高效更新：patch 算法

小羊们好，前面基于 VNode 实现了视图创建，下面继续实现视图更新的部分。

更新流程

响应式数据变化会再次调用 `componentUpdateFn`，这次进入更新阶段，再次进入 patch，不过会同时传入 prevVnode, nextVnode：

```JavaScript
const componentUpdateFn = (instance.update = () => {
 // 渲染函数提出来
 const { render } = instance.vnode.type;

 if (!instance.isMounted) {
   // 存一下首次VNode
   const vnode = (instance.subtree = render.call(instance.data));
   // ...
 } else {
   // 更新阶段
   // 从实例获取上次结果
   const prevVnode = instance.subtree;
   // 重新获取最新结果
   const nextVnode = render.call(instance.data);
   // 保存下次更新使用
   instance.subtree = nextVnode;
   patch(prevVnode, nextVnode);
 }
});
```

demo 中会进入 processElement，进而调用更新函数 `patchElement`：

```JavaScript
// 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
    patchElement(n1, n2)
  }
};
```

`patchElement` 根据节点 children 情况做对应操作：

JavaScript

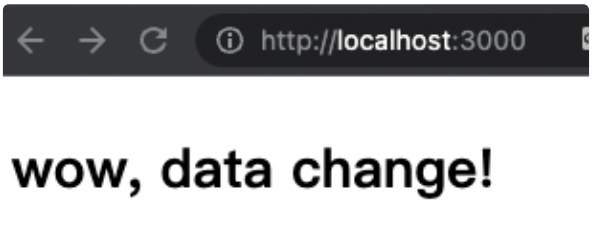
```
const {
  setElementText: hostSetElementText
} = options;

const patchElement = (n1, n2) => {
    // 获取要操作元素
    const el = n2.el = n1.el

    // 更新type相同节点，实际上还要考虑key
    if(n1.type === n2.type) {
      const oldCh = n1.children
      const newCh = n2.children

      if (typeof oldCh === 'string') {
        if (typeof newCh === 'string') {
          // 对比双方都是文本内容
          if (oldCh !== newCh) {
            hostSetElementText(el, newCh)
          }
        } else {
          // 之前是文本内容，变化之后是子元素数组
        }
      } else {
        if (typeof newCh === 'string') {
          // 之前是子元素数组，变化之后是文本内容
        } else {
          // 变化前后都是子元素数组
        }
      }
    }
  }
```

现在可以测试一下，ok!



还有另外三种情况：

- 之前是文本内容，变化之后是子元素数组
- 之前是子元素数组，变化之后是文本内容
- 变化前后都是子元素数组

之前是文本内容，变化之后是子元素数组

```JavaScript
// 这种情况先清空文本，再循环创建子元素
hostSetElementText(el, '')
newCh.forEach(v => patch(null, v, el))
```

对应的测试用例：

```JavaScript
render() {
  // const h3 = document.createElement('h3')
  // h3.textContent = this.title
  // return h3
  // 返回VNode
  // return createVNode('h3', {}, this.title)
  // text=》array
  if (Array.isArray(this.title)) {
    return createVNode(
      "h3",
      {},
      this.title.map((t) => createVNode("p", {}, t))
```

```
    );
  } else {
    return createVNode("h3", {}, this.title);
  }
},
mounted() {
  setTimeout(() => {
    // 修改为数组
    this.title = ["wow,", "data change!"];
  }, 2000);
},
```

之前是子元素数组，变化之后是文本内容

```JavaScript
el.textContent = newCh
```

最后着重说都是子元素的情况，这里使用简单 diff 算法，比如：

old： A B C D E

new：A C D E

从左往右依次 patch，AA，BC，CD，DE，最后删除 E

```JavaScript
const updateChildren = (oldCh, newCh, parentElm) => {
    // 这里暂且直接patch对应索引的两个节点
    const len = Math.min(oldCh.length, newCh.length);
    for (let i = 0; i < len; i++) {
      patch(oldCh[i], newCh[i]);
    }
    // newCh若是更长的那个，说明有新增
    if (newCh.length > oldCh.length) {
      newCh.slice(len).forEach((child) => patch(null, child, parentElm));
    } else if (newCh.length < oldCh.length) {
      // oldCh若是更长的那个，说明有删减
      oldCh.slice(len).forEach((child) => hostRemove(child.el));
    }
  };
```

增加一个删除操作

JavaScript

```
// runtime-dom/index.js
remove: (child) => {
  const parent = child.parentNode;
  if (parent) {
    parent.removeChild(child);
  }
},
```

JavaScript

```
// runtime-core/index.js
const { remove: hostRemove } = options
```

调用

JavaScript

```
updateChildren(oldCh, newCh, el)
```