复习

- 初始化过程：init -> $mount -> compile -> new Watcher -> render -> update
  - runtime/index：实现$mount
  - core/index：全局api
  - core/instance/index：声明vue构造函数
  - entry-runtime-with-compiler：覆盖了$mount
  - mountComponent：执行渲染和更新，虚拟dom -》真实dom

```
<div id="demo">
    <h1>初始化流程</h1>
</div>
<script>
    // 创建实例
    const app = new Vue({
        el: '#demo'
    });
</script>
```

- 响应化：
  - observe()：返回Observer实例
  - Observer：区分当前值类型是对象还是数组
  - defineReactive
  - Dep
  - Watcher

  对象响应化

```
<div id="demo">
    <h1>数据响应化</h1>
    <p>{{obj.foo}}</p>
</div>
<script>
    // 创建实例
    const app = new Vue({
        el: '#demo',
        data: { obj: { foo: 'foo', bar: 'bar' } },
        mounted() {
            setTimeout(() => {
                this.obj.foo = 'fooooo'
            }, 1000);
        }
    });
```

```
</script>
```

数组响应化

```
<div id="demo">
  <h1>数据响应化</h1>
    <div v-for="a in arr" :key="a">
        {{ a }}
    </div>
</div>
<script>
    // 创建实例
    const app = new Vue({
        el: '#demo',
        data: { arr: ['foo', 'bar'] },
        mounted() {
            setTimeout(() => {
                this.arr.push('baz')
            }, 1000);
        }
    });
</script>
```

异步更新队列

update() core\observer\watcher.js

dep.notify()之后watcher执行更新，执行入队操作

queueWatcher(watcher) core\observer\scheduler.js

执行watcher入队操作

nextTick(flushSchedulerQueue) core\util\next-tick.js

nextTick按照特定异步策略执行队列操作

测试代码：watcher中update执行三次，但run仅执行一次

```
<div id="demo">
    <h1>异步更新</h1>
    <p>{{foo}}</p>
</div>
<script>
    // 创建实例
    const app = new Vue({
        el: '#demo',
        data: { foo: '' },
        mounted() {
            setInterval(() => {
```

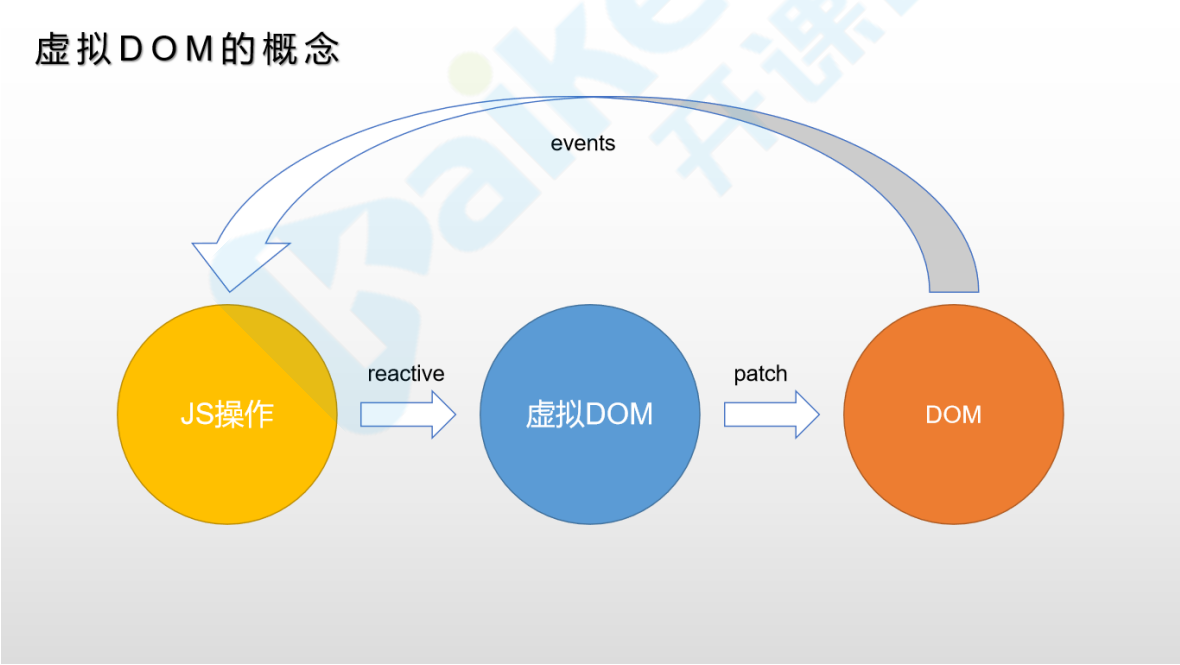
```
                this.foo = Math.random()
                this.foo = Math.random()
                this.foo = Math.random()
                console.log(p1.innerHTML)
                this.$nextTick(() => {
                    console.log(p1.innerHTML)
                })
            }, 1000);
        }
    });
</script>
```

> 宏任务和微任务相关知识补充请看这里

虚拟DOM

概念

虚拟DOM（Virtual DOM）是对DOM的JS抽象表示，它们是JS对象，能够描述DOM结构和关系。应用的各种状态变化会作用于虚拟DOM，最终映射到DOM上。



优点

- 虚拟DOM轻量、快速：当它们发生变化时通过新旧虚拟DOM比对可以得到最小DOM操作量，从而提升性能和用户体验。
- 跨平台：将虚拟dom更新转换为不同运行时特殊操作实现跨平台
- 兼容性：还可以加入兼容性代码增强操作的兼容性

必要性

vue 1.0中有细粒度的数据变化侦测，它是不需要虚拟DOM的，但是细粒度造成了大量开销，这对于大型项目来说是不可接受的。因此，vue 2.0选择了中等粒度的解决方案，每一个组件一个watcher实例，这样状态变化时只能通知到组件，再通过引入虚拟DOM去进行比对和渲染。

实现

mountComponent() core/instance/lifecycle.js

渲染、更新组件

```
// 定义更新函数
const updateComponent = () => {
    // 实际调用是在lifeCycleMixin中定义的_update和renderMixin中定义的_render
    vm._update(vm._render(), hydrating)
}
new Watcher(this.vm, updateComponent)
```

_render core/instance/render.js

生成虚拟dom

vm.$createElement() core/instance/render.js

真正用来创建vnode树的函数是vm.$createElement，其签名如下：

`vm.$createElement = (a, b, c, d) => createElement(vm, a, b, c, d, true)`

createElement() src\core\vdom\create-element.js

$createElement()是对createElement函数的封装，

createComponent core\vdom\create-component.js

用于创建组件并返回VNode

VNode

render返回的一个VNode实例，它的children还是VNode，最终构成一个树，就是虚拟DOM树，src\core\vdom\vnode.js

_update core\instance\lifecycle.js

update负责更新dom，转换vnode为dom

__patch__() platforms/web/runtime/index.js

__patch__是在平台特有代码中指定的

```
Vue.prototype.__patch__ = inBrowser ? patch : noop
```

patch是createPatchFunction的返回值，传递nodeOps和modules是web平台特别实现

```
export const patch: Function = createPatchFunction({ nodeOps, modules })
```

platforms\web\runtime\node-ops.js

定义各种原生dom基础操作方法

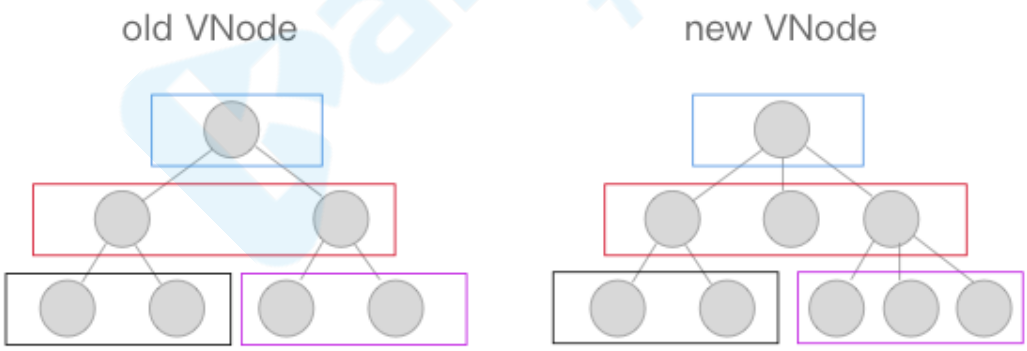
platforms\web\runtime\modules\index.js

`modules` 定义了属性更新实现

patch core\vdom\patch.js

通过**同层的树节点进行比较**而非对树进行逐层搜索遍历的方式，所以时间复杂度只有O(n)，是一种相当高效的算法。

同层级只做三件事：增删改。具体规则是：new VNode不存在就删；old VNode不存在就增；都存在就比较类型，类型不同直接替换、类型相同执行更新；



patchVnode

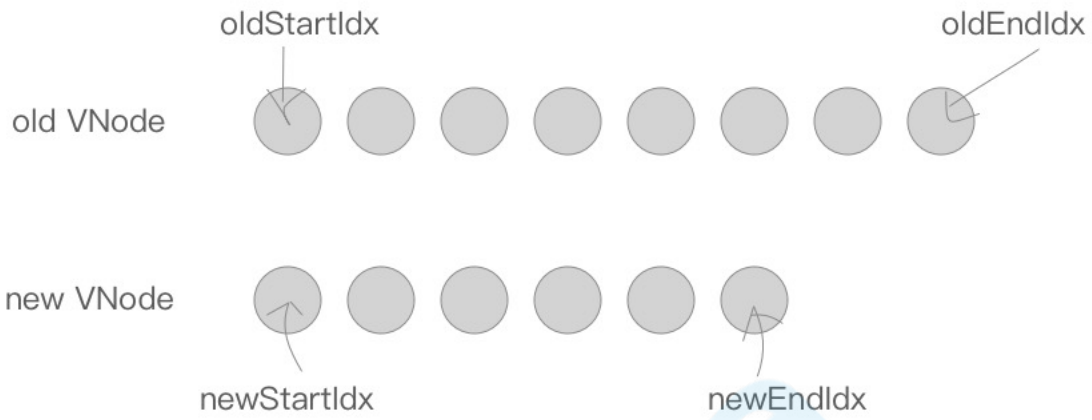
两个VNode类型相同，就执行更新操作，包括三种类型操作：**属性更新PROPS、文本更新TEXT、子节点更新REORDER**

patchVnode具体规则如下：

1. 如果新旧VNode都是**静态的**，那么只需要替换elm以及componentInstance即可。
2. 新老节点**均有children**子节点，则对子节点进行diff操作，调用**updateChildren**
3. 如果**老节点没有子节点而新节点存在子节点**，先清空老节点DOM的文本内容，然后为当前DOM节点加入子节点。
4. 当**新节点没有子节点而老节点有子节点**的时候，则移除该DOM节点的所有子节点。
5. 当**新老节点都无子节点**的时候，只是文本的替换。

updateChildren

updateChildren主要作用是用一种较高效的方式比对新旧两个VNode的children得出最小操作补丁。执行一个双循环是传统方式，vue中针对web场景特点做了特别的算法优化，我们看图说话：

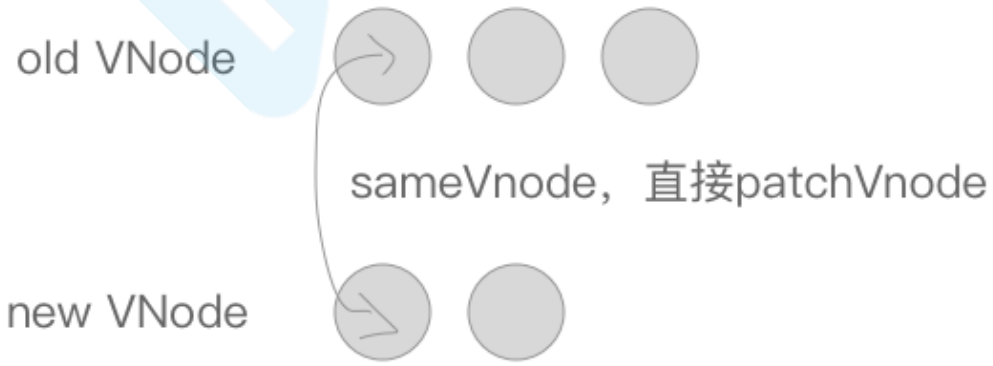


在新老两组VNode节点的左右头尾两侧都有一个变量标记，在**遍历过程中这几个变量都会向中间靠拢。**当**oldStartIdx > oldEndIdx**或者**newStartIdx > newEndIdx**时结束循环。

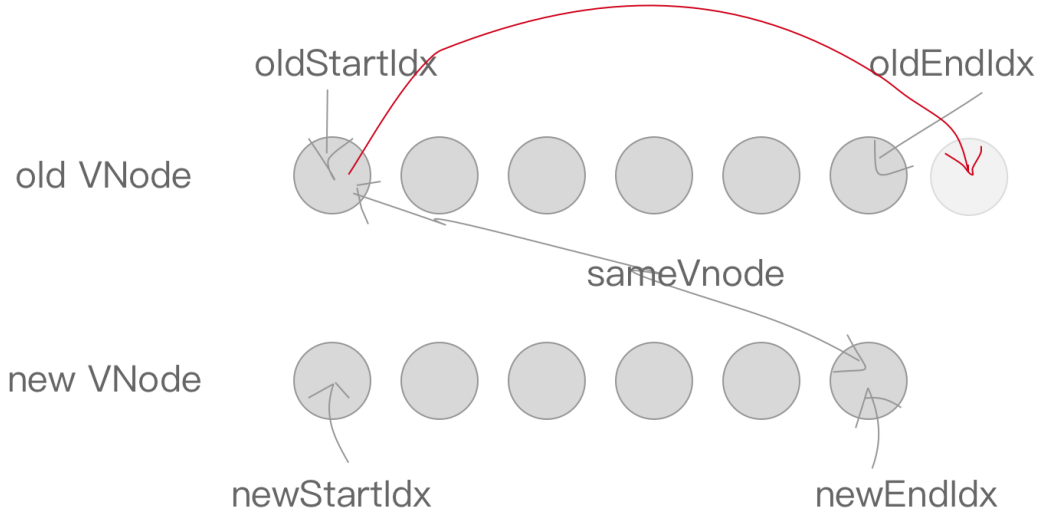
下面是遍历规则：

首先，oldStartVnode、oldEndVnode与newStartVnode、newEndVnode**两两交叉比较**，共有4种比较方法。

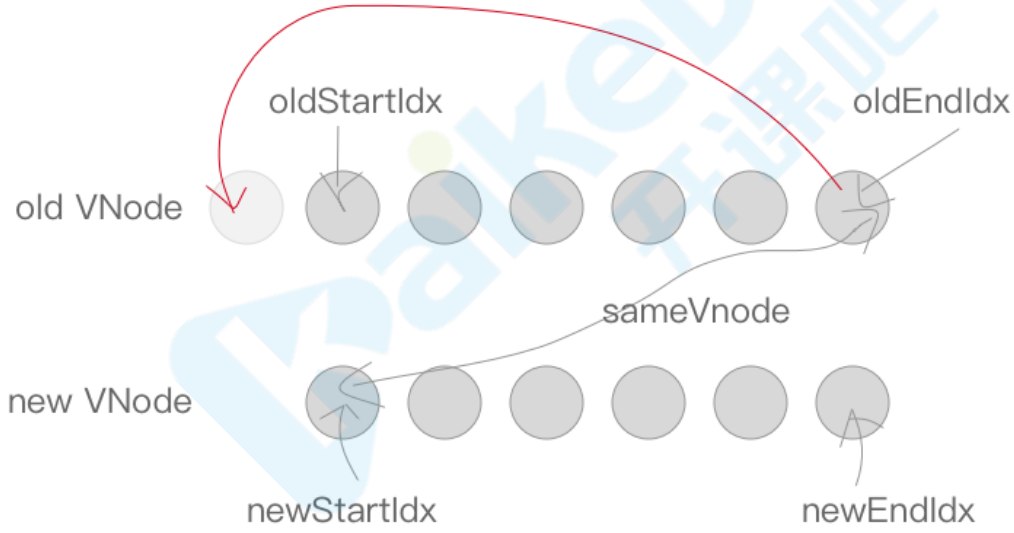
当 oldStartVnode和newStartVnode 或者 oldEndVnode和newEndVnode 满足sameVnode，直接将该VNode节点进行patchVnode即可，不需再遍历就完成了一次循环。如下图，



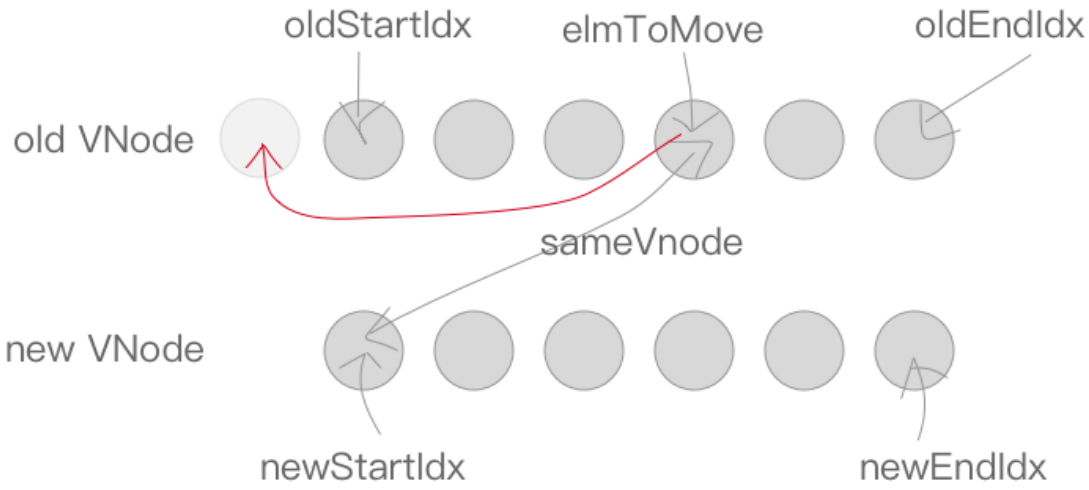
如果oldStartVnode与newEndVnode满足sameVnode。说明oldStartVnode已经跑到了oldEndVnode后面去了，进行patchVnode的同时还需要将真实DOM节点移动到oldEndVnode的后面。



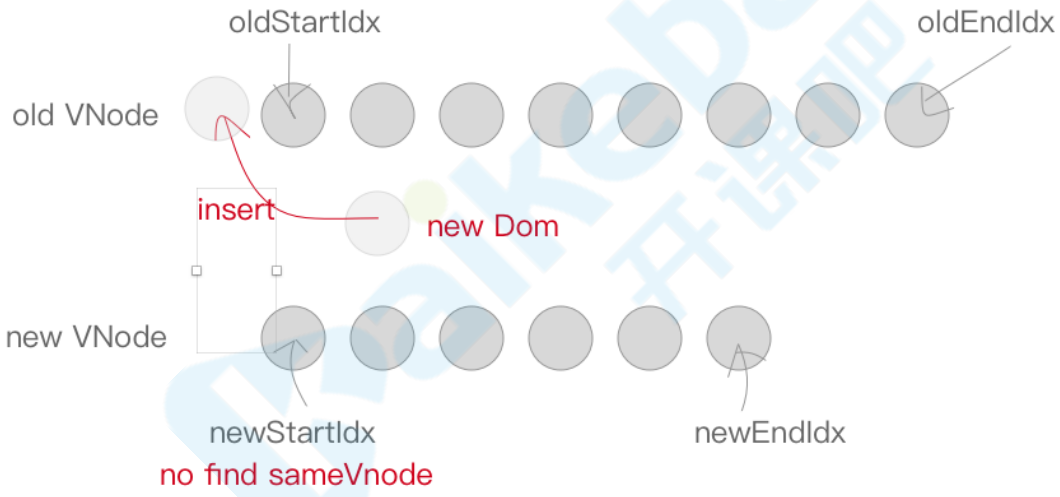
如果oldEndVnode与newStartVnode满足sameVnode，说明oldEndVnode跑到了oldStartVnode的前面，进行patchVnode的同时要将oldEndVnode对应DOM移动到oldStartVnode对应DOM的前面。



如果以上情况均不符合，则在old VNode中找与newStartVnode满足sameVnode的vnodeToMove，若存在执行patchVnode，同时将vnodeToMove对应DOM移动到oldStartVnode对应的DOM的前面。

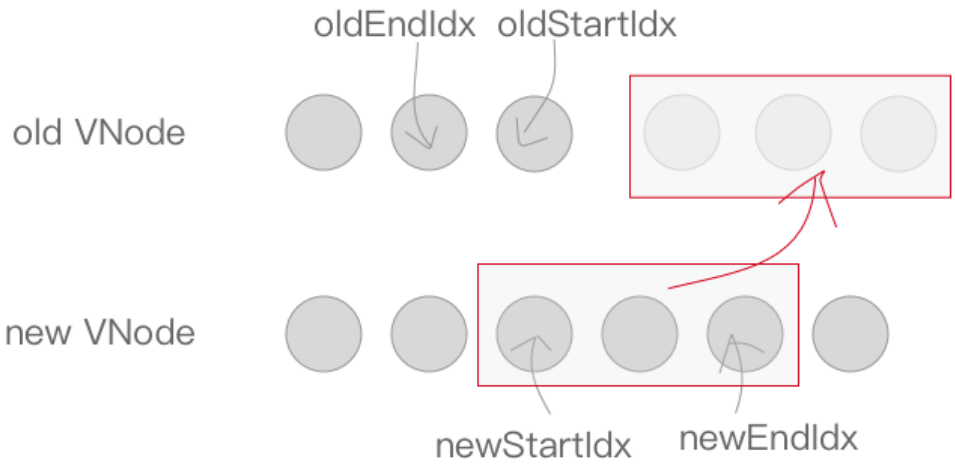


当然也有可能newStartVnode在old VNode节点中找不到一致的key，或者是即便key相同却不是sameVnode，这个时候会调用createElm创建一个新的DOM节点。

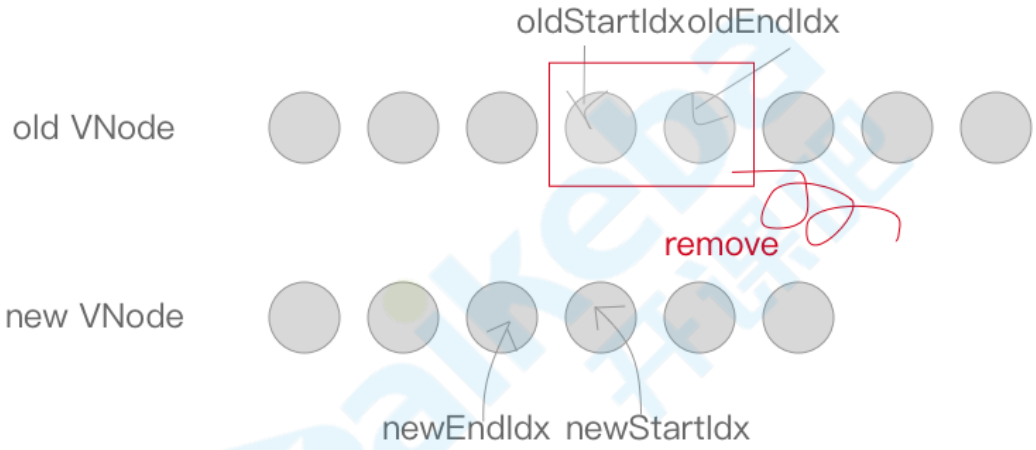


至此循环结束，但是我们还需要处理剩下的节点。

当结束时oldStartIdx > oldEndIdx，这个时候旧的VNode节点已经遍历完了，但是新的节点还没有。说明了新的VNode节点实际上比老的VNode节点多，需要将剩下的VNode对应的DOM插入到真实DOM中，此时调用addVnodes（批量调用createElm接口）。



但是，当结束时newStartIdx > newEndIdx时，说明新的VNode节点已经遍历完了，但是老的节点还有剩余，需要从文档中删 的节点删除。



作业

- 节点属性是如何更新的
- 组件化机制