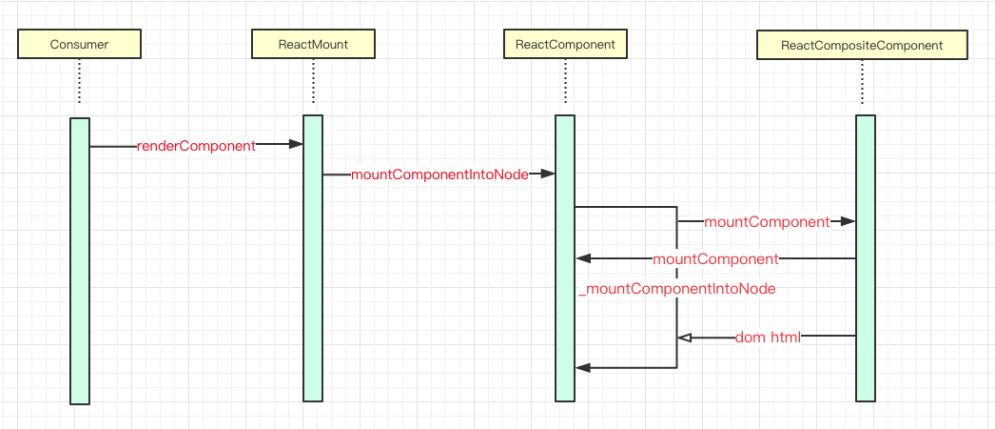
组件渲染

> 珠峰前端架构师技术分享课 <https://ke.qq.com/course/272058>

组件渲染流程



组件渲染流程相对来说就复杂多了。ReactMount.renderComponent在react初探章节讲过。如果组件渲染过，就更新组件属性，如果组件没有渲染过，挂载组件事件，并把虚拟组件渲染成真实组件插入container内。通常，我们很少去调用两次renderComponent，所以大多数情况下不会更新组件属性而是新创建dom节点并插入到container中。

ReactComponent.mountComponentIntoNode之内开启了一个事务，事务保证渲染阶段不会有任何事件触发，并阻断的componentDidMount事件，待执行后执行等，事务在功能一章我们会详细讲解，这里不细讨论。

ReactComponent._mountComponentIntoNode这个函数调用mountComponent获得要渲染的innerHTML，然后更新container的innerHTML。

ReactCompositeComponent.mountComponent是最主要的逻辑方法。这个函数内处理了react的生命周期以及componentWillComponent和componentDidMount生命周期钩子函数，调用render返回实际要渲染的内容，如果内容是复合组件，仍然会调用mountComponent，**复合组件最终一定会返回原生组件**， 并且最终调用ReactNativeComponent的mountComponent函数生成要渲染的innerHTML。

下面我们依次讲解各个函数具体都做了什么：

1. ReactComponent.mountComponentIntoNode

通过传入的组件，根组件ID，与container的dom节点，创建组件并且把组件渲染到container节点下。

这里可以看到事务机制（在机制中讲解），目前我们只需要了解事务调用了_mountComponentIntoNode函数并且把this,rootId,container作为函数参数传入就可以了。

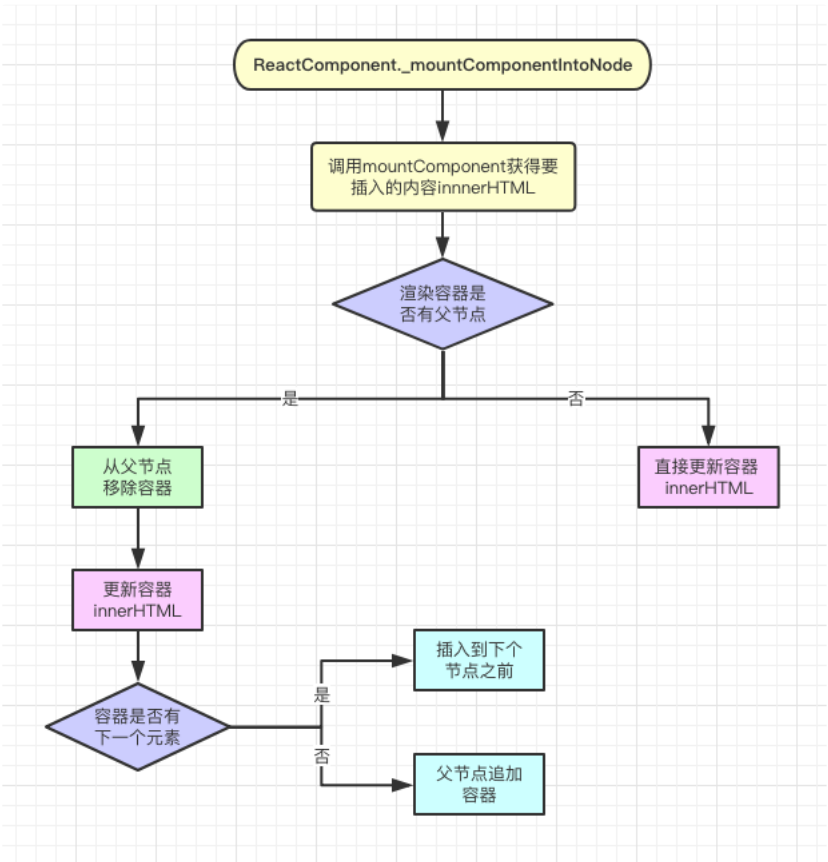
特别说明下：rootID指的是渲染组件的id，而不是容器的ID，react内部会给每个组件生成一个ID。

```
/**
  * Mounts this component and inserts it into the DOM.
  *
  * @param {string} rootID DOM ID of the root node.
```

```
  * @param {DOMElement} container DOM element to mount into.
  * @final
  * @internal
  * @see {ReactMount.renderComponent}
  */
mountComponentIntoNode: function(rootID, container) {
  // 事务机制
  var transaction = ReactComponent.ReactReconcileTransaction.getPooled();
  transaction.perform(
    this._mountComponentIntoNode,
    this,
    rootID,
    container,
    transaction
  );
  ReactComponent.ReactReconcileTransaction.release(transaction);
}
```

2. ReactComponent._mountComponentIntoNode

首先调用ReactCompositeComponent.mountComponent方法初始化组件，挂在事件并且返回组件的innerHTML(markup)。然后把返回的innerHTML更新容器的innerHTML。每次更新前，把容器从document上移除，然后更新innerHTML，最后再挂在到document上。其逻辑如下：



对应的源码如下：

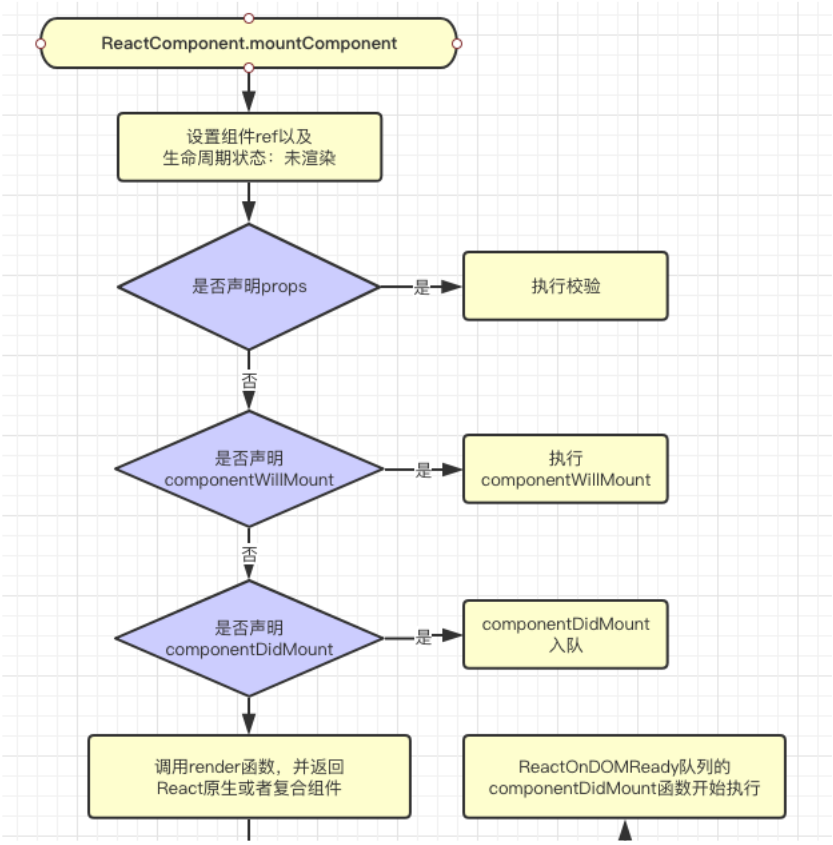
```
/**
* @param {string} rootID DOM ID of the root node.
* @param {DOMElement} container DOM element to mount into.
* @param {ReactReconcileTransaction} transaction
```

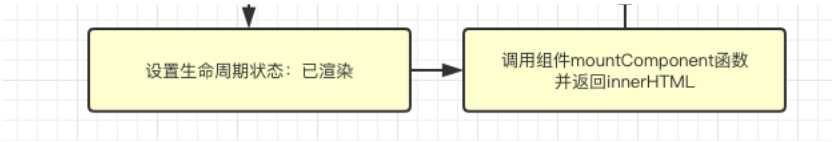
```
 * @final
 * @private
 */
_mountComponentIntoNode: function(rootID, container, transaction) {
  var renderStart = Date.now();
  // 调用ReactCompositeComponent.js
  var markup = this.mountComponent(rootID, transaction);
  ReactMount.totalInstantiationTime += (Date.now() - renderStart);

  var injectionStart = Date.now();
  // 每次更新container时，先从document删除掉，然后插入innerHTML，然后再插入到next节点的前面。
  var parent = container.parentNode;
  if (parent) {
    var next = container.nextSibling;
    parent.removeChild(container);
    container.innerHTML = markup;
    if (next) {
      parent.insertBefore(container, next);
    } else {
      parent.appendChild(container);
    }
  } else {
    container.innerHTML = markup;
  }
  ReactMount.totalInjectionTime += (Date.now() - injectionStart);
},
```

3. ReactCompositeComponent.mountComponent

注意这里的类变成了ReactCompositeComponent，源码中调用this.monutComponent，为什么不是调用ReactComponent.mountComponent呢？这里使用了多重继承机制mixin（在主要功能中讲解）。

这个函数是虚拟dom最重要的函数，操作也比较复杂，执行操作如下：





对应源码如下：

```
/**
 * Initializes the component, renders markup, and registers event listeners.
 *
 * @param {string} rootID DOM ID of the root node.
 * @param {ReactReconcileTransaction} transaction
 * @return {?string} Rendered markup to be inserted into the DOM.
 * @final
 * @internal
 */
mountComponent: function(rootID, transaction) {
  // 挂在组件ref(等于当前组件实例)到this.refs上
  ReactComponent.Mixin.mountComponent.call(this, rootID, transaction);

  // Unset `this._lifeCycleState` until after this method is finished.
  this._lifeCycleState = ReactComponent.LifeCycle.UNMOUNTED;
  this._compositeLifeCycleState = CompositeLifeCycle.MOUNTING;

  // 如果组件声明有props，执行校验
  if (this.constructor.propDeclarations) {
    this._assertValidProps(this.props);
  }

  // 为组件声明事件绑定this
  if (this.__reactAutoBindMap) {
    this._bindAutoBindMethods();
  }

  this.state = this.getInitialState ? this.getInitialState() : null;
  this._pendingState = null;

  // 如果组件声明componentWillMount函数，执行，并且把setState的结果更新到this.state上
  if (this.componentWillMount) {
    this.componentWillMount();
    // When mounting, calls to `setState` by `componentWillMount` will set
    // `this._pendingState` without triggering a re-render.
    if (this._pendingState) {
      this.state = this._pendingState;
      this._pendingState = null;
    }
  }

  // 如果组件声明componentDidMount函数，把componentDidMount函数加入到ReactOnDOMReady队列
  if (this.componentDidMount) {
    transaction.getReactOnDOMReady().enqueue(this, this.componentDidMount);
  }

  // 调用组件声明的render函数，并返回ReactComponent抽象类实例（ReactComponsiteComponent或
  // ReactNativeComponent），调用相应的mountComponent函数
  this._renderedComponent = this._renderValidatedComponent();

  // Done with mounting, `setState` will now trigger UI changes.
  this._compositeLifeCycleState = null;
  this._lifeCycleState = ReactComponent.LifeCycle.MOUNTED;

  return this._renderedComponent.mountComponent(rootID, transaction);
},
```

3.1 挂在组件ref(等于当前组件实例)到this.refs上，设置生命周期状态和rootID

```
mountComponent: function(rootID, transaction) {
  invariant(
    this._lifeCycleState === ComponentLifeCycle.UNMOUNTED,
    'mountComponent(%s, ...): Can only mount an unmounted component.',
    rootID
  );
  var props = this.props;
  if (props.ref != null) {
    ReactOwner.addComponentAsRefTo(this, props.ref, props[OWNER]);
  }
  this._rootNodeID = rootID;
  this._lifeCycleState = ComponentLifeCycle.MOUNTED;
  // Effectively: return '';
},
```

如果组件ref属性不为空，则为组件的this.refs上挂在当前组件，也就是this，实现如下：

```
addComponentAsRefTo: function(component, ref, owner) {
  invariant(
    ReactOwner.isValidOwner(owner),
    'addComponentAsRefTo(...): Only a ReactOwner can have refs.'
  );
  owner.attachRef(ref, component);
},

attachRef: function(ref, component) {
  invariant(
    component.isOwnedBy(this),
    'attachRef(%s, ...): Only a component\'s owner can store a ref to it.',
    ref
  );
  var refs = this.refs || (this.refs = {});
  refs[ref] = component;
},
```

3.2 设置组件生命周期状态

组件的生命周期状态与生命周期钩子函数是react两个概念，很多人容易混淆，这里我们只讨论生命周期状态。在react中存在两种生命周期：

- 主：组件生命周期：_lifeCycleState，用来校验react组件的在执行函数时状态值是否正确。
- 辅：复合组件生命周期：_compositeLifeCycleState，用来保证setState流程不受其他行为影响。

3.2.1 _lifeCycleState

组件生命周期状态共有个枚举值：MOUNTED与UNMOUNTED。

```
/**
 * Every React component is in one of these life cycles.
 */
var ComponentLifeCycle = keyMirror({
  /**
   * Mounted components have a DOM node representation and are capable of
```

```
   * receiving new props.
   */
  MOUNTED: null,
  /**
   * Unmounted components are inactive and cannot receive new props.
   */
  UNMOUNTED: null
});
```

其用途十分简单清晰，只要搜索_lifeCycleState即可，我们来看几个例子：

```
getDOMNode: function() {
  invariant(
    ExecutionEnvironment.canUseDOM,
    'getDOMNode(): The DOM is not supported in the current environment.'
  );
  invariant(
    this._lifeCycleState === ComponentLifeCycle.MOUNTED,
    'getDOMNode(): A component must be mounted to have a DOM node.'
  );
  var rootNode = this._rootNode;
  if (!rootNode) {
    rootNode = document.getElementById(this._rootNodeID);
    if (!rootNode) {
      // TODO: Log the frequency that we reach this path.
      rootNode = ReactMount.findReactRenderedDOMNodeSlow(this._rootNodeID);
    }
    this._rootNode = rootNode;
  }
  return rootNode;
},
```

```
unmountComponent: function() {
  debugger
  invariant(
    this._lifeCycleState === ComponentLifeCycle.MOUNTED,
    'unmountComponent(): Can only unmount a mounted component.'
  );
  var props = this.props;
  if (props.ref != null) {
    ReactOwner.removeComponentAsRefFrom(this, props.ref, props[OWNER]);
  }
  this._rootNode = null;
  this._rootNodeID = null;
  this._lifeCycleState = ComponentLifeCycle.UNMOUNTED;
},
```

```
receiveProps: function(nextProps, transaction) {
  invariant(
    this._lifeCycleState === ComponentLifeCycle.MOUNTED,
    'receiveProps(...): Can only update a mounted component.'
  );
  var props = this.props;
  // If either the owner or a `ref` has changed, make sure the newest owner
  // has stored a reference to `this`, and the previous owner (if different)
  // has forgotten the reference to `this`.
  if (nextProps[OWNER] !== props[OWNER] || nextProps.ref !== props.ref) {
    if (props.ref != null) {
      ReactOwner.removeComponentAsRefFrom(this, props.ref, props[OWNER]);
    }
    // Correct, even if the owner is the same, and only the ref has changed.
    if (nextProps.ref != null) {
```

```
        ReactOwner.addComponentAsRefTo(this, nextProps.ref, nextProps[OWNER]);
      }
    }
  },
```

从这三个例子我们能看出，_lifeCycleState只是在相应的阶段触发时候用来做校验使用，而且只是给出报错提示。

3.2.2 _compositeLifeCycleState

复合组件生命周期只在一个地方消费，既setState中：

```
replaceState: function(completeState) {
  var compositeLifeCycleState = this._compositeLifeCycleState;
  invariant(
    this._lifeCycleState === ReactComponent.LifeCycle.MOUNTED ||
    compositeLifeCycleState === CompositeLifeCycle.MOUNTING,
    'replaceState(...): Can only update a mounted (or mounting) component.'
  );
  invariant(
    compositeLifeCycleState !== CompositeLifeCycle.RECEIVING_STATE &&
    compositeLifeCycleState !== CompositeLifeCycle.UNMOUNTING,
    'replaceState(...): Cannot update while unmounting component or during ' +
    'an existing state transition (such as within `render`).'
  );

  this._pendingState = completeState;

  // Do not trigger a state transition if we are in the middle of mounting or
  // receiving props because both of those will already be doing this.
  if (compositeLifeCycleState !== CompositeLifeCycle.MOUNTING &&
      compositeLifeCycleState !== CompositeLifeCycle.RECEIVING_PROPS) {
    this._compositeLifeCycleState = CompositeLifeCycle.RECEIVING_STATE;

    var nextState = this._pendingState;
    this._pendingState = null;

    var transaction = ReactComponent.ReactReconcileTransaction.getPooled();
    transaction.perform(
      this._receivePropsAndState,
      this,
      this.props,
      nextState,
      transaction
    );
    ReactComponent.ReactReconcileTransaction.release(transaction);

    this._compositeLifeCycleState = null;
  }
```

setState会调用replaceState，然后调用_receivePropsAndState来更新界面。

如果组件正处在mounting的过程或者接收到props的过程中，那么就将state缓存在_pendingState中，并不会更新界面的值。

下一章我们以一个例子来验证这个理论，并且分析setState机制。

3.3 如果组件声明有props，执行校验

ReactCompositeComponent

```
_assertValidProps: function(props) {
  var propDeclarations = this.constructor.propDeclarations;
  var componentName = this.constructor.displayName;
  for (var propName in propDeclarations) {
    var checkProp = propDeclarations[propName];
    if (checkProp) {
      checkProp(props, propName, componentName);
    }
  }
},
```

（追踪）this.constructor.propDeclarations就是组件声明的props属性，而 props是运行时实际传入的属性。我们可以看到声明props的属性值既checkProp在此处执行。

这有什么用呢？我们先改写个小例子测试一下：

```
var ExampleApplication = React.createClass({
  props: {
    proptest: (a,b,c) => {
      debugger
    },
    // proptest: createPrimitiveTypeChecker('number')
  },

  render: function() {
    var elapsed = Math.round(this.props.elapsed  / 100);
    var seconds = elapsed / 10 + (elapsed % 10 ? '' : '.0' );
    var message =
      'React has been successfully running for ' + seconds + ' seconds.';

    return <div style={{fontSize: 20}} ref="test">{message}</div>
  }
});
var start = new Date().getTime();
setInterval(function() {
  React.renderComponent(
    <ExampleApplication elapsed={new Date().getTime() - start} proptest={1}/>,
    document.getElementById('container')
  );
}, 50);
```

我们看到a,b,c属性分别对应组件属性json，当前校验的属性名以及组件名（displayName）。这样我们就可以写自己的校验器了，校验任意一个属性的正确性。

我们先看看官方的prop-types <https://github.com/facebook/prop-types> 是如何实现的。

从package.json中找到入口文件index.js的在非生产环境（process.env.NODE_ENV !== 'production'）的引用文件（factoryWithTypeCheckers.js）的入口属性代码如下：

```
var ReactPropTypes = {
    array: createPrimitiveTypeChecker('array'),
    bool: createPrimitiveTypeChecker('boolean'),
    func: createPrimitiveTypeChecker('function'),
    number: createPrimitiveTypeChecker('number'),
    object: createPrimitiveTypeChecker('object'),
    string: createPrimitiveTypeChecker('string'),
    symbol: createPrimitiveTypeChecker('symbol'),

    any: createAnyTypeChecker(),
    arrayOf: createArrayOfTypeChecker,
    element: createElementTypeChecker(),
```

```
    instanceOf: createInstanceTypeChecker,
    node: createNodeChecker(),
    objectOf: createObjectOfTypeChecker,
    oneOf: createEnumTypeChecker,
    oneOfType: createUnionTypeChecker,
    shape: createShapeTypeChecker,
    exact: createStrictShapeTypeChecker,
};
```

参照官方的例子，我们来实现一个自己的checker：

```
function createChecker(expectedType) {
  function validate(props, propName, componentName, location, propFullName, secret) {
    var propValue = props[propName];
    var propType = typeof propValue;
    if (propType !== expectedType) {
      var preciseType = typeof propValue;
      console.warn('Invalid ' + location + ' `' + propFullName + '` of type ' + ('`' + preciseType + '` suppli
ed to `' + componentName + '`, expected ') + ('`' + expectedType + '`.'));
    }
  }

  function checkType(isRequired, props, propName, componentName, location, propFullName, secret) {
    if (!props[propName] && isRequired) {
      return console.warn('The ' + location + ' `' + propName + '` is marked as required in ' + ('`' + compone
ntName + '`, but its value is `undefined`.'));
    }
    return validate(props, propName, componentName, location, propFullName);
  }

  var chainedCheckType = checkType.bind(null, false);
  chainedCheckType.isRequired = checkType.bind(null, true);

  return chainedCheckType;
}

const numberChecker = createChecker('number')

var ExampleApplication = React.createClass({
  props: {
    proptest: numberChecker.isRequired,
  },

  render: function() {
    var elapsed = Math.round(this.props.elapsed  / 100);
    var seconds = elapsed / 10 + (elapsed % 10 ? '' : '.0' );
    var message =
      'React has been successfully running for ' + seconds + ' seconds.';

    return <div style={{fontSize: 20}} ref="test">{message}</div>
  }
});

React.renderComponent(
  <ExampleApplication elapsed={new Date().getTime()} proptest={'a'}/>,
  document.getElementById('container')
)
```

3.4 为组件事件函数绑定this

我们都知道react事件需要绑定this实例，但很少有人知道这个api，因为在es6中已经弃用了，我们先来看看es6的绑定事件this的函数（为什么要绑定this，将会在事件章节讲到）。

- 初始化调用bind
- 调用处bind
- 调用处使用箭头函数
- 声明处使用箭头函数

无论哪种绑定，最终都会编译成bind函数，而bind函数最后也会调用call函数执行。而在react在es5中实现的事件绑定却十分繁琐，经历了以下几个过程：

- 用户代码运行期：调用autoBind把包装用户声明函数，执行后重新返回函数，并把用户事件挂载到__reactAutoBind属性上。

```
/**
 * Marks the provided method to be automatically bound to the component.
 * This means the method's context will always be the component.
 *
 *   React.createClass({
 *     handleClick: React.autoBind(function() {
 *       this.setState({jumping: true});
 *     }),
 *     render: function() {
 *       return <a onClick={this.handleClick}>Jump</a>;
 *     }
 *   });
 *
 * @param {function} method Method to be bound.
 * @public
 */
autoBind: function(method) {
  function unbound() {
    invariant(
      false,
      'React.autoBind(...): Attempted to invoke an auto-bound method that ' +
      'was not correctly defined on the class specification.'
    );
  }
  unbound.__reactAutoBind = method;
  return unbound;
}
```

- 组件初始化(mixin)阶段：编译每个用户用户声明属性，如果发现有**reactAutoBind属性，则把属性所对应的用户声明事件放到**reactAutoBindMap变量缓存 。

```
if (property && property.__reactAutoBind) {  // 如果是使用React.autoBind，则放在原型链的__reactAutoBindMap属性中
  if (!proto.__reactAutoBindMap) {
    proto.__reactAutoBindMap = {};
  }
  proto.__reactAutoBindMap[name] = property.__reactAutoBind;
}
```

- 组件渲染(mount)阶段：把__reactAutoBindMap中的每个函数取出，依次包装后挂载到当前组件实例上。包装函数里再调用用户声明事件，把this指向当前组件。

```
/**
 * @private
 */
_bindAutoBindMethods: function() {
  for (var autoBindKey in this.__reactAutoBindMap) {
    if (!this.__reactAutoBindMap.hasOwnProperty(autoBindKey)) {
      continue;
    }
    var method = this.__reactAutoBindMap[autoBindKey];
```

```
    this[autoBindKey] = this._bindAutoBindMethod(method);
  }
},

/**
 * Binds a method to the component.
 *
 * @param {function} method Method to be bound.
 * @private
 */
_bindAutoBindMethod: function(method) {
  var component = this;
  var hasWarned = false;
  function autoBound(a, b, c, d, e, tooMany) {
    invariant(
      typeof tooMany === 'undefined',
      'React.autoBind(...): Methods can only take a maximum of 5 arguments.'
    );
    if (component._lifeCycleState === ReactComponent.LifeCycle.MOUNTED) {
      return method.call(component, a, b, c, d, e);
    } else if (!hasWarned) {
      hasWarned = true;
      if (__DEV__) {
        console.warn(
          'React.autoBind(...): Attempted to invoke an auto-bound method ' +
          'on an unmounted instance of `%s`. You either have a memory leak ' +
          'or an event handler that is being run after unmounting.',
          component.constructor.displayName || 'ReactCompositeComponent'
        );
      }
    }
  }
  return autoBound;
}
```

3.5 初始化state, 如果组件声明componentWillMount函数，执行，并且把setState的结果更新到this.state上

```
this.state = this.getInitialState ? this.getInitialState() : null;
this._pendingState = null;

if (this.componentWillMount) {
  this.componentWillMount();
  // When mounting, calls to `setState` by `componentWillMount` will set
  // `this._pendingState` without triggering a re-render.
  if (this._pendingState) {
    this.state = this._pendingState;
    this._pendingState = null;
  }
}
```

如果有getInitialState函数，把函数返回值赋给this.state,如果有componentWillMount函数，执行。

仅看这一段代码，是看不出来什么端倪的。当组件正在渲染，componentWillMount中的setState不会触发界面渲染(render函数)，而是保存在this._pendingState属性中，在componentWillMount结束后，把this._pendingState属性赋给this.state，仍然不会触发界面更新。这样大大提高了性能。对于同步的setState，尽可能的放在componentWillMount钩子（更优先放在getInitialState中）中而不是componentDidMount中。

下一章我们以一个例子，介绍componentWillMount与state,setState之间的关系。

3.6 如果组件声明componentDidMount函数，把componentDidMount函数加入到ReactOnDOMReady队列

队列涉及到事务机制，我们稍后讲解。此处只需要知道，我们把componentDidMount函数入队即可。

3.7 调用组件声明的render函数，并返回ReactComponent抽象类实例（ReactComponsiteComponent或ReactNativeComponent)，调用相应的mountComponent函数

```
_renderValidatedComponent: function () {
  ReactCurrentOwner.current = this;
  var renderedComponent = this.render();
  ReactCurrentOwner.current = null;
  invariant(
    ReactComponent.isValidComponent(renderedComponent),
    '%s.render(): A valid ReactComponent must be returned.',
    this.constructor.displayName || 'ReactCompositeComponent'
  );
  return renderedComponent;
},
```

之前分析过，this.render只会返回两种组件实例：原生组件或复合组件实例，既ReactComponsiteComponent或ReactNativeComponent的实例。

ReactCurrentOwner里只有一个属性current，非常让人费解。这个属性是给ReactComponent的props[OWNER]使用的。由于只有render内有值，所有，也只有render内渲染出来的组件才有OWNER。

给props[OWNER]赋值的代码:

```
construct: function(initialProps, children) {
  this.props = initialProps || {};
  if (typeof children !== 'undefined') {
    this.props.children = children;
  }
  // Record the component responsible for creating this component.
  this.props[OWNER] = ReactCurrentOwner.current;
  // All components start unmounted.
  this._lifeCycleState = ComponentLifeCycle.UNMOUNTED;
},
```

这个函数早在组件的初始化阶段就执行了，而不是在mount阶段，所以复合组件的props[OWNER]属性始终为null。

当复合组件返回原生组件时，将再次调用construct函数，这时候原生组件的owner=当前复合组件实例。

这个owner目前有两个用途：

- 原生组件不能调用setProps函数

```
replaceProps: function (props) {
  invariant(
    !this.props[OWNER],
    'replaceProps(...): You called `setProps` or `replaceProps` on a ' +
    'component with an owner. This is an anti-pattern since props will ' +
    'get reactively updated when rendered. Instead, change the owner\'s ' +
```

```
    '`render` method to pass the correct value as props to the component ' +
    'where it is created.'
  );
  var transaction = ReactComponent.ReactReconcileTransaction.getPooled();
  transaction.perform(this.receiveProps, this, props, transaction);
  ReactComponent.ReactReconcileTransaction.release(transaction);
},
```

- 复合组件不能添加ref属性

```
isOwnedBy: function (owner) {
  return this.props[OWNER] === owner;
}

attachRef: function(ref, component) {
  invariant(
    component.isOwnedBy(this),
    'attachRef(%s, ...): Only a component\'s owner can store a ref to it.',
    ref
  );
  var refs = this.refs || (this.refs = {});
  refs[ref] = component;
},
```

3.8 调用相应的mountComponent函数并返回给最终函数

终于到最后一步了，当调用ReactComponsiteComponent.mountComponent会继续递归以上过程，直到找到原生组件为止。最终调用ReactNativeComponent.mountComponent,代码如下：

```
mountComponent: function(rootID, transaction) {
  ReactComponent.Mixin.mountComponent.call(this, rootID, transaction);
  assertValidProps(this.props);
  return (
    this._createOpenTagMarkup() +
    this._createContentMarkup(transaction) +
    this._tagClose
  );
},
```

再次调用ReactComponent.mountComponent初始化原生组件ref等操作。 createOpenTagMarkup函数构造组件的起始标签，createContentMarkup构建组件的内容，_tagClose构建组件的闭合标签。

到这里一个完整mountComponent就完成了。