1.顶层 API

1.1 React

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

1.1.1React.createClass

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

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

更多关于定义组件对象的信息，参考组件定义和生命周期。

1.1.2 React.createElement

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

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

1.1.3 React.createFactory

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

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

1.1.4 React.render

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

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

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

如果提供了可选的回调函数，则该函数将会在组件渲染或者更新之后调用。

注意：

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

1.1.5 React.unmountComponentAtNode

boolean unmountComponentAtNode(DOMElement container)

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

1.1.6 React.renderToString

string renderToString(ReactElement element)

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

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

1.1.7 React.renderToStaticMarkup

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

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

1.1.8 React.isValidElement

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

判断对象是否是一个 ReactElement。

1.1.9 React.DOM

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

1.1.10 React.PropTypes

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

1.1.11 React.initializeTouchEvents

initializeTouchEvents(boolean shouldUseTouch)

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

1.1.12 React.Children

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

React.Children.map

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

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

React.Children.forEach

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

类似于 React.Children.map()，但是不返回对象。

React.Children.count

number React.Children.count(object children)

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

React.Children.only

object React.Children.only(object children)

返回 children 中仅有的子级。否则抛出异常。

2.组件 API

2.1ReactComponent

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

2.1.1 setState

setState(object nextState[, function callback])

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

注意：

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

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

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

setState() 将总是触发一次重绘，除非在 shouldComponentUpdate() 中实现了条件渲染逻辑。如果使用可变的对象，但是又不能

在 shouldComponentUpdate() 中实现这种逻辑，仅在新 state 和之前的 state 存在差异的时候调用 setState() 可以避免不必要的重新渲染。

2.1.2 replaceState

replaceState(object nextState[, function callback])

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

2.1.3 forceUpdate()

forceUpdate([function callback])

如果 render() 方法从 this.props 或者 this.state 之外的地方读取数据，你需要通过调用 forceUpdate() 告诉 React 什么时候需要再次运

行 render()。如果直接改变了 this.state，也需要调用 forceUpdate()。

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

通常情况下，应该尽量避免所有使用 forceUpdate() 的情况，

在 render() 中仅从 this.props 和 this.state 中读取数据。这会使应用大大简化，并且更加高效。

2.1.4 getDOMNode

DOMElement getDOMNode()

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

2.1.5 isMounted()

bool isMounted()

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

2.1.6 setProps

setProps(object nextProps[, function callback])

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

尽管在同一个节点上再次调用 React.render() 来更新根组件是首选的方式，也可以调用 setProps() 来改变组件的属性，触发一次重新渲染。另

外，可以传递一个可选的回调函数，该函数将会在 setProps 完成并且组件重新渲染完成之后调用。

注意：

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

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

2.1.7 replaceProps

replaceProps(object nextProps[, function callback])

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

3.组件的详细说明和生命周期（Component Specs and Lifecycle）

3.1 组件的详细说明（Component Specifications）

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

3.1.1 render

ReactComponent render()

render() 方法是必须的。

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

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

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

3.1.2 getInitialState

object getInitialState()

在组件挂载之前调用一次。返回值将会作为 this.state 的初始值。

3.1.3 getDefaultProps

object getDefaultProps()

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

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

3.1.4 propTypes

object propTypes

propTypes 对象允许验证传入到组件的 props。更多关于 propTypes 的信息，参考可重用的组件。

3.1.5 mixins

array mixins

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

3.1.6 statics

object statics

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

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

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

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

3.1.7 displayName

string displayName

displayName 字符串用于输出调试信息。JSX 自动设置该值；参考 JSX 深入。

3.2 生命周期方法

许多方法在组件生命周期中某个确定的时间点执行。

3.2.1 挂载： componentWillMount

componentWillMount()

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

3.2.2 挂载： componentDidMount

componentDidMount()

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

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

注意：

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

3.2.3 更新： componentWillReceiveProps

componentWillReceiveProps(object nextProps)

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

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

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

注意：

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

3.2.4 更新： shouldComponentUpdate

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

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

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

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

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

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

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

3.2.5 更新： componentWillUpdate

componentWillUpdate(object nextProps, object nextState)

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

使用该方法做一些更新之前的准备工作。

注意：

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

3.2.6 更新： componentDidUpdate

componentDidUpdate(object prevProps, object prevState)

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

使用该方法可以在组件更新之后操作 DOM 元素。

注意：

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

3.2.7 移除： componentWillUnmount

componentWillUnmount()

在组件从 DOM 中移除的时候立刻被调用。

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

4. 标签和属性支持

4.1 支持的标签

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

4.1.1 HTML 元素

下列的 HTML 元素是被支持的：

a abbr address area article aside audio b base bdi bdo big blockquote body br

button canvas caption cite code col colgroup data datalist dd del details dfn

dialog div dl dt em embed fieldset figcaption figure footer form h1 h2 h3 h4 h5

h6 head header hr html i iframe img input ins kbd keygen label legend li link

main map mark menu menuitem meta meter nav noscript object ol optgroup option

output p param picture pre progress q rp rt ruby s samp script section select

small source span strong style sub summary sup table tbody td textarea tfoot th

thead time title tr track u ul var video wbr

4.1.2 SVG 元素

下列的 SVG 元素是被支持的：

circle defs ellipse g line linearGradient mask path pattern polygon polyline

radialGradient rect stop svg text tspan

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

4.2 支持的属性

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

注意：

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

对于支持的事件列表，参考支持的事件。

4.2.1 HTML 属性

这些标准的属性是被支持的：

accept acceptCharset accessKey action allowFullScreen allowTransparency alt

async autoComplete autoPlay cellPadding cellSpacing charSet checked classID

className cols colSpan content contentEditable contextMenu controls coords

crossOrigin data dateTime defer dir disabled download draggable encType form

formAction formEncType formMethod formNoValidate formTarget frameBorder height

hidden href hrefLang htmlFor httpEquiv icon id label lang list loop manifest

marginHeight marginWidth max maxLength media mediaGroup method min multiple

muted name noValidate open pattern placeholder poster preload radioGroup

readOnly rel required role rows rowSpan sandbox scope scrolling seamless

selected shape size sizes span spellCheck src srcDoc srcSet start step style

tabIndex target title type useMap value width wmode

另外，下面非标准的属性也是被支持的：

- autoCapitalize autoCorrect 用于移动端的 Safari。
- property 用于 Open Graph 原标签。
- itemProp itemScope itemType 用于 HTML5 microdata。

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

4.2.2 SVG 属性

cx cy d dx dy fill fillOpacity fontFamily fontSize fx fy gradientTransform

gradientUnits markerEnd markerMid markerStart offset opacity

patternContentUnits patternUnits points preserveAspectRatio r rx ry

spreadMethod stopColor stopOpacity stroke strokeDasharray strokeLinecap

strokeOpacity strokeWidth textAnchor transform version viewBox x1 x2 x y1 y2 y

5.事件系统

5.1 虚拟事件对象

事件处理器将会传入虚拟事件对象的实例，一个对浏览器本地事件的跨浏览器封装。它有和浏览器本地事件相同的属性和方法，包括 stopPropagation() 和 preventDefault()，但是没有浏览器兼容问题。如果因为一些因素，需要底层的浏览器事件对象，只要使用 nativeEvent 属性就可以获取到它了。每一个虚拟事件对象都有下列的属性：

boolean bubbles

boolean cancelable

DOMEventTarget currentTarget

boolean defaultPrevented

number eventPhase

boolean isTrusted

DOMEvent nativeEvent

void preventDefault()

void stopPropagation()

DOMEventTarget target

number timeStamp

string type

注意：

对于 v0.12，在事件处理函数中返回 false 将不会阻止事件冒泡。取而代之的是在合适的应用场景下，手动调用 e.stopPropagation() 或者 e.preventDefault()。

5.2 支持的事件

React 标准化了事件对象，因此在不同的浏览器中都会有相同的属性。

如下的事件处理器在事件冒泡阶段触发。要在捕获阶段触发某个事件处理器，在事件名字后面追加 Capture 字符串；例如，使用 onClickCapture 而不是 onClick 来在捕获阶段处理点击事件。

5.2.1 剪贴板事件

事件名：

onCopy onCut onPaste

属性：

DOMDataTransfer clipboardData

5.2.2 键盘事件：

事件名：

onKeyDown onKeyPress onKeyUp

属性：

boolean altKey

Number charCode

boolean ctrlKey

function getModifierState(key)

String key

Number keyCode

String locale

Number location

boolean metaKey

boolean repeat

boolean shiftKey

Number which

5.2.3 焦点事件

事件名：

onFocus onBlur

属性：

DOMEventTarget relatedTarget

5.2.4 表单事件

事件名：

onChange onInput onSubmit

更多关于 onChange 事件的信息，参考表单。

5.2.5 鼠标事件

事件名：

onClick onDoubleClick onDrag onDragEnd onDragEnter onDragExit onDragLeave

onDragOver onDragStart onDrop onMouseDown onMouseEnter onMouseLeave

onMouseMove onMouseOut onMouseOver onMouseUp

属性：

boolean altKey

Number button

Number buttons

Number clientX

Number clientY

boolean ctrlKey

function getModifierState(key)

boolean metaKey

Number pageX

Number pageY

DOMEventTarget relatedTarget

Number screenX

Number screenY

boolean shiftKey

5.2.6 触摸事件

为了使触摸事件生效，在渲染所有组件之前调

用 React.initializeTouchEvents(true)。

事件名：

onTouchCancel onTouchEnd onTouchMove onTouchStart

属性：

boolean altKey

DOMTouchList changedTouches

boolean ctrlKey

function getModifierState(key)

boolean metaKey

boolean shiftKey

DOMTouchList targetTouches

DOMTouchList touches

5.2.7UI 事件

事件名：

onScroll

属性：

Number detail

DOMAbstractView view

5.2.8 鼠标滚轮滚动事件

事件名：

onWheel

属性：

Number deltaMode

Number deltaX

Number deltaY

Number deltaZ

6.与 DOM 的差异

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

- 所有 DOM 的 properties 和 attributes （包括事件处理器）应该都是驼峰命名的，以便和标准的 JavaScript 风格保持一致。我们故意和规范不同，因为规范本身就不一致。**然而**，data-* 和 aria-* 属性符合规范，应该仅是小写的。

- style 属性接收一个带有驼峰命名风格的 JavaScript 对象，而不是一个 CSS 字符串。这与 DOM 中的 style 的 JavaScript 属性保持一致，更加有效，并且弥补了 XSS 安全漏洞。
- 所有的事件对象和 W3C 规范保持一致，并且所有的事件（包括提交事件）冒泡都正确地遵循 W3C 规范。参考事件系统获取更多详细信息。
- onChange 事件表现得和你想要的一样：当表单字段改变了，该事件就被触发，而不是等到失去焦点的时候。我们故意和现有的浏览器表现得不一致，是因为 onChange 是它的行为的一个错误称呼，并且 React 依赖于此事件来实时地响应用户输入。参考表单获取更多详细信息。
- 表单输入属性，例如 value 和 checked，以及 textarea。这里有更多相关信息。

7.特殊的非 DOM 属性

除了与 DOM 的差异之外，React 也提供了一些 DOM 里面不存在的属性。

- key：可选的唯一的标识器。当组件在渲染过程中被各种打乱的时候，由于差异检测逻辑，可能会被销毁后重新创建。给组件绑定一个 key，可以持续确保组件还存在 DOM 中。更多内容请参考这里。

- ref：参考这里。
- dangerouslySetInnerHTML：提供插入纯 HTML 字符串的功能，主要为了能和生成 DOM 字符串的库整合。更多内容请参考这里。

8.Reconciliation

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

8.1 动机（Motivation）

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

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

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

1、拥有相同类的两个组件将会生成相似的树形结构，拥有不同类的两个组件将会生成不同的树形结构。

2、可以为元素提供一个唯一的标志，该元素在不同的渲染过程中保持不变。

实际上，这些假设会使在几乎所有的应用场景下，应用变得出奇地快。

8.2 两个节点的差异检查（Pair-wise diff）

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

8.2.1 不同的节点类型

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

renderA:

renderB:

=> [removeNode], [insertNode]

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

renderA: <Header />

renderB: <Content />

=> [removeNode <Header />], [insertNode <Content />]

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

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

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

8.2.2 DOM 节点

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

renderA:

renderB:

=> [replaceAttribute id "after"]

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

```
renderA: <div style={{color: 'red'}} />

renderB: <div style={{fontWeight: 'bold'}} />

=> [removeStyle color], [addStyle font-weight 'bold']
```

在属性更新完毕之后，递归检测所有的子级的属性。

8.2.3 自定义组件

我们决定两个自定义组件是相同的。因为组件是状态化的，不可能每次状态改变都要创建一个新的组件实例。React 利用新组件上的所有属性，然后在之前的组件实例上调用 component[Will/Did]ReceiveProps()。现在，之前的组件就是可操作了的。它的 render() 方法被调用，然后差异算法重新比较新的状态和上一次的状态。

8.3 子级优化差异算法（List-wise diff）

8.3.1 问题点（Problematic Case）

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

例如在末尾添加一个元素：

```
renderA: <div><span>first</span></div>

renderB: <div><span>first</span><span>second</span></div>

=> [insertNode <span>second</span>]
```

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

renderA: first

renderB: secondfirst

=> [replaceAttribute textContent 'second'], [insertNode first]

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

8.3.2 键（Keys）

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

renderA: first

renderB: second<span

key="first">first

=> [insertNode second]

在实际开发中，生成一个键值不是很困难。大多数时候，要展示的元素已经有一个唯一的标识了。当没有唯一标识的时候，可以给组件模型添

加一个新的 ID 属性，或者计算部分内容的哈希值来生成一个键值。记住，键值仅需要在兄弟节点中唯一，而不是全局唯一。

8.4 权衡（Trade-offs）

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

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

由于依赖于两个预判条件，如果这两个条件都没有满足，性能将会大打折扣。

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

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

9. React （虚拟）DOM 术语 Edit on GitHub

在 React 的术语中，有五个核心类型，区分它们是很重要的：

- ReactElement / ReactElement 工厂
- ReactNode
- ReactComponent / ReactComponent 类

9.1React 元素

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

可以通过 React.createElement 创建该类型的一个实例。

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

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

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

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

```
var child = React.createElement('li', null, 'Text Content');

var root = React.createElement('ul', { className: 'my-list' }, child);

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

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

```
var root = <ul className="my-list">

                <li>Text Content</li>

            </ul>;

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

工厂

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

```
function createFactory(type){

  return React.createElement.bind(null, type);

}
```

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

```
var div = React.createFactory('div');

var root = div({ className: 'my-div' });

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

React 已经内置了常用 HTML 标签的工厂函数：

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

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

9.2React 节点

一个 ReactNode 可以是：

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

这些被用作其它 ReactElement 实例的属性，用于表示子级。实际上它们创建了一个 ReactElement 实例树。（These are used as properties of other ReactElements to represent children. Effectively they create a tree of ReactElements.）

9.3React 组件

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

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

```
var MyComponent = React.createClass({
```

```
  render: function() {
    ...
  }
});
```

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

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

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

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

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

或者使用 JSX：

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

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

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

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

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

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

```
componentA === componentB; // true
```

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

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

9.4 正式的类型定义

入口点（Entry Point）

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

节点和元素（Nodes and Elements）

```
type ReactNode = ReactElement | ReactFragment | ReactText;


type ReactElement = ReactComponentElement | ReactDOMElement;


type ReactDOMElement = {
  type : string,
  props : {
```

```
        children : ReactNodeList,
        className : string,
        etc.
    },
    key : string | boolean | number | null,
    ref : string | null
};

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

type ReactFragment = Array<ReactNode | ReactEmpty>;

type ReactNodeList = ReactNode | ReactEmpty;

type ReactText = string | number;

type ReactEmpty = null | undefined | boolean;
```

类和组件（Classes and Components）

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



type ReactComponent<TProps> = {

  props : TProps,

  render : () => ReactElement

};
```