vue源码

Vue3 Runtime-dom

runtime-dom 调用 runtime-core , runtime-core 调用reactivity

```
const {createApp, h} = runtime-dom
const app = {
    render() {
        h('h1', 'hello world')
    }
}
createApp(app)
```

h函数用来生成虚拟dom，createApp接受一个虚拟dom作为参数让虚拟dom渲染成真实的dom然后让其渲染出来。vue3使用了composition api，上面的使用方法可以更换成下面这样的，

```
const {createApp, h} = runtime-dom

const app = {
    setup() {
        return ()=>{
            return h('h1','hello world',count)
        }
    }
}
```

setup函数作用实际和render函数相同。其主要的进步就是可以让我们把代码聚拢起来方便操作不需要上下寻找，比如下面的这样

```
function abc() {
    const a = reactivity({a: 123})

    return {a}
}
const app = {
    let a = abc()
    setup() {
        return ()=>{
            return h('h1','hello world', a    )
        }
    }
}
```

abc函数可以独立于主函数之外，其实就是解耦，提高了代码的单一原则，方便维护也方便测试。

> ⓘ setup方法可以返回对象，也可以返回函数。返回函数就是渲染使用，返回对象就是给模版使用。

runtime-dom 主要提供了dom操作的api，比如 document.query 。runtime-dom单独拿出来因为可以让其平台独立，不同的平台可以有不同的dom操作方法。比如 app，web，小程序。runtime-dom提供了dom的操作包装给runtime-core使用。

```
export const nodeOps = {
     createElement: tagName=>document.createElement(tagName),
     remove: child=>child.parentNode && child.parentNode.removeChild(child),
     insert: (child,parent,anchor=null)=>parent.insertBefore(child,anchor),
     querySelector: selector=>document.querySelector(selector),
     setElementText: (el, text)=>el.textContent = text,
     createText:text=>docuemnt.createTextNode(text),
     setText: (node, text) => node.nodeValue = text,
     getParent: (node)=>node.parentNode,
     getNextSibling: (node)=>node.getNextElementSibling
}
```

可以随我们的需要随时添加dom的操作。然后我们来处理如何对dom属性操作比如style, class, event 等等的操作。

基本逻辑我们来举个例子，传入新的style，我们首先会把旧的style 里面的所有属性遍历对比新的style里面是否有，如果有，则覆盖，如果没有则先从旧的style里面删除，代码如下，

```
const patchStyle = (el, old, new) => {
    if (next == null) {
        el.removeAttribute('style')
    } else {
        if (old) {
            for (const key in old) {
                if (!new[key]) {
                    el.style[key] = ''
                }
            }
        }

        for (const key in new ) {
         el.style[key] = new[key]
        }
    }


}
```

我们接着来处理event，这里我们首先得知道多个addEventListener不会覆盖前面一个，所以我们现在明面上可以想到的最简单的方法就是首先清除旧的然后添加新的。这么做主要的问题之一就是消耗性能。解决这个

问题有2个方法，第一就是做代理利用冒泡的原理。第二个就是做缓存，vue里面使用的就是缓存。react使用的是代理。我们这里讲一下第二种的实现方法。

```
function createInvoker(fn) {
    const invoker=(e)=>{invoker.value(e)}
    invoker.value = fn

    return invoker
}

const patchEvent = (el, key, next)=>{
    const invokers = el._vei || (el._vei = {})
    const exists = invokers[key]
    const name = key.toLowerCase().slice(2)
    if (exists) {
        exists.value = next
    } else {

        if (next) {
            const invoker = invokers[key] = createInvoker(next)
            el.addEventListener(name, (e)=>{ invoker.value(e) })
        } else {

        }
    }


}
```

代码非常简单，通过对象的特性实现了缓存，即使同一个event改变了，我们只需要改变其invoker的value值。

类似的我们可以处理其他2个class 和 attribute，

```
const patchClass = (el, next)=> {
    if (next == null) {
        next =''
    }
    el.className = next
}

const patchAttr = (el, prev, next) => {
    if (next != null ) {
        el.setAttribute = next
    } else {
        el.removeAttribute(prev)
    }
}
```