手写react

废话不多说，我们直接开撸react16+的核心源码， 在这个版本，class component已经不是必须的了，我们重点的内容，就是function compoent + hooks，底层实现fiber架构

首先，关于虚拟dom的概念，欢迎移步 这里https://www.bilibili.com/video/av62275969

代码https://github.com/shengxinjing/simple_vdom

```
import React ,{useState}from 'react'
import ReactDOM from 'react-dom'


function App(props){
  let [count,setCount] = useState(0)
  return <div>
    <h1>{props.title}</h1>
    <p>{count}</p>
    <button onClick={()=>setCount(count+1)}>add</button>
  </div>
}

ReactDOM.render(<App title="开课吧" />, document.getElementById('root'))
```

核心api 大概就是 jsx(createElement) render 函数组件 和useState

jsx

这个大家都知道了，jsx写起来像html，其实是babel转义成React.createElement来执行的，用来构建虚拟dom， 在线体验 http://react.shengxinjing.cn/

```
<h1 title="foo">Kaikeba</h1>
```

解析成

```
 React.createElement(
      "h1",
      { title: "foo" },
      "Kaikeba"
)
```

这也是为什么用了jsx的文件，必须要impoert react的原因所在

我们知道 createElement就是为了构建虚拟dom 这种形式现在已经是组件化的最佳实践了，为什么需要jsx 就需要用虚拟dom概念说起, 简单的来说，就是用js的对象，来描述真实的dom元素

因为比如上面那个简单的div，渲染成dom后，就有非常多的属性，所以dom操作一直是前端性能的杀手, 随便打开一个网站，console.dir(document)你就知道

所以我们用js的对象

```
const element = {
  type: "h1",
  props: {
    title: "foo",
    children: "Kaikeba",
  },
}
```

可以完整的描述dom，后续又任何的修改需求，只需要频繁的操作这个dom，尽可能少的操作真实dom，这也是为什么虚拟dom性能良好的原因所在 两次操作之间会做diff，只做最少的修改次数

render就是遍历这个对象，渲染dom即可，这个后续会封装render函数，从jsx到**element**对象，就i是createElement函数需要做的

```
const element = {
  type: "h1",
  props: {
    title: "foo",
```

```
    children: "Kaikeba",
  },
}
const container = document.getElementById("root")
const node = document.createElement(element.type)
node["title"] = element.props.title
const text = document.createTextNode("")
text["nodeValue"] = element.props.children
node.appendChild(text)
container.appendChild(node)
```

createElement

构建虚拟dom，如果嵌套dom，比如我们使用这个JSX

```
<div id="container">
  <input value="foo" type="text" />
  <a href="/bar"></a>
  <span></span>
</div>
```

会解析成

```
React.createElement(
      "div",
      { id: "container" },
      React.createElement("input", { value: "foo", type: "text" }),
      React.createElement("a", { href: "/bar" }),
      React.createElement("span", null)
    )
```

期待返回下面这个对象

```
const element = {
  type: "div",
  props: {
    id: "container",
    children: [
      { type: "input", props: { value: "foo", type: "text" } },
      { type: "a", props: { href: "/bar" } },
      { type: "span", props: {} }
    ]
  }
};
```

然后代码就呼之欲出了，毕竟转义的工作都被babel做了

```
/**
 *
 * @param {str|function} 类型，是字符串div 还是函数
 * @param {*} jsx传递的属性
 * @param  {...any} 子元素
 */
function createElement(type, props, ...children) {
  delete props.__source
  delete props.__self
  return {
    type,
    props: {
      ...props,
      children,
    },
  }
}

function render(vdom, container){
  container.innerHTML = "<pre>"+JSON.stringify(vdom,null,2)+"</pre>"
}

export default {
  createElement,
  render
}
```

main.js

```
import React from './yolkjs'
const ReactDOM = React
let element = <div id="container">
  <input value="foo" type="text" />
  <a href="/bar">测试</a>
  <span>开课吧</span>
</div>

ReactDOM.render( element, document.getElementById('root'))
```

```
{
  "type": "div",
  "props": {
    "id": "container",
    "children": [
      {
        "type": "input",
        "props": {
          "value": "foo",
          "type": "text",
          "children": []
        }
      },
      {
        "type": "a",
        "props": {
          "href": "/bar",
          "children": [
            "测试"
          ]
        }
      },
      {
        "type": "span",
        "props": {
          "children": [
            "开课吧"
          ]
        }
      }
    ]
  }
}
```

修正一下children的类型

```
function createElement(type, props, ...children) {
  return {
    type,
    props: {
      ...props,
      children: children.map(child =>
        typeof child === "object"
```

```
            ? child
            : createTextElement(child)
        ),
      },
    }
}
/** 文本类型vdom创建 */
function createTextElement(text) {
  return {
    type: "TEXT",
    props: {
      nodeValue: text,
      children: [],
    },
  }
}
```

```
{
  "type": "div",
  "props": {
    "id": "container",
    "children": [
      {
        "type": "input",
        "props": {
          "value": "foo",
          "type": "text",
          "children": []
        }
      },
      {
        "type": "a",
        "props": {
          "href": "/bar",
          "children": [
            {
              "type": "TEXT",
              "props": {
                "nodeValue": "测试",
                "children": []
              }
            }
          ]
        }
      },
      {
        "type": "span",
        "props": {
          "children": [
            {
              "type": "TEXT",
              "props": {
                "nodeValue": "开课吧",
                "children": []
              }
            }
          ]
        }
      }
    ]
  }
}
```

render

现在的render 只是简单的渲染一个对象，我们需要转成真实的dom 渲染 这一步没啥特别的 就是挨个遍历 创建dom 然后appendChild

```
function render(vdom, container){
  const dom = vdom.type == "TEXT"
    ? document.createTextNode("")
    : document.createElement(vdom.type)

  // 设置属性
  Object.keys(vdom.props)
    .forEach(name => {
      if(name!=="children"){
        dom[name] = vdom.props[name]
      }
    })

  // 递归渲染子元素
  vdom.props.children.forEach(child => render(child, dom))
  container.appendChild(dom)
}
```

Concurrent

注意上面的render，一旦开始，就开始递归，本身这个没啥问题，但是如果应用变得庞大后，会有卡顿， 后面状态修改后的diff也是一样， 整个vdom对象变大后，diff的过程也有会递归过多导致的卡顿

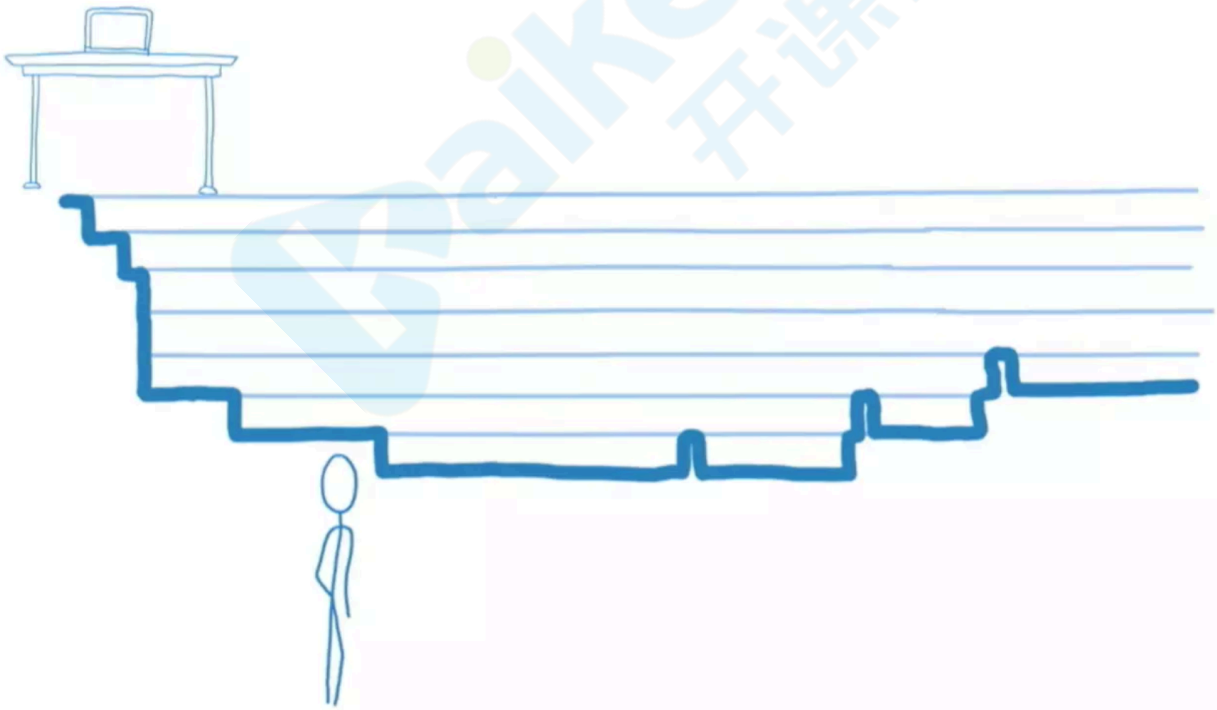
那么咋解决这个问题呢

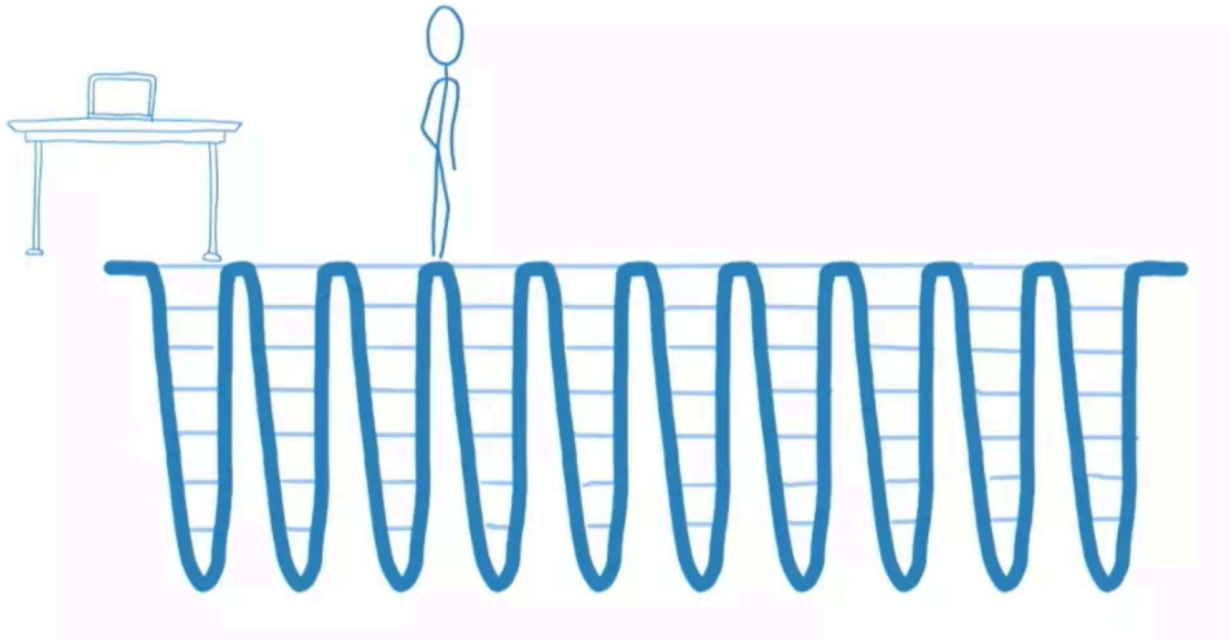
浏览器又一个api requestIdleCallback 可以利用浏览器的业余时间，我们可以把任务分成一个个的小人物，然后利用浏览器空闲时间来做diff，如果当前又任务来了，比如用户的点击或者动画，会先执行，然后空闲后，再回去把requestIdleCallback没完成的任务完成

https://developer.mozilla.org/zh-CN/docs/Web/API/Window/requestIdleCallback

```
requestIdleCallback(myNonEssentialWork);

function scedule (deadline) {

  // 如果帧内有富余的时间，或者超时 参数是requestIdleCallback传递的
  while ((deadline.timeRemaining() > 0) &&
         tasks.length > 0)
    doWorkIfNeeded();

  if (tasks.length > 0)
    requestIdleCallback(myNonEssentialWork);
}
```

```
let nextUnitOfWork = null
function workLoop(deadline) {
  // 有任务，并且当前帧还没结束
  while (nextUnitOfWork && deadline.timeRemaining()>1) {
    // 获取下一个任务单元
    nextUnitOfWork = performUnitOfWork(
      nextUnitOfWork
    )
  }
  requestIdleCallback(workLoop)
}
requestIdleCallback(workLoop)
function performUnitOfWork(nextUnitOfWork) {
  // 干活的代码
}
```

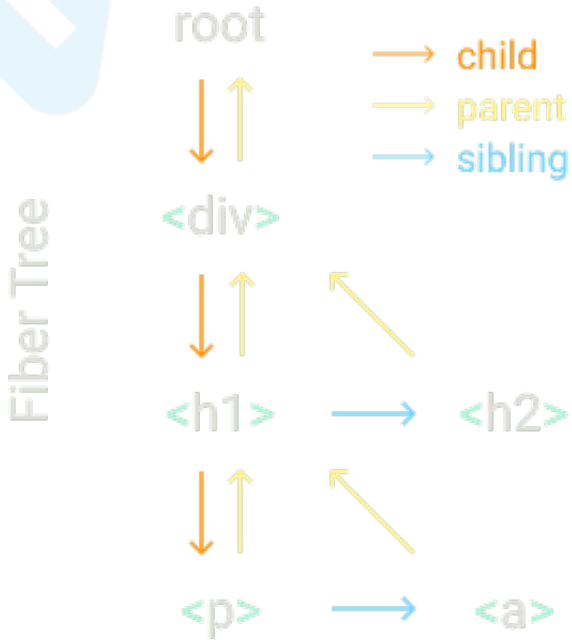




当然react已经重写了调度逻辑，不用requestIdleCallback了，但是过程是一致的

fibers

我们有了调度逻辑，之前的vdom结构是一个树形结构，他的diff过程是没法中断的。为了管理我们vdom树之间的关系，我们需要把 树形结构的内部关系，改造成链表（方便终止） 之前只是children作为一个数组递归遍历，现在父=》子，子=》父，子=》兄弟，都有关系



整个任务从render开始，然后每次只遍历一个小单元，一旦被打断 就会去执行优先级高的任务(用户交互，动画) 回来后，由于回来的元素知道父，子，兄弟元素，很容易恢复遍历状态


```
/** 创建dom,  根据vdom or fiber */
function createDom(vdom){
  const dom = vdom.type == "TEXT"
    ? document.createTextNode("")
    : document.createElement(vdom.type)

  // 设置属性
  Object.keys(vdom.props)
    .forEach(name => {
      if(name!=="children"){
        dom[name] = vdom.props[name]
      }
    })
  return dom
}
function render(vdom, container){
  // 设置全局 nextUnitOfWork
  nextUnitOfWork = {
    dom: container,
    props: {
      children: [vdom],
    },
  }
}
```

显然render后 有了全局nextUnitOfWork

```
function performUnitOfWork(fiber) {
  // TODO add dom node
```

```
  // TODO create new fibers
  // TODO return next unit of work

  // 如果没有dom 就不是入口，直接创建dom
  if (!fiber.dom) {
    fiber.dom = createDom(fiber)
  }
  // fiber父元素
  if (fiber.parent) {
    fiber.parent.dom.appendChild(fiber.dom)
  }
  // 子元素遍历， 把children数组，变成链表
  const elements = fiber.props.children
  let index = 0
  let prevSibling = null
  while (index < elements.length) {
    const element = elements[index]
    const newFiber = {
      type: element.type,
      props: element.props,
      parent: fiber,
      dom: null,
    }
    // 第一个
    if (index === 0) {
      fiber.child = newFiber
    } else {
      // 其他通过sibling
      prevSibling.sibling = newFiber
    }
    prevSibling = newFiber
    index++
  }


  // fiber遍历顺序
  // 子 =》 子的兄弟 => 没有兄弟了=> 父元素
  if (fiber.child) {
    return fiber.child
  }
  let nextFiber = fiber
  while (nextFiber) {
    if (nextFiber.sibling) {
      return nextFiber.sibling
    }
    nextFiber = nextFiber.parent
  }
}
```

提交 commit

我们给dom添加节点的时候，如果渲染的过程中，被打断的，ui渲染会变得很奇怪，所以我们应该把dom操作独立出来，我们用一个全局变量来存储正在工作的fiber根节点(workInprogress tree)

```
function commitRoot() {
  commitWork(wipRoot.child)
  // 取消wip
  wipRoot = null
}

function commitWork(fiber) {
  if (!fiber) {
    return
  }
  const domParent = fiber.parent.dom
  domParent.appendChild(fiber.dom)
  commitWork(fiber.child)
  commitWork(fiber.sibling)
}
```

Reconciliation

现在我们已经能渲染了，但是如何做更新和删除节点呢

我们需要保存一个被中断前工作的fiber节点 currentRoot,以及每个fiber 都有一个字段，存储这上一个状态的fiber

并且针对子元素，设计一个reconcileChildren函数

```
function reconcileChildren(wipFiber, elements) {
  let index = 0
  let prevSibling = null
  while (index < elements.length) {
    const element = elements[index]
    const newFiber = {
      type: element.type,
      props: element.props,
      parent: wipFiber,
      dom: null,
    }
    if (index === 0) {
```

```
      wipFiber.child = newFiber
    } else {
      prevSibling.sibling = newFiber
    }
    prevSibling = newFiber
    index++
  }
}
```

加入wip的alternate的fiber对比

```
function reconcileChildren(wipFiber, elements) {
  let index = 0
  let oldFiber =
    wipFiber.alternate && wipFiber.alternate.child
  let prevSibling = null
  while (
    index < elements.length ||
    oldFiber != null
  ) {
    const element = elements[index]
    let newFiber = null
    // 对比old和new
     const sameType =
      oldFiber &&
      element &&
      element.type == oldFiber.type
    if (sameType) {
      // TODO update the node
    }
    if (element && !sameType) {
      // TODO add this node
    }
    if (oldFiber && !sameType) {
      // TODO delete the oldFiber's node
    }


    if (oldFiber) {
      oldFiber = oldFiber.sibling
    }
    if (index === 0) {
      wipFiber.child = newFiber
    } else if (element) {
      prevSibling.sibling = newFiber
    }
    prevSibling = newFiber
    index++
```

```
  }
}
```

如果类型相同，dom可以福永，更新节点即可 用effectTag标记

```
newFiber = {
        type: oldFiber.type,
        props: element.props,
        dom: oldFiber.dom,
        parent: wipFiber,
        alternate: oldFiber,
        effectTag: "UPDATE",
      }
```

如果类型不行，直接替换

```
newFiber = {
        type: element.type,
        props: element.props,
        dom: null,
        parent: wipFiber,
        alternate: null,
        effectTag: "PLACEMENT",
      }
```

如果需要删除

```
 oldFiber.effectTag = "DELETION"
 deletions.push(oldFiber)
```

dom更新

```
// dom更新
function updateDom(dom, prevProps, nextProps) {
    Object.keys(prevProps)
    .filter(name=>name!=="children")
    .filter(name=> !(name in nextProps))
    .forEach(name => {
      // 删除
      if(name.slice(0,2)=='on'){
```

```
        dom.removeEventListener(name.slice(2).toLowerCase(),
prevProps[name],false)
      }else{
        dom[name] = ''
      }
    })

    Object.keys(nextProps)
    .filter(name=>name!=="children")
    .forEach(name => {
      // 删除
      if(name.slice(0,2)=='on'){
        dom.addEventListener(name.slice(2).toLowerCase(),
nextProps[name],false)
      }else{
        dom[name] = nextProps[name]
      }
    })

}
```

函数组件

```
function App(props){
  return <div id="container" className="red">
    <h1>{props.title}</h1>
    <input value="foo" type="text" />
    <a href="/bar">测试</a>
    <span onClick={()=>alert(3)}>开课吧</span>
  </div>
}
let element = <App title="开课吧" />
```

函数也是一样的，只不过type是函数，而不是字符串，我们需要在处理vdom的时候识别初和普通dom的区别

1. 根据type执行不同的函数来初始化fiber
2. 函数组件没有dom属性 （没有dom属性，查找dom需要想上循环查找）

```
  const isFunctionComponent = fiber.type instanceof Function
  if (isFunctionComponent) {
    updateFunctionComponent(fiber)
  } else {
    updateHostComponent(fiber)
  }
```

```
function updateFunctionComponent(fiber) {
  // 执行函数，传入props
  const children = [fiber.type(fiber.props)]
  reconcileChildren(fiber, children)
}
function updateHostComponent(fiber) {
  if (!fiber.dom) {
    fiber.dom = createDom(fiber)
  }
  reconcileChildren(fiber, fiber.props.children)
}
```

Hooks

重点来了，状态 也就是state 实际上hooks是通过链表来查找具体的state，这里我们通过数组来简单模拟一下 把useState存储的hooks，存储在fiber中

```
import React from './yolkjs'
const ReactDOM = React
function App(props){
  const [count, setCount] = React.useState(1)
  return <div id="container" className="red">
    <h1>{props.title}, {count}</h1>
    <button onClick={()=>setCount(count+1)}></button>
  </div>
}
let element = <App title="开课吧" />


ReactDOM.render( element, document.getElementById('root'))
```

渲染

```
function useState(init){
  const oldHook =
  wipFiber.base &&
  wipFiber.base.hooks &&
  wipFiber.base.hooks[hookIndex]
  const hook = {
    state: oldHook ? oldHook.state : init,
  }
  wipFiber.hooks.push(hook)
  hookIndex++
  return [hook.state]
}
```

修改set

```
function useState(init){
  const oldHook =
  wipFiber.base &&
  wipFiber.base.hooks &&
  wipFiber.base.hooks[hookIndex]
  const hook = {
    state: oldHook ? oldHook.state : init,
    queue: [],
  }
  const actions = oldHook ? oldHook.queue : []
  actions.forEach(action => {
    hook.state = action
  })
  const setState = action => {
    hook.queue.push(action)
    wipRoot = {
      dom: currentRoot.dom,
      props: currentRoot.props,
      base: currentRoot,
    }
    nextUnitOfWork = wipRoot
    deletions = []
  }
  wipFiber.hooks.push(hook)
  hookIndex++
  return [hook.state, setState]
}
```

class

由于我们的重点是hooks，我们可以i尝试用hooks简单模拟一下class

```
class Component {
  constructor(props){
    this.props = props
    // this.state = {}
  }
}
function useComponent(Component){
  return function(props){
    const component = new Component(props)
    // 简单的规避eslitn
    let initState = useState
    let [state, setState] = initState(component.state)
    component.props = props
    component.state = state
```

```
    component.setState = setState
    console.log(component)
    return component.render()
  }
}
```

```
class Demo extends React.Component{
  constructor(props){
    super(props)
    this.state = {
      count:1
    }
  }
  handleClick = ()=>{
    this.setState({
      count:this.state.count+1
    })
  }
  render(){
    return <div>
      <h2 onClick={this.handleClick}>{this.state.count}</h2>
    </div>
  }
}
Demo = React.useComponent(Demo)
```

yeah 是不是略显骚气