

# 高效更新:引入虚拟 DOM

前面代码只是完成了更新功能,但是效率极其低下,因为每次都需要全量更新。想要按需更新,就要知道视图中变化的地方在哪,这就需要引入虚拟DOM,每次更新时通过patch 比较一下,找到变化的地方。

下面咱们开始吧!

# 创建 VNode

编写一个 VNode 创建函数

```
JavaScript
// src/mini-vue/runtime-core/vnode.js
export function createVNode(type, props, children) {
  return { type, props, children };
}
```

修改 demo 中组件 render 函数,返回一个 VNode

```
// src/main.js
render() {
    // const h3 = document.createElement('h3')
    // h3.textContent = this.title
    // return h3
    // 返回VNode
    return createVNode('h3', {}, this.title)
},
```

高效更新: 引入虚拟DOM - 1

# 实现 patch 中创建部分

下面修改 render 函数并编写全新的 patch 函数用于转换传入的 VNode 为 dom。

这里分两部分: mount 和 patch, 先完成创建部分

```
JavaScript

// src/mini-vue/runtime-core/index.js

const render = (vnode, container) => {
    // 如果存在vnode则为mount或patch, 否则为unmount, 此处忽略
    if (vnode) {
        patch(container._vnode || null, vnode, container)
    }
    // 保存本次结果下次patch时作为旧节点
    container._vnode = vnode
}

const patch = (n1, n2, container) => {
    // 如果type为字符串说明是native element, 否则是组件
    const { type } = n2;
    if (typeof type === 'string') {
        // element
        processElement(n1, n2, container)
```

```
} else {
    // component
    processComponent(n1, n2, container)
}
```

#### 下面我们先处理根组件:

```
//src/mini-vue/runtime-core/index.js

const processComponent = (n1, n2, container) => {
    // 初始化时没有旧节点n1, 执行mount
    if (n1 == null) {
        // mount
        mountComponent(n2, container)
    } else {
        // patch
    }
}
```

#### 挂载组件做三件事:

- 组件实例化
- 状态初始化
- 安装渲染函数副作用

```
// src/mini-vue/runtime-core/index.js

const mountComponent = (initialVNode, container) => {
    // 创建组件实例
    const instance = {
        data: {},
        vnode: initialVNode,
        isMounted: false,
    };

// 初始化组件状态
    const { data: dataOptions } = instance.vnode.type;
```

```
instance.data = reactive(dataOptions());

// 安装渲染副作用
setupRenderEffect(instance, container);
};
```

#### 安装副作用的任务是首次渲染和建立更新机制

```
JavaScript
// src/mini-vue/runtime-core/index.js
// 执行组件首次渲染,并将更新函数设置为副作用
const setupRenderEffect = (instance, container) => {
 // 声明组件更新函数
 const componentUpdateFn = () => {
   if (!instance.isMounted) {
     // 创建阶段
     // 渲染获取组件视图VNode
     const { render } = instance.vnode.type;
     const vnode = render.call(instance.data);
     // 递归patch嵌套节点
     patch(null, vnode, container);
     // 调用钩子函数
     if (instance.vnode.type.mounted) {
       instance.vnode.type.mounted.call(instance.data);
     }
     instance.isMounted = true;
   } else {
     // 更新阶段
   }
 };
 // 设置副作用
 effect(componentUpdateFn);
 // 首次执行更新函数
 componentUpdateFn();
};
```

#### 下次递归调用 patch 时,按照 demo 中的情况就会处理 element 了

```
// src/mini-vue/runtime-core/index.js
// 处理native元素
const processElement = (n1, n2, container) => {
  if (n1 == null) {
    // 创建阶段
    mountElement(n2, container);
  } else {
    // 更新阶段
  }
};
```

#### 在 mountElement 中我们做 dom 创建

```
JavaScript
// src/mini-vue/runtime-core/index.js
// 解构各种dom操作
const {
  createElement: hostCreateElement, // createElement需要添加
 insert: hostInsert,
} = options;
// 创建元素
const mountElement = (vnode, container) => {
 // 创建元素
  const el = (vnode.el = hostCreateElement(vnode.type));
 // children为文本
  if (typeof vnode.children === "string") {
   el.textContent = vnode.children;
 } else {
   // children为数组需递归
   vnode.children.forEach((child) => patch(null, child, el));
 }
 // 插入元素
 hostInsert(el, container);
};
```

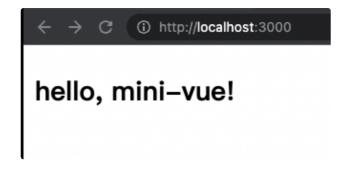
## 添加 createElement 操作

```
JavaScript

// src/mini-vue/runtime-dom/index.js

const rendererOptions = {
    createElement(tag) {
      return document.createElement(tag)
    }
};
```

## 预览一下效果



高效更新:引入虚拟DOM - 6