort}
        >
          Author
        </Sort>
      </span>
      <span style={{ width: '10%' }}>
        <Sort
          sortKey={'COMMENTS'}
          onSort={onSort}
        >
          Comments
        </Sort>
      </span>
      <span style={{ width: '10%' }}>
        <Sort
          sortKey={'POINTS'}
```

```
          onSort={onSort}
        >
          Points
        </Sort>
      </span>
      <span style={{ width: '10%' }}>
        Archive
      </span>
    </div>
# leanpub-end-insert
    {SORTS[sortKey](list).map(item =>
      ...
    )}
  </div>
```

每个 Sort 组件都有一个指定的 `sortKey` 和通用的 `onSort()` 函数。Sort 组件调用 `onSort()` 方法去设置指定的 `sortKey`。

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

```
const Sort = ({ sortKey, onSort, children }) =>
  <Button onClick={() => onSort(sortKey)}>
    {children}
  </Button>
```

如你所见，Sort 组件重用了我们的 Button 组件，当点击按钮时，每个传入的 `sortKey` 都会被 `onSort()` 方法设置。现在，我们应该能够通过点击列标题来对列表进行排序了。

这里有个改善外观的小建议。到目前为止，列标题中的按钮看起来有点傻。我们给 Sort 组件中的按钮添加一个合适的 `className`。

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

```
const Sort = ({ sortKey, onSort, children }) =>
# leanpub-start-insert
  <Button
    onClick={() => onSort(sortKey)}
    className="button-inline"
  >
# leanpub-end-insert
    {children}
  </Button>
```

现在应该看起来不错。接下来的目标是实现反向排序。如果点击 Sort 组件两次，该列表应该被反向排序。首先，我们需要用一个布尔值来定义反向状态 (isSortReverse)。排序可以反向或不反向。

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

```
this.state = {
  results: null,
  searchKey: '',
  searchTerm: DEFAULT_QUERY,
  error: null,
  isLoading: false,
  sortKey: 'NONE',
# leanpub-start-insert
  isSortReverse: false,
# leanpub-end-insert
};
```

现在在排序方法中，可以评判列表是否被反向排序。如果状态中的 sortKey 与传入的 sortKey 相同，并且反向状态 (isSortReverse) 尚未设置为 true，则相反——反向状态 (isSortReverse) 设置为 true。

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

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

同样，将反向属性 (isSortReverse) 传递给 Table 组件。

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

```
class App extends Component {

  ...

  render() {
    const {
      searchTerm,
      results,
      searchKey,
```

```
      error,
      isLoading,
      sortKey,
# leanpub-start-insert
      isSortReverse
# leanpub-end-insert
    } = this.state;

    ...

    return (
      <div className="page">
        ...
        <Table
          list={list}
          sortKey={sortKey}
# leanpub-start-insert
          isSortReverse={isSortReverse}
# leanpub-end-insert
          onSort={this.onSort}
          onDismiss={this.onDismiss}
        />
        ...
      </div>
    );
  }
}
```

现在 Table 组件有一个块体箭头函数用于计算数据。

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

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

  return(
# leanpub-end-insert
```

```
    <div className="table">
      <div className="table-header">
        ...
      </div>
# leanpub-start-insert
      {reverseSortedList.map(item =>
# leanpub-end-insert
        ...
      )}
    </div>
# leanpub-start-insert
  );
}
# leanpub-end-insert
```

反向排序现在应该可以工作了。

最后值得一提，为了改善用户体验，我们可以思考一个开放性的问题：用户可以区分当前是根据哪一列进行排序的吗？目前为止，用户是区别不出来的。我们可以给用户一个视觉反馈。

每个 Sort 组件都已经有了其的特定 `sortKey`。它可以用来识别被激活的排序。我们可以将内部组件状态 `sortKey` 作为激活排序标识 (activeSortKey) 传递给 Sort 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(
    <div className="table">
      <div className="table-header">
        <span style={{ width: '40%' }}>
          <Sort
            sortKey={'TITLE'}
            onSort={onSort}
```

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

```
  );
}
```

现在在 Sort 组件中，我们可以基于 `sortKey` 和 `activeSortKey` 得知排序是否被激活。给 Sort 组件增加一个 `className` 属性，用于在排序被激活的时候给用户一个视觉反馈。

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

```
# leanpub-start-insert
const Sort = ({
  sortKey,
  activeSortKey,
  onSort,
  children
}) => {
  const sortClass = ['button-inline'];

  if (sortKey === activeSortKey) {
    sortClass.push('button-active');
  }

  return (
    <Button
      onClick={() => onSort(sortKey)}
      className={sortClass.join(' ')}
    >
      {children}
    </Button>
  );
}
# leanpub-end-insert
```

这样定义 `sortClass` 的方法有点蠢，不是吗？有一个库可以让它看起来更优雅。首先，我们需要安装它。

{title="Command Line",lang="text"}

```
npm install classnames
```

其次，需要将其导入 `src/App.js` 文件。

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

```
import React, { Component } from 'react';
import fetch from 'isomorphic-fetch';
import { sortBy } from 'lodash';
# leanpub-start-insert
import classNames from 'classnames';
# leanpub-end-insert
import './App.css';
```

现在，我们可以通过条件式语句来定义组件的`className`。

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

```
const Sort = ({
  sortKey,
  activeSortKey,
  onSort,
  children
}) => {
# leanpub-start-insert
  const sortClass = classNames(
    'button-inline',
    { 'button-active': sortKey === activeSortKey }
  );
# leanpub-end-insert

  return (
# leanpub-start-insert
    <Button
      onClick={() => onSort(sortKey)}
      className={sortClass}
    >
# leanpub-end-insert
      {children}
    </Button>
  );
}
```

同样在运行测试时，我们会看到 Table 组件失败的快照测试，及一些失败的单元测试。由于我们再次更改了组件显示，因此可以选择接受快照测试。但是必须修复单元测试。在我们的`src/App.test.js`文件中，需要为 Table 组件提供`sortKey`和`isSortReverse`。

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

```
...

describe('Table', () => {

  const props = {
    list: [
      { title: '1', author: '1', num_comments: 1, points: 2, obj
      { title: '2', author: '2', num_comments: 1, points: 2, obj
    ],
# leanpub-start-insert
    sortKey: 'TITLE',
    isSortReverse: false,
# leanpub-end-insert
  };

  ...

});
```

可能需要再一次接受 Table 组件的失败的快照测试，因为我们给 Table 组件提供更多的 props。

现在，我们的高级排序交互完成了。

练习:

- 使用像Font Awesome这样的库来指示（反向）排序
- 就是在每个排序标题旁边显示向上箭头或向下箭头图标
- 阅读了解classnames

{pagebreak}

我们已经学会了React中的高级组件技术！现在来回顾一下本章：

- React
- 通过 ref 属性引用 DOM 节点
- 高阶组件是构建高级组件的常用方法
- 高级交互在 React 中的实现
- 帮助实现条件 classNames 的一个优雅库
- ES6
- rest 解构拆分对象和数组

你可以在官方代码库找到源代码。