React16 定义组件新特性

作者	日期
陈兴旭	2019.8.2

一、定义组件

使用es6的class定义组件，通过React.Component或React.PureComponent。定义无状态的函数式组件，通过React.memo。

1. React.Component

这是使用ES6 classes方式定义React组件时的基类。

2. React.PureCompnent

它是React15.3中新加的，与React.Component很相似,但两者的区别在于React.Component并未实现shouldComponentUpdate()，而React.PureComponent中实现了该函数，它会对props和state进行浅层比较(会比较 Object.keys(state | props) 的长度是否一致，每一个属性是否两者都有，并且是否是一个引用)，并减少了跳过必要更新的可能性。

shouldComponentUpdate()浅层比较源代码:

```
export function PureComponent(constructor: Function) {
    constructor.prototype.shouldComponentUpdate = function (nextProps, nextState) {
        return !shallowEqual(nextProps, this.props) || !shallowEqual(nextState, this.state);
    }
```

```
}
export function shallowEqual(objA: any, objB: any): boolean {
    if (objA === objB) {
        return true;
    } else {
        if (
            typeof objA !== 'object' || objA === null ||
            typeof objB !== 'object' || objB === null
        ) {
            return false;
        }

        const keysA = Object.keys(objA);
        const keysB = Object.keys(objB);

        if (keysA.length !== keysB.length) {
            return false;
        }
        // Test for A's keys different from B.
        for (let i = 0; i < keysA.length; i++) {
            if (
                !hasOwnProperty.call(objB, keysA[i]) ||
                !(objA[keysA[i]] === objB[keysA[i]])
            ) {
                return false;
            }
        }
        return true;
    }
}
```

如果赋予React组件相同的props和state，render()函数就会渲染相同的内容，那么在某些情况（如果更新的props和旧的一样，这个时候很明显UI不会变化，但是React还是要进行虚拟DOM的diff，这个diff就是多余的性能损耗，而且在DOM结构比较复杂的情况，整个diff会花费较长的时间）下使用React.PureComponent就可以通过记忆组件渲染结果的方式来提高组件的性能。

shouldComponentUpdate()仅作为性能优化的方式而存在。React默认在每一次state或者props改变之后进行渲染，会在每一次新的props或者state接受之前被调用，但是页面第一次渲染或者forceUpdate()的时候不会被调用。

不要企图通过实现shouldComponentUpdate()返回false来"阻止渲染"，因为这可能会产生bug，应该考虑使用内置的PureComponent组件，而不是手动编写shouldComponentUpdate()。如果一定要手动编写此函数，可将this.props与nextProps以及this.state与nextState进行比较，并在得到props和state没有变化的情况下返回false以告知React可以跳过更新，但返回false并不会阻止子组件在state更改时重新渲染。官方建议不要在shouldComponentUpdate()中进行深层比较或使用JSON.stringify(),因为这样非常影响效率，且会损害性能。后续版本，React 可能会将 shouldComponentUpdate 视为提示而不是严格的指令，并且，当返回 false 时，仍可能导致组件重新渲染。

3. React.memo

```
const  ErrorMessage = React.memo(
```

```
  function ErrorMessage({ errorType, onConfirm = noop }: Console.DisableUser.ErrorMessage.Props)
{
    switch (errorType) {
        case Status.CURRENT_USER_INCLUDED:
            return (
                <MessageDialog onConfirm={ onConfirm }>
                    {__('您无法禁用自身账号。')}
                </MessageDialog>
            );
        default:
            return (
                <MessageDialog onConfirm={onConfirm}>
                    {getErrorMessage(errorType)}
                </MessageDialog>
            )
    }
})
export default ErrorMessage;
```

它是高级组件（参数为组件，返回值为新组件的函数）。与React.PureComponent非常相似，但适用于函数组件，不适用于class组件。

通过将其包装在React.memo中调用，如果函数组件在给定相同props的情况下渲染相同的结果。那么就可以通过记忆组件渲染结果的方式来提高组件的性能表现。这意味着在这种情况下，React将跳过渲染组件的操作并直接复用最近一次渲染的结果。

默认情况下其只会对复杂对象做浅层对比，如果想要控制对比过程，那么请将自定义的比较函数通过第二个参数传入来实现。通过执行自定义函数返回false仅作为性能优化的方式而存在，不要依赖它来阻止渲染，因为会产生bug.

```
function MyComponent(props) {
  /* 使用 props 渲染 */
}
function areEqual(prevProps, nextProps) {
  /*
  如果nextProps 传入 render 方法的返回结果与将 prevProps 传入 render 方法的返回结果一致则返回 true,
否则返回 false
  */
}
export default React.memo(MyComponent, areEqual);
```

二、Fragments

我们之前在写组件时，如果要渲染同级的多个标签，都需要将这些标签放在一个父元素中，不然会出现语法错误，但现在React.Fragment组件能在额外创建DOM元素的情况下，它允许有key和属性，让render()方法中返回多个元素。以下实例相当于返回数组。

```
render(){
    return(

     <React.Fragment>
```

```
        Some text.
        A Heading
    </React.Fragment>
    )
}
//等价于
render() {
  return
    [
      <ChildA />
      <ChildB />
      <ChildC />
    ]
}
```

另外，现在reder()也支持返回数组或字符串。

三、Refs

1. React.createRef()

该函数创建一个能够通过ref属性附加到React元素的ref。之前的版本都是在渲染dom元素后通过字符串或回调函数来获取ref。这种方式主要是为了替换字符串ref而添加的。严格模式现在会警告使用字符串ref。注意：除了新增的createRef API，回调ref依旧可以使用。你无需替换组件中的回调ref。因为回调ref更灵活，因此仍将作为高级功能保留。

```
export default class TextInput extends TextInputBase {
    render() {
        return (
            <input
                ref={this.input}
                id={this.props.id}
                style={this.props.style}
                autoComplete="off"
                type={this.props.type}
                value={this.state.value}
            />
        )
    }
}
export default class TextInputBase extends React.PureComponent<UI.TextInput.Props, any> {
    constructor(props){
        super(props);
        this.input = React.createRef();
    }
    componentDidMount() {
        if (this.props.autoFocus) {
            this.input.focus();
        }
    }

}
```

2. React.forwardRef()

该函数会创建一个React组件，这个组件能够将其接受的ref属性转发到其组件树下的另一个组件中。Ref 转发是一项将 ref 自动地通过组件传递到其子组件的技巧。对于大多数应用中的组件来说，这通常不是必需的。但其对某些组件，尤其是可重用的组件库是很有用的。我们可以转发refs到DOM组件。

Ref 转发是一个可选特性，其允许某些组件接收 ref，并将其向下传递（即"转发"它）给子组件。如下面的例子，FancyButton使用React.forwardRef来获取获取传递给它的 ref,然后转发到它渲染的DOM button:

```
const FancyButton = React.forwardRef((props, ref) => (
    <Button ref={ref}
    {props.children}
    </Button>
));
// 这里可以直接获取 DOM button 的 ref：
const ref = React.createRef();
<FancyButton ref={ref}>Click me!;</FancyButton>
```

这样，使用 FancyButton 的组件可以获取底层 DOM 节点 button 的 ref ，并在必要时访问，就像其直接使用 DOM button 一样。这样，父组件就可以访问子组件的DOM节点。 以下是对上述示例发生情况的逐步解释：

1. 我们通过调用 React.createRef 创建了一个 React ref 并将其赋值给 ref 变量。
2. 我们通过指定 ref 为 JSX 属性，将其向下传递给 。
3. React 传递 ref 给 fowardRef 内函数 `(props, ref) => (...)`，作为其第二个参数。
4. 我们向下转发该 ref 参数到 `<button ref={ref}>`，将其指定为 JSX 属性。
5. 当 ref 挂载完成，ref.current 将指向 `<button> DOM` 节点

注意：第二个参数 ref 只在使用 React.forwardRef 定义组件时存在。常规函数和 class 组件不接收 ref 参数，且 props 中也不存在 ref。

四、lazy/Suspense

React.lazy() 提供了动态 import 组件的能力，实现代码分割。Suspense是在等待组件时暂停渲染，"等待"某些操作结束后，再进行渲染。目前 React v16.6 中 Suspense 只支持一个场景，即使用 React.lazy() 和 <React.Suspense> 实现的动态加载组件。

```
import React, {lazy, Suspense} from 'react';
const OtherComponent = lazy(() => import('./OtherComponent'));

function MyComponent() {
  return (
// 显示<div>Loading...</div>直至 OtherComponent 加载完成
    <Suspense fallback={<div>Loading...</div>}>
      <OtherComponent />
    </Suspense>
  );
}
```

说明：React.lazy()函数允许定义一个动态加载的组件。它可以延迟加载在初次渲染时未用到的组件。

注意：渲染lazy组件依赖该组件渲染树上层的<React.Suspense>组件。另外，React.Suspense可以指定加载指示器,已防其组件树中的某些子组件尚未具备渲染条件。使用 React.lazy 的动态引入特性需要 JS 环境支持 Promise。在 IE11 及以下版本的浏览器中需要通过引入 polyfill 来使用该特性。这里使用时可以将 置于你想展示加载指示器的位置，而 lazy() 则可被放置于任何你想要做代码分割的地方。

五、支持自定义DOM属性

在 React 16 中，任何标准的或自定义的 DOM 属性都是完全支持的。React 为 DOM 提供了一套以 JavaScript 为中心的 API。由于 React 组件经常采用自定义或和 DOM 相关的 props 的关系，React 采用了小驼峰命名的方式。以前的 React 版本 DOM 不识别除了 HTML 和 SVG 支持的以外属性，在 React16 版本中将会把全部的属性传递给 DOM 元素。这个新特性可以让我们摆脱可用的 React DOM 属性白名单。所以，我们可以使用自定义属性，但要注意属性名全都为小写。

六、组件的生命周期

1. Fiber

React 框架内部的运作可以分为 3 层：

- Virtual DOM 层，描述页面长什么样。
- Reconciler 层，负责调用组件生命周期方法，进行 Diff 运算等。
- Renderer 层，根据不同的平台，渲染出相应的页面，比较常见的是 ReactDOM 和 ReactNative。

这次React16改动最大的当属 Reconciler 层了，React 团队也给它起了个新的名字，叫Fiber Reconciler。

Fiber 其实指的是一种数据结构，它可以用一个纯 JS 对象来表示：

```
const fiber = {
    stateNode,    // 节点实例
    child,        // 子节点
    sibling,      // 兄弟节点
    return,       // 父节点
}
```

React之前Reconciler的调度策略像函数调用栈一样，会深度优先遍历所有的Virtual DOM节点，进行Diff。它一定要等整棵Virtual DOM计算完成之后，才将任务出栈释放主线程。所以，在浏览器主线程被React更新状态任务占据的时候，用户与浏览器进行任何的交互都不能得到反馈，只有等到任务结束，才能突然得到浏览器的响应。这样，当组件比较庞大，更新操作耗时较长时，就会导致浏览器唯一的主线程都是执行组件更新操作，而无法响应用户的输入或动画的渲染。很影响用户体验。

而React16 的Reconciler层 Fiber Reconciler每执行一段时间，都会将控制权交回给浏览器，可以分段执行。为了达到这种效果，就需要就需要有一个调度器 (Scheduler) 来进行任务分配。任务的优先级有六种：

- synchronous，与之前的React的 Reconciler操作一样，同步执行
- task，在next tick之前执行
- animation，下一帧之前执行
- high，在不久的将来立即执行
- low，稍微延迟执行也没关系
- offscreen，下一次render时或scroll时才执行

优先级高的任务（如键盘输入）可以打断优先级低的任务（如Diff）的执行，从而更快的生效。

Fiber Reconciler 在执行过程中，会分为 2 个阶段：

- 阶段一，生成 Fiber 树，得出需要更新的节点信息。这一步是一个渐进的过程，可以被打断。
- 阶段二，将需要更新的节点一次过批量更新，这个过程不能被打断

阶段一可被打断的特性，让优先级更高的任务先执行，从框架层面大大降低了页面掉帧的概率。

2. Fiber树

Fiber Reconciler 在阶段一进行 Diff 计算的时候，会生成一棵 Fiber 树。这棵树是在 Virtual DOM 树的基础上增加额外的信息来生成的，它本质来说是一个链表。

Fiber 树在首次渲染的时候会一次生成。在render函数中创建的React Element树在第一次渲染的时候会创建一棵结构一模一样的Fiber节点树。不同的React Element类型对应不同的Fiber节点类型。一个React Element的工作就由它对应的Fiber节点来负责。Fiber就是通过对象来记录组件上需要做或者已经完成的更新，一个React Element可以对应不止一个Fiber，因为Fiber在update的时候，会从原来的Fiber（称为current）clone出一个新的Fiber（我们称为alternate）。两个Fiber diff出的变化（side effect）记录在alternate上。所以一个组件在更新时最多会有两个Fiber与其对应，在更新结束后alternate会取代之前的current成为新的current节点。

在后续需要 Diff 的时候，会根据已有树和最新 Virtual DOM 的信息，生成一棵新的树。这颗新树每生成一个新的节点，都会将控制权交回给主线程，去检查有没有优先级更高的任务需要执行。如果没有，则继续构建树的过程。如果过程中有优先级更高的任务需要进行，则 Fiber Reconciler 会丢弃正在生成的树，在空闲的时候再重新执行一遍。

在构造 Fiber 树的过程中，Fiber Reconciler 会将需要更新的节点信息保存在Effect List当中，在阶段二执行的时候，会批量更新相应的节点。

3. 组件的生命周期

react16采用新的内核架构Fiber,Fiber将组件更新为两个阶段:Render Parse和Commit Parse，因此React也引入了getDerivedStatedFromProps、getSnapshotBeforeUpdate及componentDidCatch等三个全新的生命周期函数。同时也将 componentWillMount、componentWillReceiveProps 和 componentWillUpdate 标记为不安全的方法。

1. 挂载

当组件实例被创建并插入DOM中时，其生命周期调用顺序如下:

- constructor(props)：如果不初始化state或不进行方法绑定，则不需要为React组件实现构造函数。
- static getDerivedStateFromProps(nextProps, prevState)：它会在调用render方法之前调用。并且在初始挂载及后续更新时都会被调用。它应返回一个对象来更新state,如果返回null则不更新任何内容。他的作用是根据传递的props来更新state。他的一大特点是无副作用，由于处在Rednder Phase阶段，所以在每次的更新都会触发该函数，在API设计上采用了静态方法，使其无法访问实例方法、无法通过ref访问到DOM对象等，保证了该函数的纯粹高效。为了配合未来的React异步渲染机制，React v16.4对getDerivedStateFromProps做了一些改变，使其不仅在props更新时会被调用，setState时也会被触发。
  - 如果改变props的同时，有副作用的产生，这时应该使用componentDidUpdate;
  - 如果想要根据props计算属性，应该考虑将结果memoization(记忆)化；
  - 如果想要根据props变化来重置某些状态，应该考虑使用受控组件；

```
static getDerivedStateFromProps(nextProps, prevState) {
      if (nextProps.record.displayOrder !== prevState.displayOrder) {
          return {
              displayOrder:
              (!nextProps.record.displayOrder || nextProps.record.displayOrder === -1) ?
              ''
              :
              nextProps.record.displayOrder
          };
      } else {
          return null;
      }
 }
```

此方法无权访问组件实例。如果你需要，可以通过提取组件 props 的纯函数及 class 之外的状态，在 getDerivedStateFromProps()和其他 class 方法之间重用代码。

请注意，不管原因是什么，都会在每次渲染前触发此方法。这与 componentWillReceiveProps 形成对比，后者仅在父组件重新渲染时触发，而不是在内部调用 setState 时。

- render()
- componentDidMount()

2. 更新

当组件的props或state发生变化时会触发更新。组件更新的生命周期调用顺序如下：

- stiatic getDerivedStateFormProps(nextProps, prevstate)
- shouldComponentUpdate(nextProps, nextState)：根据该函数的返回值判断 React 组件的输出是否受当前 state 或 props 更改的影响。默认行为是 state 每次发生变化组件都会重新渲染。大部分情况下，你应该遵循默认行为。
- render()
- getSnapshotBeforeUpdate(prevProps, prevState)：该函数在最近一次渲染输出（提交到 DOM 节点）之前调用。它使得组件能在发生更改之前从 DOM 中捕获一些信息（例如，滚动位置）。会在组件更新之前获取一个snapshot，并且可以将计算得到的值或从DOM得到的信息传递到componentDidUpdate(prevProps, prevState, snapshot)函数的第三个参数。此用法并不常见，但它可能出现在 UI 处理中，如需要以特殊方式处理滚动位置的聊天线程等。应返回 snapshot 的值（或 null）。
- componentDidUpdate(prevProps, prevState, snapshot)：

```
componentDidUpdate(prevProps) {
  // 典型用法（不要忘记比较 props）：
  if (this.props.userID !== prevProps.userID) {
    this.fetchData(this.props.userID);
  }
}
```

该函数会在更新后会被立即调用。首次渲染不会执行此方法。当组件更新后，可以在此处对 DOM 进行操作。如果你对更新前后的 props 进行了比较，也可以选择在此处进行网络请求。（例如，当 props 未发生变化时，则不会执行网络请求）。我们也可以在 componentDidUpdate() 中直接调用 setState()，但注意它必须被包裹在一个条件语件里，正如上述的例子那样进行处理，否则会导致死循环。它还会导致额外的重新渲染，虽然用户不可见，但会影响组件性能。不要将 props 直接赋值给 state，请考虑直接使用 props。

3. 卸载

当组件从DOM中移除时会调用componentWillUnmount()

4. 错误处理

当渲染过程，生命周期，或子组件的构造函数中抛出错误时，会调用如下方法：

- static getDerivedStateFromError(error)：此生命周期会在后代组件抛出错误后被调用。它将抛出的错误作为参数，并返回一个值以更新state。注意：getDerivedStateFromError() 会在渲染阶段调用，因此不允许出现副作用。 如遇此类情况，请改用 componentDidCatch()
- componentDidCatch(error, info)：该函数让开发者可以自主处理错误信息，诸如错误信息，上报错误等。用户可以创建自己的Error Boundary来捕获错误，此生命周期在后代组件抛出错误后被调用。它接受两个参数：
  - error —— 抛出的错误。
  - info —— 带有 key为componentStack的对象，其中包含有关组件引发错误的栈信息。

  它会在“提交”阶段被调用，因此允许执行副作用。它应该用于记录错误之类的情况：

```
class ErrorBoundary extends React.Component {
  constructor(props) {
    super(props);
    this.state = { hasError: false };
  }
  static getDerivedStateFromError(error) {
    // 更新 state 使下一次渲染可以显示降级 UI
    return { hasError: true };
  }
  componentDidCatch(error, info) {
    // "组件堆栈" 例子:
    //   in ComponentThatThrows (created by App)
    //   in ErrorBoundary (created by App)
    //   in div (created by App)
    //   in App
    console.log(info.componentStack);
  }
  render() {
    if (this.state.hasError) {
      // 你可以渲染任何自定义的降级 UI
```

```
      return <h1>Something went wrong.</h1>;
    }
    return this.props.children;
  }
}
//然后你就可以直接使用它
<ErrorBoundary>
      <MyWidget/>
</ErrorBpundary>
```

注意：如果发生错误，你可以通过调用 setState 使用 componentDidCatch() 渲染降级 UI，但在未来的版本中将不推荐这样做。 可以使用静态 getDerivedStateFromError() 来处理降级渲染。

- Error boundaries：Error boundaries 是 React 组件，它会在其子组件树中的任何位置捕获 JavaScript 错误，并记录这些错误，展示降级 UI 而不是崩溃的组件树。Error boundaries 组件会捕获在渲染期间，在生命周期方法以及其整个树的构造函数中发生的错误。如果 class 组件定义了生命周期方法 static getDerivedStateFromError() 或 componentDidCatch() 中的任何一个（或两者），它就成为了 Error boundaries。通过生命周期更新 state 可让组件捕获树中未处理的 JavaScript 错误并展示降级 UI。仅使用 Error boundaries 组件来从意外异常中恢复的情况；不要将它们用于流程控制。

七、严格模式

可以在开发阶段开启严格模式，发现应用存在的潜在问题，提升应用的健壮性，其主要能检测下列问题：

- 识别被标志位不安全的生命周期函数;
- 对弃用的 API 进行警告;
- 探测某些产生副作用的方法;
- 检测是否使用 findDOMNode;
- 检测是否采用了老的 Context API;

```
class App extends React.Component {
    render() {
        return (
            <div>
                <React.StrictMode>
                    <ComponentA />
                </React.StrictMode>
            </div>
        )
    }
}
```