link: null
title: 珠峰架构师成长计划
description: src\react.js
keywords: null
author: null
date: null
publisher: 珠峰架构师成长计划
stats: paragraph=157 sentences=916, words=5615

1.实现虚拟DOM

1.1 src\index.js

```
import React from './react';

let element = (
  <div id="A1">
    <div id="B1">
      <div id="C1">div>
      <div id="C2">div>
    div>
    <div id="B2">div>
  div>
)
console.log(element);
```

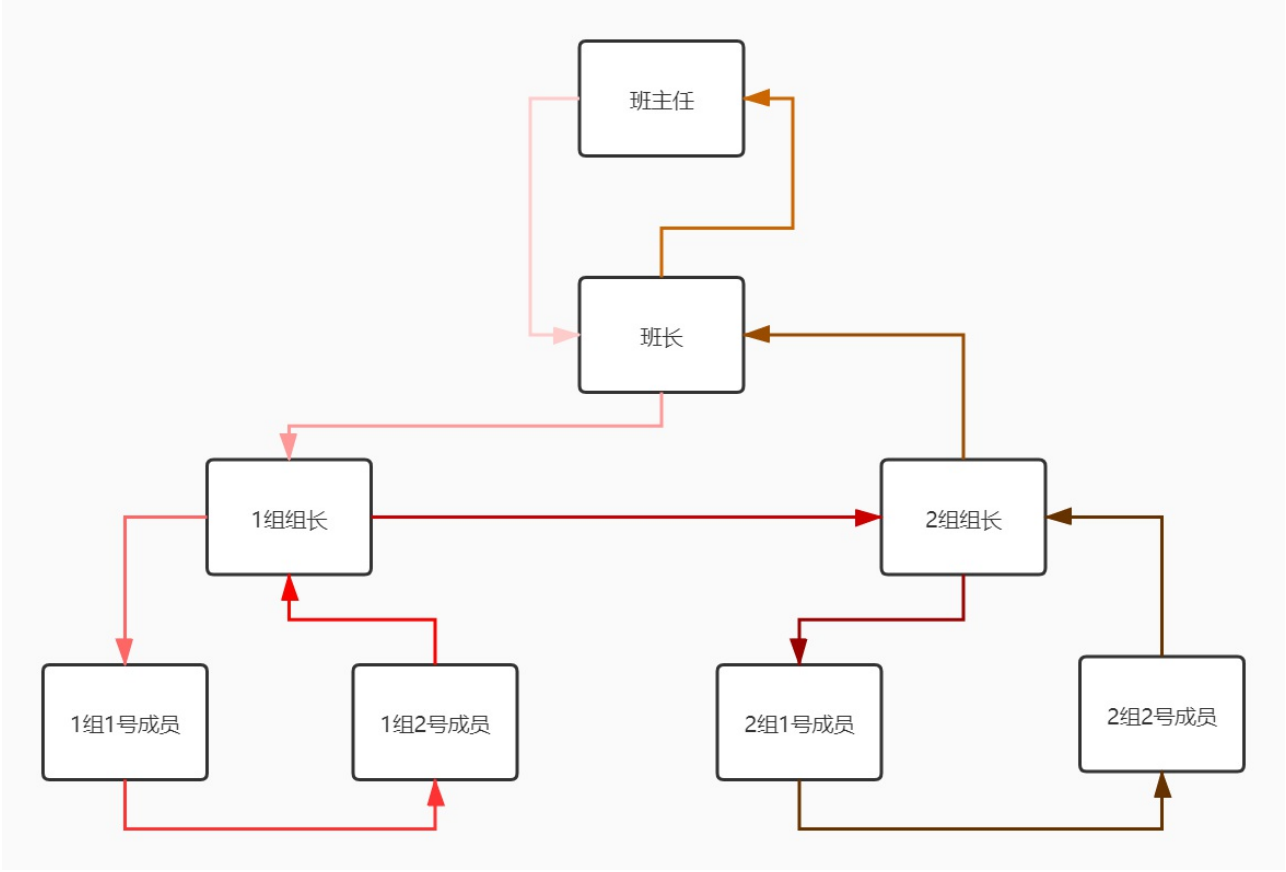
1.2 src\react.js

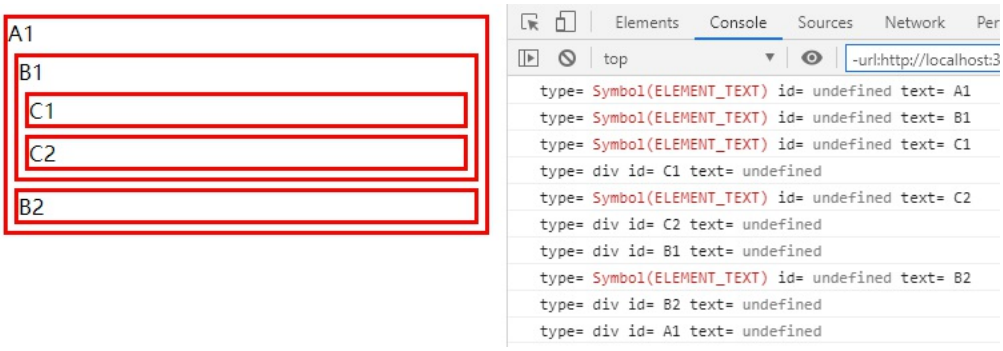
src\react.js

```
import { ELEMENT_TEXT } from './constants';
function createElement(type, config, ...children) {
    delete config.__self;
    delete config.__source;
    return {
        type,
        props: {
            ...config,
            children: children.map(
                child => typeof child === "object" ?
                    child :
                    { type: ELEMENT_TEXT, props: { text: child, children: [] } })
        }
    }
}

let React = {
    createElement
}
export default React;
```

2.实现初次渲染





2.1 index.js

```
import React from './react';
import ReactDOM from './react-dom';
+let style = { border: '3px solid red', margin: '5px' };
+let element = (
+
+    A1
+
+      B1
+        C1
+      C2
+
+    B2
+
+)
ReactDOM.render(
  element,
  document.getElementById('root')
);
```

2.2 constants.js

src\constants.js

```
+export const ELEMENT_TEXT = Symbol.for('ELEMENT_TEXT');
+export const TAG_ROOT = Symbol.for('TAG_ROOT');
+export const TAG_HOST = Symbol.for('TAG_HOST');
+export const TAG_TEXT = Symbol.for('TAG_TEXT');
+export const PLACEMENT = Symbol.for('PLACEMENT');
```

2.3 utils.js

src\utils.js

```
function setProp(dom, key, value) {
    if (/^on/.test(key)) {
        dom[key.toLowerCase()] = value;
    } else if (key === 'style') {
        if (value) {
            for (let styleName in value) {
                if (value.hasOwnProperty(styleName)) {
                    dom.style[styleName] = value[styleName];
                }
            }
        }
    } else {
        dom.setAttribute(key, value);
    }
    return dom;
}
export function setProps(elem, oldProps, newProps) {
    for (let key in oldProps) {
        if (key !== 'children') {
            if (newProps.hasOwnProperty(key)) {
                setProp(elem, key, newProps[key]);
            } else {
                elem.removeAttribute(key);
            }
        }
    }
    for (let key in newProps) {
        if (key !== 'children') {
            setProp(elem, key, newProps[key])
        }
    }
}
```

2.4 react-dom.js

src\react-dom.js

```
import { TAG_ROOT } from './constants';
import { scheduleRoot } from './scheduler';

function render(element, container) {
    let rootFiber = {
        tag: TAG_ROOT,
        stateNode: container,
        props: { children: [element] },
    }
    scheduleRoot(rootFiber);
}

export default {
    render
}
```

2.4 scheduler.js

src\scheduler.js

```
import { setProps } from './utils';
import {
    ELEMENT_TEXT, TAG_ROOT, TAG_HOST, TAG_TEXT, PLACEMENT
} from './constants';

let workInProgressRoot = null;
let nextUnitOfWork = null

export function scheduleRoot(rootFiber) {

    workInProgressRoot = rootFiber;
    nextUnitOfWork = workInProgressRoot;
}

function commitRoot() {
    let currentFiber = workInProgressRoot.firstEffect;
    while (currentFiber) {
        commitWork(currentFiber);
        currentFiber = currentFiber.nextEffect;
    }
    workInProgressRoot = null;
}
function commitWork(currentFiber) {
    if (!currentFiber) {
        return;
    }
    let returnFiber = currentFiber.return;
    const domReturn = returnFiber.stateNode;
    if (currentFiber.effectTag === PLACEMENT && currentFiber.stateNode != null) {
        let nextFiber = currentFiber;
        domReturn.appendChild(nextFiber.stateNode);
    }
    currentFiber.effectTag = null;
}

function performUnitOfWork(currentFiber) {
    beginWork(currentFiber);

    if (currentFiber.child) {
        return currentFiber.child;
    }

    while (currentFiber) {
        completeUnitOfWork(currentFiber);
        if (currentFiber.sibling) {
            return currentFiber.sibling;
        }
        currentFiber = currentFiber.return;
    }
}

function beginWork(currentFiber) {
    if (currentFiber.tag === TAG_ROOT) {
        updateHostRoot(currentFiber);
    } else if (currentFiber.tag === TAG_TEXT) {
        updateHostText(currentFiber);
    } else if (currentFiber.tag === TAG_HOST) {
        updateHostComponent(currentFiber);
    }
}

function updateHostRoot(currentFiber) {
    const newChildren = currentFiber.props.children;
    reconcileChildren(currentFiber, newChildren);
}
function updateHostText(currentFiber) {
    if (!currentFiber.stateNode) {
        currentFiber.stateNode = createDOM(currentFiber);
    }
}
function updateHostComponent(currentFiber) {
    if (!currentFiber.stateNode) {
        currentFiber.stateNode = createDOM(currentFiber);
    }
    const newChildren = currentFiber.props.children;
    reconcileChildren(currentFiber, newChildren);
}
function createDOM(currentFiber) {
    if (currentFiber.type === ELEMENT_TEXT) {
        return document.createTextNode(currentFiber.props.text);
    }
    const stateNode = document.createElement(currentFiber.type);
    updateDOM(stateNode, {}, currentFiber.props);
    return stateNode;
```

```
}

function reconcileChildren(currentFiber, newChildren) {
    let newChildIndex = 0;
    let prevSibling;
    while (newChildIndex < newChildren.length) {
        const newChild = newChildren[newChildIndex];
        let tag;
        if (newChild && newChild.type === ELEMENT_TEXT) {
            tag = TAG_TEXT;
        } else if (newChild && typeof newChild.type === 'string') {
            tag = TAG_HOST;
        }
        let newFiber = {
            tag,
            type: newChild.type,
            props: newChild.props,
            stateNode: null,
            return: currentFiber,
            effectTag: PLACEMENT,
            nextEffect: null
        }
        if (newFiber) {
            if (newChildIndex === 0) {
                currentFiber.child = newFiber;
            } else {
                prevSibling.sibling = newFiber;
            }
            prevSibling = newFiber;
        }
        prevSibling = newFiber;
        newChildIndex++;
    }
}

function updateDOM(stateNode, oldProps, newProps) {
    setProps(stateNode, oldProps, newProps);
}
function completeUnitOfWork(currentFiber) {
    const returnFiber = currentFiber.return;
    if (returnFiber) {
        if (!returnFiber.firstEffect) {
            returnFiber.firstEffect = currentFiber.firstEffect;
        }
        if (!!currentFiber.lastEffect) {
            if (!!returnFiber.lastEffect) {
                returnFiber.lastEffect.nextEffect = currentFiber.firstEffect;
            }
            returnFiber.lastEffect = currentFiber.lastEffect;
        }

        const effectTag = currentFiber.effectTag;
        if (effectTag) {
            if (!!returnFiber.lastEffect) {
                returnFiber.lastEffect.nextEffect = currentFiber;
            } else {
                returnFiber.firstEffect = currentFiber;
            }
            returnFiber.lastEffect = currentFiber;
        }
    }
}

function workLoop(deadline) {
    let shouldYield = false;
    while (nextUnitOfWork && !shouldYield) {
        nextUnitOfWork = performUnitOfWork(nextUnitOfWork);
        shouldYield = deadline.timeRemaining() < 1;
    }

    if (!nextUnitOfWork && workInProgressRoot) {
        commitRoot();
    }
    requestIdleCallback(workLoop);
}

requestIdleCallback(workLoop);
```

3.实现元素的更新

3.1 public\index.html

```
<body>
  <div id="root">div>
  <button id="reRender1">reRender1button>
  <button id="reRender2">reRender2button>
body>
```

3.2 src\index.js

src\index.js

```
import React from './react';
import ReactDOM from './react-dom';
let style = { border: '3px solid red', margin: '5px' };
let element = (

    A1

      B1
      C1
      C2

    B2

)
console.log(element);
ReactDOM.render(
  element,
  document.getElementById('root')
);

+let reRender2 = document.getElementById('reRender2');
+reRender2.addEventListener('click', () => {
+  let element2 = (
+
+      A1-new
+
+        B1-new
+        C1-new
+        C2-new
+
+      B2
+      B3
+
+  )
+  ReactDOM.render(
+    element2,
+    document.getElementById('root')
+  );
+});
+
+let reRender3 = document.getElementById('reRender3');
+reRender3.addEventListener('click', () => {
+  let element3 = (
+
+      A1-new2
+
+        B1-new2
+        C1-new2
+        C2-new2
+
+      B2
+
+  )
+  ReactDOM.render(
+    element3,
+    document.getElementById('root')
+  );
+});
```

3.3 src\constants.js

```
export const ELEMENT_TEXT = Symbol.for('ELEMENT_TEXT');

export const TAG_ROOT = Symbol.for('TAG_ROOT');
export const TAG_HOST = Symbol.for('TAG_HOST');
export const TAG_TEXT = Symbol.for('TAG_TEXT');

export const PLACEMENT = Symbol.for('PLACEMENT');
+export const UPDATE = Symbol.for('UPDATE');
+export const DELETION = Symbol.for('DELETION');
```

3.4 scheduler.js

src\scheduler.js

```
import { setProps } from './utils';
import {
    ELEMENT_TEXT, TAG_ROOT, TAG_HOST, TAG_TEXT, PLACEMENT, DELETION, UPDATE
} from './constants';
+let currentRoot = null;//当前的根Fiber
let workInProgressRoot = null;//正在渲染中的根Fiber
let nextUnitOfWork = null//下一个工作单元
+let deletions = [];//要删除的fiber节点

export function scheduleRoot(rootFiber) {
    //{tag:TAG_ROOT,stateNode:container,props: { children: [element] }}
+    if (currentRoot && currentRoot.alternate) {//偶数次更新
+        workInProgressRoot = currentRoot.alternate;
+        workInProgressRoot.firstEffect = workInProgressRoot.lastEffect = workInProgressRoot.nextEffect = null;
+        workInProgressRoot.props = rootFiber.props;
+        workInProgressRoot.alternate = currentRoot;
+    } else if (currentRoot) {//奇数次更新
+        rootFiber.alternate = currentRoot;
+        workInProgressRoot = rootFiber;
+    } else {
+        workInProgressRoot = rootFiber;//第一次渲染
+    }
    nextUnitOfWork = workInProgressRoot;
}

function commitRoot() {
+    deletions.forEach(commitWork);
    let currentFiber = workInProgressRoot.firstEffect;
    while (currentFiber) {
```

```
        commitWork(currentFiber);
        currentFiber = currentFiber.nextEffect;
    }
+    deletions.length = 0;//先把要删除的节点清空掉
+    currentRoot = workInProgressRoot;
    workInProgressRoot = null;
}
function commitWork(currentFiber) {
    if (!currentFiber) {
        return;
    }
    let returnFiber = currentFiber.return;//先获取父Fiber
    const domReturn = returnFiber.stateNode;//获取父的DOM节点
    if (currentFiber.effectTag
        let nextFiber = currentFiber;
        domReturn.appendChild(nextFiber.stateNode);
+    } else if (currentFiber.effectTag === DELETION) {//如果是删除则删除并返回
+       domReturn.removeChild(currentFiber.stateNode);
+    } else if (currentFiber.effectTag === UPDATE && currentFiber.stateNode != null) {//如果是更新
+       if (currentFiber.type === ELEMENT_TEXT) {
+           if (currentFiber.alternate.props.text != currentFiber.props.text) {
+               currentFiber.stateNode.textContent = currentFiber.props.text;
+           }
+       } else {
+           updateDOM(currentFiber.stateNode, currentFiber.alternate.props, currentFiber.props);
+       }
+    }
    currentFiber.effectTag = null;
}

function performUnitOfWork(currentFiber) {
    beginWork(currentFiber);//开始渲染前的Fiber,就是把子元素变成子fiber

    if (currentFiber.child) {//如果子节点就返回第一个子节点
        return currentFiber.child;
    }

    while (currentFiber) {//如果没有子节点说明当前节点已经完成了渲染工作
        completeUnitOfWork(currentFiber);//可以结束此fiber的渲染了
        if (currentFiber.sibling) {//如果它有弟弟就返回弟弟
            return currentFiber.sibling;
        }
        currentFiber = currentFiber.return;//如果没有弟弟让爸爸完成，然后找叔叔
    }
}

function beginWork(currentFiber) {
    if (currentFiber.tag
        updateHostRoot(currentFiber);
    } else if (currentFiber.tag
        updateHostText(currentFiber);
    } else if (currentFiber.tag
        updateHostComponent(currentFiber);
    }
}
function updateHostRoot(currentFiber) {//如果是根节点
    const newChildren = currentFiber.props.children;//直接渲染子节点
    reconcileChildren(currentFiber, newChildren);
}
function updateHostText(currentFiber) {
    if (!currentFiber.stateNode) {
        currentFiber.stateNode = createDOM(currentFiber);//先创建真实的DOM节点
    }
}
function updateHostComponent(currentFiber) {//如果是原生DOM节点
    if (!currentFiber.stateNode) {
        currentFiber.stateNode = createDOM(currentFiber);//先创建真实的DOM节点
    }
    const newChildren = currentFiber.props.children;
    reconcileChildren(currentFiber, newChildren);
}
function createDOM(currentFiber) {
    if (currentFiber.type
        return document.createTextNode(currentFiber.props.text);
    }
    const stateNode = document.createElement(currentFiber.type);
    updateDOM(stateNode, {}, currentFiber.props);
    return stateNode;
}

function reconcileChildren(currentFiber, newChildren) {
    let newChildIndex = 0;//新虚拟DOM数组中的索引
+    let oldFiber = currentFiber.alternate && currentFiber.alternate.child;//父Fiber中的第一个子Fiber
+    let prevSibling;
+    while (newChildIndex < newChildren.length || oldFiber) {
+       const newChild = newChildren[newChildIndex];
+       let newFiber;
+       const sameType = oldFiber && newChild && newChild.type === oldFiber.type;//新旧都有，并且元素类型一样
+       let tag;
+       if (newChild && newChild.type === ELEMENT_TEXT) {
+           tag = TAG_TEXT;//文本
+       } else if (newChild && typeof newChild.type === 'string') {
+           tag = TAG_HOST;//原生DOM组件
+       }
+       if (sameType) {
+           if (oldFiber.alternate) {
+               newFiber = oldFiber.alternate;
+               newFiber.props = newChild.props;
+               newFiber.alternate = oldFiber;
+               newFiber.effectTag = UPDATE;
+               newFiber.nextEffect = null;
+           } else {
+               newFiber = {
+                   tag:oldFiber.tag,//标记Fiber类型，例如是函数组件或者原生组件
```

```
+                    type: oldFiber.type,//具体的元素类型
+                    props: newChild.props,//新的属性对象
+                    stateNode: oldFiber.stateNode,//原生组件的话就存放DOM节点，类组件的话是类组件实例，函数组件的话为空，因为没有实例
+                    return: currentFiber,//父Fiber
+                    alternate: oldFiber,//上一个Fiber 指向旧树中的节点
+                    effectTag: UPDATE,//副作用标识
+                    nextEffect: null //React 同样使用链表来将所有有副作用的Fiber连接起来
+                }
+            }
+        } else {
+            if (newChild) {//类型不一样，创建新的Fiber,旧的不复用了
+                newFiber = {
+                    tag,//原生DOM组件
+                    type: newChild.type,//具体的元素类型
+                    props: newChild.props,//新的属性对象
+                    stateNode: null,//stateNode肯定是空的
+                    return: currentFiber,//父Fiber
+                    effectTag: PLACEMENT//副作用标识
+                }
+            }
+            if (oldFiber) {
+                oldFiber.effectTag = DELETION;
+                deletions.push(oldFiber);
+            }
+        }
+        if (oldFiber) {  //比较完一个元素了，老Fiber向后移动1位
+            oldFiber = oldFiber.sibling;
+        }
       if (newFiber) {
           if (newChildIndex
              currentFiber.child = newFiber;//第一个子节点挂到父节点的child属性上
          } else {
              prevSibling.sibling = newFiber;
          }
          prevSibling = newFiber;//然后newFiber变成了上一个哥哥了
       }
       prevSibling = newFiber;//然后newFiber变成了上一个哥哥了
       newChildIndex++;
    }
}

function updateDOM(stateNode, oldProps, newProps) {
    setProps(stateNode, oldProps, newProps);
}
function completeUnitOfWork(currentFiber) {
    const returnFiber = currentFiber.return;
    if (returnFiber) {
        if (!returnFiber.firstEffect) {
            returnFiber.firstEffect = currentFiber.firstEffect;
        }
        if (!!currentFiber.lastEffect) {
            if (!!returnFiber.lastEffect) {
                returnFiber.lastEffect.nextEffect = currentFiber.firstEffect;
            }
            returnFiber.lastEffect = currentFiber.lastEffect;
        }

        const effectTag = currentFiber.effectTag;
        if (effectTag) {
            if (!!returnFiber.lastEffect) {
                returnFiber.lastEffect.nextEffect = currentFiber;
            } else {
                returnFiber.firstEffect = currentFiber;
            }
            returnFiber.lastEffect = currentFiber;
        }
    }
}

function workLoop(deadline) {
    let shouldYield = false;
    while (nextUnitOfWork && !shouldYield) {
        nextUnitOfWork = performUnitOfWork(nextUnitOfWork);//执行一个任务并返回下一个任务
        shouldYield = deadline.timeRemaining() < 1;//如果剩余时间小于1毫秒就说明没有时间了，需要把控制权让给浏览器
    }
    //如果没有下一个执行单元了，并且当前渲染树存在，则进行提交阶段
    if (!nextUnitOfWork && workInProgressRoot) {
        commitRoot();
    }
    requestIdleCallback(workLoop);
}
//开始在空闲时间执行workLoop
requestIdleCallback(workLoop);
```

4.实现类组件

4.1 src\index.js

```
import React from './react';
import ReactDOM from './react-dom';
+class ClassCounter extends React.Component {
+  constructor(props) {
+    super(props);
+    this.state = { number: 0 };
+  }
+  onClick = () => {
+    this.setState(state => ({ number: state.number + 1 }));
+  }
+  render() {
+    return (
+
+        {this.state.number}
+        加1
+
+    )
+  }
+}
ReactDOM.render(
+  ,
  document.getElementById('root')
);
```

4.2 src\react.js

src\react.js

```
import { ELEMENT_TEXT } from './constants';
+import { Update, UpdateQueue } from './updateQueue';
+import { scheduleRoot } from './scheduler';
function createElement(type, config, ...children) {
    delete config.__self;
    delete config.__source;
    return {
        type,
        props: {
            ...config,
            children: children.map(
                child => typeof child
                    child :
                    { type: ELEMENT_TEXT, props: { text: child, children: [] } })
        }
    }
}
+class Component {
+    constructor(props) {
+        this.props = props;
+        this.updateQueue = new UpdateQueue();
+    }
+    setState(payload) {
+        this.internalFiber.updateQueue.enqueueUpdate(new Update(payload));
+        scheduleRoot();
+    }
+}
+Component.prototype.isReactComponent = true;
let React = {
    createElement,
+    Component
}
export default React;
```

4.3 constants.js

src\constants.js

```
export const ELEMENT_TEXT = Symbol.for('ELEMENT_TEXT');

export const TAG_ROOT = Symbol.for('TAG_ROOT');
export const TAG_HOST = Symbol.for('TAG_HOST');
export const TAG_TEXT = Symbol.for('TAG_TEXT');
+export const TAG_CLASS = Symbol.for('TAG_CLASS');

export const UPDATE = Symbol.for('UPDATE');
export const PLACEMENT = Symbol.for('PLACEMENT');
export const DELETION = Symbol.for('DELETION');
```

4.4 updateQueue.js

src\updateQueue.js

```
export class Update {
    constructor(payload) {
        this.payload = payload;
    }
}
export class UpdateQueue {
    constructor() {
        this.firstUpdate = null;
        this.lastUpdate = null;
    }
    enqueueUpdate(update) {
        if (this.lastUpdate === null) {
            this.firstUpdate = this.lastUpdate = update;
        } else {
            this.lastUpdate.nextUpdate = update;
            this.lastUpdate = update;
        }
    }
    forceUpdate(state) {
        let currentUpdate = this.firstUpdate;
        while (currentUpdate) {
            state = typeof currentUpdate.payload == 'function' ? currentUpdate.payload(state) : currentUpdate.payload;
            currentUpdate = currentUpdate.nextUpdate;
        }
        this.firstUpdate = this.lastUpdate = null;
        return state;
    }
}
```

4.5 utils.js

src\utils.js

```
function setProp(dom, key, value) {
    if (/^on/.test(key)) {
        dom[key.toLowerCase()] = value;
    } else if (key
        if (value) {
            for (let styleName in value) {
                if (value.hasOwnProperty(styleName)) {
                    dom.style[styleName] = value[styleName];
                }
            }
        }
    } else {
        dom.setAttribute(key, value);
    }
    return dom;
}
export function setProps(elem, oldProps, newProps) {
    for (let key in oldProps) {
        if (key !== 'children') {
            if (newProps.hasOwnProperty(key)) {
                setProp(elem, key, newProps[key]);
            } else {
                elem.removeAttribute(key);
            }
        }
    }
    for (let key in newProps) {
        if (key !== 'children') {
            setProp(elem, key, newProps[key])
        }
    }
}

+export function deepEquals(obj1, obj2) {
+    let { children: oldChildren, ...oldProps } = obj1;
+    let { children: newChildren, ...newProps } = obj2;
+    return JSON.stringify(oldProps) === JSON.stringify(newProps);
+}
```

4.6 scheduler.js

src\scheduler.js

```
import { setProps,deepEquals } from './utils';
+import { UpdateQueue } from './updateQueue';
+import _ from 'lodash';
import {
+    ELEMENT_TEXT, TAG_ROOT, TAG_HOST, TAG_TEXT, TAG_CLASS, PLACEMENT, DELETION, UPDATE
} from './constants';
let currentRoot = null;         //当前的根Fiber
let workInProgressRoot = null; //正在渲染中的根Fiber
let nextUnitOfWork = null;      //下一个工作单元
let deletions = [];             //要删除的fiber节点

export function scheduleRoot(rootFiber) {
+    if (currentRoot && currentRoot.alternate) {
+        workInProgressRoot = currentRoot.alternate;
+        workInProgressRoot.alternate = currentRoot;
+        if (rootFiber) {
+            workInProgressRoot.props = rootFiber.props;
+        }
+    } else if (currentRoot) {
+        if (rootFiber) {
+            rootFiber.alternate = currentRoot;
+            workInProgressRoot = rootFiber;
+        } else {
+            workInProgressRoot = {
+                ...currentRoot,
+                alternate: currentRoot
```

```
+            }
+        }
+    } else {
+        workInProgressRoot = rootFiber;
+    }
+    workInProgressRoot.firstEffect = workInProgressRoot.lastEffect = workInProgressRoot.nextEffect = null;
+    nextUnitOfWork = workInProgressRoot;
+}

function commitRoot() {
    deletions.forEach(commitWork);
    let currentFiber = workInProgressRoot.firstEffect;
    while (currentFiber) {
        commitWork(currentFiber);
        currentFiber = currentFiber.nextEffect;
    }
    deletions.length = 0;//先把要删除的节点清空掉
+   workInProgressRoot.firstEffect = workInProgressRoot.lastEffect = null;//清除effect list
    currentRoot = workInProgressRoot;
    workInProgressRoot = null;
}
function commitWork(currentFiber) {
+     if (!currentFiber) {
+         return;
+     }
    let returnFiber = currentFiber.return;//先获取父Fiber
+    while (returnFiber.tag !== TAG_HOST && returnFiber.tag !== TAG_ROOT && returnFiber.tag !== TAG_TEXT) {//如果不是DOM节点就一直向上找,比如ClassCounter
+        returnFiber = returnFiber.return;
+    }
    const domReturn = returnFiber.stateNode;//获取父的DOM节点
    if (currentFiber.effectTag
+        let nextFiber = currentFiber;
+        while (nextFiber.tag !== TAG_HOST && nextFiber.tag !== TAG_TEXT) {
+            nextFiber = nextFiber.child;//必须向下找到一个DOM节点 比如Class Counter
+        }
        domReturn.appendChild(nextFiber.stateNode);
    } else if (currentFiber.effectTag
+        commitDeletion(currentFiber, domReturn);
    } else if (currentFiber.effectTag
        if (currentFiber.type
            if (currentFiber.alternate.props.text !== currentFiber.props.text) {
                currentFiber.stateNode.textContent = currentFiber.props.text;
            }
        } else {
            updateDOM(currentFiber.stateNode, currentFiber.alternate.props, currentFiber.props);
        }
    }
    currentFiber.effectTag = null;
}
+function commitDeletion(currentFiber, domReturn) {
+    if (currentFiber.tag === TAG_HOST || currentFiber.tag === TAG_TEXT) {
+        domReturn.removeChild(currentFiber.stateNode);
+    } else {
+        commitDeletion(currentFiber.child, domReturn);
+    }
+}
function performUnitOfWork(currentFiber) {
    beginWork(currentFiber);//开始渲染前的Fiber,就是把子元素变成子fiber

    if (currentFiber.child) {//如果子节点就返回第一个子节点
        return currentFiber.child;
    }

    while (currentFiber) {//如果没有子节点说明当前节点已经完成了渲染工作
        completeUnitOfWork(currentFiber);//可以结束此fiber的渲染了
        if (currentFiber.sibling) {//如果它有弟弟就返回弟弟
            return currentFiber.sibling;
        }
        currentFiber = currentFiber.return;//如果没有弟弟让爸爸完成，然后找叔叔
    }
}

function beginWork(currentFiber) {
    if (currentFiber.tag
        updateHostRoot(currentFiber);
    } else if (currentFiber.tag
        updateHostText(currentFiber);
    } else if (currentFiber.tag
        updateHostComponent(currentFiber);
+    } else if (currentFiber.tag === TAG_CLASS) {//如果是类组件
+        updateClassComponent(currentFiber)
+    }
}
+function updateClassComponent(currentFiber) {
+    if (currentFiber.stateNode === null) {
+        currentFiber.stateNode = new currentFiber.type(currentFiber.props);
+        currentFiber.stateNode.internalFiber = currentFiber;
+        currentFiber.updateQueue = new UpdateQueue();
+    }
+    currentFiber.stateNode.state = currentFiber.updateQueue.forceUpdate(currentFiber.stateNode.state);
+    const newChildren = [currentFiber.stateNode.render()];
+    reconcileChildren(currentFiber, newChildren);
+}
function updateHostText(currentFiber) {
    if (!currentFiber.stateNode) {
        currentFiber.stateNode = createDOM(currentFiber);//先创建真实的DOM节点
    }
}
function updateHostRoot(currentFiber) {//如果是根节点
    const newChildren = currentFiber.props.children;//直接渲染子节点
    reconcileChildren(currentFiber, newChildren);
}

function updateHostComponent(currentFiber) {//如果是原生DOM节点
```

```
    if (!currentFiber.stateNode) {
        currentFiber.stateNode = createDOM(currentFiber);//先创建真实的DOM节点
    }
    const newChildren = currentFiber.props.children;
    reconcileChildren(currentFiber, newChildren);
}
function createDOM(currentFiber) {
    if (currentFiber.type
        return document.createTextNode(currentFiber.props.text);
    }
    const stateNode = document.createElement(currentFiber.type);
    updateDOM(stateNode, {}, currentFiber.props);
    return stateNode;
}

function reconcileChildren(currentFiber, newChildren) {
    let newChildIndex = 0;//新虚拟DOM数组中的索引
    let oldFiber = currentFiber.alternate && currentFiber.alternate.child;//父Fiber中的第一个子Fiber
+   if (oldFiber) oldFiber.firstEffect = oldFiber.lastEffect = oldFiber.nextEffect = null;
    let prevSibling;
    while (newChildIndex < newChildren.length || oldFiber) {
        const newChild = newChildren[newChildIndex];
        let newFiber;
        const sameType = oldFiber && newChild && newChild.type
        let tag;
+        if (newChild && typeof newChild.type === 'function' && newChild.type.prototype.isReactComponent) {
+            tag = TAG_CLASS;//类组件
+        } else if (newChild && newChild.type === ELEMENT_TEXT) {
            tag = TAG_TEXT;//文本
        } else if (newChild && typeof newChild.type
            tag = TAG_HOST;//原生DOM组件
        }
        if (sameType) {
+            let { children: oldChildren, ...oldProps } = oldFiber.props;
+            let { children: newChildren, ...newProps } = newChild.props;
+            newFiber = {
+                tag,//标记Fiber类型，例如是函数组件或者原生组件
+                type: oldFiber.type,//具体的元素类型
+                props: newChild.props,//新的属性对象
+                stateNode: oldFiber.stateNode,//原生组件的话就存放DOM节点，类组件的话是类组件实例，函数组件的话为空，因为没有实例
+                return: currentFiber,//父Fiber
+                updateQueue: oldFiber.updateQueue || new UpdateQueue(),
+                alternate: oldFiber,//上一个Fiber 指向旧树中的节点
+                effectTag: deepEquals(oldProps, newProps) ? null : UPDATE,//副作用标识
+            }
        } else {
            if (newChild) {//类型不一样，创建新的Fiber,旧的不复用了
                newFiber = {
                    tag,//原生DOM组件
                    type: newChild.type,//具体的元素类型
                    props: newChild.props,//新的属性对象
                    stateNode: null,//stateNode肯定是空的
                    return: currentFiber,//父Fiber
                    effectTag: PLACEMENT//副作用标识
                }
            }
            if (oldFiber) {
                oldFiber.effectTag = DELETION;
                deletions.push(oldFiber);
            }
        }
        if (oldFiber) {  //比较完一个元素了，老Fiber向后移动1位
            oldFiber = oldFiber.sibling;
        }
       if (newFiber) {
            if (newChildIndex
                currentFiber.child = newFiber;//第一个子节点挂到父节点的child属性上
            } else {
                prevSibling.sibling = newFiber;
            }
            prevSibling = newFiber;//然后newFiber变成了上一个哥哥了
        }
        newChildIndex++;
    }
}

function updateDOM(stateNode, oldProps, newProps) {
    setProps(stateNode, oldProps, newProps);
}
function completeUnitOfWork(currentFiber) {
    const returnFiber = currentFiber.return;
    if (returnFiber) {
        if (!returnFiber.firstEffect) {
            returnFiber.firstEffect = currentFiber.firstEffect;
        }
        if (!!currentFiber.lastEffect) {
            if (!!returnFiber.lastEffect) {
                returnFiber.lastEffect.nextEffect = currentFiber.firstEffect;
            }
            returnFiber.lastEffect = currentFiber.lastEffect;
        }

        const effectTag = currentFiber.effectTag;
        if (effectTag) {
            if (!!returnFiber.lastEffect) {
                returnFiber.lastEffect.nextEffect = currentFiber;
            } else {
                returnFiber.firstEffect = currentFiber;
            }
            returnFiber.lastEffect = currentFiber;
        }
    }
}
```

```
function workLoop(deadline) {
    let shouldYield = false;
    while (nextUnitOfWork && !shouldYield) {
        nextUnitOfWork = performUnitOfWork(nextUnitOfWork);//执行一个任务并返回下一个任务
        shouldYield = deadline.timeRemaining() < 1;//如果剩余时间小于1毫秒就说明没有时间了，需要把控制权让给浏览器
    }
    //如果没有下一个执行单元了，并且当前渲染树存在，则进行提交阶段
    if (!nextUnitOfWork && workInProgressRoot) {
        commitRoot();
    }
    requestIdleCallback(workLoop);
}
//开始在空闲时间执行workLoop
requestIdleCallback(workLoop);
```

5.实现函数组件

5.1 src\index.js

src\index.js

```
+function FunctionCounter() {
+  return (
+
+      Count:0
+
+  )
+}
ReactDOM.render(
+  ,
   document.getElementById('root')
);
```

5.2 constants.js

src\constants.js

```
export const ELEMENT_TEXT = Symbol.for('ELEMENT_TEXT');

export const TAG_ROOT = Symbol.for('TAG_ROOT');
export const TAG_HOST = Symbol.for('TAG_HOST');
export const TAG_TEXT = Symbol.for('TAG_TEXT');
export const TAG_CLASS = Symbol.for('TAG_CLASS');
+export const TAG_FUNCTION = Symbol.for('TAG_FUNCTION');
export const UPDATE = Symbol.for('UPDATE');
export const PLACEMENT = Symbol.for('PLACEMENT');
export const DELETION = Symbol.for('DELETION');
```

5.3 scheduler.js

src\scheduler.js

```
import { setProps, deepEquals } from './utils';
import { UpdateQueue } from './updateQueue';
+import {
+    ELEMENT_TEXT, TAG_ROOT, TAG_HOST, TAG_TEXT, TAG_CLASS, TAG_FUNCTION, PLACEMENT, DELETION, UPDATE
+} from './constants';
let currentRoot = null;         //当前的根Fiber
let workInProgressRoot = null; //正在渲染中的根Fiber
let nextUnitOfWork = null;      //下一个工作单元
let deletions = [];             //要删除的fiber节点

export function scheduleRoot(rootFiber) {
    if (rootFiber) {
        workInProgressRoot = rootFiber; //把当前树设置为nextUnitOfWork开始进行调度
    } else {
        if (currentRoot.alternate) {
            workInProgressRoot = currentRoot.alternate;
            workInProgressRoot.alternate = currentRoot;
        } else {
            workInProgressRoot = {
                ...currentRoot,
                alternate: currentRoot
            }
        }
    }
    deletions.length = 0;
    nextUnitOfWork = workInProgressRoot;
}

function commitRoot() {
    deletions.forEach(commitWork);
    let currentFiber = workInProgressRoot.firstEffect;
    while (currentFiber) {
        commitWork(currentFiber);
        currentFiber = currentFiber.nextEffect;
    }
    deletions.length = 0;//先把要删除的节点清空掉
    workInProgressRoot.firstEffect = workInProgressRoot.lastEffect = null;
    currentRoot = workInProgressRoot;
    workInProgressRoot = null;
}
function commitWork(currentFiber) {
    if (!currentFiber) {
        return;
    }
    let returnFiber = currentFiber.return;//先获取父Fiber
    while (returnFiber.tag !== TAG_HOST && returnFiber.tag !== TAG_ROOT && returnFiber.tag !== TAG_TEXT) {//如果不是DOM节点就一直向上找
        returnFiber = returnFiber.return;
    }
    const domReturn = returnFiber.stateNode;//获取父的DOM节点
    if (currentFiber.effectTag
```

```
        let nextFiber = currentFiber;
        while (nextFiber.tag !== TAG_HOST && nextFiber.tag !== TAG_TEXT) {//必须向下找到一个DOM节点
            nextFiber = nextFiber.child;
        }
        domReturn.appendChild(nextFiber.stateNode);
    } else if (currentFiber.effectTag
        commitDeletion(currentFiber, domReturn);
    } else if (currentFiber.effectTag
        if (currentFiber.type
            if (currentFiber.alternate.props.text !== currentFiber.props.text) {
                currentFiber.stateNode.textContent = currentFiber.props.text;
            }
        } else {
            updateDOM(currentFiber.stateNode, currentFiber.alternate.props, currentFiber.props);
        }
    }
    currentFiber.effectTag = null;
}
function commitDeletion(currentFiber, domReturn) {
    if (currentFiber.tag
        domReturn.removeChild(currentFiber.stateNode);
    } else {
        commitDeletion(currentFiber.child, domReturn);
    }
}
function performUnitOfWork(currentFiber) {
    beginWork(currentFiber);//开始渲染前的Fiber,就是把子元素变成子fiber

    if (currentFiber.child) {//如果子节点就返回第一个子节点
        return currentFiber.child;
    }

    while (currentFiber) {//如果没有子节点说明当前节点已经完成了渲染工作
        completeUnitOfWork(currentFiber);//可以结束此fiber的渲染了
        if (currentFiber.sibling) {//如果它有弟弟就返回弟弟
            return currentFiber.sibling;
        }
        currentFiber = currentFiber.return;//如果没有弟弟让爸爸完成，然后找叔叔
    }
}

function beginWork(currentFiber) {
    if (currentFiber.tag
        updateHostRoot(currentFiber);
    } else if (currentFiber.tag
        updateHostText(currentFiber);
    } else if (currentFiber.tag
        updateHostComponent(currentFiber);
    } else if (currentFiber.tag
        updateClassComponent(currentFiber)
+    } else if (currentFiber.tag === TAG_FUNCTION) {//如果是函数组件
+        updateFunctionComponent(currentFiber);
+    }
}
+function updateFunctionComponent(currentFiber) {
+    const newChildren = [currentFiber.type(currentFiber.props)];
+    reconcileChildren(currentFiber, newChildren);
+}
function updateClassComponent(currentFiber) {
    if (currentFiber.stateNode
        currentFiber.stateNode = new currentFiber.type(currentFiber.props);
        currentFiber.stateNode.internalFiber = currentFiber;
        currentFiber.updateQueue = new UpdateQueue();
    }
    currentFiber.stateNode.state = currentFiber.updateQueue.forceUpdate(currentFiber.stateNode.state);
    const newChildren = [currentFiber.stateNode.render()];
    reconcileChildren(currentFiber, newChildren);
}
function updateHostText(currentFiber) {
    if (!currentFiber.stateNode) {
        currentFiber.stateNode = createDOM(currentFiber);//先创建真实的DOM节点
    }
}
function updateHostRoot(currentFiber) {//如果是根节点
    const newChildren = currentFiber.props.children;//直接渲染子节点
    reconcileChildren(currentFiber, newChildren);
}

function updateHostComponent(currentFiber) {//如果是原生DOM节点
    if (!currentFiber.stateNode) {
        currentFiber.stateNode = createDOM(currentFiber);//先创建真实的DOM节点
    }
    const newChildren = currentFiber.props.children;
    reconcileChildren(currentFiber, newChildren);
}
function createDOM(currentFiber) {
    if (currentFiber.type
        return document.createTextNode(currentFiber.props.text);
    }
    const stateNode = document.createElement(currentFiber.type);
    updateDOM(stateNode, {}, currentFiber.props);
    return stateNode;
}

function reconcileChildren(currentFiber, newChildren) {
    let newChildIndex = 0;//新虚拟DOM数组中的索引
    let oldFiber = currentFiber.alternate && currentFiber.alternate.child;//父Fiber中的第一个子Fiber
    let prevSibling;
    while (newChildIndex < newChildren.length || oldFiber) {
        const newChild = newChildren[newChildIndex];
        let newFiber;
        const sameType = oldFiber && newChild && newChild.type
        let tag;
        if (newChild && typeof newChild.type
```

```
            tag = TAG_CLASS;//类组件
+        } else if (newChild && typeof newChild.type === 'function') {
+            tag = TAG_FUNCTION;//函数组件
+        } else if (newChild && newChild.type === ELEMENT_TEXT) {
            tag = TAG_TEXT;//文本
        } else if (newChild && typeof newChild.type
            tag = TAG_HOST;//原生DOM组件
        }
        if (sameType) {
            newFiber = {
                tag,//标记Fiber类型，例如是函数组件或者原生组件
                type: oldFiber.type,//具体的元素类型
                props: newChild.props,//新的属性对象
                stateNode: oldFiber.stateNode,//原生组件的话就存放DOM节点，类组件的话是类组件实例，函数组件的话为空，因为没有实例
                return: currentFiber,//父Fiber
                updateQueue: oldFiber.updateQueue || new UpdateQueue(),
                alternate: oldFiber,//上一个Fiber 指向旧树中的节点
                effectTag: deepEquals(oldFiber.props, newChild.props) ? null : UPDATE,//副作用标识
            }
        } else {
            if (newChild) {//类型不一样，创建新的Fiber,旧的不复用了
                newFiber = {
                    tag,//原生DOM组件
                    type: newChild.type,//具体的元素类型
                    props: newChild.props,//新的属性对象
                    stateNode: null,//stateNode肯定是空的
                    return: currentFiber,//父Fiber
                    effectTag: PLACEMENT //副作用标识
                }
            }
            if (oldFiber) {
                oldFiber.effectTag = DELETION;
                deletions.push(oldFiber);
            }
        }
        if (oldFiber) {  //比较完一个元素了，老Fiber向后移动1位
            oldFiber = oldFiber.sibling;
        }
        if (newFiber) {
            if (newChildIndex
                currentFiber.child = newFiber;//第一个子节点挂到父节点的child属性上
            } else {
                prevSibling.sibling = newFiber;
            }
            prevSibling = newFiber;//然后newFiber变成了上一个哥哥了
        }
        newChildIndex++;
    }
}

function updateDOM(stateNode, oldProps, newProps) {
    setProps(stateNode, oldProps, newProps);
}
function completeUnitOfWork(currentFiber) {
    const returnFiber = currentFiber.return;
    if (returnFiber) {
        if (!returnFiber.firstEffect) {
            returnFiber.firstEffect = currentFiber.firstEffect;
        }
        if (!!currentFiber.lastEffect) {
            if (!!returnFiber.lastEffect) {
                returnFiber.lastEffect.nextEffect = currentFiber.firstEffect;
            }
            returnFiber.lastEffect = currentFiber.lastEffect;
        }

        const effectTag = currentFiber.effectTag;
        if (effectTag) {
            if (!!returnFiber.lastEffect) {
                returnFiber.lastEffect.nextEffect = currentFiber;
            } else {
                returnFiber.firstEffect = currentFiber;
            }
            returnFiber.lastEffect = currentFiber;
        }
    }
}

function workLoop(deadline) {
    let shouldYield = false;
    while (nextUnitOfWork && !shouldYield) {
        nextUnitOfWork = performUnitOfWork(nextUnitOfWork);//执行一个任务并返回下一个任务
        shouldYield = deadline.timeRemaining() < 1;//如果剩余时间小于1毫秒就说明没有时间了，需要把控制权让给浏览器
    }
    //如果没有下一个执行单元了，并且当前渲染树存在，则进行提交阶段
    if (!nextUnitOfWork && workInProgressRoot) {
        commitRoot();
    }
    requestIdleCallback(workLoop);
}
//开始在空闲时间执行workLoop
requestIdleCallback(workLoop);
```

6.实现hooks

6.1 src\index.js

src\index.js

```
import React from './react';
import ReactDOM from './react-dom';

+function reducer(state, action) {
+  switch (action.type) {
+    case 'ADD':
+      return { count: state.count + 1 };
+    default:
+      return state;
+  }
+}
+function FunctionCounter() {
+  const [numberState, setNumberState] = React.useState({ number: 0 });
+  const [countState, dispatch] = React.useReducer(reducer, { count: 0 });
+  return (
+
+       setNumberState(state => ({ number: state.number + 1 }))}>
+        Count: {numberState.number}
+
+
+       dispatch({ type: 'ADD' })}>
+        Count: {countState.count}
+
+
+  )
+}
ReactDOM.render(
  ,
  document.getElementById('root')
);
```

6.2 src\react.js

src\react.js

```
import { ELEMENT_TEXT } from './constants';
import { Update, UpdateQueue } from './updateQueue';
+import { scheduleRoot,useState,useReducer} from './scheduler';
function createElement(type, config, ...children) {
    delete config.__self;
    delete config.__source;
    return {
        type,
        props: {
            ...config,
            children: children.map(
                child => typeof child
                    child :
                    { type: ELEMENT_TEXT, props: { text: child, children: [] } })
        }
    }
}
class Component {
    constructor(props) {
        this.props = props;
        this.updateQueue = new UpdateQueue();
    }
    setState(payload) {
        this.internalFiber.updateQueue.enqueueUpdate(new Update(payload));
        scheduleRoot();
    }
}
Component.prototype.isReactComponent = true;
let React = {
    createElement,
    Component,
+    useState,
+    useReducer
}
export default React;
```

6.3 src\scheduler.js

src\scheduler.js

```
import { setProps, deepEquals } from './utils';
+import { UpdateQueue, Update } from './updateQueue';
import {
    ELEMENT_TEXT, TAG_ROOT, TAG_HOST, TAG_TEXT, TAG_CLASS, TAG_FUNCTION, PLACEMENT, DELETION, UPDATE
} from './constants';
let currentRoot = null;         //当前的根Fiber
let workInProgressRoot = null;  //正在渲染中的根Fiber
let nextUnitOfWork = null;      //下一个工作单元
let deletions = [];             //要删除的fiber节点
+let workInProgressFiber = null; //正在工作中的fiber
+let hookIndex = 0;              //hook索引
export function scheduleRoot(rootFiber) {
    if (rootFiber) {
        workInProgressRoot = rootFiber; //把当前树设置为nextUnitOfWork开始进行调度
    } else {
        if (currentRoot.alternate) {
            workInProgressRoot = currentRoot.alternate;
            workInProgressRoot.alternate = currentRoot;
        } else {
            workInProgressRoot = {
                ...currentRoot,
                alternate: currentRoot
            }
        }
    }
    deletions.length = 0;
    nextUnitOfWork = workInProgressRoot;
}
```

```
function commitRoot() {
    deletions.forEach(commitWork);
    let currentFiber = workInProgressRoot.firstEffect;
    while (currentFiber) {
        commitWork(currentFiber);
        currentFiber = currentFiber.nextEffect;
    }
    deletions.length = 0;//先把要删除的节点清空掉
    workInProgressRoot.firstEffect = workInProgressRoot.lastEffect = null;
    currentRoot = workInProgressRoot;
    workInProgressRoot = null;
}
function commitWork(currentFiber) {
    if (!currentFiber) {
        return;
    }
    let returnFiber = currentFiber.return;//先获取父Fiber
    while (returnFiber.tag !== TAG_HOST && returnFiber.tag !== TAG_ROOT && returnFiber.tag !== TAG_TEXT) {//如果不是DOM节点就一直向上找
        returnFiber = returnFiber.return;
    }
    const domReturn = returnFiber.stateNode;//获取父的DOM节点
    if (currentFiber.effectTag
        let nextFiber = currentFiber;
        while (nextFiber.tag !== TAG_HOST && nextFiber.tag !== TAG_TEXT) {//必须向下找到一个DOM节点
            nextFiber = nextFiber.child;
        }
        domReturn.appendChild(nextFiber.stateNode);
    } else if (currentFiber.effectTag
        commitDeletion(currentFiber, domReturn);
    } else if (currentFiber.effectTag
        if (currentFiber.type
            if (currentFiber.alternate.props.text !== currentFiber.props.text) {
                currentFiber.stateNode.textContent = currentFiber.props.text;
            }
        } else {
            updateDOM(currentFiber.stateNode, currentFiber.alternate.props, currentFiber.props);
        }
    }
    currentFiber.effectTag = null;
}
function commitDeletion(currentFiber, domReturn) {
    if (currentFiber.tag
        domReturn.removeChild(currentFiber.stateNode);
    } else {
        commitDeletion(currentFiber.child, domReturn);
    }
}
function performUnitOfWork(currentFiber) {
    beginWork(currentFiber);//开始渲染前的Fiber,就是把子元素变成子fiber

    if (currentFiber.child) {//如果子节点就返回第一个子节点
        return currentFiber.child;
    }

    while (currentFiber) {//如果没有子节点说明当前节点已经完成了渲染工作
        completeUnitOfWork(currentFiber);//可以结束此fiber的渲染了
        if (currentFiber.sibling) {//如果它有弟弟就返回弟弟
            return currentFiber.sibling;
        }
        currentFiber = currentFiber.return;//如果没有弟弟让爸爸完成，然后找叔叔
    }
}

function beginWork(currentFiber) {
    if (currentFiber.tag
        updateHostRoot(currentFiber);
    } else if (currentFiber.tag
        updateHostText(currentFiber);
    } else if (currentFiber.tag
        updateHostComponent(currentFiber);
    } else if (currentFiber.tag
        updateClassComponent(currentFiber)
    } else if (currentFiber.tag
        updateFunctionComponent(currentFiber);
    }
}
function updateFunctionComponent(currentFiber) {
+    workInProgressFiber = currentFiber;
+    hookIndex = 0;
+    workInProgressFiber.hooks = [];
    const newChildren = [currentFiber.type(currentFiber.props)];
    reconcileChildren(currentFiber, newChildren);
}
function updateClassComponent(currentFiber) {
    if (currentFiber.stateNode
        currentFiber.stateNode = new currentFiber.type(currentFiber.props);
        currentFiber.stateNode.internalFiber = currentFiber;
        currentFiber.updateQueue = new UpdateQueue();
    }
    currentFiber.stateNode.state = currentFiber.updateQueue.forceUpdate(currentFiber.stateNode.state);
    const newChildren = [currentFiber.stateNode.render()];
    reconcileChildren(currentFiber, newChildren);
}
function updateHostText(currentFiber) {
    if (!currentFiber.stateNode) {
        currentFiber.stateNode = createDOM(currentFiber);//先创建真实的DOM节点
    }
}
function updateHostRoot(currentFiber) {//如果是根节点
    const newChildren = currentFiber.props.children;//直接渲染子节点
    reconcileChildren(currentFiber, newChildren);
}
```

```
function updateHostComponent(currentFiber) {//如果是原生DOM节点
    if (!currentFiber.stateNode) {
        currentFiber.stateNode = createDOM(currentFiber);//先创建真实的DOM节点
    }
    const newChildren = currentFiber.props.children;
    reconcileChildren(currentFiber, newChildren);
}
function createDOM(currentFiber) {
    if (currentFiber.type
        return document.createTextNode(currentFiber.props.text);
    }
    const stateNode = document.createElement(currentFiber.type);
    updateDOM(stateNode, {}, currentFiber.props);
    return stateNode;
}

function reconcileChildren(currentFiber, newChildren) {
    let newChildIndex = 0;//新虚拟DOM数组中的索引
    let oldFiber = currentFiber.alternate && currentFiber.alternate.child;//父Fiber中的第一个子Fiber
    let prevSibling;
    while (newChildIndex < newChildren.length || oldFiber) {
        const newChild = newChildren[newChildIndex];
        let newFiber;
        const sameType = oldFiber && newChild && newChild.type
        let tag;
        if (newChild && typeof newChild.type
            tag = TAG_CLASS;//类组件
        } else if (newChild && typeof newChild.type
            tag = TAG_FUNCTION;//函数组件
        } else if (newChild && newChild.type
            tag = TAG_TEXT;//文本
        } else if (newChild && typeof newChild.type
            tag = TAG_HOST;//原生DOM组件
        }
        if (sameType) {
            newFiber = {
                tag,//标记Fiber类型，例如是函数组件或者原生组件
                type: oldFiber.type,//具体的元素类型
                props: newChild.props,//新的属性对象
                stateNode: oldFiber.stateNode,//原生组件的话就存放DOM节点，类组件的话是类组件实例，函数组件的话为空，因为没有实例
                return: currentFiber,//父Fiber
                updateQueue: oldFiber.updateQueue || new UpdateQueue(),
                alternate: oldFiber,//上一个Fiber 指向旧树中的节点
                effectTag: deepEquals(oldFiber.props, newChild.props) ? null : UPDATE,//副作用标识
            }
        } else {
            if (newChild) {//类型不一样，创建新的Fiber,旧的不复用了
                newFiber = {
                    tag,//原生DOM组件
                    type: newChild.type,//具体的元素类型
                    props: newChild.props,//新的属性对象
                    stateNode: null,//stateNode肯定是空的
                    return: currentFiber,//父Fiber
                    effectTag: PLACEMENT //副作用标识
                }
            }
            if (oldFiber) {
                oldFiber.effectTag = DELETION;
                deletions.push(oldFiber);
            }
        }
        if (oldFiber) {  //比较完一个元素了，老Fiber向后移动1位
            oldFiber = oldFiber.sibling;
        }
        if (newFiber) {
            if (newChildIndex
                currentFiber.child = newFiber;//第一个子节点挂到父节点的child属性上
            } else {
                prevSibling.sibling = newFiber;
            }
            prevSibling = newFiber;//然后newFiber变成了上一个哥哥了
        }
        newChildIndex++;
    }
}

function updateDOM(stateNode, oldProps, newProps) {
    setProps(stateNode, oldProps, newProps);
}
function completeUnitOfWork(currentFiber) {
    const returnFiber = currentFiber.return;
    if (returnFiber) {
        if (!returnFiber.firstEffect) {
            returnFiber.firstEffect = currentFiber.firstEffect;
        }
        if (!!currentFiber.lastEffect) {
            if (!!returnFiber.lastEffect) {
                returnFiber.lastEffect.nextEffect = currentFiber.firstEffect;
            }
            returnFiber.lastEffect = currentFiber.lastEffect;
        }

        const effectTag = currentFiber.effectTag;
        if (effectTag) {
            if (!!returnFiber.lastEffect) {
                returnFiber.lastEffect.nextEffect = currentFiber;
            } else {
                returnFiber.firstEffect = currentFiber;
            }
            returnFiber.lastEffect = currentFiber;
        }
    }
}
+export function useReducer(reducer, initialValue) {
```

```
+    let oldHook =
+        workInProgressFiber.alternate &&
+        workInProgressFiber.alternate.hooks &&
+        workInProgressFiber.alternate.hooks[hookIndex];
+    let newHook = oldHook;
+    if (oldHook) {
+        oldHook.state = oldHook.updateQueue.forceUpdate(oldHook.state);
+    } else {
+        newHook = {
+            state: initialValue,
+            updateQueue: new UpdateQueue()
+        };
+    }
+    const dispatch = action => {
+        newHook.updateQueue.enqueueUpdate(
+            new Update(reducer ? reducer(newHook.state, action) : action)
+        );
+        scheduleRoot();
+    }
+    workInProgressFiber.hooks[hookIndex++] = newHook;
+    return [newHook.state, dispatch];
+}
+export function useState(initState) {
+    return useReducer(null, initState)
+}
function workLoop(deadline) {
    let shouldYield = false;
    while (nextUnitOfWork && !shouldYield) {
        nextUnitOfWork = performUnitOfWork(nextUnitOfWork);//执行一个任务并返回下一个任务
        shouldYield = deadline.timeRemaining() < 1;//如果剩余时间小于1毫秒就说明没有时间了，需要把控制权让给浏览器
    }
    //如果没有下一个执行单元了，并且当前渲染树存在，则进行提交阶段
    if (!nextUnitOfWork && workInProgressRoot) {
        commitRoot();
    }
    requestIdleCallback(workLoop);
}
//开始在空闲时间执行workLoop
requestIdleCallback(workLoop);
```