08 组件实例：组件实例到底是什么？

更新时间：2020-08-20 09:57:56

> 人的一生可能燃烧也可能腐朽，我不能腐朽，我愿意燃烧起来！——奥斯特洛夫斯基

前言

React 官方并没有较多的介绍 React 组件的实例，因为组件实例是应用程序运行时 React 在内存中维护的一种对象（数据），对开发者来说是透明的。想要了解 React 内部运行机制，组件实例是一个关键点。本文会介绍 **React 组件的继承原理，组件实例的创建以及组件实例在React应用程序运行时的作用**。

组件继承原理

前面文章提到函数组件不会被实例化，class 类型组件可以（必须）被实例化。通常定义 class 类型的组件的方式见代码示例 2.5.1。

```
class UpdateCounter extends React.Component {
  constructor(props) {
  super(props);
  this.state = {
   count: 0,
   text: '点击计数'
  };
  this.handleClick = this.handleClick.bind(this);
  }

  handleClick() {
    // case1 setState入参类型为function，函数必须有返回值
    this.setState((state) => {
      return {count: state.count + 1};
    });

    // case2 setState入参类型为object
  this.setState({
   count: this.state.count + 1
  });
  }

  render() {
    return (
     <div className="wrap-box">
       <button key="1" onClick={this.handleClick}>点击计数</button>
       <span key="2" id="spanText" className="span-text">{this.state.count}</span>
     </div>
    )
  }
}
```

代码示例 2.5.1 定义 class 类型的组件

在代码示例 2.5.1中，使用 class 定义的`UpdateCounter`组件，该组件继承于`React.Component`。我们思考一些，`UpdateCounter`组件能从`React.Component`上面继承哪些属性和方法呢？

首先来看一下`React.Component`的源码定义，见代码示例 2.5.2。

```
// 源码位置：packages/react/src/ReactBaseClasses.js
function Component(props, context, updater) {
  this.props = props;
  this.context = context;
  this.refs = emptyObject;
  this.updater = updater || ReactNoopUpdateQueue;
}
// 部分属性定义在原型上
Component.prototype.setState = function (partialState, callback) {
  // 执行setState时会先校验入参的类型是否正确，入参类型必须是object或function
  (function () {
    if (!(typeof partialState === 'object' || typeof partialState === 'function' || partialState == null)) {
      {
        throw ReactError(Error('setState(...): 参数类型必须是object或者function'));
      }
    }
  })();

  this.updater.enqueueSetState(this, partialState, callback, 'setState');
};

...
```

代码示例 2.5.2 React.Component 源码定义

根据 React.Component 的源码定义可知，React.Component 是一个构造函数，它的 props，context，refs 和 updater 属性定义在了函数内部，而 setState 方法定义在了其原型对象上。

在弄清楚 React 组件能从 React.Component 上面继承哪些属性和方法之前，我们需要对 JavaScript 中 class 继承原理进行回顾。基于 class 继承逻辑体现在以下几方面。

1. 如果子类中没有定义 constructor 方法，那么这个方法会被默认添加。
2. 如果子类中有定义 constructor 方法，那么必须结合 super() 使用，且 super() 需要出现在 this.xx 之前。
3. 使用 class 结合 extends 实现继承和 ES5 的继承在本质上有些不一样。在 ES5 中每个对象实例都有 __proto__ 属性，指向对应构造函数的 prototype 属性。class 作为构造函数的语法糖同时拥有 __proto__ 属性和 prototype 属性，因此同时存在两条继承链：

> 子类的 __proto__ 属性表示构造函数的继承，总是指向父类。
> 子类 prototype 属性的 __proto__ 属性表示方法的继承，总是指向父类的 prototype 属性

根据 JavaScript 中 class 继承原理，将 UpdateCounter 和 React.Component 两者之间的继承关系用图描述，见图 2.5.1。

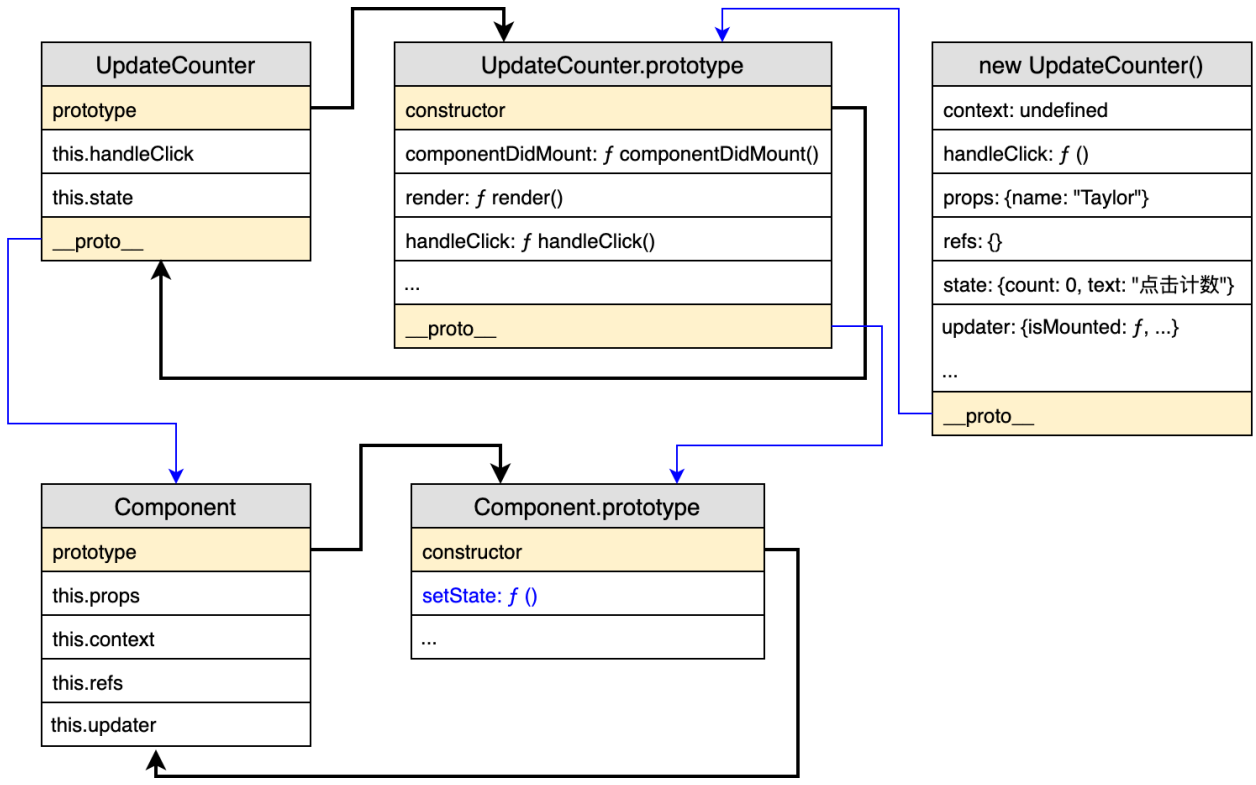

图 2.5.1 UpdateCounter 继承 React.Component 后两者之间的结构关系

UpdateCounter.__proto__ 指向了 Component 构造函数。UpdateCounter.prototype.__proto__ 指向了 Component 构造函数的原型对象 Component.prototype 。UpdateCounter 组件实例的 __proto__ 指向了 UpdateCounter 构造函数的原型对象。

组件实例的创建与形态

```
// 源码位置：packages/react-reconciler/src/ReactFiberClassComponent.js
function constructClassInstance(workInProgress, ctor, props, renderExpirationTime) {
  ...
  // ctor是定义的组件类
  var instance = new ctor(props, context);
  ...
}
```

代码示例 2.5.3 创建组件实例

代码示例 2.5.3 展示了 React 将组件实例化的方式。通过执行 constructClassInstance 函数， UpdateCounter 组件被实例化，该组件实例在内存中的具体形态（取自Chrome浏览器控制台）见代码示例 2.5.4。

```
UpdateCounter: {
  context: undefined,  // 继承属性
  handleClick: ƒ (),  // 自有属性
  props: {name: "Taylor"}, // 继承属性
  refs: {},          // 继承属性
  state: {count: 0, text: "点击计数"}, // 自有属性
  updater: {isMounted: ƒ, enqueueForceUpdate: ƒ, enqueueReplaceState: ƒ, enqueueSetState: ƒ}, // 继承属性
  isMounted: (...),   // 继承属性
  replaceState: (...), // 继承属性
  // chrome调试工具显示__proto__指向Component，实际上是UpdateCounter.prototype
  __proto__: Component
}
```

代码示例 2.5.4 组件实例在内存中的形态

UpdateCounter 组件实例（对象）中包含了自由属性，如 state，事件处理函数等，还有继承于 React.Component 的属性，如 props， setState 等。现在需要思考的是，React 使用组件实例做了哪些工作呢？

组件实例在 React 应用程序运行时的作用

一、返回组件元素与状态（state）

```
// 源码位置：packages/react-reconciler/src/ReactFiberBeginWork.js
function finishClassComponent(current$$1, workInProgress, Component, shouldUpdate, hasContext, renderExpirationTime) {
  ...
  // instance.render()返回当前组件的元素
  var nextChildren = instance.render();
  ...
  // 开始执行协调算法，返回下一个 Fiber 结点
  reconcileChildren(current$$1, workInProgress, nextChildren, renderExpirationTime);
  ...
  // 使用组件实例值来记忆当前 Fiber 结点状态，可用于后续 diff
  workInProgress.memoizedState = instance.state;
}
```

代码示例 2.5.5 使用组件实例返回组件元素与状态

二、调用生命周期函数

组件的生命周期函数是在组件实例上面进行调用的，见代码示例 2.5.6。

```
// 源码位置：packages/react-reconciler/src/ReactFiberCommitWork.js
// 调用commit完成后的生命周期函数
function commitLifeCycles(finishedRoot, current$$1, finishedWork, committedExpirationTime) {
  // tag标识了当前Fiber节点的类型，包括FunctionComponent，ClassComponent，HostComponent等
  switch (finishedWork.tag) {
      ...
      case ClassComponent:
       ...
        instance.componentDidMount();
       ...
  }
}
// 调用组件被卸载前的生命周期函数
var callComponentWillUnmountWithTimer = function (current$$1, instance) {
  ...
  instance.componentWillUnmount();
  ...
};

...
```

代码示例 2.5.6 通过组件实例调用其生命周期函数

三、触发更新（执行 setState(...) 操作）

应用程序首次渲染时会为组件实例绑定对应的更新器，当组件接收到事件触发更新时，通过组件实例上面的更新器执行更新流程，这部分逻辑见代码示例 2.5.7。

```
// 源码位置：packages/react-reconciler/src/ReactFiberClassComponent.js
// 应用程序首次渲染时会为组件实例绑定更新器
function adoptClassInstance(workInProgress, instance) {
  instance.updater = classComponentUpdater;
  workInProgress.stateNode = instance;
}
// 组件更新器
var classComponentUpdater = {
  enqueueSetState: function (inst, payload, callback) {
    // 创建更新对象
    var update = createUpdate(expirationTime, suspenseConfig);
    // 为更新对象赋值更新内容
    update.payload = payload;
    ...
    // 将更新对象加入更新队列
    enqueueUpdate(fiber, update);
    // 开始（更新）调度工作
    scheduleWork(fiber, expirationTime);
    ...
  }
  ...
}
```

代码示例 2.5.7 通过组件实例触发更新

小结

组件实例是 React 应用程序运行时组件被实例化后的状态，每一个组件实例都拥有自身的属性和继承于 React.Component 的属性。应用程序首次渲染时通过调用组件实例的 instance.render() 返回 **React** 元素，用于构建页面（UI）。当应用程序被（事件）触发更新时，组件实例调用自身的更新器（updater）进入更新渲染流程。此外，在应用程序渲染的 render 或者 commit 阶段（前后）中会通过组件实例调用对应的生命周期函数。

组件实例调用自身的 render 函数返回 **React** 元素，下一节将会详细介绍 React 元素的定义与设计原理。

}

07 生命周期：为什么 React v16 产生了新的生命周期函数

09 元素：从设计者的角度理解 React 元素