使用真实的API

现在是时候使用真实的 API 了，老是处理样本数据会变得很无聊。

如果你对 API 不熟悉，我建议你去读读我的博客，里面有关于我是怎样了解 API 的。

你知道 Hacker News 这个平台吗？它是一个很棒的技术新闻整合平台。在本书中，你将使用它的 API 来获取热门资讯。它有一个基础 API 和一个搜索 API 来获取数据。后者使我们可以去搜索 Hacker News上的资讯。你也可以通过 API 规范来了解它的数据结构。

生命周期方法

在你开始在组件中通过 API 来获取数据之前，你需要知道 React 的生命周期方法。这些方法是嵌入 React 组件生命周期中的一组挂钩。它们可以在 ES6 类组件中使用，但是不能在无状态组件中使用。

你还记得前章中讲过的 JavaScript ES6 类以及如何在 React 中使用它们吗？除了 `render()` 方法外，还有几个方法可以在 React ES6 类组件中被覆写。所有的这些都是生命周期方法。现在让我们来深入了解他们：

通过之前的学习，你已经知道两种能够用在 ES6 类组件中的生命周期方法：`constructor()` 和 `render()`。

constructor（构造函数）只有在组件实例化并插入到 DOM 中的时候才会被调用。组件实例化的过程称作组件的挂载（mount）。

`render()` 方法也会在组件挂载的过程中被调用，同时当组件更新的时候也会被调用。每当组件的状态（state）或者属性（props）改变时，组件的 `render()` 方法都会被调用。

现在你了解了更多关于这两个生命周期方法的知识，也知道它们什么时候会被调用了。你也已经在前面的学习中使用过它们了。但是 React 里还有更多的生命周期方法。

在组件挂载的过程中还有另外两个生命周期方法：`componentWillMount()` 和 `componentDidMount()`。构造函数（constructor）最先执行，`component`

`WillMount()` 会在 `render()` 方法之前执行，而 `componentDidMount()` 在 `render()` 方法之后执行。

总而言之，在挂载过程中有四个生命周期方法，它们的调用顺序是这样的：

- constructor()
- componentWillMount()
- render()
- componentDidMount()

但是当组件的状态或者属性改变的时候用来更新组件的生命周期是什么样的呢？总的来说，它一共有5个生命周期方法用于组件更新，调用顺序如下：

- componentWillReceiveProps()
- shouldComponentUpdate()
- componentWillUpdate()
- render()
- componentDidUpdate()

最后但同样重要的，组件卸载也有生命周期。它只有一个生命周期方法：`componentWillUnmount()`。

但是毕竟你不用一开始就了解所有生命周期方法。这样可能吓到你，而你也并不会用到所有的方法。即使在一个很大的 React 应用当中，除了 `constructor()` 和 `render()` 比较常用外，你只会用到一小部分生命周期函数。即使这样，了解每个生命周期方法的适用场景还是对你有帮助的：

- **constructor(props)** - 它在组件初始化时被调用。在这个方法中，你可以设置初始化状态以及绑定类方法。
- **componentWillMount()** - 它在 `render()` 方法之前被调用。这就是为什么它可以用作去设置组件内部的状态，因为它不会触发组件的再次渲染。但一般来说，还是推荐在 `constructor()` 中去初始化状态。
- **render()** - 这个生命周期方法是必须有的，它返回作为组件输出的元素。这个方法应该是一个纯函数，因此不应该在这个方法中修改组件的状态。它把属性和状态作为输入并且返回（需要渲染的）元素
- **componentDidMount()** - 它仅在组件挂载后执行一次。这是发起异步请求去 API 获取数据的绝佳时期。获取到的数据将被保存在内部组件的状态中然后在 `render()` 生命周期方法中展示出来。

- **componentWillReceiveProps(nextProps)** - 这个方法在一个更新生命周（update lifecycle）中被调用。新的属性会作为它的输入。因此你可以利用 `this.props` 来对比之后的属性和之前的属性，基于对比的结果去实现不同的行为。此外，你可以基于新的属性来设置组件的状态。

- **shouldComponentUpdate(nextProps, nextState)** - 每次组件因为状态或者属性更改而更新时，它都会被调用。你将在成熟的React应用中使用它来进行性能优化。在一个更新生命周期中，组件及其子组件将根据该方法返回的布尔值来决定是否重新渲染。这样你可以阻止组件的渲染生命周期（render lifecycle）方法，避免不必要的渲染。

- **componentWillUpdate(nextProps, nextState)** - 这个方法是 render() 执行之前的最后一个方法。你已经拥有下一个属性和状态，它们可以在这个方法中任由你处置。你可以利用这个方法在渲染之前进行最后的准备。注意在这个生命周期方法中你不能再触发 `setState()`。如果你想基于新的属性计算状态，你必须利用 `componentWillReceiveProps()`。

- **componentDidUpdate(prevProps, prevState)** - 这个方法在 render() 之后立即调用。你可以用它当成操作 DOM 或者执行更多异步请求的机会。

- **componentWillUnmount()** - 它会在组件销毁之前被调用。你可以利用这个生命周期方法去执行任何清理任务。

之前你已经用过了 `constructor()` 和 `render()` 生命周期方法。对于 ES6 类组件来说他们是常用的生命周期方法。实际上 `render()` 是必须有的，否则它将不会返回一个组件实例。

还有另一个生命周期方法：`componentDidCatch(error, info)`。它在 React 16 中引入，用来捕获组件的错误。举例来说，在你的应用中展示样本数据本来是没问题的。但是可能会有列表的本地状态被意外设置成 `null` 的情况发生（例如从外部 API 获取列表失败时，你把本地状态设置为空了）。然后它就不能像之前一样去过滤（filter）和映射（map）这个列表，因为它不是一个空列表（`[]`）而是 `null`。这时组件就会崩溃，然后整个应用就会挂

掉。现在你可以用 `componentDidCatch()` 来捕获错误，将它存在本地的状态中，然后像用户展示一条信息，说明应用发生了错误。

练习：

- 阅读更多关于 React 生命周期函数的内容。
- 阅读更多关于 React 中状态与生命周期函数的关系的内容。
- 阅读更多关于组件错误处理的内容。

获取数据

现在你已经做好了从 Hacker News API 获取数据的准备。 我们可以用上文所提到过的 `componentDidMount()` 生命周期方法来获取数据。你将使用 JavaScript 原生的 fetch API 来发起请求。

在开始之前，让我们设置好 URL 常量和默认参数，来将 API 请求分解成几步。

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

```
import React, { Component } from 'react';
import './App.css';

# leanpub-start-insert
const DEFAULT_QUERY = 'redux';

const PATH_BASE = 'https://hn.algolia.com/api/v1';
const PATH_SEARCH = '/search';
const PARAM_SEARCH = 'query=';
# leanpub-end-insert

...
```

在 JavaScript ES6 中，你可以用模板字符串（template strings）去连接字符串。你将用它来拼接最终的 API 访问地址。

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

```
// ES6
const url = `${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${DEFAULT

// ES5
```

```
var url = PATH_BASE + PATH_SEARCH + '?' + PARAM_SEARCH + DEFAULT

console.log(url);
// output: https://hn.algolia.com/api/v1/search?query=redux
```

这样就可以保证以后你 URL 组合的灵活性。

让我们开始使用 API 请求，在这个请求中将用到上述的网址。整个数据获取的过程在下面代码中一次给出，但后面会对每一步做详细解释。

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

```
...

class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
# leanpub-start-insert
      result: null,
      searchTerm: DEFAULT_QUERY,
# leanpub-end-insert
    };

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

# leanpub-start-insert
  setSearchTopStories(result) {
    this.setState({ result });
  }

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

  componentDidMount() {
```

```
    const { searchTerm } = this.state;
    this.fetchSearchTopStories(searchTerm);
  }
# leanpub-end-insert

  ...
}
```

这段代码做了很多事。我想把它分成更小的代码段，但是那样又会让人很难去理解每段代码之间的关系。接下来我就来详细解释代码中的每一步。

首先，你可以移除样本列表了，因为你将从 Hacker News API 得到一个真实的列表。这些样本数据已经没用了。现在组件将一个空的列表结果以及一个默认的搜索词作为初始状态。这个默认搜索词也同样用在 Search 组件的输入字段和第一个 API 请求中。

其次，在组件挂载之后，它用了 `componentDidMount()` 生命周期方法去获取数据。在第一次获取数据时，使用的是本地状态中的默认搜索词。它将获取与 “redux” 相关的资讯，因为它是默认的参数。

再次，这里使用的是原生的 fetch API。JavaScript ES6 模板字符串允许组件利用 `searchTerm` 来组成 URL。该 URL 是原生 fetch API 函数的参数。返回的响应需要被转化成 JSON 格式的数据结构。这是在处理 JSON 数据结构时，原生的 fetch API 中的强制步骤。最后将处理后的响应赋值给组件内部状态中的结果。此外，我们用一段 catch 代码来处理出错的情况。如果在发起请求时出现错误，这个函数会进入到 catch 中而不是 then 中。在本书之后的章节中，将涵盖错误处理的内容。

最后但同样重要的是，不要忘记在构造函数中绑定你的组件方法。

现在你可以用获取的数据去代替样本数据了。然而，你必须注意一点，这个结果不仅仅是一个数据的列表。它也是一个复杂的对象，它包含了元数据信息以及一系列的hits，在我们的应用里就是这些资讯。你可以在 `render()` 方法中用 `console.log(this.state);` 将这些信息打印出来，以便有一个直观的认识。

在接下来的步骤中，你将把之前的得到的结果渲染出来。但我们不会什么都渲染，在刚开始没有拿到结果时，我们会返回空。一旦 API 请求成功，我们会将结果保存在状态里，然后 App 组件将用更新后的状态重新渲染。

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

```
class App extends Component {

  ...

  render() {
# leanpub-start-insert
    const { searchTerm, result } = this.state;

    if (!result) { return null; }

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

让我们回顾一下在组将的整个生命周期中发生了什么。首先组件通过构造函数得到初始化，之后它将初始化的状态渲染出来。但是你阻止了组件的显示，因为此时本地状态中的结果为空。React允许组件通过返回 null 来不渲染任何东西。接着 `componentDidMount()` 生命周期函数执行。在这个方法中你从 Hacker News API 中异步地拿到了数据。一旦数据到达，组件就通过 `setSearchTopStories()` 函数改变组件内部的状态。之后，由于状态的更新，`update` 的生命周期开始运行。组件再次执行 `render()` 方法，但这次组件的内部状态中的结果已经填充，不再是空了。因此组件将重新渲染 Table 组件的内容。

你使用了大多数浏览器支持的原生 fetch API 来执行对 API 的异步请求。*create-react-app* 中的配置保证了它被所有浏览器支持。你也可以使用第三方库来代替原生 fetch API，例如：superagent 和 axios。

让我们重回到你的应用，现在你应该可以看到资讯列表了。然而，现在应用中仍然存在两个bug。第一，“Dismiss” 按钮不工作。因为它还不能处理这个复杂的 result 对象。当我们点击 “Dismiss” 按钮时，它仍然在操作之前那个简单的 result 对象。第二，当这个列表显示出来之后，你再尝试搜索其他的

东西时，它只会在客户端过滤已有的列表，即使初始化的资讯搜索是在服务器端进行的。我们期待的行为是：当我们使用 Search 组件时，从 API 拿到新的结果，而不是去过滤样本数据。不用担心，两个 bug 都将在之后的章节中得到修复。

练习

- 阅读更多关于 ES6 模板字符串的内容。
- 阅读更多关于原生 fetch API的内容。
- 阅读更多关于在 React 中获取数据的内容。

扩展操作符

“Dismiss” 按钮之所以不工作，是因为 `onDismiss()` 方法不能处理复杂的 result 对象。它现在还只能处理一个本地状态中的简单列表。但是现在这个列表已经不再是简单的平铺列表了。现在，让我们去操作这个 result 对象而不是去操作列表。

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

```
onDismiss(id) {
  const isNotId = item => item.objectID !== id;
# leanpub-start-insert
  const updatedHits = this.state.result.hits.filter(isNotId);
  this.setState({
    ...
  });
# leanpub-end-insert
}
```

那现在 `setState()` 中发生了什么呢？很遗憾，这个 result 是一个复杂的对象。资讯（hits）列表只是这个对象的众多属性之一。所以，当某一项资讯从 result 对象中移除时，只能更新资讯列表，其他的属性还是得保持原样。

解决方法之一是直接改变 result 对象中的 hits 字段。我将演示这个方法，但实际操作中我们一般不这样做。

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

```
// don`t do this
this.state.result.hits = updatedHits;
```

React 拥护不可变的数据结构。因此你不应该改变一个对象（或者直接改变状态）。更好的做法是基于现在拥有的资源来创建一个新的对象。这样就没有任何对象被改变了。这样做的好处是数据结构将保持不变，因为你总是返回一个新对象，而之前的对象保持不变。

因此你可以用 JavaScript ES6 中的 `Object.assign()` 函数来到达这样的目的。它把接收的第一个参数作为目标对象，后面的所有参数作为源对象。然后把所有的源对象合并到目标对象中。只要把目标对象设置成一个空对象，我们就得到了一个新的对象。这种做法是拥抱不变性的，因为没有任何源对象被改变。以下是代码实现：

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

```
const updatedHits = { hits: updatedHits };
const updatedResult = Object.assign({}, this.state.result, updat
```

当遇到相同的属性时，排在后面的对象会覆写先前对象的该属性。现在让我们用它来改写 `onDismiss()` 方法：

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

```
onDismiss(id) {
  const isNotId = item => item.objectID !== id;
  const updatedHits = this.state.result.hits.filter(isNotId);
  this.setState({
# leanpub-start-insert
    result: Object.assign({}, this.state.result, { hits: updated
# leanpub-end-insert
  });
}
```

这已经是一个解决方案了。但是在 JavaScript ES6 以及之后的 JavaScript 版本中还有一个更简单的方法。现在我将向你介绍扩展操作符。它只由三个点组成：`...`。当使用它时，数组或对象中的每一个值都会被拷贝到一个新的数组或对象。

让我们先来看一下 ES6 中**数组**的扩展运算符，虽然你现在还用不到它。

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

```
const userList = ['Robin', 'Andrew', 'Dan'];
const additionalUser = 'Jordan';
```

```
const allUsers = [ ...userList, additionalUser ];

console.log(allUsers);
// output: ['Robin', 'Andrew', 'Dan', 'Jordan']
```

这里 `allUsers` 是一个全新的数组变量，而变量 `userList` 和 `additionalUser` 还是和原来一样。用这个运算符，你甚至可以合并两个数组到一个新的数组中。

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

```
const oldUsers = ['Robin', 'Andrew'];
const newUsers = ['Dan', 'Jordan'];
const allUsers = [ ...oldUsers, ...newUsers ];

console.log(allUsers);
// output: ['Robin', 'Andrew', 'Dan', 'Jordan']
```

现在让我们来看看对象的扩展运算符。它并不是 JavaScript ES6 中的用法。它是针对下一个JavaScript版本的提出的，然而它已经在 React 社区开始使用了。这就是为什么需要在 *create-react-app* 配置中加入了这个功能。

本质上来说，对象的扩展运算符和数组的扩展运算符是一样的，只是用在了对象上。

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

```
const userNames = { firstname: 'Robin', lastname: 'Wieruch' };
const age = 28;
const user = { ...userNames, age };

console.log(user);
// output: { firstname: 'Robin', lastname: 'Wieruch', age: 28 }
```

类似于之前数组的例子，以下是扩展多个对象的例子。

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

```
const userNames = { firstname: 'Robin', lastname: 'Wieruch' };
const userAge = { age: 28 };
const user = { ...userNames, ...userAge };
```

```
console.log(user);
// output: { firstname: 'Robin', lastname: 'Wieruch', age: 28 }
```

最终，它可以用来代替 `Object.assign()`。

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

```
onDismiss(id) {
  const isNotId = item => item.objectID !== id;
  const updatedHits = this.state.result.hits.filter(isNotId);
  this.setState({
# leanpub-start-insert
    result: { ...this.state.result, hits: updatedHits }
# leanpub-end-insert
  });
}
```

现在 "Dismiss" 按钮可以再次工作了，因为 `onDismiss()` 方法已经能够处理这个复杂的 result 对象了，并且知道当要忽略掉列表中的某一项时怎么去更新列表了。

练习

- 阅读更多 ES6 Object.assign()的内容。
- 阅读更多 ES6 数组的扩展操作符的内容。
- 对象的扩展操作符在其中也有简单提到

条件渲染

React 应用很早就引入了条件渲染。但本书还没有提到过，因为目前为止还没有合适的用例。条件渲染用于你需要决定渲染哪个元素时。有些时候也可以是渲染一个元素或者什么都不渲染。其实最简单的条件渲染，只需要用 JSX 中的 if-else 就可以实现。

组件内部状态中的 `result` 对象的初始值为空。当 API 的结果还没返回时，此时的 App 组件没有返回任何元素。这已经是一个条件渲染了，因为在某个特定条件下，render() 方法提前返回了。根据条件，App 组件渲染它的元素或者什么都不渲染。

现在，让我们更进一步。因为只有 Table 组件的渲染依赖于 `result`，所以将它包在一个独立的条件渲染中才比较合理。即使 `result` 为空，其它的所

有组件还是应该被渲染。你只需要在 JSX 中加上一个三元运算符就可以达到这样的目的。

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

```
class App extends Component {

  ...

  render() {
# leanpub-start-insert
    const { searchTerm, result } = this.state;
# leanpub-end-insert
    return (
      <div className="page">
        <div className="interactions">
          <Search
            value={searchTerm}
            onChange={this.onSearchChange}
          >
            Search
          </Search>
        </div>
# leanpub-start-insert
        { result
          ? <Table
            list={result.hits}
            pattern={searchTerm}
            onDismiss={this.onDismiss}
          />
          : null
        }
# leanpub-end-insert
      </div>
    );
  }
}
```

这是实现条件渲染的第二种方式。第三种则是运用 `&&` 逻辑运算符。在 JavaScript中， `true && 'Hello World'` 的值永远是 “Hello World”。而 `false && 'Hello World'` 的值则永远是 false。

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

```
const result = true && 'Hello World';
console.log(result);
```

```
// output: Hello World

const result = false && 'Hello World';
console.log(result);
// output: false
```

在 React 中你也可以利用这个运算符。如果条件判断为 true，`&&` 操作符后面的表达式的值将会被输出。如果条件判断为 false，React 将会忽略并跳过后面的表达式。这个操作符可以用来实现 Table 组件的条件渲染，因为它返回一个 Table 组件或者什么都不返回。

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

```
{ result &&
  <Table
    list={result.hits}
    pattern={searchTerm}
    onDismiss={this.onDismiss}
  />
}
```

这是 React 中使用条件渲染的一些方式。你可以在条件渲染代码大全中找到更多的选择，了解不同的条件渲染方式和它们的适用场景。

现在，你应该能够在你的应用中看到获取的数据。并且当数据正在获取时，你也可以看到除了 Table 组件以外的所有东西。一旦请求完成并且数据存入本地状态之后，Table 组件也将被渲染出来。因为 `render()` 方法再次执行，而且这时条件渲染判定为展示 Table 组件。

练习

- 阅读更多关于 React条件渲染的内容。
- 阅读更多关于实现条件渲染的不同方法的内容。

客户端或服务端搜索

目前当你使用 Search 组件的输入栏时，你会在客户端过滤这个列表。所以你现在要做的是使用 Hacker News API 在服务器端来进行搜索。否则，你将只能处理第一次从 `componentDidMount()` 拿到的默认搜索词的 API 响应。

你可以在 App 组件中定义一个 `onSearchSubmit()` 方法。当 Search 组件执行搜索时，可以用这个方法来从 Hacker News API 获取结果。这与 `componentDidMount()` 生命周期方法中的获取数据的方式相同，但是这次搜索的内容变了，不用初始设定里的默认搜索词了。

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

```
class App extends Component {

  constructor(props) {
    super(props);

    this.state = {
      result: null,
      searchTerm: DEFAULT_QUERY,
    };

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

  ...

# leanpub-start-insert
  onSearchSubmit() {
    const { searchTerm } = this.state;
    this.fetchSearchTopStories(searchTerm);
  }
# leanpub-end-insert

  ...
}
```

现在 Search 组件需要增加一个新的按钮了。这个按钮需要显示地触发搜索请求。否则每次当你改变输入框中的值时，你就会向 Hacker News API 发起请求。但你想要的是用一个明确的 `onClick()` 处理器来帮你控制它。

你确实可以通过某种方式（延迟）来去除 `onChange()` 的抖动，从而省去这个按钮。但是这样也会增加复杂度而且可能得不偿失。我们现在就用简单的方式来做就好了。

首先，把 `onSearchSubmit()` 方法传给 Search 组件。

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

```
class App extends Component {

  ...

  render() {
    const { searchTerm, result } = this.state;
    return (
      <div className="page">
        <div className="interactions">
          <Search
            value={searchTerm}
            onChange={this.onSearchChange}
# leanpub-start-insert
            onSubmit={this.onSearchSubmit}
# leanpub-end-insert
          >
            Search
          </Search>
        </div>
        { result &&
          <Table
            list={result.hits}
            pattern={searchTerm}
            onDismiss={this.onDismiss}
          />
        }
      </div>
    );
  }
}
```

随后，在你的 Search 组件中加一个按钮。将这个按钮设置为 `type="submit"`，并通过表单（form）的 `onSubmit` 属性去传递 `onSubmit()` 方法。你可以复用 children 属性，但这里它会被用作按钮的显示内容。

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

```
# leanpub-start-insert
const Search = ({
  value,
  onChange,
  onSubmit,
```

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

在 Table 组件中，你可以移除过滤功能了，因为已经不会在客户端进行过滤（搜索）了。同时别忘记移除 `isSearched()` 函数。它也不会使用了。现在，当你点击 “Search” 按钮时，搜索结果将直接从 Hacker News API 中得到。

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

```
class App extends Component {

  ...

  render() {
    const { searchTerm, result } = this.state;
    return (
      <div className="page">
        ...
        { result &&
          <Table
# leanpub-start-insert
            list={result.hits}
            onDismiss={this.onDismiss}
# leanpub-end-insert
          />
        }
      </div>
    );
  }
}

...

# leanpub-start-insert
```

```
const Table = ({ list, onDismiss }) =>
# leanpub-end-insert
  <div className="table">
# leanpub-start-insert
    {list.map(item =>
# leanpub-end-insert
      ...
    )}
  </div>
```

现在当你尝试去搜索时，你会注意到浏览器重新加载了。这是提交 HTML 表单后的浏览器原生行为。在 React 中，你会经常遇到用 `preventDefault()` 事件方法来阻止类似于这样的浏览器原生行为。

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

```
# leanpub-start-insert
onSearchSubmit(event) {
# leanpub-end-insert
  const { searchTerm } = this.state;
  this.fetchSearchTopStories(searchTerm);
# leanpub-start-insert
  event.preventDefault();
# leanpub-end-insert
}
```

现在你已经能搜索不同的资讯了。非常棒，这说明你在和一个真正 API 打交道，这样也就不再需要在客户端进行搜索了。

练习

- 阅读更多关于 React 中的合成事件的内容。
- 试一试 Hacker News API

分页抓取

你仔细看过返回回来的数据结构吗？Hacker News API 返回的不仅仅只有资讯（hits）列表。确切地说它返回的是一个分页列表。利用分页属性（在第一个响应中为 `0`），将具有相同搜索词的下一页传给 API，你就可以获取更多分页的子列表了。

首先，让我们改造一下可组合的 API 常量，以便处理分页数据。

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

```
const DEFAULT_QUERY = 'redux';

const PATH_BASE = 'https://hn.algolia.com/api/v1';
const PATH_SEARCH = '/search';
const PARAM_SEARCH = 'query=';
# leanpub-start-insert
const PARAM_PAGE = 'page=';
# leanpub-end-insert
```

现在你可以使用新常量将分页参数添加到你的 API 请求中。

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

```
const url = `${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchT

console.log(url);
// output: https://hn.algolia.com/api/v1/search?query=redux&page
```

`fetchSearchTopStories()` 函数接收分页作为第二个参数。如果你不提供第二个参数，它将使用 `0` 作为初始参数并发起请求。因此 `componentDidMount()` 和 `onSearchSubmit()` 方法在第一个请求中默认获取第一页。之后的请求将根据提供的第二个参数抓取下一个页面的数据。

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

```
class App extends Component {

  ...

# leanpub-start-insert
  fetchSearchTopStories(searchTerm, page = 0) {
    fetch(`${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTer
# leanpub-end-insert
      .then(response => response.json())
      .then(result => this.setSearchTopStories(result))
      .catch(e => e);
  }

  ...

}
```

现在你可以使用在 `fetchSearchTopStories()` 中 API 返回中的当前页。你也可以通过 `onClick` 点击事件来使用这个方法，以便抓取更多的资讯。现在让我们来实现通过按钮从 Hacker News API 中获取更多的分页数据的功能。你只需要定义 `onClick()` 事件处理器，这个处理器以当前的搜索词和下一页的页码作为参数（当前页码 + 1）。

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

```
class App extends Component {

  ...

  render() {
    const { searchTerm, result } = this.state;
# leanpub-start-insert
    const page = (result && result.page) || 0;
# leanpub-end-insert
    return (
      <div className="page">
        <div className="interactions">
        ...
        { result &&
          <Table
            list={result.hits}
            onDismiss={this.onDismiss}
          />
        }
# leanpub-start-insert
        <div className="interactions">
          <Button onClick={() => this.fetchSearchTopStories(sear
            More
          </Button>
        </div>
# leanpub-end-insert
      </div>
    );
  }
}
```

此外，当结果还没有返回时，你应该保证 `render()` 方法中的默认分页为 0。记住 `render()` 方法是在 `componentDidMount()` 生命周期方法去异步获取数据之前调用的。

这里还遗漏了一步。你抓取了下一个分页的数据，但新数据会覆盖你之前的分页数据。理想的情况下，result 对象中新的列表和本地状态中老的列表应

该合并起来才对。现在让我们来实现将新的数据添加到老的数据上而不是去覆盖它。

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

```
setSearchTopStories(result) {
# leanpub-start-insert
  const { hits, page } = result;

  const oldHits = page !== 0
    ? this.state.result.hits
    : [];

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

  this.setState({
    result: { hits: updatedHits, page }
  });
# leanpub-end-insert
}
```

现在在 `setSearchTopStories()` 方法中做了以下一些操作。首先，你从 result 对象中拿到 hits 字段和 page 字段。

第二，你必须检查老的 hits 字段是否存在。当页码为0时，这应该是一个来自 `componentDidMount()` 或者 `onSearchSubmit()` 方法的新的搜索请求；所以 hits 是空的。但是当你通过点击 “More” 按钮去抓取更多的分页数据时，页码不为0；此时它是下一页。老的 hits 已经储存在状态中等待着与新的分页合并。

第三，你不想覆盖老的 hits。你可以合并老的 hits 及 API 返回的新的 hits。这两个列表的合并可以通过 JavaScript ES6 数据扩展操作符完成。

第四，将合并后的 hits 和页码设置到本地组件的状态中。

你还可以做一个最后的调整。当你尝试点击 “More” 按钮时，它只抓取一定数量的资讯。但在每个请求的中，你可以通过设置 API URL 来获取更多的资讯。同样地，你还可以添加更多的可组合路径常量。

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

```
const DEFAULT_QUERY = 'redux';
# leanpub-start-insert
const DEFAULT_HPP = '100';
# leanpub-end-insert

const PATH_BASE = 'https://hn.algolia.com/api/v1';
const PATH_SEARCH = '/search';
const PARAM_SEARCH = 'query=';
const PARAM_PAGE = 'page=';
# leanpub-start-insert
const PARAM_HPP = 'hitsPerPage=';
# leanpub-end-insert
```

现在你可以使用这些常量来扩展 API URL了。

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

```
fetchSearchTopStories(searchTerm, page = 0) {
# leanpub-start-insert
  fetch(`${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTerm}
# leanpub-end-insert
    .then(response => response.json())
    .then(result => this.setSearchTopStories(result))
    .catch(e => e);
}
```

现在，我们能够在一个请求中从 Hacker News API 获取更多的数据了。如你所见，功能强大的 Hacker News API 为你提供了大量的方法，以便让你用真实数据做练习。在学习新的东西时，你应该利用真实的 API 来让整个过程更加有趣。这就是我如何利用 API 提供的便利来学习新的编程语言或库的。

练习

- 体验 Hacker News API 变量

客户端缓存

每次提交表单都会发起一个对 Hacker News API 的请求。你可能先搜索了“redux”，然后搜索了“react”，最后再次搜索了“redux”。这样它总共发起了3次请求。但是你搜索了“redux”两次并且每次都会执行一次异步操作去获取数据。如果有客户端的缓存，它将保存每次搜索的结果。当需要请求 API 的

时候，它首先检查这个请求的结果是否已经在缓存中。如果在，那就使用缓存数据。否则再发 API 请求去获取数据。

为了实现在客户端对搜索结果的缓存，你必须在你的内部组件的状态中存储多个结果（`results`）而不是一个结果（`result`）。这些结果对象将会与搜索词映射成一个键值对。而每一个从 API 得到的结果会以搜索词为键（key）保存下来。

此时，在本地状态中，你的 result 看起应该是这样：

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

```
result: {
  hits: [ ... ],
  page: 2,
}
```

假设你已经发起了两次 API 请求。一次搜索 “redux”，另一次搜索 “react”。那你的 results 对象看起来应该是这样：

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

```
results: {
  redux: {
    hits: [ ... ],
    page: 2,
  },
  react: {
    hits: [ ... ],
    page: 1,
  },
  ...
}
```

让我们用 React 的 `setState()` 方法来实现客户端缓存。首先，在初始化组件状态中重命名 `result` 对象为 `results`。其次，定义一个临时的 `searchKey` 用来储存单个 `result`。

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

```
class App extends Component {

  constructor(props) {
```

```
    super(props);

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

    ...

  }

  ...

}
```

`searchKey` 的值必须在发起请求之前设置。它的值来自 `searchTerm`。你可能会想：为什么我们不直接使用 `searchTerm` 呢？这是在我们继续之前需要理解的重点。`searchTerm` 是一个动态的变量，因此它随输入的关键字变化而变化。然而，这里你需要的是一个稳定的变量。它保存最近一次提交给 API 的搜索词，也可以用它来检索结果集中的某个结果。由于它指向缓存中的当前返回结果，因此还可以在 `render()` 方法中用来显示当前结果。

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

```
componentDidMount() {
  const { searchTerm } = this.state;
# leanpub-start-insert
  this.setState({ searchKey: searchTerm });
# leanpub-end-insert
  this.fetchSearchTopStories(searchTerm);
}

onSearchSubmit(event) {
  const { searchTerm } = this.state;
# leanpub-start-insert
  this.setState({ searchKey: searchTerm });
# leanpub-end-insert
  this.fetchSearchTopStories(searchTerm);
  event.preventDefault();
}
```

现在，你必须去调整一下内部组件状态中储存结果的位置。它应该通过 `searchKey` 来存储每个结果。

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

```
class App extends Component {

  ...

  setSearchTopStories(result) {
    const { hits, page } = result;
# leanpub-start-insert
    const { searchKey, results } = this.state;

    const oldHits = results && results[searchKey]
      ? results[searchKey].hits
      : [];
# leanpub-end-insert

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

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

  ...

}
```

在 `results` 集中，`searchKey` 用作键名（key），其值用来保存更新后的 hits 和 page。

首先，你必须从组件状态中检索 `searchKey`。记住 `searchKey` 是在 `componentDidMount()` 和 `onSearchSubmit()` 中设置的。

第二，和之前一样，老的 hits 需要合并到新的 hits 中。但是这次老的 hits 是以 `searchKey` 为键名从 `results` 集中找到的。

第三，在状态里，一个新的 `result` 可以设置在 `results` 集中。让我们来看看 `setState()` 中的 `results` 对象是什么样子。

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

```
results: {
  ...results,
  [searchKey]: { hits: updatedHits, page }
}
```

下半部分的代码是为了保证，通过 `searchKey` 将更新后的 `result` 对象保存在 `results` 集中。它包含 hits 和 page 属性的对象。而 `searchKey` 的值就是搜索词。现在你学会了 `[searchKey]: ...` 这样的语法。这个语法中，ES6 是通过计算得到属性名的。它可以帮助你实现动态分配对象的值。

上半部分的代码则是用对象扩展运算符将所有其它包含在 results 集中的 `searchKey` 展开。否则，你将会失去之前所有储存过的 results。

现在你以搜索词为键名，将所有结果储存了起来。这是实现缓存的第一步。接下来，你可以根据稳定的 `searchKey` 从 results 集中检索 result。这就是为什么一开始你必须引进 `searchKey` 作为一个稳定的变量。不然用动态的 `searchTerm` 去检索当前的 result 时，这个检索过程会崩溃，因为它的值可能在你使用 Search 组件时改变过了。

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

```
class App extends Component {

  ...

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

    const page = (
      results &&
      results[searchKey] &&
      results[searchKey].page
    ) || 0;
```

```
    const list = (
      results &&
      results[searchKey] &&
      results[searchKey].hits
    ) || [];

# leanpub-end-insert
    return (
      <div className="page">
        <div className="interactions">
          ...
        </div>
# leanpub-start-insert
        <Table
          list={list}
          onDismiss={this.onDismiss}
        />
# leanpub-end-insert
        <div className="interactions">
# leanpub-start-insert
          <Button onClick={() => this.fetchSearchTopStories(sear
# leanpub-end-insert
            More
          </Button>
        </div>
      </div>
    );
  }
}
```

当 `searchKey` 没有对应的结果时，你默认得到一个空列表，所以现在可以节省 Table 组件的条件渲染了。此外，你需要传 `searchKey` 给 “More” 按钮来代替 `searchTerm`。否则，你抓取分页的搜索词将是 `searchTerm` 这个可能变化的值。另外，确保 “Search” 组件的输入字段用的是动态的 `searchTerm`。

这样，搜索功能就可以再次工作了。它会保存所有从 Hacker News API 返回的结果。

此外，`onDismiss()` 方法也需要优化。它仍还在处理 `result` 对象。现在它必须处理 `results` 了。

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

```
  onDismiss(id) {
# leanpub-start-insert
```

```
    const { searchKey, results } = this.state;
    const { hits, page } = results[searchKey];
# leanpub-end-insert

    const isNotId = item => item.objectID !== id;
# leanpub-start-insert
    const updatedHits = hits.filter(isNotId);

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

这样 “Dismiss” 按钮就可以再次工作了。

然而，现在我们还不能阻止应用对每一次搜索都发起一个 API 请求。即使我们保存了某一个结果，但也还没有任何阻止重复请求的检查。也就是说缓存功能仍不完整。虽然应用缓存了所有结果，但它还没有将这些结果利用起来。所有我们的最后一步就是：如果搜索的结果已经存在于缓存中，就阻止 API 请求。

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

```
class App extends Component {

  constructor(props) {

    ...

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

# leanpub-start-insert
  needsToSearchTopStories(searchTerm) {
    return !this.state.results[searchTerm];
```

```
  }
# leanpub-end-insert

  ...

  onSearchSubmit(event) {
    const { searchTerm } = this.state;
    this.setState({ searchKey: searchTerm });
# leanpub-start-insert

    if (this.needsToSearchTopStories(searchTerm)) {
      this.fetchSearchTopStories(searchTerm);
    }

# leanpub-end-insert
    event.preventDefault();
  }

  ...

}
```

现在就算你重复搜索一个词，客户端也只会发起一次请求。以这种方式进行缓存的话分页的数据也会保存下来，因为结果的每一页都将保存在 `results` 集中。这是不是一个很强大的方法来引入缓存呢？而且 Hacker News API 提供了你所需的一切，甚至可以高效地缓存分页数据。

错误处理

你与 Hacker News API 交互的所有功能都已就位。你甚至引入了一种优雅的方式来缓存 API 结果，而且通过分页列表功能还可以从 API 中获取无尽的资讯子列表。但是我们还忽略了一项。但我们忘了一件事情，不幸的是，它在日常开发中经常被忽略：错误处理。人们常常容易沉浸于主逻辑的开发中，却忘记错误也会随之而来。

在本章中，引入了一个高效的方法来为你的应用添加一个当发生错误的 API 请求时的错误处理。其实你已经掌握了在 React 中处理错误的基础知识，也就是本地状态和条件渲染。本质上来讲，错误只是 React 的另一种状态。当一个错误发生时，你先将它存在本地状态中，而后利用条件渲染在组件中显示错误信息。听起来是不是很简单。现在让我们在 App 组件中实现它，因为它是向 Hacker News API 发起请求的组件。首先，你要在本地状态中引入 error 这个状态。它初始化为 null，但当错误发生时它会被置成一个 error 对象。

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

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

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

    ...
  }

...

}
```

第二，通过结合使用 catch 和 `setState()`，你可以捕获错误对象并将它存在本地状态中。如果 API 请求失败，catch 就会执行。

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

```
class App extends Component {

  ...

  fetchSearchTopStories(searchTerm, page = 0) {
    fetch(`${PATH_BASE}${PATH_SEARCH}?${PARAM_SEARCH}${searchTer
      .then(response => response.json())
      .then(result => this.setSearchTopStories(result))
# leanpub-start-insert
      .catch(e => this.setState({ error: e }));
# leanpub-end-insert
  }

  ...

}
```

第三，如果错误发生了，你可以在 `render()` 方法中在本地状态里获取到 error 对象，然后利用条件渲染来显示一个错误信息。

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

```
class App extends Component {

  ...

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

    ...

# leanpub-start-insert
    if (error) {
      return <p>Something went wrong.</p>;
    }
# leanpub-end-insert

    return (
      <div className="page">
        ...
      </div>
    );
  }
}
```

就这么简单。如果你想测试错误处理是否工作，你可以把 API URL 换成一个不存在的 URL。

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

```
const PATH_BASE = 'https://hn.foo.bar.com/api/v1';
```

之后，你应该得到一个错误信息而不是应用界面。你可以自己决定错误信息渲染的位置。在这种情况下，整个 app 不再显示。显然这不是最佳用户体

验。我们将 Table 组件和错误信息择一渲染如何？这样的话当错误发生时，除了 Table 组件，应用其余的部分仍然可见。

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

```
class App extends Component {

  ...

  render() {
    const {
      searchTerm,
      results,
      searchKey,
      error
    } = this.state;

    const page = (
      results &&
      results[searchKey] &&
      results[searchKey].page
    ) || 0;

    const list = (
      results &&
      results[searchKey] &&
      results[searchKey].hits
    ) || [];

    return (
      <div className="page">
        <div className="interactions">
          ...
        </div>
# leanpub-start-insert
        { error
          ? <div className="interactions">
            <p>Something went wrong.</p>
          </div>
          : <Table
            list={list}
            onDismiss={this.onDismiss}
          />
        }
# leanpub-end-insert
        ...
      </div>
    );
```

```
  }
}
```

最后，别忘了把 URL 还原成一个真实的 URL。

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

```
const PATH_BASE = 'https://hn.algolia.com/api/v1';
```

你的应用应该仍然可以工作，不过此时如果 API 请求失败应用就有错误处理了。

练习

- 阅读更多关于 React 组件的错误处理的内容。

{pagebreak}

你已经学会使用 React 去与 API 交互了！现在让我们回顾一下本章内容：

- React
- 针对不同用例的 ES6 类组件生命周期方法
- componentDidMount() 如何用于 API 交互
- 条件渲染
- 表单上的合成事件
- 错误处理
- ES6
- 用模板字符串去组合字符串
- 扩展运算符用于不可变数据结构
- 可计算的属性名称
- General
- Hacker News API 交互
- 浏览器原生 fetch API
- 客户端和服务器端搜索
- 数据分页
- 客户端缓存

同样地，此时小憩一下，将学过的知识点消化理解并应用是有必要的。你还可以在之前写过的代码上做做实验，小试牛刀。

你可以在官方代码库中找到源码。