Vue源码探秘之

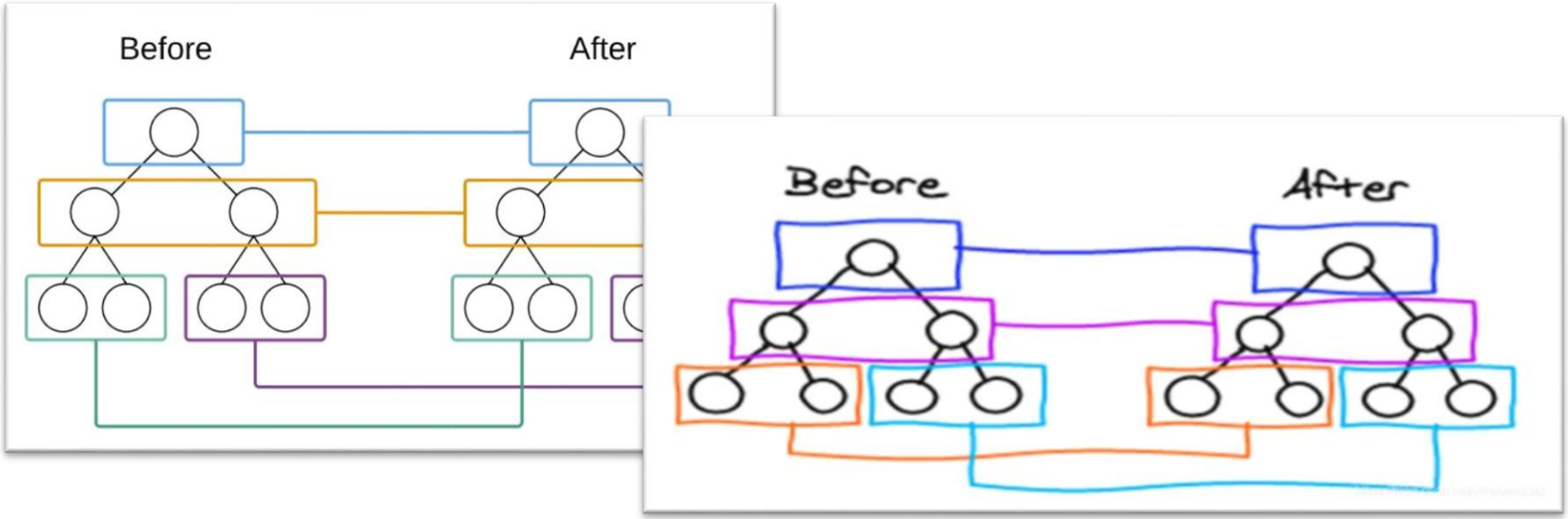
虚拟DOM和diff算法

讲师：邵山欢

课程简介

你可能粗浅了解过一些虚拟DOM和diff算法相关知识，能说出个大概……

你可能粗浅看过一些公众号文章，至少肯定见过下面的图片……



但！请扪心自问：**你到底懂不懂虚拟DOM和diff算法？？**

本课程的目的就是让大家：

真正的、彻底的弄懂虚拟DOM和diff算法

何为真正、彻底弄懂呢？

把它们的底层动手敲出来！

先简单介绍一下虚拟DOM和diff算法

变为

先简单介绍一下虚拟DOM和diff算法

拆了

重建

效率太低，代价昂贵

先简单介绍一下虚拟DOM和diff算法

diff

精细化比对
最小量更新

先简单介绍一下虚拟DOM和diff算法

```
<div class="box">
    <h3>我是一个标题</h3>
    <ul>
        <li>牛奶</li>
        <li>咖啡</li>
        <li>可乐</li>
    </ul>
</div>
```

变为

diff算法可以进行精细化比对，实现最小量更新

```
<div class="box">
    <h3>我是一个标题</h3>
    <span>我是一个新的span</span>
    <ul>
        <li>牛奶</li>
        <li>咖啡</li>
        <li>可乐</li>
        <li>雪碧</li>
    </ul>
</div>
```

先简单介绍一下虚拟DOM和diff算法

真实DOM

```html
<div class="box">
    <h3>我是一个标题</h3>
    <ul>
        <li>牛奶</li>
        <li>咖啡</li>
        <li>可乐</li>
    </ul>
</div>
```

虚拟DOM

```json
{
    "sel": "div",
    "data": {
        "class": { "box": true }
    },
    "children": [
        {
            "sel": "h3",
            "data": {},
            "text": "我是一个标题"
        },
        {
            "sel": "ul",
            "data": {},
            "children": [
                { "sel": "li", "data": {}, "text": "牛奶" },
                { "sel": "li", "data": {}, "text": "咖啡" },
                { "sel": "li", "data": {}, "text": "可乐" }
            ]
        }
    ]
}
```

diff是发生在虚拟DOM上的

新虚拟DOM和老虚拟DOM进行diff（精细化比较），算出应该如何最小量更新，最后反映到真正的DOM上。

```
{
    "sel": "div",
    "data": {
        "class": { "box": true }
    },
    "children": [
        {
            "sel": "h3",
            "text": "我是一个标题"
        },
        {
            "sel": "ul",
            "data": {},
            "children": [
                { "sel": "li", "text": "牛奶" },
                { "sel": "li", "text": "咖啡" },
                { "sel": "li", "text": "可乐" }
            ]
        }
    ]
}
```

diff

```
{
    "sel": "div",
    "data": {
        "class": { "box": true }
    },
    "children": [
        {
            "sel": "h3",
            "text": "我是一个标题"
        },
        {
            "sel": "span",
            "text": "我是一个新的span"
        },
        {
            "sel": "ul",
            "data": {},
            "children": [
                { "sel": "li", "text": "牛奶" },
                { "sel": "li", "text": "咖啡" },
                { "sel": "li", "text": "可乐" },
                { "sel": "li", "text": "雪碧" }
            ]
        }
    ]
}
```

- snabbdom简介
- snabbdom的h函数如何工作
- diff算法原理
- 手写diff算法

介绍宏观背景、历史沿革

先学会怎么用

再研究它底层机理

最后手写它，彻底掌握它！

循序渐进

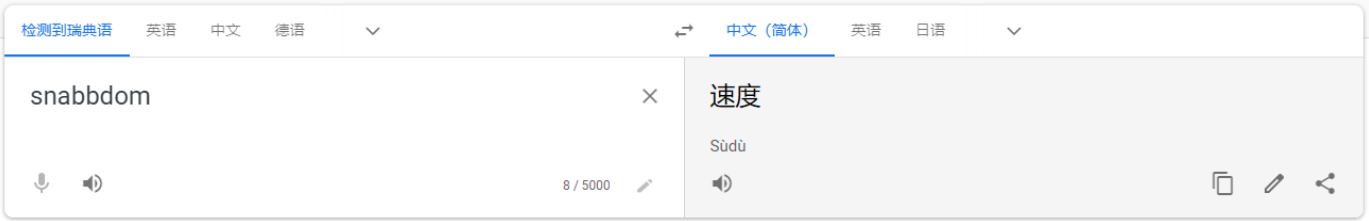
学习本课程的**知识储备前提**：

- 会Vue（Vue2.x和Vue3.x均可）
- 简单了解webpack和webpack-dev-server
- 简单了解TypeScript，我们会看TS版本源码，但是写JS代码，所以如果不了解TS也没关系，老师会讲解每行代码的意义
- 实际开发经验不限，应届生也可学习

snabbdom简介和测试环境搭建

snabbdom简介

- snabbdom是瑞典语单词，单词原意“速度”；



- snabbdom是著名的虚拟DOM库，是diff算法的鼻祖，Vue源码借鉴了snabbdom；
- 官方git：https://github.com/snabbdom/snabbdom

- 在git上的snabbdom源码是用TypeScript写的，git上并不提供编译好的JavaScript版本；
- 如果要直接使用build出来的JavaScript版的snabbdom库，可以从npm上下载：

```
npm i -D snabbdom
```

- 学习库底层时，建议大家阅读原汁原味的TS代码，最好带有库作者原注释，这样对你的源码阅读能力会有很大的提升。

- snabbdom库是DOM库，当然不能在nodejs环境运行，所以我们需要搭建webpack和webpack-dev-server开发环境，好消息是不需要安装任何loader
- 这里需要注意，必须安装最新版webpack@5，不能安装webpack@4，这是因为webpack4没有读取身份证中exports的能力，建议大家使用这样的版本：

```
npm i -D webpack@5 webpack-cli@3 webpack-dev-server@3
```

- 参考webpack官网，书写好webpack.config.js文件

- 跑通snabbdom官方git首页的demo程序，即证明调试环境已经搭建成功

  https://github.com/snabbdom/snabbdom

- 不要忘记在index.html中放置一个div#container

虚拟DOM和h函数

真实DOM

```html
<div class="box">
    <h3>我是一个标题</h3>
    <ul>
        <li>牛奶</li>
        <li>咖啡</li>
        <li>可乐</li>
    </ul>
</div>
```

虚拟DOM：用JavaScript对象描述DOM的层次结构。DOM中的一切属性都在虚拟DOM中有对应的属性。

虚拟DOM

```json
{
    "sel": "div",
    "data": {
        "class": { "box": true }
    },
    "children": [
        {
            "sel": "h3",
            "data": {},
            "text": "我是一个标题"
        },
        {
            "sel": "ul",
            "data": {},
            "children": [
                { "sel": "li", "data": {}, "text": "牛奶" },
                { "sel": "li", "data": {}, "text": "咖啡" },
                { "sel": "li", "data": {}, "text": "可乐" }
            ]
        }
    ]
}
```

diff是发生在虚拟DOM上的

新虚拟DOM和老虚拟DOM进行diff（精细化比较），算出应该如何最小量更新，最后反映到真正的DOM上。

```
{
    "sel": "div",
    "data": {
        "class": { "box": true }
    },
    "children": [
        {
            "sel": "h3",
            "text": "我是一个标题"
        },
        {
            "sel": "ul",
            "data": {},
            "children": [
                { "sel": "li", "text": "牛奶" },
                { "sel": "li", "text": "咖啡" },
                { "sel": "li", "text": "可乐" }
            ]
        }
    ]
}
```

diff

```
{
    "sel": "div",
    "data": {
        "class": { "box": true }
    },
    "children": [
        {
            "sel": "h3",
            "text": "我是一个标题"
        },
        {
            "sel": "span",
            "text": "我是一个新的span"
        },
        {
            "sel": "ul",
            "data": {},
            "children": [
                { "sel": "li", "text": "牛奶" },
                { "sel": "li", "text": "咖啡" },
                { "sel": "li", "text": "可乐" },
                { "sel": "li", "text": "雪碧" }
            ]
        }
    ]
}
```

本次课不研究DOM如何变为虚拟DOM

```html
<div class="box">
    <h3>我是一个标题</h3>
    <ul>
        <li>牛奶</li>
        <li>咖啡</li>
        <li>可乐</li>
    </ul>
</div>
```

如何变为？

```json
{
    "sel": "div",
    "data": {
        "class": { "box": true }
    },
    "children": [
        {
            "sel": "h3",
            "text": "我是一个标题"
        },
        {
            "sel": "ul",
            "data": {},
            "children": [
                { "sel": "li", "text": "牛奶" },
                { "sel": "li", "text": "咖啡" },
                { "sel": "li", "text": "可乐" }
            ]
        }
    ]
}
```

DOM如何变为虚拟DOM，属于模板编译原理范畴，本次课不研究

- **研究1**：虚拟DOM如何被渲染函数（h函数）产生？

  我们要手写h函数

- **研究2**：diff算法原理？

  我们要手写diff算法

- **研究3**：虚拟DOM如何通过diff变为真正的DOM的

  事实上，虚拟DOM变回真正的DOM，是涵盖在diff算法里面的

- h函数用来产生虚拟节点（vnode）

比如这样调用h函数：

```
h('a', { props: { href: 'http://www.atguigu.com' }}, '尚硅谷');
```

将得到这样的虚拟节点：

```
{ "sel": "a", "data": { props: { href: 'http://www.atguigu.com' } }, "text": "尚硅谷" }
```

它表示的真正的DOM节点：

```
<a href="http://www.atguigu.com">尚硅谷</a>
```

一个虚拟节点有哪些属性？


```
{
        children: undefined
        data: {}
        elm: undefined
        key: undefined
        sel: "div"
        text: "我是一个盒子"
}
```

h函数可以嵌套使用，从而得到虚拟DOM树（重要）

- 比如这样嵌套使用h函数：

```
h('ul', {}, [
    h('li', {}, '牛奶'),
    h('li', {}, '咖啡'),
    h('li', {}, '可乐')
]);
```

- 将得到这样的虚拟DOM树：

```
{
    "sel": "ul",
    "data": {},
    "children": [
        { "sel": "li", "text": "牛奶" },
        { "sel": "li", "text": "咖啡" },
        { "sel": "li", "text": "可乐" }
    ]
}
```

- 演示一下h函数的多种用法

```
const myVnode3 = h('ul', [
    h('li', {}, '苹果'),
    h('li', '西瓜'),
    h('li', [
        h('div', [
            h('p', '哈哈'),
            h('p', '嘻嘻')
        ])
    ]),
    h('li', h('p', '火龙果'))
]);
```

手写h函数

- 看源码的TS版代码，然后仿写JS代码
- 只要主干功能，放弃实现一些细节，保证能够**让大家理解核心是如何实现的**

感受diff算法

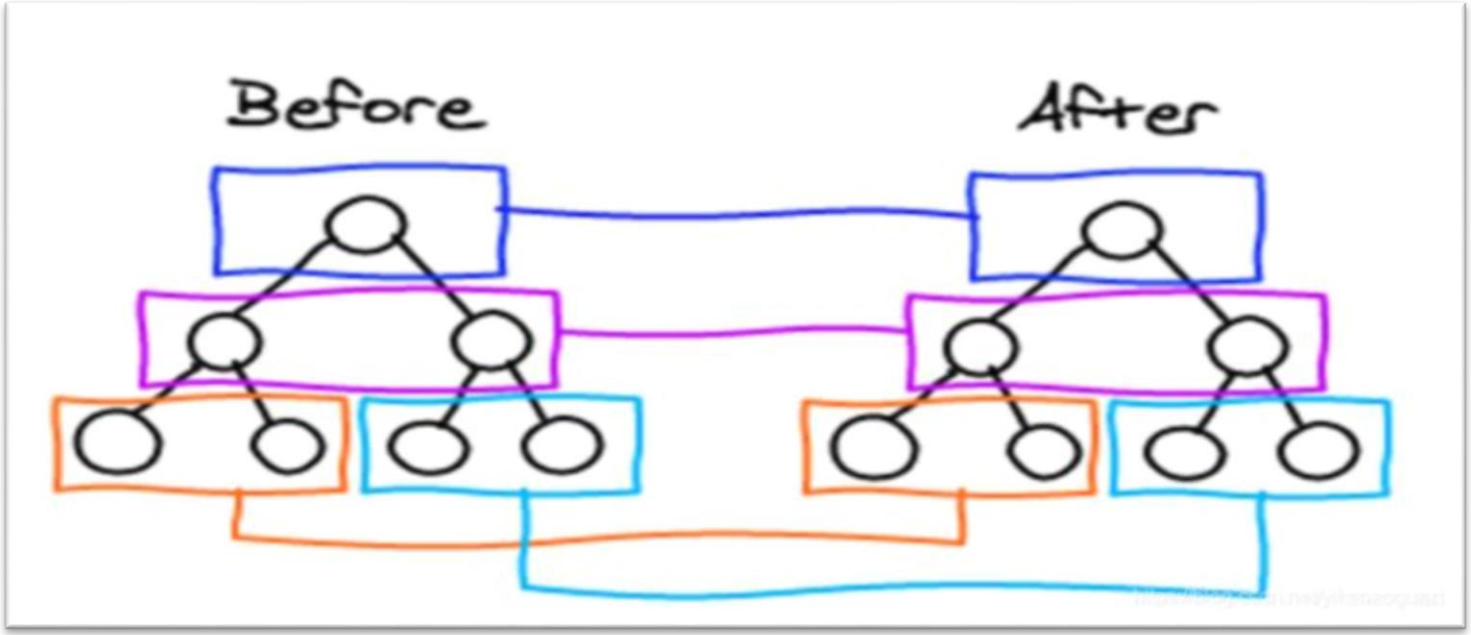
感受diff算法的心得

- 最小量更新太厉害啦！真的是最小量更新！**当然，key很重要。key是这个节点的唯一标识，告诉diff算法，在更改前后它们是同一个DOM节点。**
- **只有是同一个虚拟节点，才进行精细化比较**，否则就是暴力删除旧的、插入新的。
  延伸问题：如何定义是同一个虚拟节点？答：选择器相同且key相同。
- **只进行同层比较，不会进行跨层比较。**即使是同一片虚拟节点，但是跨层了，对不起，精细化比较不diff你，而是暴力删除旧的、然后插入新的。

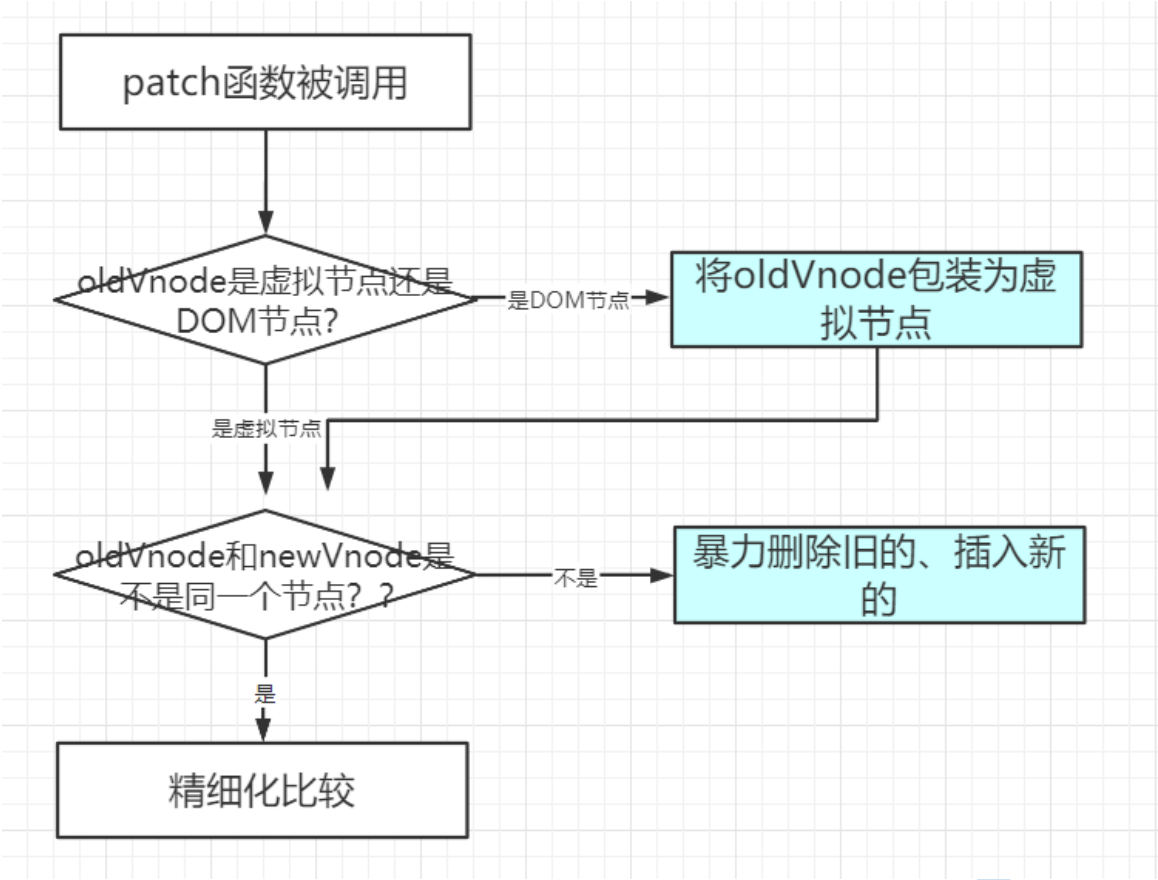
diff并不是那么的“无微不至”啊！真的影响效率么？？

答：上面2、3操作在实际Vue开发中，基本不会遇见，所以这是合理的优化机制。

同层比较示意图



diff处理新旧节点不是同一个节点时


patch函数被调用
oldVnode是虚拟节点还是DOM节点?
是DOM节点
将oldVnode包装为虚拟节点
是虚拟节点
oldVnode和newVnode是不是同一个节点？？
不是
暴力删除旧的、插入新的
是
精细化比较

如何定义“同一个节点”这个事儿


```
function sameVnode (vnode1: VNode, vnode2: VNode): boolean {
  return vnode1.key === vnode2.key && vnode1.sel === vnode2.sel
}
```

旧节点的key要和新节点的key相同
且
旧节点的选择器要和新节点的选择器相同

创建节点时，所有子节点需要递归创建的

```
if (is.array(children)) {
  for (i = 0; i < children.length; ++i) {
    const ch = children[i]
    if (ch != null) {
      api.appendChild(elm, createElm(ch as VNode, insertedVnodeQueue))
    }
  }
} else if (is.primitive(vnode.text)) {
  api.appendChild(elm, api.createTextNode(vnode.text))
}
```

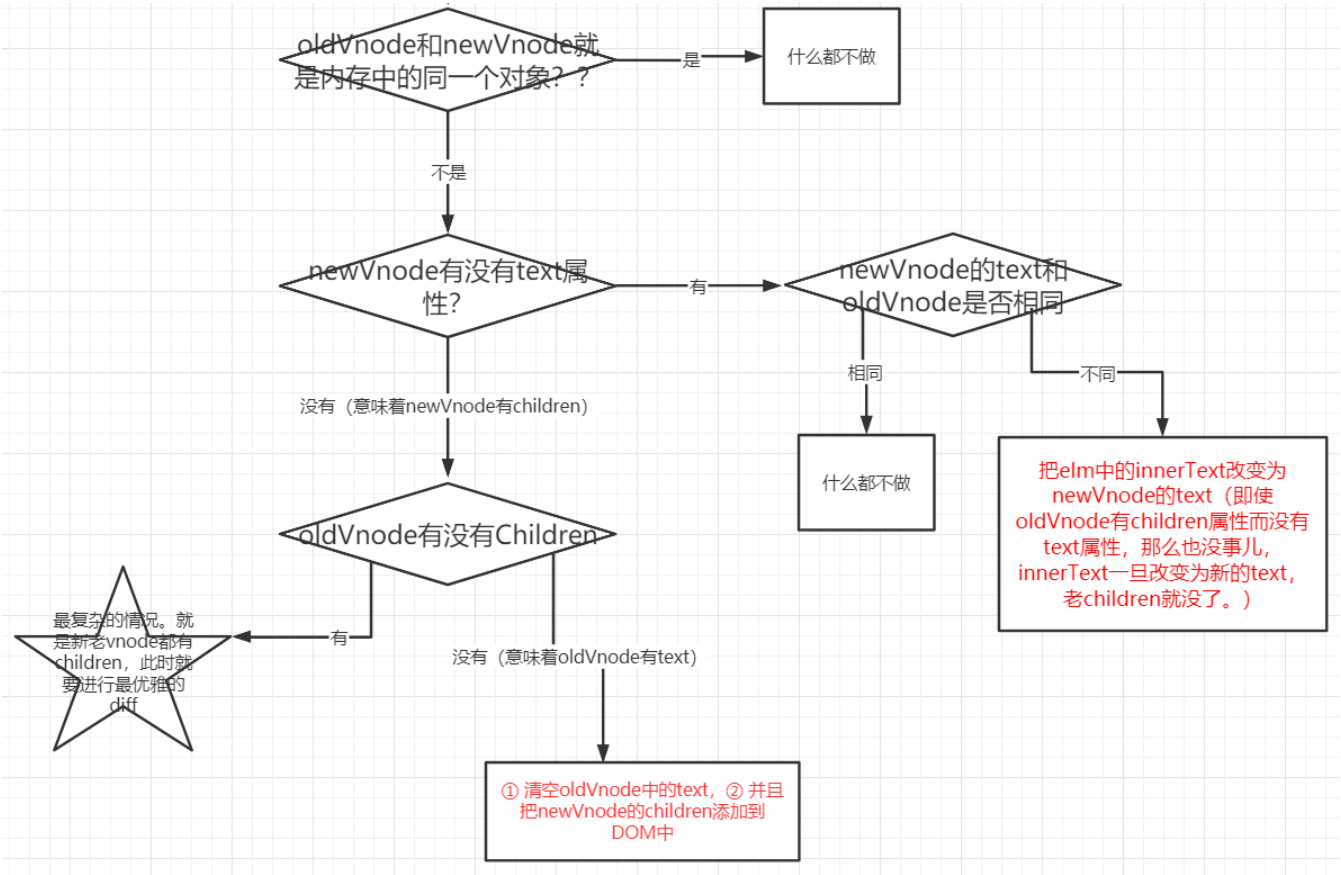
手写第一次上树时

手写递归创建子节点

```
// 真正创建节点。将vnode创建为DOM，是孤儿节点，不进行插入
export default function createElement(vnode) {
    console.log('目的是把虚拟节点', vnode, '真正变为DOM');
    // 创建一个DOM节点，这个节点现在还是孤儿节点
    let domNode = document.createElement(vnode.sel);
    // 有子节点还是有文本？？
    if (vnode.text != '' && (vnode.children == undefined || vnode.children.length == 0)) {
        // 它内部是文字
        domNode.innerText = vnode.text;
    } else if (Array.isArray(vnode.children) && vnode.children.length > 0) {
        // 它内部是子节点，就要递归创建节点
        for (let i = 0; i < vnode.children.length; i++) {
            // 得到当前这个children
            let ch = vnode.children[i];
            // 创建出它的DOM，一旦调用createElement意味着：创建出DOM了，并且它的elm属性指向了创建出的DOM，但是还
            没有上树，是一个孤儿节点。
            let chDOM = createElement(ch);
            // 上树
            domNode.appendChild(chDOM);
        }
    }
    // 补充elm属性
    vnode.elm = domNode;

    // 返回elm，elm属性是一个纯DOM对象
    return vnode.elm;
};
```

diff处理新旧节点是同一个节点时


oldVnode和newVnode就是内存中的同一个对象？？
是
什么都不做
不是
newVnode有没有text属性?
有
newVnode的text和oldVnode是否相同
相同
不同
什么都不做
把elm中的innerText改变为newVnode的text（即使oldVnode有children属性而没有text属性，那么也没事儿，innerText一旦改变为新的text，老children就没了。）
没有（意味着newVnode有children）
oldVnode有没有Children
有
最复杂的情况。就是新老vnode都有children，此时就要进行最优雅的diff
没有（意味着oldVnode有text）
① 清空oldVnode中的text，② 并且把newVnode的children添加到DOM中

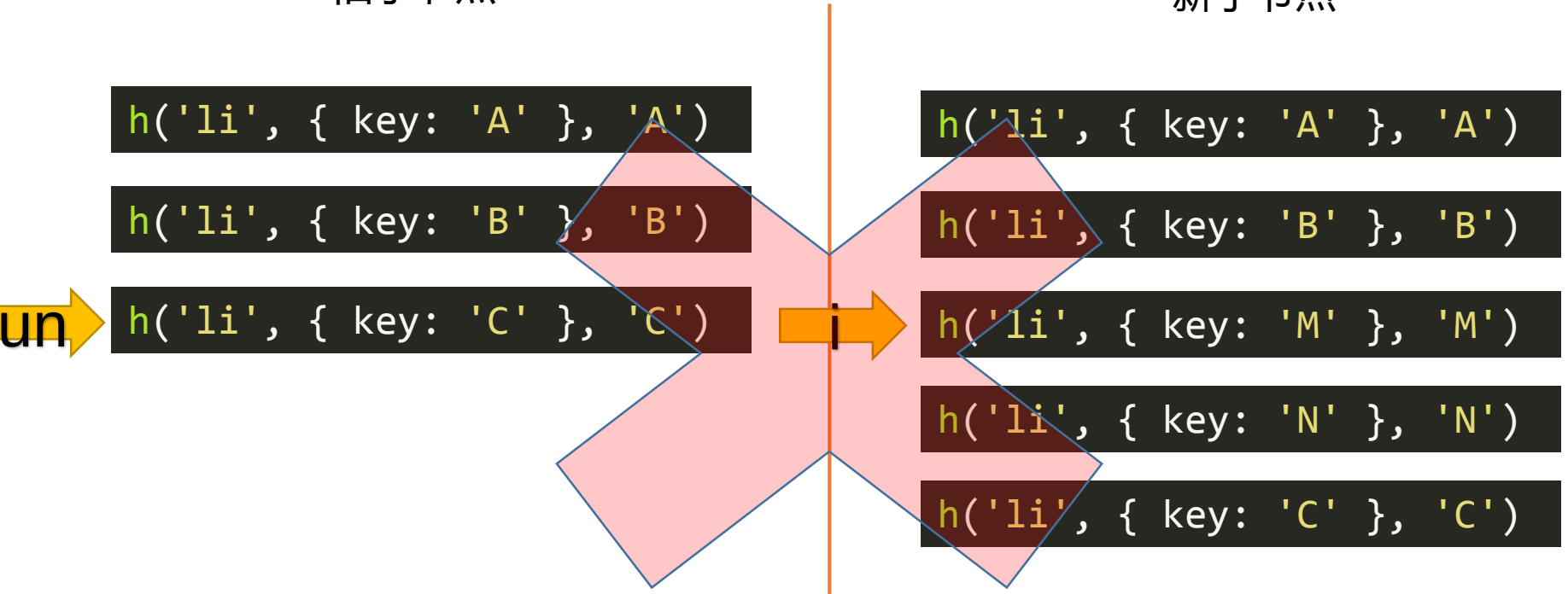
手写新旧节点text的不同情况

尝试书写diff更新子节点

新增的情况

旧子节点

新子节点



新创建的节点(newVnode.children[i].elm)插入到所有未处理的节点(oldVnode.children[um].elm)之前，而不是所有已处理节点之后

删除的情况

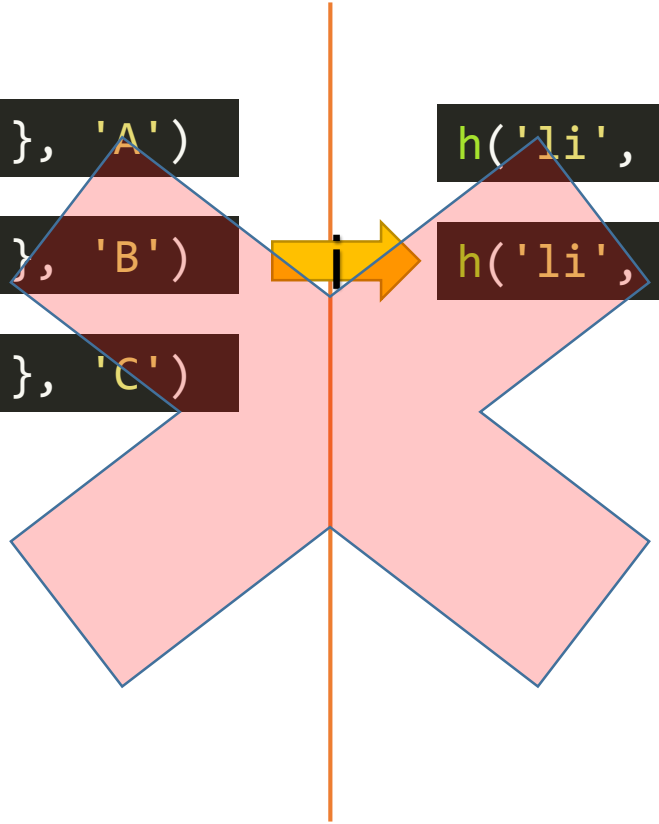
旧子节点

新子节点

```
h('li', { key: 'A' }, 'A')
h('li', { key: 'B' }, 'B')
h('li', { key: 'C' }, 'C')
```

un

```
h('li', { key: 'A' }, 'A')
h('li', { key: 'C' }, 'C')
```



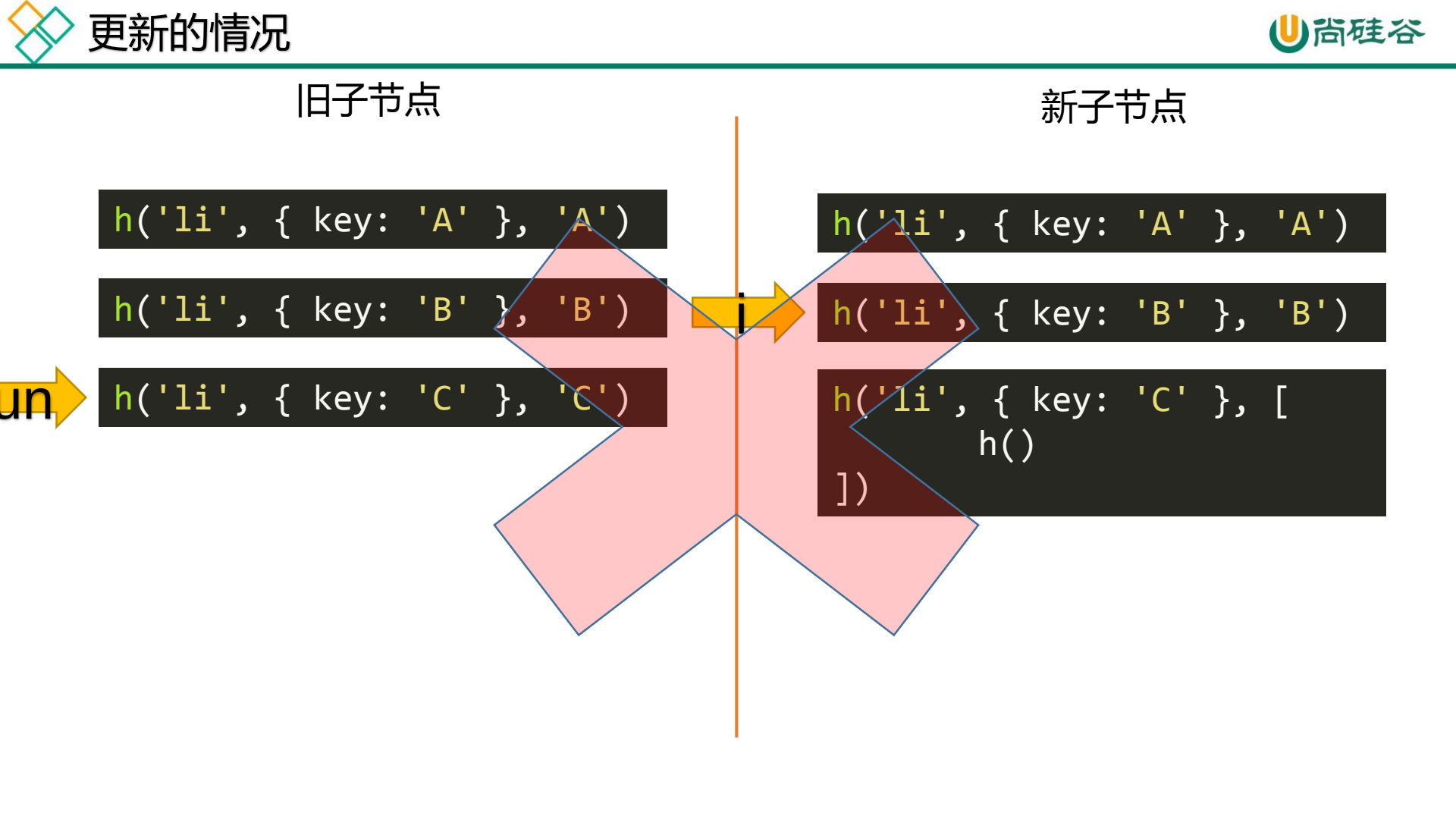
更新的情况

旧子节点

新子节点

```
h('li', { key: 'A' }, 'A')
h('li', { key: 'B' }, 'B')
h('li', { key: 'C' }, 'C')
```

```
h('li', { key: 'A' }, 'A')
h('li', { key: 'B' }, 'B')
h('li', { key: 'C' }, [
        h()
])
```



diff算法的子节点更新策略

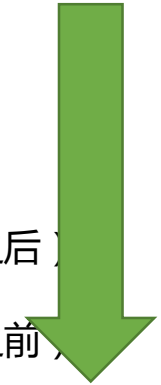
四种命中查找：

① 新前与旧前

② 新后与旧后

③ 新后与旧前（此种发生了，涉及移动节点，那么新后指向的节点，移动的旧后之后）

④ 新前与旧后（此种发生了，涉及移动节点，那么新前指向的节点，移动的旧前之前）



命中一种就不再进行命中判断了

如果都没有命中，就需要用循环来寻找了。把新前移动到旧前之前。

新增的情况

旧子节点

```
h('li', { key: 'A' }, 'A')
h('li', { key: 'B' }, 'B')
h('li', { key: 'C' }, 'C')
```

旧后

旧前

新子节点

```
h('li', { key: 'A' }, 'A')
h('li', { key: 'B' }, 'B')
h('li', { key: 'C' }, 'C')
h('li', { key: 'D' }, 'D')
h('li', { key: 'E' }, 'E')
```

新前

新后

while(新前<=新后&&旧前<=就后){
}

如果是旧节点先循环完毕，说明新节点中有要插入的节点。

新增的情况

旧子节点

```
h('li', { key: 'A' }, 'A')
h('li', { key: 'B' }, 'B')
h('li', { key: 'C' }, 'C')
h('li', { key: 'E' }, 'E')
```

旧后

旧前

新子节点

```
h('li', { key: 'A' }, 'A')
h('li', { key: 'B' }, 'B')
h('li', { key: 'D' }, 'D')
h('li', { key: 'E' }, 'E')
h('li', { key: 'C' }, 'C')
```

新前

新后

while(新前<=新后&&旧前<=就后){
}

如果是旧节点先循环完毕，说明新节点中有要插入的节点。

删除的情况

旧子节点

```
h('li', { key: 'A' }, 'A')
h('li', { key: 'B' }, 'B')
h('li', { key: 'C' }, 'C')
h('li', { key: 'D' }, 'D')
```

旧后

新子节点

```
h('li', { key: 'A' }, 'A')
h('li', { key: 'B' }, 'B')
h('li', { key: 'D' }, 'D')
```

新后

新前

while(新前<=新后&&旧前<=就后){
}

如果是新节点先循环完毕，如果老节点中还有剩余节点，说明他们是要被删除的节点。

多删除的情况

旧子节点

```
h('li', { key: 'A' }, 'A')
h('li', { key: 'B' }, 'B')
h('li', { key: 'C' }, 'C')
h('                    ')
h('li', { key: 'E' }, 'E')
```

D

undefined

旧前

旧后

新子节点

```
h('li', { key: 'A' }, 'A')
h('li', { key: 'B' }, 'B')
h('li', { key: 'D' }, 'D')
```

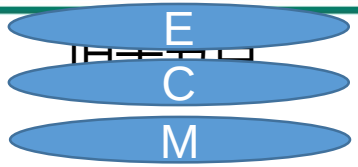
新后

新前

while(新前<=新后&&旧前<=就后){
}

如果是新节点先循环完毕，如果老节点中还有剩余节点(旧前和新后指针中间的节点)，说明他们是要被删除的节点。

复杂的情况



旧前 `h('li', { key: 'A' }, 'A')`

`h('li', { key: 'B' }, 'B')`

`h( undefined )`

旧后 `h('li', { key: 'D' }, 'D')`

`h( undefined )`

新后 `h('li', { key: 'M' }, 'M')`

新前

新子节点

`h('li', { key: 'E' }, 'E')`

`h('li', { key: 'C' }, 'C')`

当④新前与旧后命中的时候，此时要移动节点。移动新前指向的这个节点到老节点的旧前的前面

复杂的情况

旧子节点

h(undefined)

h(undefined)

h(undefined)

h(undefined)

旧后 h(undefined)

E

D

C

B

A

while(新<=新后 <=旧后){

}

新子节点

新前

新后

h('li', { key: 'E' }, 'E')

h('li', { key: 'D' }, 'D')

h('li', { key: 'C' }, 'C')

h('li', { key: 'B' }, 'B')

h('li', { key: 'A' }, 'A')

当③新后与旧前命中的时候，此时要移动节点。移动新前指向的这个节点到老节点的**旧后的后面**

手写子节点更新策略

课程总结