介绍

第二章响应系统

就是通过拦截对象的属性上的读取和设置方法，当读取的时候将与该对象的属性相关的副作用函数加入到"桶"中，当设置该属性的时候再从桶中取出副作用函数来执行。

响应式数据的基本实现

```
 const bucket = new Set()
03
04 // 原始数据
05 const data = { text: 'hello world' }
06 // 对原始数据的代理
07 const obj = new Proxy(data, {
08   // 拦截读取操作
09   get(target, key) {
10     // 将副作用函数 effect 添加到存储副作用函数的桶中
11     bucket.add(effect)
12     // 返回属性值
13     return target[key]
14   },
15   // 拦截设置操作
16   set(target, key, newVal) {
17     // 设置属性值
18     target[key] = newVal
19     // 把副作用函数从桶里取出并执行
20     bucket.forEach(fn => fn())
21     // 返回 true 代表设置操作成功
22     return true
23   }
24 })
```

目前就是拦截数据(使用到了Proxy):

- 当遇到读取操作，将副作用函数收集到桶中
- 当设置操作时，蒋从桶中取出副作用函数并执行。

搭建完整的响应系统

搭建了基本的响应式：解决了硬编码，进行了收集(trace)和触发(trigger)的封装。

```
01const bucket = new WeakMap()
const obj = new Proxy(data, {
02   // 拦截读取操作
03   get(target, key) {
04     // 将副作用函数 activeEffect 添加到存储副作用函数的桶中
05     track(target, key)
06     // 返回属性值
```

```
    return target[key]
  },
  // 拦截设置操作
  set(target, key, newVal) {
    // 设置属性值
    target[key] = newVal
    // 把副作用函数从桶里取出并执行
    trigger(target, key)
  }
})

// 在 get 拦截函数内调用 track 函数追踪变化
function track(target, key) {
  // 没有 activeEffect，直接 return
  if (!activeEffect) return
  let depsMap = bucket.get(target)
  if (!depsMap) {
    bucket.set(target, (depsMap = new Map()))
  }
  let deps = depsMap.get(key)
  if (!deps) {
    depsMap.set(key, (deps = new Set()))
  }
  deps.add(activeEffect)
}
// 在 set 拦截函数内调用 trigger 函数触发变化
function trigger(target, key) {
  const depsMap = bucket.get(target)
  if (!depsMap) return
  const effects = depsMap.get(key)
  effects && effects.forEach(fn => fn())
}

// 用一个全局变量存储被注册的副作用函数
let activeEffect
function effect(fn) {
  const effectFn = () => {
    // 当 effectFn 执行时，将其设置为当前激活的副作用函数
    activeEffect = effectFn
    fn()
  }
  // activeEffect.deps 用来存储所有与该副作用函数相关联的依赖集合
  effectFn.deps = []
  // 执行副作用函数
  effectFn()
}
```

进行了分支切换的依赖清除，在执行时清除依赖，之后执行的时候才重新收集依赖

```
function track(target, key) {
  // 没有 activeEffect，直接 return
  if (!activeEffect) return
  let depsMap = bucket.get(target)
  if (!depsMap) {
    bucket.set(target, (depsMap = new Map()))
```

```
  }
  let deps = depsMap.get(key)
  if (!deps) {
    depsMap.set(key, (deps = new Set()))
  }
  // 把当前激活的副作用函数添加到依赖集合 deps 中
  deps.add(activeEffect)
  // deps 就是一个与当前副作用函数存在联系的依赖集合
  // 将其添加到 activeEffect.deps 数组中
  activeEffect.deps.push(deps) // 新增
}
```

用一个全局变量存储被注册的副作用函数

```
let activeEffect
function effect(fn) {
  const effectFn = () => {
    // 调用 cleanup 函数完成清除工作
    cleanup(effectFn)  // 新增
    activeEffect = effectFn
    fn()
  }
  effectFn.deps = []
  effectFn()
}
```

下面是 cleanup 函数的实现：

```
function cleanup(effectFn) {
  // 遍历 effectFn.deps 数组
  for (let i = 0; i < effectFn.deps.length; i++) {
    // deps 是依赖集合
    const deps = effectFn.deps[i]
    // 将 effectFn 从依赖集合中移除
    deps.delete(effectFn)
  }
  // 最后需要重置 effectFn.deps 数组
  effectFn.deps.length = 0
}

//接着修改trigger部分，防止无线循环
function trigger(target, key) {
  const depsMap = bucket.get(target)
  if (!depsMap) return
  const effects = depsMap.get(key)

  const effectsToRun = new Set(effects)  // 新增
  effectsToRun.forEach(effectFn => effectFn())  // 新增
  // effects && effects.forEach(effectFn => effectFn()) // 删除
}
```

effect的嵌套:解决方案，声明一个栈接着在栈中推进副作用函数，当执行了之后就弹出

```
// 用一个全局变量存储当前激活的 effect 函数
let activeEffect
// effect 栈
const effectStack = []  // 新增
```

```
05
06 function effect(fn) {
07   const effectFn = () => {
08     cleanup(effectFn)
09     // 当调用 effect 注册副作用函数时，将副作用函数赋值给 activeEffect
10     activeEffect = effectFn
11     // 在调用副作用函数之前将当前副作用函数压入栈中
12     effectStack.push(effectFn)  // 新增
13     fn()
14     // 在当前副作用函数执行完毕后，将当前副作用函数弹出栈，并把 activeEffect 还原为之前的
值
15     effectStack.pop()  // 新增
16     activeEffect = effectStack[effectStack.length - 1]  // 新增
17   }
18   // activeEffect.deps 用来存储所有与该副作用函数相关的依赖集合
19   effectFn.deps = []
20   // 执行副作用函数
21   effectFn()
22 }
```

防止无线循环:obj.foo = obj.foo + 1,obj.foo++

```
01 function trigger(target, key) {
02   const depsMap = bucket.get(target)
03   if (!depsMap) return
04   const effects = depsMap.get(key)
05
06   const effectsToRun = new Set()
07   effects && effects.forEach(effectFn => {
08     // 如果 trigger 触发执行的副作用函数与当前正在执行的副作用函数相同，则不触发执行
09     if (effectFn !== activeEffect) {  // 新增
10       effectsToRun.add(effectFn)
11     }
12   })
13   effectsToRun.forEach(effectFn => effectFn())
14   // effects && effects.forEach(effectFn => effectFn())
15 }
```

可调度器

就是在触发副作用函数的时候，可以让用户来控制副作用函数的执行时机和次数。

实现计算属性computed与lazy

懒执行的effect就是lazy的effect

dirty就是做一个缓存

说一下思路

computed的脏值检查，懒值，以及响应式处理等功能

计算属性就是一个懒执行的副作用函数。

> 接受一个 `getter` 函数，并根据 `getter` 的返回值返回一个`不可变的响应式 ref 对象`
>
> computed`的特点是缓存，计算属性是一个`effect`，计算属性收集`外层effect

> 多次取值 如果依赖的属性未发生变化是不会重新计算的
>
> `__dirty` 计算属性中的缓存标识，如果依赖有变化重新执行（this.effect的 调度函数执行）没变化不重新执行

```
function effect(fn, options = {}) {
02   const effectFn = () => {
03     cleanup(effectFn)
04     activeEffect = effectFn
05     effectStack.push(effectFn)
06     // 将 fn 的执行结果存储到 res 中
07     const res = fn()  // 新增
08     effectStack.pop()
09     activeEffect = effectStack[effectStack.length - 1]
10     // 将 res 作为 effectFn 的返回值
11     return res  // 新增
12   }
13   effectFn.options = options
14   effectFn.deps = []
15   if (!options.lazy) {
16     effectFn()
17   }
18
19   return effectFn
20 }


function computed(getter) {
02   let value

//下面这一块实现的是computedRef
04   let dirty = true//默认更新
05   const effectFn = effect(getter, {
06     lazy: true,
07     scheduler() {
08       if (!dirty) {
    //依赖的值变化会更新dirty并触发更新
09         dirty = true
10         // 当计算属性依赖的响应式数据变化时，手动调用 trigger 函数触发响
11         trigger(obj, 'value')
12       }
13     }
14   })
15
16   const obj = {
17     get value() {
18       if (dirty) {
            //执行函数
19         value = effectFn()
    //关闭开关
20         dirty = false
21       }
22       // 当读取 value 时，手动调用 track 函数进行追踪
23       track(obj, 'value')
24       return value
```

```
    }
  }

  return obj
}
```

watch实现原理

当数据发生变化时，执行了对应的回调函数。watch的实质其实就是利用了effect和scheduler。

```
function watch(source, cb) {
  let getter
  if (typeof source === 'function') {
    getter = source
  } else {
    getter = () => traverse(source)
  }
  // 定义旧值与新值
  let oldValue, newValue
  // 使用 effect 注册副作用函数时，开启 lazy 选项，并把返回值存储到 effectFn 中以便后续手动调用
  const effectFn = effect(
    () => getter(),
    {
      lazy: true,
      scheduler() {
        // 在 scheduler 中重新执行副作用函数，得到的是新值
        newValue = effectFn()
        // 将旧值和新值作为回调函数的参数
        cb(newValue, oldValue)
        // 更新旧值，不然下一次会得到错误的旧值
        oldValue = newValue
      }
    }
  )
  // 手动调用副作用函数，拿到的值就是旧值
  oldValue = effectFn()
}
```

在watch函数最后面执行一下effectFn()拿到返回值（旧值），等下一次scheduler函数执行的时候调用effectFn拿到新的返回值（新值）。然后把新值和旧值传给cb函数，并更新旧值。

非原始值的响应方案

理解Proxy和Reflect

Proxy是代理对象的，拦截对象里面属性的也可以自定义对象的操作。

Reflect里面定义的一些方法和对象的方法有许多重复的，但为什么会出现他呢，就是可以更像对象编程一样，代码更加可维护。

在vue中一般就是Proxy和Reflect是配合使用的,因为Reflect的第三个参数receiver类似指定函数调用时指定this。

```
const p = new Proxy(obj,{
    get(target,key,receiver){
        track(target,key)
        //这样就是说明getter函数内部的this指向了原始对象obj，说明我们访问的就是obj.foo，并不是代理对象上的，所以就需要Reflect的第三个参数来指定this了。
       // return target[key]
       return Reflect.get(target,key,receiver)
    }
})
```

代理Object

拦截对象里面的属性的话考虑

obj.foo：get就可以拦截到

in：has来进行拦截

for...in..：ownKeys来拦截，新增字段的时候也要执行for..in相关的副作用函数

```
ownKeys(target){
    return Reflect.ownKeys(target,INTER_SYMBOL)
}
```

与新添属性不同，修改属性的话对for..in没有影响，不需要触发副作用函数重新执行，所以需要在设置属性操作的时候就需要在set拦截函数里面区分操作类型，是新增还是设置已有属性（使用Object.prototype.hasOwnProperty检查当前操作的属性是否已经存在这个目标对象上），当为ADD的时候才执行for..in相关副作用函数，Delete也类似。

合理触发响应

封装了一个reactive函数，就是对Proxy做了一层封装。

如果是响应式数据的话，那么它与副作用函数之间就会建立联系。

屏蔽原型的更新（当一个child里面没有相关属性要给他设置，就会触发它的原型获得到parent发现parent有这个属性，这样的话他们两个都对这个进行了依赖收集，当触发的时候就会执行两次）。

解决方案：判断receiver是target的代理对象的时才能触发

```
target === receiver.raw
```

浅响应和深响应

isShallow，代表是否为浅响应，默认为 false，即非浅响应

```
01 // 封装 createReactive 函数，接收一个参数 isShallow，代表是否为浅响应，默认为 false，即非浅响应
02 function createReactive(obj, isShallow = false) {
03   return new Proxy(obj, {
04     // 拦截读取操作
05     get(target, key, receiver) {
06       if (key === 'raw') {
07         return target
08       }
```

```
09
10        const res = Reflect.get(target, key, receiver)
11
12        track(target, key)
13
14        // 如果是浅响应，则直接返回原始值
15        if (isShallow) {
16          return res
17        }
18
19        if (typeof res === 'object' && res !== null) {
20          return reactive(res)
21        }
22
23        return res
24      }
25      // 省略其他拦截函数
26    })
27 }
```

只读和浅只读

就在set和deleteProperty设置警告并返回

```
```

原始值的响应方案

第三篇渲染器

第7章 渲染器设计

1. 将虚拟DOM转化为真实DOM，diff算法

```
01 function createRenderer() {

 function mountElement(vnode, container) {
02   // 调用 createElement 函数创建元素
03   const el = createElement(vnode.type)
04   if (typeof vnode.children === 'string') {
05     // 调用 setElementText 设置元素的文本节点
06     setElementText(el, vnode.children)
07   }
08   // 调用 insert 函数将元素插入到容器内
09   insert(el, container)
10 }

02    function patch(n1, n2, container) {
02   // 如果 n1 不存在，意味着挂载，则调用 mountElement 函数完成挂载
03   if (!n1) {
```

```
    mountElement(n2, container)
  } else {
    // n1 存在，意味着打补丁，暂时省略
  }
}

  function render(vnode, container) {
    if (vnode) {
      patch(container._vnode, vnode, container)
    } else {
      if (container._vnode) {
        container.innerHTML = ''
      }
    }
    container._vnode = vnode
  }

  return {
    render
  }
}

// 在创建 renderer 时传入配置项
const renderer = createRenderer({
  // 用于创建元素
  createElement(tag) {
    return document.createElement(tag)
  },
  // 用于设置元素的文本节点
  setElementText(el, text) {
    el.textContent = text
  },
  // 用于在给定的 parent 下添加指定元素
  insert(el, parent, anchor = null) {
    parent.insertBefore(el, anchor)
  }
})
```

2. 事件处理函数 + 事件冒泡

```
patchProps(el, key, prevValue, nextValue) {
  if (/^on/.test(key)) {
    const invokers = el._vei || (el._vei = {})
    let invoker = invokers[key]
    const name = key.slice(2).toLowerCase()
    if (nextValue) {
      if (!invoker) {
        invoker = el._vei[key] = (e) => {
          // e.timeStamp 是事件发生的时间
          // 如果事件发生的时间早于事件处理函数绑定的时间，则不执行事件处理函数
          if (e.timeStamp < invoker.attached) return
          if (Array.isArray(invoker.value)) {
            invoker.value.forEach(fn => fn(e))
          } else {
```

```
            invoker.value(e)
          }
        }
        invoker.value = nextValue
        // 添加 invoker.attached 属性，存储事件处理函数被绑定的时间
        invoker.attached = performance.now()
        el.addEventListener(name, invoker)
      } else {
        invoker.value = nextValue
      }
    } else if (invoker) {
      el.removeEventListener(name, invoker)
    }
  } else if (key === 'class') {
    // 省略部分代码
  } else if (shouldSetAsProps(el, key, nextValue)) {
    // 省略部分代码
  } else {
    // 省略部分代码
  }
}
```

其中做了一层优化，就是将事件处理函数放到了invoker.value中，当更新事件处理函数的时候就不需要进行removeListener了

首先，我们为伪造的事件处理函数添加了 invoker.attached 属性，用来存储事件处理函数被绑定的时间。然后，在 invoker 执行的时候，通过事件对象的 e.timeStamp 获取事件发生的时间。最后，比较两者，如果事件处理函数被绑定的时间晚于事件发生的时间，则不执行该事件处理函数。

3. 更新子节点

 7fcda6d4318866e746621effdd3addd

```
// 没有子节点
vnode = {
  type: 'div',
  children: null
}
// 文本子节点
vnode = {
  type: 'div',
  children: 'Some Text'
}
// 其他情况，子节点使用数组表示
vnode = {
  type: 'div',
  children: [
    { type: 'p' },
    'Some Text'
  ]
}


//打补丁
function patchElement(n1, n2) {
  const el = n2.el = n1.el
```

```
  const oldProps = n1.props
  const newProps = n2.props
  // 第一步：更新 props
  for (const key in newProps) {
    if (newProps[key] !== oldProps[key]) {
      patchProps(el, key, oldProps[key], newProps[key])
    }
  }
  for (const key in oldProps) {
    if (!(key in newProps)) {
      patchProps(el, key, oldProps[key], null)
    }
  }

  // 第二步：更新 children
  patchChildren(n1, n2, el)
}
```

//patchChildren有　　没有子节点，有文本节点，有数组子节点,如上图，这里面有核心diff算法，就是如果旧节点和新节点都是数组的话那么就是可以进行diff算法。

```
function patchChildren(n1, n2, container) {
  // 判断新子节点的类型是否是文本节点
  if (typeof n2.children === 'string') {
    // 旧子节点的类型有三种可能：没有子节点、文本子节点以及一组子节点
    // 只有当旧子节点为一组子节点时，才需要逐个卸载，其他情况下什么都不需要做
    if (Array.isArray(n1.children)) {
      n1.children.forEach((c) => unmount(c))
    }
    // 最后将新的文本节点内容设置给容器元素
    setElementText(container, n2.children)
  }else if (Array.isArray(n2.children)) {
    // 说明新子节点是一组子节点

    // 判断旧子节点是否也是一组子节点
    if (Array.isArray(n1.children)) {
      // 代码运行到这里，则说明新旧子节点都是一组子节点，这里涉及核心的 Diff 算法
    } else {
      // 此时：
      // 旧子节点要么是文本子节点，要么不存在
      // 但无论哪种情况，我们都只需要将容器清空，然后将新的一组子节点逐个挂载
      setElementText(container, '')
      n2.children.forEach(c => patch(null, c, container))
    }else {
    // 代码运行到这里，说明新子节点不存在
    // 旧子节点是一组子节点，只需逐个卸载即可
    if (Array.isArray(n1.children)) {
      n1.children.forEach(c => unmount(c))
    } else if (typeof n1.children === 'string') {
      // 旧子节点是文本子节点，清空内容即可
      setElementText(container, '')
    }
    // 如果也没有旧子节点，那么什么都不需要做
  }
}
```

4. 更新文本节点和注释节点

```
const renderer = createRenderer({
  createElement(tag) {
    // 省略部分代码
  },
  setElementText(el, text) {
    // 省略部分代码
  },
  insert(el, parent, anchor = null) {
    // 省略部分代码
  },
  createText(text) {
    return document.createTextNode(text)
  },
  setText(el, text) {
    el.nodeValue = text
  },
  patchProps(el, key, prevValue, nextValue) {
    // 省略部分代码
  }
})
```

```
function patch(n1, n2, container) {
  if (n1 && n1.type !== n2.type) {
    unmount(n1)
    n1 = null
  }

  const { type } = n2

  if (typeof type === 'string') {
    if (!n1) {
      mountElement(n2, container)
    } else {
      patchElement(n1, n2)
    }
  } else if (type === Text) {
    if (!n1) {
      // 调用 createText 函数创建文本节点
      const el = n2.el = createText(n2.children)
      insert(el, container)
    } else {
      const el = n2.el = n1.el
      if (n2.children !== n1.children) {
        // 调用 setText 函数更新文本节点的内容
        setText(el, n2.children)
      }
    }
  }else if (type === Fragment) { // 处理 Fragment 类型的 vnode
    if (!n1) {
      // 如果旧 vnode 不存在，则只需要将 Fragment 的 children 逐个挂载即可
      n2.children.forEach(c => patch(null, c, container))
    } else {
      // 如果旧 vnode 存在，则只需要更新 Fragment 的 children 即可
```

```
      patchChildren(n1, n2, container)
    }
  }
}
}

//卸载的话卸载Fragment
function unmount(vnode) {
  // 在卸载时，如果卸载的 vnode 类型为 Fragment，则需要卸载其 children
  if (vnode.type === Fragment) {
    vnode.children.forEach(c => unmount(c))
    return
  }
  const parent = vnode.el.parentNode
  if (parent) {
    parent.removeChild(vnode.el)
  }
}
```

5. 添加属性

因为form的属性是只读的所以使用setAttribute

```
function shouldSetAsProps(el, key, value) {
  // 特殊处理
  if (key === 'form' && el.tagName === 'INPUT') return false
  // 兜底
  return key in el
}

function mountElement(vnode, container) {
  const el = createElement(vnode.type)
  // 省略 children 的处理

  if (vnode.props) {
    for (const key in vnode.props) {
      const value = vnode.props[key]
      // 使用 shouldSetAsProps 函数判断是否应该作为 DOM Properties 设置
      if (shouldSetAsProps(el, key, value)) {
        const type = typeof el[key]
        if (type === 'boolean' && value === '') {
          el[key] = true
        } else {
          el[key] = value
        }
      } else {
        el.setAttribute(key, value)
      }
    }
  }

  insert(el, container)
}

//抽离出来patchProps
```

```
const renderer = createRenderer({
  createElement(tag) {
    return document.createElement(tag)
  },
  setElementText(el, text) {
    el.textContent = text
  },
  insert(el, parent, anchor = null) {
    parent.insertBefore(el, anchor)
  },
  // 将属性设置相关操作封装到 patchProps 函数中，并作为渲染器选项传递
  patchProps(el, key, prevValue, nextValue) {
    if (shouldSetAsProps(el, key, nextValue)) {
      const type = typeof el[key]
      if (type === 'boolean' && nextValue === '') {
        el[key] = true
      } else {
        el[key] = nextValue
      }
    } else {
      el.setAttribute(key, nextValue)
    }
  }
})
```

```
function mountElement(vnode, container) {
  const el = createElement(vnode.type)
  if (typeof vnode.children === 'string') {
    setElementText(el, vnode.children)
  } else if (Array.isArray(vnode.children)) {
    vnode.children.forEach(child => {
      patch(null, child, el)
    })
  }

  if (vnode.props) {
    for (const key in vnode.props) {
      // 调用 patchProps 函数即可
      patchProps(el, key, null, vnode.props[key])
    }
  }

  insert(el, container)
}
```

6. class特殊处理

将class进行转化为字符串，封装一个normalizeClass函数

```
const renderer = createRenderer({
  // 省略其他选项

  patchProps(el, key, prevValue, nextValue) {
    // 对 class 进行特殊处理
```

```
    if (key === 'class') {
      el.className = nextValue || ''
    } else if (shouldSetAsProps(el, key, nextValue)) {
      const type = typeof el[key]
      if (type === 'boolean' && nextValue === '') {
        el[key] = true
      } else {
        el[key] = nextValue
      }
    } else {
      el.setAttribute(key, nextValue)
    }
  }
})
```

第9章简单diff算法

基本的优化

首先的思路：遍历新节点和旧节点，查看他们的长度选择最小的进行遍历将新的内容打补丁到旧的内容上，如果遍历完之后newLength > oldLength说明需要挂在新节点，如果反过来就是要卸载旧节点。但是接着还能优化

DOM复用与key的作用

因为发现上面的思想还是可以优化的，就是出现了那种只是顺序发生了改变其他内容没有改变的话，可以通过调换位置来进行优化，而不必要进行patch或者重新挂载了（通过DOM移动比不断地执行子节点的卸载和挂载性能更好）。

思考怎样确定新的子节点是否出现在旧的子节点呢?

单纯通过vnode.type是不够的还需要借助key来实现，key就相当于虚拟节点的身份证，只有这两个相同，那么就认为他们是相同的。

key的重要性

如果没有 key，我们无法知道新子节点与旧子节点间的映射关系，也就无法知道应该如何移动节点。有 key 的话情况则不同，我们根据子节点的 key 属性，能够明确知道新子节点在旧子节点中的位置，这样就可以进行相应的 DOM 移动操作了。vue就会采用就地更新的模式

代码

```
function patchChildren(n1, n2, container) {
  if (typeof n2.children === 'string') {
    // 省略部分代码
  } else if (Array.isArray(n2.children)) {
    const oldChildren = n1.children
    const newChildren = n2.children

    // 遍历新的 children
    for (let i = 0; i < newChildren.length; i++) {
      const newVNode = newChildren[i]
      // 遍历旧的 children
```

```
      for (let j = 0; j < oldChildren.length; j++) {
        const oldVNode = oldChildren[j]
        // 如果找到了具有相同 key 值的两个节点，说明可以复用，但仍然需要调用 patch 函数更新
        if (newVNode.key === oldVNode.key) {
          patch(oldVNode, newVNode, container)
          break // 这里需要 break
        }
      }
    }

  } else {
    // 省略部分代码
  }
}
```

找到需要移动的元素

其实我们可以将节点 p-3 在旧 children 中的索引定义为：**在旧 children 中寻找具有相同 key 值节点的过程中，遇到的最大索引值。如果在后续寻找的过程中，存在索引值比当前遇到的最大索引值还要小的节点，则意味着该节点需要移动。**

```
function patchChildren(n1, n2, container) {
  if (typeof n2.children === 'string') {
    // 省略部分代码
  } else if (Array.isArray(n2.children)) {
    const oldChildren = n1.children
    const newChildren = n2.children

    // 用来存储寻找过程中遇到的最大索引值
    let lastIndex = 0
    for (let i = 0; i < newChildren.length; i++) {
      const newVNode = newChildren[i]
      for (let j = 0; j < oldChildren.length; j++) {
        const oldVNode = oldChildren[j]
        if (newVNode.key === oldVNode.key) {
          patch(oldVNode, newVNode, container)
          if (j < lastIndex) {
            // 如果当前找到的节点在旧 children 中的索引小于最大索引值 lastIndex，
            // 说明该节点对应的真实 DOM 需要移动
          } else {
            // 如果当前找到的节点在旧 children 中的索引不小于最大索引值，
            // 则更新 lastIndex 的值
            lastIndex = j
          }
          break // 这里需要 break
        }
      }
    }

  } else {
    // 省略部分代码
  }
}
```

移动元素

```
function patchChildren(n1, n2, container) {
  if (typeof n2.children === 'string') {
    // 省略部分代码
  } else if (Array.isArray(n2.children)) {
    const oldChildren = n1.children
    const newChildren = n2.children

    let lastIndex = 0
    for (let i = 0; i < newChildren.length; i++) {
      const newVNode = newChildren[i]
      let j = 0
      for (j; j < oldChildren.length; j++) {
        const oldVNode = oldChildren[j]
        if (newVNode.key === oldVNode.key) {
          patch(oldVNode, newVNode, container)
          if (j < lastIndex) {
            // 代码运行到这里，说明 newVNode 对应的真实 DOM 需要移动
            // 先获取 newVNode 的前一个 vnode，即 prevVNode
            const prevVNode = newChildren[i - 1]
            // 如果 prevVNode 不存在，则说明当前 newVNode 是第一个节点，它不需要移动
            if (prevVNode) {
              // 由于我们要将 newVNode 对应的真实 DOM 移动到 prevVNode 所对应真实
DOM 后面，
              // 所以我们需要获取 prevVNode 所对应真实 DOM 的下一个兄弟节点，并将其作为
锚点
              const anchor = prevVNode.el.nextSibling
              // 调用 insert 方法将 newVNode 对应的真实 DOM 插入到锚点元素前面，
              // 也就是 prevVNode 对应真实 DOM 的后面
              insert(newVNode.el, container, anchor)
            }
          } else {
            lastIndex = j
          }
          break
        }
      }
    }

  } else {
    // 省略部分代码
  }
}
```

添加新元素

就是遍历新节点的时候在旧节点中没有找到，所以就找到了这个新增的元素。也就是定义一个符号(find)来找是否旧节点有这个元素，没有的话就进行插入。

代码

```
function patchChildren(n1, n2, container) {
  if (typeof n2.children === 'string') {
```

```
    // 省略部分代码
  } else if (Array.isArray(n2.children)) {
    const oldChildren = n1.children
    const newChildren = n2.children

    let lastIndex = 0
    for (let i = 0; i < newChildren.length; i++) {
      const newVNode = newChildren[i]
      let j = 0
      for (j; j < oldChildren.length; j++) {
        const oldVNode = oldChildren[j]
        if (newVNode.key === oldVNode.key) {
          patch(oldVNode, newVNode, container)
          if (j < lastIndex) {
            // 代码运行到这里，说明 newVNode 对应的真实 DOM 需要移动
            // 先获取 newVNode 的前一个 vnode，即 prevVNode
            const prevVNode = newChildren[i - 1]
            // 如果 prevVNode 不存在，则说明当前 newVNode 是第一个节点，它不需要移动
            if (prevVNode) {
              // 由于我们要将 newVNode 对应的真实 DOM 移动到 prevVNode 所对应真实 DOM 后面，
              // 所以我们需要获取 prevVNode 所对应真实 DOM 的下一个兄弟节点，并将其作为锚点
              const anchor = prevVNode.el.nextSibling
              // 调用 insert 方法将 newVNode 对应的真实 DOM 插入到锚点元素前面，
              // 也就是 prevVNode 对应真实 DOM 的后面
              insert(newVNode.el, container, anchor)
            }
          } else {
            lastIndex = j
          }
          break
        }
      }
    }

  } else {
    // 省略部分代码
  }
}
```

移除不存在的元素

上面的更新完了之后，就去遍历旧的节点，在新节点中寻找，如果没有就卸载。

```
function patchChildren(n1, n2, container) {
  if (typeof n2.children === 'string') {
    // 省略部分代码
  } else if (Array.isArray(n2.children)) {
    const oldChildren = n1.children
    const newChildren = n2.children

    let lastIndex = 0
    for (let i = 0; i < newChildren.length; i++) {
      // 省略部分代码
```

```
    }

    // 上一步的更新操作完成后
    // 遍历旧的一组子节点
    for (let i = 0; i < oldChildren.length; i++) {
      const oldVNode = oldChildren[i]
      // 拿旧子节点 oldVNode 去新的一组子节点中寻找具有相同 key 值的节点
      const has = newChildren.find(
        vnode => vnode.key === oldVNode.key
      )
      if (!has) {
        // 如果没有找到具有相同 key 值的节点，则说明需要删除该节点
        // 调用 unmount 函数将其卸载
        unmount(oldVNode)
      }
    }

  } else {
    // 省略部分代码
  }
}
```

第10章双端diff算法

双端diff的原理

是一种同时对新旧的两个端点进行比较的算法，所以需要四个索引值，分别指向新旧两组子节点的端点。

如果正常命中下面的四种情况，代码

```
while (oldStartIdx <= oldEndIdx && newStartIdx <= newEndIdx) {
  if (oldStartVNode.key === newStartVNode.key) {
    // 调用 patch 函数在 oldStartVNode 与 newStartVNode 之间打补丁
    patch(oldStartVNode, newStartVNode, container)
    // 更新相关索引，指向下一个位置
    oldStartVNode = oldChildren[++oldStartIdx]
    newStartVNode = newChildren[++newStartIdx]
  } else if (oldEndVNode.key === newEndVNode.key) {
    patch(oldEndVNode, newren[--oldEndIdx]
    newEndVNode = newChildren[--newEndIdx]
  } else if (oldStartVNode.key === newEndVNode.key) {
    patch(oldStartVNode, newEndVNode, container)
    insert(oldStartVNode.el, container, oldEndVNode.el.nextSibling)

    oldStartVNode = oldChildren[++oldStartIdx]
    newEndVNode = newChildren[--newEndIdx]
  } else if (oldEndVNode.key === newStartVNode.key) {
    patch(oldEndVNode, newStartVNode, container)
    insert(oldEndVNode.el, container, oldStartVNode.el)

```

```
    oldEndVNode = oldChildren[--oldEndIdx]
    newStartVNode = newChildren[++newStartIdx]
  }
}
```

非理想状况处理

四种情况都没有命中，只能额外处理：那新的一组子节点中的节点与旧的一组子结点中寻找。

```
while (oldStartIdx <= oldEndIdx && newStartIdx <= newEndIdx) {
  // 增加两个判断分支，如果头尾部节点为 undefined，则说明该节点已经被处理过了，直接跳到下一个位置
  if (!oldStartVNode) {
    oldStartVNode = oldChildren[++oldStartIdx]
  } else if (!oldEndVNode) {
    oldEndVNode = oldChildren[--oldEndIdx]
  } else if (oldStartVNode.key === newStartVNode.key) {
    // 省略部分代码
  } else if (oldEndVNode.key === newEndVNode.key) {
    // 省略部分代码
  } else if (oldStartVNode.key === newEndVNode.key) {
    // 省略部分代码
  } else if (oldEndVNode.key === newStartVNode.key) {
    // 省略部分代码
  } else {
    const idxInOld = oldChildren.findIndex(
      node => node.key === newStartVNode.key
    )
    if (idxInOld > 0) {
      const vnodeToMove = oldChildren[idxInOld]
      patch(vnodeToMove, newStartVNode, container)
      insert(vnodeToMove.el, container, oldStartVNode.el)
      oldChildren[idxInOld] = undefined
      newStartVNode = newChildren[++newStartIdx]
    }

  }
}
```

新增元素

```
while (oldStartIdx <= oldEndIdx && newStartIdx <= newEndIdx) {
  // 省略部分代码
}

// 循环结束后检查索引值的情况，
if (oldEndIdx < oldStartIdx && newStartIdx <= newEndIdx) {
  // 如果满足条件，则说明有新的节点遗留，需要挂载它们
  for (let i = newStartIdx; i <= newEndIdx; i++) {
    patch(null, newChildren[i], container, oldStartVNode.el)
  }
}
```

移除不存在的元素

```
while (oldStartIdx <= oldEndIdx && newStartIdx <= newEndIdx) {
  // 省略部分代码
}

if (oldEndIdx < oldStartIdx && newStartIdx <= newEndIdx) {
  // 添加新节点
  // 省略部分代码
} else if (newEndIdx < newStartIdx && oldStartIdx <= oldEndIdx) {
  // 移除操作
  for (let i = oldStartIdx; i <= oldEndIdx; i++) {
    unmount(oldChildren[i])
  }
}
```

第11章快速diff

相同前置元素和后置元素

快速diff包含预处理步骤，对于内容相同的问题就不需要进行核心的diff操作了。

就是设置一个索引，新节点和旧节点都用这个索引来遍历前置元素。设置newEnd和oldEnd来遍历他们的后置元素，反正过程中遇到相同的就进行打补丁。

还有就是结束情况:当oldEnd < j 说明旧节点遍历完了，newEnd >= j说明新增了节点就要把他们挂载。

当newEnd < j说明新节点遍历完了，oldEnd >= j说明旧节点有多余的元素需要进行卸载。

```
function patchKeyedChildren(n1, n2, container) {
  //1.首先先预处理。
  const newChildren = n2.children
  const oldChildren = n1.children
  // 更新相同的前置节点
  let j = 0
  let oldVNode = oldChildren[j]
  let newVNode = newChildren[j]
  while (oldVNode.key === newVNode.key) {
    patch(oldVNode, newVNode, container)
    j++
    oldVNode = oldChildren[j]
    newVNode = newChildren[j]
  }

  // 更新相同的后置节点
  // 索引 oldEnd 指向旧的一组子节点的最后一个节点
  let oldEnd = oldChildren.length - 1
  // 索引 newEnd 指向新的一组子节点的最后一个节点
  let newEnd = newChildren.length - 1

  oldVNode = oldChildren[oldEnd]
  newVNode = newChildren[newEnd]
```

```
23
24   // while 循环从后向前遍历，直到遇到拥有不同 key 值的节点为止
25   while (oldVNode.key === newVNode.key) {
26     // 调用 patch 函数进行更新
27     patch(oldVNode, newVNode, container)
28     // 递减 oldEnd 和 nextEnd
29     oldEnd--
30     newEnd--
31     oldVNode = oldChildren[oldEnd]
32     newVNode = newChildren[newEnd]
33   }
    //
10   // 预处理完毕后，如果满足如下条件，则说明从 j --> newEnd 之间的节点应作为新节点插入
11   if (j > oldEnd && j <= newEnd) {
12     // 锚点的索引
13     const anchorIndex = newEnd + 1
14     // 锚点元素
15     const anchor = anchorIndex < newChildren.length ?
newChildren[anchorIndex].el : null
16     // 采用 while 循环，调用 patch 函数逐个挂载新增节点
17     while (j <= newEnd) {
18       patch(null, newChildren[j++], container, anchor)
19     }
20   }

    if (j > oldEnd && j <= newEnd) {
11     // 省略部分代码
12   } else if (j > newEnd && j <= oldEnd) {
13     // j -> oldEnd 之间的节点应该被卸载
14     while (j <= oldEnd) {
15       unmount(oldChildren[j++])
16     }
17   }

34
35 }
```

判断是否需要进行DOM移动操作

先是进行填充一个source数组，通过一个索引表来记录新节点的位置，接着通过旧节点中找到相应的位置填充到source中

判断是否要进行移动和简单diff一样就是设置标记moved来看是否需要进行移动，还有一个标记就是pos来寻找最大索引，如果后面索引小于它的话那么这个元素就需要进行移动。

```
01 if (j > oldEnd && j <= newEnd) {
02   // 省略部分代码
03 } else if (j > newEnd && j <= oldEnd) {
04   // 省略部分代码
05 } else {
06   // 构造 source 数组
07   const count = newEnd - j + 1
08   const source = new Array(count)
```

```
  source.fill(-1)

  const oldStart = j
  const newStart = j
  let moved = false
  let pos = 0
  const keyIndex = {}
  for(let i = newStart; i <= newEnd; i++) {
    keyIndex[newChildren[i].key] = i
  }
  // 新增 patched 变量，代表更新过的节点数量
  let patched = 0
  for(let i = oldStart; i <= oldEnd; i++) {
    oldVNode = oldChildren[i]
    // 如果更新过的节点数量小于等于需要更新的节点数量，则执行更新
    if (patched <= count) {
      const k = keyIndex[oldVNode.key]
      if (typeof k !== 'undefined') {
        newVNode = newChildren[k]
        patch(oldVNode, newVNode, container)
        // 每更新一个节点，都将 patched 变量 +1
        patched++
        source[k - newStart] = i
        if (k < pos) {
          moved = true
        } else {
        // 没找到
        unmount(oldVNode)
      }
    } else {
      // 如果更新过的节点数量大于需要更新的节点数量，则卸载多余的节点
      unmount(oldVNode)
    }
  }
}
```

如何移动元素

采用最长递增子序列的方法来进行移动元素，在最长递增子序列中的不需要进行移动。

设置两个索引一个索引s指向最长子序列的最后一个元素，一个索引i指向新元素的最后一个元素接着就进行判断如果i === seq[s]那么不需要移动i--,如果等于-1，需要进行挂载，如果不等于i === seq[s]，那么就需要移动

```
if (moved) {
  const seq = lis(sources)

  // s 指向最长递增子序列的最后一个元素
  let s = seq.length - 1
  // i 指向新的一组子节点的最后一个元素
  let i = count - 1
  // for 循环使得 i 递减，即按照图 11-24 中箭头的方向移动
  for (i; i >= 0; i--) {
    if (source[i] === -1) {
      // 说明索引为 i 的节点是全新的节点，应该将其挂载
```

```
      // 该节点在新 children 中的真实位置索引
      const pos = i + newStart
      const newVNode = newChildren[pos]
      // 该节点的下一个节点的位置索引
      const nextPos = pos + 1
      // 锚点
      const anchor = nextPos < newChildren.length
        ? newChildren[nextPos].el
        : null
      // 挂载
      patch(null, newVNode, container, anchor)
    } else if (i !== seq[s]) {
      // 如果节点的索引 i 不等于 seq[s] 的值，说明该节点需要移动
     // 说明该节点需要移动
      // 该节点在新的一组子节点中的真实位置索引
      const pos = i + newStart
      const newVNode = newChildren[pos]
      // 该节点的下一个节点的位置索引
      const nextPos = pos + 1
      // 锚点
      const anchor = nextPos < newChildren.length
        ? newChildren[nextPos].el
        : null
      // 移动
      insert(newVNode.el, container, anchor)
    } else {
      // 当 i === seq[s] 时，说明该位置的节点不需要移动
      // 只需要让 s 指向下一个位置
      s--
    }
  }
}
```

第四篇组件化

第14章Teleport

这个组件的作用主要用来将模板内的 DOM 元素移动到其他位置。

Teleport从渲染器中抽离出来了

- 可以避免渲染器逻辑代码“膨胀”；
- 当用户没有使用 Teleport 组件时，由于 Teleport 的渲染逻辑被分离，因此可以利用 TreeShaking 机制在最终的 bundle 中删除 Teleport 相关的代码，使得最终构建包的体积变小。

在patch函数中调用process将控制权交接出去。

通过判断旧的虚拟节点（n1）是否存在，来决定是执行挂载还是执行更新。如果要执行挂载，则需要根据 props.to 属性的值来取得真正的挂载点。最后，遍历 Teleport 组件的 children 属性，并逐一调用 patch 函数完成子节点的挂载。

```
const Teleport = {
  __isTeleport: true,
  process(n1, n2, container, anchor, internals) {
    // 通过 internals 参数取得渲染器的内部方法
    const { patch } = internals
```

```
    // 如果旧 VNode n1 不存在，则是全新的挂载，否则执行更新
    if (!n1) {
      // 挂载
      // 获取容器，即挂载点
      const target = typeof n2.props.to === 'string'
        ? document.querySelector(n2.props.to)
        : n2.props.to
      // 将 n2.children 渲染到指定挂载点即可
      n2.children.forEach(c => patch(null, c, target, anchor))
    } else {
      // 更新
      patchChildren(n1, n2, container)
      // 如果新旧 to 参数的值不同，则需要对内容进行移动
      if (n2.props.to !== n1.props.to) {
        // 获取新的容器
        const newTarget = typeof n2.props.to === 'string'
          ? document.querySelector(n2.props.to)
          : n2.props.to
        // 移动到新的容器
        n2.children.forEach(c => move(c, newTarget))
      }
    }
  }
}
```