

React入门指南

来源：<http://reactjs.cn/react/docs/getting-started.html>

JSFiddle

开始学习 React 最简单的方式是使用如下JSFiddle的 Hello World例子：

- [React JSFiddle](#)
- [React JSFiddle without JSX](#)

初学者教程包 (Starter Kit)

开始先下载初学者教程包。

[下载初学者教程包 0.13.0](#)

在初学者教程包的根目录，创建一个包含以下内容的

`helloworld.html`。

```
<!DOCTYPE html>
<html>
  <head>
    <script src="build/react.js"></script>
    <script src="build/JSXTransformer.js"></script>
  </head>
  <body>
    <div id="example"></div>
    <script type="text/jsx">
      React.render(
        <h1>Hello, world!</h1>,
        document.getElementById('example')
      );
    </script>
  </body>
</html>
```

在 JavaScript 代码里写着 XML 格式的代码称为 JSX；可以去 [JSX 语法](#) 里学习更多 JSX 相关的知识。为了把 JSX 转成标准的 JavaScript，我们用

`<script type="text/jsx">` 标签包裹着含有 JSX 的代

码，然后引入 `JSXTransformer.js` 库来实现在浏览器里的代码转换。

分离文件

你的 React JSX 代码文件可以写在单独的文件里。创建 `src/helloworld.js` 文件，内容如下：

```
React.render(  
  <h1>Hello, world!</h1>,  
  document.getElementById('example')  
)
```

然后在 `helloworld.html` 引用它：

```
<script type="text/jsx" src="src/helloworld.js"></script>
```

离线转换

先安装命令行工具（依赖 [npm](#)）：

```
npm install -g react-tools
```

然后将你的 `src/helloworld.js` 文件转成标准的 JavaScript:

```
jsx --watch src/ build/
```

一旦你修改了，`build/helloworld.js` 文件会自动生成。

```
React.render(  
  React.createElement('h1', null, 'Hello, world!'),  
  document.getElementById('example')  
)
```

对照以下内容更新你的 HTML 代码

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello React!</title>
    <script src="build/react.js"></script>
    <!-- 不需要JSXTransformer! -->
  </head>
  <body>
    <div id="example"></div>
    <script src="build/helloworld.js"></script>
  </body>
</html>
```

想要遵循 CommonJS 规范?

如果你想在使用 React 时，遵循 [browserify](#)，[webpack](#) 或者其它兼容CommonJS的模块系统，只要使用 `react` npm包即可。而且，`jsx` 转换工具可以很轻松地集成到大部分打包系统里（不仅仅是 CommonJS）。

下一步

接着学习更多 [入门教程](#) 和初学者教程包 `examples` 目录下的其它例子。

我们还有一个社区开发者共同建设的 Wiki：[workflows](#), [UI-components](#), [routing](#), [data management](#) etc.

祝你好运，欢迎来到 React 的世界。

我们将构建一个简单却真实的评论框，你可以将它放入你的博客，类似disqus、livefyre、facebook提供的实时评论的基础版。

我们将提供以下内容：

- 一个展示所有评论的视图
- 一个提交评论的表单
- 用于构建自定义后台的接口链接（hooks）

同时也包含一些简洁的特性：

- **评论体验优化**：评论在保存到服务器之前就展现在评论列表，因此用户体验很快。
- **实时更新**：其他用户的评论将会实时展示。
- **Markdown格式**：用户可以使用Markdown格式来编辑文本。

目录：

- [想要跳过所有内容，只查看源代码？](#)
- [运行一个服务器](#)
- [开始学习](#)
- [你的第一个组件](#)
 - [JSX语法](#)
 - [发生了什么](#)
- [制作组件](#)
 - [组件属性](#)
 - [使用props](#)
 - [添加 Markdown](#)

- 接入数据模型
- 从服务器获取数据
- 响应状态变化 (Reactive state)
 - 更新状态
- 添加新的评论
 - 事件
 - Refs
 - 回调函数作为属性
- 优化：提前更新
- 祝贺你!

想要跳过所有内容，只查看源代码？

所有代码都在[GitHub](#)。

运行一个服务器

虽然它不是入门教程的必需品，但接下来我们会添加一个功能，发送 `POST` 请求到服务器。如果这是你熟知的事并且你想创建你自己的服务器，那么就这样干吧。而对于另外的一部分人，为了让你集中精力学习，而不用担忧服务器端方面，我们已经用了以下一系列的语言编写了简单的服务器代码 - JavaScript（使用Node.js），Python和Ruby。所有代码都在GitHub。你可以[查看代码](#)或者[下载zip文件](#)来开始学习。

开始使用下载的教程，只需开始编辑 `public/index.html`。

开始学习

在这个教程里面，我们将使用放在 CDN 上预构建好的 JavaScript 文件。打开你最喜欢的编辑器，创建一个新的 HTML 文档：

```
<!-- index.html -->
<html>
  <head>
    <title>Hello React</title>
    <script src="http://fb.me/react-0.13.0.js"></script>
    <script src="http://fb.me/JSXTransformer-0.13.0.js"></script>
    <script src="http://code.jquery.com/jquery-1.10.0.min.js"></script>
  </head>
  <body>
    <div id="content"></div>
    <script type="text/jsx">
      // Your code here
    </script>
  </body>
</html>
```

在本教程其余的部分，我们将在此 script 标签中编写我们的 JavaScript 代码。

注意：

因为我们想简化 ajax 请求代码，所以在这里引入 jQuery，但是它对 React 并不是必须的。

你的第一个组件

React 中全是模块化、可组装的组件。以我们的评论框为例，我们将有如下的组件结构：

```
- CommentBox
  - CommentList
    - Comment
  - CommentForm
```


让我们构造 `CommentBox` 组件，它只是一个简单的“而已”：

```
// tutorial11.js
var CommentBox = React.createClass({
  render: function() {
    return (
      <div className="commentBox">
        Hello, world! I am a CommentBox.
      </div>
    );
  }
});
React.render(
  <CommentBox />,
  document.getElementById('content')
);
```

JSX语法

首先你注意到 JavaScript 代码中 XML 式的语法语句。我们有一个简单的预编译器，用于将这种语法糖转换成纯的 JavaScript 代码：

```
// tutorial11-raw.js
var CommentBox = React.createClass({displayName: 'CommentBox',
  render: function() {
    return (
      React.createElement('div', {className: "commentBox"},
        "Hello, world! I am a CommentBox."
      )
    );
  }
});
React.render(
  React.createElement(CommentBox, null),
  document.getElementById('content')
);
```

JSX 语法是可选的，但是我们发现 JSX 语句比纯 JavaScript 更加容易使用。阅读更多关于[JSX 语法的文章](#)。

发生了什么

我们通过 JavaScript 对象传递一些方法到

`React.createClass()` 来创建一个新的 React 组件。其中最重要的方法是 `render`，该方法返回一颗 React 组件树，这棵树最终将会渲染成 HTML。

这个 `<div>` 标签不是真实的 DOM 节点；他们是 `React` `div` 组件的实例。你可以认为这些就是 React 知道如何处理的标记或者一些数据。React 是**安全的**。我们不生成 HTML 字符串，因此默认阻止了 XSS 攻击。

你没有必要返回基本的 HTML。你可以返回一个你（或者其他人）创建的组件树。这就使得 React 变得**组件化**：一个关键的前端维护原则。

`React.render()` 实例化根组件，启动框架，注入标记到原始的 DOM 元素中，作为第二个参数提供。

制作组件

让我们为 `CommentList` 和 `CommentForm` 构建骨架，这也会是一些简单的“：

```
// tutorial2.js
var CommentList = React.createClass({
  render: function() {
    return (
      <div className="commentList">
        Hello, world! I am a CommentList.
      </div>
    );
  }
});

var CommentForm = React.createClass({
  render: function() {
    return (
```

```

    <div className="commentForm">
      Hello, world! I am a CommentForm.
    </div>
  );
}
});

```

下一步，更新 `CommentBox` 组件，使用这些新的组件：

```

// tutorial3.js
var CommentBox = React.createClass({
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList />
        <CommentForm />
      </div>
    );
  }
});

```

注意我们是如何混合 HTML 标签和我们创建的组件。

HTML 组件就是普通的 React 组件，就像你定义的一样，只有一点不一样。JSX 编译器会自动重写 HTML 标签为 `React.createElement(tagName)` 表达式，其它什么都不做。这是为了避免全局命名空间污染。

组件属性

让我们创建我们的第三个组件，`Comment`。我们想传递给它作者名字和评论文本，以便于我们能够对每一个独立的评论重用相同的代码。首先让我们添加一些评论到

`CommentList`：

```

// tutorial4.js
var CommentList = React.createClass({
  render: function() {
    return (
      <div className="commentList">
        <Comment author="Pete Hunt">This is one comment</Comment>
        <Comment author="Jordan Walke">This is *another* comment</Co

```

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

请注意，我们已经从父节点 `CommentList` 组件传递给子节点 `Comment` 组件一些数据。例如，我们传递了 *Pete Hunt*（通过一个属性）和 `_This is one comment`（通过类似于XML的子节点）给第一个 `Comment`。从父节点传递到子节点的数据称为 `_props**`，是属性（properties）的缩写。

使用props

让我们创建评论组件。通过 **props**，就能够从中读取到从 `CommentList` 传递过来的数据，然后渲染一些标记：

```
// tutorial5.js
var Comment = React.createClass({
  render: function() {
    return (
      <div className="comment">
        <h2 className="commentAuthor">
          {this.props.author}
        </h2>
        {this.props.children}
      </div>
    );
  }
});
```

在 JSX 中通过将 JavaScript 表达式放在大括号中（作为属性或者子节点），你可以生成文本或者 React 组件到节点树中。我们访问传递给组件的命名属性作为 `this.props` 的键，任何内嵌的元素作为 `this.props.children`。

添加 Markdown

Markdown 是一种简单的格式化内联文本的方式。例如，用星号包裹文本将会使其强调突出。

首先，添加第三方的 **Showdown** 库到你的应用。这是一个 JavaScript 库，处理 Markdown 文本并且转换为原始的 HTML。这需要在你的头部添加一个 script 标签（我们已经在 React 操练场上包含了这个标签）：

```
<!-- index.html -->
<head>
  <title>Hello React</title>
  <script src="http://fb.me/react-0.13.0.js"></script>
  <script src="http://fb.me/JSXTransformer-0.13.0.js"></script>
  <script src="http://code.jquery.com/jquery-1.10.0.min.js"></script>
  <script src="http://cdnjs.cloudflare.com/ajax/libs/showdown/0.3.1/
</head>
```

下一步，让我们转换评论文本为 Markdown 格式，然后输出它：

```
// tutorial6.js
var converter = new Showdown.converter();
var Comment = React.createClass({
  render: function() {
    return (
      <div className="comment">
        <h2 className="commentAuthor">
          {this.props.author}
        </h2>
        {converter.makeHtml(this.props.children.toString())}
      </div>
    );
  }
});
```

我们在这里唯一需要做的就是调用 Showdown 库。我们需要把 `this.props.children` 从 React 的包裹文本转换成 Showdown 能处理的原始的字符串，所以我们显示地调用了 `toString()`。

但是这里有一个问题！我们渲染的评论在浏览器里面看起来像这样：“This is another comment”。我们想这些标签真正地渲染成 HTML。

那是 React 在保护你免受 XSS 攻击。这里有一种方法解决这个问题，但是框架会警告你别使用这种方法：

```
// tutorial7.js
var converter = new Showdown.converter();
var Comment = React.createClass({
  render: function() {
    var rawMarkup = converter.makeHtml(this.props.children.toString());
    return (
      <div className="comment">
        <h2 className="commentAuthor">
          {this.props.author}
        </h2>
        <span dangerouslySetInnerHTML={{__html: rawMarkup}} />
      </div>
    );
  }
});
```

这是一个特殊的 API，故意让插入原始的 HTML 变得困难，但是对于 Showdown，我们将利用这个后门。

记住：使用这个功能，你会依赖于 Showdown 的安全性。

接入数据模型

到目前为止，我们已经在源代码里面直接插入了评论数据。相反，让我们渲染一小块 JSON 数据到评论列表。最终，数据将会来自服务器，但是现在，写在你的源代码中：

```
// tutorial8.js
var data = [
  {author: "Pete Hunt", text: "This is one comment"},
  {author: "Jordan Walke", text: "This is *another* comment"}
];
```

我们需要用一种模块化的方式将数据传入到 `CommentList`。修改 `CommentBox` 和 `React.render()` 方法，通过 `props` 传递数据到 `CommentList`：

```
// tutorial9.js
var CommentBox = React.createClass({
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList data={this.props.data} />
        <CommentForm />
      </div>
    );
  }
});

React.render(
  <CommentBox data={data} />,
  document.getElementById('content')
);
```

现在数据在 `CommentList` 中可用了，让我们动态地渲染评论：

```
// tutorial10.js
var CommentList = React.createClass({
  render: function() {
    var commentNodes = this.props.data.map(function (comment) {
      return (
        <Comment author={comment.author}>
          {comment.text}
        </Comment>
      );
    });
    return (
      <div className="commentList">
        {commentNodes}
      </div>
    );
  }
});
```

就是这样！

从服务器获取数据

让我们用一些从服务器获取的动态数据替换硬编码的数据。我们将移除数据属性，用获取数据的URL来替换它：

```
// tutorial11.js
React.render(
  <CommentBox url="comments.json" />,
  document.getElementById('content')
);
```

这个组件和前面的组件是不一样的，因为它必须重新渲染自己。该组件将不会有任何数据，直到请求从服务器返回，此时该组件或许需要渲染一些新的评论。

响应状态变化 (Reactive state)

到目前为止，每一个组件都根据自己的 props 渲染了自己一次。props 是不可变的：它们从父节点传递过来，被父节点“拥有”。为了实现交互，我们给组件引进了可变的 state。this.state 是组件私有的，可以通过调用 this.setState() 来改变它。当状态更新之后，组件重新渲染自己。

render() methods are written declaratively as functions of this.props and this.state. 框架确保UI始终和输入保持一致。

当服务器获取数据的时候，我们将会用已有的数据改变评论。让我们给 CommentBox 组件添加一个评论数组作为它的状态：

```
// tutorial12.js
var CommentBox = React.createClass({
  getInitialState: function() {
```



```

    return {data: []};
  },
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList data={this.state.data} />
        <CommentForm />
      </div>
    );
  }
});

```

`getInitialState()` 在组件的生命周期中仅执行一次，设置组件的初始化状态。

更新状态

当组件第一次创建的时候，我们想从服务器获取（使用GET方法）一些JSON数据，更新状态，反映出最新的数据。在真实的应用中，这将会是一个动态功能点，但是对于这个例子，我们将会使用一个静态的JSON文件来使事情变得简单：

```

// tutorial13.json
[
  {"author": "Pete Hunt", "text": "This is one comment"},
  {"author": "Jordan Walke", "text": "This is *another* comment"}
]

```

我们将会使用jQuery帮助发出一个一步的请求到服务器。

注意：因为这会变成一个AJAX应用，你将会需要使用一个web服务器来开发你的应用，而不是一个放置在你的文件系统上面的一个文件。[如上所述](#)，我们已经在[GitHub](#)上面提供了几个你可以使用的服务器。这些服务器提供了你学习下面教程所需的功能。

```
// tutorial13.js
var CommentBox = React.createClass({
  getInitialState: function() {
    return {data: []};
  },
  componentDidMount: function() {
    $.ajax({
      url: this.props.url,
      dataType: 'json',
      success: function(data) {
        this.setState({data: data});
      }.bind(this),
      error: function(xhr, status, err) {
        console.error(this.props.url, status, err.toString());
      }.bind(this)
    });
  },
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList data={this.state.data} />
        <CommentForm />
      </div>
    );
  }
});
```

在这里，`componentDidMount` 是一个在组件被渲染的时候 React 自动调用的方法。动态更新的关键点是调用 `this.setState()`。我们把旧的评论数组替换成从服务器拿到的新的数组，然后 UI 自动更新。正是有了这种响应式，一个小的改变都会触发实时的更新。这里我们将使用简单的轮询，但是你可以简单地使用 WebSockets 或者其它技术。

```
// tutorial14.js
var CommentBox = React.createClass({
  loadCommentsFromServer: function() {
    $.ajax({
      url: this.props.url,
      dataType: 'json',
      success: function(data) {
        this.setState({data: data});
      }.bind(this),
      error: function(xhr, status, err) {
```

```

    error: function(xhr, status, err) {
      console.error(this.props.url, status, err.toString());
    }.bind(this)
  });
},
getInitialState: function() {
  return {data: []};
},
componentDidMount: function() {
  this.loadCommentsFromServer();

  setInterval(this.loadCommentsFromServer, this.props.pollInterval);
},
render: function() {
  return (
    <div className="commentBox">
      <h1>Comments</h1>
      <CommentList data={this.state.data} />
      <CommentForm />
    </div>
  );
}
});

React.render(
  <CommentBox url="comments.json" pollInterval={2000} />,
  document.getElementById('content')
);

```

我们在这里所做的就是把AJAX调用移到一个分离的方法中去，组件第一次加载以及之后每隔两秒钟，调用这个方法。尝试在你的浏览器中运行，然后改变 `comments.json` 文件；在两秒钟之内，改变将会显示出来！

添加新的评论

现在是时候构造表单了。我们的 `CommentForm` 组件应该询问用户的名字和评论内容，然后发送一个请求到服务器，保存这条评论。

```

// tutorial15.js
var CommentForm = React.createClass({
  render: function() {
    return (
      <form className="commentForm">

```

```

        <input type="text" placeholder="Your name" />
        <input type="text" placeholder="Say something..." />
        <input type="submit" value="Post" />
      </form>
    );
  }
});

```

让我们使表单可交互。当用户提交表单的时候，我们应该清空表单，提交一个请求到服务器，然后刷新评论列表。首先，让我们监听表单的提交事件和清空表单。

```

// tutorial16.js
var CommentForm = React.createClass({
  handleSubmit: function(e) {
    e.preventDefault();
    var author = this.refs.author.getDOMNode().value.trim();
    var text = this.refs.text.getDOMNode().value.trim();
    if (!text || !author) {
      return;
    }
    // TODO: send request to the server
    this.refs.author.getDOMNode().value = '';
    this.refs.text.getDOMNode().value = '';
    return;
  },
  render: function() {
    return (
      <form className="commentForm" onSubmit={this.handleSubmit}>
        <input type="text" placeholder="Your name" ref="author" />
        <input type="text" placeholder="Say something..." ref="text" />
        <input type="submit" value="Post" />
      </form>
    );
  }
});

```

事件

React使用驼峰命名规范的方式给组件绑定事件处理器。我们给表单绑定一个 `onSubmit` 处理器，用于当表单提交了合法的输入后清空表单字段。

在事件回调中调用 `preventDefault()` 来避免浏览器默认地提交表单。

Refs

我们利用 `Ref` 属性给子组件命名，`this.refs` 引用组件。我们可以在组件上调用 `getDOMNode()` 获取浏览器本地的 DOM 元素。

回调函数作为属性

当用户提交评论的时候，我们需要刷新评论列表来加进这条新评论。在 `CommentBox` 中完成所有逻辑是合适的，因为 `CommentBox` 拥有代表评论列表的状态（state）。

我们需要从子组件传回数据到它的父组件。我们在父组件的 `render` 方法中做这件事：传递一个新的回调函数（`handleCommentSubmit`）到子组件，绑定它到子组件的 `onCommentSubmit` 事件上。无论事件什么时候触发，回调函数都将会被调用：

```
// tutorial17.js
var CommentBox = React.createClass({
  loadCommentsFromServer: function() {
    $.ajax({
      url: this.props.url,
      dataType: 'json',
      success: function(data) {
        this.setState({data: data});
      }.bind(this),
      error: function(xhr, status, err) {
        console.error(this.props.url, status, err.toString());
      }.bind(this)
    });
  },
  handleCommentSubmit: function(comment) {
    // TODO: submit to the server and refresh the list
  },
  getInitialState: function() {
    return {data: []};
  },
});
```

```

componentDidMount: function() {
  this.loadCommentsFromServer();
  setInterval(this.loadCommentsFromServer, this.props.pollInterval
},
render: function() {
  return (
    <div className="commentBox">
      <h1>Comments</h1>
      <CommentList data={this.state.data} />
      <CommentForm onCommentSubmit={this.handleCommentSubmit} />
    </div>
  );
}
});

```

当用户提交表单的时候，让我们在 `CommentForm` 中调用这个回调函数：

```

// tutorial18.js
var CommentForm = React.createClass({
  handleSubmit: function(e) {
    e.preventDefault();
    var author = this.refs.author.getDOMNode().value.trim();
    var text = this.refs.text.getDOMNode().value.trim();
    if (!text || !author) {
      return;
    }
    this.props.onCommentSubmit({author: author, text: text});

    this.refs.author.getDOMNode().value = '';
    this.refs.text.getDOMNode().value = '';
    return;
  },
  render: function() {
    return (
      <form className="commentForm" onSubmit={this.handleSubmit}>
        <input type="text" placeholder="Your name" ref="author" />
        <input type="text" placeholder="Say something..." ref="text" />
        <input type="submit" value="Post" />
      </form>
    );
  }
});

```

现在回调函数已经就绪，唯一我们需要做的就是提交到服务器，然后刷新列表：

```
// tutorial19.js
var CommentBox = React.createClass({
  loadCommentsFromServer: function() {
    $.ajax({
      url: this.props.url,
      dataType: 'json',
      success: function(data) {
        this.setState({data: data});
      }.bind(this),
      error: function(xhr, status, err) {
        console.error(this.props.url, status, err.toString());
      }.bind(this)
    });
  },
  handleCommentSubmit: function(comment) {
    $.ajax({
      url: this.props.url,
      dataType: 'json',
      type: 'POST',
      data: comment,
      success: function(data) {
        this.setState({data: data});
      }.bind(this),
      error: function(xhr, status, err) {
        console.error(this.props.url, status, err.toString());
      }.bind(this)
    });
  },
  getInitialState: function() {
    return {data: []};
  },
  componentDidMount: function() {
    this.loadCommentsFromServer();
    setInterval(this.loadCommentsFromServer, this.props.pollInterval);
  },
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList data={this.state.data} />
        <CommentForm onCommentSubmit={this.handleCommentSubmit} />
      </div>
    );
  }
});
```

优化：提前更新

我们的应用现在已经完成了所有功能，但是在你的评论出现在列表之前，你必须等待请求完成，感觉很慢。我们可以提前添加这条评论到列表中，从而使应用感觉更快。

```
// tutorial20.js
var CommentBox = React.createClass({
  loadCommentsFromServer: function() {
    $.ajax({
      url: this.props.url,
      dataType: 'json',
      success: function(data) {
        this.setState({data: data});
      }.bind(this),
      error: function(xhr, status, err) {
        console.error(this.props.url, status, err.toString());
      }.bind(this)
    });
  },
  handleCommentSubmit: function(comment) {
    var comments = this.state.data;
    var newComments = comments.concat([comment]);
    this.setState({data: newComments});
    $.ajax({
      url: this.props.url,
      dataType: 'json',
      type: 'POST',
      data: comment,
      success: function(data) {
        this.setState({data: data});
      }.bind(this),
      error: function(xhr, status, err) {
        console.error(this.props.url, status, err.toString());
      }.bind(this)
    });
  },
  getInitialState: function() {
    return {data: []};
  },
  componentDidMount: function() {
    this.loadCommentsFromServer();
    setInterval(this.loadCommentsFromServer, this.props.pollInterval);
  },
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList data={this.state.data} />
        <CommentForm onCommentSubmit={this.handleCommentSubmit} />
      </div>
    );
  }
});
```



```
}  
});
```

祝贺你!

你刚刚通过一些简单步骤够早了一个评论框。了解更多关于[为什么使用React](#)的内容，或者深入学习[API参考](#)，开始专研！祝你好运！

这是一篇源自[官方博客](#)的文章。

在我看来，React 是较早使用 JavaScript 构建大型、快速的 Web 应用程序的技术方案。它已经被我们广泛应用于 Facebook 和 Instagram。

React 众多优秀特征中的其中一部分就是，教会你去重新思考如何构建应用程序。

本文中，我将跟你一起使用 React 构建一个具备搜索功能的产品列表。

注意：

如果你无法看到本页内嵌的代码片段，请确认你不是用 `https` 协议加载本页的。

- [从原型（mock）开始](#)
- [第一步：拆分用户界面为一个组件树](#)
- [第二步：利用 React，创建应用的一个静态版本](#)
 - [穿插一小段内容：props 与 state 比较](#)
- [第三步：识别出最小的（但是完整的）代表 UI 的 state](#)
- [第四步：确认 state 的生命周期](#)
- [第五步：添加反向数据流](#)
- [就这么简单](#)

从原型（mock）开始

假设我们已经拥有了一个 JSON API 和设计师设计的原型。我们的设计师显然不够好，因为原型看起来如下：

☐ Only show products in stock

Name	Price
Sporting Goods	
Football	\$49.99
Baseball	\$9.99
Basketball	\$29.99
Electronics	
iPod Touch	\$99.99
iPhone 5	\$399.99
Nexus 7	\$199.99

JSON接口返回数据如下：

```
[
  {category: "Sporting Goods", price: "$49.99", stocked: true, name: "Football"},
  {category: "Sporting Goods", price: "$9.99", stocked: true, name: "Baseball"},
  {category: "Sporting Goods", price: "$29.99", stocked: false, name: "Basketball"},
  {category: "Electronics", price: "$99.99", stocked: true, name: "iPod Touch"},
  {category: "Electronics", price: "$399.99", stocked: false, name: "iPhone 5"},
  {category: "Electronics", price: "$199.99", stocked: true, name: "Nexus 7"}
];
```

第一步：拆分用户界面为一个组件树

你要做的第一件事是，为所有组件（及子组件）命名并画上线框图。假如你和设计师一起工作，也许他们已经完成了这项工作，所以赶紧去跟他们沟通！他们的 Photoshop 图层名也许最终可以直接用于你的 React 组件名。

然而你如何知道哪些才能成为组件？想象一下，当你创建一些函数或对象时，用到一些类似的技术。其中一项技术就是[单一功能原则](#)，指的是，理想状态下一个组件应该只

做一件事，假如它功能逐渐变大就需要被拆分成更小的子组件。

由于你经常需要将一个JSON数据模型展示给用户，因此你需要检查这个模型结构是否正确以便你的 UI（在这里指组件结构）是否能够正确的映射到这个模型上。这是因为用户界面和数据模型在 *信息构造* 方面都要一致，这意味着将你可以省下很多将 UI 分割成组件的麻烦事。你需要做的仅仅只是将数据模型分隔成一小块一小块的组件，以便它们都能够表示成组件。

The diagram shows a user interface for searching products. It consists of a search bar with the placeholder text "Search...", a checkbox labeled "Only show products in stock", and a table of products. The table has two columns: "Name" and "Price". The products are grouped into "Sporting Goods" and "Electronics". The "Basketball" and "iPhone 5" items are highlighted in red. The entire interface is enclosed in an orange border, the search bar is in a blue border, and the product table is in a green border.

Name	Price
Sporting Goods	
Football	\$49.99
Baseball	\$9.99
Basketball	\$29.99
Electronics	
iPod Touch	\$99.99
iPhone 5	\$399.99
Nexus 7	\$199.99

由此可见，我们的 app 中包含五个组件。下面我已经用斜体标示出每个组件对应的数据。

1. `FilterableProductTable` **(橘色)** : 包含整个例子的容器
2. `SearchBar` **(蓝色)** : 接受所有 *用户输入 (user input)*

3. `ProductTable` (绿色) : 根据 用户输入 (*user input*) 过滤和展示 数据集合 (*data collection*)
4. `ProductCategoryRow` (青色) : 为每个 分类 (*category*) 展示一列表头
5. `ProductRow` (红色) : 为每个 产品 (*product*) 展示一行

如果你仔细观察 `ProductTable` , 你会发现表头 (包含“Name”和“Price”标签) 并不是单独的组件。这只是一种个人偏好, 也有一定的争论。在这个例子当中, 我把表头当做 `ProductTable` 的一部分, 因为它是渲染“数据集合”的一份子, 这也是 `ProductTable` 的职责。但是, 当这个表头变得复杂起来的时候 (例如, 添加排序功能), 就应该单独地写一个 `ProductTableHeader` 组件。

既然我们在原型当中定义了这个组件, 让我们把这些元素组成一棵树形结构。这很简单。被包含在其它组件中的组件在属性机构中应该是子级:

- `FilterableProductTable`
 - `SearchBar`
 - `ProductTable`
 - `ProductCategoryRow`
 - `ProductRow`

第二步: 利用 React , 创建应用的一个静态版本

既然已经拥有了组件树，是时候开始实现应用了。最简单的方式就是创建一个应用，这个应用将数据模型渲染到 UI 上，但是没有交互功能。拆分这两个过程是最简单的，因为构建一个静态的版本仅需要大量的输入，而不需要思考；但是添加交互功能却需要大量的思考和少量的输入。我们将会知道这是为什么。

为了创建一个渲染数据模型的应用的静态版本，你将会构造一些组件，这些组件重用其它组件，并且通过 *props* 传递数据。*props* 是一种从父级向子级传递数据的方式。如果你对 `_state_` 概念熟悉，那么_不要使用 *state* _来构建这个静态版本。*state* 仅用于实现交互功能，也就是说，数据随着时间变化。因为这是一个静态的应用版本，所以你并不需要 *state* 。

你可以从上至下或者从下至上来构建应用。也就是说，你可以从属性结构的顶部开始构建这些组件（例如，从 `FilterableProductTable` 开始），或者从底部开始（`ProductRow`）。在简单的应用中，通常情况下从上至下的方式更加简单；在大型的项目中，从下至上的方式更加简单，这样也可以在构建的同时写测试代码。

在这步结束的时候，将会有有一个可重用的组件库来渲染数据模型。这些组件将会仅有 `render()` 方法，因为这是应用的一个静态版本。位于树形结构顶部的组件（`FilterableProductTable`）将会使用数据模型作为 `prop`。如果你改变底层数据模型，然后再次调用 `React.render()`，UI 将会更新。查看 UI 如何被更新和什么地方改变都是很容易的，因为 React 的**单向数据流**

（也被称作“单向绑定”）保持了一切东西模块化，很容易查错，并且速度很快，没有什么复杂的。

如果你在这步中需要帮助，请查看 [React 文档](#)。

穿插一小段内容： props 与 state 比较

在 React 中有两种类型的数据“模型”： props 和 state 。理解两者的区别是很重要的；如果你不太确定两者有什么区别，请大致浏览一下[官方的 React 文档](#)。

第三步：识别出最小的（但是完整的）代表 UI 的 state

为了使 UI 可交互，需要能够触发底层数据模型的变化。React 通过 **state** 使这变得简单。

为了正确构建应用，首先需要考虑应用需要的最小的可变 state 数据模型集合。此处关键点在于精简：*不要存储重复的数据*。构造出绝对最小的满足应用需要的最小 state 是有必要的，并且计算出其它强烈需要的东西。例如，如果构建一个 TODO 列表，仅保存一个 TODO 列表项的数组，而不要保存另外一个指代数组长度的 state 变量。当想要渲染 TODO 列表项总数的时候，简单地取出 TODO 列表项数组的长度就可以了。

思考示例应用中的所有数据片段，有：

- 最初的 products 列表
- 用户输入的搜索文本
- 复选框的值
- 过滤后的 products 列表

让我们分析每一项，指出哪一个是 state 。简单地对每一项数据提出三个问题：

1. 是否是从父级通过 props 传入的？如果是，可能不是 state 。
2. 是否会随着时间改变？如果不是，可能不是 state 。
3. 能根据组件中其它 state 数据或者 props 计算出来吗？如果是，就不是 state 。

初始的 products 列表通过 props 传入，所以不是 state 。搜索文本和复选框看起来像是 state ，因为它们随着时间改变，也不能根据其它数据计算出来。最后，过滤的 products 列表不是 state ，因为可以通过搜索文本和复选框的值从初始的 products 列表计算出来。

所以最终，state 是：

- 用户输入的搜索文本
- 复选框的值

第四步：确认 state 的生命周期

OK，我们辨别出了应用的 state 数据模型的最小集合。接下来，需要指出哪个组件会改变或者说_拥有_这个 state 数据模型。

记住：React 中数据是沿着组件树从上到下单向流动的。可能不会立刻明白哪个组件应该拥有哪些 state 数据模型。**这对新手通常是最难理解和最具挑战的**，因此跟随以下步骤来弄清楚这点：

对于应用中的每一个 state 数据：

- 找出每一个基于那个 state 渲染界面的组件。
- 找出共同的祖先组件（某个单个的组件，在组件树中位于需要这个 state 的所有组件的上面）。
- 要么是共同的祖先组件，要么是另外一个在组件树中位于更高层级的组件应该拥有这个 state。
- 如果找不出拥有这个 state 数据模型的合适的组件，创建一个新的组件来维护这个 state，然后添加到组件树中，层级位于所有共同拥有者组件的上面。

让我们在应用中应用这个策略：

- `ProductTable` 需要基于 state 过滤产品列表，`SearchBar` 需要显示搜索文本和复选框状态。
- 共同拥有者组件是 `FilterableProductTable`。
- 理论上，过滤文本和复选框值位于 `FilterableProductTable` 中是合适的。

太酷了，我们决定了 state 数据模型位于 `FilterableProductTable` 之中。首先，给 `FilterableProductTable` 添加 `getInitialState()` 方法，该方法返回 `{filterText: '', inStockOnly: false}` 来反映应用的初始化状态。然后传递 `filterText` 和 `inStockOnly` 给 `ProductTable` 和 `SearchBar` 作为 prop。最后，使用这些 props 来过滤 `ProductTable` 中的行，设置在 `SearchBar` 中表单字段的值。

你可以开始观察应用将会如何运行：设置 `filterText` 为 `"ball"`，然后刷新应用。将会看到数据表格被正确更新了。

第五步：添加反向数据流

到目前为止，已经构建了渲染正确的基于 props 和 state 的沿着组件树从上至下单向数据流动的应用。现在，是时候支持另外一种数据流动方式了：组件树中层级很深的表单组件需要更新 `FilterableProductTable` 中的 state。

React 让这种数据流动非常明确，从而很容易理解应用是如何工作的，但是相对于传统的双向数据绑定，确实需要输入更多的东西。React 提供了一个叫做 `ReactLink` 的插件来使其和双向数据绑定一样方便，但是考虑到这篇文章的目的，我们将会保持所有东西都直截了当。

如果你尝试在示例的当前版本中输入或者选中复选框，将会发现 React 会忽略你的输入。这是有意的，因为已经设置了 `input` 的 `value` 属性，使其总是与从 `FilterableProductTable` 传递过来的 `state` 一致。

让我们思考下我们希望发生什么。我们想确保无论何时用户改变了表单，都要更新 state 来反映用户的输入。由于组件只能更新自己的 state，`FilterableProductTable` 将会传递一个回调函数给 `SearchBar`，此函数将会在 state 应该被改变的时候触发。我们可以使用 `input` 的 `onChange` 事件来监听用户输入，从而确定何时触发回调函数。`FilterableProductTable` 传递的回调函数将会调用 `setState()`，然后应用将会被更新。

虽然这听起来有很多内容，但是实际上仅仅需要几行代码。并且关于数据在应用中如何流动真的非常清晰明确。

就这么简单

希望以上内容让你明白了如何思考用 React 去构造组件和应用。虽然可能比你之前要输入更多的代码，记住，读代码的时间远比写代码的时间多，并且阅读这种模块化的清晰的代码是相当容易的。当你开始构建大型的组件库的时候，你将会非常感激这种清晰性和模块化，并且随着代码的复用，整个项目代码量就开始变少了 :)。

React 是一个 Facebook 和 Instagram 用来创建用户界面的 JavaScript 库。

很多人认为 React 是 **MVC** 中的 **V**（视图）。

我们创造 React 是为了解决一个问题：**构建随着时间数据不断变化的大规模应用程序**。为了达到这个目标，React 采用下面两个主要的思想。

简单

仅仅只要表达出你的应用程序在任一个时间点应该长的样子，然后当底层的数据变了，React 会自动处理所有用户界面的更新。

声明式 (Declarative)

数据变化后，React 概念上与点击“刷新”按钮类似，但仅会更新变化的部分。

构建可组合的组件

React 都是关于构建可复用的组件。事实上，通过 React 你唯一要做的事情就是构建组件。得益于其良好的封装性，组件使代码复用、测试和关注分离（separation of concerns）更加简单。

给它5分钟的时间

React 挑战了很多传统的知识，第一眼看上去可能很多想法有点疯狂。当你阅读这篇指南，请[给它5分钟的时间](#)；

那些疯狂的想法已经帮助 Facebook 和 Instagram 从里到外创建了上千的组件了。

了解更多

你可以从这篇[博客](#)了解更多我们创造 React 的动机。

用户界面能做的最基础的事就是呈现一些数据。React 让显示数据变得简单，当数据变化时，用户界面会自动同步更新。

快速开始

让我们看一个非常简单的例子。新建一个名为 `hello-react.html` 的文件，内容如下：

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello React</title>
    <script src="http://fb.me/react-0.13.0.js"></script>
    <script src="http://fb.me/JSXTransformer-0.13.0.js"></script>
  </head>
  <body>
    <div id="example"></div>
    <script type="text/jsx">

      // ** 在这里替换成你的代码 **

    </script>
  </body>
</html>
```

在接下去的文档中，我们只关注 JavaScript 代码，假设我们把代码插入到上面那个模板中。用下面的代码替换掉上面用来占位的注释。

```
var HelloWorld = React.createClass({
  render: function() {
    return (
      <p>
        Hello, <input type="text" placeholder="Your name here" />!

        It is {this.props.date.toTimeString()}
      </p>
    );
  }
});

setInterval(function() {
  React.render(
    <HelloWorld date={new Date()} />,
    document.getElementById('example')
  );
}, 500);
```

响应式更新 (Reactive Updates)

在浏览器中打开 `hello-react.html`，在输入框输入你的名字。你会发现 React 在用户界面中只改变了时间，你在输入框的输入内容会保留着，即使你没有写任何代码来完成这个功能。React 也为你解决了这个问题，做了正确的事。

我们想到的解决方案是 React 是不会去操作 DOM 的，除非不得不操作 DOM。它用一种更快的内置仿造的 DOM 来操作差异，为你计算出效率最高的 DOM 改变。

这个组件的输入被称为 `props` - “properties”的缩写。它们通过 JSX 语法进行参数传递。你必须知道，在组件里这些属性是不可直接改变的，也就是说 `this.props` 是只读的。

组件就像是函数

React 组件非常简单。你可以认为它们就是简单的函数，接受 `props` 和 `state` (后面会讨论) 作为参数，然后渲染出 HTML。正是由于它们如此简单，使得它们非常容易理解。

注意:

一个限制: React 组件只能渲染单个根节点。如果你想要返回多个节点，它们_必须_被包含在同一个节点里。

JSX 语法

我们始终坚信，组件使用了正确方法去做关注分离，而不是通过“模板引擎”和“展示逻辑”。我们认为标签和生成它的代码是紧密相连的。此外，展示逻辑通常是很复杂的，通过模板语言实现这些逻辑会产生大量代码。

我们得出解决这个问题最好的方案是通过 JavaScript 直接生成模板，这样你就可以用一个真正语言的所有表达能力去构建用户界面。为了使这变得更简单，我们做了一个非常简单、**可选**类似 HTML 语法，通过函数调用即可生成模板的编译器，称为 JSX。

JSX 让你可以用 HTML 语法去写 JavaScript 函数调用。 为了在 React 生成一个链接，通过纯 JavaScript 你可以这么写：

```
React.createElement('a', {href: 'http://facebook.github.io/react/'}, 'Hello React!')
```

。通过 JSX 这就变成了 `Hello React!`。我们发现这会使搭建 React 应用更加简单，设计师也偏向用这种语法，但是每个人都有自己的工作流，所以**JSX 并不强制必须使用的**。

JSX 非常小；上面“hello, world”的例子使用了 JSX 所有的特性。想要了解更多，请看 [深入理解 JSX](#)。或者直接使用[在线 JSX 编译器](#)观察变化过程。

JSX 类似于 HTML，但不是完全一样。参考 [JSX 陷阱](#) 学习关键区别。

最简单开始学习 JSX 的方法就是使用浏览器端的 `JSXTransformer`。我们强烈建议你不要在生产环境中使用它。你可以通过我们的命令行工具 `react-tools` 包来预编译你的代码。

没有 JSX 的 React

你完全可以选择是否使用 JSX，并不是 React 必须的。你可以通过

`React.createElement` 来创建一个树。第一个参数是标签，第二个参数是一个属性对象，第三个是子节点。

```
var child = React.createElement('li', null, 'Text Content');
var root = React.createElement('ul', { className: 'my-list' }, child);
React.render(root, document.body);
```

方便起见，你可以创建基于自定义组件的速记工厂方法。

```
var Factory = React.createFactory(ComponentClass);  
...  
var root = Factory({ custom: 'prop' });  
React.render(root, document.body);
```

React 已经为 HTML 标签提供内置工厂方法。

```
var root = React.DOM.ul({ className: 'my-list' },  
  React.DOM.li(null, 'Text Content')  
);
```


JSX 是一个看起来很像 XML 的 JavaScript 语法扩展。

React 可以用来做简单的 JSX 句法转换。

- [为什么要使用 JSX ?](#)
- [HTML 标签 与 React 组件 对比](#)
- [转换](#)
- [JavaScript 表达式](#)
 - [属性表达式](#)
 - [子节点表达式](#)
 - [注释](#)

为什么要使用 JSX ?

你不需要为了 React 使用 JSX，可以直接使用纯粹的 JS。但我们更建议使用 JSX，因为它能定义简洁且我们熟知的包含属性的树状结构语法。

对于非专职开发者（比如设计师）同样比较熟悉。

XML 有固定的标签开启和闭合。这能让复杂的树更易于阅读，优于方法调用和对象字面量的形式。

它没有修改 JavaScript 语义。

HTML 标签 与 React 组件 对比

React 可以渲染 HTML 标签 (strings) 或 React 组件 (classes)。

要渲染 HTML 标签，只需在 JSX 里使用小写字母开头的标签名。

```
var myDivElement = <div className="foo" />;
React.render(myDivElement, document.body);
```

要渲染 React 组件，只需创建一个大写字母开头的本地变量。

```
var MyComponent = React.createClass({/*...*/});
var myElement = <MyComponent someProperty={true} />;
React.render(myElement, document.body);
```

React 的 JSX 里约定分别使用首字母大、小写来区分本地组件的类和 HTML 标签。

注意:

由于 JSX 就是 JavaScript，一些标识符像 `class` 和 `for` 不建议作为 XML 属性名。作为替代，React DOM 使用 `className` 和 `htmlFor` 来做对应的属性。

转换

JSX 把类 XML 的语法转成纯粹 JavaScript，XML 元素、属性和子节点被转换成 `React.createElement` 的参数。

```
var Nav;
// 输入 (JSX):
var app = <Nav color="blue" />;
// 输出 (JS):
var app = React.createElement(Nav, {color:"blue"});
```

注意，要想使用 ```，`Nav`` 变量一定要在作用区间内。

JSX 也支持使用 XML 语法定义子结点：

```
var Nav, Profile;
// 输入 (JSX):
var app = <Nav color="blue"><Profile>click</Profile></Nav>;
// 输出 (JS):
var app = React.createElement(
  Nav,
  {color:"blue"},
  React.createElement(Profile, null, "click")
);
```

使用 [JSX 编译器](#) 来试用 JSX 并理解它是如何转换到原生 JavaScript，还有 [HTML 到 JSX 转换器](#) 来把现有 HTML 转成 JSX。

如果你要使用 JSX，这篇 [新手入门](#) 教程来教你如何搭建环境。

注意:

JSX 表达式总是会当作 `ReactElement` 执行。具体的实际细节可能不同。一种优化的模式是把 `ReactElement` 当作一个行内的对象字面量形式来绕过

`React.createElement` 里的校验代码。

JavaScript 表达式

属性表达式

要使用 JavaScript 表达式作为属性值，只需把这个表达式用一对大括号 (`{ }`) 包起来，不要用引号 (`" "`)。

```
// 输入 (JSX):
var person = <Person name={window.isLoggedIn ? window.name : ''} />;
// 输出 (JS):
var person = React.createElement(
  Person,
  {name: window.isLoggedIn ? window.name : ''}
);
```

子节点表达式

同样地，JavaScript 表达式可用于描述子结点：

```
// 输入 (JSX):
var content = <Container>{window.isLoggedIn ? <Nav /> : <Login />}</Container>
// 输出 (JS):
var content = React.createElement(
  Container,
  null,
  window.isLoggedIn ? React.createElement(Nav) : React.createElement(Login)
);
```

注释

JSX 里添加注释很容易；它们只是 JS 表达式而已。你只需要在一个标签的子节点内(非最外层)小心地用 `{ }` 包围要注释的部分。

```
var content = (
  <Nav>
    { /* 一般注释, 用 {} 包围 */ }
    <Person>
      /* 多
       行
       注释 */
      name={window.isLoggedIn ? window.name : ''} // 行尾注释
    </Person>
  </Nav>
);
```

注意:

JSX 类似于 HTML，但不完全一样。参考 [JSX 陷阱](#) 了解主要不同。

如果你事先知道组件需要的全部 Props（属性），JSX 很容易地这样写：

```
var component = <Component foo={x} bar={y} />;
```

修改 Props 是不好的，明白吗

如果你不知道要设置哪些 Props，那么现在最好不要设置它：

```
var component = <Component />;  
component.props.foo = x; // 不好  
component.props.bar = y; // 同样不好
```

这样是反模式，因为 React 不能帮你检查属性类型（propTypes）。这样即使你的 属性类型有错误也不能得到清晰的错误提示。

Props 应该被当作禁止修改的。修改 props 对象可能会导致预料之外的结果，所以最好不要去修改 props 对象。

延展属性（Spread Attributes）

现在你可以使用 JSX 的新特性 - 延展属性：

```
var props = {};  
props.foo = x;  
props.bar = y;  
var component = <Component {...props} />;
```

传入对象的属性会被复制到组件内。

它能被多次使用，也可以和其它属性一起用。注意顺序很重要，后面的会覆盖掉前面的。

```
var props = { foo: 'default' };  
var component = <Component {...props} foo={'override'} />;  
console.log(component.props.foo); // 'override'
```

这个奇怪的 `...` 标记是什么？

这个 `...` 操作符（也被叫做延展操作符 - spread operator）已经被 [ES6 数组](#) 支持。相关的还有 ES7 规范草案中的 [Object 剩余和延展属性 \(Rest and Spread Properties\)](#)。我们利用了这些还在制定中标准中已经被支持的特性来使 JSX 拥有更优雅的语法。

JSX 与 HTML 非常相似，但是有些关键区别要注意。

注意:

关于 DOM 的区别，如行内样式属性 `style`，参考 [DOM 区别](#)

HTML 实体

HTML 实体可以插入到 JSX 的文本中。

```
<div>First &middot; Second</div>
```

如果想在 JSX 表达式中显示 HTML 实体，可以会遇到二次转义的问题，因为 React 默认会转义所有字符串，为了防止各种 XSS 攻击。

```
// 错误：会显示 "First &middot; Second"  
<div>{'First &middot; Second'}</div>
```

有多种绕过的方法。最简单的是直接用 Unicode 字符。这时要确保文件是 UTF-8 编码且网页也指定为 UTF-8 编码。

```
<div>{'First · Second'}</div>
```

安全的做法是先找到 [实体的 Unicode 编号](#)，然后在 JavaScript 字符串里使用。

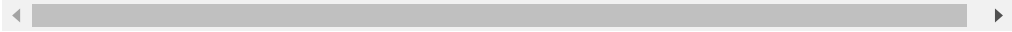
```
<div>{'First \u00b7 Second'}</div>  
<div>{'First ' + String.fromCharCode(183) + ' Second'}</div>
```

可以在数组里混合使用字符串和 JSX 元素。

```
<div>(['First ', <span>&middot;</span>, ' Second'])</div>
```

万不得已，可以直接使用原始 HTML。

```
<div dangerouslySetInnerHTML={{__html: 'First &middot; Second'}} />
```



自定义 HTML 属性

如果往原生 HTML 元素里传入 HTML 规范里不存在的属性，React 不会显示它们。如果需要使用自定义属性，要加 `data-` 前缀。

```
<div data-custom-attribute="foo" />
```

以 `aria-` 开头的 [网络无障碍] 属性可以正常使用。

```
<div aria-hidden={true} />
```


我们已经学习如何使用 React [呈现数据](#)。现在让我们来学习如何创建交互式界面。

简单例子

```
var LikeButton = React.createClass({
  getInitialState: function() {
    return {liked: false};
  },
  handleClick: function(event) {
    this.setState({liked: !this.state.liked});
  },
  render: function() {
    var text = this.state.liked ? 'like' : 'haven\'t liked';
    return (
      <p onClick={this.handleClick}>
        You {text} this. Click to toggle.
      </p>
    );
  }
});

React.render(
  <LikeButton />,
  document.getElementById('example')
);
```

事件处理与合成事件 (Synthetic Events)

React 里只需把事件处理器 (event handler) 以驼峰命名 (camelCased) 形式当作组件的 props 传入即可，就像使用普通 HTML 那样。React 内部创建一套合成事件系统来使所有事件在 IE8 和以上浏览器表现一致。也就是说，React 知道如何冒泡和捕获事件，而且你的事件处理器接收到的 events 参数与 [W3C 规范](#) 一致，无论你使用哪种浏览器。

如果需要在手机或平板等触摸设备上使用 React，需要调用 `React.initializeTouchEvents(true)`；启用触摸事件处理。

幕后原理：自动绑定和事件代理

在幕后，React 做了一些操作来让代码高效运行且易于理解。

Autobinding: 在 JavaScript 里创建回调的时候，为了保证 `this` 的正确性，一般都需要显式地绑定方法到它的实例上。有了 React，所有方法被自动绑定到了它的组件实例上。React 还缓存这些绑定方法，所以 CPU 和内存都是非常高效。而且还能减少打字！

事件代理： React 实际并没有把事件处理器绑定到节点本身。当 React 启动的时候，它在最外层使用唯一一个事件监听器处理所有事件。当组件被加载和卸载时，只是在内部映射里添加或删除事件处理器。当事件触发，React 根据映射来决定如何分发。当映射里处理器时，会当作空操作处理。参考 [David Walsh 很棒的文章](#) 了解这样做高效的原因。

组件其实是状态机 (State Machines)

React 把用户界面当作简单状态机。把用户界面想像成拥有不同状态然后渲染这些状态，可以轻松让用户界面和数据保持一致。

React 里，只需更新组件的 state，然后根据新的 state 重新渲染用户界面（不要操作 DOM）。React 来决定如何最高效地更新 DOM。

State 工作原理

常用的通知 React 数据变化的方法是调用

`setState(data, callback)`。这个方法会合并（merge）`data` 到 `this.state`，并重新渲染组件。渲染完成后，调用可选的 `callback` 回调。大部分情况下不需要提供 `callback`，因为 React 会负责把界面更新到最新状态。

哪些组件应该有 State？

大部分组件的工作应该是从 `props` 里取数据并渲染出来。但是，有时需要对用户输入、服务器请求或者时间变化等作出响应，这时才需要使用 State。

尝试把尽可能多的组件无状态化。这样做能隔离 state，把它放到最合理的地方，也能减少冗余，同时易于解释程序运作过程。

常用的模式是创建多个只负责渲染数据的无状态（stateless）组件，在它们的上层创建一个有状态（stateful）组件并把它的状态通过 `props` 传给子级。这个有状态的组件封装了所有用户的交互逻辑，而这些无状态组件则负责声明式地渲染数据。

哪些 应该作为 State？

State 应该包括那些可能被组件的事件处理器改变并触发用户界面更新的数据。 真实的应用中这种数据一般都很小且能被 JSON 序列化。当创建一个状态化的组件时，想象一下表示它的状态最少需要哪些数据，并只把这些数据存入 `this.state`。在 `render()` 里再根据 state 来计算你需要的其它数据。你会发现以这种方式思考和开发程序最终往往是正确的，因为如果在 state 里添加冗余数据或计算所得数据，需要你经常手动保持数据同步，不能让 React 来帮你处理。

哪些 不应该作为 State ？

`this.state` 应该仅包括能表示用户界面状态所需的最少数据。因此，它不应该包括：

- **计算所得数据：** 不要担心根据 state 来预先计算数据——把所有的计算都放到 `render()` 里更容易保证用户界面和数据的一致性。例如，在 state 里有一个数组 (`listItems`)，我们要把数组长度渲染成字符串，直接在 `render()` 里使用 `this.state.listItems.length + ' list items'` 比把它放到 state 里好的多。
- **React 组件：** 在 `render()` 里使用当前 props 和 state 来创建它。
- **基于 props 的重复数据：** 尽可能使用 props 来作为唯一数据来源。把 props 保存到 state 的一个有效的场景是需要知道它以前值的时候，因为未来的 props 可能会变化。

目前为止，我们已经学了如何用单个组件来展示数据和处理用户输入。下一步让我们来体验 React 最激动人心的特性之一：可组合性（composability）。

- [动机：关注分离](#)
- [组合实例](#)
- [从属关系](#)
- [子级](#)
 - [子级校正（Reconciliation）](#)
 - [子组件状态管理](#)
 - [动态子级](#)
- [数据流](#)
- [性能提醒](#)

动机：关注分离

通过复用那些接口定义良好的组件来开发新的模块化组件，我们得到了与使用函数和类相似的好处。具体来说就是能够通过开发简单的组件把程序的不同关注面分离。如果为程序开发一套自定义的组件库，那么就能以最适合业务场景的方式来展示你的用户界面。

组合实例

一起来使用 Facebook Graph API 开发显示个人图片和用户名的简单 Avatar 组件吧。

```
var Avatar = React.createClass({
  render: function() {
    return (
```

```
    <div>
      <ProfilePic username={this.props.username} />
      <ProfileLink username={this.props.username} />
    </div>
  );
}
});

var ProfilePic = React.createClass({
  render: function() {
    return (
      <img src={'http://graph.facebook.com/' + this.props.username +
    });
  }
});

var ProfileLink = React.createClass({
  render: function() {
    return (
      <a href={'http://www.facebook.com/' + this.props.username}>
        {this.props.username}
      </a>
    );
  }
});

React.render(
  <Avatar username="pwh" />,
  document.getElementById('example')
);
```

从属关系

上面例子中，`Avatar` 拥有 `ProfilePic` 和 `ProfileLink` 的实例。拥有者 就是给其它组件设置 `props` 的那个组件。更正式地说，如果组件 `Y` 在 `render()` 方法是创建了组件 `X`，那么 `Y` 就拥有 `X`。上面讲过，组件不能修改自身的 `props` - 它们总是与它们拥有者设置的保持一致。这是保持用户界面一致性的关键性原则。

把从属关系与父子关系加以区别至关重要。从属关系是 React 特有的，而父子关系简单来讲就是DOM 里的标签的

关系。在上一个例子中，`Avatar` 拥有 `div`、`ProfilePic` 和 `ProfileLink` 实例，`div` 是 `ProfilePic` 和 `ProfileLink` 实例的**父级**（但不是拥有者）。

子级

实例化 React 组件时，你可以在开始标签和结束标签之间引用在 React 组件或者 Javascript 表达式：

```
<Parent><Child /></Parent>
```

`Parent` 能通过专门的 `this.props.children` props 读取子级。`this.props.children` 是一个不透明的数据结构：通过 [React.Children 工具类](#) 来操作。

子级校正（Reconciliation）

校正就是每次 `render` 方法调用后 React 更新 DOM 的过程。一般情况下，子级会根据它们被渲染的顺序来做校正。例如，下面代码描述了两次渲染的过程：

```
// 第一次渲染
<Card>
  <p>Paragraph 1</p>
  <p>Paragraph 2</p>
</Card>
// 第二次渲染
<Card>
  <p>Paragraph 2</p>
</Card>
```

直观来看，只是删除了 `Paragraph 1`。事实上，React 先更新第一个子级的内容，然后删除最后一个组件。React 是根据子级的 `_顺序_` 来校正的。

子组件状态管理

对于大多数组件，这没什么大碍。但是，对于使用 `this.state` 来在多次渲染过程中里维持数据的状态化组件，这样做潜在很多问题。

多数情况下，可以通过隐藏组件而不是删除它们来绕过这些问题。

```
// 第一次渲染
<Card>
  <p>Paragraph 1</p>
  <p>Paragraph 2</p>
</Card>
// 第二次渲染
<Card>
  <p style={{display: 'none'}}>Paragraph 1</p>
  <p>Paragraph 2</p>
</Card>
```

动态子级

如果子组件位置会改变（如在搜索结果中）或者有新组件添加到列表开头（如在流中）情况会变得更加复杂。如果子级要在多个渲染阶段保持自己的特征和状态，在这种情况下，你可以通过给子级设置惟一标识的 `key` 来区分。

```
render: function() {
  var results = this.props.results;
  return (
    <ol>
      {results.map(function(result) {
        return <li key={result.id}>{result.text}</li>;
      })}
    </ol>
  );
}
```

当 React 校正带有 `key` 的子级时，它会确保它们被重新排序（而不是破坏）或者删除（而不是重用）。务必把

`key` 添加到子级数组里组件本身上，而不是每个子级内部最外层 HTML 上：

```
// 错误！
var ListItemWrapper = React.createClass({
  render: function() {
    return <li key={this.props.data.id}>{this.props.data.text}</li>;
  }
});

var MyComponent = React.createClass({
  render: function() {
    return (
      <ul>
        {this.props.results.map(function(result) {
          return <ListItemWrapper data={result}/>;
        })}
      </ul>
    );
  }
});

// 正确 :)
var ListItemWrapper = React.createClass({
  render: function() {
    return <li>{this.props.data.text}</li>;
  }
});

var MyComponent = React.createClass({
  render: function() {
    return (
      <ul>
        {this.props.results.map(function(result) {
          return <ListItemWrapper key={result.id} data={result}/>;
        })}
      </ul>
    );
  }
});
```

也可以传递 object 来做有 key 的子级。object 的 key 会被当作每个组件的 `key`。但是一定要牢记 JavaScript 并不总是保证属性的顺序会被保留。实际上浏览器一般会保留属性的顺序，**除了** 使用 32 位无符号数字做为 key 的属性。数字型属性会按大小排序并且排在其它属性前面。一

旦发生这种情况，React 渲染组件的顺序就是混乱。可能在 `key` 前面加一个字符串前缀来避免：

```
render: function() {
  var items = {};

  this.props.results.forEach(function(result) {
    // 如果 result.id 看起来是一个数字（比如短哈希），那么
    // 对象字面量的顺序就得不到保证。这种情况下，需要添加前缀
    // 来确保 key 是字符串。
    items['result-' + result.id] = <li>{result.text}</li>;
  });

  return (
    <ol>
      {items}
    </ol>
  );
}
```

数据流

React 里，数据通过上面介绍过的 `props` 从拥有者流向归属者。这就是高效的单向数据绑定(one-way data binding)：拥有者通过它的 `props` 或 `state` 计算出一些值，并把这些值绑定到它们拥有的组件的 `props` 上。因为这个过程会递归地调用，所以数据变化会自动在所有被使用的地方自动反映出来。

性能提醒

你或许会担心如果一个拥有者有大量子级时，对于数据变化做出响应非常耗费性能。值得庆幸的是执行 JavaScript 非常的快，而且 `render()` 方法一般比较简单，所以在大部分应用里这样做速度极快。此外，性能的瓶颈大多是因

为 DOM 更新，而非 JS 执行，而且 React 会通过批量更新和变化检测来优化性能。

但是，有时候需要做细粒度的性能控制。这种情况下，可以重写 `shouldComponentUpdate()` 方法返回 `false` 来让 React 跳过对子树的处理。参考 [React reference docs](#) 了解更多。

注意：

如果在数据变化时让 `shouldComponentUpdate()` 返回 `false`，React 就不能保证用户界面同步。当使用它的时候一定确保你清楚到底做了什么，并且只在遇到明显性能问题的时候才使用它。不要低估 JavaScript 的速度，DOM 操作通常才是慢的原因。

设计接口的时候，把通用的设计元素（按钮，表单框，布局组件等）拆成接口良好定义的可复用的组件。这样，下次开发相同界面程序时就可以写更少的代码，也意味着更高的开发效率，更少的 Bug 和更少的程序体积。

Prop 验证

随着应用不断变大，保证组件被正确使用变得非常有用。为此我们引入 `propTypes`。`React.PropTypes` 提供很多验证器 (validator) 来验证传入数据的有效性。当向 props 传入无效数据时，JavaScript 控制台会抛出警告。注意为了性能考虑，只在开发环境验证 `propTypes`。下面用例子来说明不同验证器的区别：

```
React.createClass({
  propTypes: {
    // 可以声明 prop 为指定的 JS 基本类型。默认
    // 情况下，这些 prop 都是可传可不传的。
    optionalArray: React.PropTypes.array,
    optionalBool: React.PropTypes.bool,
    optionalFunc: React.PropTypes.func,
    optionalNumber: React.PropTypes.number,
    optionalObject: React.PropTypes.object,
    optionalString: React.PropTypes.string,

    // 所有可以被渲染的对象：数字，
    // 字符串，DOM 元素或包含这些类型的数组。
    optionalNode: React.PropTypes.node,

    // React 元素
    optionalElement: React.PropTypes.element,

    // 用 JS 的 instanceof 操作符声明 prop 为类的实例。
    optionalMessage: React.PropTypes.instanceOf(Message),

    // 用 enum 来限制 prop 只接受指定的值。
    optionalEnum: React.PropTypes.oneOf(['News', 'Photos']),

    // 指定的多个对象类型中的一个
```

```

optionalUnion: React.PropTypes.oneOfType([
  React.PropTypes.string,
  React.PropTypes.number,
  React.PropTypes.instanceOf(Message)
]),

// 指定类型组成的数组
optionalArrayOf: React.PropTypes.arrayOf(React.PropTypes.number)

// 指定类型的属性构成的对象
optionalObjectOf: React.PropTypes.objectOf(React.PropTypes.number)

// 特定形状参数的对象
optionalObjectWithShape: React.PropTypes.shape({
  color: React.PropTypes.string,
  fontSize: React.PropTypes.number
}),

// 以后任意类型加上 `isRequired` 来使 prop 不可空。
requiredFunc: React.PropTypes.func.isRequired,

// 不可空的任意类型
requiredAny: React.PropTypes.any.isRequired,

// 自定义验证器。如果验证失败需要返回一个 Error 对象。不要直接
// 使用 `console.warn` 或抛异常，因为这样 `oneOfType` 会失效。
customProp: function(props, propName, componentName) {
  if (!/matchme/.test(props[propName])) {
    return new Error('Validation failed!');
  }
}
},
/* ... */
});

```

默认 Prop 值

React 支持以声明式的方式来定义 `props` 的默认值。

```

var ComponentWithDefaultProps = React.createClass({
  getDefaultProps: function() {
    return {
      value: 'default value'
    };
  }
/* ... */
});

```

当父级没有传入 props 时，`getDefaultProps()` 可以保证 `this.props.value` 有默认值，注意 `getDefaultProps` 的结果会被缓存。得益于此，你可以直接使用 props，而不必手动编写一些重复或无意义的代码。

传递 Props：小技巧

有一些常用的 React 组件只是对 HTML 做简单扩展。通常，你想少写点代码来把传入组件的 props 复制到对应的 HTML 元素上。这时 JSX 的 *spread* 语法会帮到你：

```
var CheckLink = React.createClass({
  render: function() {
    // 这样会把 CheckList 所有的 props 复制到 <a>
    return <a {...this.props}>{'\ '}{this.props.children}</a>;
  }
});

React.render(
  <CheckLink href="/checked.html">
    Click here!
  </CheckLink>,
  document.getElementById('example')
);
```

单个子级

`React.PropTypes.element` 可以限定只能有一个子级传入。

```
var MyComponent = React.createClass({
  propTypes: {
    children: React.PropTypes.element.isRequired
  },

  render: function() {
    return (
      <div>
        {this.props.children} // 有且仅有一个元素，否则会抛异常。
      </div>
    );
  }
});
```

```
    );  
  }  
  
});
```

Mixins

组件是 React 里复用代码最佳方式，但是有时一些复杂的组件间也需要共用一些功能。有时会被称为 [跨切面关注点](#)。React 使用 `mixins` 来解决这类问题。

一个通用的场景是：一个组件需要定期更新。用

`setInterval()` 做很容易，但当不需要它的时候取消定时器来节省内存是非常重要的。React 提供 [生命周期方法](#) 来告知组件创建或销毁的时间。下面来做一个简单的 mixin，使用 `setInterval()` 并保证在组件销毁时清理定时器。

```
var SetIntervalMixin = {  
  componentWillMount: function() {  
    this.intervals = [];  
  },  
  setInterval: function() {  
    this.intervals.push(setInterval.apply(null, arguments));  
  },  
  componentWillUnmount: function() {  
    this.intervals.map(clearInterval);  
  }  
};  
  
var TickTock = React.createClass({  
  mixins: [SetIntervalMixin], // 引用 mixin  
  getInitialState: function() {  
    return {seconds: 0};  
  },  
  componentDidMount: function() {  
    this.setInterval(this.tick, 1000); // 调用 mixin 的方法  
  },  
  tick: function() {  
    this.setState({seconds: this.state.seconds + 1});  
  },  
  render: function() {  
    return (  

```

```
        <p>
          React has been running for {this.state.seconds} seconds.
        </p>
      );
    }
  });

  React.render(
    <TickTock />,
    document.getElementById('example')
  );
```

关于 mixin 值得一提的优点是，如果一个组件使用了多个 mixin，并且有多个 mixin 定义了同样的生命周期方法（如：多个 mixin 都需要在组件销毁时做资源清理操作），所有这些生命周期方法都保证会被执行到。方法执行顺序是：首先按 mixin 引入顺序执行 mixin 里方法，最后执行组件内定义的方法。

React 里有一个非常常用的模式就是对组件做一层抽象。组件对外公开一个简单的属性（Props）来实现功能，但内部细节可能有非常复杂的实现。

可以使用 [JSX 展开属性](#) 来合并现有的 props 和其它值：

```
return <Component {...this.props} more="values" />;
```

如果不使用 JSX，可以使用一些对象辅助方法如 ES6 的 `Object.assign` 或 Underscore `_.` `extend`。

```
return Component(Object.assign({}, this.props, { more: 'values' }));
```

下面的教程介绍一些最佳实践。使用了 JSX 和 ES7 的还在试验阶段的特性。

手动传递

大部分情况下你应该显式地向下传递 props。这样可以确保只公开你认为是安全的内部 API 的子集。

```
var FancyCheckbox = React.createClass({
  render: function() {
    var fancyClass = this.props.checked ? 'FancyChecked' : 'FancyUnc
    return (
      <div className={fancyClass} onClick={this.props.onClick}>
        {this.props.children}
      </div>
    );
  }
});

React.render(
  <FancyCheckbox checked={true} onClick={console.log.bind(console)}>
    Hello world!
  </FancyCheckbox>,
  document.body
);
```

但 `name` 这个属性怎么办？还有 `title`、`onMouseOver` 这些 props？

在 JSX 里使用 `...` 传递

有时把所有属性都传下去是不安全或啰嗦的。这时可以使用 [解构赋值](#) 中的剩余属性特性来把未知属性批量提取出来。

列出所有要当前使用的属性，后面跟着 `...other`。

```
var { checked, ...other } = this.props;
```

这样能确保把所有 props 传下去，除了那些已经被使用了的。

```
var FancyCheckbox = React.createClass({
  render: function() {
    var { checked, ...other } = this.props;
    var fancyClass = checked ? 'FancyChecked' : 'FancyUnchecked';
    // `other` 包含 { onClick: console.log } 但 checked 属性除外
    return (
      <div {...other} className={fancyClass} />
    );
  }
});

React.render(
  <FancyCheckbox checked={true} onClick={console.log.bind(console)}>
    Hello world!
  </FancyCheckbox>,
  document.body
);
```

注意：

上面例子中，`checked` 属性也是一个有效的 DOM 属性。如果你没有使用解构赋值，那么可能无意中把它传

下去。

在传递这些未知的 `other` 属性时，要经常使用解构赋值模式。

```
var FancyCheckbox = React.createClass({
  render: function() {
    var fancyClass = this.props.checked ? 'FancyChecked' : 'FancyUnc
    // 反模式：`checked` 会被传到里面的组件里
    return (
      <div {...this.props} className={fancyClass} />
    );
  }
});
```

使用和传递同一个 Prop

如果组件需要使用一个属性又要往下传递，可以直接使用 `checked={checked}` 再传一次。这样做比传整个 `this.props` 对象要好，因为更利于重构和语法检查。

```
var FancyCheckbox = React.createClass({
  render: function() {
    var { checked, title, ...other } = this.props;
    var fancyClass = checked ? 'FancyChecked' : 'FancyUnchecked';
    var fancyTitle = checked ? 'X ' + title : 'O ' + title;
    return (
      <label>
        <input {...other}
          checked={checked}
          className={fancyClass}
          type="checkbox"
        />
        {fancyTitle}
      </label>
    );
  }
});
```

注意：

顺序很重要，把 `{...other}` 放到 JSX props 前面会使它不被覆盖。上面例子中我们可以保证 input 的 type 是 `"checkbox"`。

剩余属性和展开属性 ...

剩余属性可以把对象剩下的属性提取到一个新的对象。会把所有在解构赋值中列出的属性剔除。

这是 [ES7 草案](#) 中的试验特性。

```
var { x, y, ...z } = { x: 1, y: 2, a: 3, b: 4 };  
x; // 1  
y; // 2  
z; // { a: 3, b: 4 }
```

注意:

使用 [JSX 命令行工具](#) 配合 `--harmony` 标记来启用 ES7 语法。

使用 Underscore 来传递

如果不使用 JSX，可以使用一些库来实现相同效果。

Underscore 提供 `_.omit` 来过滤属性，`_.extend` 复制属性到新的对象。

```
javascript var FancyCheckbox = React.createClass({ render: function()
```

诸如 `<input>`、`<textarea>`、`<option>` 这样的表单组件不同于其他组件，因为他们可以通过用户交互发生变化。这些组件提供的界面使响应用户交互的表单数据处理更加容易。

关于 `<form>` 事件详情请查看 [表单事件](#)。

交互属性

表单组件支持几个受用户交互影响的属性：

- `value`，用于 `<input>`、`<textarea>` 组件。
- `checked`，用于类型为 `checkbox` 或者 `radio` 的 `<input>` 组件。
- `selected`，用于 `<option>` 组件。

在 HTML 中，`<textarea>` 的值通过子节点设置；在 React 中则应该使用 `value` 代替。

表单组件可以通过 `onChange` 回调函数来监听组件变化。当用户做出以下交互时，`onChange` 执行并通过浏览器做出响应：

- `<input>` 或 `<textarea>` 的 `value` 发生变化时。
- `<input>` 的 `checked` 状态改变时。
- `<option>` 的 `selected` 状态改变时。

和所有 DOM 事件一样，所有的 HTML 原生组件都支持 `onChange` 属性，而且可以用来监听冒泡的 `change` 事件。

受限组件

设置了 `value` 的 `<input>` 是一个_受限_组件。对于受限的 `<input>`，渲染出来的 HTML 元素始终保持 `value` 属性的值。例如：

```
render: function() {  
  return <input type="text" value="Hello!" />;  
}
```

上面的代码将渲染出一个值为 `Hello!` 的 `input` 元素。用户在渲染出来的元素里输入任何值都不起作用，因为 React 已经赋值为 `Hello!`。如果想响应更新用户输入的值，就得使用 `onChange` 事件：

```
getInitialState: function() {  
  return {value: 'Hello!'};  
},  
handleChange: function(event) {  
  this.setState({value: event.target.value});  
},  
render: function() {  
  var value = this.state.value;  
  return <input type="text" value={value} onChange={this.handleCha  
}
```

上面的代码中，React 将用户输入的值更新到 `<input>` 组件的 `value` 属性。这样实现响应或者验证用户输入的界面就很容易了。例如：

```
handleChange: function(event) {  
  this.setState({value: event.target.value.substr(0, 140)});  
}
```

上面的代码接受用户输入，并截取前 140 个字符。

不受限组件

没有设置 `value` (或者设为 `null`) 的 `<input>` 组件是一个_不受限_组件。对于不受限的 `<input>` 组件，渲染出来的元素直接反应用户输入。例如：

```
render: function() {  
  return <input type="text" />;  
}
```

上面的代码将渲染出一个空值的输入框，用户输入将立即反应到元素上。和受限元素一样，使用 `onChange` 事件可以监听值的变化。

如果想给组件设置一个非空的初始值，可以使用

`defaultValue` 属性。例如：

```
render: function() {  
  return <input type="text" defaultValue="Hello!" />;  
}
```

上面的代码渲染出来的元素和**受限组件**一样有一个初始值，但这个值用户可以改变并会反应到界面上。

同样地，类型为 `radio`、`checkbox` 的 `<input>` 支持 `defaultChecked` 属性，`<select>` 支持 `defaultValue` 属性。

```
render: function() {  
  return (  
    <div>  
      <input type="radio" name="opt" defaultChecked /> Option  
      <input type="radio" name="opt" /> Option 2  
      <select defaultValue="C">  
        <option value="A">Apple</option>  
        <option value="B">Banana</option>  
        <option value="C">Cranberry</option>  
      </select>  
    </div>  
  )  
}
```

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

高级主题

为什么使用受限组件？

在 React 中使用诸如 `<input>` 的表单组件时，遇到了一个在传统 HTML 中没有的挑战。

比如下面的代码：

```
<input type="text" name="title" value="Untitled" />
```

在 HTML 中将渲染初始值为 `Untitled` 的输入框。用户改变输入框的值时，节点的 `value` 属性 (*property*) 将随之变化，但是 `node.getAttribute('value')` 还是会返回初始设置的值 `Untitled`。

与 HTML 不同，React 组件必须在任何时间点描绘视图的状态，而不仅仅是在初始化时。比如在 React 中：

```
render: function() {
  return <input type="text" name="title" value="Untitled" />;
}
```

该方法在任何时间点渲染组件以后，输入框的值就应该始终为 `Untitled`。

为什么 `<textarea>` 使用 `value` 属性？

在 HTML 中，`<textarea>` 的值通常使用子节点设置：

```
<!-- 反例：在 React 中不要这样使用！ -->
<textarea name="description">This is the description </textarea>
```



```
<textarea name="description">this is the description.</textarea>
```

对 HTML 而言，让开发者设置多行的值很容易。但是，React 是 JavaScript，没有字符限制，可以使用 `\n` 实现换行。简言之，React 已经有 `value`、`defaultValue` 属性，`</textarea>` 组件的子节点扮演什么角色就有点模棱两可了。基于此，设置 `<textarea>` 值时不应该使用子节点：

```
<textarea name="description" value="This is a description." />
```

如果非要使用子节点，效果和使用 `defaultValue` 一样。

为什么 `<select>` 使用 `value` 属性

HTML 中 `<select>` 通常使用 `<option>` 的 `selected` 属性设置选中状态；React 为了更方面的控制组件，采用以下方式代替：

```
<select value="B">
  <option value="A">Apple</option>
  <option value="B">Banana</option>
  <option value="C">Cranberry</option>
</select>
```

如果是不受限组件，则使用 `defaultValue`。

注意：

给 `value` 属性传递一个数组，可以选中多个选项：

```
<select multiple={true} value={['B', 'C']}>。
```

React提供了强大的抽象，让你在大多数应用场景中不再直接操作DOM，但是有时你需要简单地调用底层的API，或者借助于第三方库或已有的代码。

虚拟DOM

React是很快，因为它从不直接操作DOM。React在内存中维护一个快速响应的DOM描述。`render()`方法返回一个DOM的描述，React能够利用内存中的描述来快速地计算出差异，然后更新浏览器中的DOM。

另外，React实现了一个完备的虚拟事件系统，尽管各个浏览器都有自己的怪异行为，React确保所有事件对象都符合W3C规范，并且持续冒泡，用一种高性能的方式跨浏览器（and everything bubbles consistently and in a performant way cross-browser）。你甚至可以在IE8中使用一些HTML5的事件！

大多数时候你应该呆在React的“虚拟浏览器”世界里面，因为它性能更好，并且容易思考。但是，有时你简单地需要调用底层的API，或许借助于第三方的类似于jQuery插件这种库。React为你提供了直接使用底层DOM API的途径。

Refs和getDOMNode()

为了和浏览器交互，你将需要对DOM节点的引用。每一个挂载的React组件有一个`getDOMNode()`方法，你可以调用这个方法来获取对该节点的引用。

注意：

`getDOMNode()`仅在挂载的组件上有效（也就是说，组件已经被放进了DOM中）。如果你尝试在一个未被挂载的组件上调用这个函数（例如在创建组件的`render()`函数中调用`getDOMNode()`），将会抛出异常。

为了获取一个到React组件的引用，你可以使用`this`来得到当前的React组件，或者你可以使用refs来指向一个你拥有的组件。它们像这样工

作：

```
var MyComponent = React.createClass({
  handleClick: function() {
    // Explicitly focus the text input using the raw DOM API.
    this.refs.myTextInput.getDOMNode().focus();
  },
  render: function() {
    // The ref attribute adds a reference to the component to
    // this.refs when the component is mounted.
    return (
      <div>
        <input type="text" ref="myTextInput" />
        <input
          type="button"
          value="Focus the text input"
          onClick={this.handleClick}
        />
      </div>
    );
  }
});

React.render(
  <MyComponent />,
  document.getElementById('example')
);
```

更多关于Refs

为了学习更多有关Refs的内容，包括如何有效地使用它们，参考我们的[更多关于Refs](#)文档。

组件生命周期

组件的生命周期包含三个主要部分：

- **挂载**：组件被插入到DOM中。
- **更新**：组件被重新渲染，查明DOM是否应该刷新。
- **移除**：组件从DOM中移除。

React提供生命周期方法，你可以在这些方法中放入自己的代码。我们提供**will**方法，会在某些行为发生之前调用，和**did**方法，会在某些行为发生之后调用。

挂载

- `getInitialState(): object` 在组件被挂载之前调用。状态化的组件应该实现这个方法，返回初始的state数据。

- `componentWillMount()` 在挂载发生之前立即被调用。
- `componentDidMount()` 在挂载结束之后马上被调用。需要DOM节点的初始化操作应该放在这里。

更新

- `componentWillReceiveProps(object nextProps)` 当一个挂载的组件接收到新的props的时候被调用。该方法应该用于比较 `this.props` 和 `nextProps`，然后使用 `this.setState()` 来改变state。
- `shouldComponentUpdate(object nextProps, object nextState): boolean` 当组件做出是否要更新DOM的决定的时候被调用。实现该函数，优化 `this.props` 和 `nextProps`，以及 `this.state` 和 `nextState` 的比较，如果不需要React更新DOM，则返回false。
- `componentWillUpdate(object nextProps, object nextState)` 在更新发生之前被调用。你可以在这里调用 `this.setState()`。
- `componentDidUpdate(object prevProps, object prevState)` 在更新发生之后调用。

移除

- `componentWillUnmount()` 在组件移除和销毁之前被调用。清理工作应该放在这里。

挂载的方法 (Mounted Methods)

`_挂载的_`复合组件也支持如下方法：

- `getDOMNode(): DOMElement` 可以在任何挂载的组件上面调用，用于获取一个指向它的渲染DOM节点的引用。
- `forceUpdate()` 当你知道一些很深的组件state已经改变了的时候，可以在该组件上面调用，而不是使用 `this.setState()`。

跨浏览器支持和兼容代码 (Browser Support and Polyfills)

在Facebook，我们支持低版本的浏览器，包括IE8。我们已经写好兼容代码很长时间了，这能让我们写有远见的JS。这意味着我们没有零散的骇客代码充斥在我们的代码库里面，并且我们依然能够预计我们的代码“正常工作起来”。例如，不使用 `+new Date()`，我们能够写

`Date.now()`。At Facebook, we support older browsers, including IE8. We've had polyfills in place for a long time to allow us to write forward-thinking JS. This means we don't have a bunch of hacks scattered throughout our codebase and we can still expect our code to “just work”. For example, instead of seeing `+new Date()`, we can just write `Date.now()`. Since the open source React is the same as what we use internally, we've carried over this philosophy of using forward thinking JS.

In addition to that philosophy, we've also taken the stance that we, as authors of a JS library, should not be shipping polyfills as a part of our library. If every library did this, there's a good chance you'd be sending down the same polyfill multiple times, which could be a sizable chunk of dead code. If your product needs to support older browsers, chances are you're already using something like [es5-shim](#).

支持低版本浏览器的兼容代码

[kriskowal的es5-shim](#) `es5-shim.js` 提供了以下react需要的api：

- `Array.isArray`
- `Array.prototype.every`
- `Array.prototype.forEach`
- `Array.prototype.indexOf`
- `Array.prototype.map`
- `Date.now`
- `Function.prototype.bind`
- `Object.keys`
- `String.prototype.split`
- `String.prototype.trim`

[kriskowal的es5-shim](#) `es5-sham.js` 同样提供了以下react需要的api：

- `Object.create`
- `Object.freeze`

The unminified build of React needs the following from [paulmillr's console-polyfill](#).

- `console.*`

When using HTML5 elements in IE8 including `<div>`, ``, and `<script>`, it's also necessary to include [html5shiv](#) or a similar script.

Cross-browser Issues

Although React is pretty good at abstracting browser differences, some browsers are limited or present quirky behaviors that we couldn't find a workaround for.

onScroll event on IE8

On IE8 the `onScroll` event doesn't bubble and IE8 doesn't have an API to define handlers to the capturing phase of an event, meaning there is no way for React to listen to these events. Currently a handler to this event is ignored on IE8.

See the [onScroll doesn't work in IE8](#) GitHub issue for more information. We've carried over this philosophy of using forward thinking JS.

In addition to that philosophy, we've also taken the stance that we, as authors of a JS library, should not be shipping polyfills as a part of our library. If every library did this, there's a good chance you'd be sending down the same polyfill multiple times, which could be a sizable chunk of dead code. If your product needs to support older browsers, chances are you're already using something like [es5-shim](#).

Polyfills Needed to Support Older Browsers

`es5-shim.js` from [kriskowal's es5-shim](#) provides the following that React needs:

- `Array.isArray`
- `Array.prototype.every`

- `Array.prototype.forEach`
- `Array.prototype.indexOf`
- `Array.prototype.map`
- `Date.now`
- `Function.prototype.bind`
- `Object.keys`
- `String.prototype.split`
- `String.prototype.trim`

`es5-sham.js` , also from [kriskowal's es5-shim](#), provides the following that React needs:

- `Object.create`
- `Object.freeze`

The unminified build of React needs the following from [paulmillr's console-polyfill](#).

- `console.*`

When using HTML5 elements in IE8 including `<div>` , `` , and `<input>` , it's also necessary to include [html5shiv](#) or a similar script.

Cross-browser Issues

Although React is pretty good at abstracting browser differences, some browsers are limited or present quirky behaviors that we couldn't find a workaround for.

onScroll event on IE8

On IE8 the `onScroll` event doesn't bubble and IE8 doesn't have an API to define handlers to the capturing phase of an event, meaning there is no way for React to listen to these events. Currently a handler to this event is ignored on IE8.

See the [onScroll doesn't work in IE8](#) GitHub issue for more information.

每个项目使用不同的系统来构建和部署JavaScript。我们尝试尽量让React环境无关。

React

CDN托管的React

我们在我们的[下载页面](#)提供了React的CDN托管版本。这些预构建的文件使用UMD模块格式。直接简单地把它们放在`<script>`标签中将会给你环境的全局作用域引入一个`React`对象。

React也可以在CommonJS和AMD环境下正常工作。

使用主分支

我们在[GitHub仓库](#)的主分支上有一些构建指令。我们在`build/modules`下构建了符合CommonJS模块规范的树形目录，你可以放置在任何环境或者使用任何打包工具，只要支持CommonJS规范。

JSX

浏览器中的JSX转换

如果你喜欢使用JSX，我们在[我们的下载页面](#)提供了一个用于开发的浏览器中的JSX转换器。简单地用一个“`<script>`”标签来触发JSX转换器。

注意：

浏览器中的JSX转换器是相当大的，并且会在客户端导致无谓的计算，这些计算是可以避免的。不要在生产环

境使用 - 参考下一节。

生产环境化：预编译JSX

如果你有[npm](#)，你可以简单地运行

`npm install -g react-tools` 来安装我们的命令行 `jsx` 工具。这个工具会把使用JSX语法的文件转换成纯的可以直接在浏览器里面运行起来的JavaScript文件。它也会为你监视目录，然后自动转换变化的文件；例如：

`jsx --watch src/ build/`。运行 `jsx --help` 来查看更多关于如何使用这个工具的信息。

有用的开源项目

开源社区开发了在几款编辑器中集成JSX的插件和构建系统。点击[JSX集成](#)查看所有内容。

`React.addons` 是为了构建 React 应用而放置的一些有用工具的地方。此功能应当被视为实验性的，但最终将会被添加进核心代码中或者有用的工具库中：

- `TransitionGroup` 和 `CSSTransitionGroup`，用于处理动画和过渡，这些通常实现起来都不简单，例如在一个组件移除之前执行一段动画。
- `LinkingStateMixin`，用于简化用户表单输入数据和组件 state 之间的双向数据绑定。
- `classSet`，用于更加干净简洁地操作 DOM 中的 `class` 字符串。
- `cloneWithProps`，用于实现 React 组件浅复制，同时改变它们的 props。
- `update`，一个辅助方法，使得在 JavaScript 中处理不可变数据更加容易。
- `PureRenderMixin`，在某些场景下的性能检测器。

以下插件只存在于 React 开发版（未压缩）：

- `TestUtils`，简单的辅助工具，用于编写测试用例（仅存在于未压缩版）。
- `Perf`，用于性能测评，并帮助你检查出可优化的功能点。

要使用这些插件，需要用 `react-with-addons.js`（和它的最小化副本）替换常规的 `React.js`。

当通过 npm 使用 react 包的时候，只要简单地用

`require('react/addons')` 替换 `require('react')` 来得

到带有所有插件的React。

React为动画提供一个 `ReactTransitionGroup` 插件组件作为一个底层的API，一个 `ReactCSSTransitionGroup` 来简单地实现基本的CSS动画和过渡。

高级API：

`ReactCSSTransitionGroup`

`ReactCSSTransitionGroup` 是基于 `ReactTransitionGroup` 的，在React组件进入或者离开DOM的时候，它是一种简单地执行CSS过渡和动画的方式。这个的灵感来自于优秀的[ng-animate](#)库。

快速开始

`ReactCSSTransitionGroup` 是 `ReactTransitions` 的接口。这是一个简单的元素，包含了所有你对其动画感兴趣的组件。这里是一个例子，例子中我们让列表项淡入淡出。

```
var ReactCSSTransitionGroup = React.addons.CSSTransitionGroup;

var TodoList = React.createClass({
  getInitialState: function() {
    return {items: ['hello', 'world', 'click', 'me']};
  },
  handleAdd: function() {
    var newItems =
      this.state.items.concat([prompt('Enter some text')]);
    this.setState({items: newItems});
  },
  handleRemove: function(i) {
    var newItems = this.state.items;
    newItems.splice(i, 1);
    this.setState({items: newItems});
  },
  render: function() {
    var items = this.state.items.map(function(item, i) {
      return (
        <div key={item} onClick={this.handleRemove.bind(this, i)}>
          {item}
        </div>
      );
    });
    return (
      <ReactCSSTransitionGroup
        transitionName="example"
        transitionEnterTimeout={500}
        transitionLeaveTimeout={500}>
        {items}
      </ReactCSSTransitionGroup>
    );
  }
});
```

```
    );
    }.bind(this));
    return (
      <div>
        <button onClick={this.handleAdd}>Add Item</button>
        <ReactCSSTransitionGroup transitionName="example">
          {items}
        </ReactCSSTransitionGroup>
      </div>
    );
  }
});
```

注意：

你必须为 `ReactCSSTransitionGroup` 的所有子级提供 **键 属性**，即使只渲染一项。这就是React确定哪一个子级插入了，移除了，或者停留在那里。

在这个组件当中，当一个新的项被添加到 `ReactCSSTransitionGroup`，它将会被添加 `example-enter` 类，然后在下一时刻被添加 `example-enter-active` CSS类。这是一个基于 `transitionName` prop的约定。

你可以使用这些类来触发一个CSS动画或者过渡。例如，尝试添加这段CSS代码，然后插入一个新的列表项：

```
.example-enter {
  opacity: 0.01;
  transition: opacity .5s ease-in;
}

.example-enter.example-enter-active {
  opacity: 1;
}
```

你将注意到，当你尝试移除一项的时候，

`ReactCSSTransitionGroup` 保持该项在DOM里。如果你正使用一个带有插件的未压缩的React构建版本，你将会看到一条警告：React期待一次动画或者过渡发生。那是因为 `ReactCSSTransitionGroup` 保持你的DOM元素一直在页面上，直到动画完成。尝试添加这段CSS代码：

```
.example-leave {
  opacity: 1;
  transition: opacity .5s ease-in;
}

.example-leave.example-leave-active {
  opacity: 0.01;
}
```

一组动画必须要挂载了才能生效

为了能够给它的子级应用过渡效果，

`ReactCSSTransitionGroup` 必须已经挂载到了DOM。下面的例子不会生效，因为 `ReactCSSTransitionGroup` 被挂载到新项，而不是新项被挂载到 `ReactCSSTransitionGroup` 里。将这个与上面的[快速开始](#)部分比较一下，看看有什么差异。

```
render: function() {
  var items = this.state.items.map(function(item, i) {
    return (
      <div key={item} onClick={this.handleRemove.bind(this, i)}>
        <ReactCSSTransitionGroup transitionName="example">
          {item}
        </ReactCSSTransitionGroup>
      </div>
    );
  }, this);
  return (
    <div>
      <button onClick={this.handleAdd}>Add Item</button>
      {items}
    </div>
  );
}
```

```
);  
}
```

让一项或者零项动起来 (Animating One or Zero Items)

虽然在上面的例子中，我们渲染了一个项目列表到

`ReactCSSTransitionGroup` 里，

`ReactCSSTransitionGroup` 的子级可以是一个或零个项目。这使它能够让一个元素实现进入和离开的动画。同样，你可以通过移动一个新的元素来替换当前元素。随着新元素的移入，当前元素移出。例如，我们可以由此实现一个简单的图片轮播器：

```
var ReactCSSTransitionGroup = React.addons.CSSTransitionGroup;  
  
var ImageCarousel = React.createClass({  
  propTypes: {  
    imageSrc: React.PropTypes.string.isRequired  
  },  
  render: function() {  
    return (  
      <div>  
        <ReactCSSTransitionGroup transitionName="carousel">  
          <img src={this.props.imageSrc} key={this.props.imageSrc} />  
        </ReactCSSTransitionGroup>  
      </div>  
    );  
  }  
});
```

禁用动画

如果你想，你可以禁用 入场 或者 出场 动画。例如，有些时候，你可能想要一个 入场 动画，不要 出场 动画，但是 `ReactCSSTransitionGroup` 会在移除DOM节点之前等待一个动画完成。你可以给 `ReactCSSTransitionGroup` 添加

`transitionEnter={false}` 或者

`transitionLeave={false}` `props`来禁用这些动画。

注意：

当使用 `ReactCSSTransitionGroup` 的时候，没有办法通知你在过渡效果结束或者在执行动画的时候做一些复杂的运算。如果你想要更多细粒度的控制，你可以使用底层的 `ReactTransitionGroup` API，该API提供了你自定义过渡效果所需要的函数。

底层的API：

`ReactTransitionGroup`

`ReactTransitionGroup` 是动画的基础。它可以通过 `React.addons.TransitionGroup` 得到。当子级被添加或者从其中移除（就像上面的例子）的时候，特殊的生命周期函数就会在它们上面被调用。

`componentWillEnter(callback)`

在组件被添加到已有的 `TransitionGroup` 中的时候，该函数和 `componentDidMount()` 被同时调用。这将会阻塞其它动画触发，直到 `callback` 被调用。该函数不会在 `TransitionGroup` 初始化渲染的时候调用。

`componentDidEnter()`

该函数在传给 `componentWillEnter` 的 `callback` 函数被调用之后调用。

`componentWillLeave(callback)`

该函数在子级从 `ReactTransitionGroup` 中移除的时候调用。虽然子级被移除了，`ReactTransitionGroup` 将会使它继续在DOM中，直到 `callback` 被调用。

`componentDidLeave()`

该函数在 `willLeave` `callback` 被调用的时候调用（与 `componentWillUnmount` 是同一时间）。

渲染一个不同的组件

默认情况下 `ReactTransitionGroup` 渲染一个 `span`。你可以通过提供一个 `component` prop来改变这种行为。例如，以下是你将如何渲染一个“：

```
<ReactTransitionGroup component="ul">
  ...
</ReactTransitionGroup>
```

每一个React能渲染的DOM组件都是可用的。但是，`组件` 并不需要是一个DOM组件。它可以是任何你想要的React组件；甚至是你自己已经写好的！

注意：

v0.12之前，当使用DOM组件的时候，`组件` prop需要是一个指向 `React.DOM.*` 的引用。既然组件简单地传递到了 `React.createElement`，它必须是一个字符串。组装的组件必须传递构造函数。

任何额外的、用户定义的属性将会成为已渲染的组件的属性。例如，以下是你将如何渲染一个带有css类的“：

```
<ReactTransitionGroup component="ul" className="animated-list">  
  ...  
</ReactTransitionGroup>
```

`ReactLink` 是一种简单表达React双向绑定的方式。

注意：

如果你是这个框架的初学者，记住 `ReactLink` 对于大多数应用来说都是不需要的，应该谨慎使用。

在React里面，数据流是一个方向的：从拥有者到子节点。这是因为根据[the Von Neumann model of computing](#)，数据仅向一个方向传递。你可以认为它是 单向数据绑定。

然而，有很多应用需要你读取一些数据，然后传回给你的程序。例如，在开发表单的时候，当你接收到用户输入时，你将会频繁地想更新某些React `state`。或者你想在JavaScript中演算布局，然后反应到某些DOM元素的尺寸上。

在React中，你可以通过监听一个“change”事件来实现这个功能，从你的数据源（通常是DOM）读取，然后在你某个组件上调用 `setState()`。“关闭数据流循环”明显会引导写出更加容易理解的和维护的程序。查看[我们的表单文档](#)来获取更多信息。

双向绑定 — 隐式地强制在DOM里面的数据总是和某些React `state` 保持一致 — 是简明的，并且支持非常多的应用。我们已经提供了 `ReactLink`：如上所述，是一种设置通用数据流循环模型的语法糖，或者说“关联”某些数据到React `state`。

注意：

ReactLink仅仅是一个 `onChange / setState()` 模式的简单包装和约定。它不会从根本上改变数据在你的React应用中如何流动。

ReactLink：前后对比

这是一个简单的表单示例，没有使用 `ReactLink`：

```
var NoLink = React.createClass({
  getInitialState: function() {
    return {message: 'Hello!'};
  },
  handleChange: function(event) {
    this.setState({message: event.target.value});
  },
  render: function() {
    var message = this.state.message;
    return <input type="text" value={message} onChange={this.handleC
  }
});
```

这段代码运行地很好，数据如何流动是非常清晰的，但是，如果表单有大量的字段，代码就会很冗长了。让我们使用 `ReactLink` 来减少打字输入：

```
var WithLink = React.createClass({
  mixins: [React.addons.LinkedStateMixin],
  getInitialState: function() {
    return {message: 'Hello!'};
  },
  render: function() {
    return <input type="text" valueLink={this.linkState('message')}
  }
});
```

`LinkStateMixin` 给你的React组件添加一个叫做 `linkState()` 的方法。 `linkState()` 返回一个 `ReactLink` 对象，包含React state当前的值和一个用来改变它的回调函数。

`ReactLink` 对象可以在树中作为props被向上传递或者向下传递，so it's easy (and explicit) to set up two-way binding between a component deep in the hierarchy and state that lives higher in the hierarchy.

注意，对于checkbox的 `value` 属性，有一个特殊的行为，如果checkbox被选中（默认是 `on`），`value` 属性值将会在表单提交的时候发送出去。当checkbox被选中或者取消选中的时候，`value` 属性是不会更新的。对于checkbox，你应该使用 `checkLink` 而不是 `valueLink`：

```
<input type="checkbox" checkedLink={this.linkState('booleanValue')} />
```

底层原理 (Under the Hood)

对于 `ReactLink`，有两块儿：你创建 `ReactLink` 实例的地方和你使用它的地方。为了证明 `ReactLink` 是多么的简单，让我们单独地重写每一块儿，以便显得更加明了。

不带ReactLink的LinkStateMixin (ReactLink Without LinkStateMixin)

```
var WithoutMixin = React.createClass({
  getInitialState: function() {
    return {message: 'Hello!'};
  },
```

```
handleChange: function(newValue) {
  this.setState({message: newValue});
},
render: function() {
  var valueLink = {
    value: this.state.message,
    requestChange: this.handleChange
  };
  return <input type="text" valueLink={valueLink} />;
}
});
```

如你所见，`ReactLink` 对象是非常简单的，仅仅有一个 `value` 和 `requestChange` 属性。`LinkStateMixin` 也同样简单：它仅占据这些字段，用来自于 `this.state` 的值和一个调用 `this.setState()` 的回调函数。（And `LinkStateMixin` is similarly simple: it just populates those fields with a value from `this.state` and a callback that calls `this.setState()` .)

不带valueLink的ReactLink (ReactLink Without valueLink)

```
var WithoutLink = React.createClass({
  mixins: [React.addons.LinkStateMixin],
  getInitialState: function() {
    return {message: 'Hello!'};
  },
  render: function() {
    var valueLink = this.linkState('message');
    var handleChange = function(e) {
      valueLink.requestChange(e.target.value);
    };
    return <input type="text" value={valueLink.value} onChange={hand
  }
});
```

`valueLink` 属性也很简单。它简单地处理 `onChange` 事件，然后调用 `this.props.valueLink.requestChange()` ，

同时也用 `this.props.valueLink.value` 替换
`this.props.value` 。就这么简单！

`classSet()` 是一个简洁的工具，用于简单操作DOM中的 `class` 字符串。

这里是一个常见的场景，处理方式中没有使用

`classSet()`：

```
// inside some `

<Message />

` React component
render: function() {
  var classString = 'message';
  if (this.props.isImportant) {
    classString += ' message-important';
  }
  if (this.props.isRead) {
    classString += ' message-read';
  }
  // 'message message-important message-read'
  return <div className={classString}>Great, I'll be there.</div>;
}
```

这会很快变得单调乏味，因为指定类名的代码很难阅读，并且容易出错。 `classSet()` 解决了这个问题：

```
render: function() {
  var cx = React.addons.classSet;
  var classes = cx({
    'message': true,
    'message-important': this.props.isImportant,
    'message-read': this.props.isRead
  });
  // same final string, but much cleaner
  return <div className={classes}>Great, I'll be there.</div>;
}
```

当使用 `classSet()` 的时候，传递一个对象，对象上的键是你需要或者不需要的CSS类名。对应真值的键将会成为结果字符串的一部分。

`classSet` 也允许传递一些类名作为参数，然后拼接这些类名：

```
render: function() {  
  var cx = React.addons.classSet;  
  var importantModifier = 'message-important';  
  var readModifier = 'message-read';  
  var classes = cx('message', importantModifier, readModifier);  
  // Final string is 'message message-important message-read'  
  return <div className={classes}>Great, I'll be there.</div>;  
}
```

没有更多需要钻研的字符串拼接！

`React.addons.TestUtils` 使得在你选择的测试框架中测试 React 组件变得简单（我们使用 [Jest](#)）。

模拟

```
Simulate.{eventName}(DOMElement element, object eventData)
```

模拟事件在 DOM 节点上派发，附带可选的 `eventData` 事件数据。这可能是在 `ReactTestUtils` 中最有用的工具。

使用示例：

```
var node = this.refs.input.getDOMNode();
React.addons.TestUtils.Simulate.click(node);
React.addons.TestUtils.Simulate.change(node, {target: {value: 'Hello'}});
React.addons.TestUtils.Simulate.keyDown(node, {key: "Enter"});
```

`Simulate` 有一个方法适用于每个事件，这些事件都是 React 能识别的。

renderIntoDocument

```
ReactDOMComponent renderIntoDocument(ReactDOMComponent instance)
```

把一个组件渲染成一个在文档中分离的 DOM 节点。这个函数需要 DOM。

mockComponent

```
object mockComponent(function componentClass, string? mockTagName)
```

传递一个虚拟的组件模块给这个方法，给这个组件扩充一些有用的方法，让组件能够被当成一个 React 组件的仿制

品来使用。这个组件将会变成一个简单的

`（或者是其它标签，如果 `mockTagName`` 提供了的话），包含任何提供的子节点，而不是像往常一样渲染出来。

isElementType

```
boolean isElementType(ReactElement element, function componentClass
```

如果 `element` 是一个类型为 `React componentClass` 的 `React` 元素，则返回 `true`。

isDOMComponent

```
boolean isDOMComponent(ReactComponent instance)
```

如果 `instance` 是一个 `DOM` 组件（例如 `或者` ），则返回 `true`。

isCompositeComponent

```
boolean isCompositeComponent(ReactComponent instance)`
```

如果 `instance` 是一个合成的组件（通过 `React.createClass()` 创建），则返回 `true`。

isCompositeComponentWithType

```
boolean isCompositeComponentWithType(ReactComponent instance, functi
```

如果 `instance` 是一个合成的组件（通过 `React.createClass()` 创建），此组件的类型是 `React componentClass`，则返回 `true`。

findAllInRenderedTree

```
array findAllInRenderedTree(ReactComponent tree, function test)
```

遍历 `tree` 中所有组件，搜集 `test(component)` 返回true的所有组件。就这个本身来说不是很有用，但是它可以为其它测试提供原始数据。

scryRenderedDOMComponentsWithClass

```
array scryRenderedDOMComponentsWithClass(ReactComponent tree, string
```

查找组件的所有实例，这些实例都在渲染后的树中，并且是带有 `className` 类名的DOM组件。

findRenderedDOMComponentWithClass

```
ReactComponent findRenderedDOMComponentWithClass(ReactComponent tree
```

类似于 `scryRenderedDOMComponentsWithClass()`，但是它只返回一个结果，如果有其它满足条件的，则会抛出异常。

scryRenderedDOMComponentsWithTag

```
array scryRenderedDOMComponentsWithTag(ReactComponent tree, string t
```

在渲染后的树中找出所有组件实例，并且是标签名字符合 `tagName` 的DOM组件。

findRenderedDOMComponentWithTag

```
ReactDOM findRenderedDOMComponentWithTag(ReactDOM tree,
```

类似于 `scryRenderedDOMComponentsWithTag()`，但是它只返回一个结果，如果有其它满足条件的，则会抛出异常。

scryRenderedComponentsWithType

```
array scryRenderedComponentsWithType(ReactDOM tree, function c
```

找出所有组件实例，这些组件的类型为 `componentClass`

。

findRenderedComponentWithType

```
ReactDOM findRenderedComponentWithType(ReactDOM tree, fu
```

类似于 `scryRenderedComponentsWithType()`，但是它只返回一个结果，如果有其它满足条件的，则会抛出异常。

在极少数应用场景中，一个组件可能想改变另一个它不拥有的组件的props（就像改变一个组件的 `className`，这个组件又作为 `this.props.children` 传入）。其它的时候，可能想生成传进来的一个组件的多个拷贝。`cloneWithProps()` 使其成为可能。

ReactComponent React.addons.cloneWithProps(ReactComponent component, object? extraProps)

做一个 `component` 的浅复制，合并 `extraProps` 提供的每一个props。`className` 和 `style` props将会被智能合并。

注意：

`cloneWithProps` 并不传递 `key` 到克隆的组件中。如果你希望保留key，将其添加到

`extraProps` 对象：

```
js var clonedComponent = cloneWithProps(originalComponent, { key : originalComponent.key })
```

也一样不会保留。

React让你可以使用任何你想要的数据管理风格，包括数据可变风格。然而，如果你能够在你应用中讲究性能的部分使用不可变数据，就可以很方便地实现一个快速的 `shouldComponentUpdate()` 方法来显著提升你应用的速度。

在JavaScript中处理不可变数据比在语言层面上就设计好要难，像[Clojure](#)。但是，我们提供了一个简单的不可变辅助工具，`update()`，这就让处理这种类型的数据更加简单了，根本不会改变你数据的表示的形式。（Dealing with immutable data in JavaScript is more difficult than in languages designed for it, like [Clojure](#). However, we've provided a simple immutability helper, `update()`, that makes dealing with this type of data much easier, without fundamentally changing how your data is represented.）

主要思想（The main idea）

如果你像这样改变数据：

```
myData.x.y.z = 7;  
// or...  
myData.a.b.push(9);
```

你无法确定哪个数据改变了，因为之前的副本被覆盖了。相反，你需要创建一个新的 `myDate` 副本，仅仅改变需要改变的部分。然后你能够在 `shouldComponentUpdate()`

中使用第三方的相等判断来比较 `myData` 的旧副本和新对象：

```
var newData = deepCopy(myData);
newData.x.y.z = 7;
newData.a.b.push(9);
```

不幸的是，深拷贝是很昂贵的，而且某些时候还不可能完成。你可以通过仅拷贝需要改变的对象，重用未改变的对象来缓解这个问题。不幸的是，在当今的JavaScript里面，这会变得很笨拙：

```
var newData = extend(myData, {
  x: extend(myData.x, {
    y: extend(myData.x.y, {z: 7}),
  }),
  a: extend(myData.a, {b: myData.a.b.concat(9)})
});
```

虽然这能够非常好地提升性能（因为仅仅浅复制 $\log n$ 个对象，重用余下的），但是写起来很痛苦。看看所有的重复书写！这不仅仅是恼人，也提供了一个巨大的出bug的区域。

`update()` 在这种情形下提供了简单的语法糖，使得写这种代码变得更加简单。代码变为：

```
var newData = React.addons.update(myData, {
  x: {y: {z: {$set: 7}}},
  a: {b: {$push: [9]}}
});
```

虽然这种语法糖需要花点精力适应（尽管这是受 [MongoDB's query language](#) 的启发），但是它没有冗余，是静态可分析的，并且比可变的版本少打了很多字。

(While the syntax takes a little getting used to (though it's inspired by [MongoDB's query language](#)) there's no redundancy, it's statically analyzable and it's not much more typing than the mutative version.)

以 `$` 为前缀的键被称作命令。他们“改变”的数据结构被称为目标。 (The `$`-prefixed keys are called *commands*. The data structure they are “mutating” is called the *target*.)

可用的命令 (Available commands)

- `{ $push: array }` 利用 `push()` 把目标上所有的元素放进 数组 (`push()` all the items in `array` on the target.) 。
- `{ $unshift: array }` 利用 `unshift()` 把目标上所有的元素放进 数组 (`unshift()` all the items in `array` on the target.) 。
- `{ $splice: array of arrays }` 对于 `array` 中的每一个元素，用元素提供的参数在目标上调用 `splice()` (for each item in `arrays` call `splice()` on the target with the parameters provided by the item.) 。
- `{ $set: any }` 整体替换目标 (replace the target entirely.) 。
- `{ $merge: object }` 合并目标和 `object` 的键。
- `{ $apply: function }` 传入当前的值到函数，然后用新返回的值更新它 (passes in the current value to the function and updates it with the new returned value.) 。

示例

简单的入栈

```
var initialArray = [1, 2, 3];  
var newArray = update(initialArray, {$push: [4]}); // => [1, 2, 3, 4]
```

`initialArray` 仍然是 `[1, 2, 3]` 。

嵌入的集合

```
var collection = [1, 2, {a: [12, 17, 15]}];  
var newCollection = update(collection, {2: {a: {$splice: [[1, 1, 13,  
// => [1, 2, {a: [12, 13, 14, 15]]}}
```

获取 `collection` 中索引是 2 的对象，然后取得该对象键为 `a` 的值，删掉索引从 1 开始的一个元素（即移除 17），插入 13 和 14。（This accesses `collection` 's index 2, key `a`, and does a splice of one item starting from index 1 (to remove 17) while inserting 13 and 14 .)

根据现有的值更新

```
var obj = {a: 5, b: 3};  
var newObj = update(obj, {b: {$apply: function(x) {return x * 2;}}});  
// => {a: 5, b: 6}  
// This is equivalent, but gets verbose for deeply nested collection  
var newObj2 = update(obj, {b: {$set: obj.b * 2}});
```

(浅) 合并

```
var obj = {a: 5, b: 3};  
var newObj = update(obj, {$merge: {b: 6, c: 7}}); // => {a: 5, b: 6,
```

如果你的React组件的渲染函数是“纯粹的”（换句话说，当传给它同样的props和state，它渲染出同样的效果），在某些场景下，你可以利用这个插件来极大地提升性能。

例如：

```
var PureRenderMixin = require('react').addons.PureRenderMixin;
React.createClass({
  mixins: [PureRenderMixin],

  render: function() {
    return <div className={this.props.className}>foo</div>;
  }
});
```

在底层，该插件实现了 [shouldComponentUpdate](#)，在这里面，它比较当前的props、state和接下来的props、state，当两者相等的时候返回 `false`。

注意：

仅仅是浅比较对象。如果对象包含了复杂的数据结构，深层次的差异可能会产生误判。仅用于拥有简单props和state的组件，或者当你知道很深的数据结构已经变化了的时候使用 `forceUpdate()`。或者，考虑使用 [immutable objects](#)来帮助嵌套数据快速比较。

此外，`shouldComponentUpdate` 会跳过更新整个组件子树。确保所有的子组件也是“纯粹的”。

通常情况下，React在沙箱中是非常快的。但是，在你应用的一些情景中，你需要仔细推敲每一个性能点。React提供了一个函数`shouldComponentUpdate`，通过这个函数，你能够给React的差异检查添加优化代码。

为了给你一个你的应用总体的性能概览，ReactPerf是一个分析工具，告诉你需要把这些钩子函数放在哪里。

注意：

开发的构建过程比生产的构建过程要慢，是因为所有额外逻辑的提供，例如，友好的控制台警告（生产构建时会去掉）。因此，分析工具仅用于指出你应用中相对影响性能的部分。

通用API

当使用 `react-with-addons.js` 在开发模式下构建的时候，这里描述的 `Perf` 对象是以 `React.addons.Perf` 的形式暴露出来的。

`Perf.start()` 和 `Perf.stop()`

开始/停止检测。React的中间操作被记录下来，用于下面的分析。花费一段微不足道的时间的操作会被忽略。

`Perf.printInclusive(measurements)`

打印花费的总时间。如果不传递任何参数，默认打印最后的所有检测记录。它会在控制台中打印一个漂亮的格式化的表格，像这样：

(index)	Owner > component	Inclusive time (ms)	Instances
0	"<root> > App"	15.31	37
1	"App > Box"	6.47	37
2	"Box > Box2"	1.96	37

Perf.printExclusive (measurements)

“独占 (Exclusive)”时间不包含挂载组件的时间：处理 props，getInitialState，调用 componentWillMount 和 componentDidMount，等等。

(index)	Component c...	Total inclu...	Exclusive m...	Exclusive r...	Mount time ...	Render time...	Instances
0	"App"	15.31	1.76	1.23	0.04	0.03	37

Perf.printWasted (measurements)

分析器最有用的部分。

“浪费”的时间被花在根本没有渲染任何东西的组件上，例如界面渲染后保持不变，没有操作DOM元素。

(index)	Owner > component	Wasted time (ms)	Instances
0	"<root> > App"	15.317999947001226	37
1	"App > Box"	6.479999952716753	37
2	"Box > Box2"	1.9630000460892916	37

Perf.printDOM (measurements)

打印底层DOM操作，例如，“设置innerHTML”和“移除节点”。

(index)	data-reactid	type	args
0	""	"set innerHTML"	""<div data-reactid=""\...

高级API

上面的打印方法使用 Perf.getLastMeasurements() 打印好看的结果。

Perf.getLastMeasurements ()

从最后的开始-停止会话中得到检测结果数组。该数组包含对象，每个对象看起来像这样：

```
{
  // The term "inclusive" and "exclusive" are explained below
```

```
"exclusive": {},  
// '.0.0' is the React ID of the node  
  
"inclusive": {".0.0": 0.0670000008540228, ".0": 0.3259999939473346  
"render": {".0": 0.036999990697950125, ".0.0": 0.01000000338535755  
// Number of instances  
"counts": {".0": 1, ".0.0": 1},  
// DOM touches  
"writes": {},  
// Extra debugging info  
"displayNames": {  
  ".0": {"current": "App", "owner": "<root>"},  
  ".0.0": {"current": "Box", "owner": "App"}  
},  
"totalTime": 0.48499999684281647  
}
```