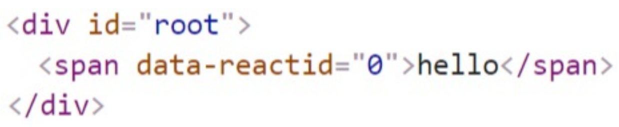
link: null
title: 珠峰架构师成长计划
description: null
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=193 sentences=1084, words=6793

1.初始化项目

```
create-react-app zhufeng_react5
cd zhufeng_react5
cnpm i jquery -S
npm start
```

2. 渲染文本

2.1 渲染效果



```
<div id="root">
  <span data-reactid="0">hello</span>
</div>
```

2.2 实现

2.2.1 index.js

src\index.js

```
import React from './react';
React.render('hello',document.getElementById('root'));
```

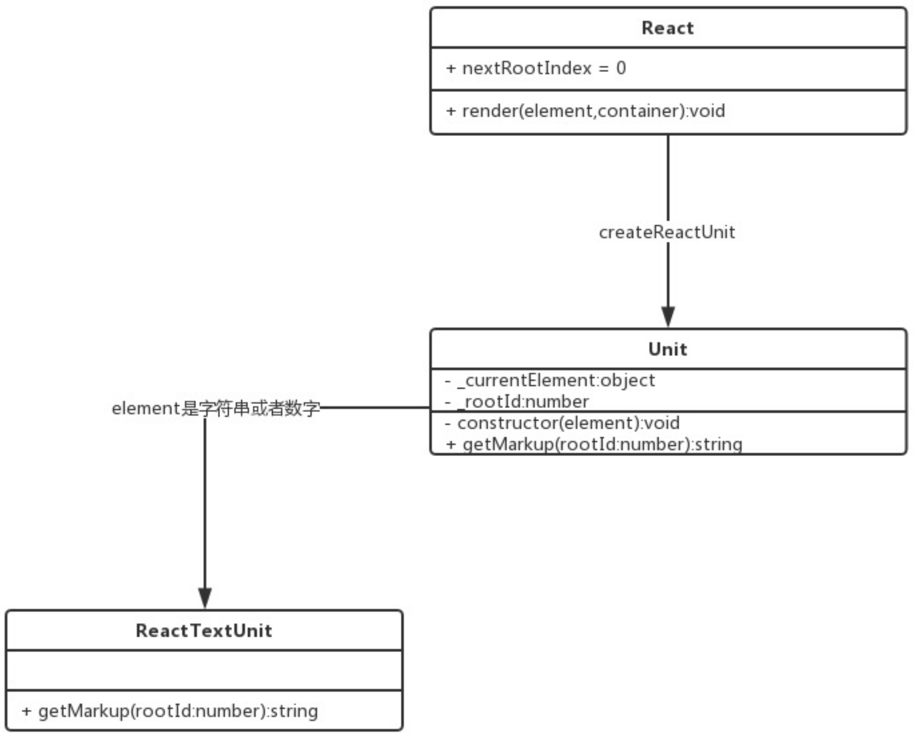
2.2.2 react.js

src\react.js

```
import $ from 'jquery';
let React = {
    rootIndex:0,
    render
}
function render(element,container){
   container.innerHTML = `${React.rootIndex}">${element}`;
}

export default React;
```

3. 重构

3.2 类图


React
+ nextRootIndex = 0
+ render(element,container):void
createReactUnit
Unit
- _currentElement:object
- _rootId:number
- constructor(element):void
+ getMarkup(rootId:number):string
element是字符串或者数字
ReactTextUnit
+ getMarkup(rootId:number):string

3.3 实现

3.3.1 index.js

src\index.js

```
import React from './react';
React.render('hello',document.getElementById('root'));
```

3.3.2 react\index.js

src\react\index.js

```
import $ from 'jquery';
import {createUnit} from './unit';
let React = {
    rootIndex:0,
    render
}
function render(element,container){
   let unit = createUnit(element);
   let markup = unit.getMarkUp(React.rootIndex);
   $(container).html(markup);
   $(document).trigger('mounted');
}

export default React;
```

3.3.3 react\unit.js

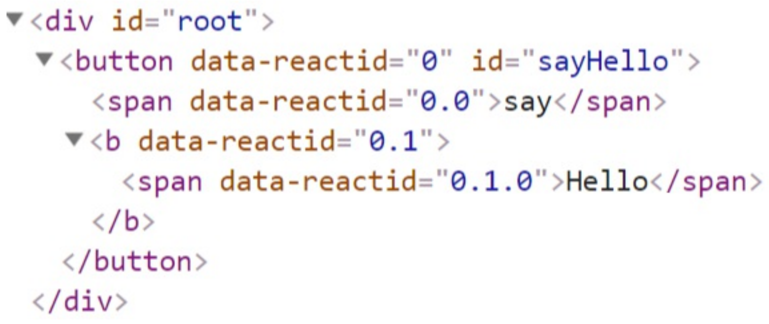
src\react\unit.js

```
class Unit {
    constructor(element){
        this._currentElement = element;
    }
    getMarkUp(){
        throw new Error('不能调用此方法');
    }
}
class TextUnit extends Unit{
    getMarkUp(reactid){
         this._reactid = reactid;

        return `${reactid}">${this._currentElement}`;
    }
}

function createUnit(element){
  if(typeof element =='string' || typeof element =='number'){
      return new TextUnit(element);
  }
}

export {
    createUnit
}
```

4. 渲染原生DOM组件

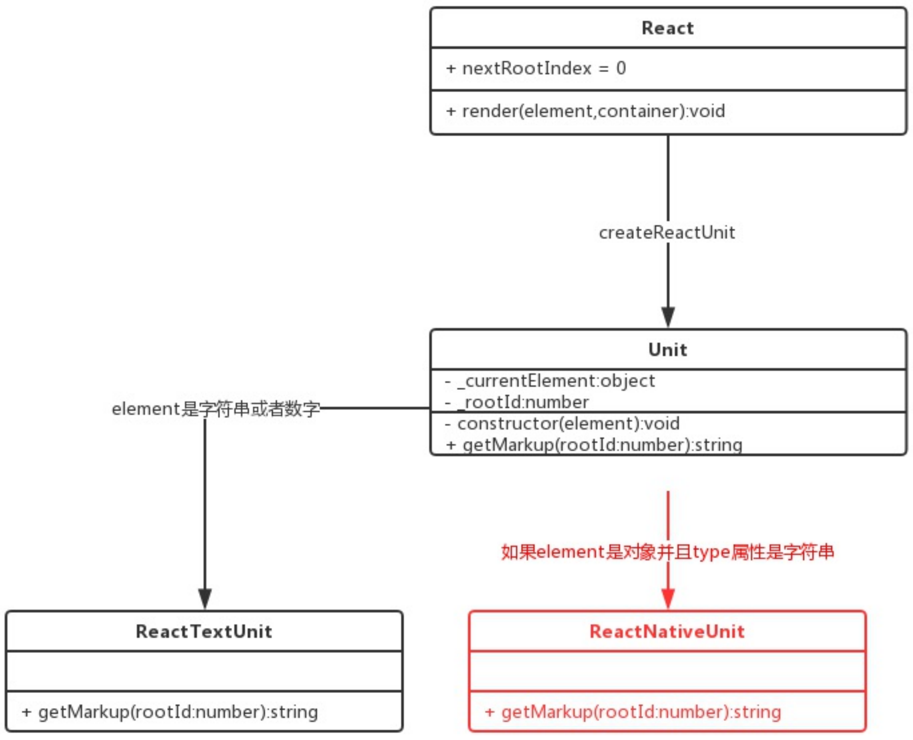
- babeljs (//https://babeljs.io/repl/)

4.1 渲染效果



```
▼<div id="root">
  ▼<button data-reactid="0" id="sayHello">
     <span data-reactid="0.0">say</span>
   ▼<b data-reactid="0.1">
       <span data-reactid="0.1.0">Hello</span>
     </b>
   </button>
 </div>
```

4.2 类图


React
+ nextRootIndex = 0
+ render(element,container):void
createReactUnit
Unit
- _currentElement:object
- _rootId:number
- constructor(element):void
+ getMarkup(rootId:number):string
element是字符串或者数字
如果element是对象并且type属性是字符串
ReactTextUnit
+ getMarkup(rootId:number):string
ReactNativeUnit
+ getMarkup(rootId:number):string

4.3 JSX语法

□

4.3.1 JSX

```
"sayHello" onClick={sayHello}>saycolor:'red'}}>Hello</b>button>
```

4.3.2 JavaScript

```
let element = React.createElement("button", {
    id: "sayHello",
    onClick: sayHello
  }, "say", React.createElement("b", {style:{color:'red'}}, "Hello"));
```

4.4 实现

4.4.1 index.js

src/index.js

```
import React from './react';
function sayHello(){
  alert('hello');
}
let element = React.createElement(
    'button',{id:'sayHello',onClick:sayHello},
    'say',
    React.createElement('b',{style:{color:'green'}},'hello'),
);
React.render(element,document.getElementById('root'));
```

4.4.2 react/index.js

src/react/index.js

```
import $ from 'jquery';
import {createUnit} from './unit';
+import {createElement} from './element';
let React = {
    rootIndex:0,
    render,
+    createElement
}
function render(element,container){
   let unit = createUnit(element);
   let markup = unit.getMarkUp(React.rootIndex);
   $(container).html(markup);
   $(document).trigger('mounted');//componentDidMount
}

export default React;
```

4.4.3 react/element.js

src/react/element.js

```
class Element{
    constructor(type,props){
        this.type = type;
        this.props = props;
    }
}
function createElement(type,props,...children){
    props=props||{};
    props.children  = children;
    return new Element(type,props);
}

export {
    createElement
}
```

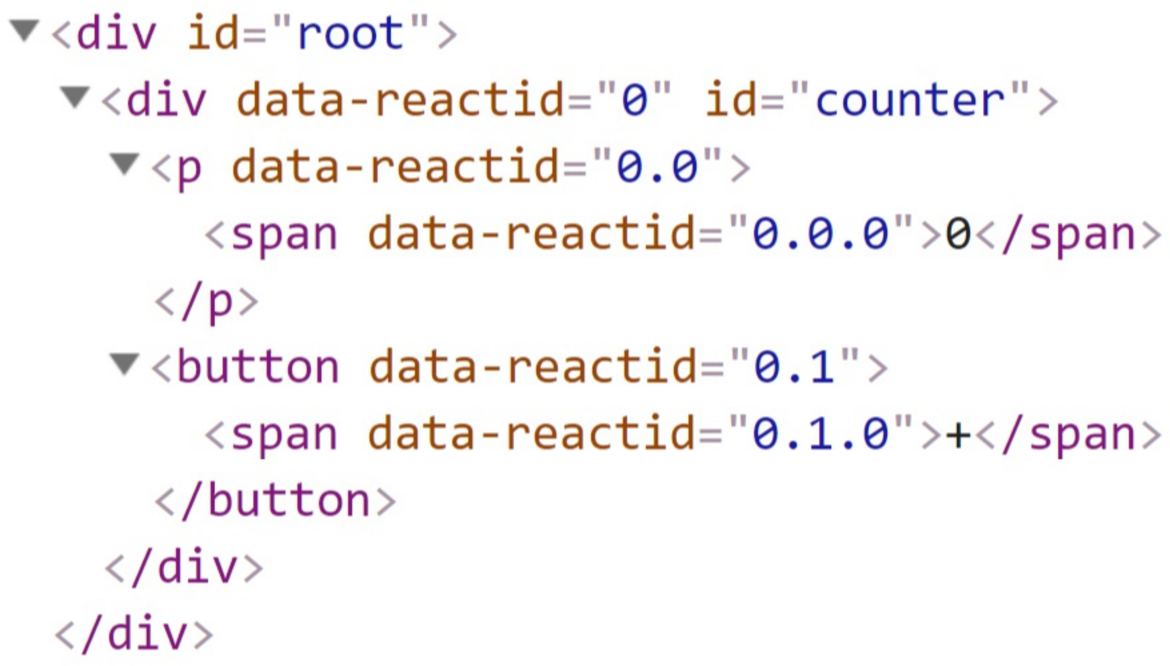
4.4.4 react/unit.js

src/react/unit.js

```
+import {Element} from './element';
+import $ from 'jquery';
class Unit {
    constructor(element){
        this._currentElement = element;
    }
    getMarkUp(){
        throw new Error('不能调用此方法');
    }
}
class TextUnit extends Unit{
    getMarkUp(reactid){
         this._reactid = reactid;//保存记录reactid
        //返回文本节点对应的HTML字符串
        return `${this._currentElement}`;
    }
}
+class NativeUnit extends Unit {
+     getMarkUp(reactid){
+         this._reactid = reactid;//保存记录reactid
+        //返回文本节点对应的HTML字符串
+        let {type,props} = this._currentElement;
+        let tagOpen = `
+        let tagClose = ``;
+        let content = '';
+        for(let propName in props){
+            if(/^on[A-Z]/.test(propName)){
+                let eventName = propName.slice(2).toLowerCase();
+                $(document).delegate(`[data-reactid="${reactid}"]`,`${eventName}.${reactid}`,props[propName]);
+            }else if(propName === 'style'){
+                let styleObj = props[propName];
+                let styles = Object.keys(styleObj).map(attr=>`${attr}:${styleObj[attr]}`).join(';');
+                tagOpen += (` style="${styles}" `);
+            }else if (propName === 'children'){
+                let children = props.children||[];
+                children.map((child,index)=>{
+                    let childUnit = createUnit(child);
+                    let childMarkUp = childUnit.getMarkUp(`${reactid}.${index}`);
+                    content += childMarkUp;
+                });
+            }else{
+                tagOpen += ` ${propName}=${props[propName]} `;
+            }
+        }
+        return tagOpen + '>' + content + tagClose;
+    }
+}
function createUnit(element){
  if(typeof element =='string' || typeof element =='number'){
      return new TextUnit(element);
  }
+  if(element instanceof Element && typeof element.type === 'string'){
+      return new NativeUnit(element);
+  }
}

export {
    createUnit
}
```

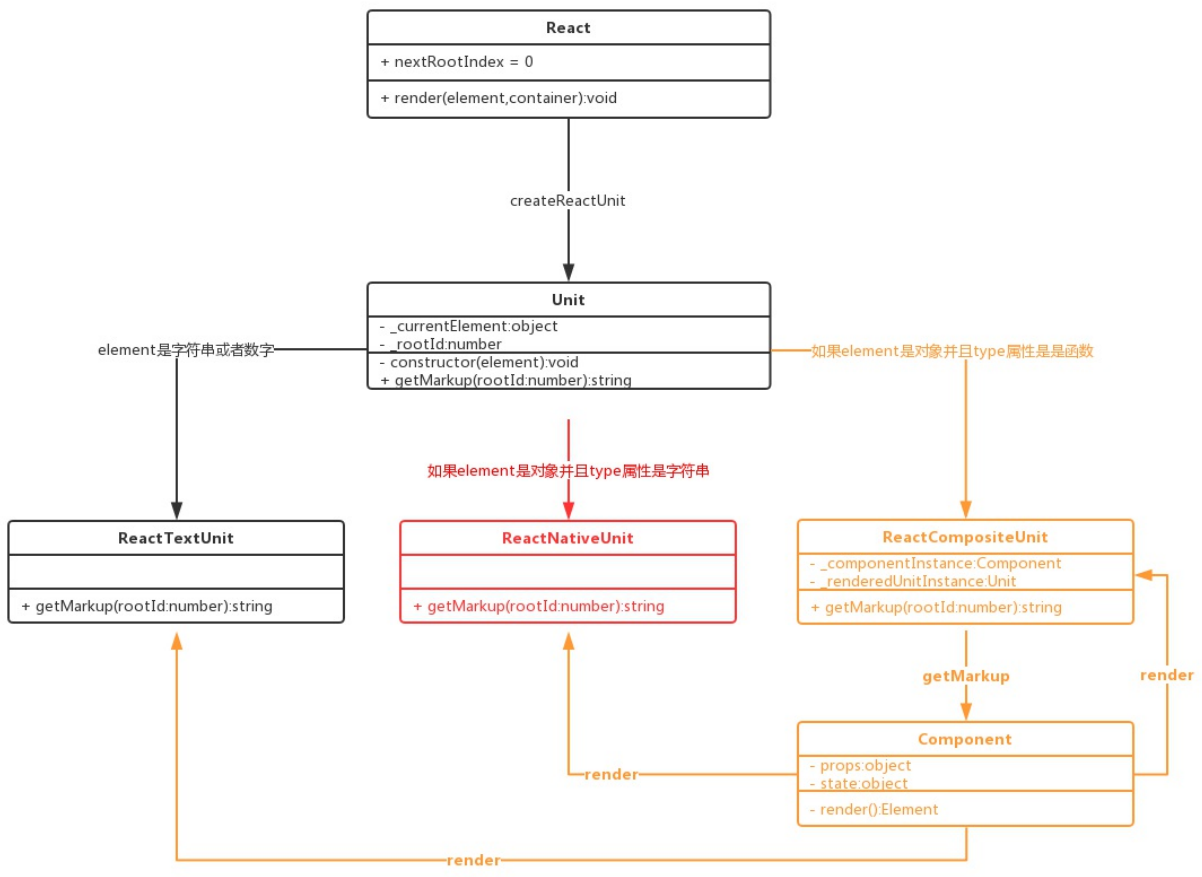
5. 渲染自定义组件

5.1 渲染效果



```
▼<div id="root">
  ▼<div data-reactid="0" id="counter">
    ▼<p data-reactid="0.0">
        <span data-reactid="0.0.0">0</span>
      </p>
    ▼<button data-reactid="0.1">
        <span data-reactid="0.1.0">+</span>
      </button>
    </div>
  </div>
```

5.2 类图


React
+ nextRootIndex = 0
+ render(element,container):void
createReactUnit
Unit
- _currentElement:object
- _rootId:number
- constructor(element):void
+ getMarkup(rootId:number):string
element是字符串或者数字
如果element是对象并且type属性是是函数
如果element是对象并且type属性是字符串
ReactTextUnit
+ getMarkup(rootId:number):string
ReactNativeUnit
+ getMarkup(rootId:number):string
ReactCompositeUnit
- _componentInstance:Component
- _renderedUnitInstance:Unit
+ getMarkup(rootId:number):string
getMarkup
render
Component
- props:object
- state:object
- render():Element
render
render

5.3 实现

5.3.1 src/index.js

```
import React from './react';
class Counter extends React.Component{
  constructor(props){
    super(props);
    this.state = {number:0};
  }
  componentWillMount(){
    console.log('Counter componentWillMount')
  }
  componentDidMount() {
     console.log('Counter componentDidMount')
  }
  handleClick = ()=>{
    this.setState({number:this.state.number+1});
  }
  render(){
    let p = React.createElement('p',{style:{color:'red'}},this.state.number);
    let button = React.createElement('button',{onClick:this.handleClick},'+');
    return React.createElement('div',{id:'counter'},p,button);
  }
}
let element = React.createElement(Counter);
React.render(element,document.getElementById('root'));
```

5.3.2 react/index.js

src/react/index.js

```
import $ from 'jquery';
import {createUnit} from './unit';
import {createElement} from './element';
+import {Component} from './component';
let React = {
    rootIndex:0,
    render,
    createElement,
+    Component
}
function render(element,container){
   let unit = createUnit(element);
   let markup = unit.getMarkUp(React.rootIndex);
   $(container).html(markup);
   $(document).trigger('mounted');//componentDidMount
}

export default React;
```

5.3.3 react/component.js

src/react/component.js

```
class Component{
    constructor(props){
        this.props = props;
    }
}
export {Component}
```

5.3.4 react/unit.js

src/react/unit.js

```
import {Element} from './element';
import $ from 'jquery';
class Unit {
    constructor(element){
        this._currentElement = element;
    }
    getMarkUp(){
        throw new Error('不能调用此方法');
    }
}
class TextUnit extends Unit{
    getMarkUp(reactid){
         this._reactid = reactid;//保存记录reactid
        //返回文本节点对应的HTML字符串
        return `${this._currentElement}`;
    }
}
class NativeUnit extends Unit {
     getMarkUp(reactid){
         this._reactid = reactid;//保存记录reactid
        //返回文本节点对应的HTML字符串
        let {type,props} = this._currentElement;
        let tagOpen = ``;
        let content = '';
        for(let propName in props){
            if(/^on[A-Z]/.test(propName)){
                let eventName = propName.slice(2).toLowerCase();
                $(document).delegate(`[data-reactid="${reactid}"]`,`${eventName}.${reactid}`,props[propName]);
            }else if(propName
                let styleObj = props[propName];
                let styles = Object.keys(styleObj).map(attr=>`${attr}:${styleObj[attr]}`).join(';');
                tagOpen += (` style="${styles}" `);
            }else if (propName
                let children = props.children||[];
                children.map((child,index)=>{
                    let childUnit = createUnit(child);
                    let childMarkUp = childUnit.getMarkUp(`${reactid}.${index}`);
                    content += childMarkUp;
                });
            }else{
                tagOpen += ` ${propName}=${props[propName]} `;
            }
        }
        return tagOpen + '>' + content + tagClose;
    }
}

+class CompositeUnit extends Unit{
+    getMarkUp(reactid){
+        this._reactid = reactid;
+        //type是一个自定义组件的类的定义
+        let {type:Component,props} = this._currentElement;
+        //创建Component类的实例
+        let componentInstance = new Component(props);
+        //组件将要渲染
+        componentInstance.componentWillMount&&componentInstance.componentWillMount();
+        //执行render方法获得虚拟DOM元素实例
+        let renderedElement = componentInstance.render();
+        //根据虚拟DOM元素得到unit,可能是TextUnit NativeUnit CompositeUnit
+        let renderedUnitInstance = this._renderedUnitInstance= createUnit(renderedElement);
+        //获得此unit的HTML标记字符串
+        let renderedMarkUp = renderedUnitInstance.getMarkUp(reactid);
+        //注册挂载完成的监听，越底层的组件越先监听，越先执行
+        $(document).on('mounted',()=>componentInstance.componentDidMount&&componentInstance.componentDidMount());
+        return renderedMarkUp;
+    }
+}

function createUnit(element){
  if(typeof element =='string' || typeof element =='number'){
      return new TextUnit(element);
  }
  if(element instanceof Element && typeof element.type
      return new NativeUnit(element);
  }
+  if(element instanceof Element && typeof element.type === 'function'){
+      return new CompositeUnit(element);
+  }
}

export {
    createUnit
}
```

6. 实现setState

6.1 src/index.js

```
import React from './react';
class Counter extends React.Component{
  constructor(props){
    super(props);
    this.state = {number:0};
  }
  componentWillMount(){
    console.log('Counter componentWillMount')
  }
  componentDidMount() {
     setInterval(() => {
       this.setState({number:this.state.number+1});
     }, 1000);
  }
  render(){
   return this.state.number;
  }
}
let element = React.createElement(Counter);
React.render(element,document.getElementById('root'));
```

6.2 react/component.js

src/react/component.js

```
class Component{
    constructor(props){
        this.props = props;
    }
    setState(partialState){
        this._currentUnit.update(null,partialState);
    }
}
export {Component}
```

6.3 react/unit.js

src/react/unit.js

```
import {Element} from './element';
import $ from 'jquery';
class Unit {
    constructor(element){
        this._currentElement = element;
    }
    getMarkUp(){
        throw new Error('不能调用此方法');
    }
}
class TextUnit extends Unit{
    getMarkUp(reactid){
         this._reactid = reactid;//保存记录reactid
        //返回文本节点对应的HTML字符串
        return `${this._currentElement}`;
    }
+    update(nextElement){
+        if(this._currentElement != nextElement){
+            this._currentElement = nextElement;
+            $(`[data-reactid="${this._reactid}"]`).html(this._currentElement);
+        }
+    }
}
class NativeUnit extends Unit {
     getMarkUp(reactid){
         this._reactid = reactid;//保存记录reactid
        //返回文本节点对应的HTML字符串
        let {type,props} = this._currentElement;
        let tagOpen = ``;
        let content = '';
        for(let propName in props){
            if(/^on[A-Z]/.test(propName)){
                let eventName = propName.slice(2).toLowerCase();
                $(document).delegate(`[data-reactid="${reactid}"]`,`${eventName}.${reactid}`,props[propName]);
            }else if(propName
                let styleObj = props[propName];
                let styles = Object.keys(styleObj).map(attr=>`${attr}:${styleObj[attr]}`).join(';');
                tagOpen += (` style="${styles}" `);
            }else if (propName
                let children = props.children||[];
                children.map((child,index)=>{
                    let childUnit = createUnit(child);
                    let childMarkUp = childUnit.getMarkUp(`${reactid}.${index}`);
                    content += childMarkUp;
                });
            }else{
                tagOpen += ` ${propName}=${props[propName]} `;
            }
        }
        return tagOpen + '>' + content + tagClose;
    }
}

class CompositeUnit extends Unit{
    //接收到新的更新，自定义组件传第二个参数，原生组件和text传处一个参数
+    update(nextElement,partialState){
+        //如果传过来了新的元素，则使用新的元素
+        this._currentElement = nextElement||this._currentElement;
+        //获取新的状态对象和属性对象
+        let nextState = this._componentInstance.state= Object.assign(this._componentInstance.state,partialState);
+        let nextProps = this._currentElement.props;
+        //如果shouldComponentUpdate返回了false则不需要继续更新
+        if(this._componentInstance.shouldComponentUpdate&&this._componentInstance.shouldComponentUpdate(nextProps,nextState)===false){return;}
+        //获得上次渲染出来的unit实例
+        let prevRenderedUnitInstance = this._renderedUnitInstance;
+        //从unit实例中获取
+        let prevRenderedElement = prevRenderedUnitInstance._currentElement;
+        //获取新的虚拟DOM
+        let nextRenderElement = this._componentInstance.render();
+        //进行domdiff对比
+        if(shouldDeepCompare(prevRenderedElement,nextRenderElement)){
+            //如果需要更新，则继续调用子节点的upate方法进行更新,传入新的element更新子节点
+            prevRenderedUnitInstance.update(nextRenderElement);
+            this._componentInstance.componentDidUpdate&&this._componentInstance.componentDidUpdate();
+        }else{
+            //如果发现不需要对比，干脆重新渲染
+            this._renderedUnitInstance =  createUnit(nextRenderElement);
+            let nextMarkUp = this._renderedUnitInstance.getMarkUp(this._reactid);
+            //替换整个节点
+            $(`[data-reactid="${this._reactid}"]`).replaceWith(nextMarkUp);
+        }

    }
    getMarkUp(reactid){
        this._reactid = reactid;
        //type是一个自定义组件的类的定义
        let {type:Component,props} = this._currentElement;
        //创建Component类的实例
+        let componentInstance = this._componentInstance = new Component(props);
        //组件实例关联上自己的unit实例
+        componentInstance._currentUnit  = this;
        //组件将要渲染
        componentInstance.componentWillMount&&componentInstance.componentWillMount();
        //执行render方法获得虚拟DOM元素实例
        let renderedElement = componentInstance.render();
        //根据虚拟DOM元素得到unit,可能是TextUnit NativeUnit CompositeUnit
        let renderedUnitInstance = this._renderedUnitInstance= createUnit(renderedElement);
        //获得此unit的HTML标记字符串
        let renderedMarkUp = renderedUnitInstance.getMarkUp(reactid);
        //注册挂载完成的监听，越底层的组件越先监听，越先执行
        $(document).on('mounted',()=>componentInstance.componentDidMount&&componentInstance.componentDidMount());
        return renderedMarkUp;
    }
}

+function shouldDeepCompare(prevElement,nextElement){
+    if(prevElement!==null && nextElement!=null){
+        let prevType = typeof prevElement;
+        let nextType = typeof nextElement;
+        //如果新老节点都是文本可以进行比较
+        if((prevType === 'string' ||prevType === 'number')&&(nextType === 'string' ||nextType === 'number')){
+            return true;
+        }
+        if(prevElement instanceof Element && nextElement instanceof Element){
+            return prevElement.type === nextElement.type;
+        }
+    }
+    return false;
+}
function createUnit(element){
  if(typeof element =='string' || typeof element =='number'){
      return new TextUnit(element);
  }
  if(element instanceof Element && typeof element.type
      return new NativeUnit(element);
  }
  if(element instanceof Element && typeof element.type
      return new CompositeUnit(element);
  }
}

export {
    createUnit
}
```

6.4 react/element.js

src/react/element.js

```
class Element{
    constructor(type,props){
        this.type = type;
+        this.key = props.key;
        this.props = props;
    }
}
```

7. 对比属性

- 实现点击加1功能

7.1 src/index.js

src/index.js

```
import React from './react';
class Counter extends React.Component{
  constructor(props){
    super(props);
    this.state = {number:0};
  }
  componentWillMount(){
    console.log('Counter componentWillMount')
  }
  componentDidMount() {
     console.log('Counter componentDidMount')
  }
  handleClick= ()=>{
    this.setState({number:this.state.number+1});
  }
  render(){
   let p = React.createElement('p',{},this.state.number);
   let button = React.createElement('button',{onClick:this.handleClick},'+');
   return React.createElement('div',{id:'counter',style:
{color:this.state.number%2===0?'red':'green',backgroundColor:this.state.number%2===0?'green':'red'}},p,button);
  }
}
let element = React.createElement(Counter);
React.render(element,document.getElementById('root'));
```

7.2 react/unit.js

src/react/unit.js

```
import {Element} from './element';
import $ from 'jquery';
class Unit {
    constructor(element){
        this._currentElement = element;
    }
    getMarkUp(){
        throw new Error('不能调用此方法');
    }
}
class TextUnit extends Unit{
    getMarkUp(reactid){
         this._reactid = reactid;//保存记录reactid
        //返回文本节点对应的HTML字符串
        return `${this._currentElement}`;
    }
    update(nextElement){
        if(this._currentElement != nextElement){
            this._currentElement = nextElement;
            $(`[data-reactid="${this._reactid}"]`).html(this._currentElement);
        }
    }

}
class NativeUnit extends Unit {
     getMarkUp(reactid){
         this._reactid = reactid;//保存记录reactid
        //返回文本节点对应的HTML字符串
        let {type,props} = this._currentElement;
        let tagOpen = ``;
        let content = '';
        for(let propName in props){
            if(/^on[A-Z]/.test(propName)){
                let eventName = propName.slice(2).toLowerCase();
                $(document).delegate(`[data-reactid="${reactid}"]`,`${eventName}.${reactid}`,props[propName]);
            }else if(propName
+                let styleObj = props[propName];
+                let styles = Object.keys(styleObj).map(attr=>{
+                    let attrName = attr.replace(/([A-Z])/g,function(matched,group){
+                        return `-${group.toLowerCase()}`;
+                    })
+                    return `${attrName}:${styleObj[attr]}`;
+                }).join(';');
+                tagOpen += (` style="${styles}" `);
            }else if (propName
                let children = props.children||[];
                children.map((child,index)=>{
                    let childUnit = createUnit(child);
                    let childMarkUp = childUnit.getMarkUp(`${reactid}.${index}`);
                    content += childMarkUp;
                });
            }else{
                tagOpen += ` ${propName}=${props[propName]} `;
            }
        }
        return tagOpen + '>' + content + tagClose;
    }
+    update(nextElement){
+        let oldProps = this._currentElement.props;
+        let newProps = nextElement.props;
+        this.updateDOMproperties(oldProps,newProps);
+        //this.updateDOMChildren(nextElement.props.children);
+    }
+    updateDOMproperties(oldProps,newProps){
+        let propName;
+        //把新属性对象上没有属性给删除掉
+        for(propName in oldProps){
+            if(!newProps.hasOwnProperty(propName)){
+                $(`[data-reactid="${this._reactid}"]`).removeAttr(propName);
+            }
+            if(/^on[A-Z]/.test(propName)){
+                $(document).undelegate(`.${this.reactid}`);
+            }
+        }
+        for(propName in newProps){
```

```
+            if(propName == 'children'){
+
+            }else if(/^on[A-Z]/.test(propName)){
+                let eventName = propName.slice(2).toLowerCase();
+                $(document).delegate(`[data-reactid="${this._reactid}"]`,`${eventName}.${this._reactid}`,newProps[propName]);
+            }else if(propName === 'style'){
+                let styleObj = newProps[propName];
+                Object.entries(styleObj).forEach(([attr,value])=>{
+                  $(`[data-reactid="${this._reactid}"]`).css(attr,value);
+                })
+            }else{
+                $(`[data-reactid="${this._reactid}"]`).prop(propName,newProps[propName]);
+            }
+        }
+    }
+}

class CompositeUnit extends Unit{
    //接收到新的更新，自定义组件传第二个参数，原生组件和text传处一个参数
    update(nextElement,partialState){
        //如果传过来了新的元素，则使用新的元素
        this._currentElement = nextElement||this._currentElement;
        //获取新的状态对象和属性对象
        let nextState = this._componentInstance.state= Object.assign(this._componentInstance.state,partialState);
        let nextProps = this._currentElement.props;
        //如果shouldComponentUpdate返回了false则不需要继续更新
        if(this._componentInstance.shouldComponentUpdate&&this._componentInstance.shouldComponentUpdate(nextProps,nextState)
        //获得上次渲染出来的unit实例
        let prevRenderedUnitInstance = this._renderedUnitInstance;
        //从unit实例中获取
        let prevRenderedElement = prevRenderedUnitInstance._currentElement;
        //获取新的虚拟DOM
        let nextRenderElement = this._componentInstance.render();
        //进行domdiff对比
        if(shouldDeepCompare(prevRenderedElement,nextRenderElement)){
            //如果需要更新，则继续调用子节点的upate方法进行更新,传入新的element更新子节点
            prevRenderedUnitInstance.update(nextRenderElement);
            this._componentInstance.componentDidUpdate&&this._componentInstance.componentDidUpdate();
        }else{
            //如果发现不需要对比，干脆重新渲染
            this._renderedUnitInstance =  createUnit(nextRenderElement);
            let nextMarkUp = this._renderedUnitInstance.getMarkUp(this._reactid);
            //替换整个节点
            $(`[data-reactid="${this._reactid}"]`).replaceWith(nextMarkUp);
        }

    }
    getMarkUp(reactid){
        this._reactid = reactid;
        //type是一个自定义组件的类的定义
        let {type:Component,props} = this._currentElement;
        //创建Component类的实例
        let componentInstance = this._componentInstance = new Component(props);
        //组件实例关联上自己的unit实例
        componentInstance._currentUnit  = this;
        //组件将要渲染
        componentInstance.componentWillMount&&componentInstance.componentWillMount();
        //执行render方法获得虚拟DOM元素实例
        let renderedElement = componentInstance.render();
        //根据虚拟DOM元素得到unit,可能是TextUnit NativeUnit CompositeUnit
        let renderedUnitInstance = this._renderedUnitInstance= createUnit(renderedElement);
        //获得此unit的HTML标记字符串
        let renderedMarkUp = renderedUnitInstance.getMarkUp(reactid);
        //注册挂载完成的监听，越底层的组件越先监听，越先执行
        $(document).on('mounted',()=>componentInstance.componentDidMount&&componentInstance.componentDidMount());
        return renderedMarkUp;
    }
}

function shouldDeepCompare(prevElement,nextElement){
   if(prevElement!==null && nextElement!=null){
       let prevType = typeof prevElement;
       let nextType = typeof nextElement;
       //如果新老节点都是文本可以进行比较
       if((prevType
           return true;
       }
       if(prevElement instanceof Element && nextElement instanceof Element){
           return prevElement.type
       }
   }
   return false;
}
function createUnit(element){
  if(typeof element =='string' || typeof element =='number'){
      return new TextUnit(element);
  }
  if(element instanceof Element && typeof element.type
      return new NativeUnit(element);
  }
  if(element instanceof Element && typeof element.type
      return new CompositeUnit(element);
  }
}

export {
    createUnit
}
```

8. 对比子元素

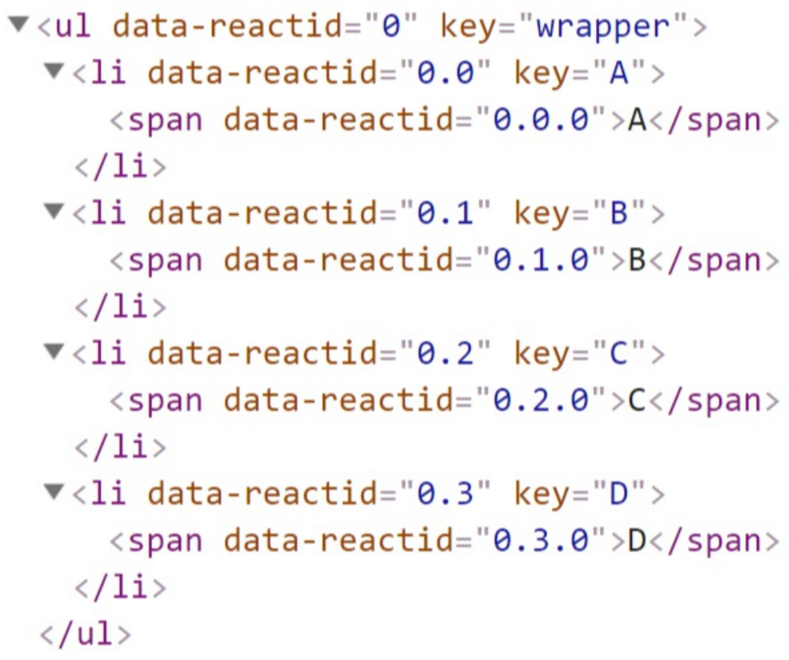
8.1 src/unit.js

src\unit.js

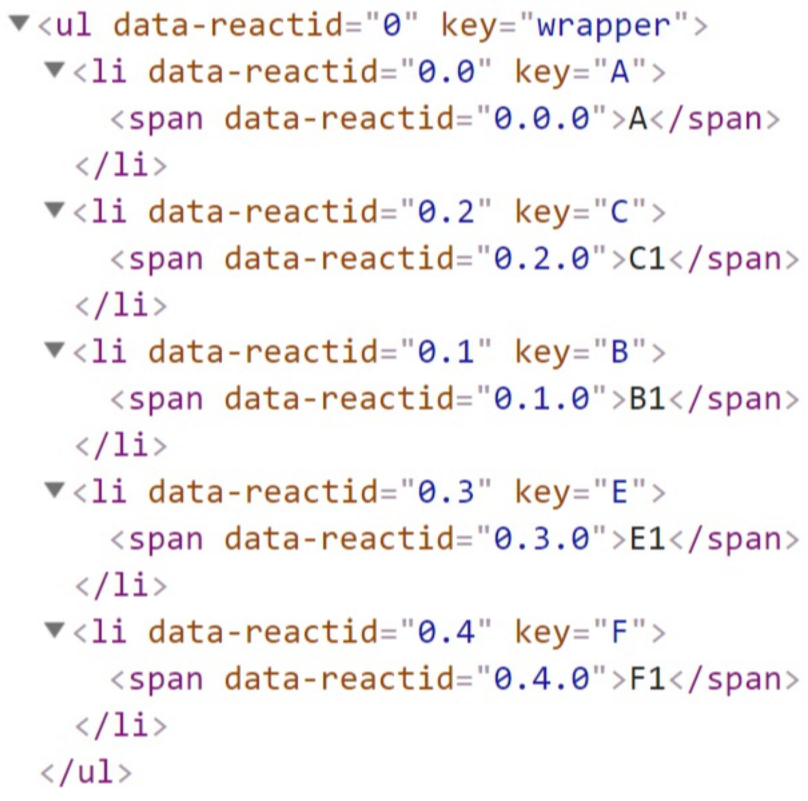
```
import {Element} from './element';
import $ from 'jquery';
+ let diffQueue = [];
class Unit {
    constructor(element){
        this._currentElement = element;
    }
    getMarkUp(){
        throw new Error('不能调用此方法');
    }
}
class TextUnit extends Unit{
    getMarkUp(reactid){
         this._reactid = reactid;//保存记录reactid
        //返回文本节点对应的HTML字符串
        return `${this._currentElement}`;
    }
    update(nextElement){
        if(this._currentElement != nextElement){
            this._currentElement = nextElement;
            $(`[data-reactid="${this._reactid}"]`).html(this._currentElement);
        }
    }

}
class NativeUnit extends Unit {
     getMarkUp(reactid){
         this._reactid = reactid;//保存记录reactid
        //返回文本节点对应的HTML字符串
        let {type,props} = this._currentElement;
        let tagOpen = ``;
        let content = '';
+        let renderedChildUnits=[];
        for(let propName in props){
            if(/^on[A-Z]/.test(propName)){
                let eventName = propName.slice(2).toLowerCase();
                $(document).delegate(`[data-reactid="${reactid}"]`,`${eventName}.${reactid}`,props[propName]);
            }else if(propName
                let styleObj = props[propName];
                let styles = Object.keys(styleObj).map(attr=>{
                    let attrName = attr.replace(/([A-Z])/g,function(matched,group){
                        return `-${group.toLowerCase()}`;
                    })
                    return `${attrName}:${styleObj[attr]}`;
                }).join(';');
                tagOpen += (` style="${styles}" `);
            }else if (propName
                let children = props.children||[];
                children.map((child,index)=>{
                    let childUnit = createUnit(child);
+                    renderedChildUnits.push(childUnit);
                    let childMarkUp = childUnit.getMarkUp(`${reactid}.${index}`);
                    content += childMarkUp;
                });
            }else{
                tagOpen += ` ${propName}=${props[propName]} `;
            }
        }
+        this._renderedChildUnits = renderedChildUnits;
        return tagOpen + '>' + content + tagClose;
    }
    update(nextElement){
        let oldProps = this._currentElement.props;
        let newProps = nextElement.props;
        this.updateDOMproperties(oldProps,newProps);
+        this.updateDOMChildren(nextElement.props.children);
    }
+    //对比子元素
+    updateDOMChildren(newChildrenElements){
+        this.diff(diffQueue,newChildrenElements);
+    }
+    diff(diffQueue,newChildrenElements){
+        let oldChildUnitsMap = this.getChildrenMap(this._renderedChildUnits);
+        let newChildren = this.getNewChildren(oldChildUnitsMap,newChildrenElements);
+    }
+    getNewChildren(oldChildUnitsMap,newChildrenElements){
+        let newChildren = [];
+        newChildrenElements.forEach((newElement,index)=>{
+            let newKey = newElement.key||index.toString();
+            let oldUnit = oldChildUnitsMap[newKey];//获得老的unit
+            let oldElement = oldUnit&&oldUnit._currentElement;//获得老的element
+            if(shouldDeepCompare(oldElement,newElement)){//如果可以更进一步深比较
+                oldUnit.update(newElement);
+                newChildren.push(oldUnit);
+            }else{
+                let newChildUnit = createUnit(newElement);//如果不需要深比较则直接创建新的unit
+                newChildren.push(newChildUnit);
+            }
+        });
+        return newChildren;
+    }
+    getChildrenMap(childUnits=[]){
+        let map = {};
+        for(let i=0;i
+            let key = childUnits[i].key||i.toString();
+            map[key]=childUnits[i];
+        }
+        return map;
+    }
    updateDOMproperties(oldProps,newProps){
        let propName;
        //把新属性对象上没有属性给删除掉
        for(propName in oldProps){
            if(!newProps.hasOwnProperty(propName)){
                $(`[data-reactid="${this._reactid}"]`).removeAttr(propName);
            }
            if(/^on[A-Z]/.test(propName)){
                $(document).undelegate(`.${this._reactid}`);
            }
        }
        for(propName in newProps){
            if(propName == 'children'){

            }else if(/^on[A-Z]/.test(propName)){
                let eventName = propName.slice(2).toLowerCase();
                $(document).undelegate(`.${this._reactid}`);
                $(document).delegate(`[data-reactid="${this._reactid}"]`,`${eventName}.${this._reactid}`,newProps[propName]);
            }else if(propName
                let styleObj = newProps[propName];
                Object.entries(styleObj).forEach(([attr,value])=>{
                  $(`[data-reactid="${this._reactid}"]`).css(attr,value);
                })
            }else{
                $(`[data-reactid="${this._reactid}"]`).prop(propName,newProps[propName]);
            }
        }
    }
}

class CompositeUnit extends Unit{
    //接收到新的更新，自定义组件传第二个参数，原生组件和text传处一个参数
    update(nextElement,partialState){
        //如果传过来了新的元素，则使用新的元素
        this._currentElement = nextElement||this._currentElement;
        //获取新的状态对象和属性对象
        let nextState = this._componentInstance.state= Object.assign(this._componentInstance.state,partialState);
        let nextProps = this._currentElement.props;
        //如果shouldComponentUpdate返回了false则不需要继续更新
        if(this._componentInstance.shouldComponentUpdate&&this._componentInstance.shouldComponentUpdate(nextProps,nextState)
        //获得上次渲染出来的unit实例
        let prevRenderedUnitInstance = this._renderedUnitInstance;
        //从unit实例中获取
        let prevRenderedElement = prevRenderedUnitInstance._currentElement;
        //获取新的虚拟DOM
        let nextRenderElement = this._componentInstance.render();
        //进行domdiff对比
        if(shouldDeepCompare(prevRenderedElement,nextRenderElement)){
            //如果需要更新，则继续调用子节点的upate方法进行更新,传入新的element更新子节点
            prevRenderedUnitInstance.update(nextRenderElement);
            this._componentInstance.componentDidUpdate&&this._componentInstance.componentDidUpdate();
        }else{
            //如果发现不需要对比，干脆重新渲染
            this._renderedUnitInstance =  createUnit(nextRenderElement);
            let nextMarkUp = this._renderedUnitInstance.getMarkUp(this._reactid);
            //替换整个节点
            $(`[data-reactid="${this._reactid}"]`).replaceWith(nextMarkUp);
        }

    }
    getMarkUp(reactid){
        this._reactid = reactid;
        //type是一个自定义组件的类的定义
        let {type:Component,props} = this._currentElement;
        //创建Component类的实例
        let componentInstance = this._componentInstance = new Component(props);
        //组件实例关联上自己的unit实例
        componentInstance._currentUnit  = this;
        //组件将要渲染
        componentInstance.componentWillMount&&componentInstance.componentWillMount();
        //执行render方法获得虚拟DOM元素实例
        let renderedElement = componentInstance.render();
        //根据虚拟DOM元素得到unit,可能是TextUnit NativeUnit CompositeUnit
        let renderedUnitInstance = this._renderedUnitInstance= createUnit(renderedElement);
        //获得此unit的HTML标记字符串
        let renderedMarkUp = renderedUnitInstance.getMarkUp(reactid);
        //注册挂载完成的监听，越底层的组件越先监听，越先执行
        $(document).on('mounted',()=>componentInstance.componentDidMount&&componentInstance.componentDidMount());
        return renderedMarkUp;
    }
}

function shouldDeepCompare(prevElement,nextElement){
   if(prevElement!==null && nextElement!=null){
       let prevType = typeof prevElement;
       let nextType = typeof nextElement;
       //如果新老节点都是文本可以进行比较
       if((prevType
           return true;
       }
       if(prevElement instanceof Element && nextElement instanceof Element){
           return prevElement.type
       }
   }
   return false;
}
function createUnit(element){
  if(typeof element =='string' || typeof element =='number'){
      return new TextUnit(element);
  }
  if(element instanceof Element && typeof element.type
      return new NativeUnit(element);
  }
  if(element instanceof Element && typeof element.type
      return new CompositeUnit(element);
  }
}
```

```
export {
    createUnit
}
```

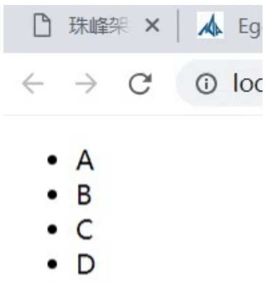
9. 获得补丁数组 #



```
▼<ul data-reactid="0" key="wrapper">
  ▼<li data-reactid="0.0" key="A">
     <span data-reactid="0.0.0">A</span>
   </li>
  ▼<li data-reactid="0.1" key="B">
     <span data-reactid="0.1.0">B</span>
   </li>
  ▼<li data-reactid="0.2" key="C">
     <span data-reactid="0.2.0">C</span>
   </li>
  ▼<li data-reactid="0.3" key="D">
     <span data-reactid="0.3.0">D</span>
   </li>
 </ul>
```



```
▼<ul data-reactid="0" key="wrapper">
  ▼<li data-reactid="0.0" key="A">
     <span data-reactid="0.0.0">A</span>
   </li>
  ▼<li data-reactid="0.2" key="C">
     <span data-reactid="0.2.0">C1</span>
   </li>
  ▼<li data-reactid="0.1" key="B">
     <span data-reactid="0.1.0">B1</span>
   </li>
  ▼<li data-reactid="0.3" key="E">
     <span data-reactid="0.3.0">E1</span>
   </li>
  ▼<li data-reactid="0.4" key="F">
     <span data-reactid="0.4.0">F1</span>
   </li>
 </ul>
```



- A
- B
- C
- D

9.1 src/index.js

src/index.js

```
import React from './react';
class Counter extends React.Component{
  constructor(props){
    super(props);
    this.state = {odd:true};
  }
  componentDidMount(){
   setTimeout(()=>{
    this.setState({odd:!this.state.odd});
   },1000);
  }
  render(){
    if(this.state.odd){
      return React.createElement('ul',{key:'wrapper'},
        React.createElement('li',{key:'A'},'A'),
        React.createElement('li',{key:'B'},'B'),
        React.createElement('li',{key:'C'},'C'),
        React.createElement('li',{key:'D'},'D'),
      );
    }
    return React.createElement('ul',{key:'wrapper'},
      React.createElement('li',{key:'A'},'A1'),
      React.createElement('li',{key:'C'},'C1'),
      React.createElement('li',{key:'B'},'B1'),
      React.createElement('li',{key:'E'},'E1'),
      React.createElement('li',{key:'F'},'F1')
      );
  }
}
let element = React.createElement(Counter);
React.render(element,document.getElementById('root'));
```

9.2 src/react/unit.js

src/react/unit.js

```
import {Element} from './element';
import $ from 'jquery';
+import types from './types';
+let diffQueue = [];
+let updateDepth=0;
class Unit {
    constructor(element){
        this._currentElement = element;
    }
    getMarkUp(){
        throw new Error('不能调用此方法');
    }
}
class TextUnit extends Unit{
    getMarkUp(reactid){
         this._reactid = reactid;//保存记录reactid
        //返回文本节点对应的HTML字符串
        return `${this._currentElement}`;
    }
    update(nextElement){
        if(this._currentElement != nextElement){
            this._currentElement = nextElement;
            $(`[data-reactid="${this._reactid}"]`).html(this._currentElement);
        }
    }

}
class NativeUnit extends Unit {
     getMarkUp(reactid){
         this._reactid = reactid;//保存记录reactid
        //返回文本节点对应的HTML字符串
        let {type,props} = this._currentElement;
        let tagOpen = ``;
        let content = '';
        let renderedChildUnits=[];
        for(let propName in props){
            if(/^on[A-Z]/.test(propName)){
                let eventName = propName.slice(2).toLowerCase();
                $(document).delegate(`[data-reactid="${reactid}"]`,`${eventName}.${reactid}`,props[propName]);
            }else if(propName
                let styleObj = props[propName];
                let styles = Object.keys(styleObj).map(attr=>{
                    let attrName = attr.replace(/([A-Z])/g,function(matched,group){
                        return `-${group.toLowerCase()}`;
                    })
                    return `${attrName}:${styleObj[attr]}`;
                }).join(';');
                tagOpen += (` style="${styles}" `);
            }else if (propName
                let children = props.children||[];
                children.map((child,index)=>{
                    let childUnit = createUnit(child);
                    childUnit._mountIndex = index;
                    renderedChildUnits.push(childUnit);
                    let childMarkUp = childUnit.getMarkUp(`${reactid}.${index}`);
                    content += childMarkUp;
                });
            }else{
                tagOpen += ` ${propName}=${props[propName]} `;
            }
        }
        this._renderedChildUnits = renderedChildUnits;
        return tagOpen + '>' + content + tagClose;
    }
    update(nextElement){
        let oldProps = this._currentElement.props;
        let newProps = nextElement.props;
        this.updateDOMproperties(oldProps,newProps);
        this.updateDOMChildren(nextElement.props.children);
    }
    //对比子元素
+    updateDOMChildren(newChildrenElements){
+        updateDepth++;
+        this.diff(diffQueue,newChildrenElements);
+        updateDepth--;
+        if(updateDepth===0){
+            console.log('diffQueue',diffQueue);
+            diffQueue=[];
+        }
+    }
+    diff(diffQueue,newChildrenElements){
+        let oldChildUnitsMap = this.getChildrenMap(this._renderedChildUnits);
+        let {newChildrenMap,newChildren} = this.getNewChildren(oldChildUnitsMap,newChildrenElements);
+        // lastIndex里存放着被复用的子元素的最大索引
+        let lastIndex = 0;
+        for(let i=0;i
+            let newChild = newChildren[i];//取得新元素
+            let newKey = (newChild._currentElement.props&&newChild._currentElement.key)||i.toString();//取得新key
+            let oldChild = oldChildUnitsMap[newKey];
+            if(oldChild === newChild){
+                if(oldChild._mountIndex < lastIndex){
+                    diffQueue.push({
+                        parentId:this._reactid,
+                        parentNode:$(`[data-reactid="${this._reactid}"]`),
+                        type:types.MOVE,
+                        fromIndex:oldChild._mountIndex,
+                        toIndex:i
+                    });
+                }
+                lastIndex = Math.max(oldChild._mountIndex,lastIndex);
+                //否则根本不用移动，直接修改挂载索引为新索引i即可
+            }else{
+                if(oldChild){
+                    diffQueue.push({
+                        parentId:this._reactid,
+                        parentNode:$(`[data-reactid="${this._reactid}"]`),
+                        type:types.REMOVE,
+                        fromIndex:oldChild._mountIndex
+                    });
+                    $(document).undelegate(`.${oldChild._reactid}`);
+                }
+                 diffQueue.push({
+                        parentId:this._reactid,
+                        parentNode:$(`[data-reactid="${this._reactid}"]`),
+                        type:types.INSERT,
+                        toIndex:i,
+                        markUp:newChild.getMarkUp(`${this._reactid}.${i}`)
+                });
+            }
+            newChild._mountIndex = i;
+        }
+        for(let oldKey in oldChildUnitsMap){
+            if(!newChildrenMap.hasOwnProperty(oldKey)){
+                let oldChild = oldChildUnitsMap[oldKey];
+                diffQueue.push({
+                        parentId:this._reactid,
+                        parentNode:$(`[data-reactid="${this._reactid}"]`),
+                        type:types.REMOVE,
+                        fromIndex:oldChild._mountIndex
+                });
+            }
+        }
+    }
+    getNewChildren(oldChildUnitsMap,newChildrenElements){
+        let newChildren = [];
+        let newChildrenMap={};
+        newChildrenElements.forEach((newElement,index)=>{
+            let newKey = newElement.key||index.toString();
+            let oldUnit = oldChildUnitsMap[newKey];//获得老的unit
+            let oldElement = oldUnit&&oldUnit._currentElement;//获得老的element
+            if(shouldDeepCompare(oldElement,newElement)){//如果可以更进一步深比较
+                oldUnit.update(newElement);
+                newChildren.push(oldUnit);
+                newChildrenMap[newKey]=oldUnit;
+            }else{
+                let newChildUnit = createUnit(newElement);//如果不需要深比较则直接创建新的unit
+                newChildren.push(newChildUnit);
+                 newChildrenMap[newKey]=newChildUnit;
+            }
+        });
+        return {newChildrenMap,newChildren};
+    }
+    getChildrenMap(childUnits=[]){
+        let map = {};
+        for(let i=0;i
+            let key = (childUnits[i]._currentElement.props&&childUnits[i]._currentElement.props.key)||i.toString();
+            map[key]=childUnits[i];
+        }
+        return map;
+    }
+    updateDOMproperties(oldProps,newProps){
+        let propName;
+        //把新属性对象上没有属性给删除掉
+        for(propName in oldProps){
+            if(!newProps.hasOwnProperty(propName)){
+                $(`[data-reactid="${this._reactid}"]`).removeAttr(propName);
+            }
+            if(/^on[A-Z]/.test(propName)){
+                $(document).undelegate(`.${this._reactid}`);
+            }
+        }
+        for(propName in newProps){
+            if(propName == 'children'){
+
+            }else if(/^on[A-Z]/.test(propName)){
+                let eventName = propName.slice(2).toLowerCase();
+                $(document).undelegate(`.${this._reactid}`);
+                $(document).delegate(`[data-reactid="${this._reactid}"]`,`${eventName}.${this._reactid}`,newProps[propName]);
+            }else if(propName === 'style'){
+                let styleObj = newProps[propName];
+                Object.entries(styleObj).forEach(([attr,value])=>{
+                  $(`[data-reactid="${this._reactid}"]`).css(attr,value);
+                })
+            }else{
+                $(`[data-reactid="${this._reactid}"]`).prop(propName,newProps[propName]);
+            }
+        }
+    }
+}

class CompositeUnit extends Unit{
    //接收到新的更新，自定义组件传第二个参数，原生组件和text传处一个参数
    update(nextElement,partialState){
        //如果传过来了新的元素，则使用新的元素
        this._currentElement = nextElement||this._currentElement;
        //获取新的状态对象和属性对象
        let nextState = this._componentInstance.state= Object.assign(this._componentInstance.state,partialState);
        let nextProps = this._currentElement.props;
        //如果shouldComponentUpdate返回了false则不需要继续更新
        if(this._componentInstance.shouldComponentUpdate&&this._componentInstance.shouldComponentUpdate(nextProps,nextState)
        //获得上次渲染出来的unit实例
        let prevRenderedUnitInstance = this._renderedUnitInstance;
        //从unit实例中获取
        let prevRenderedElement = prevRenderedUnitInstance._currentElement;
        //获取新的虚拟DOM
        let nextRenderElement = this._componentInstance.render();
        //进行domdiff对比
        if(shouldDeepCompare(prevRenderedElement,nextRenderElement)){
            //如果需要更新，则继续调用子节点的upate方法进行更新,传入新的element更新子节点
            prevRenderedUnitInstance.update(nextRenderElement);
            this._componentInstance.componentDidUpdate&&this._componentInstance.componentDidUpdate();
        }else{
            //如果发现不需要对比，干脆重新渲染
            this._renderedUnitInstance =  createUnit(nextRenderElement);
            let nextMarkUp = this._renderedUnitInstance.getMarkUp(this._reactid);
            //替换整个节点
            $(`[data-reactid="${this._reactid}"]`).replaceWith(nextMarkUp);
        }

    }
    getMarkUp(reactid){
        this._reactid = reactid;
        //type是一个自定义组件的类的定义
        let {type:Component,props} = this._currentElement;
        //创建Component类的实例
        let componentInstance = this._componentInstance = new Component(props);
        //组件实例关联上自己的unit实例
        componentInstance._currentUnit  = this;
        //组件将要渲染
        componentInstance.componentWillMount&&componentInstance.componentWillMount();
        //执行render方法获得虚拟DOM元素实例
        let renderedElement = componentInstance.render();
        //根据虚拟DOM元素得到unit,可能是TextUnit NativeUnit CompositeUnit
        let renderedUnitInstance = this._renderedUnitInstance= createUnit(renderedElement);
        //获得此unit的HTML标记字符串
        let renderedMarkUp = renderedUnitInstance.getMarkUp(reactid);
        //注册挂载完成的监听，越底层的组件越先监听，越先执行
        $(document).on('mounted',()=>componentInstance.componentDidMount&&componentInstance.componentDidMount());
        return renderedMarkUp;
    }
}

function shouldDeepCompare(prevElement,nextElement){
   if(prevElement!==null && nextElement!=null){
       let prevType = typeof prevElement;
       let nextType = typeof nextElement;
       //如果新老节点都是文本可以进行比较
       if((prevType
           return true;
       }
       if(prevElement instanceof Element && nextElement instanceof Element){
           return prevElement.type
        }
    }
    return false;
}
function createUnit(element){
  if(typeof element =='string' || typeof element =='number'){
      return new TextUnit(element);
  }
  if(element instanceof Element && typeof element.type
      return new NativeUnit(element);
  }
  if(element instanceof Element && typeof element.type
      return new CompositeUnit(element);
  }
}

export {
    createUnit
}
```

9.3 types.js

src\types.js

```
export default  {
    MOVE:'MOVE',
    INSERT:"INSERT",
    REMOVE:"REMOVE"
}
```

10. 打补丁

10.1 src/index.js

src/index.js

```
import React from './react';
class Counter extends React.Component{
  constructor(props){
    super(props);
    this.state = {odd:true};
  }
  componentDidMount(){
   setTimeout(()=>{
    this.setState({odd:!this.state.odd});
   },5000);
  }
  render(){
    if(this.state.odd){
      return React.createElement('ul',{key:'wrapper'},
        React.createElement('li',{key:'A'},'A'),
        React.createElement('li',{key:'B'},'B'),
        React.createElement('li',{key:'C'},'C'),
        React.createElement('li',{key:'D'},'D'),
      );
    }
    return React.createElement('ul',{key:'wrapper'},
      React.createElement('li',{key:'A'},'A'),
      React.createElement('li',{key:'C'},'C1'),
      React.createElement('li',{key:'B'},'B1'),
      React.createElement('li',{key:'E'},'E1'),
      React.createElement('li',{key:'F'},'F1')
      );
  }
}
let element = React.createElement(Counter);
React.render(element,document.getElementById('root'));
```

10.2 react/unit.js

src/react/unit.js

```
import {Element} from './element';
import $ from 'jquery';
import types from './types';
let diffQueue = [];
let updateDepth=0;
class Unit {
    constructor(element){
        this._currentElement = element;
    }
    getMarkUp(){
        throw new Error('不能调用此方法');
    }
}
class TextUnit extends Unit{
    getMarkUp(reactid){
         this._reactid = reactid;//保存记录reactid
        //返回文本节点对应的HTML字符串
        return `${this._currentElement}`;
    }
    update(nextElement){
        if(this._currentElement != nextElement){
            this._currentElement = nextElement;
            $(`[data-reactid="${this._reactid}"]`).html(this._currentElement);
        }
    }

}
class NativeUnit extends Unit {
     getMarkUp(reactid){
         this._reactid = reactid;//保存记录reactid
        //返回文本节点对应的HTML字符串
        let {type,props} = this._currentElement;
        let tagOpen = ``;
        let content = '';
        let renderedChildUnits=[];
        for(let propName in props){
            if(/^on[A-Z]/.test(propName)){
                let eventName = propName.slice(2).toLowerCase();
                $(document).delegate(`[data-reactid="${reactid}"]`,`${eventName}.${reactid}`,props[propName]);
            }else if(propName
                let styleObj = props[propName];
                let styles = Object.keys(styleObj).map(attr=>{
                    let attrName = attr.replace(/([A-Z])/g,function(matched,group){
                        return `-${group.toLowerCase()}`;
                    })
                    return `${attrName}:${styleObj[attr]}`;
                }).join(';');
                tagOpen += (` style="${styles}" `);
            }else if (propName
                let children = props.children||[];
                children.map((child,index)=>{
                    let childUnit = createUnit(child);
                    childUnit._mountIndex = index;
                    renderedChildUnits.push(childUnit);
                    let childMarkUp = childUnit.getMarkUp(`${reactid}.${index}`);
                    content += childMarkUp;
                });
            }else{
                tagOpen += ` ${propName}=${props[propName]} `;
            }
        }
        this._renderedChildUnits = renderedChildUnits;
        return tagOpen + '>' + content + tagClose;
    }
    update(nextElement){
        let oldProps = this._currentElement.props;
        let newProps = nextElement.props;
        this.updateDOMproperties(oldProps,newProps);
        this.updateDOMChildren(nextElement.props.children);
    }
    //对比子元素
    updateDOMChildren(newChildrenElements){
        updateDepth++;
        this.diff(diffQueue,newChildrenElements);
        updateDepth--;
        if(updateDepth
            console.log('diffQueue',diffQueue);
+            this.patch(diffQueue);
            diffQueue=[];
        }
    }
+    patch(diffQueue){
+        let deleteChildren = [];
+        let deleteMap={};
+        for(let i=0;i
+            let difference = diffQueue[i];
+            if(difference.type===types.MOVE || difference.type===types.REMOVE){
+                let fromIndex = difference.fromIndex;
+                let oldChild = $(difference.parentNode.children().get(fromIndex));
+                deleteMap[fromIndex]=oldChild;
+                deleteChildren.push(oldChild);
+            }
+        }
+        $.each(deleteChildren,(idx,child)=>{
+            $(child).remove();
+        });

+        for(let k=0;k
+            let difference = diffQueue[k];
+            switch(difference.type){
+              case types.INSERT:
+                this.insertChildAt(difference.parentNode,$(difference.markUp),difference.toIndex);
+                break;
+              case types.MOVE:
+                this.insertChildAt(difference.parentNode,deleteMap[difference.fromIndex],difference.toIndex);
+                break;
+              default:
+               break;
+            }
+        }
+    }
+    insertChildAt(parentNode,childNode,index){
+        let oldChild = parentNode.children().get(index);
+        oldChild?childNode.insertBefore(oldChild):childNode.appendTo(parentNode);
+    }
    diff(diffQueue,newChildrenElements){
        let oldChildUnitsMap = this.getChildrenMap(this._renderedChildUnits);
        let {newChildrenMap,newChildren} = this.getNewChildren(oldChildUnitsMap,newChildrenElements);
        // lastIndex里存放着被复用的子元素的最大索引
        let lastIndex = 0;
        for(let i=0;i{
            let newKey = newElement.key||index.toString();
            let oldUnit = oldChildUnitsMap[newKey];//获得老的unit
            let oldElement = oldUnit&&oldUnit._currentElement;//获得老的element
            if(shouldDeepCompare(oldElement,newElement)){//如果可以更进一步深比较
                oldUnit.update(newElement);
                newChildren.push(oldUnit);
                newChildrenMap[newKey]=oldUnit;
            }else{
                let newChildUnit = createUnit(newElement);//如果不需要深比较则直接创建新的unit
                newChildren.push(newChildUnit);
                 newChildrenMap[newKey]=newChildUnit;
            }
        });
        return {newChildrenMap,newChildren};
    }
    getChildrenMap(childUnits=[]){
        let map = {};
        for(let i=0;i{
                  $(`[data-reactid="${this._reactid}"]`).css(attr,value);
                })
            }else{
                $(`[data-reactid="${this._reactid}"]`).prop(propName,newProps[propName]);
            }
        }
    }
}

class CompositeUnit extends Unit{
    //接收到新的更新，自定义组件传第二个参数，原生组件和text传处一个参数
    update(nextElement,partialState){
        //如果传过来了新的元素，则使用新的元素
        this._currentElement = nextElement||this._currentElement;
        //获取新的状态对象和属性对象
        let nextState = this._componentInstance.state= Object.assign(this._componentInstance.state,partialState);
        let nextProps = this._currentElement.props;
        //如果shouldComponentUpdate返回了false则不需要继续更新
        if(this._componentInstance.shouldComponentUpdate&&this._componentInstance.shouldComponentUpdate(nextProps,nextState)
        //获得上次渲染出来的unit实例
        let prevRenderedUnitInstance = this._renderedUnitInstance;
        //从unit实例中获取
        let prevRenderedElement = prevRenderedUnitInstance._currentElement;
        //获取新的虚拟DOM
        let nextRenderElement = this._componentInstance.render();
        //进行domdiff对比
        if(shouldDeepCompare(prevRenderedElement,nextRenderElement)){
            //如果需要更新，则继续调用子节点的upate方法进行更新,传入新的element更新子节点
            prevRenderedUnitInstance.update(nextRenderElement);
            this._componentInstance.componentDidUpdate&&this._componentInstance.componentDidUpdate();
        }else{
            //如果发现不需要对比，干脆重新渲染
            this._renderedUnitInstance =  createUnit(nextRenderElement);
            let nextMarkUp = this._renderedUnitInstance.getMarkUp(this._reactid);
            //替换整个节点
            $(`[data-reactid="${this._reactid}"]`).replaceWith(nextMarkUp);
        }

    }
    getMarkUp(reactid){
        this._reactid = reactid;
        //type是一个自定义组件的类的定义
        let {type:Component,props} = this._currentElement;
        //创建Component类的实例
        let componentInstance = this._componentInstance = new Component(props);
        //组件实例关联上自己的unit实例
        componentInstance._currentUnit  = this;
        //组件将要渲染
        componentInstance.componentWillMount&&componentInstance.componentWillMount();
        //执行render方法获得虚拟DOM元素实例
        let renderedElement = componentInstance.render();
        //根据虚拟DOM元素得到unit,可能是TextUnit NativeUnit CompositeUnit
        let renderedUnitInstance = this._renderedUnitInstance= createUnit(renderedElement);
        //获得此unit的HTML标记字符串
        let renderedMarkUp = renderedUnitInstance.getMarkUp(reactid);
        //注册挂载完成的监听，越底层的组件越先监听，越先执行
        $(document).on('mounted',()=>componentInstance.componentDidMount&&componentInstance.componentDidMount());
        return renderedMarkUp;
    }
}

function shouldDeepCompare(prevElement,nextElement){
   if(prevElement!==null && nextElement!=null){
       let prevType = typeof prevElement;
       let nextType = typeof nextElement;
       //如果新老节点都是文本可以进行比较
       if((prevType
           return true;
       }
       if(prevElement instanceof Element && nextElement instanceof Element){
           return prevElement.type
       }
   }
   return false;
}
function createUnit(element){
  if(typeof element =='string' || typeof element =='number'){
      return new TextUnit(element);
  }
  if(element instanceof Element && typeof element.type
      return new NativeUnit(element);
  }
  if(element instanceof Element && typeof element.type
      return new CompositeUnit(element);
  }
}

export {
    createUnit
}
```

11. todos

commit (https://gitee.com/zhufengpeixun/zhufeng_react2/commit/466915a20e4bb8d0d019b3325ad47867a3891fae)

11.1 src/index.js

src/index.js

```
import React from './react';
class Todos extends React.Component{
    constructor(props){
        super(props);
        this.state = {list:[],text:''};
    }
    add(){
      if(this.state.text && this.state.text.length>0){
        this.setState({list:[...this.state.list,this.state.text],text:''});
      }
    }
    onChange(event) {
        this.setState({text: event.target.value});
    }
    onDel(index) {
        this.state.list.splice(index,1);
        this.setState({list: this.state.list});
    }
    render(){
        var createItem = (itemText,index)=> {
            return React.createElement("div", {}, itemText,React.createElement('button',{onClick: this.onDel.bind(this,index)},'X'));
        };

        var lists = this.state.list.map(createItem);
        var input = React.createElement("input", {onKeyup: this.onChange.bind(this),value: this.state.text});
        var button = React.createElement("button", {onClick: this.add.bind(this)}, 'Add')
        return React.createElement('div',{},input,button,...lists);
    }
}
let todos = React.createElement(Todos);
React.render(todos,document.getElementById('root'));
```

11.2 react/unit.js

src/react/unit.js

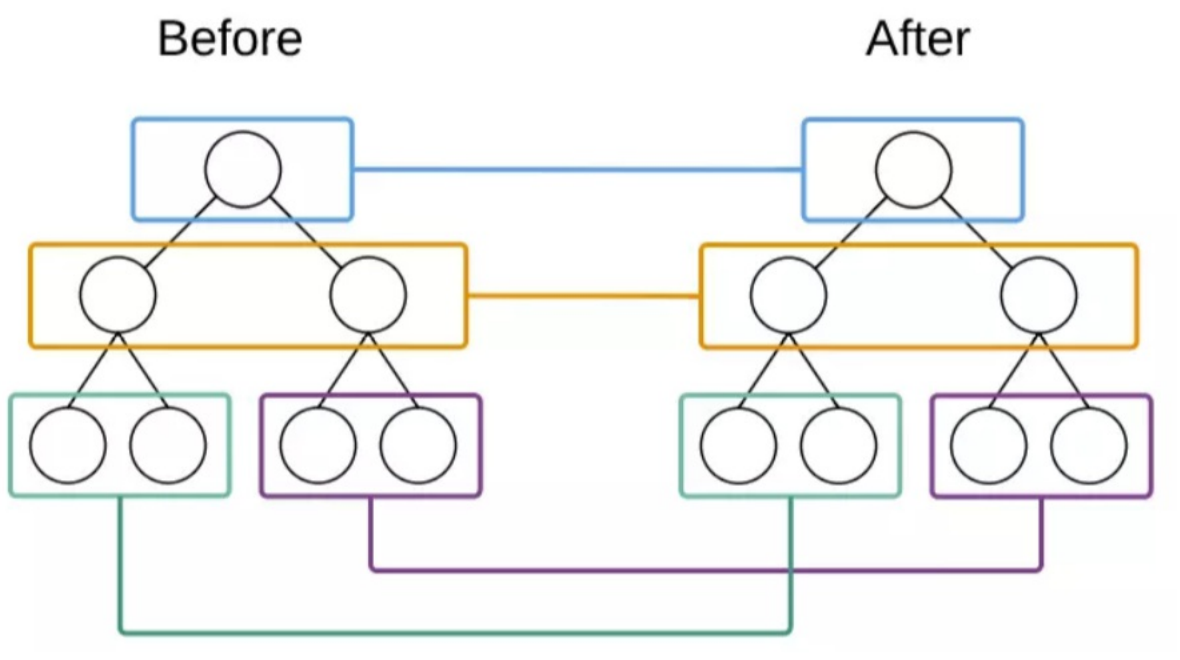
```
import {Element} from './element';
import $ from 'jquery';
import types from './types';
let diffQueue = [];
let updateDepth=0;
class Unit {
    constructor(element){
        this._currentElement = element;
    }
    getMarkUp(){
        throw new Error('不能调用此方法');
    }
}
class TextUnit extends Unit{
    getMarkUp(reactid){
         this._reactid = reactid;//保存记录reactid
        //返回文本节点对应的HTML字符串
        return `${this._currentElement}`;
    }
    update(nextElement){
        if(this._currentElement != nextElement){
            this._currentElement = nextElement;
            $(`[data-reactid="${this._reactid}"]`).html(this._currentElement);
        }
    }

}
class NativeUnit extends Unit {
     getMarkUp(reactid){
         this._reactid = reactid;//保存记录reactid
        //返回文本节点对应的HTML字符串
        let {type,props} = this._currentElement;
        let tagOpen = ``;
        let content = '';
        let renderedChildUnits=[];
        for(let propName in props){
            if(/^on[A-Z]/.test(propName)){
                let eventName = propName.slice(2).toLowerCase();
                $(document).delegate(`[data-reactid="${reactid}"]`,`${eventName}.${reactid}`,props[propName]);
            }else if(propName
                let styleObj = props[propName];
                let styles = Object.keys(styleObj).map(attr=>{
                    let attrName = attr.replace(/([A-Z])/g,function(matched,group){
                        return `-${group.toLowerCase()}`;
                    })
                    return `${attrName}:${styleObj[attr]}`;
                }).join(';');
                tagOpen += (` style="${styles}" `);
            }else if (propName
                let children = props.children||[];
                children.map((child,index)=>{
                    let childUnit = createUnit(child);
                    childUnit._mountIndex = index;
                    renderedChildUnits.push(childUnit);
                    let childMarkUp = childUnit.getMarkUp(`${reactid}.${index}`);
                    content += childMarkUp;
                });
            }else{
                tagOpen += ` ${propName}=${props[propName]} `;
            }
        }
        this._renderedChildUnits = renderedChildUnits;
        return tagOpen + '>' + content + tagClose;
    }
    update(nextElement){
        let oldProps = this._currentElement.props;
        let newProps = nextElement.props;
        this.updateDOMproperties(oldProps,newProps);
        this.updateDOMChildren(nextElement.props.children);
    }
    //对比子元素
    updateDOMChildren(newChildrenElements){
        updateDepth++;
        this.diff(diffQueue,newChildrenElements);
        updateDepth--;
        if(updateDepth
            console.log('diffQueue',diffQueue);
            this.patch(diffQueue);
            diffQueue=[];
        }
    }
    patch(diffQueue){
        let deleteChildren = [];
        let deleteMap={};
        for(let i=0;i+                let parentId = difference.parentId;
+                let oldChild = $(difference.parentNode.children().get(fromIndex));
+                deleteMap[parentId]={};
+                deleteMap[parentId][fromIndex]=oldChild;
                deleteChildren.push(oldChild);
            }
        }
        $.each(deleteChildren,(idx,child)=>{
            $(child).remove();
        });

        for(let k=0;k+                this.insertChildAt(difference.parentNode,deleteMap[difference.parentId][difference.fromIndex],difference.toIndex);
                break;
              default:
               break;
            }
        }
    }
    insertChildAt(parentNode,childNode,index){
        let oldChild = parentNode.children().get(index);
        oldChild?childNode.insertBefore(oldChild):childNode.appendTo(parentNode);
    }
    diff(diffQueue,newChildrenElements){
        let oldChildUnitsMap = this.getChildrenMap(this._renderedChildUnits);
        let {newChildrenMap,newChildren} = this.getNewChildren(oldChildUnitsMap,newChildrenElements);
        // lastIndex里存放着被复用的子元素的最大索引
        let lastIndex = 0;
        for(let i=0;i+                this._renderedChildUnits = this._renderedChildUnits.filter(item=>item != oldChild);
+                $(document).undelegate(`.${oldChild._reactid}`);
            }
        }
    }
    getNewChildren(oldChildUnitsMap,newChildrenElements){
        let newChildren = [];
        let newChildrenMap={};
        newChildrenElements.forEach((newElement,index)=>{
            let newKey = newElement.key||index.toString();
            let oldUnit = oldChildUnitsMap[newKey];//获得老的unit
            let oldElement = oldUnit&&oldUnit._currentElement;//获得老的element
            if(shouldDeepCompare(oldElement,newElement)){//如果可以更进一步深比较
                oldUnit.update(newElement);
                newChildren.push(oldUnit);
                newChildrenMap[newKey]=oldUnit;
            }else{
                let newChildUnit = createUnit(newElement);//如果不需要深比较则直接创建新的unit
                newChildren.push(newChildUnit);
                newChildrenMap[newKey]=newChildUnit;
                this._renderedChildUnits[index]=newChildUnit;
            }
        });
        return {newChildrenMap,newChildren};
    }
    getChildrenMap(childUnits=[]){
        let map = {};
        for(let i=0;i{
                  $(`[data-reactid="${this._reactid}"]`).css(attr,value);
                })
            }else{
                $(`[data-reactid="${this._reactid}"]`).prop(propName,newProps[propName]);
            }
        }
    }
}

class CompositeUnit extends Unit{
    //接收到新的更新，自定义组件传第二个参数，原生组件和text传处一个参数
    update(nextElement,partialState){
        //如果传过来了新的元素，则使用新的元素
        this._currentElement = nextElement||this._currentElement;
        //获取新的状态对象和属性对象
        let nextState = this._componentInstance.state= Object.assign(this._componentInstance.state,partialState);
        let nextProps = this._currentElement.props;
        //如果shouldComponentUpdate返回了false则不需要继续更新
        if(this._componentInstance.shouldComponentUpdate&&this._componentInstance.shouldComponentUpdate(nextProps,nextState)
        //获得上次渲染出来的unit实例
        let prevRenderedUnitInstance = this._renderedUnitInstance;
        //从unit实例中获取
        let prevRenderedElement = prevRenderedUnitInstance._currentElement;
        //获取新的虚拟DOM
        let nextRenderElement = this._componentInstance.render();
        //进行domdiff对比
        if(shouldDeepCompare(prevRenderedElement,nextRenderElement)){
            //如果需要更新，则继续调用子节点的upate方法进行更新，传入新的element更新子节点
            prevRenderedUnitInstance.update(nextRenderElement);
            this._componentInstance.componentDidUpdate&&this._componentInstance.componentDidUpdate();
        }else{
            //如果发现不需要对比，干脆重新渲染
            this._renderedUnitInstance =  createUnit(nextRenderElement);
            let nextMarkUp = this._renderedUnitInstance.getMarkUp(this._reactid);
            //替换整个节点
            $(`[data-reactid="${this._reactid}"]`).replaceWith(nextMarkUp);
        }

    }
    getMarkUp(reactid){
        this._reactid = reactid;
        //type是一个自定义组件的类的定义
        let {type:Component,props} = this._currentElement;
        //创建Component类的实例
        let componentInstance = this._componentInstance = new Component(props);
        //组件实例关联上自己的unit实例
        componentInstance._currentUnit  = this;
        //组件将要渲染
        componentInstance.componentWillMount&&componentInstance.componentWillMount();
        //执行render方法获得虚拟DOM元素实例
        let renderedElement = componentInstance.render();
        //根据虚拟DOM元素得到unit,可能是TextUnit NativeUnit CompositeUnit
        let renderedUnitInstance = this._renderedUnitInstance= createUnit(renderedElement);
        //获得此unit的HTML标记字符串
        let renderedMarkUp = renderedUnitInstance.getMarkUp(reactid);
        //注册挂载完成的监听，越底层的组件越先监听，越先执行
        $(document).on('mounted',()=>componentInstance.componentDidMount&&componentInstance.componentDidMount());
        return renderedMarkUp;
    }
}

function shouldDeepCompare(prevElement,nextElement){
   if(prevElement!==null && nextElement!=null){
       let prevType = typeof prevElement;
       let nextType = typeof nextElement;
       //如果新老节点都是文本可以进行比较
       if((prevType
           return true;
       }
       if(prevElement instanceof Element && nextElement instanceof Element){
           return prevElement.type
       }
   }
   return false;
}
function createUnit(element){
  if(typeof element =='string' || typeof element =='number'){
      return new TextUnit(element);
  }
  if(element instanceof Element && typeof element.type
      return new NativeUnit(element);
  }
  if(element instanceof Element && typeof element.type
      return new CompositeUnit(element);
  }
}

export {
    createUnit
}
```

10. diff 策略

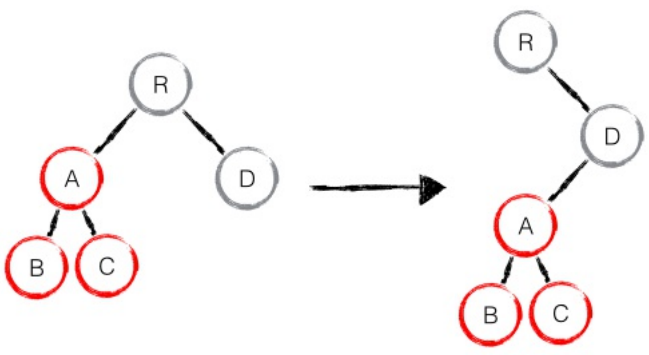
- Web UI 中 DOM 节点跨层级的移动操作特别少，可以忽略不计。
- 拥有相同类的两个组件将会生成相似的树形结构，拥有不同类的两个组件将会生成不同的树形结构。
- 对于同一层级的一组子节点，它们可以通过唯一 key进行区分。

10.1 tree diff

- React 对树的算法进行了简洁明了的优化，即对树进行分层比较，两棵树只会对同一层次的节点进行比较

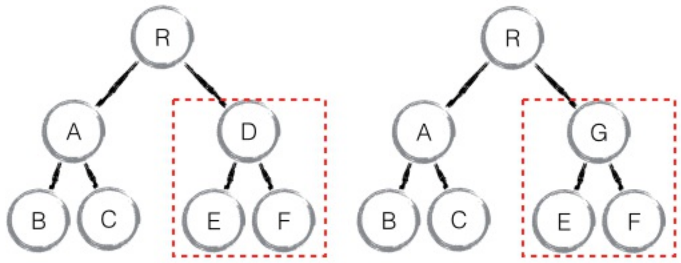


- 当出现节点跨层级移动时，并不会出现设想中的移动操作，而是以 A 为根节点的树被整个重新创建



10.2 component diff

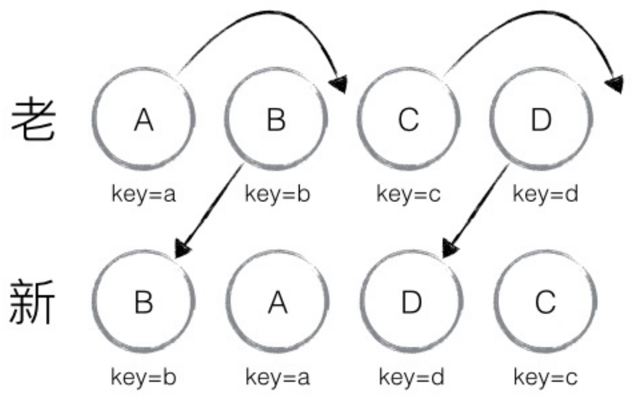
- 如果是同一类型的组件，按照原策略继续比较 `virtual DOM tree`
- 如果不是，则将该组件判断为 `dirty component`,从而替换整个组件下的所有子节点

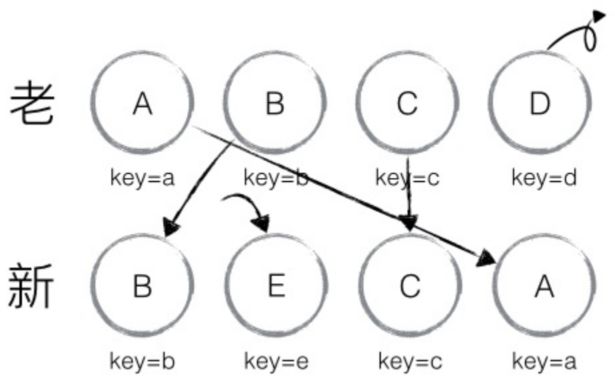


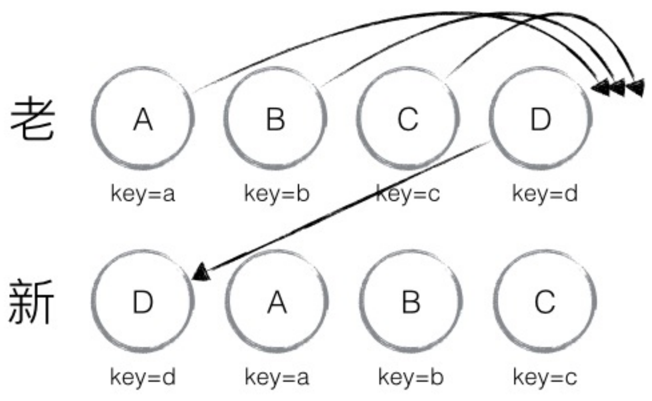
10.3 element diff

- 当节点处于同一层级时，React diff 提供了三种节点操作,分别为：INSERT(插入)、MOVE(移动)和 REMOVE(删除)
- INSERT 新的 component 类型不在老集合里， 即是全新的节点，需要对新节点执行插入操作
- MOVE 在老集合有新 component 类型，就需要做移动操作，可以复用以前的 DOM 节点
- REMOVE 老 component 不在新集合里的，也需要执行删除操作

10.4 key







11.delegate

- delegate() 方法为指定的元素(属于被选元素的子元素)添加一个或多个事件处理程序，并规定当这些事件发生时运行的函数，使用 delegate() 方法的事件处理程序适用于当前或未来的元素(比如由脚本创建的新元素)

参数名称 参数含义 childSelector 必需,规定要附加事件处理程序的一个或多个子元素 event 必需,规定附加到元素的一个或多个事件,由空格分隔多个事件值。必须是有效的事件 data 可选,规定传递到函数的额外数据 function 必需,规定当事件发生时运行的函数

- delegate (https://api.jquery.com/delegate/)
- undelegate (http://api.jquery.com/undelegate/)
- 参数 events还支持为事件类型附加额外的命名空间
- 当为同一元素绑定多个相同类型的事件处理函数时。使用命名空间，可以在触发事件、移除事件时限定触发或移除的范围。

```
var $document = $(document);

$document.delegate("#btn1", "click.foo.bar", function(event){
    alert("click-1");
});

$document.delegate("#btn1", "click.test", function(event){
    alert("click-2");
});

$document.delegate("#btn1", "click.test.foo", function(event){
    alert("click-3");
});

$btn1.trigger("click");

$btn1.trigger("click.foo");

$btn1.trigger("click.bar");

$btn1.undelegate( "click.foo" );
```