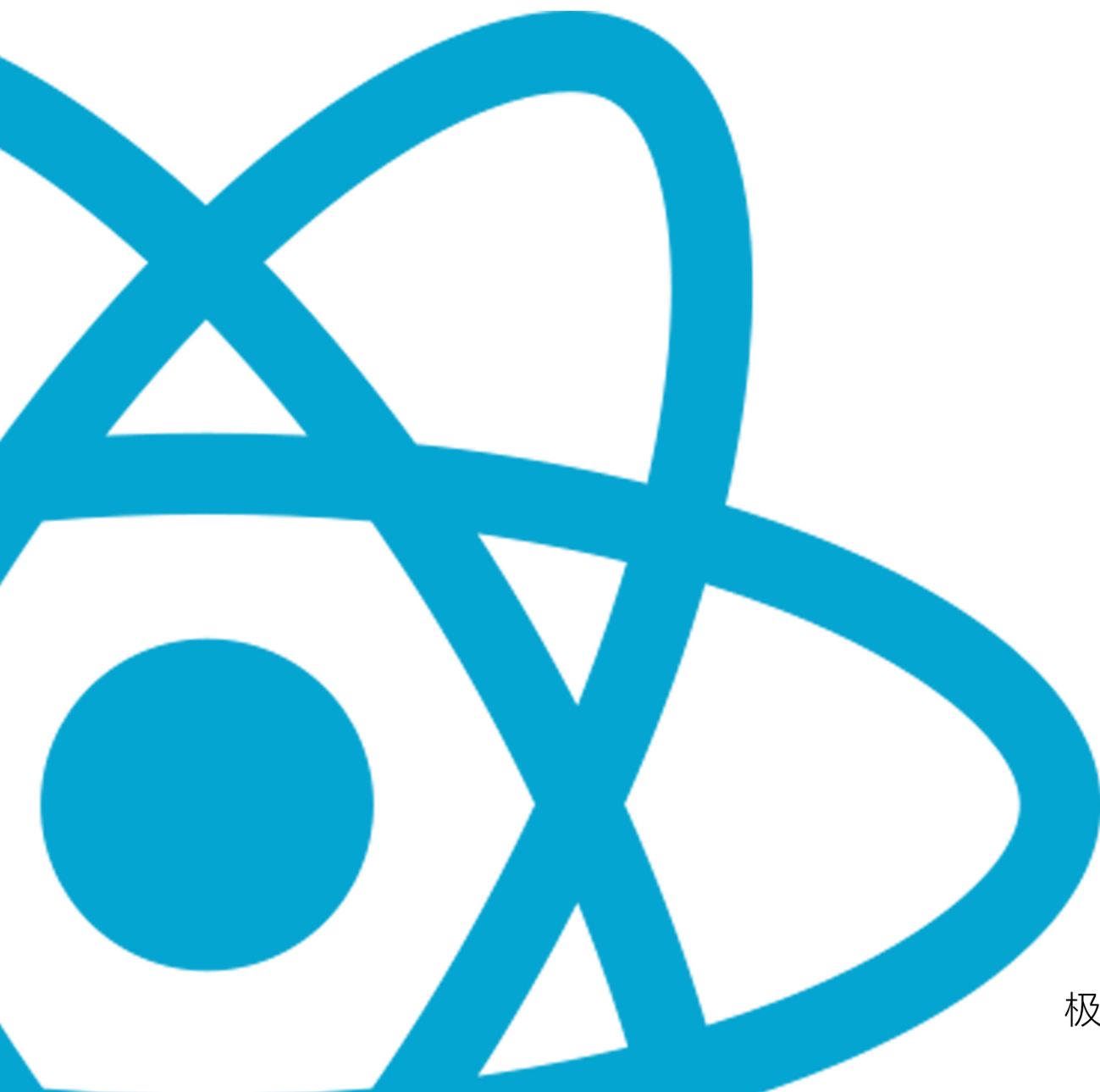


用于构建用户界面的 JavaScript 库

React

--中文版



极客学院出版

前言

React 是 Facebook 推出的一个用来构建用户界面的 JavaScript 库。具备以下特性：

- 不是一个 MVC 框架
- 不使用模板
- 响应式更新非常简单
- HTML5 仅仅是个开始

| 仅仅是 UI

许多人使用 React 作为 MVC 架构的 V 层。尽管 React 并没有假设过你的其余技术栈，但它仍可以作为一个小特征轻易地在已有项目中使用

| 虚拟 DOM

React 为了更高超的性能而使用虚拟 DOM 作为其不同的实现。它同时也可以由服务端 Node.js 渲染，而不需要过重的浏览器 DOM 支持

| 数据流

React 实现了单向响应的数据流，从而减少了重复代码，这也是它为什么比传统数据绑定更简单。

一个简单的组件

React 组件通过一个 `render()` 方法，接受输入的参数并返回展示的对象。

以下这个例子使用了 JSX，它类似于 XML 的语法

输入的参数通过 `render()` 传入组件后，将存储在 `this.props`

JSX 是可选的，并不强制要求使用。

点击 “Compiled JS” 可以看到 JSX 编译之后的 JavaScript 代码

Live JSX Editor

```
var HelloMessage = React.createClass({
  render: function() {
    return <div>Hello {this.props.name}</div>;
  }
});

React.render(<HelloMessage name="John" />, mountNode)
Hello John
```

Compiled JS

```
var HelloMessage = React.createClass({displayName: "HelloMessage",
  render: function() {
    return React.createElement("div", null, "Hello ", this.props.name);
  }
});

React.render(React.createElement(HelloMessage, {name: "John"}), mountNode);
```

一个有状态的组件

除了接受输入数据（通过 `this.props` ），组件还可以保持内部状态数据（通过 `this.state` ）。当一个组件的状态数据的变化，展现的标记将被重新调用 `render()` 更新。

Live JSX Editor

```
var Timer = React.createClass({
  getInitialState: function() {
    return {secondsElapsed: 0};
  },
  tick: function() {
    this.setState({secondsElapsed: this.state.secondsElapsed + 1});
  },
  componentDidMount: function() {
    this.interval = setInterval(this.tick, 1000);
  },
  componentWillUnmount: function() {
    clearInterval(this.interval);
  },
  render: function() {
    return (
      <div>Seconds Elapsed: {this.state.secondsElapsed}</div>
    );
  }
});

React.render(<Timer />, mountNode);
```

Compiled JS

```
var Timer = React.createClass({displayName: "Timer",
  getInitialState: function() {
    return {secondsElapsed: 0};
  },
  tick: function() {
    this.setState({secondsElapsed: this.state.secondsElapsed + 1});
  },
  componentDidMount: function() {
    this.interval = setInterval(this.tick, 1000);
  },
  componentWillUnmount: function() {
    clearInterval(this.interval);
  },
  render: function() {
    return (
      React.createElement("div", null, "Seconds Elapsed: ", this.state.secondsElapsed)
    );
  }
});

React.render(React.createElement(Timer, null), mountNode);
```

一个应用程序

通过使用 `props` 和 `state`，我们可以组合构建一个小型的 `Todo` 程序。

下面例子使用 `state` 去监测当前列表的项以及用户已经输入的文本。 尽管事件绑定似乎是以内联的方式，但他们将被收集起来并以事件代理的方式实现。

```
Live JSX EditorCompiled JS

var TodoList = React.createClass({
  render: function() {
    var createItem = function(itemText) {
      return <li>{itemText}</li>;
    };
    return <ul>{this.props.items.map(createItem)}</ul>;
  }
});

var TodoApp = React.createClass({
  getInitialState: function() {
    return {items: [], text: ''};
  },
  onChange: function(e) {
```

```

    this.setState({text: e.target.value});
  },
  handleSubmit: function(e) {
    e.preventDefault();
    var nextItems = this.state.items.concat([this.state.text]);
    var nextText = '';
    this.setState({items: nextItems, text: nextText});
  },
  render: function() {
    return (
      <div>
        <h3>TODO</h3>
        <TodoList items={this.state.items} />
        <form onSubmit={this.handleSubmit}>
          <input onChange={this.onChange} value={this.state.text} />
          <button>{' Add #' + (this.state.items.length + 1)}</button>
        </form>
      </div>
    );
  }
});

React.render(<TodoApp />, mountNode);

```

Compiled JS

```

var TodoList = React.createClass({displayName: "TodoList",
  render: function() {
    var createItem = function(itemText) {
      return React.createElement("li", null, itemText);
    };
    return React.createElement("ul", null, this.props.items.map(createItem));
  }
});

var TodoApp = React.createClass({displayName: "TodoApp",
  getInitialState: function() {
    return {items: [], text: ''};
  },
  onChange: function(e) {
    this.setState({text: e.target.value});
  },
  handleSubmit: function(e) {
    e.preventDefault();
    var nextItems = this.state.items.concat([this.state.text]);
    var nextText = '';
    this.setState({items: nextItems, text: nextText});
  }
});

```

```

},
render: function() {
  return (
    React.createElement("div", null,
      React.createElement("h3", null, "TODO"),
      React.createElement(TodoList, {items: this.state.items}),
      React.createElement("form", {onSubmit: this.handleSubmit},
        React.createElement("input", {onChange: this.onChange, value: this.state.text}),
        React.createElement("button", null, 'Add #' + (this.state.items.length + 1))
      )
    )
  );
}
});

React.render(React.createElement(TodoApp, null), mountNode);

```

一个使用外部插件的组件

React 是灵活的，并且提供方法允许你跟其他库和框架对接。

下面例子展现了一个案例，使用外部库 Markdown 实时转化 textarea 的值。

Live JSX Editor

```

var converter = new Showdown.converter();

var MarkdownEditor = React.createClass({
  getInitialState: function() {
    return {value: 'Type some *markdown* here!'};
  },
  handleChange: function() {
    this.setState({value: this.refs.textarea.getDOMNode().value});
  },
  render: function() {
    return (
      <div className="MarkdownEditor">
        <h3>Input</h3>
        <textarea
          onChange={this.handleChange}
          ref="textarea"
          defaultValue={this.state.value} />
        <h3>Output</h3>
        <div
          className="content"
          dangerouslySetInnerHTML={{
            __html: converter.makeHtml(this.state.value)
          }}
        />
      </div>
    );
  }
});

```

```

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

React.render(<MarkdownEditor />, mountNode);

```

Compiled JS

```

var converter = new Showdown.converter();

var MarkdownEditor = React.createClass({displayName: "MarkdownEditor",
  getInitialState: function() {
    return {value: 'Type some *markdown* here!'};
  },
  handleChange: function() {
    this.setState({value: this.refs.textarea.getDOMNode().value});
  },
  render: function() {
    return (
      React.createElement("div", {className: "MarkdownEditor"},
        React.createElement("h3", null, "Input"),
        React.createElement("textarea", {
          onChange: this.handleChange,
          ref: "textarea",
          defaultValue: this.state.value}),
        React.createElement("h3", null, "Output"),
        React.createElement("div", {
          className: "content",
          dangerouslySetInnerHTML: {
            __html: converter.makeHtml(this.state.value)
          }
        })
      )
    );
  }
});

React.render(React.createElement(MarkdownEditor, null), mountNode);

```

[快速入门](#)

[下载 React v0.13.0](#)

本教程部分内容来源于 [React 中文网 - reactjs.cn](http://reactjs.cn)

React 官网: <http://facebook.github.io/react/>

更新日期	更新内容
2015-04-10	React 中文版发布

版本信息

书中演示代码基于以下版本:

语言/框架	版本信息
react	0.13.1

目录

前言	1
第 1 章 快速入门 - QUICK START	10
入门	11
教程	14
深入理解 React	30
第 2 章 指南 - GUIDES	36
为什么使用 React?	37
显示数据	38
动态交互式用户界面	41
复合组件	44
可复用组件	49
传递 Props	54
表单组件	58
浏览器中的工作原理	62
工具集成 (ToolingIntegration)	67
插件	69
高级性能	70
第 3 章 参考 - REFERENCE	76
顶层 API	77
组件 API	81
组件的详细说明和生命周期	84
标签和属性支持	89
事件系统	91
与 DOM 的差异	95

	特殊的非 DOM 属性	96
	Reconciliation	97
	React (虚拟) DOM 术语	101
第 4 章	温馨提示 - TIPS	105
	简介	106
	行内样式	107
	JSX 中的 If-Else	108
	自闭合标签	110
	JSX 根节点的最大数量	111
	在样式props中快速制定像素值.	112
	子 props 的类型.	113
	Controlled Input 值为 null 的情况.	114
	Mounting 后 componentWillReceiveProps 未被触发	115
	getInitialState 里的 Props 是一个反模式.	116
	组件的 DOM 事件监听.	118
	通过 AJAX 加载初始数据	119
	JSX 的 false 处理.	120
	组件间的通信	121
	公开组件功能	122
	组件的引用	124
	this.props.children undefined.	125
	与其他类库并行使用 React	126
	Dangerously Set innerHTML.	127



1

快速入门 - QUICK START



入门

JSFiddle

开始 Hack React 的最简单的方法是用下面 JSFiddle 的 Hello Worlds:

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

入门教程包 (Starter Kit)

[开始先下载入门教程包](#)

在入门教程包的根目录，创建一个含有下面代码的 `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 代码文件可以写在另外的文件里。新建下面的 `src/helloworld.js`。

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

然后在 `helloworld.html` 引用该文件：

```
<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
```

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

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

只要你修改了，`build/helloworld.js` 文件会自动生成。

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

对照下面更新你的 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>
```

想用 CommonJS?

如果你想在 [browserify](#), [webpack](#) 或者或其它兼容CommonJS的模块系统里使用 React, 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

去看看入门教程和入门教程包 `examples` 目录下的其它例子学习更多。

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

恭喜你, 欢迎来到 React 的世界。

教程

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。

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- {{site.react_version}}.js"></script>
    <script src="http://fb.me/JSXTransformer- {{site.react_version}}.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 `<div>` 而已：

```

// tutorial1.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：

```
// tutorial1-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）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 `<div>`：

```
// 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</Comment>
      </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-{{site.react_version}}.js"></script>
  <script src="http://fb.me/JSXTransformer-{{site.react_version}}.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/showdown.min.js"></script>
</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：“`<p>` This is `` another `` comment `</p>`”。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) {
        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` 属性给予组件命名，`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 onSubmit={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.onSubmit({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 onSubmit={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 \(页 0\)](#)，开始专研！祝你好运！

深入理解 React

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。

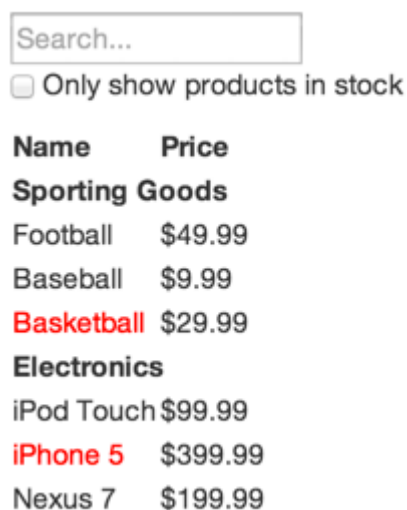
本文中，我将跟你一起使用 React 构建一个具备搜索功能的产品列表。

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

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：



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

图片 1.1 Mockup

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: true, name: "iPhone 5"},
  {category: "Electronics", price: "$199.99", stocked: true, name: "Nexus 7"}
]
```

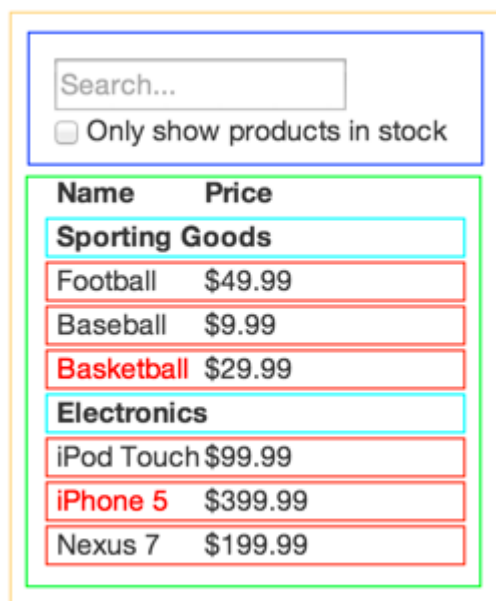
```
{category: "Electronics", price: "$399.99", stocked: false, name: "iPhone 5"},
{category: "Electronics", price: "$199.99", stocked: true, name: "Nexus 7"}
];
```

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。



图片 1.2 Component diagram

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 :)。



2

指南 - GUIDES



为什么使用 React?

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。

显示数据

用户界面能做的最基础的事就是显示一些数据。React 让显示数据变得简单，当数据变化的时候，用户界面会自动同步更新。

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，代码如下：

```
<!DOCTYPE html>
<html>
  <head>
    <title>Hello React</title>
    <script src="http://fb.me/react-{{site.react_version}}.js"></script>
    <script src="http://fb.me/JSXTransformer-{{site.react_version}}.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);
```

被动更新 (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，任何你在输入框输入的内容一直保留着，即使你没有写任何代码来完成这个功能。React 为你解决了这个问题，做了正确的事。

我们想到的方法是除非不得不操作 DOM，React 是不会去操作 DOM 的。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。

对这个组件的输入称为 `props` - “properties”的缩写。得益于 JSX 语法，它们通过参数传递。你必须知道在组件里，这些属性是不可改变的，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。正是应为它们是这么的简单，这使得它们非常容易理解。

注意：

只有一个限制：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

我们坚信组件是正确方法去做关注分离，而不是通过“模板”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

```
<a href="http://facebook.github.io/react/">Hello React!</a>。
```

我们发现这会使搭建 React 应用更加简单，设计师也偏向用这语法，但是每个人可以有它们自己的工作流，所以 JSX 不是必须用的。

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，每三个是子节点。

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

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.getElementById('example'));
```

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')
);
```

动态交互式用户界面

我们已经学习如何使用 React 显示数据。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。

幕后原理：自动绑定（Autobinding）和事件代理（Event Delegation）

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，也能减少冗余并，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 来作为惟一数据来源。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。

复合组件

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

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 + '/picture'} />
    );
  }
});

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

```

    </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 `<p>Paragraph 1</p>`。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，你可以通过给予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。实际情况下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。

可复用组件

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，而不必写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，并用有多个 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。

传递 Props

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' : 'FancyUnchecked';
    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.getElementById('example')
);
```

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.getElementById('example')
);
```

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' : 'FancyUnchecked';
    // 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 : '0 ' + 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。

```
var FancyCheckbox = React.createClass({
  render: function() {
    var checked = this.props.checked;
    var other = _.omit(this.props, 'checked');
    var fancyClass = checked ? 'FancyChecked' : 'FancyUnchecked';
    return (
      React.DOM.div(_.extend({}, other, { className: fancyClass }))
    );
  }
});
```

表单组件

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。

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.handleChange} />;
}
```

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 1
      <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>
  );
}
```

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>
```

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']}>`。

浏览器中的工作原理

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，因为它性能更加好，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，我们支持老版本的浏览器，包括 IE8。我们已经拥有适当地 polyfills 很长一段时间了，能够允许我们写有远见的 JS。这意味着在我们的代码库中没有黑客，我们仍然可以期待我们的代码“工作”。例如，不用写 `+new Date()`，我们可以只写 `Date.now()`。由于开源的 React 和我们内部使用的是一样的，所以我们采取了使用超前思维 JS 的这种理念。

除了那种理念，我们也确立了我们的立场，作为一个 JS 库的作者，不应该把 polyfills 作为库的一部分 shipping。如果每个库都这么做，很有可能你会多次发送相同的 polyfill，这可能相当于一部分无用代码。如果你的产品需要支持老版本的浏览器，很有可能你已经在使用类似 [es5-shim](#) 的东西了。

为了支持老版本浏览器所需的 polyfill

来自 [kriskowal's es5-shim](#) 的 `es5-shim.js` 提供了 React 所需的如下方法：

- `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-shim.js`，同样来自 [kriskowal's es5-shim](#)，提供了 React 所需的如下方法：

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

React 的 unminified 构建需要来自 [paulmillr's console-polyfill](#) 的如下方法：

- `console.*`

当使用 IE8 中的 HTML5 元素时，包括 `<section>`，`<article>`，`<nav>`，`<header>` 和 `<footer>`，包含 [html5shiv](#) 或类似的一个脚本也是必须的。

跨浏览器问题

虽然 React 非常擅长抽象浏览器的差异，但是一些浏览器是有限制的或呈现出怪异的行为，对于这种行为我们不能找到解决方案。

IE8 的 onScroll 事件

在 IE8 中，`onScroll` 事件并不 bubble，并且 IE8 没有一个 API 定义处理程序来捕获一个事件的阶段，这意味着 React 没有办法监听这些事件。目前在 IE8 中，事件处理程序被忽略了。

更多信息请看 [onScroll 在 IE8 中不起作用的 GitHub 问题](#)。

工具集成 (ToolingIntegration)

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 type="text/jsx">` 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。

插件

`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。
- `LinkStateMixin`，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。

高级性能

人们首先会考虑的是 React 是能否和其他非 React 版本一样能快速和响应一个项目。重新绘制组件的整个子树来回应每一个状态变化的想法让人怀疑是否这个过程中对性能产生负面影响。React 使用几个巧妙的技巧，以减少所需的更新用户界面所需要的昂贵的文档用户模型操作的数目。

避免调和文档对象模型

React 使用了一个*虚拟的 DOM*，这是的浏览器中对于 DOM 树呈现的一个描述符。这种并行表示形式让 React 避免产生 DOM 节点和访问现有的节点，比 JavaScript 对象的操作速度较慢。当一个组件的道具或状态改变，React 决定通过构建一个新的虚拟 DOM 来进行实际的 DOM 更新和旧的 DOM 相比是否必要。只有在比较结果不一样的情况下，React 尽可能少的应用转变来融合文档对象模型。

在此之上，React 提供了一个组件的生命周期功能，`shouldComponentUpdate`，这是在重新绘制过程开始之前触发（虚拟 DOM 比较，可能最终调和 DOM），使开发人员能够减少过程中的循环步骤。这个函数的默认实现返回 `true`，让 React 来执行更新：

```
shouldComponentUpdate: function(nextProps, nextState) {  
  return true;  
}
```

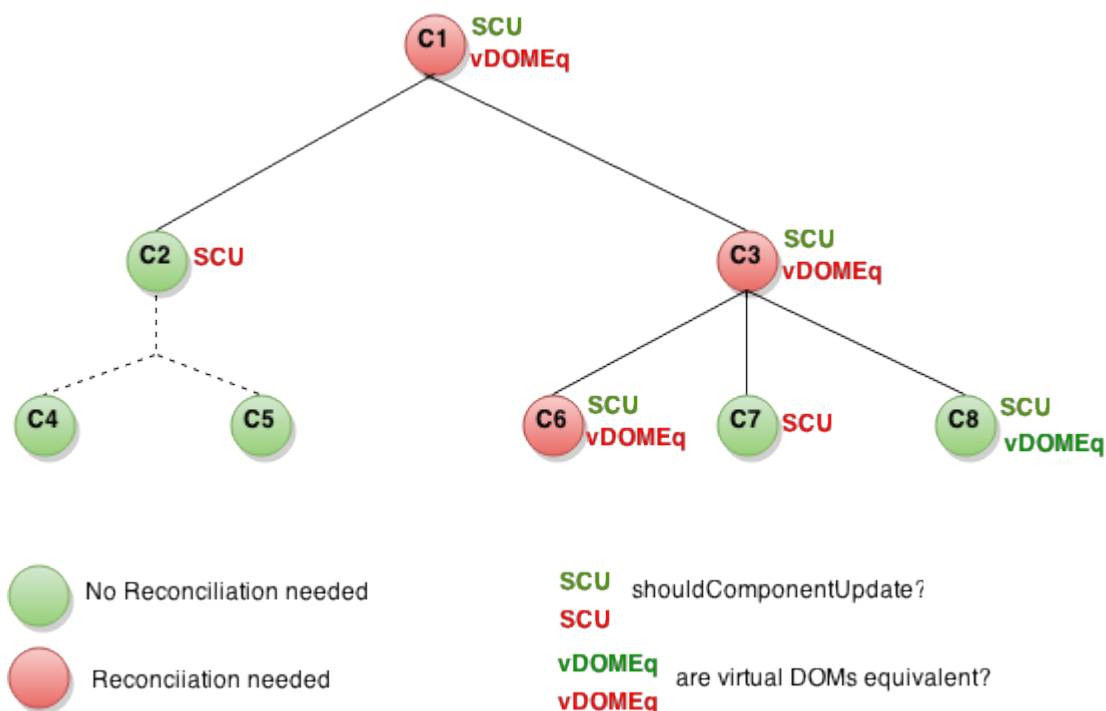
请记住，React 将非常频繁的调用这个函数，所以实现必须要快。比如说你有一个由几个聊天线程组成的消息应用程序。假设只有一个线程已经改变。如果我们在 `ChatThread` 上执行 `shouldComponentUpdate`，React 可以为其他线程跳过描绘步骤：

```
shouldComponentUpdate: function(nextProps, nextState) {  
  // TODO: return whether or not current chat thread is  
  // different to former one.  
}
```

因此，简言之，React 避免调和 DOM 产生的复杂的 DOM 操作，允许用户使用 `shouldComponentUpdate` 缩短过程中的循环步骤，而且，对于那些需要更新，通过对比虚拟的 DOM 来实现。

起作用的 `shouldComponentUpdate`

下面是组件的子树。对于每一个组件表示 `shouldComponentUpdate` 的返回值，以及是否与虚拟的 DOM 是等价的。最后，圆的颜色指示组件是否必须调和。



图片 2.1 should-component-update

在上面的例子中，由于 `shouldComponentUpdate` 返回值为 `false`，存在 C2 中，React 没有必要产生新的虚拟的 DOM，并且因此，也不需要调和 DOM。需要注意的是 react 甚至没有在 C4 和 C5 处调用 `shouldComponentUpdate`。

对于 C1 和 C3 `shouldComponentUpdate` 返回 `true`，所以 React 不得不深入到叶子节点并且进行检查。

对于 C6 它返回 `true`；由于和虚拟的 DOM 并不等同，它不得不调和 DOM。最后一个有趣的例子是 C8。此节点的 React 必须计算虚拟 DOM，但因为它和原来的 DOM 相同，它不需要调和 DOM。

请注意，只有 C6 需要 React 不得不对 DOM 做转变，这是不可避免的。对于 C8 通过比较虚拟的 DOM 他不需要转变，但是对 C2 的子树和 C7来说，它甚至没有计算虚拟 DOM，只需要通过执行 `shouldComponentUpdate`。

所以，我们应该如何执行 `shouldComponentUpdate` 呢？比如现有一个仅需呈现一个字符串值的组件：

```
React.createClass({
  propTypes: {
    value: React.PropTypes.string.isRequired
  },
  render: function() {
    return <div>this.props.value</div>;
  }
});
```

我们可以很容易地实现 `shouldComponentUpdate`，如下：


```
shouldComponentUpdate: function(nextProps, nextState) {
  return this.props.value !== nextProps.value;
}
```

到目前为止，React 处理这类简单的道具/态结构非常简单。我们甚至可以泛化一个基于浅层相等实现，并混合到组件上。事实上，React 已经提供了这样实现：[PureRenderMixin](#)。

但如果你的组件的道具或状态是可变的数据结构呢？比如说组件接受的道具，而不是像 `'bar'` 的这样的字符串，而是一个是包含，如，`{foo: 'bar'}` 这样一个字符串的 JavaScript 对象：

```
React.createClass({
  propTypes: {
    value: React.PropTypes.object.isRequired
  },
  render: function() {
    return <div>this.props.value.foo</div>;
  }
});
```

我们之前的 `shouldComponentUpdate` 实现总是不会如我们预期一样的实现：

```
/ assume this.props.value is { foo: 'bar' }
// assume nextProps.value is { foo: 'bar' },
// but this reference is different to this.props.value
this.props.value !== nextProps.value; // true
```

问题是当道具实际上并没有改变时，`shouldComponentUpdate` 将返回 `true`。为了解决这个问题，我们可以用这个替代的试行方案：

```
shouldComponentUpdate: function(nextProps, nextState) {
  return this.props.value.foo !== nextProps.value.foo;
}
```

基本上，我们为确保我们正确地跟踪变化，最后做了深刻的对比。这种做法是在性能方面相当昂贵和复杂的，它并不能扩展，因为我们要为每个模型写不同的深度相等代码。最重要的是，如果我们不小心管理对象的引用，它甚至可能没有工作。比如说母节点组件的引用：

```
React.createClass({
  getInitialState: function() {
    return { value: { foo: 'bar' } };
  },
  onClick: function() {
    var value = this.state.value;
    value.foo += 'bar'; // ANTI-PATTERN!
    this.setState({ value: value });
  }
});
```

```

    },
    render: function() {
      return (
        <div>
          <InnerComponent value={this.state.value} />
          <a onClick={this.onClick}>Click me</a>
        </div>
      );
    }
  });

```

在第一时间内部组件得到呈现是 `{F00: 'bar'}`，它将作为道具的值。如果用户点击，母组件的状态将得到更新为 `{value: {F00: 'barbar'}}`，引发了内部部件在过程中重新呈现，将接收 `{foo: "barbar"}` 为道具的新值。

问题是，由于母体和内部部件共享一个参考同一个对象，当对象在第二行的 `onClick` 功能函数中突变时，道具的内部部件具有将发生变化。这样，当再描绘处理开始时，`shouldComponentUpdate` 被调用，`this.props.value.foo` 将等于 `nextProps.value.foo`，因为事实上，`this.props.value` 和 `nextProps.value` 引用相同的对象。

因此，由于我们错过了道具和缩短步骤重新渲染过程中的变化，用户界面将不会得到从 `'bar'` 到 `'barbar'` 的更新。

Immutable-JS 救援

[Immutable-JS](#) 是 Lee Byron 写的脚本语言集合库，其中 Facebook 最近开源 Javascript 的集合库。它提供了通过结构性共享一成不变持久化集合。让我们看看这些性能：

- **不可变的：**一旦创建，集合不能在另一个时间点改变。
- **持久性：**新的集合可以由从早先的集合和突变结合创建。在创建新的集合后，原来集合仍然有效。
- **结构共享：**使用新的集合创建为与对原始集合大致相同的结构，减少了拷贝的最低限度，以实现空间效率和可接受的性能。如果新的集合等于原始的集合，则通常会返回原来的集合。

不变性使得跟踪更改方便；而变化将总是产生在新的对象，所以我们只需要检查的已经改变参考对象。例如，在这个 Javascript 代码中：

```

var x = { foo: "bar" };
var y = x;
y.foo = "baz";
x === y; // true

```

虽然 `y` 被修改了，但因为它是对相同对象 `x` 的引用，所以这个比较返回 `true`。然而，这段代码可以用 `immutableJS` 这样写：

```
var SomeRecord = Immutable.Record({ foo: null });
var x = new SomeRecord({ foo: 'bar' });
var y = x.set('foo', 'baz');
x === y; // false
```

在这种情况下，由于 `x` 突变，当一个新的引用被返回，我们可以安全地假设 `x` 已经改变。

另一种跟踪变化的可能的方法是通过设置标志来做 `dirty` 检查。这种方法的一个问题是，它迫使你使用 `setter` 或者写很多额外的代码，或某种类工具。或者，你可以突变之前深入对象复制并深入比较，以确定是否有变化。这种方法的一个问题是 `deepCopy` 和 `deepCompare` 是耗费大且复杂的操作。

因此，不可变的数据结构为您提供了一种廉价和更简洁的方式来跟踪对象的变化，这就是我们为了实现 `shouldComponentUpdate` 所需要的。因此，如果我们利用 `immutableJS` 提供的抽象特性来支持和声明属性，我们就可以使用 `PureRenderMixin` 并获得 `perf` 的一个很好的推动。

ImmutableJS 和通量

如果你使用通量，你应该开始使用 `immutableJS` 写你的库。看看[完整的 API](#)。

让我们来看看使用不可变的数据结构来模拟线程的一种可行的办法。首先，我们需要为每一个我们正在尝试模拟的实体定义一个 `record`。记录是不可变的容器，为一组特定的域保存值：

```
var User = Immutable.Record({
  id: undefined,
  name: undefined,
  email: undefined
});
var Message = Immutable.Record({
  timestamp: new Date(),
  sender: undefined,
  text: ''
});
```

`Record` 函数接收的对象定义了对象的域和它们的默认值。

消息库可以使用以下两个列表来跟踪用户和信息：

```
this.users = Immutable.List();
this.messages = Immutable.List();
```

实现处理每个负载类型的功能应该是相当容易的。例如，当库看到负载正在显示一个新的消息，我们只要创建一个新的记录，并把它添加到消息列表中：

```
this.messages = this.messages.push(new Message({
  timestamp: payload.timestamp,
  sender: payload.sender,
  text: payload.text
}));
```

请注意，由于数据结构是不可改变的，我们需要把推送功能的结果分配到 `this.messages` 中。

在 React 中，如果我们也使用 `immutable-JS` 数据结构来保存组件的状态下，我们可以混合 `PureRenderMixin` 到所有的组件，并且缩短重新呈现的过程。



3

参考 - REFERENCE



顶层 API

React

`React` 是 `React` 库的入口。如果使用的是预编译包，则 `React` 是全局的；如果使用 `CommonJS` 模块系统，则可以用 `require()` 函数引入 `React`。

`React.createClass`

```
ReactClass.createClass(object specification)
```

创建一个组件类，并作出定义。组件实现了 `render()` 方法，该方法返回一个子级。该子级可能包含很深的子级结构。组件与标准原型类的不同之处在于，你不需要使用 `new` 来实例化。组件是一种很方便的封装，可以（通过 `new`）为你创建后台实例。

更多关于定义组件对象的信息，参考[组件定义和生命周期](#)。

`React.createElement`

```
ReactDOM.createElement(  
  string/ReactClass type,  
  [object props],  
  [children ...]  
)
```

创建并返回一个新的指定类型的 `ReactDOMElement`。`type` 参数可以是一个 `html` 标签名字字符串（例如，“`div`”，“`span`”，等等），或者是 `ReactClass`（通过 `React.createClass` 创建的）。

`React.createFactory`

```
factoryFunction createFactory(  
  string/ReactClass type  
)
```

返回一个生成指定类型 `ReactDOMElements` 的函数。比如 `React.createElement`，`type` 参数可以是一个 `html` 标签名字字符串（例如，“`div`”，“`span`”，等等），或者是 `ReactClass`。

React.render

```
ReactComponent render(
  ReactElement element,
  DOMElement container,
  [function callback]
)
```

渲染一个 `ReactElement` 到 DOM 中，放在 `container` 指定的 DOM 元素下，返回一个到该组件的引用。

如果 `ReactElement` 之前就被渲染到了 `container` 中，该函数将会更新此 `ReactElement`，仅改变需要改变的 DOM 节点以展示最新的 React 组件。

如果提供了可选的回调函数，则该函数将会在组件渲染或者更新之后调用。

注意：

`React.render()` 替换传入的容器节点内容。在将来，或许可能插入组件到已存在的 DOM 节点中，但不覆盖已有的子节点。

React.unmountComponentAtNode

```
boolean unmountComponentAtNode(DOMElement container)
```

从 DOM 中移除已经挂载的 React 组件，清除相应的事件处理器和 state。如果在 `container` 内没有组件挂载，这个函数将什么都不做。如果组件成功移除，则返回 `true`；如果没有组件被移除，则返回 `false`。

React.renderToString

```
string renderToString(ReactElement element)
```

把组件渲染成原始的 HTML 字符串。该方法应该仅在服务器端使用。React 将会返回一个 HTML 字符串。你可以在服务器端用此方法生成 HTML，然后将这些标记发送给客户端，这样可以获得更快的页面加载速度，并且有利于搜索引擎抓取页面，方便做 SEO。

如果在一个节点上面调用 `React.render()`，并且该节点已经有了服务器渲染的标记，React 将会维护该节点，并且仅绑定事件处理器，保证有一个高效的首屏加载体验。

React.renderToStaticMarkup

```
string renderToStaticMarkup(ReactElement element)
```

和 `renderToString` 类似，除了不创建额外的 DOM 属性，例如 `data-react-id`，因为这些属性仅在 React 内部使用。如果你想用 React 做一个简单的静态页面生成器，这是很有用的，因为丢掉额外的属性能够节省很多字节。

React.isValidElement

```
boolean isValidElement(* object)
```

判断对象是否是一个 `ReactElement`。

React.DOM

`React.DOM` 运用 `React.createElement` 为 DOM 组件提供了方便的包装。该方式仅在未使用 JSX 的时候适用。例如，`React.DOM.div(null, 'Hello World!')`。

React.PropTypes

`React.PropTypes` 包含了能与组件 `propTypes` 对象共用的类型，用于验证传入组件的 props。更多有关 `propTypes` 的信息，参考[复用组件](#)。

React.initializeTouchEvents

```
initializeTouchEvents(boolean shouldUseTouch)
```

配置 React 的事件系统，使 React 能处理移动设备的触摸（touch）事件。

React.Children

`React.Children` 为处理 `this.props.children` 这个封闭的数据结构提供了有用的工具。

React.Children.map

```
object React.Children.map(object children, function fn [, object context])
```


在每一个直接子级（包含在 `children` 参数中的）上调用 `fn` 函数，此函数中的 `this` 指向 `上下文`。如果 `children` 是一个内嵌的对象或者数组，它将被遍历：不会传入容器对象到 `fn` 中。如果 `children` 参数是 `null` 或者 `undefined`，那么返回 `null` 或者 `undefined` 而不是一个空对象。

`React.Children.forEach`

```
React.Children.forEach(object children, function fn [, object context])
```

类似于 `React.Children.map()`，但是不返回对象。

`React.Children.count`

```
number React.Children.count(object children)
```

返回 `children` 当中的组件总数，和传递给 `map` 或者 `forEach` 的回调函数的调用次数一致。

`React.Children.only`

```
object React.Children.only(object children)
```

返回 `children` 中仅有的子级。否则抛出异常。

组件 API

ReactComponent

React 组件实例在渲染的时候创建。这些实例在接下来的渲染中被重复使用，可以在组件方法中通过 `this` 访问。唯一一种在 React 之外获取 React 组件实例句柄的方式就是保存 `React.render` 的返回值。在其它组件内，可以使用 `refs` 得到相同的结果。

setState

```
setState(object nextState[, function callback])
```

合并 `nextState` 和当前 `state`。这是在事件处理函数中和请求回调函数中触发 UI 更新的主要方法。另外，也支持可选的回调函数，该函数在 `setState` 执行完毕并且组件重新渲染完成之后调用。

注意：

绝对不要直接改变 `this.state`，因为在之后调用 `setState()` 可能会替换掉你做的更改。把 `this.state` 当做不可变的。

`setState()` 不会立刻改变 `this.state`，而是创建一个即将处理的 `state` 转变。在调用该方法之后获取 `this.state` 的值可能会得到现有的值，而不是最新设置的值。

不保证 `setState()` 调用的同步性，为了提升性能，可能会批量执行 `state` 转变和 DOM 渲染。

`setState()` 将总是触发一次重绘，除非在 `shouldComponentUpdate()` 中实现了条件渲染逻辑。如果使用可变的对象，但是又不能在 `shouldComponentUpdate()` 中实现这种逻辑，仅在新 `state` 和之前的 `state` 存在差异的时候调用 `setState()` 可以避免不必要的重新渲染。

replaceState

```
replaceState(object nextState[, function callback])
```

类似于 `setState()`，但是删除之前所有已存在的 `state` 键，这些键都不在 `nextState` 中。

forceUpdate()

```
forceUpdate([function callback])
```

如果 `render()` 方法从 `this.props` 或者 `this.state` 之外的地方读取数据，你需要通过调用 `forceUpdate()` 告诉 React 什么时候需要再次运行 `render()`。如果直接改变了 `this.state`，也需要调用 `forceUpdate()`。

调用 `forceUpdate()` 将会导致 `render()` 方法在相应的组件上被调用，并且子级组件也会调用自己的 `render()`，但是如果标记改变了，那么 React 仅会更新 DOM。

通常情况下，应该尽量避免所有使用 `forceUpdate()` 的情况，在 `render()` 中仅从 `this.props` 和 `this.state` 中读取数据。这会使应用大大简化，并且更加高效。

getDOMNode

```
DOMNode getDOMNode()
```

如果组件已经挂载到了 DOM 上，该方法返回相应的本地浏览器 DOM 元素。从 DOM 中读取值的时候，该方法很有用，比如获取表单字段的值和做一些 DOM 操作。当 `render` 返回 `null` 或者 `false` 的时候，`this.getDOMNode()` 返回 `null`。

isMounted()

```
bool isMounted()
```

如果组件渲染到了 DOM 中，`isMounted()` 返回 `true`。可以使用该方法保证 `setState()` 和 `forceUpdate()` 在异步场景下的调用不会出错。

setProps

```
setProps(object nextProps[, function callback])
```

当和一个外部的 JavaScript 应用集成的时候，你可能想给一个用 `React.render()` 渲染的组件打上改变的标记。

尽管在同一个节点上再次调用 `React.render()` 来更新根组件是首选的方式，也可以调用 `setProps()` 来改变组件的属性，触发一次重新渲染。另外，可以传递一个可选的回调函数，该函数将会在 `setProps` 完成并且组件重新渲染完成之后调用。

注意：

When possible, the declarative approach of calling `React.render()` again is preferred; it tends to make updates easier to reason about. (There's no significant performance difference between the two approaches.)

该方法仅在根组件上面调用。也就是说，仅在直接传给 `React.render()` 的组件上可用，在它的子级组件上不可用。如果你倾向于在子组件上使用 `setProps()`，不要利用响应式更新，而是当子组件在 `render()` 中创建的时候传入新的 prop 到子组件中。

replaceProps

```
replaceProps(object nextProps[, function callback])
```

类似于 `setProps()`，但是删除所有已存在的 props，而不是合并新旧两个 props 对象。

组件的详细说明和生命周期

组件的详细说明 (Component Specifications)

当通过调用 `React.createClass()` 来创建组件的时候，你应该提供一个包含 `render` 方法的对象，并且也可以包含其它的在这里描述的生命周期方法。

render

```
ReactComponent render()
```

`render()` 方法是必须的。

当调用的时候，会检测 `this.props` 和 `this.state`，返回一个单子级组件。该子级组件可以是虚拟的本地 DOM 组件（比如 `<div />` 或者 `React.DOM.div()`），也可以是自定义的复合组件。

你也可以返回 `null` 或者 `false` 来表明不需要渲染任何东西。实际上，React 渲染一个 `<noscript>` 标签来处理当前的差异检查逻辑。当返回 `null` 或者 `false` 的时候，`this.getDOMNode()` 将返回 `null`。

`render()` 函数应该是纯粹的，也就是说该函数不修改组件 `state`，每次调用都返回相同的结果，不读写 DOM 信息，也不和浏览器交互（例如通过使用 `setTimeout`）。如果需要和浏览器交互，在 `componentDidMount()` 中或者其它生命周期方法中做这件事。保持 `render()` 纯粹，可以使服务器端渲染更加切实可行，也使组件更容易被理解。

getInitialState

```
object getInitialState()
```

在组件挂载之前调用一次。返回值将会作为 `this.state` 的初始值。

getDefaultProps

```
object getDefaultProps()
```

在组件类创建的时候调用一次，然后返回值被缓存下来。如果父组件没有指定 `props` 中的某个键，则此处返回的对象中的相应属性将会合并到 `this.props`（使用 `in` 检测属性）。

该方法在任何实例创建之前调用，因此不能依赖于 `this.props`。另外，`getDefaultProps()` 返回的任何复杂对象将会在实例间共享，而不是每个实例拥有一份拷贝。

propTypes

```
object propTypes
```

`propTypes` 对象允许验证传入到组件的 `props`。更多关于 `propTypes` 的信息，参考[可重用的组件](#)。

mixins

```
array mixins
```

`mixin` 数组允许使用混合来在多个组件之间共享行为。更多关于混合的信息，参考[可重用的组件](#)。

statics

```
object statics
```

`statics` 对象允许你定义静态的方法，这些静态的方法可以在组件类上调用。例如：

```
var MyComponent = React.createClass({
  statics: {
    customMethod: function(foo) {
      return foo === 'bar';
    }
  },
  render: function() {
  }
});

MyComponent.customMethod('bar'); // true
```

在这个块儿里面定义的方法都是静态的，意味着你可以在任何组件实例创建之前调用它们，这些方法不能获取组件的 `props` 和 `state`。如果你想在静态方法中检查 `props` 的值，在调用处把 `props` 作为参数传入到静态方法。

displayName

```
string displayName
```

`displayName` 字符串用于输出调试信息。JSX 自动设置该值；参考[JSX 深入](#)。

生命周期方法

许多方法在组件生命周期中某个确定的时间点执行。

挂载: `componentWillMount`

```
componentWillMount()
```

服务器端和客户端都只调用一次，在初始化渲染执行之前立刻调用。如果在这个方法内调用 `setState`，`render()` 将会感知到更新后的 `state`，将会执行仅一次，尽管 `state` 改变了。

挂载: `componentDidMount`

```
componentDidMount()
```

在初始化渲染执行之后立刻调用一次，仅客户端有效（服务器端不会调用）。在生命周期中的这个时间点，组件拥有一个 DOM 展现，你可以通过 `this.getNode()` 来获取相应 DOM 节点。

如果想和其它 JavaScript 框架集成，使用 `setTimeout` 或者 `setInterval` 来设置定时器，或者发送 AJAX 请求，可以在该方法中执行这些操作。

注意：

为了兼容 v0.9，DOM 节点作为最后一个参数传入。你依然可以通过 `this.getNode()` 获取 DOM 节点。

更新: `componentWillReceiveProps`

```
componentWillReceiveProps(object nextProps)
```

在组件接收到新的 `props` 的时候调用。在初始化渲染的时候，该方法不会调用。

用此函数可以作为 `react` 在 `prop` 传入之后，`render()` 渲染之前更新 `state` 的机会。老的 `props` 可以通过 `this.props` 获取到。在该函数中调用 `this.setState()` 将不会引起第二次渲染。

```
componentWillReceiveProps: function(nextProps) {  
  this.setState({  
    likesIncreasing: nextProps.likeCount > this.props.likeCount  
  });  
}
```

注意：

对于 `state`，没有相似的方法：`componentWillReceiveState`。将要传进来的 `prop` 可能会引起 `state` 改变，反之则不然。如果需要在 `state` 改变的时候执行一些操作，请使用 `componentWillUpdate`。

更新： `shouldComponentUpdate`

```
boolean shouldComponentUpdate(object nextProps, object nextState)
```

在接收到新的 `props` 或者 `state`，将要渲染之前调用。该方法在初始化渲染的时候不会调用，在使用 `forceUpdate` 方法的时候也不会。

如果确定新的 `props` 和 `state` 不会导致组件更新，则此处应该返回 `false`。

```
shouldComponentUpdate: function(nextProps, nextState) {
  return nextProps.id !== this.props.id;
}
```

如果 `shouldComponentUpdate` 返回 `false`，则 `render()` 将不会执行，直到下一次 `state` 改变。（另外，`componentWillUpdate` 和 `componentDidUpdate` 也不会被调用。）

默认情况下，`shouldComponentUpdate` 总会返回 `true`，在 `state` 改变的时候避免细微的 `bug`，但是如果总是小心地把 `state` 当做不可变的，在 `render()` 中只从 `props` 和 `state` 读取值，此时你可以覆盖 `shouldComponentUpdate` 方法，实现新老 `props` 和 `state` 的比对逻辑。

如果性能是个瓶颈，尤其是有几十个甚至上百个组件的时候，使用 `shouldComponentUpdate` 可以提升应用的性能。

更新： `componentWillUpdate`

```
componentWillUpdate(object nextProps, object nextState)
```

在接收到新的 `props` 或者 `state` 之前立刻调用。在初始化渲染的时候该方法不会被调用。

使用该方法做一些更新之前的准备工作。

注意：

你不能在刚方法中使用 `this.setState()`。如果需要更新 `state` 来响应某个 `prop` 的改变，请使用 `componentWillReceiveProps`。

更新: `componentDidUpdate`

```
componentDidUpdate(object prevProps, object prevState)
```

在组件的更新已经同步到 DOM 中之后立刻被调用。该方法不会在初始化渲染的时候调用。

使用该方法可以在组件更新之后操作 DOM 元素。

注意:

为了兼容 v0.9, DOM 节点会作为最后一个参数传入。如果使用这个方法, 你仍然可以使用 `this.getDOMNode()` 来访问 DOM 节点。

移除: `componentWillUnmount`

```
componentWillUnmount()
```

在组件从 DOM 中移除的时候立刻被调用。

在该方法中执行任何必要的清理, 比如无效的定时器, 或者清除在 `componentDidMount` 中创建的 DOM 元素。

标签和属性支持

支持的标签

React 尝试支持所用常用的元素。如果你需要的元素没有在下面列出来，请提交一个问题（issue）。

HTML 元素

下列的 HTML 元素是被支持的：

```
a abbr address area article aside audio b base bdi bdo big blockquote body br
button canvas caption cite code col colgroup data datalist dd del details dfn
dialog div dl dt em embed fieldset figcaption figure footer form h1 h2 h3 h4 h5
h6 head header hr html i iframe img input ins kbd keygen label legend li link
main map mark menu menuitem meta meter nav noscript object ol optgroup option
output p param picture pre progress q rp rt ruby s samp script section select
small source span strong style sub summary sup table tbody td textarea tfoot th
thead time title tr track u ul var video wbr
```

SVG 元素

下列的 SVG 元素是被支持的：

```
circle defs ellipse g line linearGradient mask path pattern polygon polyline
radialGradient rect stop svg text tspan
```

你或许对 [react-art](#) 也感兴趣，它是一个为 React 写的渲染到 Canvas、SVG 或者 VML（IE8）的绘图库。

支持的属性

React 支持所有 `data-*` 和 `aria-*` 属性，也支持下面列出的属性。

注意：

所有的属性都是驼峰命名的，`class` 属性和 `for` 属性分别改为 `className` 和 `htmlFor`，来符合 DOM API 规范。

对于支持的事件列表，参考[支持的事件](#)。

HTML 属性

这些标准的属性是被支持的：

```
accept acceptCharset accessKey action allowFullScreen allowTransparency alt
async autoComplete autoPlay cellPadding cellSpacing charSet checked classID
className cols colSpan content contentEditable contextMenu controls coords
crossOrigin data dateTime defer dir disabled download draggable encType form
formAction formEncType formMethod formNoValidate formTarget frameBorder height
hidden href hrefLang htmlFor httpEquiv icon id label lang list loop manifest
marginHeight marginWidth max maxLength media mediaGroup method min multiple
muted name noValidate open pattern placeholder poster preload radioGroup
readOnly rel required role rows rowSpan sandbox scope scrolling seamless
selected shape size sizes span spellCheck src srcDoc srcSet start step style
tabIndex target title type useMap value width wmode
```

另外，下面非标准的属性也是被支持的：

- `autoCapitalize` `autoCorrect` 用于移动端的 Safari。
- `property` 用于 [Open Graph](#) 原标签。
- `itemProp` `itemScope` `itemType` 用于 [HTML5 microdata](#)。

也有 React 特有的属性 `dangerouslySetInnerHTML` ([更多信息](#)，用于直接插入 HTML 字符串到组件中)。

SVG 属性

```
cx cy d dx dy fill fillOpacity fontFamily fontSize fx fy gradientTransform
gradientUnits markerEnd markerMid markerStart offset opacity
patternContentUnits patternUnits points preserveAspectRatio r rx ry
spreadMethod stopColor stopOpacity stroke strokeDasharray strokeLinecap
strokeOpacity strokeWidth textAnchor transform version viewBox x1 x2 x y1 y2 y
```

事件系统

合成事件

事件处理程序通过 `合成事件`（`SyntheticEvent`）的实例传递，`SyntheticEvent` 是浏览器原生事件跨浏览器的封装。`SyntheticEvent` 和浏览器原生事件一样有 `stopPropagation()`、`preventDefault()` 接口，而且这些接口跨浏览器兼容。

如果出于某些原因想使用浏览器原生事件，可以使用 `nativeEvent` 属性获取。每个合成事件（`SyntheticEvent`）对象都有以下属性：

```
boolean bubbles
boolean cancelable
DOMEventTarget currentTarget
boolean defaultPrevented
Number eventPhase
boolean isTrusted
DOMEvent nativeEvent
void preventDefault()
void stopPropagation()
DOMEventTarget target
Date timeStamp
String type
```

注意：

React v0.12 中，事件处理程序返回 `false` 不再停止事件传播，取而代之，应该根据需要手动触发 `e.stopPropagation()` 或 `e.preventDefault()`。

支持的事件

React 将事件统一化，使事件在不同浏览器上有一致的属性。

下面的事件处理程序在事件冒泡阶段被触发。如果要注册事件捕获处理程序，应该使用 `Capture` 事件，例如使用 `onClickCapture` 处理点击事件捕获阶段，而不是 `onClick`。

剪贴板事件

事件名称:

```
onCopy onCut onPaste
```

属性:

```
DOMDataTransfer clipboardData
```

键盘事件

事件名称:

```
onKeyDown onKeyPress onKeyUp
```

属性:

```
boolean altKey  
Number charCode  
boolean ctrlKey  
function getModifierState(key)  
String key  
Number keyCode  
String locale  
Number location  
boolean metaKey  
boolean repeat  
boolean shiftKey  
Number which
```

焦点事件

事件名称

```
onFocus onBlur
```

属性:

```
DOMEventTarget relatedTarget
```

表单事件

事件名称:

```
onChange onInput onSubmit
```

关于 `onChange` 事件的更多信息，参见 [表单组件](#)。

鼠标事件

事件名称:

```
onClick onContextMenu onDoubleClick onDrag onDragEnd onDragEnter onDragExit
onDragLeave onDragOver onDragStart onDrop onMouseDown onMouseEnter onMouseLeave
onMouseMove onMouseOut onMouseOver onMouseUp
```

属性:

```
boolean altKey
Number button
Number buttons
Number clientX
Number clientY
boolean ctrlKey
function getModifierState(key)
boolean metaKey
Number pageX
Number pageY
DOMEventTarget relatedTarget
Number screenX
Number screenY
boolean shiftKey
```

触控事件

事件名称:

```
onTouchCancel onTouchEnd onTouchMove onTouchStart
```

属性:

```
boolean altKey
DOMTouchList changedTouches
boolean ctrlKey
function getModifierState(key)
boolean metaKey
boolean shiftKey
DOMTouchList targetTouches
DOMTouchList touches
```

用户界面事件

事件名称:

```
onScroll
```

属性:

```
Number detail
DOMAbstractView view
```

滚轮事件

事件名称:

```
onWheel
```

属性:

```
Number deltaMode
Number deltaX
Number deltaY
Number deltaZ
```

与 DOM 的差异

React 为了性能和跨浏览器的原因，实现了一个独立于浏览器的事件和 DOM 系统。利用此功能，可以屏蔽掉一些浏览器的 DOM 的粗糙实现。

- 所有 DOM 的 properties 和 attributes （包括事件处理器）应该都是驼峰命名的，以便和标准的 JavaScript 风格保持一致。我们故意和规范不同，因为规范本身就不一致。然而，`data-*` 和 `aria-*` 属性符合规范，应该仅是小写的。
- `style` 属性接收一个带有驼峰命名风格的 JavaScript 对象，而不是一个 CSS 字符串。这与 DOM 中的 `style` 的 JavaScript 属性保持一致，更加有效，并且弥补了 XSS 安全漏洞。
- 所有的事件对象和 W3C 规范保持一致，并且所有的事件（包括提交事件）冒泡都正确地遵循 W3C 规范。参考[事件系统](#)获取更多详细信息。
- `onChange` 事件表现得和你想要的一样：当表单字段改变了，该事件就被触发，而不是等到失去焦点的时候。我们故意和现有的浏览器表现得不一致，是因为 `onChange` 是它的行为的一个错误称呼，并且 React 依赖于此事件来实时地响应用户输入。参考[表单](#)获取更多详细信息。
- 表单输入属性，例如 `value` 和 `checked`，以及 `textarea`。[这里有更多相关信息](#)。

特殊的非 DOM 属性

除了与 [DOM 的差异](#) 之外，React 也提供了一些 DOM 里面不存在的属性。

- `key`：可选的唯一的标识器。当组件在 `渲染` 过程中被各种打乱的时候，由于差异检测逻辑，可能会被销毁后重新创建。给组件绑定一个 `key`，可以持续确保组件还存在 DOM 中。更多内容请参考[这里](#)。
- `ref`：参考[这里](#)。
- `dangerouslySetInnerHTML`：提供插入纯 HTML 字符串的功能，主要为了能和生成 DOM 字符串的库整合。更多内容请参考[这里](#)。

Reconciliation

React 的关键设计目标是使 API 看起来就像每一次有数据更新的时候，整个应用重新渲染了一样。这就极大地简化了应用的编写，但是同时使 React 易于驾驭，也是一个很大的挑战。这篇文章解释了我们如何使用强大的试探法来将 $O(n^3)$ 复杂度的问题转换成 $O(n)$ 复杂度的问题。

动机 (Motivation)

生成最少的将一颗树形结构转换成另一颗树形结构的操作，是一个复杂的，并且值得研究的问题。[最优算法](#)的复杂度是 $O(n^3)$ ， n 是树中节点的总数。

这意味着要展示1000个节点，就要依次执行上十亿次的比较。这对我们的使用场景来说太昂贵了。准确地感受下这个数字：现今的 CPU 每秒钟能执行大约三十亿条指令。因此即便是最高效的实现，也不可能在一秒内计算出差异情况。

既然最优的算法都不好处理这个问题，我们实现一个非最优的 $O(n)$ 算法，使用试探法，基于如下两个假设：

1、拥有相同类的两个组件将会生成相似的树形结构，拥有不同类的两个组件将会生成不同的树形结构。 2、可以为元素提供一个唯一的标志，该元素在不同的渲染过程中保持不变。

实际上，这些假设会使在几乎所有的应用场景下，应用变得出奇地快。

两个节点的差异检查 (Pair-wise diff)

为了进行一次树结构的差异检查，首先需要能够检查两个节点的差异。此处有三种不同的情况需要处理：

不同的节点类型

如果节点的类型不同，React 将会把它们当做两个不同的子树，移除之前的那棵子树，然后创建并插入第二棵子树。

```
renderA: <div />
renderB: <span />
=> [removeNode <div />], [insertNode <span />]
```

该方法也同样应用于传统的组件。如果它们不是相同的类型，React 甚至将不会尝试计算出该渲染什么，仅会从 DOM 中移除之前的节点，然后插入新的节点。

```
renderA: <Header />
renderB: <Content />
=> [removeNode <Header />], [insertNode <Content />]
```

具备这种高级的知识点对于理解为什么 React 的差异检测逻辑又快又精确是很重要的。它对于避开树形结构大部分的检测，然后聚焦于似乎相同的部分，提供了启发。

一个 `<Header>` 元素与一个 `<Content>` 元素生成的 DOM 结构不太可能一样。React 将重新创建树形结构，而不是耗费时间去尝试匹配这两个树形结构。

如果在两个连续的渲染过程中的相同位置都有一个 `<Header>` 元素，将会希望生成一个非常相似的 DOM 结构，因此值得去做一做匹配。

DOM 节点

当比较两个 DOM 节点的时候，我们查看两者的属性，然后能够找出哪一个属性随着时间产生了变化。

```
renderA: <div id="before" />
renderB: <div id="after" />
=> [replaceAttribute id "after"]
```

React 不会把 `style` 当做难以操作的字符串，而是使用键值对对象。这就很容易地仅更新改变了的样式属性。

```
renderA: <div style={{'{'}}color: 'red'}} />
renderB: <div style={{'{'}}fontWeight: 'bold'}} />
=> [removeStyle color], [addStyle font-weight 'bold']
```

在属性更新完毕之后，递归检测所有的子级的属性。

自定义组件

我们决定两个自定义组件是相同的。因为组件是状态化的，不可能每次状态改变都要创建一个新的组件实例。React 利用新组件上的所有属性，然后在之前的组件实例上调用 `component[Will/Did]ReceiveProps()`。

现在，之前的组件就是可操作了的。它的 `render()` 方法被调用，然后差异算法重新比较新的状态和上一次的状态。

子级优化差异算法 (List-wise diff)

问题点 (Problematic Case)

为了完成子级更新, React 选用了一种很原始的方法。React 同时遍历两个子级列表, 当发现差异的时候, 就产生一次 DOM 修改。

例如在末尾添加一个元素:

```
renderA: <div><span>first</span></div>
renderB: <div><span>first</span><span>second</span></div>
=> [insertNode <span>second</span>]
```

在开始处插入元素比较麻烦。React 发现两个节点都是 span, 因此直接修改已有 span 的文本内容, 然后在后面插入一个新的 span 节点。

```
renderA: <div><span>first</span></div>
renderB: <div><span>second</span><span>first</span></div>
=> [replaceAttribute textContent 'second'], [insertNode <span>first</span>]
```

有很多的算法尝试找出变换一组元素的最小操作集合。[Levenshtein distance](#)算法能够找出这个最小的操作集合, 使用单一元素插入、删除和替换, 复杂度为 $O(n^2)$ 。即使使用 Levenshtein 算法, 不会检测出一个节点已经移到了另外一个位置去了, 要实现这个检测算法, 会引入更加糟糕的复杂度。

键 (Keys)

为了解决这个看起来很棘手的问题, 引入了一个可选的属性。可以给每个子级一个键值, 用于将来的匹配比较。如果指定了一个键值, React 就能够检测出节点插入、移除和替换, 并且借助哈希表使节点移动复杂度为 $O(n)$ 。

```
renderA: <div><span key="first">first</span></div>
renderB: <div><span key="second">second</span><span key="first">first</span></div>
=> [insertNode <span>second</span>]
```

在实际开发中, 生成一个键值不是很困难。大多数时候, 要展示的元素已经有一个唯一的标识了。当没有唯一标识的时候, 可以给组件模型添加一个新的 ID 属性, 或者计算部分内容的哈希值来生成一个键值。记住, 键值仅需要在兄弟节点中唯一, 而不是全局唯一。

■ 权衡 (Trade-offs)

同步更新算法只是一种实现细节，记住这点很重要。React 能在每次操作中重新渲染整个应用，最终的结果将会是一样的。我们定期优化这个启发式算法来使常规的应用场景更加快速。

在当前的实现中，能够检测到某个子级树已经从它的兄弟节点中移除，但是不能指出它是否已经移到了其它某个地方。当前算法将会重新渲染整个子树。

由于依赖于两个预判条件，如果这两个条件都没有满足，性能将会大打折扣。

1、算法将不会尝试匹配不同组件类的子树。如果发现正在使用的两个组件类输出的 DOM 结构非常相似，你或许想把这两个组件类改成一个组件类。实际上，这不是个问题。

2、如果没有提供稳定的键值（例如通过 `Math.random()` 生成），所有子树将会在每次数据更新中重新渲染。通过给开发者设置键值的机会，能够给特定场景写出更优化的代码。

React （虚拟）DOM 术语

在 React 的术语中，有五个核心类型，区分它们是很重要的：

React 元素

React 中最主要的类型就是 `ReactElement`。它有四个属性：`type`，`props`，`key` 和 `ref`。它没有方法，并且原型上什么都没有。

可以通过 `React.createElement` 创建该类型的一个实例。

```
var root = React.createElement('div');
```

为了渲染一个新的树形结构到 DOM 中，你创建若干个 `ReactElement`，然后传给 `React.render` 作为第一个参数，同时将第二个参数设为一个正规的 DOM 元素（`HTMLElement` 或者 `SVGElement`）。不要混淆 `ReactElement` 实例和 DOM 元素实例。一个 `ReactElement` 实例是一个轻量的，无状态的，不可变的，虚拟的 DOM 元素的表示。是一个虚拟 DOM。

```
React.render(root, document.body);
```

要添加属性到 DOM 元素，把属性对象作为第二个参数传入 `React.render`，把子级作为第三个参数传给 `React.render`。

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

如果使用 React JSX 语法，这些 `ReactElement` 实例自动创建。所以，如下代码是等价的：

```
var root = <ul className="my-list">
  <li>Text Content</li>
</ul>;
React.render(root, document.body);
```

工厂

一个 `ReactElement` 工厂就是一个简单的函数，该函数生成一个带有特殊 `type` 属性的 `ReactElement`。React 有一个内置的辅助方法用于创建工厂函数。事实上该方法就是这样的：

```
function createFactory(type) {
  return React.createElement.bind(null, type);
}
```

该函数能创建一个方便的短函数，而不是总调用 `React.createElement('div')`。

```
var div = React.createFactory('div');
var root = div({ className: 'my-div' });
React.render(root, document.body);
```

React 已经内置了常用 HTML 标签的工厂函数：

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

如果使用 JSX 语法，就不需要工厂函数了。JSX 已经提供了一种方便的短函数来创建 `ReactElement` 实例。

React 节点

一个 `ReactNode` 可以是：

- `ReactElement`
- `string` （又名 `ReactText`）
- `number` （又名 `ReactText`）
- `ReactNode` 实例数组 （又名 `ReactFragment`）

这些被用作其它 `ReactElement` 实例的属性，用于表示子级。实际上它们创建了一个 `ReactElement` 实例树。

(These are used as properties of other `ReactElement`s to represent children. Effectively they create a tree of `ReactElement`s.)

React 组件

在使用 React 开发中，可以仅使用 `ReactElement` 实例，但是，要充分利用 React，就要使用 `ReactComponent` 来封装状态化的组件。

一个 `ReactComponent` 类就是一个简单的 JavaScript 类（或者说是“构造函数”）。

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

```

    }
  });

```

当该构造函数调用的时候，应该会返回一个对象，该对象至少带有一个 `render` 方法。该对象指向一个 `ReactComponent` 实例。

```
var component = new MyComponent(props); // never do this
```

除非为了测试，正常情况下不要自己调用该构造函数。React 帮你调用这个函数。

相反，把 `ReactComponent` 类传给 `createElement`，就会得到一个 `ReactElement` 实例。

```
var element = React.createElement(MyComponent);
```

或者使用 JSX：

```
var element = <MyComponent />;
```

当该实例传给 `React.render` 的时候，React 将会调用构造函数，然后创建并返回一个 `ReactComponent`。

```
var component = React.render(element, document.body);
```

如果一直用相同的 `ReactElement` 类型和相同的 DOM 元素容器调用 `React.render`，将会总是返回相同的实例。该实例是状态化的。

```
var componentA = React.render(<MyComponent />, document.body);
var componentB = React.render(<MyComponent />, document.body);
componentA === componentB; // true
```

这就是为什么不应该创建你自己的实例。相反，在创建之前，`ReactElement` 是一个虚拟的 `ReactComponent`。新旧 `ReactElement` 可以比对，从而决定是创建一个新的 `ReactComponent` 实例还是重用已有的实例。

`ReactComponent` 的 `render` 方法应该返回另一个 `ReactElement`，这就允许组件被组装。（The `render` method of a `ReactComponent` is expected to return another `ReactElement`. This allows these components to be composed. Ultimately the render resolves into `ReactElement` with a `string` tag which instantiates a DOM `Element` instance and inserts it into the document.）

正式的类型定义

入口点 (Entry Point)

```
React.render = (ReactElement, HTMLElement | SVGELEMENT) => ReactComponent;
```

节点和元素 (Nodes and Elements)


```

type ReactNode = ReactElement | ReactFragment | ReactText;

type ReactElement = ReactComponentElement | ReactDOMElement;

type ReactDOMElement = {
  type : string,
  props : {
    children : ReactNodeList,
    className : string,
    etc.
  },
  key : string | boolean | number | null,
  ref : string | null
};

type ReactComponentElement<TProps> = {
  type : ReactClass<TProps>,
  props : TProps,
  key : string | boolean | number | null,
  ref : string | null
};

type ReactFragment = Array<ReactNode | ReactEmpty>;

type ReactNodeList = ReactNode | ReactEmpty;

type ReactText = string | number;

type ReactEmpty = null | undefined | boolean;

```

类和组件 (Classes and Components)

```

type ReactClass<TProps> = (TProps) => ReactComponent<TProps>;

type ReactComponent<TProps> = {
  props : TProps,
  render : () => ReactElement
};

```



4

温馨提示 - TIPS



简介

React tips 部分提供了一些信息，来解答你可能遇到的常见问题，也为了警告你避免常见的错误。

贡献

按照 [current tips](#) 的样式向 [React repository](#) following the [current tips](#) 提交 pull request。如果你有一个需要先 review 的文章，通过提交 PR 你可以在 [reactjs channel on freenode](#) 或 [reactjs Google group](#) 获得帮助。

行内样式

在 React 中，行内样式并不是以字符串的形式出现，而是通过一个特定的样式对象来指定。在这个对象中，key 值是用驼峰形式表示的样式名，而其对应的值则是样式值，通常来说这个值是个字符串([了解更多](#)):

```
var divStyle = {  
  color: 'white',  
  backgroundImage: 'url(' + imgUrl + ')',  
  WebkitTransition: 'all', // 注意这里的首字母'W'是大写  
  msTransition: 'all' // 'ms'是唯一一个首字母需要小写的浏览器前缀  
};  
  
React.render(<div style={divStyle}>Hello World!</div>, mountNode);
```

样式的 key 用驼峰形式表示，是为了方便与JS中通过DOM节点获取样式属性的方式保持一致（比如 'node.style.backgroundImage'）。另外浏览器前缀除了 `ms` 以外 首字母应该大写。想必 `WebkitTransition` 的首字母是“W”就不难理解了。

JSX 中的 If-Else

你没法在 JSX 中使用 `if-else` 语句，因为 JSX 只是函数调用和对象创建的语法糖。看下面这个例子：

```
// This JSX:
React.render(<div id="msg">Hello World!</div>, mountNode);

// Is transformed to this JS:
React.render(React.createElement("div", {id:"msg"}, "Hello World!"), mountNode);

// JSX 代码:
React.render(<div id="msg">Hello World!</div>, mountNode);

// 编译成 JS 是这样的:
React.render(React.createElement("div", {id:"msg"}, "Hello World!"), mountNode);
```

这意味着 `if` 语句不合适。看下面这个栗子

```
// This JSX:
<div id={if (condition) { 'msg' }}>Hello World!</div>

// Is transformed to this JS:
React.createElement("div", {id: if (condition) { 'msg' }}, "Hello World!");
```

这是不合语法的 JS 代码。不过你可以采用三元操作表达式：

```
React.render(<div id={condition ? 'msg' : ''}>Hello World!</div>, mountNode);
```

当三元操作表达式不够健壮，你也可以使用 `if` 语句来决定应该渲染那个组件。

```
var loginButton;
if (loggedIn) {
  loginButton = <LogoutButton />;
} else {
  loginButton = <LoginButton />;
}

return (
  <nav>
    <Home />
    {loginButton}
  </nav>
)
```

马上开始使用[JSX compiler](#)吧。

自闭合标签

在 JSX 中，`<MyComponent />` 是合法的，而 `<MyComponent>` 就不合法。所有的标签都必须闭合，可以是自闭和的形式，也可以是常规的闭合。

注意：

所有 React component 都可以采用自闭和的形式：`<div />`。 `<div></div>` 它们是等价的。

JSX 根节点的最大数量

目前，一个 component 的 `render`，只能返回一个节点。如果你需要返回一堆 `div`，那你必须将你的组件用一个 `div` 或 `span` 或任何其他组件包裹。

切记，JSX 会被编译成常规的 JS；因此返回两个函数也就没什么意义了，同样地，千万不要在三元操作符中放入超过一个子节点。

在样式props中快速制定像素值

当为内联样式指定一个像素值得时候，React 会在你的数字后面自动加上 “px”，所以下面这样的写法是有效的：

```
var divStyle = {height: 10}; // rendered as "height:10px"
React.render(<div style={divStyle}>Hello World!</div>, mountNode);
```

查看 [Inline Styles](#) 获得更多信息。

有时候你的确需要保持你的CSS属性的独立性。下面是不会自动加 “px” 后缀的 css 属性列表：

- columnCount
- fillOpacity
- flex
- flexGrow
- flexShrink
- fontWeight
- lineClamp
- lineHeight
- opacity
- order
- orphans
- strokeOpacity
- widows
- zIndex
- zoom

子 props 的类型

通常，一个组件的子代（`this.props.children`）是一个组件的数组：

```
var GenericWrapper = React.createClass({
  componentDidMount: function() {
    console.log(Array.isArray(this.props.children)); // => true
  },

  render: function() {
    return <div />;
  }
});

React.render(
  <GenericWrapper><span/><span/><span/></GenericWrapper>,
  mountNode
);
```

然而，当只有一个子代的时候，`this.props.children` 将会变成一个单独的组件，而不是数组形式。这样就减少了数组的占用。

```
var GenericWrapper = React.createClass({
  componentDidMount: function() {
    console.log(Array.isArray(this.props.children)); // => false

    // 注意：结果将是 5，而不是 1，因为 `this.props.children` 不是数组，而是 'hello' 字符串！
    console.log(this.props.children.length);
  },

  render: function() {
    return <div />;
  }
});

React.render(<GenericWrapper>hello</GenericWrapper>, mountNode);
```

为了让处理 `this.props.children` 更简单，我们提供了 `React.Children utilities`。

Controlled Input 值为 null 的情况

为每个 controlled component 指定 `value` 属性，来防止用户修改输入除非你希望如此。

你也许会遇到这种问题：虽然已经指定了 `value`，但是 `input` 依然可以未经允许就改变。这种情况，可能是因为一不小心将 `value` 设置成了 `undefined` 或 `null`。

下面这条代码片段展示了这个现象，一秒钟之后，文本变得可编辑了。

```
React.render(<input value="hi" />, mountNode);

setTimeout(function() {
  React.render(<input value={null} />, mountNode);
}, 1000);
```

Mounting 后 componentWillReceiveProps 未被触发

当节点初次被放入的时候 `componentWillReceiveProps` 并不会被触发。这是故意这么设计的。查看更多 [其他生命周期的方法](#)。

原因是因为 `componentWillReceiveProps` 经常会处理一些和 old props 比较的逻辑，而且会在变化之前执行；不在组件即将渲染的时候触发，这也是这个方法设计的初衷。

getInitialState 里的 Props 是一个反模式

注意：

这实际上不是一篇单独的 React 提示，因为类似的反模式设计也经常会在平时的编码中出现；这里，React 只是简单清晰地指出来这个问题

使用 props，自父级向下级传递，在使用 `getInitialState` 生成 state 的时候，经常会导致重复的“来源信任”，i.e. 如果有可能，请尽量在使用的时候计算值，以此来确保不会出现同步延迟的问题和状态保持的问题。

糟糕的例子

```
var MessageBox = React.createClass({
  getInitialState: function() {
    return {nameWithQualifier: 'Mr. ' + this.props.name};
  },

  render: function() {
    return <div>{this.state.nameWithQualifier}</div>;
  }
});

React.render(<MessageBox name="Rogers"/>, mountNode);
```

Better:

更好的写法：

```
var MessageBox = React.createClass({
  render: function() {
    return <div>{'Mr. ' + this.props.name}</div>;
  }
});

React.render(<MessageBox name="Rogers"/>, mountNode);
```

(For more complex logic, simply isolate the computation in a method.)

(对于更复杂的逻辑，最好通过方法将数据处理分离开来)

然而，如果你理清了这些，那么它也就 不是 反模式了。两者兼得不是我们的目标：

```
var Counter = React.createClass({
  getInitialState: function() {
    // naming it initialX clearly indicates that the only purpose
    // of the passed down prop is to initialize something internally
    return {count: this.props.initialCount};
  },

  handleClick: function() {
    this.setState({count: this.state.count + 1});
  },

  render: function() {
    return <div onClick={this.handleClick}>{this.state.count}</div>;
  }
});

React.render(<Counter initialCount={7}/>, mountNode);
```

组件的 DOM 事件监听

注意：

这篇文章是讲如何给 DOM 元素绑定 React 未提供的事件 ([check here for more info](#))。当你想和其他类库比如 jQuery 一起使用的时候，需要知道这些。

Try to resize the window:

```
var Box = React.createClass({
  getInitialState: function() {
    return {windowWidth: window.innerWidth};
  },

  handleResize: function(e) {
    this.setState({windowWidth: window.innerWidth});
  },

  componentDidMount: function() {
    window.addEventListener('resize', this.handleResize);
  },

  componentWillUnmount: function() {
    window.removeEventListener('resize', this.handleResize);
  },

  render: function() {
    return <div>Current window width: {this.state.windowWidth}</div>;
  }
});

React.render(<Box />, mountNode);
```

`componentDidMount` 会在 `component` 渲染完成且已经有了 DOM 结构的时候被调用。通常情况下，你可以在这绑定普通的 DOM 事件。

注意，事件的回调被绑定在了 `react` 组件上，而不是原始的元素上。React 通过一个 `autobinding` 过程自动将方法绑定到当前的组件实例上。

通过 AJAX 加载初始数据

在 `componentDidMount` 时加载数据。当加载成功，将数据存储在 `state` 中，触发 `render` 来更新你的 UI。

当执行同步请求的响应时，在更新 `state` 前，一定要先通过 `this.isMounted()` 来检测组件的状态是否还是 `mounted`。

下面这个例子请求了一个 Github 用户最近的 gist：

```
var UserGist = React.createClass({
  getInitialState: function() {
    return {
      username: '',
      lastGistUrl: ''
    };
  },

  componentDidMount: function() {
    $.get(this.props.source, function(result) {
      var lastGist = result[0];
      if (this.isMounted()) {
        this.setState({
          username: lastGist.owner.login,
          lastGistUrl: lastGist.html_url
        });
      }
    }).bind(this);
  },

  render: function() {
    return (
      <div>
        {this.state.username}'s last gist is
        <a href={this.state.lastGistUrl}>here</a>.
      </div>
    );
  }
});

React.render(
  <UserGist source="https://api.github.com/users/octocat/gists" />,
  mountNode
);
```


JSX 的 false 处理

这篇文章展示了在不同上下文中 `false` 的渲染：

被渲染成 `id="false"`：

```
React.render(<div id={false} />, mountNode);
```

String `"false"` as input value:

input value 的值将会是 `"false"` 字符串

```
React.render(<input value={false} />, mountNode);
```

没有子节点

```
React.render(<div>{false}</div>, mountNode);
```

上面这个没有被渲染成 `"false"` 字符串是应考虑到这种情况：`<div>{x > 1 && 'You have more than one item'}</div>` .

组件间的通信

对于 父-子 通信，直接 `pass props`。

对于 子-父 通信：例如：`GroceryList` 组件有一些通过数组生成的子节点。当这些节点被点击的时候，你想要展示这个节点的名字：

```
var GroceryList = React.createClass({
  handleClick: function(i) {
    console.log('You clicked: ' + this.props.items[i]);
  },

  render: function() {
    return (
      <div>
        {this.props.items.map(function(item, i) {
          return (
            <div onClick={this.handleClick.bind(this, i)} key={i}>{item}</div>
          );
        }, this)}
      </div>
    );
  }
});

React.render(
  <GroceryList items={['Apple', 'Banana', 'Cranberry']} />, mountNode
);
```

注意 `bind(this, arg1, arg2, ...)` 的使用：我们通过它向 `handleClick` 传递参数。这不是 React 的新概念，而是 JavaScript 的。

对于没有 父-子 关系的组件间的通信，你可以设置你自己的全局时间系统。在 `componentDidMount()` 里订阅事件，在 `componentWillUnmount()` 里退订，然后在事件回调里调用 `setState()`。

公开组件功能

[组件之间的交互](#)还有另一种(不常见的)方法：对父组件要调用的子组件简单的公开方法。

以一个待办事项列表为例，点击后该列表会被删除。如果只剩下一个未完成的待办事项，给它添加动画：

```
var Todo = React.createClass({
  render: function() {
    return <div onClick={this.props.onClick}>{this.props.title}</div>;
  },
  //this component will be accessed by the parent through the `ref` attribute
  animate: function() {
    console.log('Pretend %s is animating', this.props.title);
  }
});

var Todos = React.createClass({
  getInitialState: function() {
    return {items: ['Apple', 'Banana', 'Cranberry']};
  },
  handleClick: function(index) {
    var items = this.state.items.filter(function(item, i) {
      return index !== i;
    });
    this.setState({items: items}, function() {
      if (items.length === 1) {
        this.refs.item0.animate();
      }
    }).bind(this);
  },
  render: function() {
    return (
      <div>
        {this.state.items.map(function(item, i) {
          var boundClick = this.handleClick.bind(this, i);
          return (
            <Todo onClick={boundClick} key={i} title={item} ref={'item' + i} />
          );
        }, this)}
      </div>
    );
  }
});

React.render(<Todos />, mountNode);
```

或者，你可能已经通过把 `isLastUnfinishedItem` 道具传递给 `todo` 实现了上述操作，并让它在 `componentDidUpdate` 中检查道具，然后对其本身进行动画控制；然而，如果你传递不同的道具来控制动画，这很快就会变得混乱。

组件的引用

如果你正在一个大型的非 React 应用里使用 React 组件，或者准备将你的代码转换成 React，你可能需要保持组件的引用。`React.render` 会返回一个渲染后的组件的引用：

```
var myComponent = React.render(<MyComponent />, myContainer);
```

记住，JSX 并不会返回组件的引用！它只是一个 `ReactElement`：一个用来告知 React 渲染后的组件应该长什么样子的轻便的标识符。

```
var myComponentElement = <MyComponent />; // 只是 ReactElement.  
  
// Some code here...  
  
var myComponentInstance = React.render(myComponentElement, myContainer);
```

注意：

这中引用只能在最顶层级使用。在组件内部，让 `props` 和 `state` 来处理组件间的通信，而且只能通过 [refs](#) 来引用。

this.props.children undefined

你没办法通过 `this.props.children` 取得当前组件的子元素。因为 `this.props.children` 返回的是组件拥有者传递给你的 `passed onto you` 子节点。

```
var App = React.createClass({
  componentDidMount: function() {
    // This doesn't refer to the `span`s! It refers to the children between
    // last line's `</App>`, which are undefined.
    console.log(this.props.children);
  },

  render: function() {
    return <div><span/><span/></div>;
  }
});

React.render(<App></App>, mountNode);
```

如果想看更多地例子， 可以参考在 [front page](#) 里最后一个例子。

与其他类库并行使用 React

你不用非得全部采用 React。组件的生命周期事件，特别是 `componentDidMount` 和 `componentDidUpdate`，非常适合放置其他类库的逻辑代码。

```
var App = React.createClass({
  getInitialState: function() {
    return {myModel: new myBackboneModel({items: [1, 2, 3]})};
  },

  componentDidMount: function() {
    $(this.refs.placeholder.getDOMNode()).append($('
```

你还可以通过这种方式来绑定你自己的事件监听（event listeners）甚至是 [事件流（event streams）](#)。

Dangerously Set innerHTML

不合时宜的使用 `innerHTML` 可能会导致 [cross-site scripting \(XSS\)](#) 攻击。净化用户的输入来显示的时候，经常会出现错误，不合适的净化也是[导致网页攻击](#)的原因之一。

我们的设计哲学是让确保安全应该是简单的，开发者在执行“不安全”的操作的时候应该清楚地知道他们自己的意图。`dangerouslySetInnerHTML` 这个 prop 的命名是故意这么设计的，以此来警告，它的 prop 值（一个对象而不是字符串）应该被用来表明净化后的数据。

在彻底的理解安全问题后果并正确地净化数据之后，生成只包含唯一 key `__html` 的对象，并且对象的值是净化后的数据。下面是一个使用 JSX 语法的栗子：

```
function createMarkup() { return {__html: 'First &middot; Second'}; };  
<div dangerouslySetInnerHTML={createMarkup()} />
```

这么做的意义在于，当你不是有意地使用 `<div dangerouslySetInnerHTML={getUsername()} />` 时候，它并不会被渲染，因为 `getUsername()` 返回的格式是 `字符串` 而不是一个 `{__html: ''}` 对象。`{__html:...}` 背后的目的是表明它会被当成“type/taint”类型处理。这种包裹对象，可以通过方法调用返回净化后的数据，随后这种标记过的数据可以被传递给 `dangerouslySetInnerHTML`。基于这种原因，我们不推荐写这种形式的代码：`<div dangerouslySetInnerHTML={{'__html': getMarkup()}} />`。

这个功能主要被用来与 DOM 字符串操作类库一起使用，所以提供的 HTML 必须要格式清晰（例如：传递 XML 校验）

极客学院

jikexueyuan.com

中国最大的IT职业在线教育平台



更多信息请访问 

<http://wiki.jikexueyuan.com/project/react/>