

目錄

React	1.1
快速开始	1.2
新手入门	1.2.1
教程	1.2.2
React 编程思想	1.2.3
社区资源	1.3
会议	1.3.1
视频	1.3.2
补充工具	1.3.3
示例	1.3.4
手册	1.4
为什么使用 React?	1.4.1
显示数据	1.4.2
深入 JSX	1.4.2.1
JSX 展开属性	1.4.2.2
JSX 陷阱	1.4.2.3
动态交互式用户界面	1.4.3
复合组件	1.4.4
可复用组件	1.4.5
传递 Props	1.4.6
表单组件	1.4.7
与浏览器协作	1.4.8
关于Refs的更多内容	1.4.8.1
工具集成	1.4.9
插件	1.4.10
动画	1.4.10.1
双向绑定辅助	1.4.10.2
类名操纵	1.4.10.3
Test Utilities	1.4.10.4
Cloning ReactElements	1.4.10.5

Keyed Fragments	1.4.10.6
Immutability Helpers	1.4.10.7
PureRenderMixin	1.4.10.8
Performance Tools	1.4.10.9
浅比较	1.4.10.10
Advanced Performance	1.4.11
Context	1.4.12
参考	1.5
Top-Level API	1.5.1
组件 API	1.5.2
组件的规范和生命周期	1.5.3
Tags 和属性	1.5.4
事件系统	1.5.5
DOM 的不同之处	1.5.6
特殊的 Non-DOM Attributes	1.5.7
Reconciliation	1.5.8
Web Components	1.5.9
React (虚拟) DOM 术语	1.5.10
FLUX	1.6
概览	1.6.1
TodoMVC 教程	1.6.2
TIPS	1.7
Introduction	1.7.1
Inline Styles	1.7.2
If-Else in JSX	1.7.3
Self-Closing Tag	1.7.4
Maximum Number of JSX Root Nodes	1.7.5
Shorthand for Specifying Pixel Values in style props	1.7.6
Type of the Children props	1.7.7
Value of null for Controlled Input	1.7.8
componentWillReceiveProps Not Triggered After Mounting	1.7.9
Props in getInitialState Is an Anti-Pattern	1.7.10
DOM Event Listeners in a Component	1.7.11
Load Initial Data via AJAX	1.7.12

False in JSX	1.7.13
Communicate Between Components	1.7.14
Expose Component Functions	1.7.15
this.props.children undefined	1.7.16
Use React with Other Libraries	1.7.17
Dangerously Set innerHTML	1.7.18

React

React 是用于构建用户界面的 JAVASCRIPT 库。

- 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。

快速开始

- [新手入门](#)
- [教程](#)
- [React 编程思想](#)

新手入门

JSFiddle

开始学习 React 的最简单的方法是用下面 JSFiddle 的 Hello World：

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

通过 npm 使用 React

我们建议在 React 中使用 CommonJS 模块系统，比如 [browserify](#) 或 [webpack](#)。使用 `react` 和 `react-dom` npm 包。

```
// main.js
var React = require('react');
var ReactDOM = require('react-dom');

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

要用 [browserify](#) 安装 React DOM 和构建你的包。

```
$ npm install --save react react-dom babelify babel-preset-react
$ browserify -t [ babelify --presets [ react ] ] main.js -o bundle.js
```

要用 [webpack](#) 安装 React DOM 和构建你的包：

```
$ npm install --save react react-dom babel-preset-react
$ webpack
```

注意：

如果你正在使用 ES2015，你将要使用 `babel-preset-es2015` 包。

不用 npm 的快速开始

如果你现在还没准备要使用 npm，你可以下载这个已经包含了预构建的 React 和 React DOM 拷贝的入门套件。

下载入门套件 0.14.7

在入门教程包的根目录，创建一个含有如下代码的 `helloworld.html`。

```
<!DOCTYPE html>
<html>
  <head>
    <meta charset="UTF-8" />
    <title>Hello React!</title>
    <script src="build/react.js"></script>
    <script src="build/react-dom.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/browser.min.js"></script>
  </head>
  <body>
    <div id="example"></div>
    <script type="text/babel">
      ReactDOM.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/babel">` 标签，并引入 Babel 来完成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`。

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

然后在 `helloworld.html` 引用该文件：

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

注意一些浏览器（比如 Chrome）会在使用 HTTP 以外的协议加载文件时失败。

离线转换

先安装 [Babel](#) 命令行工具（需要 [npm](#)）：

```
npm install --global babel-cli
npm install babel-preset-react
```

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

```
babel --presets react src --watch --out-dir build
```

注意：

如果你正在使用 ES2015, 你将需要使用 `babel-preset-es2015` 包。

`build/helloworld.js` 会在你对文件进行修改时自动生成。阅读 [Babel CLI 文档](#) 了解高级用法。

```
ReactDOM.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>
    <meta charset="UTF-8" />
    <title>Hello React!</title>
    <script src="build/react.js"></script>
    <script src="build/react-dom.js"></script>
    <!-- 不需要 Babel! -->
  </head>
  <body>
    <div id="example"></div>
    <script src="build/helloworld.js"></script>
  </body>
</html>
```

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](#) 等

恭喜你，欢迎来到 React 的世界。

教程

我们将建立一个你可以放进博客的简单却真实的评论框，一个 Disqus、LiveFyre 或 Facebook comments 提供的实时评论的基础版本。

我们将提供：

- 一个所有评论的视图
- 一个用于提交评论的表单
- 为你提供制定后台的挂钩（Hooks）

同时也会有一些简洁的功能：

- 优化的评论：评论在它们保存到服务器之前就显示在列表里,所以感觉很快。
- 实时更新：其他用户的评论被实时浮现到评论中。
- **Markdown** 格式化：用户可以用 Markdown 格式化它们的文字。

想要跳过所有内容，只查看源代码？

全在 [GitHub](#)。

运行服务器

为了开始本教程，我们将需要需要一个运行着的服务器。这将是纯粹用来获取和保存数据的伺服终端。为了让这尽可能的容易，我们已经用许多不同的语言编写了简单的服务器，它正好完成我们需要的事。你可以[查看源代码](#) 或者 [下载 zip 文件](#) 包括了所有你开始学习需要的东西

为了简单起见，我们将要运行的服务器使用 `JSON` 文件作为数据库。你不会在生产环境运行这个，但是它让我们更容易模拟使用一个 **API** 时你可能会做的事。一旦你启动服务器，它将会支持我们的 **API** 终端，同时也将伺服我们需要的静态页面。

开始

对于此教程，我们将使它尽可能的容易。被包括在上面讨论的服务器包里的是一个我们将在其中工作的 **HTML** 文件。在你最喜欢的编辑器里打开 `public/index.html`。它应该看起来像这样（可能有一些小的不同，稍后我们将添加一个额外的 `<script>` 标签）：

```
<!-- index.html -->
<!DOCTYPE html>
<html>
  <head>
    <meta charset="utf-8" />
    <title>React Tutorial</title>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.7/react.js"></script>
  >
    <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.7/react-dom.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/browser.min.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/2.2.0/jquery.min.js"></script>
  </head>
  <body>
    <div id="content"></div>
    <script type="text/babel" src="scripts/example.js"></script>
    <script type="text/babel">
      // To get started with this tutorial running your own code, simply remove
      // the script tag loading scripts/example.js and start writing code here.
    </script>
  </body>
</html>
```

在本教程剩余的部分，我们将在此 `script` 标签中编写我们的 JavaScript 代码。我们没有任何高级的实时加载所以在保存以后你需要刷新你的浏览器来观察更新。通过在浏览器打开 `http://localhost:3000` 关注你的进展。当你没有任何修改第一次加载时，你将看到我们将要准备建立的已经完成的产品。当你准备开始工作，请删除前面的 `<script>` 标签然后你就可以继续了。

注意：

我们在这里引入 jQuery 是因为我们想简化我们未来的 ajax 请求，但这对 React 的正常工作不是必要的。

你的第一个组件

React 中都是关于模块化、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，仅是一个简单的 `<div>`：

```
// tutorial1.js
var CommentBox = React.createClass({
  render: function() {
    return (
      <div className="commentBox">
        Hello, world! I am a CommentBox.
      </div>
    );
  }
});
ReactDOM.render(
  <CommentBox />,
  document.getElementById('content')
);
```

注意原生的 HTML 元素以小写开头，而制定的 **React** 类以大写开头。

JSX 语法

首先你会注意到你的 JavaScript 中 XML 式的语法。我们有一个简单的预编译器，将这种语法糖转换成单纯的 JavaScript：

```
// tutorial1-raw.js
var CommentBox = React.createClass({displayName: 'CommentBox',
  render: function() {
    return (
      React.createElement('div', {className: "commentBox"},
        "Hello, world! I am a CommentBox."
      )
    );
  }
});
ReactDOM.render(
  React.createElement(CommentBox, null),
  document.getElementById('content')
);
```

它的使用是可选的，但是我们发现 JSX 语法比单纯的 JavaScript 更加容易使用。阅读更多关于 [JSX 语法的文章](#)。

发生了什么

我们在一个 JavaScript 对象中传递一些方法到 `React.createClass()` 来创建一个新的 React 组件。这些方法中最重要的是 `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` 组件的实例化。你可以把这些看做是 React 知道如何处理的标记或者是一些数据。React 是安全的。我们不生成 HTML 字符串，因此 XSS 防护是默认特性。

你没有必要返回基本的 HTML。你可以返回一个你（或者其他入）创建的组件树。这就使 React 组件化：一个可维护前端的关键原则。

`ReactDOM.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。

组合组件

让我们为 `CommentList` 和 `CommentForm` 建造骨架，它们将会，再一次的，是一些简单的 `<div>`。添加这两个组件到你的文件里，保持现存的 `CommentBox` 声明和 `ReactDOM.render` 调用：

```
// 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>
    );
  }
});
```

接着，更新 `CommentBox` 以使用这些新的组件：

```
// 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 标签和我们建立的组件。HTML 组件是正常的 React 组件，就和你定义的一样，只有一个区别。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。这是为了避免污染全局命名空间。

使用 props

让我们创建 `Comment` 组件，它将依赖于从父级传来的数据。从父级传来的数据在子组件里作为‘属性’可供使用。这些‘属性’可以通过 `this.props` 访问。使用属性，我们将能读取从 `CommentList` 传递给 `Comment` 的数据，并且渲染一些标记：

```
// tutorial4.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），你可以把文本或者 React 组件放置到树中。我们以 `this.props` 的 `keys` 访问传递给组件的命名属性，以 `this.props.children` 访问任何嵌套的元素。

组件的属性

既然我们已经定义了 `Comment` 组件，我们将要传递作者名和评论文字给它。这允许我们为每个评论重用相同的代码。现在让我们在我们的 `CommentList` 里添加一些评论。

```
// tutorial5.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>
    );
  }
});
```

注意，我们已经从 `CommentList` 组件传递了一些数据到 `Comment` 组件。例如，我们传递了 *Pete Hunt*（通过属性）和 *This is one comment*（通过 XML- 风格的子节点）给第一个 `Comment`。如上面提到的那样，`Comment` 组件将会通过 `this.props.author` 和 `this.props.children` 访问这些‘属性’。

添加 Markdown

Markdown 是一种简单的内联格式化你的文字的方法。例如，用星号包围文本将会使其强调突出。

首先，添加第三方库 **marked** 到你的应用。这是一个 JavaScript 库，接受 Markdown 文本并且转换为原始的 HTML。这需要在你的头部有一个 `script` 标签（那个我们已经在 React 操场上包含了的标签）：

```
<!-- index.html -->
<head>
  <meta charset="utf-8" />
  <title>React Tutorial</title>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.7/react.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/react/0.14.7/react-dom.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/browser.min.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/jquery/2.1.1/jquery.min.js"></script>
  <script src="https://cdnjs.cloudflare.com/ajax/libs/marked/0.3.5/marked.min.js"></script>
</head>
```

然后，让我们转换评论文本为 Markdown 并输出它：


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

我们在这里唯一做的就是调用 `marked` 库。我们需要把从 `React` 的包裹文本来的 `this.props.children` 转换成 `marked` 能理解的原始字符串，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看起来像这样：`"<p> This is another comment </p>"`。我们想让这些标签真正地渲染为 HTML。

那是 `React` 在保护你免受 [XSS 攻击](#)。有一个方法解决这个问题，但是框架会警告你别使用这种方法：

```
// tutorial7.js
var Comment = React.createClass({
  rawMarkup: function() {
    var rawMarkup = marked(this.props.children.toString(), {sanitize: true});
    return { __html: rawMarkup };
  },

  render: function() {
    return (
      <div className="comment">
        <h2 className="commentAuthor">
          {this.props.author}
        </h2>
        <span dangerouslySetInnerHTML={this.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 `marked` 我们将利用这个后门。

记住：使用这个功能你会依赖于 `marked` 是安全的。既然如此，我们传递 `sanitize: true` 告诉 `marked` escape 源码里任何的 HTML 标记，而不是直接不变的让他们通过。

挂钩数据模型

到目前为止我们已经完成了在源码里直接插入评论。作为替代，让我们渲染一团 JSON 数据到评论列表里。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"}
];
```

我们需要以一种模块化的方式将这个数据传入到 `CommentList`。修改 `CommentBox` 和 `ReactDOM.render()` 方法，以通过 `props` 传入数据到 `CommentList`：

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

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

既然现在数据在 `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

让我们用一些来自服务器的动态数据替换硬编码的数据。我们将移除数据的 `prop`，用获取数据的 `URL` 来替换它：

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

这个组件不同于和前面的组件，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。

注意：此代码在这个阶段不会工作。

Reactive state

迄今为止,基于它自己的`props`，每个组件都渲染了自己一次。`props`是不可变的：它们从父级传来并被父级“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()` 改变它。每当`state`更新，组件就重新渲染自己。

`render()` 方法被声明为一个带有 `this.props` 和 `this.state` 的函数。框架保证了 UI 总是与输入一致。

当服务器获取数据时，我们将会改变我们已有的评论数据。让我们给 `CommentBox` 组件添加一组评论数据作为它的状态：

```
// 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()` 在生命周期里只执行一次，并设置组件的初始状态。

更新状态

当组件第一次创建时，我们想从服务器获取一些 JSON 并且更新状态以反映最新的数据。我们将用 jQuery 来发送一个异步请求到我们刚才启动的服务器以获取我们需要的数据。看起来像这样：

```
[
  {"author": "Pete Hunt", "text": "This is one comment"},
  {"author": "Jordan Walke", "text": "This is *another* comment"}
]
```

```
// tutorial13.js
var CommentBox = React.createClass({
  getInitialState: function() {
    return {data: []};
  },
  componentDidMount: function() {
    $.ajax({
      url: this.props.url,
      dataType: 'json',
      cache: false,
      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>
    );
  }
});
```

这里，`componentDidMount` 是一个当组件被渲染时被 **React** 自动调用的方法。动态更新的关键是对 `this.setState()` 的调用。我们用新的从服务器来的替换掉旧的评论组，然后 **UI** 自动更新自己。因为这种反应性，仅是一个微小的变化就添加了实时更新。我们这里将用简单的轮询，但是你可以容易的使用 **WebSockets** 或者其他技术。

```
// tutorial14.js
var CommentBox = React.createClass({
  loadCommentsFromServer: function() {
    $.ajax({
      url: this.props.url,
      dataType: 'json',
      cache: false,
      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>
    );
  }
});

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

我们在这里做的全部事情是把 **AJAX** 调用移动到独立的方法里，然后在组件第一次加载时及其后每2秒调用它。试着在你的浏览器里运行它并且改变 `comments.json` 文件（在你的服务器的相同目录）；2秒内，变化将会显现！

添加新评论

现在是时候建立表单了，我们的 `CommentForm` 组件应该询问用户他们的名字和评论文本然后发送一个请求到服务器来保存评论。

```
// 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>
    );
  }
});
```

让我们做一个交互式的表单。当用户提交表单时，我们应该清空它，提交一个请求给服务器，和刷新评论列表。要开始，让我们监听表单的提交事件并清空它。

```
// tutorial16.js
var CommentForm = React.createClass({
  handleSubmit: function(e) {
    e.preventDefault();
    var author = React.findDOMNode(this.refs.author).value.trim();
    var text = React.findDOMNode(this.refs.text).value.trim();
    if (!text || !author) {
      return;
    }
    // TODO: send request to the server
    React.findDOMNode(this.refs.author).value = '';
    React.findDOMNode(this.refs.text).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(camelCase)给组件绑定事件处理器。我们给表单绑定一个 `onSubmit` 处理器，它在表单提交了合法的输入后清空表单字段。

在事件中调用 `preventDefault()` 来阻止浏览器提交表单的默认行为。

Refs

我们利用 `ref` 属性给予组件赋予名字，`this.refs` 引用组件。我们可以在组件上调用 `React.findDOMNode(component)` 获取原生的浏览器DOM元素。

回调函数作为属性

当用户提交评论时，我们需要刷新评论列表来包含这条新评论。在 `CommentBox` 中完成所有逻辑是有道理的，因为 `CommentBox` 拥有代表了评论列表的状态(`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` 方法中以传递一个新的回调函数（`handleCommentSubmit`）到子组件完成这件事，绑定它到子组件的 `onCommentSubmit` 事件上。无论事件什么时候触发，回调函数都将被调用：

```
// tutorial17.js
var CommentBox = React.createClass({
  loadCommentsFromServer: function() {
    $.ajax({
      url: this.props.url,
      dataType: 'json',
      cache: false,
      success: function(data) {
        this.setState({data: data});
      }.bind(this),
      error: function(xhr, status, err) {
        console.error(this.props.url, status, err.toString());
      }.bind(this)
    });
  },
  handleCommentSubmit: function(comment) {
    // TODO: submit to the server and refresh the list
  },
  getInitialState: function() {
    return {data: []};
  },
  componentDidMount: function() {
    this.loadCommentsFromServer();
    setInterval(this.loadCommentsFromServer, this.props.pollInterval);
  },
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList data={this.state.data} />
        <CommentForm onCommentSubmit={this.handleCommentSubmit} />
      </div>
    );
  }
});
```

当用户提交表单时，让我们从 `CommentForm` 调用这个回调函数：


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

既然现在回调函数已经就绪，我们所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',
      cache: false,
      success: function(data) {
        this.setState({data: data});
      }.bind(this),
      error: function(xhr, status, err) {
        console.error(this.props.url, status, err.toString());
      }.bind(this)
    });
  },
  handleCommentSubmit: function(comment) {
    $.ajax({
      url: this.props.url,
      dataType: 'json',
      type: 'POST',
      data: comment,
      success: function(data) {
        this.setState({data: data});
      }.bind(this),
      error: function(xhr, status, err) {
        console.error(this.props.url, status, err.toString());
      }.bind(this)
    });
  },
  getInitialState: function() {
    return {data: []};
  },
  componentDidMount: function() {
    this.loadCommentsFromServer();
    setInterval(this.loadCommentsFromServer, this.props.pollInterval);
  },
  render: function() {
    return (
      <div className="commentBox">
        <h1>Comments</h1>
        <CommentList data={this.state.data} />
        <CommentForm onCommentSubmit={this.handleCommentSubmit} />
      </div>
    );
  }
});
```

优化: 优化的更新

我们的应用现在已经功能完备，但是它感觉很慢，因为在评论出现在列表前必须等待请求完成。我们可以优化添加这条评论到列表以使应用感觉更快。

```
// tutorial20.js
var CommentBox = React.createClass({
  loadCommentsFromServer: function() {
    $.ajax({
      url: this.props.url,
      dataType: 'json',
      cache: false,
      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) {
        this.setState({data: comments});
        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！

你刚刚通过几个简单的步骤建立了一个评论框。学习更多关于[为什么使用 React](#), 或者深入[API 参考](#) 开始钻研！祝你好运！

React 编程思想

by Pete Hunt

在我看来，React 是构建大型，快速 Web app 的首选方式。它已经在 Facebook 和 Instagram 被我们有了广泛的应用。

React 许多优秀的部分之一，是它使得你在构建 app 的过程中不断思考。在本文里，我将带你经历一次使用 React 构建可搜索的商品数据表的思考过程。

从模型（mock）开始

想象我们已经有个一个 JSON API 和一个来自设计师的模型。我们的设计师显然做得不够好，因为模型看起来像这样：

☐ Only show products in stock

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

我们的 JSON API 返回一些看起来像这样的数据：

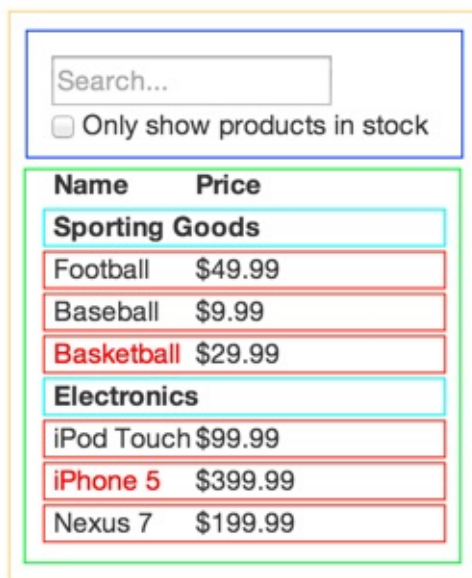
```
[
  {category: "Sporting Goods", price: "$49.99", stocked: true, name: "Football"},
  {category: "Sporting Goods", price: "$9.99", stocked: true, name: "Baseball"},
  {category: "Sporting Goods", price: "$29.99", stocked: false, name: "Basketball"},
  {category: "Electronics", price: "$99.99", stocked: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

第一步：把UI拆分为一个组件的层级

首先你想要做的，是在模型里的每一个组件周围绘制边框，并给它们命名。如果你和设计师一起工作，他们应该已经完成这步了，所以去和他们谈谈！他们的 Photoshop 图层名也许最终会成为你的 React 组件名。

但是你如何知道什么东西应该是独立的组件？只需在你创建一个函数或者对象时，根据是否使用过相同技术来做决定。一种这样的技术是[单一功能原则（single responsibility principle）](#)，也就是一个组件在理想情况下只做一件事情。如果它最终增长了，它就应该被分解为更小的组件。

既然你频繁显示一个 JSON 的数据模型给用户，你会发现，如果你的模型构建正确，你的 UI（因此也有你的组件结构）就将映射良好。那是因为 UI 和数据模型趋向附着于相同的信息架构，这意味着，把你的 UI 分离为组件的工作通常是琐碎的，只需把 UI 拆分成能准确对应数据模型的每块组件。



在这里你会看到，在我们的简单 APP 里有五个组件。我用斜体表示每个组件的数据。


1. `FilterableProductTable` (橙色): 包含示例的整体
2. `SearchBar` (蓝色): 接收所有 用户输入
3. `ProductTable` (绿色): 基于 用户输入 显示和过滤 数据集合(*data collection*)
4. `ProductCategoryRow` (蓝绿色): 为每个 分类 显示一个列表头
5. `ProductRow` (红色): 为每个 商品 显示一行

如果你看着 `ProductTable`，你会看到表头(包含了 "Name" 和 "Price" 标签) 不是独立的组件。这是一个个人喜好问题，并且无论采用哪种方式都有争论。对于这个例子，我把它留做 `ProductTable` 的一部分，因为它是 *data collection* 渲染的一部分，而 *data collection* 渲染是 `ProductTable` 的职责。然而，当列表头增长到复杂的时候(例如 如果我们添加排序功能)，那么使它成为独立的 `ProductTableHeader` 组件无疑是有意义的。

既然现在我们已经识别出了我们模型中的组件，让我们把他们安排到一个层级中。这很容易。在模型中，出现在一个组件里面的另一组件，应该在层级中表现为一种子级关系：

- FilterableProductTable
 - searchBar
 - ProductTable
 - ProductCategoryRow
 - ProductRow

第二步：用**React**创建一个静态版本

```
JavaScript 1.7  HTML  CSS  Resources  Result  Edit in JSFiddle 
```

```
var ProductCategoryRow = React.createClass({
  render: function() {
    return (<tr><th colSpan="2">{this.props.category}</th></tr>);
  }
});

var ProductRow = React.createClass({
  render: function() {
    var name = this.props.product.stocked ?
      this.props.product.name :
      <span style={{color: 'red'}}>
        {this.props.product.name}
      </span>;
    return (
      <tr>
        <td>{name}</td>
        <td>{this.props.product.price}</td>
      </tr>
    );
  }
});

var ProductTable = React.createClass({
  render: function() {
    var rows = [];
    var lastCategory = null;
```

既然你已经有了你的组件层级，是时候实现你的app了。简单的方式是构建一个版本，它取走你的数据模型并渲染UI，除了没有互动性。这是将过程解耦的最好办法，因为构建一个静态版本需要不假思索地写很多代码，而添加互动性需要很多思考但不需要太多代码。之后我们将会看到原因。

要构建一个静态版本 app 来渲染你的数据模型，你将会想到构建一个重用其它组件并利用 *props* 传递数据的组件。*props* 是一种从父级传递数据到子级的方式。如果你对 *state* 的观念很熟悉，绝不要用 *state* 来构建这个静态版本。*State* 仅仅是为互动性，也就是随时间变化的

数据所预留的。由于这是一个静态版本，你还不需要用到它。

你可以自顶向下或自底向上的构建。也就是说，你可以既从较高的层级（比如从 `FilterableProductTable` 开始）也可以从较低的层级（`ProductRow`）开始构建组件。在较简单的例子里，通常自顶向下要容易一些，然而在更大的项目上，自底向上地构建更容易，并且更方便伴随着构建写测试。

在这一步的最后，你会获得一个渲染数据模型的可重用组件库。这些组件只有 `render()` 方法，因为这是一个静态版本。在层级顶端的组件（`FilterableProductTable`）将会接受你的数据模型，并将其作为一个 `prop`。如果你改变了底层数据模型，并且再次调用 `React.render()`，UI 将会更新。你可以很容易地看到 UI 是如何更新的，以及哪里变动了，因为这没什么复杂的。`React` 的单向数据流（也被称为单向绑定）使一切保持了模块化和快速。

如果你在执行这步时需要帮助，请参阅 [React 文档](#)。

小插曲：props vs state

在 `React` 里有两种数据“模型”：`props` 和 `state`。明白这二者之间的区别是很重要的；如果你不是很确定它们之间的区别，请概览 [React 官方文档](#)

第三步：确定最小（但完备）的 UI state 表达

要让你的 UI 互动，你需要做到触发底层数据模型发生变化。`React` 用 `state` 来让此变得容易。

要正确的构建你的 app，你首先需要思考你的 app 需要的可变 `state` 的最小组。这里的关键是 DRY 原则：*Don't Repeat Yourself*（不要重复自己）。想出哪些是你的应用需要的绝对最小 `state` 表达，并按需计算其他任何数据。例如，如果你要构建一个 TODO list，只要保持一个 TODO 项的数组；不要为了计数保持一个单独的 `state` 变量。当你想渲染 TODO 的计数时，简单的采用 TODO 项目的数组长度作为替代。

考虑我们示例应用中的数据所有块，包括：

- 原始的商品列表
- 用户输入的搜索文本
- 复选框的值
- 商品的过滤列表

让我们逐个检查出哪一个是 `state`，只需要简单地问以下三个问题：


1. 它是通过 `props` 从父级传递来的吗？如果是，它可能不是 `state`。
2. 它随时间变化吗？如果不是，它可能不是 `state`。
3. 你能基于其他任何组件里的 `state` 或者 `props` 计算出它吗？如果是，它可能不是 `state`。

原始的商品列表以 `props` 传入，所以它不是 `state`。搜索文本和复选框看起来是 `state`，因为他们随时间变化并且不能从任何东西计算出。最后，过滤出的商品列表不是 `state`，因为它可以通过原始列表与搜索文本及复选框的值组合计算得出。

所以最后,我们的 `state` 是:

- 用户输入的搜索文本
- `checkbox` 的值

第四步：确定你的 `state` 应该存在于哪里

```
JavaScript 1.7  HTML  CSS  Resources  Result  Edit in JSFiddle 
```

```
var ProductCategoryRow = React.createClass({
  render: function() {
    return (<tr><th colSpan="2">{this.props.category}</th></tr>);
  }
});

var ProductRow = React.createClass({
  render: function() {
    var name = this.props.product.stocked ?
      this.props.product.name :
      <span style={{color: 'red'}}>
        {this.props.product.name}
      </span>;
    return (
      <tr>
        <td>{name}</td>
        <td>{this.props.product.price}</td>
      </tr>
    );
  }
});

var ProductTable = React.createClass({
  render: function() {
    var rows = [];
    var lastCategory = null;
```

OK，我们已经确定好应用的最小 `state` 集合是什么。接下来，我们需要确定哪个组件可以改变，或者拥有这个 `state`。

记住：`React` 总是在组件层级中单向数据流动的。可能不能立刻明白哪些组件应该拥有哪些 `state`。这对于新手在理解上经常是最具挑战的一部分，所以跟着这几步来弄明白它：

对于你的应用里每一个数据块：

- 确定哪些组件要基于 **state** 来渲染内容。
- 找到一个共同的拥有者组件（在所有需要这个**state**组件的层次之上，找出共有的单一组件）。
- 要么是共同拥有者，要么是其他在层级里更高级的组件应该拥有这个**state**。
- 如果你不能找到一个组件让其可以有意义地拥有这个 **state**，可以简单地创建一个新的组件 **hold** 住这个**state**，并把它添加到比共同拥有者组件更高的层级上。

让我们使用这个策略浏览一遍我们的应用：

- `ProductTable` 需要基于 **state** 过滤产品列表，`SearchBar` 需要显示搜索文本和选择状态。
- 共同拥有者组件是 `FilterableProductTable`。
- 对于过滤文本和选择框值存在于 `FilterableProductTable`，从概念上讲是有意义的。

酷，我们已经决定了我们的 **state** 存在于 `FilterableProductTable`。首先，添加一个 `getInitialState()` 方法到 `FilterableProductTable`，返回 `{filterText: '', inStockOnly: false}` 来反映应用的初始状态。然后，传递 `filterText` 和 `inStockOnly` 给 `ProductTable` 和 `SearchBar` 作为 **prop**。最后，使用这些 **prop** 来过滤 `ProductTable` 中的行和设置 `SearchBar` 的表单项的值。

你可以开始看看你的应用将有怎样的行为了：设置 `filterText` 为 `"ball"` 并刷新你的 **app**。你将可以看到数据表被正确地更新。

第五步：添加反向数据流

```

var ProductCategoryRow = React.createClass({
  render: function() {
    return (<tr><th colSpan="2">{this.props.category}</th></tr>);
  }
});

var ProductRow = React.createClass({
  render: function() {
    var name = this.props.product.stocked ?
      this.props.product.name :
      <span style={{color: 'red'}}>
        {this.props.product.name}
      </span>;
    return (
      <tr>
        <td>{name}</td>
        <td>{this.props.product.price}</td>
      </tr>
    );
  }
});

var ProductTable = React.createClass({
  render: function() {
    var rows = [];
    var lastCategory = null;

```

到目前为止，我们已经构建了一个应用，它以 **props** 和 **state** 沿着层级向下流动的功能正确渲染。现在是时候支持另一种数据流动了：在层级深处的表单组件需要更新

`FilterableProductTable` 里的 **state**。

React 让数据显式流动，使你理解应用如何工作变得简单，但是相对于传统的双向数据绑定，确实需要多打一些字。**React** 提供了一个叫做 `ReactLink` 的插件来使这种模式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。

如果你尝试在当前版本的示例中输入或者选中复选框，你会发现 **React** 忽略了你的输入。这是有意的，因为我们已经设置了 `input` 的 `value` **prop** 值总是与 `FilterableProductTable` 传递过来的 `state` 一致。

让我们思考下希望发生什么。我们想确保每当用户改变表单，就通过更新 **state** 来反映用户的输入。由于组件应该只更新自己拥有的 **state**，`FilterableProductTable` 将会传递一个回调函数给 `SearchBar`，每当 **state** 应被更新时回调函数就会被调用。我们可以使用 `input` 的 `onChange` 事件来接受它的通知。`FilterableProductTable` 传递的回调函数将会调用 `setState()`，然后应用将会被更新。

虽然这听起来复杂，但是实际上只是数行代码。并且这明确显示出了数据在应用中从始至终是如何流动的。

好了，就是这样

希望这给了你一个怎样思考用React构建组件和应用的概念。虽然可能比你过往的习惯要多敲一点代码，但记住，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、显式的代码是极为容易的。当你开始构建大型组件库时，你会非常感激这种清晰性和模块化，并且随着代码的重用，你的代码行数将会开始缩减。:)

社区资源

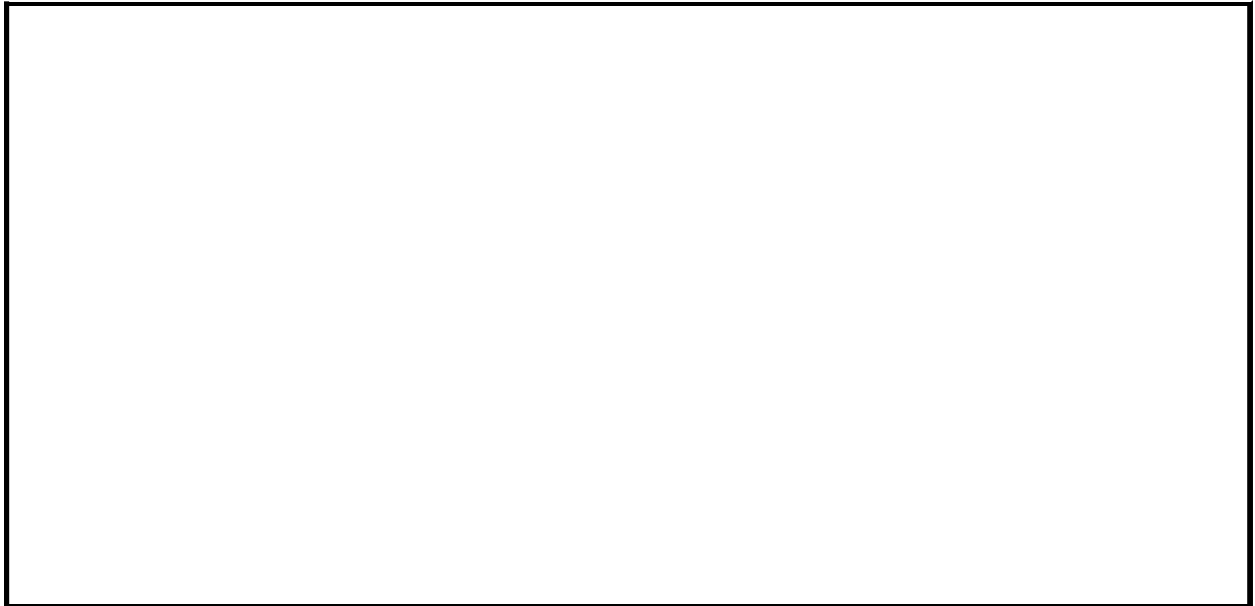
- [会议](#)
- [视频](#)
- [补充工具](#)
- [示例](#)

会议

React.js Conf 2015

一月 28 & 29

[Website](#) - [Schedule](#) - [Videos](#)



ReactEurope 2015

七月 2 & 3

[Website](#) - [Schedule](#)

Reactive 2015

十一月 2-4

[Website](#) - [Schedule](#)

ReactEurope 2016

六月 2 & 3

[Website](#) - [Schedule](#)

视频

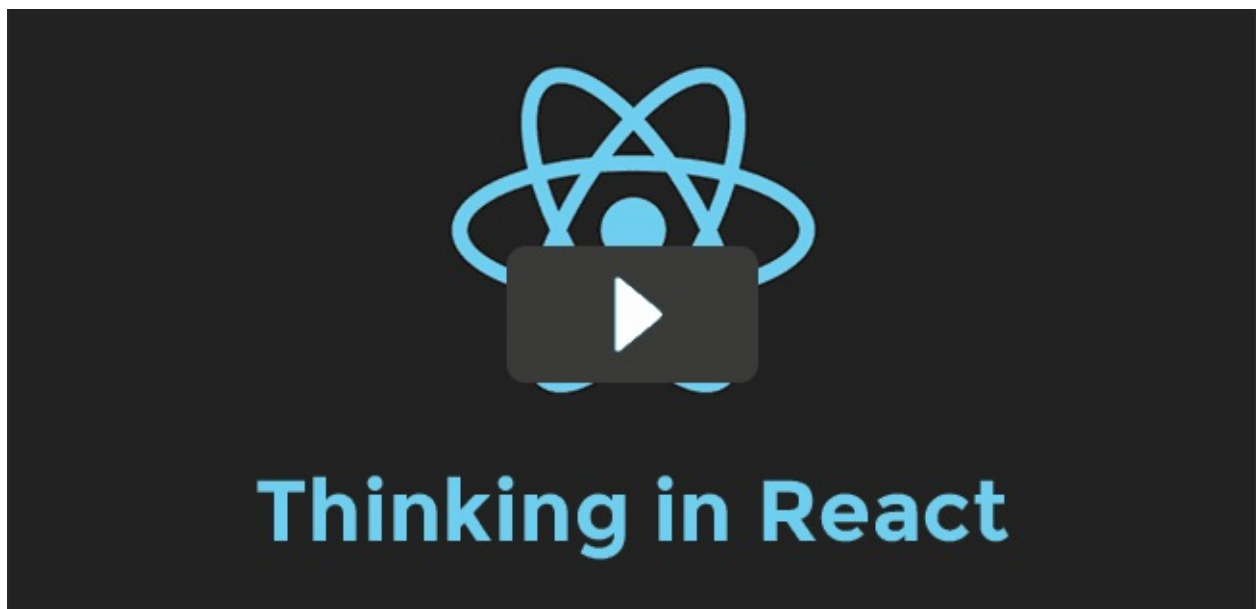
Rethinking best practices - JSConf.eu



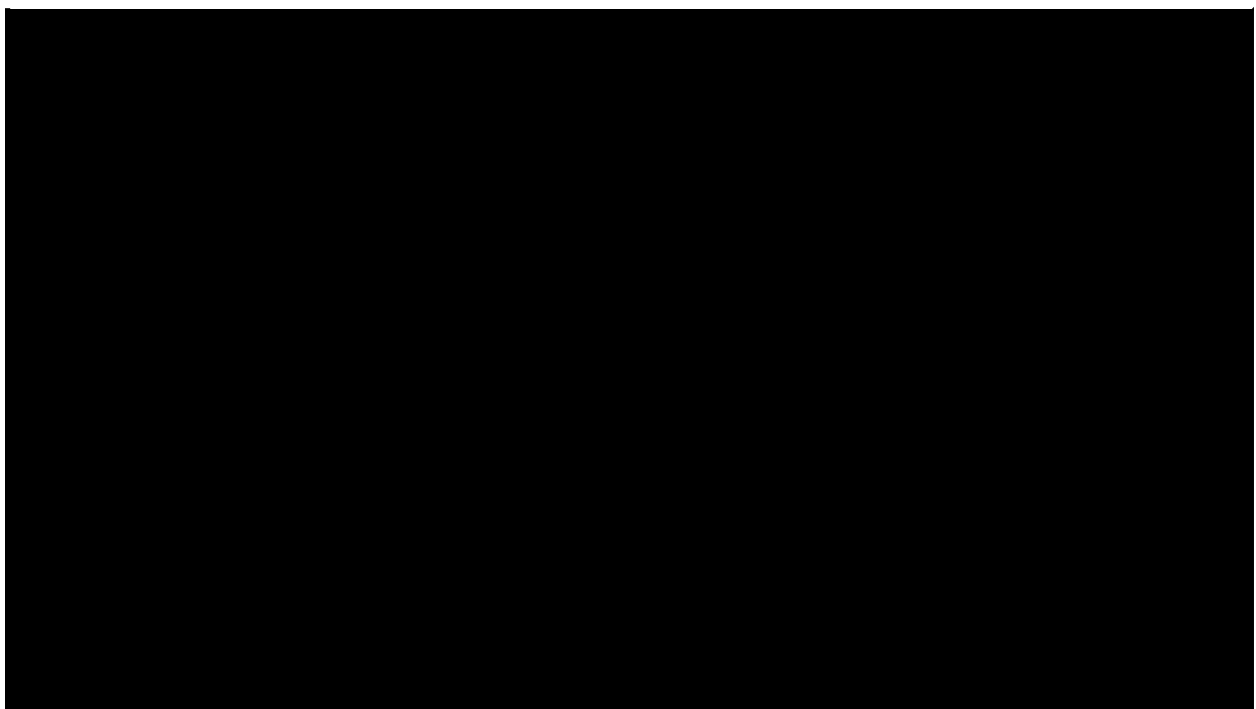
"在 Facebook 和 Instagram, 我们正在努力挑战React在web上能达到的极限。我的讲话会从对框架的简单介绍开始，然后深入三个有争议的话题：扔掉模板的概念并用JavaScript构建 views, 当数据改变“re-rendering”你的整个应用，以及一个DOM和events的轻量级实现。" -- [Pete Hunt](#)

Thinking in react - tagtree.tv

一个 [tagtree.tv](#) 传达 [Thinking in React](#) 原则的视频 在构建一个简单app时。



Secrets of the Virtual DOM - MtnWest JS



"在这次讲座里，我会讨论为什么我们构建了一个虚拟 DOM，它比起其他系统如何，以及它与未来浏览器技术的关系。"-- [Pete Hunt](#)

Going big with React

"理论上，所有的JS框架都大有可为：干净的实现,快速的代码设计,完美的执行。但是当你压力测试时Javascript会怎样？当你丢进6MB的代码时会怎样？在这次演讲中，我们会探究React在高压环境下如何表现，以及它如何帮助我们的团队在大规模时构建安全代码。"

Enter Flux

Controlling our components' data

Immutable Data

Controller

view controller

Reads models, sends UI state to components

Component

view

Receives state, renders UI

Mutable Data

Flux Store

data store

Defines and mutates models

learn - innovate - share

London Ajax User Group

Going big with React

Areeb Malik

08/07/2014

skillsmatter.com
@skillsmatter

CodeWinds

Pete Hunt 与 Jeff Barczewski 在 CodeWinds Episode 4 上关于 React 的谈话.



02:08 - 什么是React，为什么我们用它？

03:08 - ClojureScript 和 React 的共生关系

04:54 - React 的历史以及为什么它被创造

09:43 - 用React更新Web页面，而不绑定数据

13:11 - 用虚拟DOM来改变浏览器DOM

13:57 - 用React编程，绘制目标HTML，canvas和其他

16:45 - 和设计师一起工作，对比于Ember 和 AngularJS

21:45 - JSX编译器桥接HTML和 React javascript

23:50 - React的自动绑定JSX以及浏览器内工具

24:50 - 用React工作的提示和技巧，入门

27:17 - 在服务器端用Node.js渲染HTML。后端渲染

29:20 - React在Facebook通过优胜劣汰进化

30:15 - 用web sockets，在服务器端和客户端持有状态的想法持有

32:05 - 多用户React - 用 Firebase 分布式共享可变状态

33:03 - 用状态转换，事件重放来更好的调式React

34:08 - 来自Web组件的不同之处

34:25 - 使用React的著名公司

35:16 - 一个React的后端插件可以用来创建PDF吗？

36:30 - React的未来，下一步是什么？

39:38 - 贡献和获得帮助

[Read the episode notes](#)

JavaScript Jabber

[Pete Hunt](#) 和 [Jordan Walke](#) 在 JavaScript Jabber 73 上关于React的谈话.

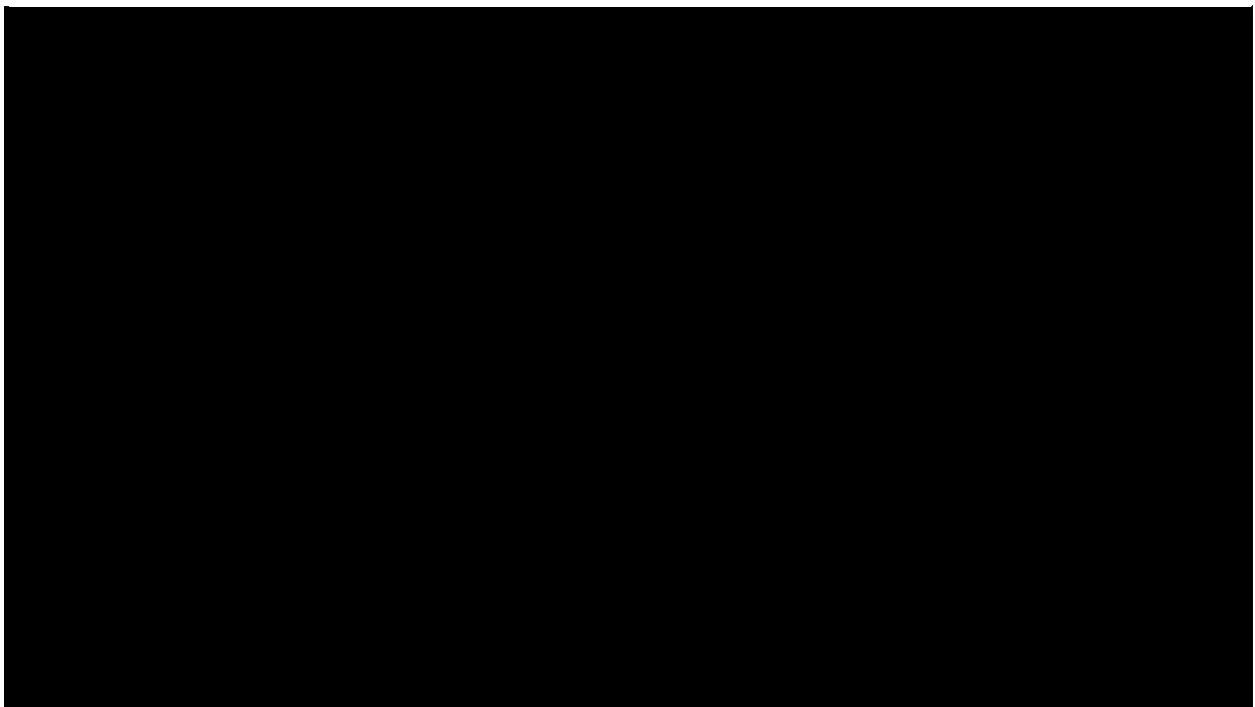


01:34 – Pete Hunt 介绍
02:45 – Jordan Walke 介绍
04:15 – React
06:38 – 60 帧每秒
09:34 – 数据绑定
12:31 – 性能
17:39 – Diffing 算法
19:36 – DOM 操纵

23:06 – 支持 node.js
24:03 – rendr
26:02 – JSX
30:31 – requestAnimationFrame
34:15 – React 和应用
38:12 – React 用户 Khan Academy
39:53 – 使其工作

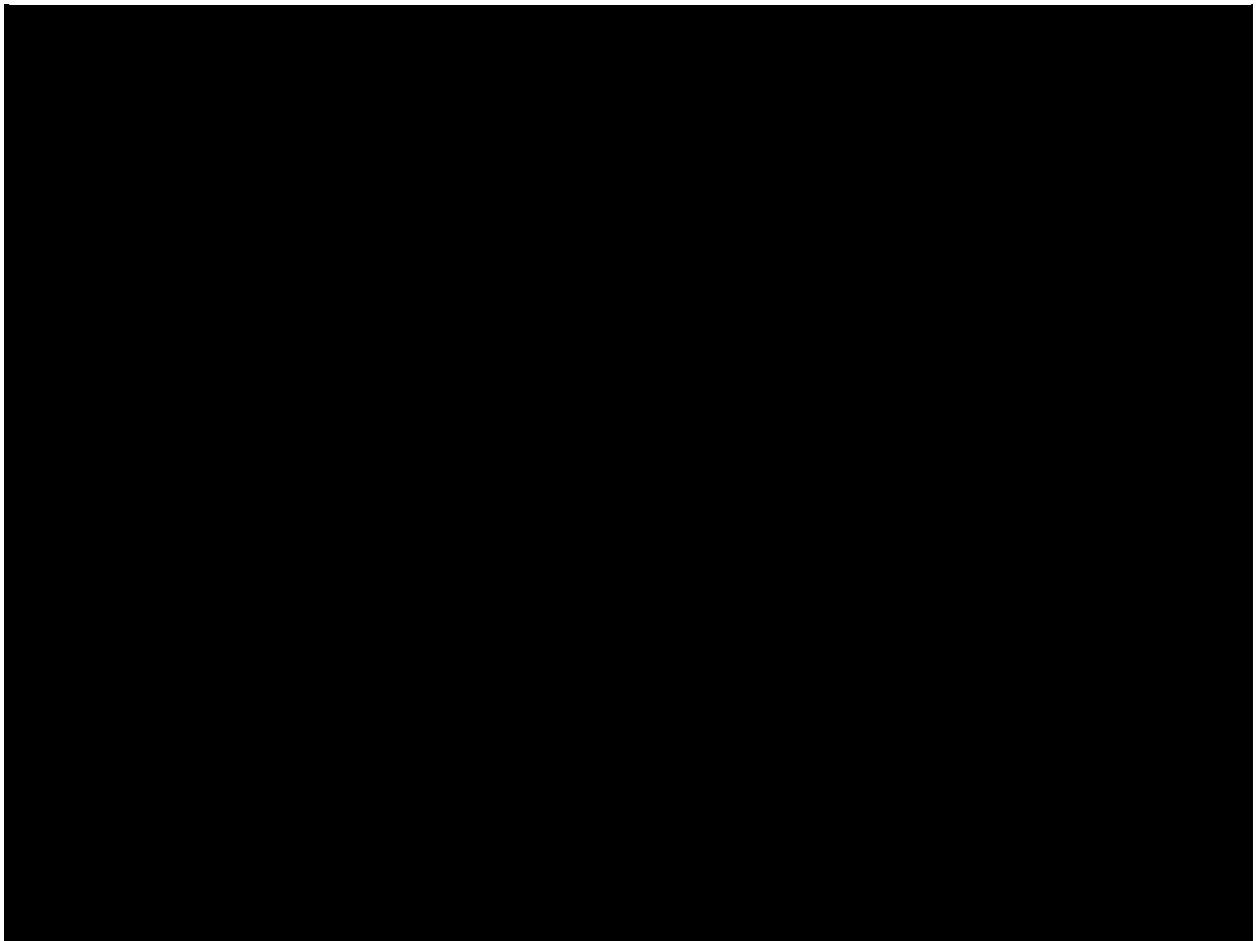
[Read the full transcript](#)

Introduction to React.js - Facebook Seattle



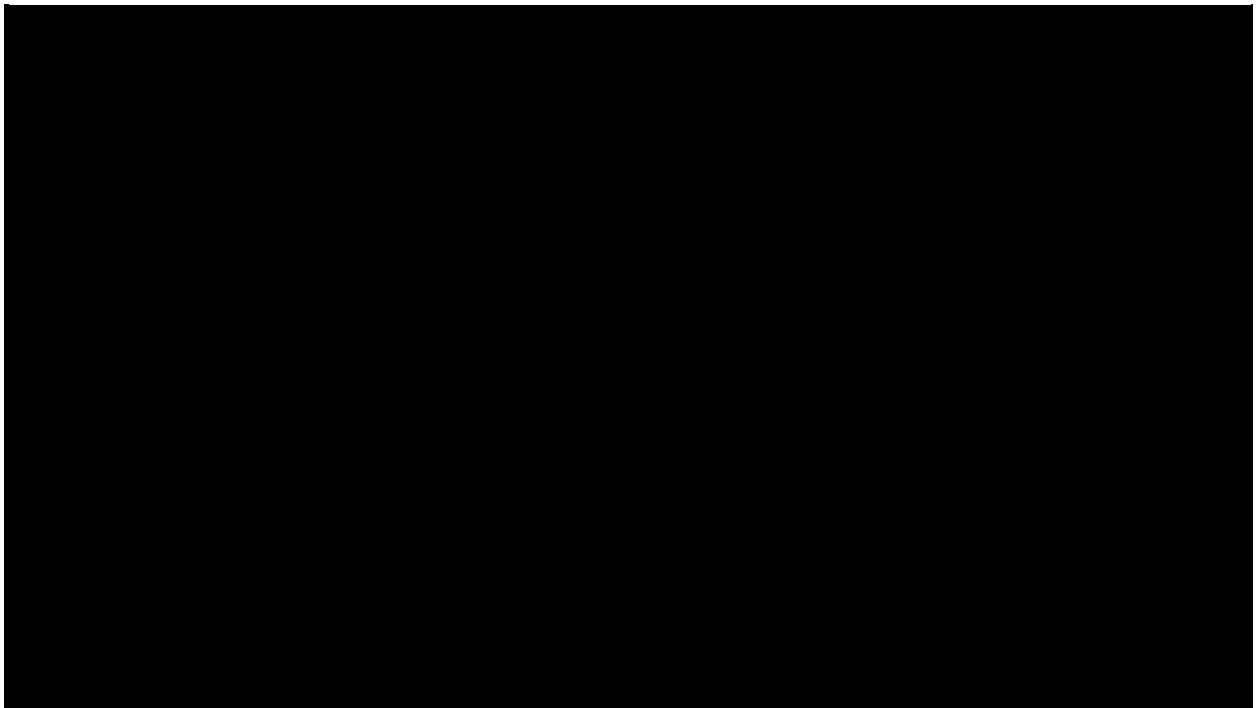
由 [Tom Occhino](#) 和 [Jordan Walke](#)

Backbone + React + Middleman Screencast



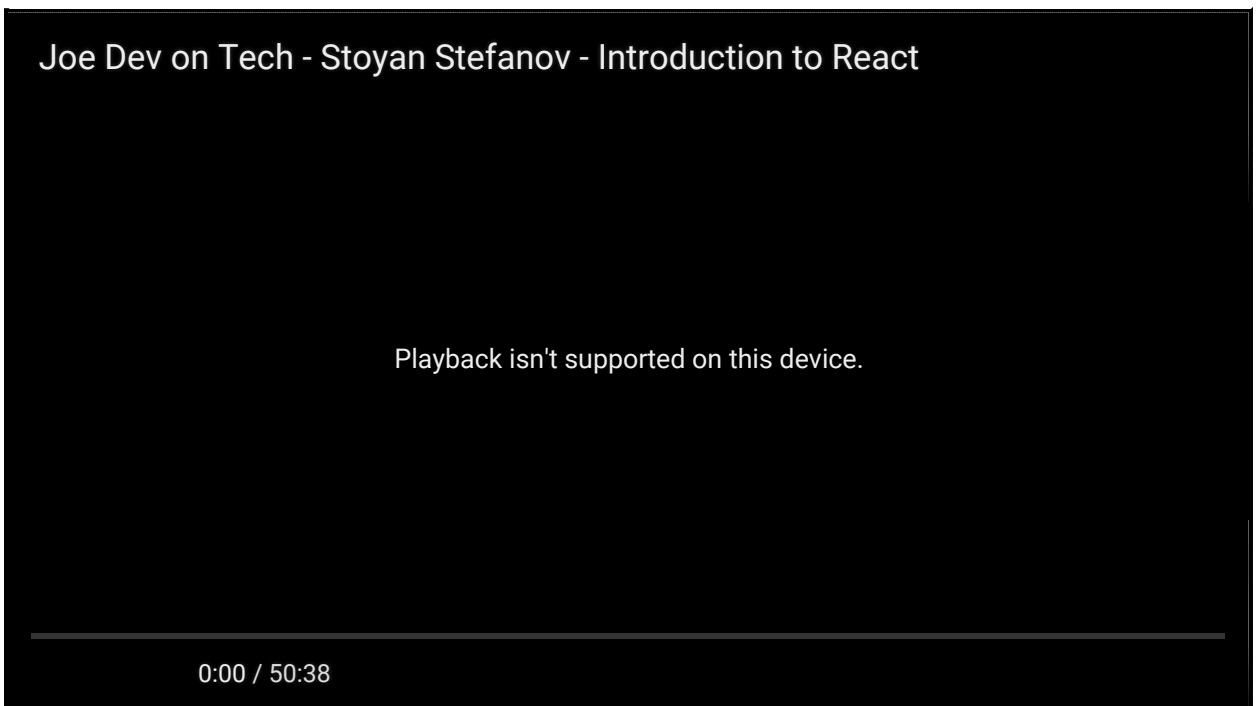
Backbone 是一个在用React实现 REST API 接口的极好方法。这个屏博展示了用 [Backbone-React-Component](#)如何整合两者. Middleman 是在本例中使用的框架但很容易被替换成其他框架。对此可支持的template可以在[这里](#) 找到. -- [Open Minded Innovations](#)

Developing User Interfaces With React - Super VanJS



来自 [Steven Luscher](#)

Introduction to React - LAWebSpeed meetup



来自 [Stoyan Stefanov](#)

React, or how to make life simpler - FrontEnd Dev Conf '14

React, или как начать жить проще, Александр Соловьев

Playback isn't supported on this device.

0:00 / 43:28

俄语 by [Alexander Solovyov](#)

"Functional DOM programming" - Meteor DevShop 11

Pete Hunt: High performance functional programming with React and ...

Playback isn't supported on this device.

0:00 / 30:54

"Rethinking Web App Development at Facebook" - Facebook F8 Conference 2014

Hacker Way: Rethinking Web App Development at Facebook

Playback isn't supported on this device.

0:00 / 44:36

React and Flux: Building Applications with a Unidirectional Data Flow - Forward JS 2014

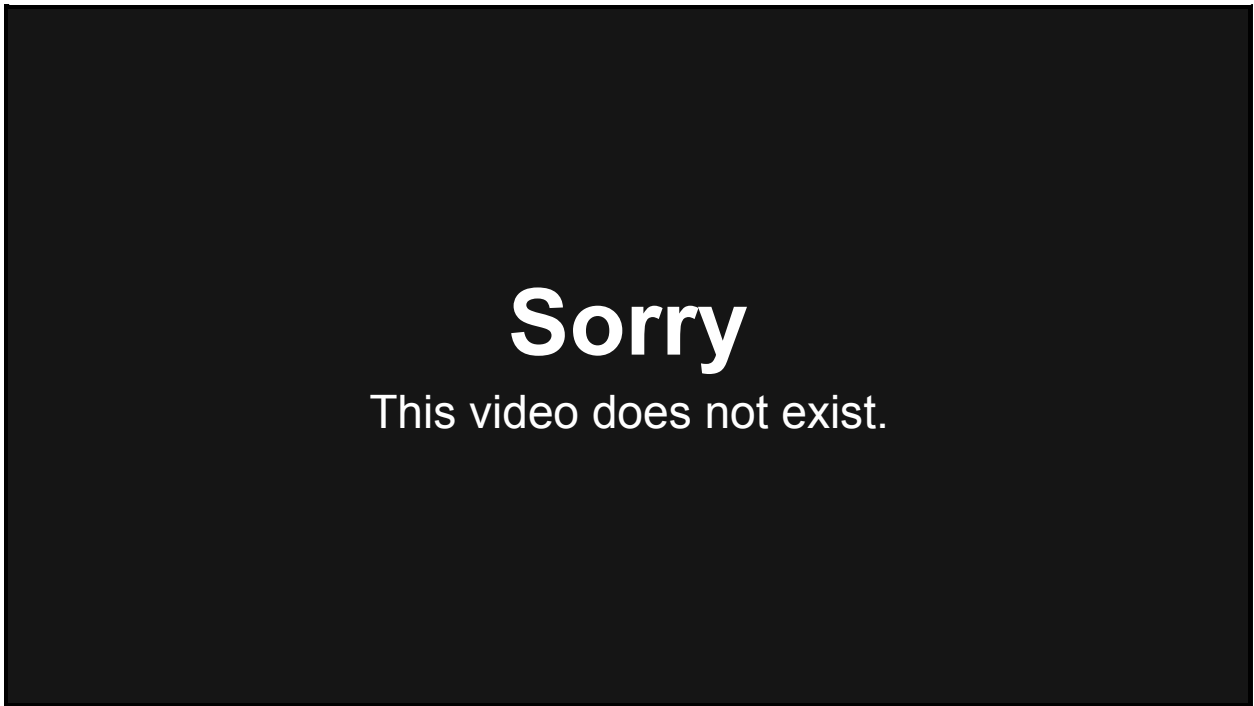
React and Flux: Building Applications with a Unidirectional Data Flow

Playback isn't supported on this device.

0:00 / 36:28

Facebook 工程师 [Bill Fisher](#) 和 [Jing Chen](#) 谈论 Flux 和 React, 以及如何使用单向数据流的程序架构清理他们的代码。

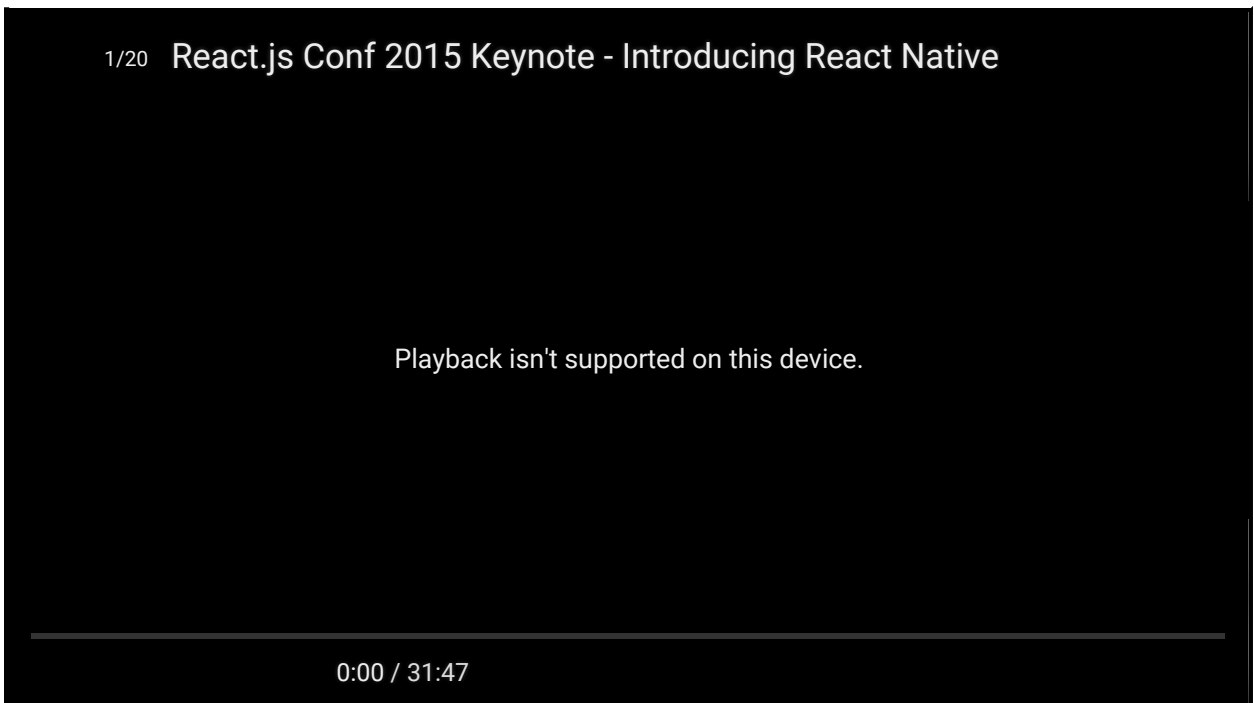
Server-Side Rendering of Isomorphic Apps at SoundCloud



来自 [Andres Suarez](#) 的演练，关于 [SoundCloud](#) 如何使用 React 和 Flux 在服务器端渲染。

[幻灯片和示例代码](#)

Introducing React Native (+Playlist) - React.js Conf 2015



[Tom Occhino](#) 回顾了React的过去和现在，在2015年。梳理了下一步要做什么。

补充工具

本页被移到了 [GitHub wiki](#)。

示例

本页被移到了 [GitHub wiki](#)。

手册

- [为什么使用 React?](#)
- [显示数据](#)
- [动态交互式用户界面](#)
- [复合组件](#)
- [可复用组件](#)
- [传递 Props](#)
- [表单组件](#)
- [与浏览器协作](#)
- [工具集成](#)
- [插件](#)
- [Advanced Performance](#)
- [Context](#)

为什么使用 **React**?

React 是一个 Facebook 和 Instagram 用来创建用户界面的 JavaScript 库。很多人选择将 React 认为是 **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。

简单

仅仅只要表达出你的应用程序在任一个时间点应该呈现的样子，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>
    <meta charset="UTF-8" />
    <title>Hello React</title>
    <script src="https://fb.me/react-0.14.7.js"></script>
    <script src="https://fb.me/react-dom-0.14.7.js"></script>
    <script src="https://cdnjs.cloudflare.com/ajax/libs/babel-core/5.8.23/browser.min.js"></script>
  </head>
  <body>
    <div id="example"></div>
    <script type="text/babel">

      // ** 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() {
  ReactDOM.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：

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

通过 **JSX** 这就变成了

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

我们发现这会使搭建 **React** 应用更加简单，设计师也偏向用这用语法，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[Babel REPL](#)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。

[[Babel 公开了一些使用 JSX 的方式](#)],从命令行工具到 **Ruby on Rails** 集成。选择一个对你来说最合适的工具。

没有 **JSX** 的 **React**

JSX完全是可选的；你无需在 **React** 中必须使用 **JSX**。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 child1 = React.createElement('li', null, 'First Text Content');
var child2 = React.createElement('li', null, 'Second Text Content');
var root = React.createElement('ul', { className: 'my-list' }, child1, child2);
ReactDOM.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' });  
ReactDOM.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')  
);
```

深入 JSX

JSX 是一个看起来很像 XML 的 JavaScript 语法扩展。React 可以用来做简单的 JSX 句法转换。

为什么要用 JSX？

你不需要为了 React 使用 JSX，可以直接使用原生 JS。但是，我们建议使用 JSX 是因为它能精确，也是常用的定义包含属性的树状结构的语法。

它对于非专职开发者比如设计师也比较熟悉。

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

它没有修改 JavaScript 语义。

HTML 标签对比 React 组件

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

要渲染 HTML 标签，只需在 JSX 里使用小写字母的标签名。

```
var myDivElement = <div className="foo" />;
ReactDOM.render(myDivElement, document.getElementById('example'));
```

要渲染 React 组件，只需创建一个大写字母开头的本地变量。

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

React 的 JSX 使用大、小写的约定来区分本地组件的类和 HTML 标签。

注意:

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

转换（Transform）

JSX 把类 XML 的语法转成原生 JavaScript，XML 元素、属性和子节点被转换成

`React.createElement` 的参数。

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

注意，要想使用 `<Nav />`，`Nav` 变量一定要在作用区间内。

JSX 也支持使用 XML 语法定义子结点：

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

当显示名称没有定义时，JSX 会根据变量赋值来推断类的 [显示名称](#)：

```
// 输入 (JSX):
var Nav = React.createClass({ });
// 输出 (JS):
var Nav = React.createClass({displayName: "Nav", });
```

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

如果你要使用 JSX，这篇 [新手入门](#) 教程来教你如何搭建环境。

注意：

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

命名组件（Namespaced Components）

如果你正在构建一个有很多子组件的组件，比如表单，你也许会最终得到许多的变量声明。

```
// 尴尬的变量声明块
var Form = MyFormComponent;
var FormRow = Form.Row;
var FormLabel = Form.Label;
var FormInput = Form.Input;

var App = (
  <Form>
    <FormRow>
      <FormLabel />
      <FormInput />
    </FormRow>
  </Form>
);
```

为了使其更简单和容易，命名组件令你使用包含其他组件作为属性的单一的组件。

```
var Form = MyFormComponent;

var App = (
  <Form>
    <Form.Row>
      <Form.Label />
      <Form.Input />
    </Form.Row>
  </Form>
);
```

要做到这一点，你只需要把你的"子组件"创建为主组件的属性。

```
var MyFormComponent = React.createClass({ ... });

MyFormComponent.Row = React.createClass({ ... });
MyFormComponent.Label = React.createClass({ ... });
MyFormComponent.Input = React.createClass({ ... });
```

当编译你的代码时，JSX会恰当的进行处理。

```
var App = (  
  React.createElement(Form, null,  
    React.createElement(Form.Row, null,  
      React.createElement(Form.Label, null),  
      React.createElement(Form.Input, null)  
    )  
  )  
);
```

注意:

此特性在 **v0.11** 及以上可用。

JavaScript 表达式

属性表达式

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

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

Boolean 属性

省略一个属性的值会导致JSX把它当做 `true`。要传值 `false` 必须使用属性表达式。这常出现于使用HTML表单元素，含有属性如 `disabled`，`required`，`checked` 和 `readOnly`。

```
// 在JSX中，对于禁用按钮这二者是相同的。  
<input type="button" disabled />;  
<input type="button" disabled={true} />;  
  
// 在JSX中，对于不禁用按钮这二者是相同的。  
<input type="button" />;  
<input type="button" disabled={false} />;
```

子节点表达式

同样地，JavaScript 表达式可用于描述子结点：

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

注释

JSX 里添加注释很容易；它们只是 JS 表达式而已。你仅仅需要小心的是当你在一个标签的子节点块时，要用 `{}` 包围要注释的部分。

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

注意：

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

JSX 展开属性

如果你事先知道组件需要的全部 Props（属性），JSX 很容易地这样写：

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

修改 Props 是不好的，明白吗

如果你不知道要设置哪些 Props，那么现在最好不要设置它：

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

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

Props 应该被认为是不可变的。在别处修改 props 对象可能会导致预料之外的结果，所以原则上这将是一个冻结的对象。

展开属性（Spread Attributes）

现在你可以使用 JSX 的新特性 - 展开属性：

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

传入对象的属性会被复制到组件内。

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

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

这个奇怪的 `...` 标记是什么？

这个 `...` 操作符（增强的操作符）已经被 [ES6 数组](#) 支持。相关的还有 ECMAScript 规范草案中的 [Object 剩余和展开属性（Rest and Spread Properties）](#)。我们利用了这些还在制定中标准中已经被支持的特性来使 JSX 拥有更优雅的语法。

JSX 陷阱

JSX 与 HTML 非常相似，但是有些关键区别要注意。

注意:

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

HTML 实体

HTML 实体可以插入到 JSX 的文本中。

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

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

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

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

```
<div>{'First · Second'}</div>
```

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

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

可以在数组里混合使用字符串和 JSX 元素。

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

万不得已，可以直接[插入原始HTML](#)。

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

自定义 HTML 属性

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

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

然而，在自定义元素中任意的属性都是被支持的（那些在 `tag` 名里带有连接符或者 `is="..."`
属性的）

```
<x-my-component custom-attribute="foo" />
```

以 `aria-` 开头的 [网络无障碍](#) 属性可以正常使用。

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

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>
    );
  }
});

ReactDOM.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。在 React，所有方法被自动绑定到了它的组件实例上（除非使用 ES6 的 `class` 符号）。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`）组件并把它的状态通过 `props` 传给子级。这个有状态的组件封装了所有用户的交互逻辑，而这些无状态组件则负责声明式地渲染数据。

哪些应该作为 State？

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

哪些 不应该 作为 **State**？

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

- 计算所得数据：不要担心根据 `state` 来预先计算数据 —— 把所有的计算都放到 `render()` 里更容易保证用户界面和数据的一致性。例如，在 `state` 里有一个数组 (`listItems`)，我们要把数组长度渲染成字符串，直接在 `render()` 里使用 `this.state.listItems.length + ' list items'` 比把它放到 `state` 里好的多。
- **React** 组件：在 `render()` 里使用当前 `props` 和 `state` 来创建它。
- 基于 **props** 的重复数据：尽可能使用 `props` 来作为实际状态的源。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 `props` 可能因为父组件重绘的结果而变化。

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={'https://graph.facebook.com/' + this.props.username + '/picture'} />
    );
  }
});

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

ReactDOM.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基本不变量。

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 `ReactFragment` 对象 来做有 `key` 的子级。详见 [Keyed Fragments](#)

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要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(or fragment) 。
    optionalNode: React.PropTypes.node,

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

    // 你同样可以断言一个 prop 是一个类的实例。
    // 用 JS 的 instanceof 操作符声明 prop 为类的实例。
    optionalMessage: React.PropTypes.instanceOf(Message),

    // 你可以用 enum 的方式
    // 确保你的 prop 被限定为指定值。
    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
}),

// 你可以在任意东西后面加上 `isRequired`
// 来确保 如果 prop 没有提供 就会显示一个警告。
requiredFunc: React.PropTypes.func.isRequired,

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

// 你可以自定义一个验证器。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，
// 因为这在 `oneOfType` 里不起作用。
customProp: function(props, propName, componentName) {
  if (!/matchme/.test(props[propName])) {
    return new Error('Validation failed!');
  }
},
/* ... */
});
```

Single Child

用 `React.PropTypes.element` 你可以指定仅有一个子级能被传送给组件

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

  render: function() {
    return (
      <div>
        {this.props.children} // 这里必须是一个元素否则就会警告
      </div>
    );
  }
});
```

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：捷径

有一些常用的 React 组件只是对 HTML 做简单扩展。通常，你想复制任何传进你的组件的 HTML 属性到底层的 HTML 元素上。为了减少输入，你可以用 **JSX spread** 语法来完成：

```
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>;
  }
});

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

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.forEach(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>
    );
  }
});

ReactDOM.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。

ES6 Classes

你也可以以一个简单的JavaScript 类来定义你的React classes。使用ES6 class的例子:

```
class HelloMessage extends React.Component {
  render() {
    return <div>Hello {this.props.name}</div>;
  }
}
ReactDOM.render(<HelloMessage name="Sebastian" />, mountNode);
```

API近似于 `React.createClass` 除了 `getInitialState`。你应该在构造函数里设置你的 `state`，而不是提供一个单独的 `getInitialState` 方法。

另一个不同是 `propTypes` 和 `defaultProps` 在构造函数而不是class body里被定义为属性。

```
export class Counter extends React.Component {
  constructor(props) {
    super(props);
    this.state = {count: props.initialCount};
  }
  tick() {
    this.setState({count: this.state.count + 1});
  }
  render() {
    return (
      <div onClick={this.tick.bind(this)}>
        Clicks: {this.state.count}
      </div>
    );
  }
}
Counter.propTypes = { initialCount: React.PropTypes.number };
Counter.defaultProps = { initialCount: 0 };
```

无自动绑定

方法遵循正式的ES6 class的语义，意味着它们不会自动绑定 `this` 到实例上。你必须显示的使用 `.bind(this)` or [箭头函数](#) `=>`。

没有 Mixins

不幸的是ES6的发布没有任何mixin的支持。因此，当你在ES6 classes下使用React时不支持mixins。作为替代，我们正在努力使它更容易支持这些用例不依靠mixins。

无状态函数

你也可以用 JavaScript 函数来定义你的 React 类。例如使用无状态函数语法：

```
function HelloMessage(props) {  
  return <div>Hello {props.name}</div>;  
}  
ReactDOM.render(<HelloMessage name="Sebastian" />, mountNode);
```

或者使用新的ES6箭头函数：

```
const HelloMessage = (props) => <div>Hello {props.name}</div>;  
ReactDOM.render(<HelloMessage name="Sebastian" />, mountNode);
```

这个简化的组件API旨在用于那些纯函数态的组件。这些组件必须没有保持任何内部状态，没有备份实例，也没有组件生命周期方法。他们纯粹的函数式的转化他们的输入，没有引用。然而，你仍然可以以设置为函数的`properties`的方式来指定 `.propTypes` 和 `.defaultProps`，就像你在ES6类里设置他们那样。

注意：

因为无状态函数没有备份实例，你不能附加一个引用到一个无状态函数组件。通常这不是问题，因为无状态函数不提供一个命令式的API。没有命令式的API，你就没有任何需要实例来做的事。然而，如果用户想查找无状态函数组件的DOM节点，他们必须把这个组件包装在一个有状态组件里（比如，ES6 类组件）并且连接一个引用到有状态的包装组件。

在理想世界里，你的大多数组件都应该都是无状态函数式的，因为这些无状态组件可以在React核心里经过一个快速的代码路径。如果可能，这是推荐的模式。

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：

```
<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 的 `_extend`。

```
React.createElement(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 试验性的 ECMAScript 语法。

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。

```
function FancyCheckbox(props) {
  var fancyClass = props.checked ? 'FancyChecked' : 'FancyUnchecked';
  return (
    <div className={fancyClass} onClick={props.onClick}>
      {props.children}
    </div>
  );
}
ReactDOM.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

注意:

在下面的例子中，`--harmony` 标志是必须的因为这个语法是ES7的实验性语法。如果用浏览器中的JSX转换器，以 `<script type="text/jsx;harmony=true">` 简单的打开你脚本就行了。详见[Rest and Spread Properties ...](#)

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 } = props;
```

这样能确保把所有 props 传下去，除了 那些已经被使用了的。

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

```
function FancyCheckbox(props) {
  var fancyClass = props.checked ? 'FancyChecked' : 'FancyUnchecked';
  // 反模式：`checked` 会被传到里面的组件里
  return (
    <div {...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。

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

注意:

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

剩余属性和展开属性 ...

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

这是 [ECMAScript 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:

要用 Babel 6 转换 rest 和 spread 属性，你需要安装 `es2015` preset，`transform-object-rest-spread` 插件并在 `.babelrc` 里配置他们。

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。

```
function FancyCheckbox(props) {  
  var checked = props.checked;  
  var other = _.omit(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。

关于 `<form>` 事件详情请查看 [表单事件](#)。

交互属性

表单组件支持几个受用户交互影响的属性：

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

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

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

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

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

注意：

对于 `<input>` and `<textarea>`，`onChange` 替代 — 一般应该用来替代 — the DOM's 内建的 `oninput` 事件处理。

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。

复选框与单选按钮的潜在问题

当心，在力图标准化复选框与单选按钮的变换处理中，`React` 使用 `click` 事件代替 `change` 事件。在大多数情况下它们表现的如同预期，除了在 `change handler` 中调用 `preventDefault`。 `preventDefault` 阻止了浏览器视觉上更新输入，即使 `checked` 被触发。变通的方式是要么移除 `preventDefault` 的调用，要么把 `checked` 的触发放在一个 `setTimeout` 里。

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。

默认值

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!" />;  
}
```

这个例子会像上面的 不受限组件 例子一样运行。

同样的，`<input>` 支持 `defaultChecked` 、 `<select>` 支持 `defaultValue` 。

注意：

`defaultValue` 和 `defaultChecked` props 只能在内部渲染时被使用。如果你需要在随后的渲染更新值，你需要使用 [受限组件](#)。

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 为了更方便的控制组件，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']}>`。

与浏览器协作

React提供了强大的抽象机制使你在大多数情况下免于直接接触DOM，但有时你仅仅只需要访问底层API，也许是为了与第三方库或者已有的代码协作。

虚拟DOM

React非常快速是因为它从不直接操作DOM。React维持了一个快速的内存中的DOM表示。`render()`方法实际上返回一个对DOM的描述，然后React能根据内存中的“描述”来比较此“描述”以计算出最快速的方法更新浏览器。

此外，React实现了一个完备的合成事件(synthetic event)系统，以使得所有的事件对象都被保证符合W3C细则，而不论各个浏览器的怪癖，并且所有事件跨浏览器地一致并高效的冒泡(bubbles)，你甚至能在IE8里使用一些HTML5事件！

大多数时间你应该和React的“伪造浏览器”呆在一起，因为它更高性能并且容易推理。然而，有时你只需要访问底层API，或许是为了与第三方库比如一个jQuery插件协作。React为你提供了安全仓口来直接使用底层API。

Refs 和 findDOMNode()

为了与浏览器互动，你需要一个指向DOM node的引用。你可以连接一个 `ref` 到任何的元素，这允许你引用组件的 **backing instance**。它很有用，如果你需要在组件上调用命令式函数，或者想访问底层的DOM节点。要了解很多关于 refs，包括更有效使用他们的方法，请查看我们的 [关于Refs的更多内容](#) 文档。

组件的生命周期

组件的生命周期有三个主要部分：

- **挂载**: 组件被注入DOM。
- **更新**: 组件被重新渲染来决定DOM是否应被更新。
- **卸载**: 组件从DOM中被移除。

React提供生命周期方法，以便你可以指定钩挂到这个过程中。我们提供了 **will** 方法，该方法在某事发生前被调用，**did**方法，在某事发生后被调用。

挂载

- `getInitialState(): object` 在组件挂载前被调用。有状态组件(Stateful components) 应该实现此函数并返回初始state的数据。
- `componentWillMount()` 在挂载发生前被立即调用。
- `componentDidMount()` 在挂载发生后被立即调用。需要DOM node的初始化应该放在这里。

更新

- `componentWillReceiveProps(object nextProps)` 当挂载的组件接收到新的props时被调用。此方法应该被用于比较 `this.props` 和 `nextProps` 以用于使用 `this.setState()` 执行状态转换。
- `shouldComponentUpdate(object nextProps, object nextState): boolean` 当组件决定任何改变是否要更新到DOM时被调用。作为一个优化实现比较 `this.props` 和 `nextProps` 、 `this.state` 和 `nextState` ，如果React应该跳过更新，返回 `false` 。
- `componentWillUpdate(object nextProps, object nextState)` 在更新发生前被立即调用。你不能在此调用 `this.setState()` 。
- `componentDidUpdate(object prevProps, object prevState)` 在更新发生后被立即调用。

卸载

- `componentWillUnmount()` 在组件被卸载和摧毁后被立即调用。清理应该放在这里。

已挂载的方法

Mounted 复合组件同样支持以下方法:

- `component.forceUpdate()` 可以在任何已挂载的组件上调用，在你知道 某些深处的组件状态被未使用 `this.setState()` 改变了时。

浏览器支持和填充物(polyfills)

在 Facebook，我们支持老浏览器，包括IE8。我们由来已久的有适当的填充物(polyfills)来让我们写前瞻性的js。这意味着我们在代码库中没有一堆散落在各处的技巧(hacks)并且我们依然能期望我们的代码"可行(just work)"。例如，我可以只写 `Date.now()`，而不是额外看到 `+new Date()`。既然开源的React和我们内部使用的一样，我们也应用了这种使用前瞻性js的哲学。

除了这种哲学外，我们也采用了这样的立场，我们，作为一个JS库的作者，不应该把polyfills作为我们库的一部分。如果所有的库这样做，就有很大的机会发送同样的polyfill多次，这可能是一个相当大的无用代码。如果你的产品需要支援老的浏览器，你很有可能已经在使用某些东西比如[es5-shim](#)。

需要用来支持旧浏览器的Polyfills

来自 [kriskowal's es5-shim](#) 的 `es5-shim.js` 提供了如下React需要的东西：

- `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`

同样来自 [kriskowal's es5-shim](#) 的 `es5-sham.js`，提供了如下React需要的东西：

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

非最小化的React build需要如下，来自 [paulmillr's console-polyfill](#).

- `console.*`

当在IE8里使用HTML5元素，包括 `<section>`，`<article>`，`<nav>`，`<header>`，和 `<footer>`，同样必须包含[html5shiv](#) 或者类似的脚本。

跨浏览器问题

尽管React在抽象浏览器不同时做的相当好，但一些浏览器被限制或者表现出怪异的行为，我们没能找到变通的方案解决。

IE8的onScroll事件

在IE8 `onScroll` 事件不冒泡，并且IE8没有定义事件捕获阶段handlers的API，意味React这没有办法去监听这些事件。目前这个事件的handler在IE8中是被忽略的。

参见 [onScroll doesn't work in IE8](#) GitHub问题 来获得更多信息.

关于Refs的更多内容

在建立你的组件以后，你也许发现你想“接触”并且调用从 `render()` 返回的组件实例上的方法。大部分情况下，这是不必要的因为响应式的数据流总是确保最近的 `props` 被送到每一个从 `render()` 输出的子级。然而，有一些情况下它仍旧是必要或者有益的，所以 **React** 提供了一个被称为 `refs` 的安全舱口。这些 `refs`（引用）在你需要时特别有用如：查找被组件绘制的 DOM 标记（例如，绝对定位它），使用 **React** 组件在一个大的非**React**组件里，或者转换你已有的代码库到**React**。

让我们来看看怎样取得一个`ref`，然后深入完整的例子。

从 **ReactDOM.render** 返回的 `ref`

不要被 你在你的组件（它返回一个虚拟的DOM元素）里定义的 `render()` 迷惑了，**ReactDOM.render()** 会返回一个对你的组件的 **backing instance** 的引用（或者 `null` for **stateless components**）。

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

记住，不管怎样，**JSX**不会返回一个组件的实例！它只是一个 **ReactElement**：一个轻量级的表达，告诉**React**被挂载的组件应该长什么样。

```
var myComponentElement = <MyComponent />; // 这只是一个 ReactElement.

// 省略一些代码 ...

var myComponentInstance = ReactDOM.render(myComponentElement, myContainer);
myComponentInstance.doSomething();
```

注意：

这应该用在顶层上。在组件内部，让你的 `props` 和 `state` 来处理 and 子组件的通信，或者使用其他获取`ref`的方法（`string attribute or callbacks`）。

`ref` Callback 属性

React支持一种非常特殊的属性，你可以附加到任何的组件上。`ref` 属性可以是一个回调函数，这个回调函数会在组件被挂载后立即执行。被引用的组件会被作为参数传递，回调函数可以用立即使用这个组件，或者保存引用以后使用（或者二者皆是）。

简单的说就是添加一个 `ref` 属性到任何从 `render` 返回的东西上：

```
render: function() {
  return (
    <TextInput
      ref={function(input) {
        if (input != null) {
          input.focus();
        }
      }} />
  );
},
```

或者使用ES6的箭头函数：

```
render: function() {
  return <TextInput ref={(c) => this._input = c} />;
},
componentDidMount: function() {
  this._input.focus();
},
```

当连接一个`ref`到一个DOM组件如 `<div />`，你取回DOM节点；当连接一个`ref`到一个复合组件如 `<TextInput />`，你会得到React类的实例。在后一种情况下，你可以调用任何那个组件的类暴露的方法。

注意当被引用的组件卸载和每当`ref`变动，旧的`ref`将会被以 `null` 做参数调用。这阻止了在实例被保存的情况下的内存泄露，如第一个例子。注意当像在这里的例子，使用内联函数表达式写`refs`，React在每次更新都看到不同的函数对象，`ref`将会被以 `null` 立即调用在它被以组件实例调用前。

ref String 属性

React同样支持使用一个字符串（代替回调函数）在任意组件上作为一个 `ref prop`，尽管这个方法在这点上主要是遗留物。

1. 赋值 `ref` 属性为任何从 `render` 返回的东西，比如：

```
<input ref="myInput" />
```

2. 在其他一些代码中（典型的事件处理代码），通过 `this.refs` 访问 支持实例(backing instance)，如：

```
var input = this.refs.myInput;
var inputValue = input.value;
var inputRect = input.getBoundingClientRect();
```

完整的示例

为了获取一个React组件的引用，你既可以使用 `this` 来获取当前的React组件，也可以使用一个 `ref` 来获取一个你拥有的组件的引用。他们像这样工作：

```
var MyComponent = React.createClass({
  handleClick: function() {
    // Explicitly focus the text input using the raw DOM API.
    this.myTextInput.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={(ref) => this.myTextInput = ref} />
        <input
          type="button"
          value="Focus the text input"
          onClick={this.handleClick}
        />
      </div>
    );
  }
});

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

在这个例子中，我们获得一个对 **text input backing instance** 的引用 并且当按钮被点击时我们调用 `focus()`。

对于复合组件，引用会指向一个组件类的实例所以你可以调用那个类定义的任何方法。如果你需要访问那个组件的底层的DOM节点，你可以使用 `ReactDOM.findDOMNode` 作为一个 **安全舱口** 但是我们不推荐这样，因为它打破了封装，在大多数情况下都有一个清晰的方法来以React模式构建你的代码。

总结

Refs是一种很好的发送消息给特定子实例(通过流式的Reactive props 和 state 来做会不方便)的方式。它们应该, 不论怎样, 不是你数据流通你的应用的首选。默认情况下, 使用响应式数据流, 并为本身不是reactive的用例保存 refs。

优点:

- 你可以在你的组件类里定义任何的公开方法 (比如在一个Typeahead的重置方法) 然后通过refs调用那些公开方法 (比如 `this.refs.myTypeahead.reset()`)。在大多数情况下, 使用内建的React数据流更清晰, 而不是使用强制的ref。
- 实行DOM测量几乎总是需要接触到 "原生" 组件比如 `<input />` 并且通过 ref 访问它的底层DOM节点。Refs 是唯一一个可靠的完成这件事的实际方式。
- Refs 是为你自动管理的! 如果子级被销毁了, 它的ref也同样为你销毁了。这里不用担心内存 (除非你做了一些疯狂的事情来自己保持一份引用)。

注意事项:

- 绝不在任何组件的 render 方法中访问 refs - 或者当任何组件的render方法还在调用栈上的任何地方运行时。
- 如果你想要保留Google Closure Compiler advanced-mode crushing resilience, 务必不要以属性的方式访问指明为字符串的属性。这意味你必须用 `this.refs['myRefString']` 访问, 如果你的ref被定义为 `ref="myRefString"`。
- 如果你没有用React写过数个程序, 你的第一反应通常是打算试着用refs来在你的应用里"让事情发生"。如果是这样, 花一些时间并且更精密的思考 state 应该属于组件层级的哪个位置。常常, 这会变得清晰: 正确的"拥有"那个属性的地方应该在层级的更高层上。把state放在那里 往往消除了任何使用 refs 来"让事情发生"的渴望 - 作为替代, 数据流通常将完成你的目标。
- Refs 不能连接到一个 [stateless function \(无状态函数\)](#), 因为这些组件没有支持实例。你总是可以包装一个无状态组件在一个标准复合组件里并且连接一个ref到这个复合组件。

工具集成

每个项目使用一个不同的系统来建立和部署JavaScript.我们努力使React尽可能环境无关。

React

CDN-hosted React

我们提供了CDN-hosted版本的React[在我们的下载页面](#).这些预构建的文件使用UMD模块格式。将他们放进一个简单的 `<script>` 标签将会注入一个 `React` 全局变量到你的环境里。这也同样在CommonJS 和 AMD 环境里开箱即用。

使用 master

我们[在我们的 GitHub repository](#)有从 `master` 构建的说明。我们在 `build/modules` 下构建了一个CommonJS模块的树，你可以把它放到任何支持CommonJS的环境或者打包工具里。

JSX

浏览器中的JSX转化

如果你喜欢使用JSX，Babel提供了一个被称为browser.js的[开发用的浏览器中的 ES6 和 JSX 转换器](#)，它可以从 `babel-core` npm release 或者[CDNJS](#) 中 include。Include `<script type="text/babel">` 标记来使用 JSX 转换器。

注意:

浏览器中的JSX转换器相当大并且导致额外的本可避免的客户端计算。不要在生产环境中使用 - 见下一节。

投入生产：预编译 JSX

如果你有npm，你可以运行 `npm install -g babel-cli`。Babel 对React v0.12+ 有内建支持。标签被自动转化为它们的等价物 `React.createElement(...)`，`displayName` 被自动推断并添加到所有的`React.createClass` 调用。

这个工具会把使用JSX语法的文件转化为简单的可直接在浏览器运行的JavaScript文件。它同样将为你监视目录并自动转化文件当他们变动时；例如：`babel --watch src/ --out-dir lib/ .`

从 Babel 6 开始，默认不再包含转换。这意味这必须在运行 `babel` 命令时指定选项，或者 `.babelrc` 必须指定选项。附加的捆绑了一大批转化的包(presets)也同样需要被安装.协同 React工作最常用的是 `es2015` 和 `react` presets.更多关于 Babel 变化的信息可以在 [Babel 6 博客发布的信息](#)上找到.

这里是一个要使用ES2015 语法和 React 你该怎样做的例子：

```
npm install babel-preset-es2015 babel-preset-react
babel --presets es2015,react --watch src/ --out-dir lib/
```

默认模式下带有 `.js` 后缀的JSX文件被转化。运行 `babel --help` 获取更多关于如何使用 Babel 的信息。

输出的例子：

```
$ cat test.jsx
```

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

ReactDOM.render(<HelloMessage name="John" />, mountNode);
```

```
$ babel test.jsx
```

```
"use strict";

var HelloMessage = React.createClass({
  displayName: "HelloMessage",

  render: function render() {
    return React.createElement(
      "div",
      null,
      "Hello ",
      this.props.name
    );
  }
});

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

有帮助的开源项目

开源社区已经创建了一些集成JSX到数个编辑器和构建系统的工具。全部列表请见[JSX integrations](#)。

插件

React插件是一系列的用来构建React app的有用模块。这些应该被认为是实验性的 并趋向于比core变动更频繁。

- `TransitionGroup` 和 `CSSTransitionGroup` , 用来处理通常不能简单实现的动画和转换, 比如在组件移除之前。
- `LinkStateMixin` , 简化用户的表单输入数据与组件状态的协调。
- `cloneWithProps` , 创建React组件的浅拷贝并改变它们的props。
- `createFragment` , 创建一组外键的子级。
- `update` , 一个使不可变数据在JavaScript里更易处理的辅助函数。
- `PureRenderMixin` , 一个特定情况下的性能优化器。
- `shallowCompare` , 一个辅助函数, 用来对 props 和 state在组件里 执行浅比较 以决定一个组件是否应该更新。

下面的插件只存在开发版(未压缩)React中：

- `TestUtils` , 用于写测试用例的简单的辅助工具（仅存在于未压缩版本）。
- `Perf` , 用于测量性能并给你提示哪里可以优化。

要获取插件，单独从npm安装他们(例如 `npm install react-addons-pure-render-mixin`). 我们不支持使用插件如果你没有用npm.

动画

React 提供了一个 `ReactTransitionGroup` 插件作为动画的底层API,和一个 `ReactCSSTransitionGroup` 用于轻松实现基础的CSS动画和过渡。

高级 API: `ReactCSSTransitionGroup`

`ReactCSSTransitionGroup` 基于 `ReactTransitionGroup` 是一个当React组件进入或离开DOM时，执行CSS动画和过渡的简单方法。它的灵感来自于杰出的 [ng-animate](#) 库。

入门指南

`ReactCSSTransitionGroup` 是 `ReactTransitions` 的接口。这是一个简单的元素，包裹了所有你感兴趣的动画组件。这里是一个淡入和淡出列表项目的例子。

```

var ReactCSSTransitionGroup = require('react-addons-css-transition-group');

var TodoList = React.createClass({
  getInitialState: function() {
    return {items: ['hello', 'world', 'click', 'me']};
  },
  handleAdd: function() {
    var newItems =
      this.state.items.concat([prompt('Enter some text')]);
    this.setState({items: newItems});
  },
  handleRemove: function(i) {
    var newItems = this.state.items.slice();
    newItems.splice(i, 1);
    this.setState({items: newItems});
  },
  render: function() {
    var items = this.state.items.map(function(item, i) {
      return (
        <div key={item} onClick={this.handleRemove.bind(this, i)}>
          {item}
        </div>
      );
    }).bind(this);
    return (
      <div>
        <button onClick={this.handleAdd}>Add Item</button>
        <ReactCSSTransitionGroup transitionName="example" transitionEnterTimeout={500}
transitionLeaveTimeout={300}>
          {items}
        </ReactCSSTransitionGroup>
      </div>
    );
  }
});

```

注意:

你必须为 `ReactCSSTransitionGroup` 的所有子级提供 `key` 属性，即使只渲染一个项目。这就是React将决定哪一个子级进入，离开，或者停留

在这个组件，当一个新的项目被添加到 `ReactCSSTransitionGroup` ,他将得到 `example-enter` CSS类 并且在下一刻 `example-enter-active` CSS类被添加。这是一个基于 `transitionName` prop 的约定。

你可以使用这些类来触发CSS动画和过渡。比如，尝试添加这个CSS和添加一个新的列表项：

```
.example-enter {  
  opacity: 0.01;  
}  
  
.example-enter.example-enter-active {  
  opacity: 1;  
  transition: opacity 500ms ease-in;  
}  
  
.example-leave {  
  opacity: 1;  
}  
  
.example-leave.example-leave-active {  
  opacity: 0.01;  
  transition: opacity 300ms ease-in;  
}
```

你会注意到动画持续时间需要被同时在CSS和渲染方法里被指定;这告诉React什么时候从元素中移除动画类，并且 -- 如果它正在离开 -- 何时从DOM移除元素。

让初始化挂载动画

`ReactCSSTransitionGroup` 提供了可选的prop `transitionAppear`，来为在组件初始挂载添加一个额外的过渡阶段。通常在初始化挂载时没有过渡阶段因为 `transitionAppear` 的默认值为 `false`。下面是一个传递 `transitionAppear` 为值 `true` 的例子。

```
render: function() {  
  return (  
    <ReactCSSTransitionGroup transitionName="example" transitionAppear={true} transitionAppearTimeout={500}>  
      <h1>Fading at Initial Mount</h1>  
    </ReactCSSTransitionGroup>  
  );  
}
```

在初始化挂载时 `ReactCSSTransitionGroup` 将获得 `example-appear` CSS类 并且 `example-appear-active` CSS 类在下一刻被添加。

```
.example-appear {
  opacity: 0.01;
}

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

在初始化挂载，所有的 `ReactCSSTransitionGroup` 子级将会 `appear` 但不 `enter`。然而，所有后来添加到已存在的 `ReactCSSTransitionGroup` 的子级将 `enter` 但不 `appear`。

注意:

prop `transitionAppear` 在版本 `0.13` 被添加到 `ReactCSSTransitionGroup`。为了保持向后兼容，默认值被设置为 `false`。

制定类

可以为你的每一步过渡使用制定类名字。代理传递一个字符串到 `transitionName`，你可以传递一个含有 `enter` 或者 `leave` 类名的对象，或者一个含有 `enter`，`enter-active`，`leave-active`，和 `leave` 类名的对象。只要提供了 `enter` 和 `leave` 的类，`enter-active` 和 `leave-active` 类会被决定为后缀 `-active` 到类名的尾部。这里是两个使用制定类的例子：

```
...
<ReactCSSTransitionGroup
  transitionName={ {
    enter: 'enter',
    enterActive: 'enterActive',
    leave: 'leave',
    leaveActive: 'leaveActive',
    appear: 'appear',
    appearActive: 'appearActive'
  } }>
  {item}
</ReactCSSTransitionGroup>

<ReactCSSTransitionGroup
  transitionName={ {
    enter: 'enter',
    leave: 'leave',
    appear: 'appear'
  } }>
  {item2}
</ReactCSSTransitionGroup>
...
```


动画组必须挂载才工作

为了使过渡效果应用到子级上，`ReactCSSTransitionGroup` 必须已经挂载到了DOM或者 `prop transitionAppear` 必须被设置为 `true`。下面的例子不会工作，因为 `ReactCSSTransitionGroup` 随同新项目被挂载，而不是新项目在它内部被挂载。将这与上面的[入门指南](#)部分比较一下，看看不同。

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

动画一个或者零个项目 Animating One or Zero Items

在上面的例子中，我们渲染了一系列的项目到 `ReactCSSTransitionGroup` 里。然而 `ReactCSSTransitionGroup` 的子级同样可以是一个或零个项目。这使它能够动画化单个元素的进入和离开。同样，你可以动画化一个新的元素替换当前元素。例如，我们可以像这样实现一个简单的图片轮播器：

```
var ReactCSSTransitionGroup = require('react-addons-css-transition-group');

var ImageCarousel = React.createClass({
  propTypes: {
    imageUrl: React.PropTypes.string.isRequired
  },
  render: function() {
    return (
      <div>
        <ReactCSSTransitionGroup transitionName="carousel" transitionEnterTimeout={300}
        transitionLeaveTimeout={300}>
          <img src={this.props.imageUrl} key={this.props.imageUrl} />
        </ReactCSSTransitionGroup>
      </div>
    );
  }
});
```

禁用动画

如果你想,你可以禁用 `enter` 或者 `leave` 动画。例如,有时你可能想要一个 `enter` 动画,不要 `leave` 动画,但是 `ReactCSSTransitionGroup` 会在移除你的DOM节点之前等待一个动画完成。你可以添加 `transitionEnter={false}` 或者 `transitionLeave={false}` props 到 `ReactCSSTransitionGroup` 来禁用这些动画。

注意：

当使用 `ReactCSSTransitionGroup` 时,没有办法通知你的组件何时过渡效果结束或者在动画时执行任何复杂的逻辑运算。如果你想要更多细粒度的控制,你可以使用底层的 `ReactTransitionGroup` API,它提供了你自定义过渡效果所需要的挂钩。

底层 API: `ReactTransitionGroup`

`ReactTransitionGroup` 是动画的基础。它通过 `require('react-addons-transition-group')` 访问。当子级被声明式的从其中添加或移除(就像上面的例子)时,特殊的生命周期挂钩会在它们上面被调用。

`componentWillAppear(callback)`

对于被初始化挂载到 `TransitionGroup` 的组件,它和 `componentDidMount()` 在相同时间被调用。它将会阻塞其它动画发生,直到 `callback` 被调用。它只会在 `TransitionGroup` 初始化渲染时被调用。

componentDidAppear()

在传给 `componentWillAppear` 的回调函数被调用后调用。

componentWillEnter(callback)

对于被添加到已存在的 `TransitionGroup` 的组件，它和 `componentDidMount()` 在相同时间被调用。它将会阻塞其它动画发生，直到 `callback` 被调用。它不会在 `TransitionGroup` 初始化渲染时被调用。

componentDidEnter()

在传给 `componentWillEnter` 的回调函数被调用之后调用。

componentWillLeave(callback)

在子级从 `ReactTransitionGroup` 中移除时调用。虽然子级被移除了，`ReactTransitionGroup` 将会保持它在DOM中，直到 `callback` 被调用。

componentDidLeave()

在 `willLeave` `callback` 被调用的时候调用（与 `componentWillUnmount` 同一时间）。

渲染一个不同的组件

默认情况下 `ReactTransitionGroup` 渲染为一个 `span`。你可以通过提供一个 `component` prop 来改变这种行为。例如，下面是你将如何渲染一个 ``：

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

每一个React能渲染的DOM组件都是可用的。然而，`组件` 不需要是一个DOM组件。它可以是任何你想要的React组件；甚至是你自己已经写好的！只要写 `component={List}` 你的组件会收到 `this.props.children`

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

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


双向绑定辅助

`ReactLink` 是一个用 `React` 表达双向绑定的简单方法。

注意：

如果你刚学这个框架，注意 `ReactLink` 对大多数应用是不需要的，应该慎重的使用。

在 `React` 里，数据单向流动：从拥有者到子级。这是因为数据只单向流动 [the Von Neumann model of computing](#)。你可以把它想象为“单向数据绑定”。

然而，有很多应用需要你去读某些数据并回流他们到你的程序。例如，当开发 `forms`，你会常常想更新一些 `React state` 当你收到用户输入的时候。或者也许你想在 `JavaScript` 完成布局并相应一些 `DOM` 元素大小的变化。

在 `React` 里，你可以用监听 `"change"` 事件来实现它，从你的数据源（通常是 `DOM`）读取并在你的某个组件调用 `setState()`。明确的 `"Closing the data flow loop"` 致使了更容易理解和维护的程序。更多信息见 [our forms documentation](#)。

双向绑定 -- 隐含的强迫 `DOM` 里的某些值总是和某些 `React state` 同步 -- 简洁并支持大量多样的应用。我们提供了 `ReactLink`：设置如上描述的通用数据回流模式的语法糖，或者 `"linking"` 某些数据结构到 `React state`。

注意：

`ReactLink` 只是一层对 `onChange` `V` `setState()` 模式的薄包装。它没有根本性的改变你的 `React` 应用里数据如何流动。

ReactLink: 之前和之后

这里有一个简单的不用 `ReactLink` 的 `form` 例子：

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

这个工作的很好并且数据如何流动很清晰，然而，当有大量的 `form fields` 时，可能会有些冗长。让我们使用 `ReactLink` 来节省我们的输入：

```
var LinkedStateMixin = require('react-addons-linked-state-mixin');

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

`LinkedStateMixin` 添加了一个 `linkState()` 方法到你的 `React` 组件。`linkState()` 返回一个 `ReactLink` 包含当前 `React state` 值的对象和一个改变它的回调函数。

`ReactLink` 对象可以作为 `props` 在树中上下传递，所以很容易（显示的）在深层次的组件和高层次的 `state` 之间 设置双向绑定。

注意 `checkboxes` 有一个关于他们 `value` 属性的特殊行为，这个行为是 如果 `checkbox` 被选中 值会在表单提交时被发送。`value` 不会 `checkbox` 选中或是不选中时更新。对于 `checkboxes`，你应该用 `checkedLink` 代替 `valueLink`：

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

引擎盖下

There are two sides to `ReactLink` : the place where you create the `ReactLink` instance and the place where you use it. To prove how simple `ReactLink` is, let's rewrite each side separately to be more explicit.

ReactLink Without LinkedStateMixin

```

var WithoutMixin = React.createClass({
  getInitialState: function() {
    return {message: 'Hello!'};
  },
  handleChange: function(newValue) {
    this.setState({message: newValue});
  },
  render: function() {
    var valueLink = {
      value: this.state.message,
      requestChange: this.handleChange
    };
    return <input type="text" valueLink={valueLink} />;
  }
});

```

As you can see, `ReactLink` objects are very simple objects that just have a `value` and `requestChange` prop. And `LinkStateMixin` is similarly simple: it just populates those fields with a value from `this.state` and a callback that calls `this.setState()`.

ReactLink Without valueLink

```

var LinkStateMixin = require('react-addons-linked-state-mixin');

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

```

The `valueLink` prop is also quite simple. It simply handles the `onChange` event and calls `this.props.valueLink.requestChange()` and also uses `this.props.valueLink.value` instead of `this.props.value`. That's it!

类名操纵

NOTE:

此模块已被弃用; 用 [JedWatson/classnames](#) 替代.

Test Utilities

`ReactTestUtils` makes it easy to test React components in the testing framework of your choice (we use [Jest](#)).

```
var ReactTestUtils = require('react-addons-test-utils');
```

Simulate

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

Simulate an event dispatch on a DOM node with optional `eventData` event data. **This is possibly the single most useful utility in `ReactTestUtils`.**

Clicking an element

```
// <button ref="button">...</button>
var node = this.refs.button;
ReactTestUtils.Simulate.click(node);
```

Changing the value of an input field and then pressing ENTER.

```
// <input ref="input" />
var node = this.refs.input;
node.value = 'giraffe'
ReactTestUtils.Simulate.change(node);
ReactTestUtils.Simulate.keyDown(node, {key: "Enter", keyCode: 13, which: 13});
```

Note that you will have to provide any event property that you're using in your component (e.g. `keyCode`, `which`, etc...) as React is not creating any of these for you.

`Simulate` has a method for [every event that React understands](#).

renderIntoDocument

```
ReactDOM.renderIntoDocument(  
  ReactElement instance  
)
```

Render a component into a detached DOM node in the document. **This function requires a DOM.**

Note:

You will need to have `window`, `window.document` and `window.document.createElement` globally available **before** you import React. Otherwise React will think it can't access the DOM and methods like `setState` won't work.

mockComponent

```
object mockComponent(  
  function componentClass,  
  [string mockTagName]  
)
```

Pass a mocked component module to this method to augment it with useful methods that allow it to be used as a dummy React component. Instead of rendering as usual, the component will become a simple `<div>` (or other tag if `mockTagName` is provided) containing any provided children.

isElement

```
boolean isElement(  
  ReactElement element  
)
```

Returns `true` if `element` is any `ReactElement`.

isElementOfType

```
boolean isElementOfType(  
  ReactElement element,  
  function componentClass  
)
```

Returns `true` if `element` is a `ReactElement` whose type is of a React `componentClass`.

isDOMComponent

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

Returns `true` if `instance` is a DOM component (such as a `<div>` or ``).

isCompositeComponent

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

Returns `true` if `instance` is a composite component (created with `React.createClass()`).

isCompositeComponentWithType

```
boolean isCompositeComponentWithType(  
  ReactComponent instance,  
  function componentClass  
)
```

Returns `true` if `instance` is a composite component (created with `React.createClass()`) whose type is of a React `componentClass` .

findAllInRenderedTree

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

Traverse all components in `tree` and accumulate all components where `test(component)` is `true` . This is not that useful on its own, but it's used as a primitive for other test utils.

scryRenderedDOMComponentsWithClass

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

Finds all instances of components in the rendered tree that are DOM components with the class name matching `className` .

findRenderedDOMComponentWithClass

```
ReactComponent findRenderedDOMComponentWithClass(  
  ReactComponent tree,  
  string className  
)
```

Like `scryRenderedDOMComponentsWithClass()` but expects there to be one result, and returns that one result, or throws exception if there is any other number of matches besides one.

scryRenderedDOMComponentsWithTag

```
array scryRenderedDOMComponentsWithTag(  
  ReactComponent tree,  
  string tagName  
)
```

Finds all instances of components in the rendered tree that are DOM components with the tag name matching `tagName` .

findRenderedDOMComponentWithTag

```
ReactComponent findRenderedDOMComponentWithTag(  
  ReactComponent tree,  
  string tagName  
)
```

Like `scryRenderedDOMComponentsWithTag()` but expects there to be one result, and returns that one result, or throws exception if there is any other number of matches besides one.

scryRenderedComponentsWithType

```
array scryRenderedComponentsWithType(  
  ReactComponent tree,  
  function componentClass  
)
```

Finds all instances of components with type equal to `componentClass` .

findRenderedComponentWithType

```
ReactComponent findRenderedComponentWithType(  
  ReactComponent tree, function componentClass  
)
```

Same as `scryRenderedComponentsWithType()` but expects there to be one result and returns that one result, or throws exception if there is any other number of matches besides one.

Shallow rendering

Shallow rendering is an experimental feature that lets you render a component "one level deep" and assert facts about what its render method returns, without worrying about the behavior of child components, which are not instantiated or rendered. This does not require a DOM.

```
ReactShallowRenderer createRenderer()
```

Call this in your tests to create a shallow renderer. You can think of this as a "place" to render the component you're testing, where it can respond to events and update itself.

```
shallowRenderer.render(  
  ReactElement element  
)
```

Similar to `ReactDOM.render` .

```
ReactElement shallowRenderer.getRenderOutput()
```

After `render` has been called, returns shallowly rendered output. You can then begin to assert facts about the output. For example, if your component's render method returns:

```
<div>
  <span className="heading">Title</span>
  <Subcomponent foo="bar" />
</div>
```

Then you can assert:

```
var renderer = ReactTestUtils.createRenderer();
result = renderer.getRenderOutput();
expect(result.type).toBe('div');
expect(result.props.children).toEqual([
  <span className="heading">Title</span>,
  <Subcomponent foo="bar" />
]);
```

Shallow testing currently has some limitations, namely not supporting refs. We're releasing this feature early and would appreciate the React community's feedback on how it should evolve.

Cloning ReactElements

Note: `cloneWithProps` is deprecated. Use [React.cloneElement](#) instead.

In rare situations, you may want to create a copy of a React element with different props from those of the original element. One example is cloning the elements passed into `this.props.children` and rendering them with different props:

```
var cloneWithProps = require('react-addons-clone-with-props');

var _makeBlue = function(element) {
  return cloneWithProps(element, {style: {color: 'blue'}});
};

var Blue = React.createClass({
  render: function() {
    var blueChildren = React.Children.map(this.props.children, _makeBlue);
    return <div>{blueChildren}</div>;
  }
});

ReactDOM.render(
  <Blue>
    <p>This text is blue.</p>
  </Blue>,
  document.getElementById('container')
);
```

`cloneWithProps` does not transfer `key` or `ref` to the cloned element. `className` and `style` props are automatically merged.

Keyed Fragments

In most cases, you can use the `key` prop to specify keys on the elements you're returning from `render`. However, this breaks down in one situation: if you have two sets of children that you need to reorder, there's no way to put a key on each set without adding a wrapper element.

That is, if you have a component such as:

```
var Swapper = React.createClass({
  propTypes: {
    // `leftChildren` and `rightChildren` can be a string, element, array, etc.
    leftChildren: React.PropTypes.node,
    rightChildren: React.PropTypes.node,

    swapped: React.PropTypes.bool
  },
  render: function() {
    var children;
    if (this.props.swapped) {
      children = [this.props.rightChildren, this.props.leftChildren];
    } else {
      children = [this.props.leftChildren, this.props.rightChildren];
    }
    return <div>{children}</div>;
  }
});
```

The children will unmount and remount as you change the `swapped` prop because there aren't any keys marked on the two sets of children.

To solve this problem, you can use the `createFragment` add-on to give keys to the sets of children.

```
Array<ReactNode> createFragment(object children)
```

Instead of creating arrays, we write:


```
var createFragment = require('react-addons-create-fragment');

if (this.props.swapped) {
  children = createFragment({
    right: this.props.rightChildren,
    left: this.props.leftChildren
  });
} else {
  children = createFragment({
    left: this.props.leftChildren,
    right: this.props.rightChildren
  });
}
```

The keys of the passed object (that is, `left` and `right`) are used as keys for the entire set of children, and the order of the object's keys is used to determine the order of the rendered children. With this change, the two sets of children will be properly reordered in the DOM without unmounting.

The return value of `createFragment` should be treated as an opaque object; you can use the `React.Children` helpers to loop through a fragment but should not access it directly. Note also that we're relying on the JavaScript engine preserving object enumeration order here, which is not guaranteed by the spec but is implemented by all major browsers and VMs for objects with non-numeric keys.

Note:

In the future, `createFragment` may be replaced by an API such as

```
return (
  <div>
    <x:frag key="right">{this.props.rightChildren}</x:frag>,
    <x:frag key="left">{this.props.leftChildren}</x:frag>
  </div>
);
```

allowing you to assign keys directly in JSX without adding wrapper elements.

Immutability Helpers

React lets you use whatever style of data management you want, including mutation. However, if you can use immutable data in performance-critical parts of your application it's easy to implement a fast `shouldComponentUpdate()` method to significantly speed up your app.

Dealing with immutable data in JavaScript is more difficult than in languages designed for it, like [Clojure](#). However, we've provided a simple immutability helper, `update()`, that makes dealing with this type of data much easier, *without* fundamentally changing how your data is represented. You can also take a look at Facebook's [Immutable-js](#) and the [Advanced Performance](#) section for more detail on Immutable-js.

The main idea

If you mutate data like this:

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

You have no way of determining which data has changed since the previous copy has been overwritten. Instead, you need to create a new copy of `myData` and change only the parts of it that need to be changed. Then you can compare the old copy of `myData` with the new one in `shouldComponentUpdate()` using triple-equals:

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

Unfortunately, deep copies are expensive, and sometimes impossible. You can alleviate this by only copying objects that need to be changed and by reusing the objects that haven't changed. Unfortunately, in today's JavaScript this can be cumbersome:

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

While this is fairly performant (since it only makes a shallow copy of `log n` objects and reuses the rest), it's a big pain to write. Look at all the repetition! This is not only annoying, but also provides a large surface area for bugs.

`update()` provides simple syntactic sugar around this pattern to make writing this code easier. This code becomes:

```
var update = require('react-addons-update');

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

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

The `$`-prefixed keys are called *commands*. The data structure they are "mutating" is called the *target*.

Available commands

- `{$push: array}` `push()` all the items in `array` on the target.
- `{$unshift: array}` `unshift()` all the items in `array` on the target.
- `{$splice: array of arrays}` for each item in `arrays` call `splice()` on the target with the parameters provided by the item.
- `{$set: any}` replace the target entirely.
- `{$merge: object}` merge the keys of `object` with the target.
- `{$apply: function}` passes in the current value to the function and updates it with the new returned value.

Examples

Simple push

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

`initialArray` is still `[1, 2, 3]` .

Nested collections

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

This accesses `collection` 's index `2` , key `a` , and does a splice of one item starting from index `1` (to remove `17`) while inserting `13` and `14` .

Updating a value based on its current one

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

(Shallow) merge

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

PureRenderMixin

If your React component's render function is "pure" (in other words, it renders the same result given the same props and state), you can use this mixin for a performance boost in some cases.

Example:

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

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

Under the hood, the mixin implements [shouldComponentUpdate](#), in which it compares the current props and state with the next ones and returns `false` if the equalities pass.

Note:

This only shallowly compares the objects. If these contain complex data structures, it may produce false-negatives for deeper differences. Only mix into components which have simple props and state, or use `forceUpdate()` when you know deep data structures have changed. Or, consider using [immutable objects](#) to facilitate fast comparisons of nested data.

Furthermore, `shouldComponentUpdate` skips updates for the whole component subtree. Make sure all the children components are also "pure".

Performance Tools

React is usually quite fast out of the box. However, in situations where you need to squeeze every ounce of performance out of your app, it provides a [shouldComponentUpdate](#) hook where you can add optimization hints to React's diff algorithm.

In addition to giving you an overview of your app's overall performance, ReactPerf is a profiling tool that tells you exactly where you need to put these hooks.

General API

The `Perf` object documented here is exposed as `require('react-addons-perf')` and can be used with React in development mode only. You should not include this bundle when building your app for production.

Note:

The dev build of React is slower than the prod build, due to all the extra logic for providing, for example, React's friendly console warnings (stripped away in the prod build). Therefore, the profiler only serves to indicate the *relatively* expensive parts of your app.

`Perf.start()` and `Perf.stop()`

Start/stop the measurement. The React operations in-between are recorded for analyses below. Operations that took an insignificant amount of time are ignored.

After stopping, you will need `Perf.getLastMeasurements()` (described below) to get the measurements.

`Perf.printInclusive(measurements)`

Prints the overall time taken. If no argument's passed, defaults to all the measurements from the last recording. This prints a nicely formatted table in the console, like so:

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

`Perf.printExclusive(measurements)`

"Exclusive" times don't include the times taken to mount the components: processing props, `getInitialState`, call `componentWillMount` and `componentDidMount`, etc.

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

Perf.printWasted(measurements)

The most useful part of the profiler.

"Wasted" time is spent on components that didn't actually render anything, e.g. the render stayed the same, so the DOM wasn't touched.

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

Perf.printDOM(measurements)

Prints the underlying DOM manipulations, e.g. "set innerHTML" and "remove".

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

Advanced API

The above print methods use `Perf.getLastMeasurements()` to pretty-print the result.

Perf.getLastMeasurements()

Get the measurements array from the last start-stop session. The array contains objects, each of which looks like this:

```
{
  // The term "inclusive" and "exclusive" are explained below
  "exclusive": {},
  // '.0.0' is the React ID of the node
  "inclusive": {".0.0": 0.0670000008540228, ".0": 0.3259999939473346},
  "render": {".0": 0.036999990697950125, ".0.0": 0.010000003385357559},
  // Number of instances
  "counts": {".0": 1, ".0.0": 1},
  // DOM touches
  "writes": {},
  // Extra debugging info
  "displayNames": {
    ".0": {"current": "App", "owner": "<root>"},
    ".0.0": {"current": "Box", "owner": "App"}
  },
  "totalTime": 0.48499999684281647
}
```


浅比较

`shallowCompare` 是一个辅助函数 在以ES6类使用React时，完成和 `PureRenderMixin` 相同的功能。

如果你的React组件的绘制函数是“干净的”（换句话说，它在给定的 `props` 和 `state` 下绘制相同的结果），你可以使用这个辅助函数以在某些情况下提升性能。

例如：

```
var shallowCompare = require('react-addons-shallow-compare');
export class SampleComponent extends React.Component {
  shouldComponentUpdate(nextProps, nextState) {
    return shallowCompare(this, nextProps, nextState);
  }

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

`shallowCompare` 对当前的 `props` 和 `nextProps` 对象 执行一个浅的相等检查，同样对于 `state` 和 `nextState` 对象。它通过迭代比较对象的`keys` 并在 对象的`key`值不严格相等时返回`false` 实现此功能。

`shallowCompare` 返回 `true` 如果对 `props` 或 `state`的浅比较失败，因此组件应该更新。

`shallowCompare` 返回 `false` 如果对 `props` 或 `state`的浅比较都通过了，因此组件不应该更新。

Advanced Performance

One of the first questions people ask when considering React for a project is whether their application will be as fast and responsive as an equivalent non-React version. The idea of re-rendering an entire subtree of components in response to every state change makes people wonder whether this process negatively impacts performance. React uses several clever techniques to minimize the number of costly DOM operations required to update the UI.

Avoiding reconciling the DOM

React makes use of a *virtual DOM*, which is a descriptor of a DOM subtree rendered in the browser. This parallel representation allows React to avoid creating DOM nodes and accessing existing ones, which is slower than operations on JavaScript objects. When a component's props or state change, React decides whether an actual DOM update is necessary by constructing a new virtual DOM and comparing it to the old one. Only in the case they are not equal, will React [reconcile](#) the DOM, applying as few mutations as possible.

On top of this, React provides a component lifecycle function, `shouldComponentUpdate`, which is triggered before the re-rendering process starts (virtual DOM comparison and possible eventual DOM reconciliation), giving the developer the ability to short circuit this process. The default implementation of this function returns `true`, leaving React to perform the update:

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

Keep in mind that React will invoke this function pretty often, so the implementation has to be fast.

Say you have a messaging application with several chat threads. Suppose only one of the threads has changed. If we implement `shouldComponentUpdate` on the `ChatThread` component, React can skip the rendering step for the other threads:

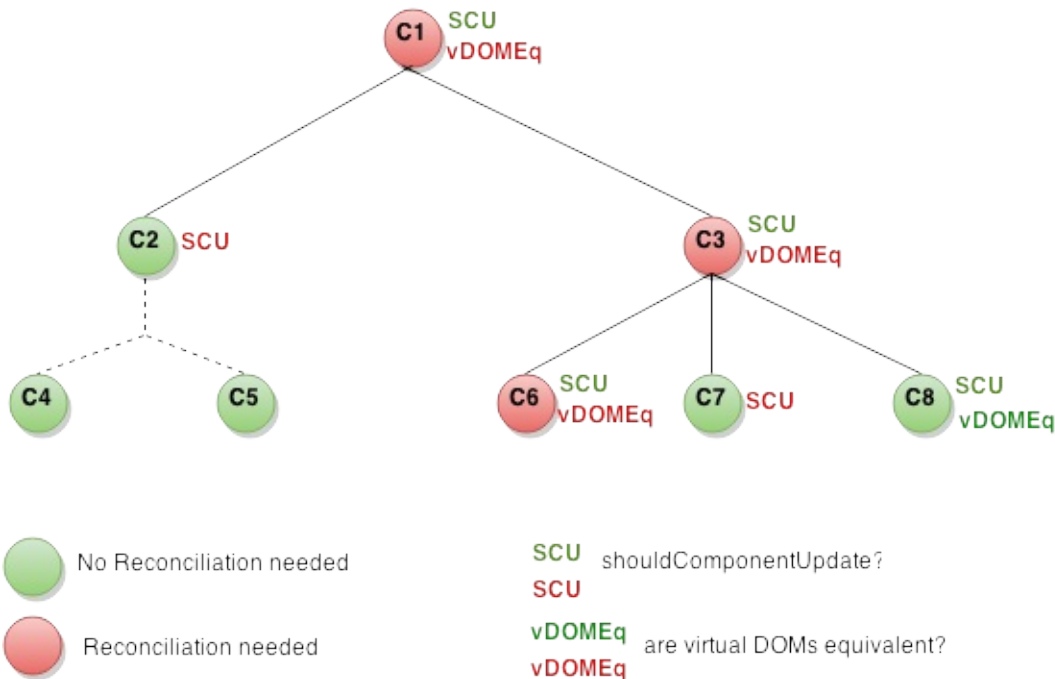
```
shouldComponentUpdate: function(nextProps, nextState) {
  // TODO: return whether or not current chat thread is
  // different to former one.
}
```

So, in summary, React avoids carrying out expensive DOM operations required to reconcile subtrees of the DOM by allowing the user to short circuit the process using

`shouldComponentUpdate`, and, for those which should update, by comparing virtual DOMs.

shouldComponentUpdate in action

Here's a subtree of components. For each one is indicated what `shouldComponentUpdate` returned and whether or not the virtual DOMs were equivalent. Finally, the circle's color indicates whether the component had to be reconciled or not.



In the example above, since `shouldComponentUpdate` returned `false` for the subtree rooted at C2, React had no need to generate the new virtual DOM, and therefore, it neither needed to reconcile the DOM. Note that React didn't even have to invoke `shouldComponentUpdate` on C4 and C5.

For C1 and C3 `shouldComponentUpdate` returned `true`, so React had to go down to the leaves and check them. For C6 it returned `true`; since the virtual DOMs weren't equivalent it had to reconcile the DOM. The last interesting case is C8. For this node React had to compute the virtual DOM, but since it was equal to the old one, it didn't have to reconcile its DOM.

Note that React only had to do DOM mutations for C6, which was inevitable. For C8, it bailed out by comparing the virtual DOMs, and for C2's subtree and C7, it didn't even have to compute the virtual DOM as we bailed out on `shouldComponentUpdate`.

So, how should we implement `shouldComponentUpdate`? Say that you have a component that just renders a string value:

```
React.createClass({
  propTypes: {
    value: React.PropTypes.string.isRequired
  },

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

We could easily implement `shouldComponentUpdate` as follows:

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

So far so good, dealing with such simple props/state structures is easy. We could even generalize an implementation based on shallow equality and mix it into components. In fact, React already provides such implementation: [PureRenderMixin](#).

But what if your components' props or state are mutable data structures? Say the prop the component receives, instead of being a string like `'bar'`, is a JavaScript object that contains a string such as, `{ foo: 'bar' }`:

```
React.createClass({
  propTypes: {
    value: React.PropTypes.object.isRequired
  },

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

The implementation of `shouldComponentUpdate` we had before wouldn't always work as expected:

```
// 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
```

The problem is `shouldComponentUpdate` will return `true` when the prop actually didn't change. To fix this, we could come up with this alternative implementation:

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

Basically, we ended up doing a deep comparison to make sure we properly track changes. In terms of performance, this approach is pretty expensive. It doesn't scale as we would have to write different deep equality code for each model. On top of that, it might not even work if we don't carefully manage object references. Say this component is used by a parent:

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

The first time the inner component gets rendered, it will have `{ foo: 'bar' }` as the value prop. If the user clicks on the anchor, the parent component's state will get updated to `{ value: { foo: 'barbar' } }`, triggering the re-rendering process of the inner component, which will receive `{ foo: 'barbar' }` as the new value for the prop.

The problem is that since the parent and inner components share a reference to the same object, when the object gets mutated on line 2 of the `onClick` function, the prop the inner component had will change. So, when the re-rendering process starts, and

`shouldComponentUpdate` gets invoked, `this.props.value.foo` will be equal to `nextProps.value.foo`, because in fact, `this.props.value` references the same object as `nextProps.value`.

Consequently, since we'll miss the change on the prop and short circuit the re-rendering process, the UI won't get updated from `'bar'` to `'barbar'`.

Immutable-js to the rescue

[Immutable-js](#) is a JavaScript collections library written by Lee Byron, which Facebook recently open-sourced. It provides *immutable persistent* collections via *structural sharing*. Let's see what these properties mean:

- *Immutable*: once created, a collection cannot be altered at another point in time.
- *Persistent*: new collections can be created from a previous collection and a mutation such as `set`. The original collection is still valid after the new collection is created.
- *Structural Sharing*: new collections are created using as much of the same structure as the original collection as possible, reducing copying to a minimum to achieve space efficiency and acceptable performance. If the new collection is equal to the original, the original is often returned.

Immutability makes tracking changes cheap; a change will always result in a new object so we only need to check if the reference to the object has changed. For example, in this regular JavaScript code:

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

Although `y` was edited, since it's a reference to the same object as `x`, this comparison returns `true`. However, this code could be written using `immutable-js` as follows:

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

In this case, since a new reference is returned when mutating `x`, we can safely assume that `x` has changed.

Another possible way to track changes could be doing dirty checking by having a flag set by setters. A problem with this approach is that it forces you to use setters and, either write a lot of additional code, or somehow instrument your classes. Alternatively, you could deep copy the object just before the mutations and deep compare to determine whether there was a change or not. A problem with this approach is both `deepCopy` and `deepCompare` are expensive operations.

So, Immutable data structures provides you a cheap and less verbose way to track changes on objects, which is all we need to implement `shouldComponentUpdate`. Therefore, if we model props and state attributes using the abstractions provided by `immutable-js` we'll be able to use `PureRenderMixin` and get a nice boost in perf.

Immutable-js and Flux

If you're using [Flux](#), you should start writing your stores using `immutable-js`. Take a look at the [full API](#).

Let's see one possible way to model the thread example using Immutable data structures. First, we need to define a `Record` for each of the entities we're trying to model. Records are just immutable containers that hold values for a specific set of fields:

```
var User = Immutable.Record({
  id: undefined,
  name: undefined,
  email: undefined
});

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

The `Record` function receives an object that defines the fields the object has and its default values.

The messages *store* could keep track of the users and messages using two lists:

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

It should be pretty straightforward to implement functions to process each *payload* type. For instance, when the store sees a payload representing a new message, we can just create a new record and append it to the messages list:

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

Note that since the data structures are immutable, we need to assign the result of the push function to `this.messages`.

On the React side, if we also use immutable-js data structures to hold the components' state, we could mix `PureRenderMixin` into all our components and short circuit the re-rendering process.

Context

React最大的优势之一是他很容易从你的React组件里跟踪数据流动。当你看着一个组件，你可以很容易准确看出哪个props被传入，这让你的APP很容易推断。

偶尔，你想通过组件树传递数据，而不在每一级上手工下传prop，React的 "context" 让你做到这点。

> 注意：> > Context是一个先进的实验性特性·这个 API 很可能在未来版本变化.>

大多数应用将不会需要用到 context. 尤其是如果你刚开始用React,你很可能不会想用 context.使用 context 将会使你的代码很难理解因为它让数据流不清晰.它类似于在你的应用里使用全局变量传递state.

如果你必须使用 **context** ,保守的使用它

不论你正在创建一个应用或者是库,试着分离你对 context 的使用到一个小区域,并尽可能避免直接使用 context API,以便在API变动时容易升级.

从树里自动传递info

假设你有一个这样的结构:

```
var Button = React.createClass({
  render: function() {
    return (
      <button style={{'{'}}background: this.props.color}}>
        {this.props.children}
      </button>
    );
  }
});

var Message = React.createClass({
  render: function() {
    return (
      <div>
        {this.props.text} <Button color={this.props.color}>Delete</Button>
      </div>
    );
  }
});

var MessageList = React.createClass({
  render: function() {
    var color = "purple";
    var children = this.props.messages.map(function(message) {
      return <Message text={message.text} color={color} />;
    });
    return <div>{children}</div>;
  }
});
```

在这里例子里,我们手工穿透一个 `color prop` 以便于恰当格式化 `Button` 和 `Message` 组件. 主题是一个很好的例子,当你可能想整个子树都可以访问一部分信息时(比如`color`). 使用 `context` 我们可以自动传过这个树:

```
var Button = React.createClass({
  contextTypes: {
    color: React.PropTypes.string
  },
  render: function() {
    return (
      <button style={{'background': this.context.color}}>
        {this.props.children}
      </button>
    );
  }
});

var Message = React.createClass({
  render: function() {
    return (
      <div>
        {this.props.text} <Button>Delete</Button>
      </div>
    );
  }
});

var MessageList = React.createClass({
  childContextTypes: {
    color: React.PropTypes.string
  },
  getChildContext: function() {
    return {color: "purple"};
  },
  render: function() {
    var children = this.props.messages.map(function(message) {
      return <Message text={message.text} />;
    });
    return <div>{children}</div>;
  }
});
```

通过添加 `childContextTypes` 和 `getChildContext` 到 `MessageList` (`context` 提供),`React`下传信息到子树中的任何组件(在这个例子中, `Button` 可以由定义 `contextTypes` 来访问它).

如果 `contextTypes` 没有定义,那么 `this.context` 将是一个空对象.

父子耦合

`Context` 同样可以使你构建这样的 APT:

```
<Menu>
  <MenuItem>aubergine</MenuItem>
  <MenuItem>butternut squash</MenuItem>
  <MenuItem>clementine</MenuItem>
</Menu>
```

通过在 `Menu` 组件下传相关的信息,每个 `MenuItem` 可以与包含他们的 `Menu` 组件沟通.

在你用这个**API**构建组件以前,考虑一下是否有清晰的替代方案 我们 喜欢像这样简单的用数组传递items:

```
<Menu items={['aubergine', 'butternut squash', 'clementine']} />
```

记住你同样可以在props里传递整个React组件,如果你想.

在生命周期方法里引用 **context**

如果 `contextTypes` 在一个组件中定义,接下来的生命周期方法会收到一个额外的参数,`context` 对象:

```
void componentWillReceiveProps(
  object nextProps, object nextContext
)

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

void componentWillUpdate(
  object nextProps, object nextState, object nextContext
)

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

在无状态函数组件里引用 **context**

无状态函数同样能够引用 `context` 如果 `contextTypes` 被定义为函数的属性.下面的代码展示了被写为无状态函数组件的 `Button` 组件.

```
function Button(props, context) {
  return (
    <button style={{'{{'}}background: context.color}}>
      {props.children}
    </button>
  );
}
Button.contextTypes = {color: React.PropTypes.string};
```

什么时候不用 **context**

正像全局变量是在写清晰代码时最好要避免的,你应该在大多数情况下避免使用**context**. 特别是,在它来"节省输入"和代替显示传入**props**时要三思.

context最好的使用场景是隐式的传入登录的用户,当前的语言,或者主题信息.要不然所有这些可能就是全局变量,但是**context**让你限定他们到一个单独的**React**树里.

不要用**context**在组件里传递你的模型数据.通过树显示的传递你的数据更容易理解.使用**context**使你的组件更耦合和不可复用,因为 依赖于他们在哪里渲染,他们会表现不同的行为.

已知的限制

如果一个由组件提供的**context**值变动,使用那个值的子级不会更新,如果一个直接的父级从 `shouldComponentUpdate` 返回 `false` .详见 [issue #2517](#) .

参考

- [Top-Level API](#)
- [组件 API](#)
- [组件的规范和生命周期](#)
- [Tags 和属性](#)
- [事件系统](#)
- [DOM 的不同之处](#)
- [特殊的 Non-DOM Attributes](#)
- [Reconciliation](#)
- [Web Components](#)
- [React \(虚拟\) DOM 术语](#)

Top-Level API

React

`React` 是 `React` 库的入口点。如果你使用预编译包中的一个，则 `React` 为全局变量；如果你使用 `CommonJS` 模块，你可以 `require()` 它。

React.Component

```
class Component
```

当使用ES6 类定义时，`React.Component`是 `React` 组件的基类。如何在`React`中使用 `ES6 class` 请参见 [可重用组件](#)。基类实际提供了哪些方法 请参见 [组件 API](#)。

React.createClass

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

给定一份规格（`specification`），创建一个组件类。组件通常要实现一个 `render()` 方法，它返回 单个的 子级。该子级可能包含任意深度的子级结构。组件与标准原型类的不同之处在于，你不需要对它们调用 `new`。它们是为你在后台构造实例（通过 `new`）的便利的包装器。

更多关于规格对象（`specification object`）的信息，请见 [组件规格和生命周期](#)。

React.createElement

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

创建并返回一个新的给定类型的 `ReactDOMElement`。 `type` 参数既可以是一个 `html` 标签名字符串（例如，“`div`”，“`span`”，等等），也可以是一个 `ReactClass`（用 `React.createClass` 创建的）。

React.cloneElement

```
ReactElement cloneElement(  
  ReactElement element,  
  [object props],  
  [children ...]  
)
```

使用 `element` 作为起点，克隆并返回一个新的 `ReactElement`。生成的 `element` 将会拥有原始 `element` 的 `props` 与新的 `props` 的浅合并。新的子级将会替换现存的子级。不同于 `React.addons.cloneWithProps`，来自原始 `element` 的 `key` 和 `ref` 将会保留。对于合并任何 `props` 没有特别的行为（不同于 `cloneWithProps`）。更多细节详见[v0.13 RC2 blog post](#)。

React.createFactory

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

返回一个生成给定类型的 `ReactElements` 的函数。如同 `React.createElement`，`type` 参数既可以是一个 html 标签名字字符串（例如“div”，“span”，等等），也可以是一个 `ReactClass`。

React.isValidElement

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

验证对象是否是一个 `ReactElement`。

React.DOM

`React.DOM` 用 `React.createElement` 为 DOM 组件提供了便利的包装器。该方式应该只在不使用 JSX 的时使用。例如，`React.DOM.div(null, 'Hello World!')`。

React.PropTypes

`React.PropTypes` 包含了能与组件的 `propTypes` 对象一起使用的类型，用以验证传入你的组件的 `props`。更多有关 `propTypes` 的信息，请见[可重用组件](#)。

React.Children

`React.Children` 为处理 `this.props.children` 这个不透明的数据结构提供了工具。

React.Children.map

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

在每一个包含在 `children` 中的直接子级上调用 `fn`，`fn` 中的 `this` 设置为 `thisArg`。如果 `children` 是一个嵌套的对象或者数组，它将被遍历：不会传入容器对象到 `fn` 中。如果 `children` 是 `null` 或者 `undefined`，则返回 `null` 或者 `undefined` 而不是一个空数组。

React.Children.forEach

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

类似 `React.Children.map()`，但是不返回数组。

React.Children.count

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

返回 `children` 中的组件总数，相等于传递给 `map` 或者 `forEach` 的回调函数应被调用次数。

React.Children.only

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

返回 `children` 中仅有的子级。否则抛出异常。

React.Children.toArray

```
array React.Children.toArray(object children)
```

以赋key给每个child的平坦的数组形式,返回不透明的 `children` 数据结构.如果你想操纵你的渲染方法的子级的合集这很有用,尤其如果你想在 `this.props.children` 传下之前渲染或者切割.

ReactDOM

`react-dom` 包提供了 具体的DOM方法,这些方法可以在你的app的顶层作为一个你需要时脱离 React模式的安全舱口 被使用.你的大多数组件不需要使用这个模块.

ReactDOM.render

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

渲染一个 `ReactElement` 到 DOM 里提供的 容器 (`container`) 中，并返回一个对 组件(或者返回 `null` 对于 无状态组件) 的引用

如果 `ReactElement` 之前被渲染到了 `container` 中，这将对它执行一次更新，并仅变动需要变动的 DOM 来反映最新的 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被执行。

注意:

`ReactDOM.render()` 控制你传入的 `container` 节点的内容。当初次调用时，任何现存于内的 DOM 元素将被替换。其后的调用使用 React 的 diffing 算法来有效率的更新。

`ReactDOM.render()` 不会修改 `container` 节点（只修改 `container` 的子级）。将来，也许能够直接插入一个组件到已经存在的 DOM 节点而不覆盖 现有的子级。

ReactDOM.unmountComponentAtNode

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

从 DOM 中移除已经挂载的 React 组件，并清除它的事件处理器和 `state`。如果在 `container` 中没有组件被挂载，调用此函数将什么都不做。如果组件被卸载返回 `true`，如果没有组件被卸载返回 `false`。

ReactDOM.findDOMNode

```
DOMElement findDOMNode(ReactComponent component)
```

如果这个组件已经被挂载到了 DOM，它返回相应的浏览器原生的 DOM 元素。这个方法对于读取 DOM 的值很有用，比如表单域的值和执行 DOM 的测量。在大多数情况下,你可以连接一个 `ref` 到 DOM 节点上,并避免使用 `findDOMNode` 如果 `render` 返回 `null` 或者 `false`，

`findDOMNode` 返回 `null` .

注意:

`findDOMNode()` 是一个用来访问底层DOM节点的安全舱口.大多数情况下,使用这个安全舱口是不被鼓励的,因为它穿破了组件的抽象.

`findDOMNode()` 只在已挂载的组件上工作(即是,已经被放置到DOM里的组件).如果你尝试在没有被挂载的组件上调用这个方法(比如在一个没有被创建的组件的 `render()` 里调用 `findDOMNode()`)会抛出一个异常.

`findDOMNode()` 不能用在无状态组件.

ReactDOMServer

`react-dom/server` 允许你在服务器上渲染你的组件.

ReactDOMServer.renderToString

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

把 `ReactElement` 渲染为它原始的 HTML 。这应该仅在服务器端使用。React 将会返回一个 HTML 字符串。你可以用这种方法在服务器端生成 HTML，然后在初始请求下传这些标记，以获得更快的页面加载速度及允许搜索引擎抓取页面（便于 SEO）。

如果在一个在已经有了这种服务器预渲染标记的节点上面调用 `ReactDOM.render()`，React 将会维护该节点，仅绑定事件处理器，让你有一个非常高效的初次加载体验。

ReactDOMServer.renderToStaticMarkup

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

类似于 `renderToString`，除了不创建额外的 DOM 属性，比如 `data-react-id`，这仅在 React 内部使用的属性。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，因为去除额外的属性能够节省很多字节。

组件 API

React.Component

当渲染时，React 组件的实例在 React 内部被创建。这些实例在随后的渲染中被重复使用，并可以在组件方法中通过 `this` 访问。唯一的在 React 之外获取 React 组件实例句柄的方法是保存 `ReactDOM.render` 的返回值。在其它组件内，你可以使用 `refs` 得到相同的结果。

setState

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

执行一个 `nextState` 到当前 `state` 的浅合并。这是你从事件处理器和服务器请求回调用来触发 UI 更新的主要手段。

第一个参数可以是一个对象（包含0或者多个keys来更新）或者一个（`state` 和 `props`的）函数，它返回一个包含要更新的keys的对象。

这里是一个简单的运用：

```
setState({mykey: 'my new value'});
```

也可以以 `function(state, props)` 传递一个函数。当你想要把一个在设置任何值之前参考前一次 `state+props` 的值的原子更新放在队列中 这会有很用。例如，假如我们想在 `state` 增加一个值：

```
setState(function(previousState, currentProps) {  
  return {myInteger: previousState.myInteger + 1};  
});
```

第二个（可选）的参数是一个将会在 `setState` 完成和组件被重绘后执行的回调函数。

注意:

绝对不要 直接改变 `this.state`，因为之后调用 `setState()` 可能会替换掉你做的改变。把 `this.state` 当做是不可变的。

`setState()` 不会立刻改变 `this.state`，而是创建一个即将处理的 `state` 转变。在调用该方法之后访问 `this.state` 可能会返回现有的值。

对 `setState` 的调用没有任何同步性的保证，并且调用可能会为了性能收益批量执行。

`setState()` 将总是触发一次重绘，除非在 `shouldComponentUpdate()` 中实现了条件渲染逻辑。如果可变对象被使用了，但又不能在 `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

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

类似于 `setState()`，但是删除任何 先前存在但不在 `nextState` 里的 `state` 键。

注意:

这个方法在从 `React.Component` 扩展的 ES6 `class` 组件里不可用。它也许会在未来的 React 版本中被完全移除。

forceUpdate

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

默认情况下，当你的组件的 `state` 或者 `props` 改变，你的组件将会重绘。然而，如果它们隐式的改变（例如：在对象深处的数据改变了但没有改变对象本身）或者如果你的 `render()` 方法依赖于其他的数据，你可以用调用 `forceUpdate()` 来告诉 React 它需要重新运行 `render()`。

调用 `forceUpdate()` 将会导致 `render()` 跳过 `shouldComponentUpdate()` 在组件上被调用，这会为子级触发正常的生命周期方法。包括每个子级的 `shouldComponentUpdate()` 方法。如果标记改变了，React 仍仅只更新 DOM。

通常你应该试着避免所有对 `forceUpdate()` 的使用并且在 `render()` 里只从 `this.props` 和 `this.state` 读取。这会使你的组件 "纯粹" 并且你的组件会更简单和高效。

getDOMNode

```
DOMNode getDOMNode()
```

如果这个组件已经被挂载到了 DOM，它返回相应的浏览器原生的 DOM 元素。这个方法对于读取 DOM 的值很有用，比如表单域的值和执行 DOM 的测量。如果 `render` 返回 `null` 或者 `false` 的时候，`this.getDOMNode()` 返回 `null`。

Note:

`getDOMNode` 被废弃了，已经被 [ReactDOM.findDOMNode\(\)](#) 替换。

这个方法在从 `React.Component` 扩展的 ES6 `class` 组件里不可用。它也许会在未来的 React 版本中被完全移除。

isMounted

```
boolean isMounted()
```

如果组件渲染到了 DOM 中，`isMounted()` 返回 `true`，否则返回 `false`。可以使用该方法来控制对 `setState()` 和 `forceUpdate()` 的异步调用。

注意:

这个方法在从 `React.Component` 扩展的 ES6 `class` 组件里不可用。它也许会在未来的 React 版本中被完全移除,所以你要移除它 [start migrating away from isMounted\(\) now](#)

setProps

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

当和一个外部的 JavaScript 应用整合的时候，你也许会想用 `ReactDOM.render()` 给 React 组件标示一个改变。

在根组件上调用 `setProps()` 会改变他的属性并触发一次重绘。另外，你可以提供一个可选的回调函数，一旦 `setProps` 完成并且组件被重绘它就执行。

注意:

这个方法被弃用了并会很快移除.这个方法在从 `React.Component` 扩展的 ES6 `class` 组件里不可用. 取代调用 `setProps`, 试着以新的 `props` 再次调用 `ReactDOM.render()`. 更多的注意事项, 见我们的[blog post about using the Top Level API](#)

如果可能, 上述的在同一个节点上再次调用 `ReactDOM.render()` 的方法是优先替代的。它往往使更新更容易理解。(两种方法并没有显著的性能区别。)

这个方法仅能在根组件上被调用。也就是说, 它仅在直接传给 `ReactDOM.render()` 的组件上可用, 在它的子级上不可用。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

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

类似于 `setProps()`, 但是删除任何先前存在的 `props`, 而不是合并这两个对象。

注意:

这个方法被弃用了并会很快移除.这个方法在从 `React.Component` 扩展的 ES6 `class` 组件里不可用. 取代调用 `replaceProps`, 试着以新的 `props` 再次调用 `ReactDOM.render()`. 更多的注意事项, 见我们的[blog post about using the Top Level API](#)

组件的规范和生命周期

组件规范(Specifications)

当调用 `React.createClass()` 创建一个组件类时,你应该提供一个包含有 `render` 方法以及可选的其他生命周期方法的 规范(Specifications)对象。

注意:

同样可以使用单纯的 JavaScript 类作为组件类. 这些类可以实现大多数相同的方法,虽然有一些不同.更多关于不同的信息,请阅读我们关于[ES6 classes](#)的文档.

render

```
ReactDOM.render()
```

`render()` 是必须的

当被调用时,它应该检查 `this.props` 和 `this.state` 并返回单个子元素.这个子元素即可以是一个 对原生DOM的虚拟表达(比如 `<div />` 或 `ReactDOM.div()`)也可以是其他你自定义的复合组件.

你也可以返回 `null` 或 `false` 来指示你不想要任何东西被渲染.幕后,React 渲染一个

`<noscript>` tag 来与我们当前的diffing算法协同工作.当返回 `null` 或 `false` , `ReactDOM.findDOMNode(this)` 会返回 `null` .

`render()` 函数应该是纯净的,意味着它不改变组件的状态,它在每次调用时返回相同的结果,并且它不读和写 DOM 或者其他方式与浏览器互动(例如,使用 `setTimeout`).如果你需要与浏览器互动,在 `componentDidMount()` 里执行你的工作,或者其他生命周期方法里.保持 `render()` 纯净使服务器渲染更实用并且让组件更容易被思考.

getInitialState

```
object getInitialState()
```

当组件被挂载时调用一次.返回值会被用作 `this.state` 的初始值.

getDefaultProps


```
object getDefaultProps()
```

在类被创建时调用一次并被缓存.在这个mapping里的值会被设置给 `this.props` 如果父组件没有指定对应的 `prop` (例如 使用一个 `in` 检查).

这个方法在任何实例被创建之前调用,因此不能依赖于 `this.props` .另外,小心,任何被 `getDefaultProps()` 返回的复杂对象会被跨实例共享,而不是被拷贝.

propTypes

```
object propTypes
```

`propTypes` 对象允许你验证传递到你的组建的 `props`.更多关于 `propTypes` 的信息,见 [Reusable Components](#).

mixins

```
array mixins
```

`mixins` 数组允许你用 `mixins` 来在多个组件间共享行为.更多关于 `mixins` 的信息,见 [Reusable Components](#).

statics

```
object statics
```

`statics` 对象允许你定义可以在组件类上调用的静态方法.例如:

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

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

在这个块里定义的方法是 **static**,意味着你可以在任何组件实例被创建前运行他们,并且这些方法没有对你组件的 **props** 或 **state** 的访问权.如果你在静态方法里检查**props**的值,把调用者作为参数传入**props**给静态函数.

displayName

```
string displayName
```

`displayName` 字符串被用在调试信息.JSX 自动设置这个值;见 [JSX in Depth](#).

Lifecycle Methods

多种方法在组件生命周期的特定点上被执行.

Mounting: componentWillMount

```
void componentWillMount()
```

被调用一次,即在客户端也在服务端,在最初的渲染发生之前 立即被调用.如果你在这个方法里调用 `setState` , `render()` 将会看到更新的 **state** 并不论**state**的变化只执行一次.

Mounting: componentDidMount

```
void componentDidMount()
```

被调用一次,只在客户端(不在服务端),在最初的渲染发生之后 立即被调用.在生命周期的这个点上,你可以访问任何对你的子级的**refs** (比如 访问底层的**DOM**表达).子组件的

`componentDidMount()` 方法在父组件之前被调用.

如果你想与其他 **JavaScript** 框架整合,用 `setTimeout` 或 `setInterval` 设置**timers**,或者发送 **AJAX** 请求,执行这些操作在此方法中.

Updating: componentWillReceiveProps

```
void componentWillReceiveProps(  
  object nextProps  
)
```

当一个组件收到新的`props`时被调用.这个方法不会为最初的渲染调用.

使用它作为响应 `prop` 转换的时机(在 `render()` 被用 `this.setState()` 更新`state`调用 之前). 旧的 `props` 可以通过 `this.props` 访问. 在这个函数里调用 `this.setState()` 不会触发任何额外的渲染.

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

注意:

并没有类似的 `componentWillReceiveState` 方法. 一个即将到来的 `prop` 转变可能会导致一个 `state` 变化,但是反之不是. 如果你需要实现一个对 `state` 变化相应的操作,使用

`componentWillUpdate` .

Updating: shouldComponentUpdate

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

当新的`props`或者`state`被收到,在渲染前被调用.这个方法不会在最初的渲染或者 `forceUpdate` 时被调用.

使用此方法作为一个 `return false` 的时机,当你确定新的 `props` 和 `state` 的转换不需要组件更新时.

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

如果 `shouldComponentUpdate` 返回`false`, `render()` 会在下次`state`变化前被完全跳过. 另外, `componentWillUpdate` 和 `componentDidUpdate` 将不会被调用.

默认情况下, `shouldComponentUpdate` 总是返回 `true` 来阻止当 `state` 突变时的细微bug,但是如果你仔细的把 `state` 作为不变量对待并只从 `render()` 里的 `props` 和 `state` 读,你就可以用一个比较旧的`props`和`state`与他们的替换者的实现来重写 `shouldComponentUpdate` .

如果性能是瓶颈,尤其是随着成百上千的组件,使用 `shouldComponentUpdate` 来加速你的app.

Updating: componentWillUpdate

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

当新的props或者state被接受时,在渲染前被立即调用.这个方法不会被初始渲染调用.

使用这个方法作为 在更新发生前执行一些准备 的时机.

Note:

你 不能 在这个方法里使用 `this.setState()` .如果你需要响应一个prop变化来更新state, 使用 `componentWillReceiveProps` 来替代.

Updating: componentDidUpdate

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

在组件的更新被刷新到DOM后立即被调用.这个方法不会被初始渲染调用.

使用这个方法作为 当组件被更新后在DOM上操作 的时机.

Unmounting: componentWillUnmount

```
void componentWillUnmount()
```

在组件被从DOM卸载 前 被立即调用.

在这个方法里执行一些必要的清理操作,比如无效化 timers 或者清理任何被

`componentDidMount` 创建的DOM元素.

Tags 和属性

支持的 Tags

React 试着支持所有常见的元素。如果你需要一个没有列在这里的元素，请 [file an issue](#)。

HTML 元素

下面的 HTML 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

下面的 SVG 元素是被支持的:

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

你也许对 [react-art](#) 有兴趣, 一个让 React 绘制 Canvas, SVG, 或者 VML (for IE8) 的绘制库.

支持的属性

React 支持所有的 `data-*` 和 `aria-*` 以及下列的属性.

注意:

所有的属性都是 camel-cased, `class` 和 `for` 分别是 `className` 和 `htmlFor`, 来符合 DOM API 规范.

关于事件的列表, 见 [Supported Events](#).

HTML 属性

下面的标准属性是被支持的:

```
accept acceptCharset accessKey action allowFullScreen allowTransparency alt
async autoComplete autoFocus autoplay capture cellPadding cellSpacing charSet
challenge 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
headers height hidden high href hrefLang htmlFor httpEquiv icon id inputMode
keyParams keyType label lang list loop low manifest marginHeight marginWidth max
maxLength media mediaGroup method min minLength multiple muted name noValidate open
optimum pattern placeholder poster preload radioGroup readOnly rel required role
rows rowSpan sandbox scope scoped scrolling seamless selected shape size sizes
span spellCheck src srcDoc srcSet start step style summary tabIndex target title
type useMap value width wmode wrap
```

另外,支持下面的非标准属性:

- `autoCapitalize` `autoCorrect` for Mobile Safari.
- `property` for [Open Graph](#) meta tags.
- `itemProp` `itemScope` `itemType` `itemRef` `itemID` for [HTML5 microdata](#).
- `unselectable` for Internet Explorer.
- `results` `autoSave` for WebKit/Blink input fields of type `search`.

同样有React规范的属性 `dangerouslySetInnerHTML` ([more here](#)),用于直接插入HTML字符串到组件里.

SVG 属性

```
clipPath 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 xlinkActuate xlinkArcrole xlinkHref xlinkRole
xlinkShow xlinkTitle xlinkType xmlBase xmlLang xmlSpace 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()
boolean isDefaultPrevented()
void stopPropagation()
boolean isPropagationStopped()
DOMEventTarget target
number timeStamp
string type
```

注意：

React v0.14 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()`。

事件池

`SyntheticEvent` 是池化的。这意味着 `SyntheticEvent` 对象将会被重用并且所有的属性都会在事件回调被调用后被 `nullified`。这是因为性能的原因。因此，你不能异步的访问事件。

```
function onClick(event) {
  console.log(event); // => nullified object.
  console.log(event.type); // => "click"
  var eventType = event.type; // => "click"

  setTimeout(function() {
    console.log(event.type); // => null
    console.log(eventType); // => "click"
  }, 0);

  this.setState({clickEvent: event}); // Won't work. this.state.clickEvent will only contain null values.
  this.setState({eventType: event.type}); // You can still export event properties.
}
```

注意:

如果你想异步访问事件属性,你应该在事件上调用 `event.persist()` ,这会从池中移除合成事件并允许对事件的引用被用会保留。

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` 处理点击事件的捕获阶段,而不是 `onClick`。

剪贴板事件

事件名称:

```
onCopy onCut onPaste
```

属性:

```
DOMDataTransfer clipboardData
```

Composition 事件

事件名称:

```
onCompositionEnd onCompositionStart onCompositionUpdate
```


属性:

```
string data
```

键盘事件

事件名称：

```
onKeyDown onKeyPress onKeyUp
```

属性:

```
boolean altKey  
number charCode  
boolean ctrlKey  
boolean 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
```

焦点事件在所有的React DOM上工作,不仅仅是表单元素.

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
```

`onmouseenter` 和 `onmouseleave` 事件从离开的元素传播到进入的元素,代替冒泡排序并且没有捕获阶段.

属性：

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

Selection Events

事件名称:

```
onSelect
```

触控事件

事件名称：

```
onTouchCancel onTouchEnd onTouchMove onTouchStart
```

属性：

```
boolean altKey
DOMTouchList changedTouches
boolean ctrlKey
boolean 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
```

Media Events

事件名称：

```
onAbort onCanPlay onCanPlayThrough onDurationChange onEmptied onEncrypted onEnded onError
onLoadedData onLoadedMetadata onLoadStart onPause onPlay onPlaying onProgress onRateChange
onSeeked onSeeking onStalled onSuspend onTimeUpdate onVolumeChange onWaiting
```

Image Events

事件名称:

onLoad onError

DOM 的不同之处

React 实现了一个浏览器无关的事件和 DOM 系统，原因是为了性能和跨浏览器的兼容性。我们利用这个机会来清理了一些浏览器 DOM 实现的一些粗糙边缘。

- 所有的 DOM properties 和 attributes (包括事件处理器) 都应该 camelCased 来保持和标准的 JavaScript 风格一致。我们在这里故意打破了这个规范是因为规范是不一致的。然而，data-* 和 aria-* attributes conform to the specs 应该只 lower-cased。
- style attribute 接受 camelCased properties 的 JavaScript 对象而不是一个 CSS 字符串。这保持了和 DOM style JavaScript property 的一致,更有效率，而且阻止了 XSS 安全漏洞。
- 因为 class 和 for 是 JavaScript 的保留字，内建 DOM nodes 的 JSX 元素应该分别使用 attribute 名 className 和 htmlFor，（比如 `<div className="foo" />`）。自定义元素应该直接使用 class 和 for（例如。`<my-tag class="foo" />`）。
- 所有事件对象遵循 W3C 规范，所有事件（包括提交）正确按 W3C 规范冒泡。详见 [事件系统](#)。
- onChange 事件表现的像你期望的那样：不论何时一个表单域改变了这个事件就会激发，而不是模糊的不一致。我们故意打破了已有浏览器的行为，因为 onChange 对于他的行为来说用词不当，并且 React 依赖于这个事件来实时响应用户的输入。详见 [表单组件](#)。
- 表单输入 attributes 类似 value 和 checked，就像 textarea 一样 [More here](#)。

特殊的 Non-DOM Attributes

和 [DOM 的不同之处](#) 相比，React 提供了一些不存在于 DOM 的 attributes。

- `key`：一个可选的。独特的标识。当你的组件穿梭于 `render` 的 pass，它也许会因为 diff 算法被摧毁和重建。赋予它一个持久的 key 保证这个 component 可达。详见[这里](#)。
- `ref`：见[这里](#)。
- `dangerouslySetInnerHTML`：提供了直接插入 raw HTML 的能力，主要是为了与操纵 DOM 字符串的库协作。详见[这里](#)。

Reconciliation

React's key design decision is to make the API seem like it re-renders the whole app on every update. This makes writing applications a lot easier but is also an incredible challenge to make it tractable. This article explains how with powerful heuristics we managed to turn a $O(n^3)$ problem into a $O(n)$ one.

Motivation

Generating the minimum number of operations to transform one tree into another is a complex and well-studied problem. The [state of the art algorithms](#) have a complexity in the order of $O(n^3)$ where n is the number of nodes in the tree.

This means that displaying 1000 nodes would require in the order of one billion comparisons. This is far too expensive for our use case. To put this number in perspective, CPUs nowadays execute roughly 3 billion instructions per second. So even with the most performant implementation, we wouldn't be able to compute that diff in less than a second.

Since an optimal algorithm is not tractable, we implement a non-optimal $O(n)$ algorithm using heuristics based on two assumptions:

1. Two components of the same class will generate similar trees and two components of different classes will generate different trees.
2. It is possible to provide a unique key for elements that is stable across different renders.

In practice, these assumptions are ridiculously fast for almost all practical use cases.

Pair-wise diff

In order to do a tree diff, we first need to be able to diff two nodes. There are three different cases being handled.

Different Node Types

If the node type is different, React is going to treat them as two different sub-trees, throw away the first one and build/insert the second one.

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

The same logic is used for custom components. If they are not of the same type, React is not going to even try at matching what they render. It is just going to remove the first one from the DOM and insert the second one.

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

Having this high level knowledge is a very important aspect of why React's diff algorithm is both fast and precise. It provides a good heuristic to quickly prune big parts of the tree and focus on parts likely to be similar.

It is very unlikely that a `<Header>` element is going to generate a DOM that is going to look like what a `<Content>` would generate. Instead of spending time trying to match those two structures, React just re-builds the tree from scratch.

As a corollary, if there is a `<Header>` element at the same position in two consecutive renders, you would expect to see a very similar structure and it is worth exploring it.

DOM Nodes

When comparing two DOM nodes, we look at the attributes of both and can decide which of them changed in linear time.

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

Instead of treating style as an opaque string, a key-value object is used instead. This lets us update only the properties that changed.

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

After the attributes have been updated, we recurse on all the children.

Custom Components

We decided that the two custom components are the same. Since components are stateful, we cannot just use the new component and call it a day. React takes all the attributes from the new component and calls `component[Will/Did]ReceiveProps()` on the previous one.

The previous component is now operational. Its `render()` method is called and the diff algorithm restarts with the new result and the previous result.

List-wise diff

Problematic Case

In order to do children reconciliation, React adopts a very naive approach. It goes over both lists of children at the same time and generates a mutation whenever there's a difference.

For example if you add an element at the end:

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

Inserting an element at the beginning is problematic. React is going to see that both nodes are spans and therefore run into a mutation mode.

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

There are many algorithms that attempt to find the minimum sets of operations to transform a list of elements. [Levenshtein distance](#) can find the minimum using single element insertion, deletion and substitution in $O(n^2)$. Even if we were to use Levenshtein, this doesn't find when a node has moved into another position and algorithms to do that have much worse complexity.

Keys

In order to solve this seemingly intractable issue, an optional attribute has been introduced. You can provide for each child a key that is going to be used to do the matching. If you specify a key, React is now able to find insertion, deletion, substitution and moves in $O(n)$ using a hash table.

```
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>]
```

In practice, finding a key is not really hard. Most of the time, the element you are going to display already has a unique id. When that's not the case, you can add a new ID property to your model or hash some parts of the content to generate a key. Remember that the key only has to be unique among its siblings, not globally unique.

Trade-offs

It is important to remember that the reconciliation algorithm is an implementation detail. React could re-render the whole app on every action; the end result would be the same. We are regularly refining the heuristics in order to make common use cases faster.

In the current implementation, you can express the fact that a sub-tree has been moved amongst its siblings, but you cannot tell that it has moved somewhere else. The algorithm will re-render that full sub-tree.

Because we rely on two heuristics, if the assumptions behind them are not met, performance will suffer.

1. The algorithm will not try to match sub-trees of different components classes. If you see yourself alternating between two components classes with very similar output, you may want to make it the same class. In practice, we haven't found this to be an issue.
2. Keys should be stable, predictable, and unique. Unstable keys (like those produced by `Math.random()`) will cause many nodes to be unnecessarily re-created, which can cause performance degradation and lost state in child components.

Web Components

Trying to compare and contrast React with WebComponents inevitably results in specious conclusions, because the two libraries are built to solve different problems. WebComponents provide strong encapsulation for reusable components, while React provides a declarative library that keeps the DOM in sync with your data. The two goals are complementary; engineers can mix-and-match the technologies. As a developer, you are free to use React in your WebComponents, or to use WebComponents in React, or both.

Using Web Components in React

```
class HelloMessage extends React.Component{
  render() {
    return <div>Hello <x-search>{this.props.name}</x-search>!</div>;
  }
}
```

Note:

The programming models of the two component systems (web components vs. react components) differ in that web components often expose an imperative API (for instance, a `video` web component might expose `play()` and `pause()` functions). To the extent that web components are declarative functions of their attributes, they should work, but to access the imperative APIs of a web component, you will need to attach a ref to the component and interact with the DOM node directly. If you are using third-party web components, the recommended solution is to write a React component that behaves as a wrapper for your web component.

At this time, events emitted by a web component may not properly propagate through a React render tree. You will need to manually attach event handlers to handle these events within your React components.

Using React in your Web Components

```
var proto = Object.create(HTMLElement.prototype, {
  createdCallback: {
    value: function() {
      var mountPoint = document.createElement('span');
      this.createShadowRoot().appendChild(mountPoint);

      var name = this.getAttribute('name');
      var url = 'https://www.google.com/search?q=' + encodeURIComponent(name);
      ReactDOM.render(<a href={url}>{name}</a>, mountPoint);
    }
  }
});
document.registerElement('x-search', {prototype: proto});
```

Complete Example

Check out the `webcomponents` example in the [starter kit](#) for a complete example.

React (虚拟) DOM 术语

在 React 的术语中,有五个要重点区分的核心类型:

- [ReactElement / ReactElement Factory](#)
- [ReactNode](#)
- [ReactComponent / ReactComponent Class](#)

React Elements(React 元素)

React里的首要类型是 `ReactElement` .它有四个 properties: `type` , `props` , `key` 和 `ref` .它没有方法,在 `prototype` 上什么也没有.

你可以通过 `React.createElement` 来创建这些对象.

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

要渲染一个新的树到DOM上,你创建 `ReactElement` s 并传递他们到 `ReactDOM.render` 伴随着一个标准的 DOM `Element` (`HTMLElement` 或 `SVGElement`). `ReactElement` s 不要与 DOM `Element` s 混淆. `ReactElement` 是一个轻的,有状态的,不可变的,虚拟的DOM `Element` 的表达.它是一个虚拟 DOM.

```
ReactDOM.render(root, document.getElementById('example'));
```

要给一个DOM元素添加 properties,传递一个properties 对象作为第二个参数,第三个参数传递子级.

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

如果你使用 React JSX,这些 `ReactElement` s 已经为你创建了.所以 这是等价的:

```
var root = <ul className="my-list">
  <li>Text Content</li>
</ul>;
ReactDOM.render(root, document.getElementById('example'));
```

Factories(工厂)

`ReactDOM.createElement` 工厂是一个产生特定 `type` `property` 的 `ReactDOM` 的函数。`React` 有一个为你内建的辅助工具来创建工厂. 它想这样起作用:

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

它允许你创建一个方便的速记 来代替每次输入 `React.createElement('div')` .

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

`React` 已经具备用于常用 `HTML tags` 的内建工厂

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

如果你使用 `JSX` 你没有必要使用工厂. `JSX` 已经为创建 `ReactDOM` `s` 提供了一个 方便的速记.

React Nodes

一个 `ReactNode` 可以是:

- `ReactDOM`
- `string` (aka `ReactDOM`)
- `number` (aka `ReactDOM`)
- `Array of ReactNode s` (aka `ReactDOM`)

他们被用作其他 `ReactDOM` `s` 的 `properties` 来表示子级. 事实上他们创建了一个 `ReactDOM` `s` 的树.

React Components

你可以使用 `React` 只使用 `ReactDOM` `s` 但是要真正利用 `React`, 你将要使用 `ReactDOM` `s` 来创建内嵌 `state` 的封装.

一个 `ReactDOM` 类就是一个 `JavaScript` 类 (或者 "constructor function").

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

当这个构造函数被调用,期望返回一个至少有一个 `render` 方法的对象.这个对象被称为一个 `ReactComponent` .

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

与测试不同,你可能通常 绝不会 亲自调用这个构造函数.`React` 为你调用它.

作为替代,你传递 `ReactComponent` 类到 `createElement` ,你得到一个 `ReactElement` .

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

或者用 `JSX`:

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

当这个被传给 `ReactDOM.render` ,`React` 会为你调用构造函数并创建一个 `ReactComponent` ,返回给你.

```
var component = ReactDOM.render(element, document.getElementById('example'));
```

如果你保持用相同类型的 `ReactElement` 和相同的 `DOM Element` 容器调用 `ReactDOM.render` ,它总是会返回相同的实例.这个实例是状态化的.

```
var componentA = ReactDOM.render(<MyComponent />, document.getElementById('example'));
var componentB = ReactDOM.render(<MyComponent />, document.getElementById('example'));
componentA === componentB; // true
```

这就是为什么你不应该构造你自己的实例.作为替代, `ReactElement` 在它被构造以前 是一个虚拟的 `ReactComponent` .一个老的和新的 `ReactElement` 可以被比较来判断 一个新的 `ReactComponent` 实例是否需要被创建或者已经存在的是否应该被重用.

`ReactComponent` 的 `render` 方法被期望返回另一个 `ReactElement` .这允许这些组件被结构化.最后,渲染分解为 带着一个 `string tag` 的 `ReactElement` ,它实例化一个 `DOM Element` 实例并把它插入 `document` 里.

Formal Type Definitions

Entry Point

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

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;
```

Classes and Components


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

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

FLUX

- [概览](#)
- [TodoMVC 教程](#)

概览

Flux is the application architecture that Facebook uses for building client-side web applications. It complements React's composable view components by utilizing a unidirectional data flow. It's more of a pattern rather than a formal framework, and you can start using Flux immediately without a lot of new code.



Flux applications have three major parts: the dispatcher, the stores, and the views (React components). These should not be confused with Model-View-Controller. Controllers do exist in a Flux application, but they are controller-views — views often found at the top of the hierarchy that retrieve data from the stores and pass this data down to their children. Additionally, action creators — dispatcher helper methods — are used to support a semantic API that describes all changes that are possible in the application. It can be useful to think of them as a fourth part of the Flux update cycle.

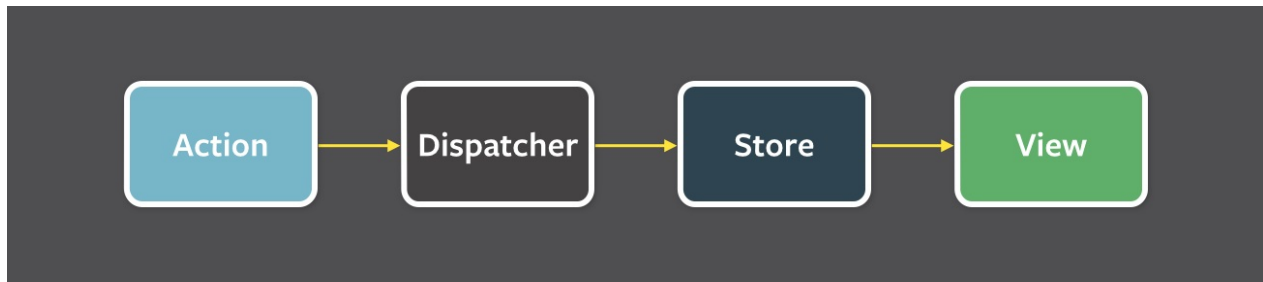
Flux eschews MVC in favor of a unidirectional data flow. When a user interacts with a React view, the view propagates an action through a central dispatcher, to the various stores that hold the application's data and business logic, which updates all of the views that are affected. This works especially well with React's declarative programming style, which allows the store to send updates without specifying how to transition views between states.

We originally set out to deal correctly with derived data: for example, we wanted to show an unread count for message threads while another view showed a list of threads, with the unread ones highlighted. This was difficult to handle with MVC — marking a single thread as read would update the thread model, and then also need to update the unread count model. These dependencies and cascading updates often occur in a large MVC application, leading to a tangled weave of data flow and unpredictable results.

Control is inverted with stores: the stores accept updates and reconcile them as appropriate, rather than depending on something external to update its data in a consistent way. Nothing outside the store has any insight into how it manages the data for its domain, helping to keep a clear separation of concerns. Stores have no direct setter methods like `setAsRead()`, but instead have only a single way of getting new data into their self-contained world — the callback they register with the dispatcher.

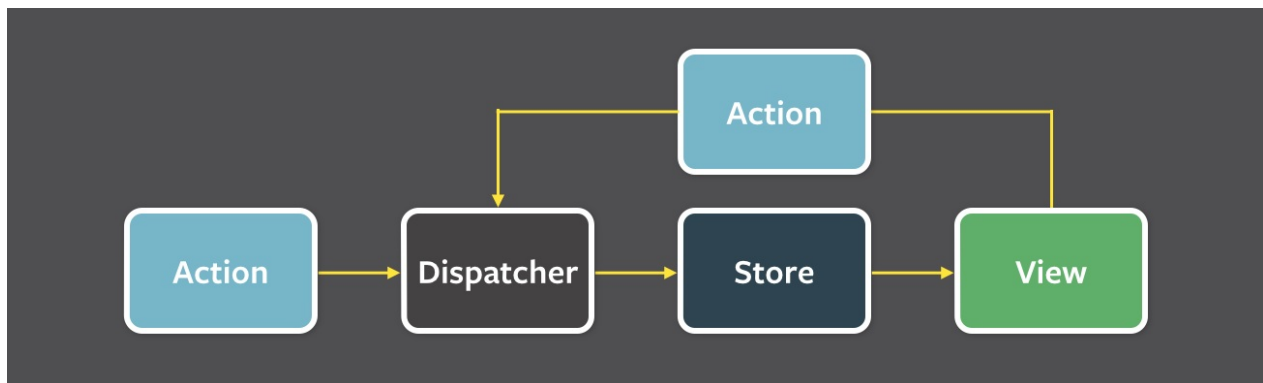
Structure and Data Flow

Data in a Flux application flows in a single direction:

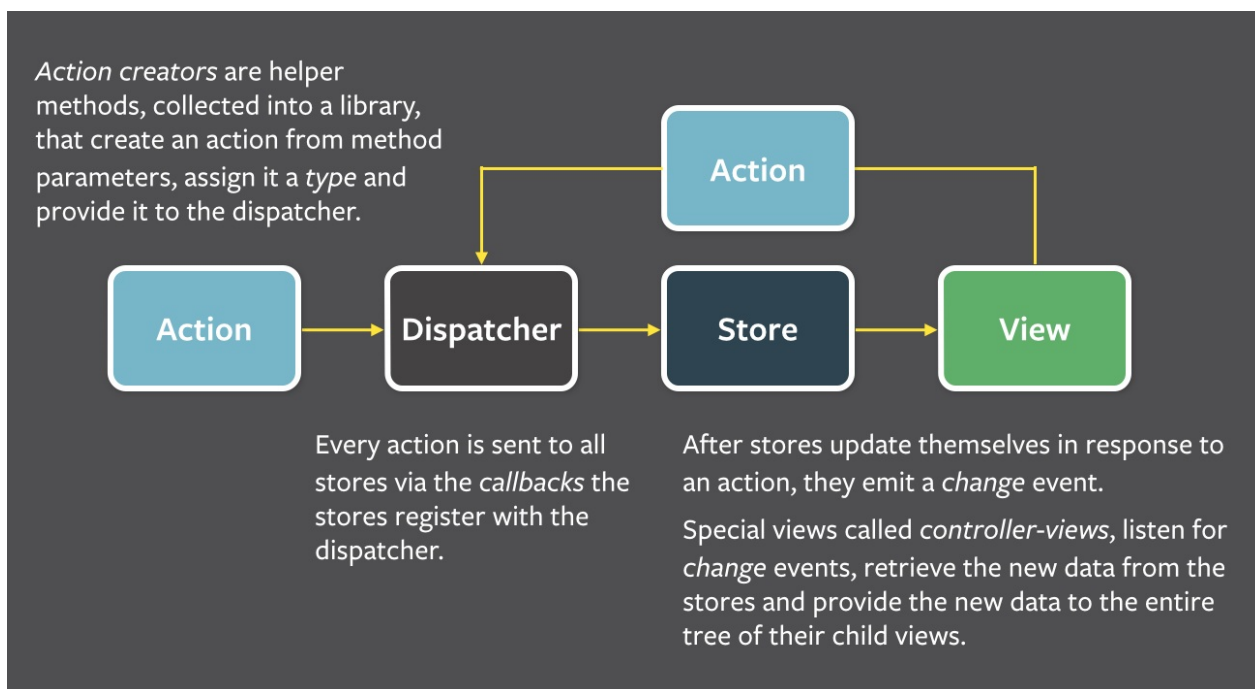


A unidirectional data flow is central to the Flux pattern, and the above diagram should be **the primary mental model for the Flux programmer**. The dispatcher, stores and views are independent nodes with distinct inputs and outputs. The actions are simple objects containing the new data and an identifying *type* property.

The views may cause a new action to be propagated through the system in response to user interactions:



All data flows through the dispatcher as a central hub. Actions are provided to the dispatcher in an *action creator* method, and most often originate from user interactions with the views. The dispatcher then invokes the callbacks that the stores have registered with it, dispatching actions to all stores. Within their registered callbacks, stores respond to whichever actions are relevant to the state they maintain. The stores then emit a *change* event to alert the controller-views that a change to the data layer has occurred. Controller-views listen for these events and retrieve data from the stores in an event handler. The controller-views call their own `setState()` method, causing a re-rendering of themselves and all of their descendants in the component tree.



This structure allows us to reason easily about our application in a way that is reminiscent of *functional reactive programming*, or more specifically *data-flow programming* or *flow-based programming*, where data flows through the application in a single direction — there are no two-way bindings. Application state is maintained only in the stores, allowing the different parts of the application to remain highly decoupled. Where dependencies do occur between stores, they are kept in a strict hierarchy, with synchronous updates managed by the dispatcher.

We found that two-way data bindings led to cascading updates, where changing one object led to another object changing, which could also trigger more updates. As applications grew, these cascading updates made it very difficult to predict what would change as the result of one user interaction. When updates can only change data within a single round, the system as a whole becomes more predictable.

Let's look at the various parts of Flux up close. A good place to start is the dispatcher.

A Single Dispatcher

The dispatcher is the central hub that manages all data flow in a Flux application. It is essentially a registry of callbacks into the stores and has no real intelligence of its own — it is a simple mechanism for distributing the actions to the stores. Each store registers itself and provides a callback. When an action creator provides the dispatcher with a new action, all stores in the application receive the action via the callbacks in the registry.

As an application grows, the dispatcher becomes more vital, as it can be used to manage dependencies between the stores by invoking the registered callbacks in a specific order. Stores can declaratively wait for other stores to finish updating, and then update themselves

accordingly.

The same dispatcher that Facebook uses in production is available through [npm](#), [Bower](#), or [GitHub](#).

Stores

Stores contain the application state and logic. Their role is somewhat similar to a model in a traditional MVC, but they manage the state of many objects — they do not represent a single record of data like ORM models do. Nor are they the same as Backbone's collections. More than simply managing a collection of ORM-style objects, stores manage the application state for a particular **domain** within the application.

For example, Facebook's [Lookback Video Editor](#) utilized a TimeStore that kept track of the playback time position and the playback state. On the other hand, the same application's ImageStore kept track of a collection of images. The TodoStore in our [TodoMVC example](#) is similar in that it manages a collection of to-do items. A store exhibits characteristics of both a collection of models and a singleton model of a logical domain.

As mentioned above, a store registers itself with the dispatcher and provides it with a callback. This callback receives the action as a parameter. Within the store's registered callback, a switch statement based on the action's type is used to interpret the action and to provide the proper hooks into the store's internal methods. This allows an action to result in an update to the state of the store, via the dispatcher. After the stores are updated, they broadcast an event declaring that their state has changed, so the views may query the new state and update themselves.

Views and Controller-Views

React provides the kind of composable and freely re-renderable views we need for the view layer. Close to the top of the nested view hierarchy, a special kind of view listens for events that are broadcast by the stores that it depends on. We call this a controller-view, as it provides the glue code to get the data from the stores and to pass this data down the chain of its descendants. We might have one of these controller-views governing any significant section of the page.

When it receives the event from the store, it first requests the new data it needs via the stores' public getter methods. It then calls its own `setState()` or `forceUpdate()` methods, causing its `render()` method and the `render()` method of all its descendants to run.

We often pass the entire state of the store down the chain of views in a single object, allowing different descendants to use what they need. In addition to keeping the controller-like behavior at the top of the hierarchy, and thus keeping our descendant views as

functionally pure as possible, passing down the entire state of the store in a single object also has the effect of reducing the number of props we need to manage.

Occasionally we may need to add additional controller-views deeper in the hierarchy to keep components simple. This might help us to better encapsulate a section of the hierarchy related to a specific data domain. Be aware, however, that controller-views deeper in the hierarchy can violate the singular flow of data by introducing a new, potentially conflicting entry point for the data flow. In making the decision of whether to add a deep controller-view, balance the gain of simpler components against the complexity of multiple data updates flowing into the hierarchy at different points. These multiple data updates can lead to odd effects, with React's render method getting invoked repeatedly by updates from different controller-views, potentially increasing the difficulty of debugging.

Actions

The dispatcher exposes a method that allows us to trigger a dispatch to the stores, and to include a payload of data, which we call an action. The action's creation may be wrapped into a semantic helper method which sends the action to the dispatcher. For example, we may want to change the text of a to-do item in a to-do list application. We would create an action with a function signature like `updateText(todoId, newText)` in our `TodoActions` module. This method may be invoked from within our views' event handlers, so we can call it in response to a user interaction. This action creator method also adds a *type* to the action, so that when the action is interpreted in the store, it can respond appropriately. In our example, this type might be named something like `TODO_UPDATE_TEXT`.

Actions may also come from other places, such as the server. This happens, for example, during data initialization. It may also happen when the server returns an error code or when the server has updates to provide to the application.

What About that Dispatcher?

As mentioned earlier, the dispatcher is also able to manage dependencies between stores. This functionality is available through the `waitFor()` method within the Dispatcher class. We did not need to use this method within the extremely simple [TodoMVC application](#), but it becomes vital in a larger, more complex application.

Within the `TodoStore`'s registered callback we could explicitly wait for any dependencies to first update before moving forward:

```
case 'TODO_CREATE':
  Dispatcher.waitFor([
    PrependedTextStore.dispatchToken,
    YetAnotherStore.dispatchToken
  ]);

  TodoStore.create(PrependedTextStore.getText() + ' ' + action.text);
  break;
```

`waitFor()` accepts a single argument which is an array of dispatcher registry indexes, often called *dispatch tokens*. Thus the store that is invoking `waitFor()` can depend on the state of another store to inform how it should update its own state.

A dispatch token is returned by `register()` when registering callbacks for the Dispatcher:

```
PrependedTextStore.dispatchToken = Dispatcher.register(function (payload) {
  // ...
});
```

For more on `waitFor()`, actions, action creators and the dispatcher, please see [Flux: Actions and the Dispatcher](#).

TodoMVC 教程

To demonstrate the Flux architecture with some example code, let's take on the classic TodoMVC application. The entire application is available in the React GitHub repo within the [flux-todomvc](#) example directory, but let's walk through the development of it a step at a time.

To begin, we'll need some boilerplate and get up and running with a module system. Node's module system, based on CommonJS, will fit the bill very nicely and we can build off of [react-boilerplate](#) to get up and running quickly. Assuming you have npm installed, simply clone the react-boilerplate code from GitHub, and navigate into the resulting directory in Terminal (or whatever CLI application you like). Next run the npm scripts to get up and running: `npm install`, then `npm run build`, and lastly `npm start` to continuously build using Browserify.

The TodoMVC example has all this built into it as well, but if you're starting with react-boilerplate make sure you edit your package.json file to match the file structure and dependencies described in the TodoMVC example's [package.json](#), or else your code won't match up with the explanations below.

Source Code Structure

The index.html file may be used as the entry point into our app which loads the resulting bundle.js file, but we'll put most of our code in a 'js' directory. Let's let Browserify do its thing, and now we'll open a new tab in Terminal (or a GUI file browser) to look at the directory. It should look something like this:

```
myapp
|
+ ...
+ js
  |
  + app.js
  + bundle.js // generated by Browserify whenever we make changes.
+ index.html
+ ...
```

Next we'll dive into the js directory, and layout our application's primary directory structure:

```
myapp
|
+ ...
+ js
  |
  + actions
  + components // all React components, both views and controller-views
  + constants
  + dispatcher
  + stores
  + app.js
  + bundle.js
+ index.html
+ ...
```

Creating a Dispatcher

Now we are ready to create a dispatcher. The same dispatcher that Facebook uses in production is available through [npm](#), [Bower](#), or [GitHub](#). It provides you with a default implementation, `Dispatcher`. All we need to do is to instantiate the Dispatcher and export it as a singleton:

```
var Dispatcher = require('flux').Dispatcher;

module.exports = new Dispatcher();
```

The public API of the dispatcher consists of two main methods: `register()` and `dispatch()`. We'll use `register()` within our stores to register each store's callback. We'll use `dispatch()` within our actions to trigger the invocation of the callbacks.

Creating Stores

A store updates itself in response to some action, and should emit a change event when done with it. We thus can use Node's `EventEmitter` to get started with a store: we will use `EventEmitter` to broadcast the 'change' event to our controller-views so that they can re-render, if needed. So let's take a look at what that looks like. I've omitted some of the code for the sake of brevity, but for the full version see [TodoStore.js](#) in the TodoMVC example code.

```
var AppDispatcher = require('../dispatcher/AppDispatcher');
var EventEmitter = require('events').EventEmitter;
var TodoConstants = require('../constants/TodoConstants');
```

```

var assign = require('object-assign');

var CHANGE_EVENT = 'change';

var _todos = {}; // collection of todo items

/**
 * Create a TODO item.
 * @param {string} text The content of the TODO
 */
function create(text) {
  // Using the current timestamp in place of a real id.
  var id = Date.now();
  _todos[id] = {
    id: id,
    complete: false,
    text: text
  };
}

/**
 * Delete a TODO item.
 * @param {string} id
 */
function destroy(id) {
  delete _todos[id];
}

var TodoStore = assign({}, EventEmitter.prototype, {

  /**
   * Get the entire collection of TODOs.
   * @return {object}
   */
  getAll: function() {
    return _todos;
  },

  emitChange: function() {
    this.emit(CHANGE_EVENT);
  },

  /**
   * Controller-views will use this to listen for any changes on the store and,
   * maybe, re-render themselves.
   * @param {function} callback
   */
  addChangeListener: function(callback) {
    this.on(CHANGE_EVENT, callback);
  },

  /**
   * Controller-views will use this to stop listening to the store.

```

```
* @param {function} callback
*/
removeChangeListener: function(callback) {
  this.removeListener(CHANGE_EVENT, callback);
}
});

// Register a callback to handle all updates.
AppDispatcher.register(function(action) {
  var text;

  switch(action.actionType) {
    case TodoConstants.TODO_CREATE:
      text = action.text.trim();
      if (text !== '') {
        create(text);
      }
      TodoStore.emitChange();
      break;

    case TodoConstants.TODO_DESTROY:
      destroy(action.id);
      TodoStore.emitChange();
      break;

    // add more cases for other actionTypes, like TODO_UPDATE, etc.

    default:
      // no op
  }
})

module.exports = TodoStore;
```

There are a few important things to note in the above code. To start, we are maintaining a private data structure called `_todos`. This object contains all the individual to-do items. Because this variable lives outside the class, but within the closure of the module, it remains private — it cannot be directly changed from outside of the module. This helps us preserve a distinct input/output interface for the flow of data by making it impossible to update the store without using an action.

Another important part is the registration of the store's callback with the dispatcher. The callback function currently only handles two actionTypes, but later we can add as many as we need.

Listening to Changes with a Controller-View

We need a React component near the top of our component hierarchy to listen for changes in the store. In a larger app, we would have more of these listening components, perhaps one for every section of the page. In Facebook's Ads Creation Tool, we have many of these controller-like views, each governing a specific section of the UI. In the Lookback Video Editor, we only had two: one for the animated preview and one for the image selection interface. Here's one for our TodoMVC example. Again, this is slightly abbreviated, but for the full code you can take a look at the TodoMVC example's [TodoApp.react.js](#)

```

var Footer = require('./Footer.react');
var Header = require('./Header.react');
var MainSection = require('./MainSection.react');
var React = require('react');
var TodoStore = require('../stores/TodoStore');

/**
 * Retrieve the current TODO data from the TodoStore.
 */
function getTodoState() {
  return {
    allTodos: TodoStore.getAll()
  };
}

var TodoApp = React.createClass({

  getInitialState: function() {
    return getTodoState();
  },

  componentDidMount: function() {
    TodoStore.addChangeListener(this._onChange);
  },

  componentWillUnmount: function() {
    TodoStore.removeChangeListener(this._onChange);
  },

  /**
   * @return {object}
   */
  render: function() {
    return (
      <div>
        <Header />
        <MainSection
          allTodos={this.state.allTodos}
        />
        <Footer allTodos={this.state.allTodos} />
      </div>
    );
  },

  _onChange: function() {
    this.setState(getTodoState());
  }

});

module.exports = TodoApp;

```

Now we're in our familiar React territory, utilizing React's lifecycle methods. We set up the initial state of this controller-view in `getInitialState()`, register an event listener in `componentDidMount()`, and then clean up after ourselves within `componentWillUnmount()`. We render a containing div and pass down the collection of states we got from the `TodoStore`.

The `Header` component contains the primary text input for the application, but it does not need to know the state of the store. The `MainSection` and `Footer` do need this data, so we pass it down to them.

More Views

At a high level, the React component hierarchy of the app looks like this:

```
<TodoApp>
  <Header>
    <TodoTextInput />
  </Header>

  <MainSection>
    <ul>
      <TodoItem />
    </ul>
  </MainSection>
</TodoApp>
```

If a `TodoItem` is in edit mode, it will also render a `TodoTextInput` as a child. Let's take a look at how some of these components display the data they receive as props, and how they communicate through actions with the dispatcher.

The `MainSection` needs to iterate over the collection of to-do items it received from `TodoApp` to create the list of `TodoItems`. In the component's `render()` method, we can do that iteration like so:

```
var allTodos = this.props.allTodos;

for (var key in allTodos) {
  todos.push(<TodoItem key={key} todo={allTodos[key]} />);
}

return (
  <section id="main">
    <ul id="todo-list">{todos}</ul>
  </section>
);
```

Now each TodoItem can display its own text, and perform actions utilizing its own ID. Explaining all the different actions that a TodoItem can invoke in the TodoMVC example goes beyond the scope of this article, but let's just take a look at the action that deletes one of the to-do items. Here is an abbreviated version of the TodoItem:

```
var React = require('react');
var ReactPropTypes = React.PropTypes;
var TodoActions = require('../actions/TodoActions');
var TodoTextInput = require('../TodoTextInput.react');

var TodoItem = React.createClass({

  propTypes: {
    todo: ReactPropTypes.object.isRequired
  },

  render: function() {
    var todo = this.props.todo;

    return (
      <li
        key={todo.id}>
        <label>
          {todo.text}
        </label>
        <button className="destroy" onClick={this._onDestroyClick} />
      </li>
    );
  },

  _onDestroyClick: function() {
    TodoActions.destroy(this.props.todo.id);
  }

});

module.exports = TodoItem;
```


With a destroy action available in our library of `TodoActions`, and a store ready to handle it, connecting the user's interaction with application state changes could not be simpler. We just wrap our `onClick` handler around the destroy action, provide it with the `id`, and we're done. Now the user can click the destroy button and kick off the Flux cycle to update the rest of the application.

Text input, on the other hand, is just a bit more complicated because we need to hang on to the state of the text input within the React component itself. Let's take a look at how `TodoTextInput` works.

As you'll see below, with every change to the input, React expects us to update the state of the component. So when we are finally ready to save the text inside the input, we will put the value held in the component's state in the action's payload. This is UI state, rather than application state, and keeping that distinction in mind is a good guide for where state should live. All application state should live in the store, while components occasionally hold on to UI state. Ideally, React components preserve as little state as possible.

Because `TodoTextInput` is being used in multiple places within our application, with different behaviors, we'll need to pass the `onSave` method in as a prop from the component's parent. This allows `onSave` to invoke different actions depending on where it is used.

```
var React = require('react');
var ReactPropTypes = React.PropTypes;

var ENTER_KEY_CODE = 13;

var TodoTextInput = React.createClass({

  propTypes: {
    className: ReactPropTypes.string,
    id: ReactPropTypes.string,
    placeholder: ReactPropTypes.string,
    onSave: ReactPropTypes.func.isRequired,
    value: ReactPropTypes.string
  },

  getInitialState: function() {
    return {
      value: this.props.value || ''
    };
  },

  /**
   * @return {object}
   */
  render: function() /*object*/ {
    return (
      <input
```

```

        className={this.props.className}
        id={this.props.id}
        placeholder={this.props.placeholder}
        onBlur={this._save}
        onChange={this._onChange}
        onKeyDown={this._onKeyDown}
        value={this.state.value}
        autoFocus={true}
      />
    );
  },

  /**
   * Invokes the callback passed in as onSave, allowing this component to be
   * used in different ways.
   */
  _save: function() {
    this.props.onSave(this.state.value);
    this.setState({
      value: ''
    });
  },

  /**
   * @param {object} event
   */
  _onChange: function(/*object*/ event) {
    this.setState({
      value: event.target.value
    });
  },

  /**
   * @param {object} event
   */
  _onKeyDown: function(event) {
    if (event.keyCode === ENTER_KEY_CODE) {
      this._save();
    }
  }
});

module.exports = TodoTextInput;

```

The Header passes in the onSave method as a prop to allow the TodoTextInput to create new to-do items:

```

var React = require('react');
var TodoActions = require('../actions/ToDoActions');
var TodoTextInput = require('../ToDoTextInput.react');

var Header = React.createClass({

  /**
   * @return {Object}
   */
  render: function() {
    return (
      <header id="header">
        <h1>todos</h1>
        <ToDoTextInput
          id="new-todo"
          placeholder="What needs to be done?"
          onSave={this._onSave}
        />
      </header>
    );
  },

  /**
   * Event handler called within ToDoTextInput.
   * Defining this here allows ToDoTextInput to be used in multiple places
   * in different ways.
   * @param {string} text
   */
  _onSave: function(text) {
    TodoActions.create(text);
  }

});

module.exports = Header;

```

In a different context, such as in editing mode for an existing to-do item, we might pass an `onSave` callback that invokes `TodoActions.update(text)` instead.

Creating Semantic Actions

Here is the basic code for the two actions we used above in our views:

```
/**
 * TodoActions
 */

var AppDispatcher = require('../dispatcher/AppDispatcher');
var TodoConstants = require('../constants/TodoConstants');

var TodoActions = {

  /**
   * @param {string} text
   */
  create: function(text) {
    AppDispatcher.dispatch({
      actionType: TodoConstants.TODO_CREATE,
      text: text
    });
  },

  /**
   * @param {string} id
   */
  destroy: function(id) {
    AppDispatcher.dispatch({
      actionType: TodoConstants.TODO_DESTROY,
      id: id
    });
  },

};

module.exports = TodoActions;
```

When the user creates a new to-do item, the payload produced by the `TodoActions.create()` will look like:

```
{
  actionType: 'TODO_CREATE',
  text: 'Write blog post about Flux'
}
```

Let's wrap it up. When the user validates the text input in the Header component, `TodoActions.create` is called with the text for the new-todo. A new action is created, with a payload containing both the text and the action type. This action is dispatched to the stores which registered to the dispatcher, through a callback mechanism. In response to the dispatched action, the `TodoStore` updates itself by creating a new todo-item, then emits a 'change' event. The controller-view `TodoApp`, which is the root React component in this application, is listening for such events broadcasted by the store. It responds to the 'change'

event by fetching the new collection of to-do items from the TodoStore and changes its state. React kicks in: this change in state automatically causes the TodoApp component to call its own render() method, and the render() method of all of its owned components. Any relevant, required updates to the DOM are performed by React at the end of this unidirectional "Flux chain".

Start Me Up

The TodoApp component has still to be created. The bootstrap file of our application will be app.js. It simply takes the TodoApp component and renders it in the root element of the application.

```
var React = require('react');

var TodoApp = require('./components/TodoApp.react');

React.render(
  <TodoApp />,
  document.getElementById('todoapp')
);
```

Adding Dependency Management to the Dispatcher

As I said previously, our Dispatcher implementation is a bit naive. It's pretty good, but it will not suffice for most applications. We need a way to be able to manage dependencies between Stores. Let's add that functionality with a waitFor() method within the main body of the Dispatcher class.

We'll need another public method, waitFor(). Note that it returns a Promise that can in turn be returned from the Store callback.

```
/**
 * @param {array} promiseIndexes
 * @param {function} callback
 */
waitFor: function(promiseIndexes, callback) {
  var selectedPromises = promiseIndexes.map(function(index) {
    return _promises[index];
  });
  return Promise.all(selectedPromises).then(callback);
}
```

Now within the `TodoStore` callback we can explicitly wait for any dependencies to first update before moving forward. However, if Store A waits for Store B, and B waits for A, then a circular dependency will occur. A more robust dispatcher is required to flag this scenario with warnings in the console.

The Future of Flux

A lot of people ask if Facebook will release Flux as an open source framework. Really, Flux is just an architecture, not a framework, but we have released `flux/utils` which is a small set of utilities we use at Facebook to build Flux applications. There are also many other great frameworks out there that enable the Flux architecture.

Thanks for taking the time to read about how we build client-side applications at Facebook. We hope Flux proves as useful to you as it has to us.

TIPS

- [Introduction](#)
- [Inline Styles](#)
- [If-Else in JSX](#)
- [Self-Closing Tag](#)
- [Maximum Number of JSX Root Nodes](#)
- [Shorthand for Specifying Pixel Values in style props](#)
- [Type of the Children props](#)
- [Value of null for Controlled Input](#)
- [componentWillReceiveProps Not Triggered After Mounting](#)
- [Props in getInitialState Is an Anti-Pattern](#)
- [DOM Event Listeners in a Component](#)
- [Load Initial Data via AJAX](#)
- [False in JSX](#)
- [Communicate Between Components](#)
- [Expose Component Functions](#)
- [this.props.children undefined](#)
- [Use React with Other Libraries](#)
- [Dangerously Set innerHTML](#)

Introduction

The React tips section provides bite-sized information that can answer lots of questions you might have and warn you against common pitfalls.

Contributing

Submit a pull request to the [React repository](#) following the [current tips](#) entries' style. If you have a recipe that needs review prior to submitting a PR you can find help in the [#reactjs channel on freenode](#) or the [discuss.reactjs.org forum](#).

Inline Styles

In React, inline styles are not specified as a string. Instead they are specified with an object whose key is the camelCased version of the style name, and whose value is the style's value, usually a string ([more on that later](#)):

```
var divStyle = {
  color: 'white',
  backgroundImage: 'url(' + imgUrl + ')',
  WebkitTransition: 'all', // note the capital 'W' here
  msTransition: 'all' // 'ms' is the only lowercase vendor prefix
};

ReactDOM.render(<div style={divStyle}>Hello World!</div>, mountNode);
```

Style keys are camelCased in order to be consistent with accessing the properties on DOM nodes from JS (e.g. `node.style.backgroundImage`). Vendor prefixes [other than](#) `ms` should begin with a capital letter. This is why `WebkitTransition` has an uppercase "W".

If-Else in JSX

`if-else` statements don't work inside JSX. This is because JSX is just syntactic sugar for function calls and object construction. Take this basic example:

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

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

This means that `if` statements don't fit in. Take this example:

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

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

That's not valid JS. You probably want to make use of a ternary expression:

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

If a ternary expression isn't robust enough, you can use `if` statements outside of your JSX to determine which components should be used:

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

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

Or if you prefer a more "inline" aesthetic, define [immediately-invoked function expressions](#) *inside* your JSX:

```
return (  
  <section>  
    <h1>Color</h1>  
    <h3>Name</h3>  
    <p>{this.state.color || "white"}</p>  
    <h3>Hex</h3>  
    <p>  
      {(() => {  
        switch (this.state.color) {  
          case "red":    return "#FF0000";  
          case "green":  return "#00FF00";  
          case "blue":   return "#0000FF";  
          default:       return "#FFFFFF";  
        }  
      })()}  
    </p>  
  </section>  
)  
);
```

Note:

In the example above, an ES6 [arrow function](#) is utilized to lexically bind the value of `this`.

Try using it today with the [Babel REPL](#).

Self-Closing Tag

In JSX, `<MyComponent />` alone is valid while `<MyComponent>` isn't. All tags must be closed, either with the self-closing format or with a corresponding closing tag (`</MyComponent>`).

Note:

Every React component can be self-closing: `<div />` . `<div></div>` is also an equivalent.

Maximum Number of JSX Root Nodes

Currently, in a component's `render`, you can only return one node; if you have, say, a list of `div`s to return, you must wrap your components within a `div`, `span` or any other component.

Don't forget that JSX compiles into regular JS; returning two functions doesn't really make syntactic sense. Likewise, don't put more than one child in a ternary.

Shorthand for Specifying Pixel Values in style props

When specifying a pixel value for your inline `style` prop, React automatically appends the string "px" for you after your number value, so this works:

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

See [Inline Styles](#) for more info.

Sometimes you *do* want to keep the CSS properties unitless. Here's a list of properties that won't get the automatic "px" suffix:

- `animationIterationCount`
- `boxFlex`
- `boxFlexGroup`
- `boxOrdinalGroup`
- `columnCount`
- `fillOpacity`
- `flex`
- `flexGrow`
- `flexPositive`
- `flexShrink`
- `flexNegative`
- `flexOrder`
- `fontWeight`
- `lineClamp`
- `lineHeight`
- `opacity`
- `order`
- `orphans`
- `stopOpacity`
- `strokeDashoffset`
- `strokeOpacity`
- `strokeWidth`
- `tabSize`
- `widows`
- `zIndex`

- `zoom`

Type of the Children props

Usually, a component's children (`this.props.children`) is an array of components:

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

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

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

However, when there is only a single child, `this.props.children` will be the single child component itself *without the array wrapper*. This saves an array allocation.

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

    // warning: yields 5 for length of the string 'hello', not 1 for the
    // length of the non-existent array wrapper!
    console.log(this.props.children.length);
  },

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

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

To make `this.props.children` easy to deal with, we've provided the [React.Children utilities](#).

Value of null for Controlled Input

Specifying the `value` prop on a [controlled component](#) prevents the user from changing the input unless you desire so.

You might have run into a problem where `value` is specified, but the input can still be changed without consent. In this case, you might have accidentally set `value` to `undefined` or `null`.

The snippet below shows this phenomenon; after a second, the text becomes editable.

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

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

componentWillReceiveProps Not Triggered After Mounting

`componentWillReceiveProps` isn't triggered after the node is put on scene. This is by design. Check out [other lifecycle methods](#) for the one that suits your needs.

The reason for that is because `componentWillReceiveProps` often handles the logic of comparing with the old props and acting upon changes; not triggering it at mounting (where there are no old props) helps in defining what the method does.

Props in `getInitialState` Is an Anti-Pattern

Note:

This isn't really a React-specific tip, as such anti-patterns often occur in code in general; in this case, React simply points them out more clearly.

Using props to generate state in `getInitialState` often leads to duplication of "source of truth", i.e. where the real data is. This is because `getInitialState` is only invoked when the component is first created.

Whenever possible, compute values on-the-fly to ensure that they don't get out of sync later on and cause maintenance trouble.

Bad example:

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

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

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

Better:

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

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

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

However, it's **not** an anti-pattern if you make it clear that the prop is only seed data for the component's internally-controlled state:

```
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>;
  }
});

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

DOM Event Listeners in a Component

Note:

This entry shows how to attach DOM events not provided by React ([check here for more info](#)). This is good for integrations with other libraries such as jQuery.

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>;
  }
});

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

`componentDidMount` is called after the component is mounted and has a DOM representation. This is often a place where you would attach generic DOM events.

Notice that the event callback is bound to the react component and not the original element. React automatically binds methods to the current component instance for you through a process of [autobinding](#).

Load Initial Data via AJAX

Fetch data in `componentDidMount`. When the response arrives, store the data in state, triggering a render to update your UI.

When fetching data asynchronously, use `componentWillUnmount` to cancel any outstanding requests before the component is unmounted.

This example fetches the desired Github user's latest gist:

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

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

  componentWillUnmount: function() {
    this.serverRequest.abort();
  },

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

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


False in JSX

Here's how `false` renders in different situations:

Renders as `id="false"` :

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

String `"false"` as input value:

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

No child:

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

The reason why this one doesn't render as the string `"false"` as a `div` child is to allow the more common use-case: `<div>{x > 1 && 'You have more than one item'}</div>` .

Communicate Between Components

For parent-child communication, simply [pass props](#).

For child-parent communication: Say your `GroceryList` component has a list of items generated through an array. When a list item is clicked, you want to display its name:

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

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

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

Notice the use of `bind(this, arg1, arg2, ...)` : we're simply passing more arguments to `handleClick` . This is not a new React concept; it's just JavaScript.

For communication between two components that don't have a parent-child relationship, you can set up your own global event system. Subscribe to events in `componentDidMount()` , unsubscribe in `componentWillUnmount()` , and call `setState()` when you receive an event. [Flux](#) pattern is one of the possible ways to arrange this.

Expose Component Functions

There's another (uncommon) way of [communicating between components](#): simply expose a method on the child component for the parent to call.

Say a list of todos, which upon clicking get removed. If there's only one unfinished todo left, animate it:

```

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>
    );
  }
});

ReactDOM.render(<Todos />, mountNode);

```

Alternatively, you could have achieved this by passing the `todo` and `isLastUnfinishedItem` prop, let it check this prop in `componentDidUpdate`, then animate itself; however, this quickly gets messy if you pass around different props to control animations.

this.props.children undefined

You can't access the children of your component through `this.props.children`.

`this.props.children` designates the children being **passed onto you** by the owner:

```
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>;
  }
});

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

To access your own subcomponents (the `span` s), place [refs](#) on them.

Use React with Other Libraries

You don't have to go full React. The component [lifecycle events](#), especially `componentDidMount` and `componentDidUpdate`, are good places to put your other libraries' logic.

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

  componentDidMount: function() {
    $(this.refs.placeholder).append('<span />');
  },

  componentWillUnmount: function() {
    // Clean up work here.
  },

  shouldComponentUpdate: function() {
    // Let's just never update this component again.
    return false;
  },

  render: function() {
    return <div ref="placeholder"/>;
  }
});

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

You can attach your own [event listeners](#) and even [event streams](#) this way.

Dangerously Set innerHTML

Improper use of the `innerHTML` can open you up to a [cross-site scripting \(XSS\)](#) attack. Sanitizing user input for display is notoriously error-prone, and failure to properly sanitize is one of the [leading causes of web vulnerabilities](#) on the internet.

Our design philosophy is that it should be “easy” to make things safe, and developers should explicitly state their intent when performing “unsafe” operations. The prop name `dangerouslySetInnerHTML` is intentionally chosen to be frightening, and the prop value (an object instead of a string) can be used to indicate sanitized data.

After fully understanding the security ramifications and properly sanitizing the data, create a new object containing only the key `__html` and your sanitized data as the value. Here is an example using the JSX syntax:

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

The point being that if you unintentionally do say `<div dangerouslySetInnerHTML={getUsername()} />` it will not be rendered because `getUsername()` would return a plain string and not a `{__html: ''}` object. The intent behind the `{__html:...}` syntax is that it be considered a “type/taint” of sorts. Sanitized data can be returned from a function using this wrapper object, and this marked data can subsequently be passed into `dangerouslySetInnerHTML`. For this reason, we recommend against writing code of the form `<div dangerouslySetInnerHTML={{'{'__html: getMarkup()}} />`.

This functionality is mainly provided for cooperation with DOM string manipulation libraries, so the HTML provided must be well-formed (ie., pass XML validation).

For a more complete usage example, refer to the last example on the [front page](#).