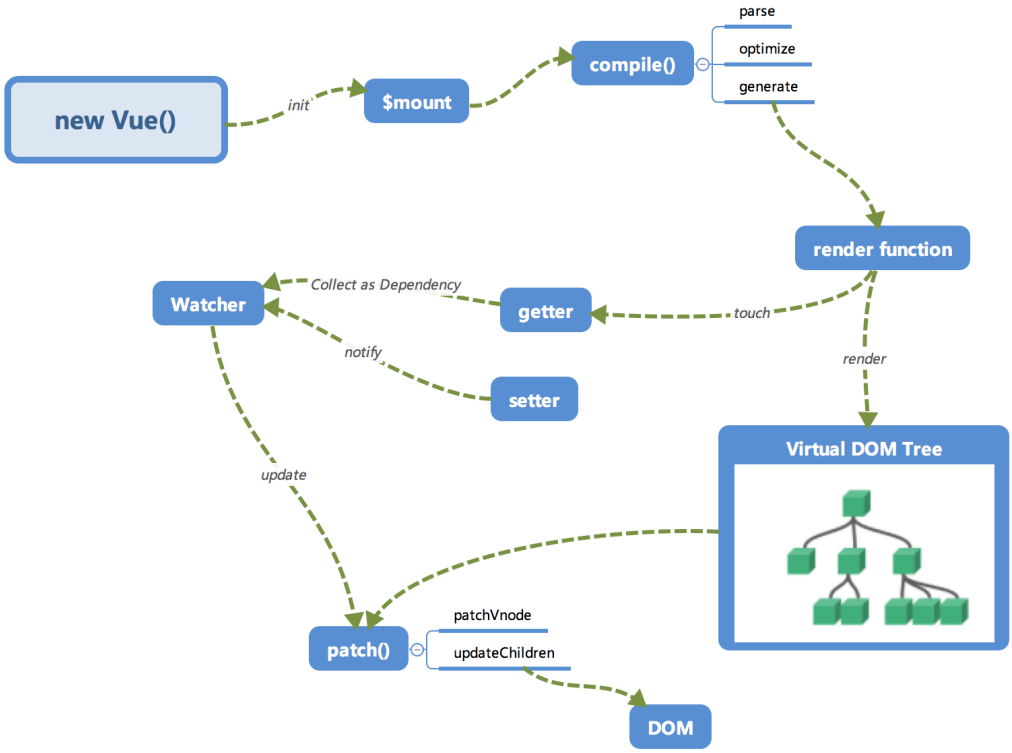
Vue.js 运行机制全局概览

全局概览

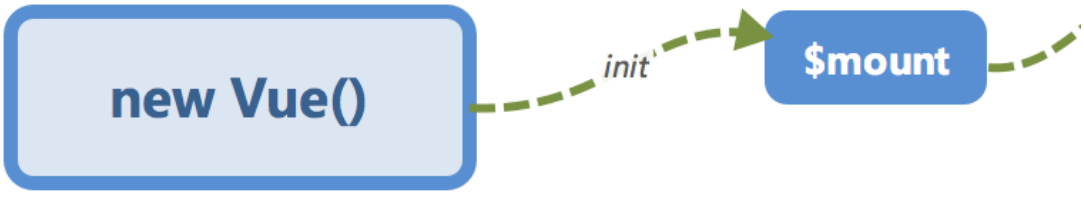
这一节笔者将为大家介绍一下 Vue.js 内部的整个流程，希望能让大家对全局有一个整体的印象，然后我们再来逐个模块进行讲解。从来没有了解过 Vue.js 实现的同学可能会对一些内容感到疑惑，这是很正常的，这一节的目的主要是为了让大家对整个流程有一个大概的认识，算是一个概览预备的过程，当把整本小册认真读完以后，再来阅读这一节，相信会有收获的。

首先我们来看一下笔者画的内部流程图。



大家第一次看到这个图一定是一头雾水的，没有关系，我们来逐个讲一下这些模块的作用以及调用关系。相信讲完之后大家对Vue.js内部运行机制会有一个大概的认识。

初始化及挂载

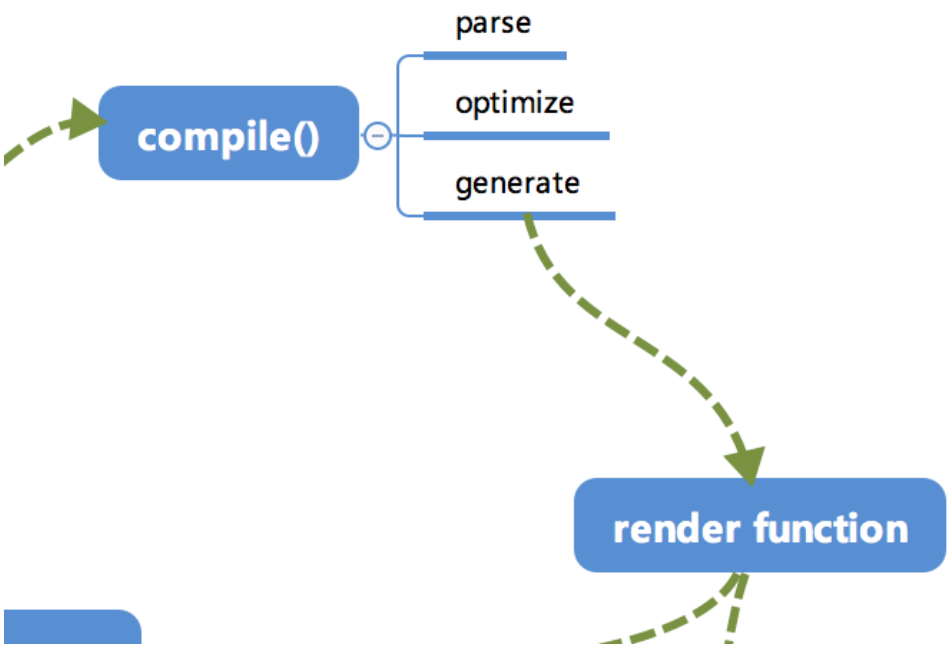


在 `new Vue()` 之后。 Vue 会调用 `_init` 函数进行初始化，也就是这里的 `init` 过程，它会初始化生命周期、事件、 props、 methods、 data、 computed 与 watch 等。其中最重要的是通过 `Object.defineProperty` 设置 `setter` 与 `getter` 函数，用来实现「**响应式**」以及「**依赖收集**」，后面会详细讲到，这里只要有一个印象即可。

初始化之后调用 `$mount` 会挂载组件，如果是运行时编译，即不存在 render function 但是存在 template 的情况，需要进行「**编译**」步骤。

编译

compile编译可以分成 `parse`、 `optimize` 与 `generate` 三个阶段，最终需要得到 render function。



parse

`parse` 会用正则等方式解析 template 模板中的指令、class、style等数据，形成AST。

optimize

`optimize` 的主要作用是标记 static 静态节点，这是 Vue 在编译过程中的一处优化，后面当 `update` 更新界面时，会有一个 `patch` 的过程， diff 算法会直接跳过静态节点，从而减少了比较的过程，优化了 `patch` 的性能。

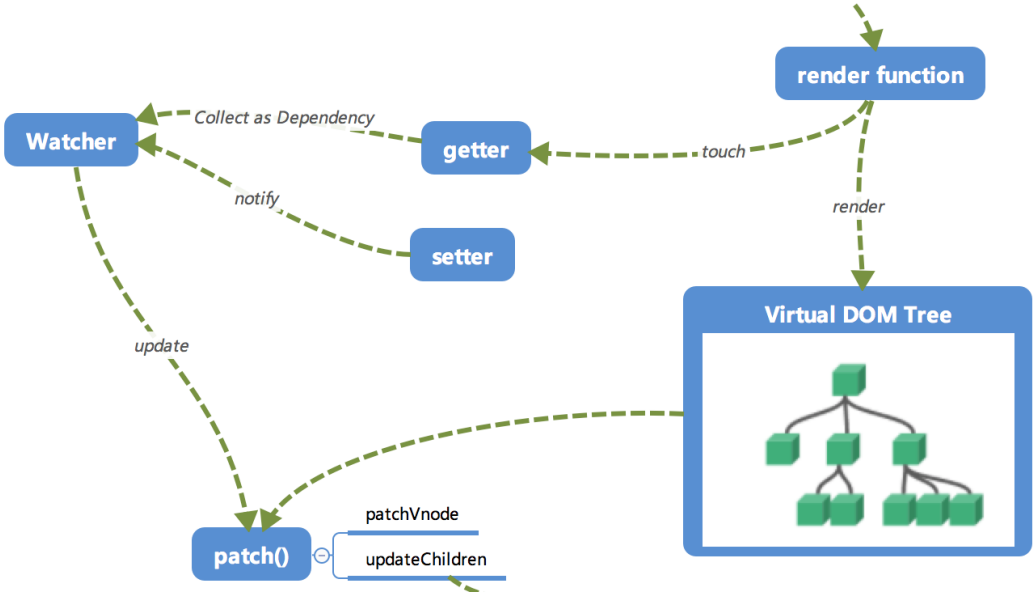
generate

`generate` 是将 AST 转化成 render function 字符串的过程，得到结果是 render 的字符串以及 staticRenderFns 字符串。

在经历过 `parse`、 `optimize` 与 `generate` 这三个阶段以后，组件中就会存在渲染 VNode 所需的 render function 了。

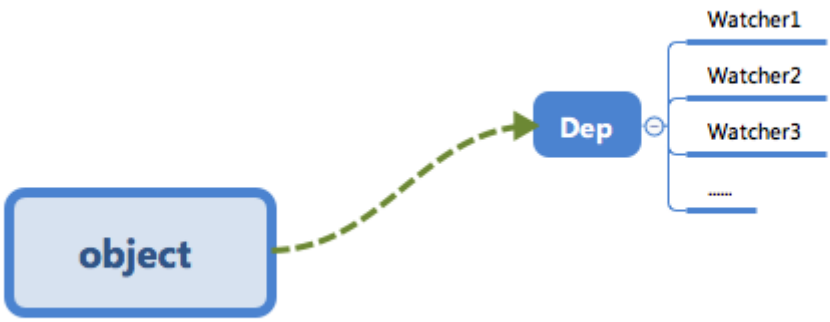
响应式

接下来也就是 Vue.js 响应式核心部分。



这里的 `getter` 跟 `setter` 已经在之前介绍过了，在 `init` 的时候通过 `Object.defineProperty` 进行了绑定，它使得当被设置的对象被读取的时候会执行 `getter` 函数，而在当被赋值的时候会执行 `setter` 函数。

当 render function 被渲染的时候，因为会读取所需对象的值，所以会触发 `getter` 函数进行「**依赖收集**」，「**依赖收集**」的目的是将观察者 Watcher 对象存放到当前闭包中的订阅者 Dep 的 subs 中。形成如下所示的这样一个关系。



在修改对象的值的时候，会触发对应的 `setter`， `setter` 通知之前「**依赖收集**」得到的 Dep 中的每一个 Watcher，告诉它们自己的值改变了，需要重新渲染视图。这时候这些 Watcher 就会开始调用 `update` 来更新视图，当然这中间还有一个 `patch` 的过程以及使用队列来异步更新的策略，这个我们后面再讲。

Virtual DOM

我们知道，render function 会被转化成 VNode 节点。Virtual DOM 其实就是一棵以 JavaScript 对象（VNode 节点）作为基础的树，用对象属性来描述节点，实际上它只是一层对真实 DOM 的抽象。最终可以通过一系列操作使这棵树映射到真实环境上。由于 Virtual DOM 是以 JavaScript 对象为基础而不依赖真实平台环境，所以使它具有了跨平台的能力，比如说浏览器平台、Weex、Node 等。

比如说下面这样一个例子：

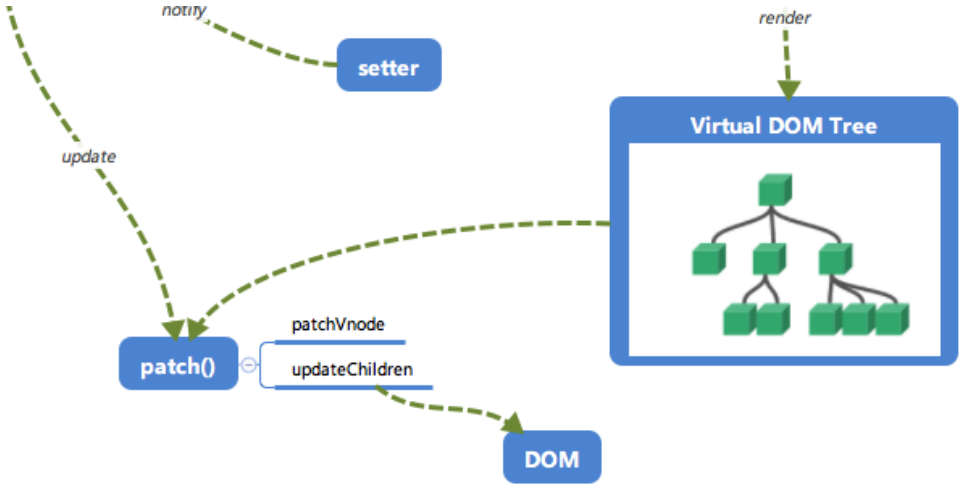
```
{
    tag: 'div',                 /*说明这是一个div标签*/
    children: [                 /*存放该标签的子节点*/
        {
            tag: 'a',           /*说明这是一个a标签*/
            text: 'click me'    /*标签的内容*/
        }
    ]
}
```

渲染后可以得到

```
<div>
    <a>click me</a>
</div>
```

这只是一个简单的例子，实际上的节点有更多的属性来标志节点，比如 isStatic （代表是否为静态节点）、 isComment （代表是否为注释节点）等。

更新视图

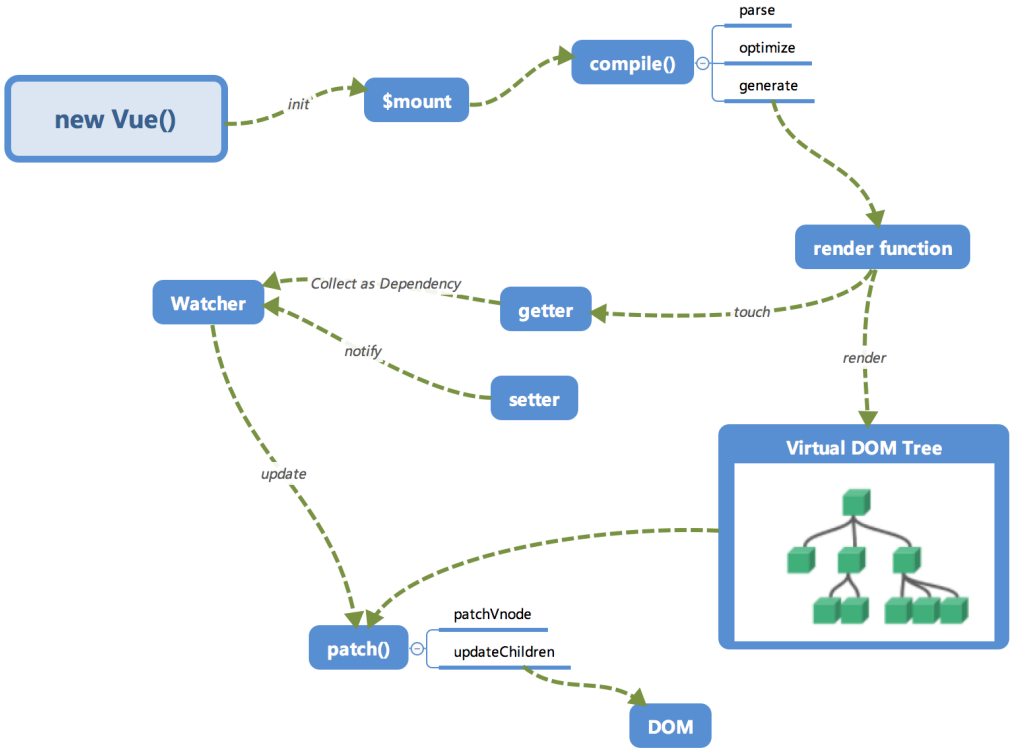


前面我们说到，在修改一个对象值的时候，会通过 `setter -> Watcher -> update` 的流程来修改对应的视图，那么最终是如何更新视图的呢?

当数据变化后，执行 render function 就可以得到一个新的 VNode 节点，我们如果想要得到新的视图，最简单粗暴的方法就是直接解析这个新的 VNode 节点，然后用 `innerHTML` 直接全部渲染到真实 DOM 中。但是其实我们只对其中的一小块内容进行了修改，这样做似乎有些「**浪费**」。

那么我们为什么不能只修改那些「改变了的地方」呢？这个时候就要介绍我们的「`patch`」了。我们会将新的 VNode 与旧的 VNode 一起传入 `patch` 进行比较，经过 diff 算法得出它们的「**差异**」。最后我们只需要将这些「**差异**」的对应 DOM 进行修改即可。

再看全局



回过头再来看看这张图，是不是大脑中已经有一个大概的脉络了呢？

那么，让我们继续学习每一个模块吧!

响应式系统的基本原理

响应式系统

Vue.js 是一款 MVVM 框架，数据模型仅仅是普通的 JavaScript 对象，但是对这些对象进行操作时，却能影响对应视图，它的核心实现就是「**响应式系统**」。尽管我们在使用 Vue.js 进行开发时不会直接修改「**响应式系统**」，但是理解它的实现有助于避开一些常见的「**坑**」，也有助于在遇见一些琢磨不透的问题时可以深入其原理来解决它。

`Object.defineProperty`

首先我们来介绍一下 `Object.defineProperty`，Vue.js就是基于它实现「**响应式系统**」的。

首先是使用方法：

```
/*
    obj: 目标对象
    prop: 需要操作的目标对象的属性名
    descriptor: 描述符

    return value 传入对象
*/
Object.defineProperty(obj, prop, descriptor)
```

descriptor的一些属性，简单介绍几个属性，具体可以参考 MDN 文档。

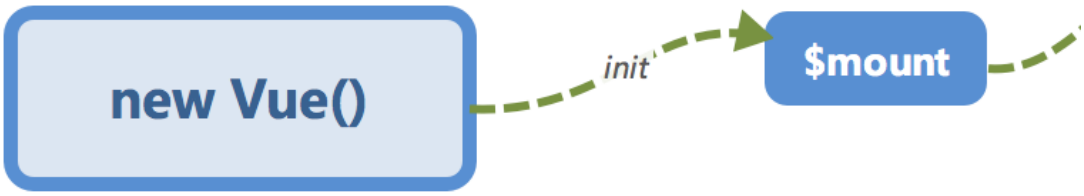
- `enumerable`，属性是否可枚举，默认 false。

- `configurable`，属性是否可以被修改或者删除，默认 false。
- `get`，获取属性的方法。
- `set`，设置属性的方法。

实现 `observer`（可观察的）

知道了 `Object.defineProperty` 以后，我们来用它使对象变成可观察的。

这一部分的内容我们在第二小节中已经初步介绍过，在 `init` 的阶段会进行初始化，对数据进行「**响应式化**」。



为了便于理解，我们不考虑数组等复杂的情况，只对对象进行处理。

首先我们定义一个 `cb` 函数，这个函数用来模拟视图更新，调用它即代表更新视图，内部可以是一些更新视图的方法。

```
function cb (val) {
    /* 渲染视图 */
    console.log("视图更新啦～");
}
```

然后我们定义一个 `defineReactive`，这个方法通过 `Object.defineProperty` 来实现对对象的「**响应式**」化，入参是一个 obj（需要绑定的对象）、key（obj的某一个属性），val（具体的值）。经过 `defineReactive` 处理以后，我们的 obj 的 key 属性在「读」的时候会触发 `reactiveGetter` 方法，而在该属性被「写」的时候则会触发 `reactiveSetter` 方法。

```
function defineReactive (obj, key, val) {
    Object.defineProperty(obj, key, {
        enumerable: true,        /* 属性可枚举 */
        configurable: true,      /* 属性可被修改或删除 */
        get: function reactiveGetter () {
            return val;          /* 实际上会依赖收集，下一小节会讲 */
        },
        set: function reactiveSetter (newVal) {
            if (newVal === val) return;
            cb(newVal);
        }
    });
}
```

当然这是不够的，我们需要在上面再封装一层 `observer` 。这个函数传入一个 value（需要「**响应式**」化的对象），通过遍历所有属性的方式对该对象的每一个属性都通过 `defineReactive` 处理。（注：实际上 observer 会进行递归调用，为了便于理解去掉了递归的过程）

```
function observer (value) {
    if (!value || (typeof value !== 'object')) {
        return;
    }

    Object.keys(value).forEach((key) => {
        defineReactive(value, key, value[key]);
    });
}
```

最后，让我们用 `observer` 来封装一个 Vue 吧！

在 Vue 的构造函数中，对 `options` 的 `data` 进行处理，这里的 `data` 想必大家很熟悉，就是平时我们在写 Vue 项目时组件中的 `data` 属性（实际上是一个函数，这里当作一个对象来简单处理）。

```
class Vue {
    /* Vue构造类 */
    constructor(options) {
        this._data = options.data;
        observer(this._data);
    }
}
```

这样我们只要 new 一个 Vue 对象，就会将 `data` 中的数据进行「**响应式**」化。如果我们对 `data` 的属性进行下面的操作，就会触发 `cb` 方法更新视图。

```
let o = new Vue({
    data: {
        test: "I am test."
    }
});
o._data.test = "hello,world.";  /* 视图更新啦～ */
```

至此，响应式原理已经介绍完了，接下来让我们学习「**响应式系统**」的另一部分 ——「**依赖收集**」。

注：本节代码参考《响应式系统的基本原理》。

响应式系统的依赖收集追踪原理

为什么要依赖收集？

先举个栗子🌰

我们现在有这么一个 Vue 对象。

```
new Vue({
    template:
        `<div>
            <span>{{text1}}</span>
            <span>{{text2}}</span>
        <div>`,
    data: {
        text1: 'text1',
        text2: 'text2',
        text3: 'text3'
    }
});
```

然后我们做了这么一个操作。

```
this.text3 = 'modify text3';
```

我们修改了 `data` 中 `text3` 的数据，但是因为视图中并不需要用到 `text3` ，所以我们并不需要触发上一章所讲的 `cb` 函数来更新视图，调用 `cb` 显然是不正确的。

再来一个栗子🌰

假设我们现在有一个全局的对象，我们可能会在多个 Vue 对象中用到它进行展示。

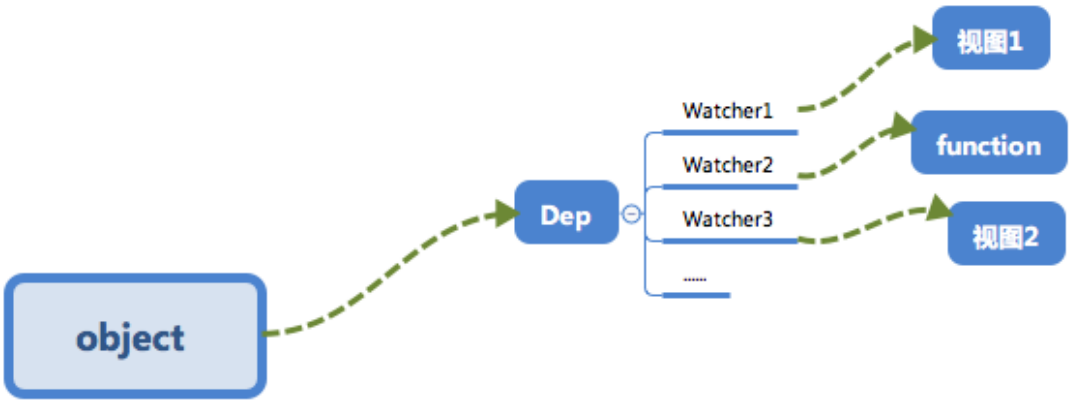
```
let globalObj = {
    text1: 'text1'
};

let o1 = new Vue({
    template:
        `<div>
            <span>{{text1}}</span>
        <div>`,
    data: globalObj
});

let o2 = new Vue({
    template:
        `<div>
            <span>{{text1}}</span>
        <div>`,
    data: globalObj
});
```

这个时候，我们执行了如下操作。

```
globalObj.text1 = 'hello,text1';
```

我们应该需要通知 `o1` 以及 `o2` 两个vm实例进行视图的更新，「依赖收集」会让 `text1` 这个数据知道“哦～有两个地方依赖我的数据，我变化的时候需要通知它们～”。

最终会形成数据与视图的一种对应关系，如下图。



接下来我们来介绍一下「依赖收集」是如何实现的。

订阅者 Dep

首先我们来实现一个订阅者 Dep ，它的主要作用是用来存放 `Watcher` 观察者对象。

```
class Dep {
    constructor () {
        /* 用来存放Watcher对象的数组 */
        this.subs = [];
    }

    /* 在subs中添加一个Watcher对象 */
    addSub (sub) {
        this.subs.push(sub);
    }

    /* 通知所有Watcher对象更新视图 */
    notify () {
        this.subs.forEach((sub) => {
            sub.update();
        })
    }
}
```

为了便于理解我们只实现了添加的部分代码，主要是两件事情：

1. 用 `addSub` 方法可以在目前的 `Dep` 对象中增加一个 `Watcher` 的订阅操作；
2. 用 `notify` 方法通知目前 `Dep` 对象的 `subs` 中的所有 `Watcher` 对象触发更新操作。

观察者 Watcher

```
class Watcher {
    constructor () {
        /* 在new一个Watcher对象时将该对象赋值给Dep.target，在get中会用到 */
        Dep.target = this;
    }
```

```
    /* 更新视图的方法 */
    update () {
        console.log("视图更新啦～");
    }
}

Dep.target = null;
```

依赖收集

接下来我们修改一下 `defineReactive` 以及 Vue 的构造函数，来完成依赖收集。

我们在闭包中增加了一个 Dep 类的对象，用来收集 `Watcher` 对象。在对象被「读」的时候，会触发 `reactiveGetter` 函数把当前的 `Watcher` 对象（存放在 Dep.target 中）收集到 `Dep` 类中去。之后如果当该对象被「**写**」的时候，则会触发 `reactiveSetter` 方法，通知 `Dep` 类调用 `notify` 来触发所有 `Watcher` 对象的 `update` 方法更新对应视图。

```
function defineReactive (obj, key, val) {
    /* 一个Dep类对象 */
    const dep = new Dep();

    Object.defineProperty(obj, key, {
        enumerable: true,
        configurable: true,
        get: function reactiveGetter () {
            /* 将Dep.target（即当前的Watcher对象存入dep的subs中） */
            dep.addSub(Dep.target);
            return val;
        },
        set: function reactiveSetter (newVal) {
            if (newVal === val) return;
            /* 在set的时候触发dep的notify来通知所有的Watcher对象更新视图 */
            dep.notify();
        }
    });
}

class Vue {
    constructor(options) {
        this._data = options.data;
        observer(this._data);
        /* 新建一个Watcher观察者对象，这时候Dep.target会指向这个Watcher对象 */
        new Watcher();
        /* 在这里模拟render的过程，为了触发test属性的get函数 */
        console.log('render~', this._data.test);
    }
}
```

小结

总结一下。

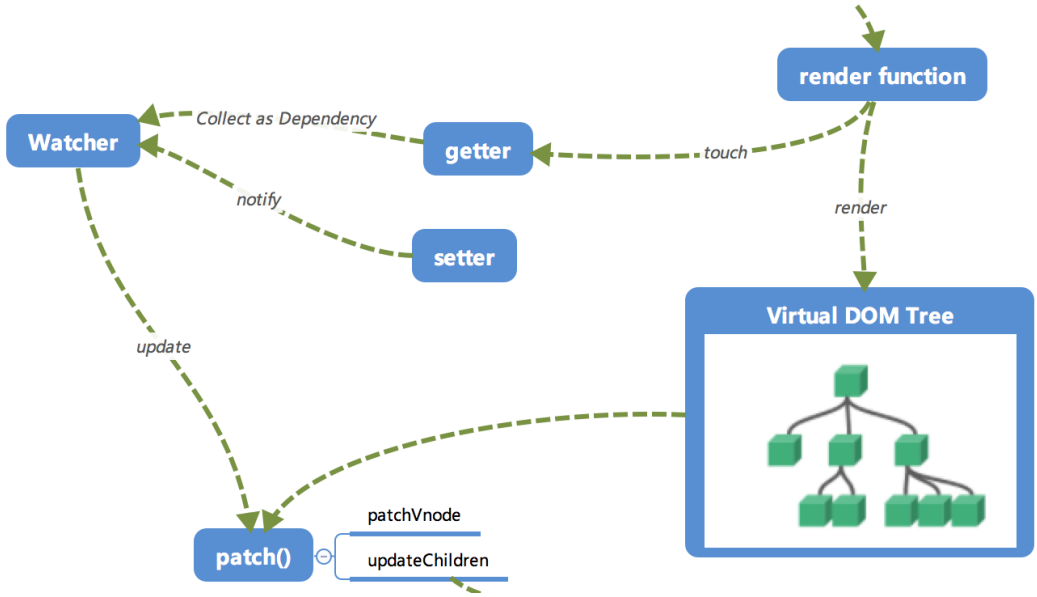
首先在 `observer` 的过程中会注册 `get` 方法，该方法用来进行「**依赖收集**」。在它的闭包中会有一个 `Dep` 对象，这个对象用来存放 Watcher 对象的实例。其实「**依赖收集**」的过程就是把 `Watcher` 实例存放到对应的 `Dep` 对象中去。`get` 方法可以让当前的 `Watcher` 对象（Dep.target）存放到它的 subs 中（`addSub`）方法，在数据变化时，`set` 会调用 `Dep` 对象的 `notify` 方法通知它内部所有的 `Watcher` 对象进行视图更新。

这是 `Object.defineProperty` 的 `set/get` 方法处理的事情，那么「**依赖收集**」的前提条件还有两个：

1. 触发 `get` 方法；
2. 新建一个 Watcher 对象。

这个我们在 Vue 的构造类中处理。新建一个 `Watcher` 对象只需要 new 出来，这时候 `Dep.target` 已经指向了这个 new 出来的 `Watcher` 对象来。而触发 `get` 方法也很简单，实际上只要把 render function 进行渲染，那么其中的依赖的对象都会被「读取」，这里我们通过打印来模拟这个过程，读取 test 来触发 `get` 进行「依赖收集」。

本章我们介绍了「依赖收集」的过程，配合之前的响应式原理，已经把整个「响应式系统」介绍完毕了。其主要就是 `get` 进行「依赖收集」。`set` 通过观察者来更新视图，配合下图仔细捋一捋，相信一定能搞懂它！



注：本节代码参考《响应式系统的依赖收集追踪原理》。

实现 Virtual DOM 下的一个 VNode 节点

什么是VNode

我们知道，render function 会被转化成 VNode 节点。Virtual DOM 其实就是一棵以 JavaScript 对象（VNode 节点）作为基础的树，用对象属性来描述节点，实际上它只是一层对真实 DOM 的抽象。最终可以通过一系列操作使这棵树映射到真实环境上。由于 Virtual DOM 是以 JavaScript 对象为基础而不依赖真实平台环境，所以使它具有了跨平台的能力，比如说浏览器平台、Weex、Node 等。

实现一个VNode

VNode 归根结底就是一个 JavaScript 对象，只要这个类的一些属性可以正确直观地描述清楚当前节点的信息即可。我们来实现一个简单的 `VNode` 类，加入一些基本属性，为了便于理解，我们先不考虑复杂的情况。

```
class VNode {
    constructor (tag, data, children, text, elm) {
        /*当前节点的标签名*/
        this.tag = tag;
        /*当前节点的一些数据信息，比如props、attrs等数据*/
        this.data = data;
        /*当前节点的子节点，是一个数组*/
        this.children = children;
        /*当前节点的文本*/
        this.text = text;
        /*当前虚拟节点对应的真实dom节点*/
        this.elm = elm;
    }
}
```

比如我目前有这么一个 Vue 组件。

```
<template>
  <span class="demo" v-show="isShow">
    This is a span.
  </span>
</template>
```

用 JavaScript 代码形式就是这样的。

```
function render () {
    return new VNode(
        'span',
        {
            /* 指令集合数组 */
            directives: [
                {
                    /* v-show指令 */
                    rawName: 'v-show',
                    expression: 'isShow',
                    name: 'show',
                    value: true
                }
            ],
            /* 静态class */
            staticClass: 'demo'
        },
        [ new VNode(undefined, undefined, undefined, 'This is a span.') ]
    );
}
```

看看转换成 VNode 以后的情况。

```
{
    tag: 'span',
    data: {
        /* 指令集合数组 */
```

```
        directives: [
            {
                /* v-show指令 */
                rawName: 'v-show',
                expression: 'isShow',
                name: 'show',
                value: true
            }
        ],
        /* 静态class */
        staticClass: 'demo'
    },
    text: undefined,
    children: [
        /* 子节点是一个文本VNode节点 */
        {
            tag: undefined,
            data: undefined,
            text: 'This is a span.',
            children: undefined
        }
    ]
}
```

然后我们可以将 VNode 进一步封装一下，可以实现一些产生常用 VNode 的方法。

- 创建一个空节点

```
function createEmptyVNode () {
    const node = new VNode();
    node.text = '';
    return node;
}
```

- 创建一个文本节点

```
function createTextVNode (val) {
  return new VNode(undefined, undefined, undefined, String(val));
}
```

- 克隆一个 VNode 节点

```
function cloneVNode (node) {
    const cloneVnode = new VNode(
        node.tag,
        node.data,
        node.children,
        node.text,
        node.elm
    );
    return cloneVnode;
}
```

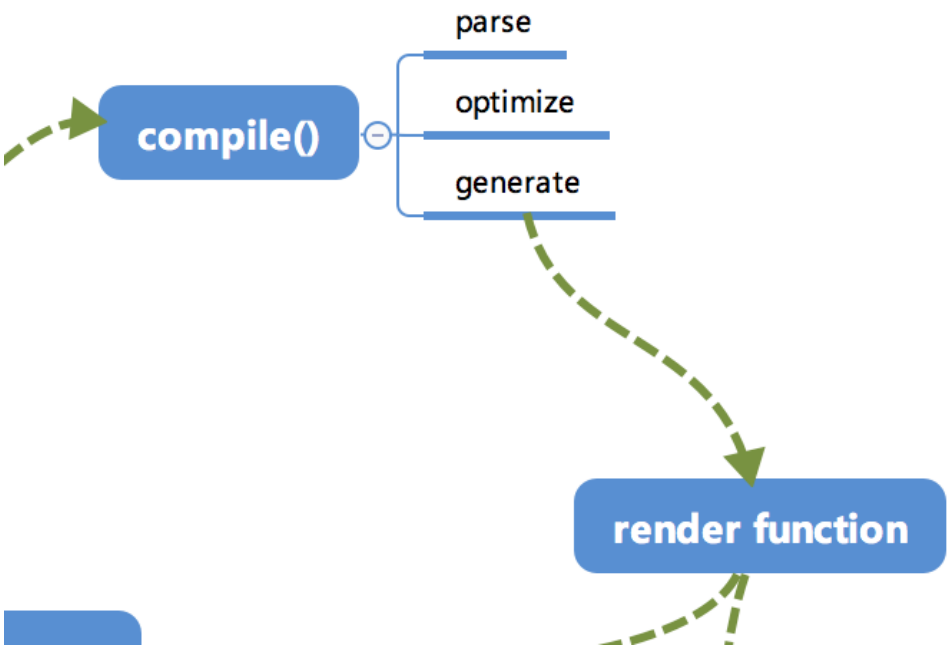
总的来说，VNode 就是一个 JavaScript 对象，用 JavaScript 对象的属性来描述当前节点的一些状态，用 VNode 节点的形式来模拟一棵 Virtual DOM 树。

注：本节代码参考《实现 Virtual DOM 下的一个 VNode 节点》。

template 模板是怎样通过 Compile 编译的

Compile

`compile` 编译可以分成 `parse`、`optimize` 与 `generate` 三个阶段，最终需要得到 render function。这部分内容不算 Vue.js 的响应式核心，只是用来编译的，笔者认为在精力有限的情况下不需要追究其全部的实现细节，能够把握如何解析的大致流程即可。



由于解析过程比较复杂，直接上代码可能会导致不了解这部分内容的同学一头雾水。所以笔者准备提供一个 template 的示例，通过这个示例的变化来看解析的过程。但是解析的过程及结果都是将最重要的部分抽离出来展示，希望能让读者更好地了解其核心部分的实现。

```
<div :class="c" class="demo" v-if="isShow">
    <span v-for="item in sz">{{item}}</span>
</div>
```

```
var html = '<div :class="c" class="demo" v-if="isShow"><span v-for="item in sz">
{{item}}</span></div>';
```

接下来的过程都会依赖这个示例来进行。

parse

首先是 `parse`，`parse` 会用正则等方式将 template 模板中进行字符串解析，得到指令、class、style 等数据，形成 AST（在计算机科学中，抽象语法树（abstract syntax tree或者缩写为AST），或者语法树（syntax tree），是源代码的抽象语法结构的树状表现形式，这里特指编程语言的源代码。）。

这个过程比较复杂，会涉及到比较多的正则进行字符串解析，我们来看一下得到的 AST 的样子。

```
{
    /* 标签属性的map，记录了标签上属性 */
    'attrsMap': {
        ':class': 'c',
        'class': 'demo',
        'v-if': 'isShow'
    },
    /* 解析得到的:class */
    'classBinding': 'c',
    /* 标签属性v-if */
    'if': 'isShow',
    /* v-if的条件 */
    'ifConditions': [
        {
            'exp': 'isShow'
        }
    ],
    /* 标签属性class */
    'staticClass': 'demo',
    /* 标签的tag */
    'tag': 'div',
    /* 子标签数组 */
    'children': [
        {
            'attrsMap': {
                'v-for': "item in sz"
            },
            /* for循环的参数 */
            'alias': "item",
            /* for循环的对象 */
            'for': 'sz',
            /* for循环是否已经被处理的标记位 */
            'forProcessed': true,
            'tag': 'span',
            'children': [
                {
                    /* 表达式，_s是一个转字符串的函数 */
                    'expression': '_s(item)',
                    'text': '{{item}}'
                }
            ]
        }
    ]
}
```

最终得到的 AST 通过一些特定的属性，能够比较清晰地描述出标签的属性以及依赖关系。

接下来我们用代码来讲解一下如何使用正则来把 template 编译成我们需要的 AST 的。

正则

首先我们定义一下接下来我们会用到的正则。

```
const ncname = '[a-zA-Z_][\\w\\-\\.]*';
const singleAttrIdentifier = /([^\s"'<>/=]+)/
const singleAttrAssign = /(?:=)/
const singleAttrValues = [
  /"([^"]*)"+/.source,
  /'([^']*)'+/.source,
  /([^\s"'=<>`]+)/.source
]
const attribute = new RegExp(
  '^\\s*' + singleAttrIdentifier.source +
  '(?:\\s*(' + singleAttrAssign.source + ')' +
  '\\s*(?:' + singleAttrValues.join('|') + '))?'
)

const qnameCapture = '((?:' + ncname + '\\:)?' + ncname + ')'
const startTagOpen = new RegExp('^<' + qnameCapture)
const startTagClose = /^\s*(\/?)>/

const endTag = new RegExp('^<\\/' + qnameCapture + '[^>]*>')

const defaultTagRE = /\{\{((?:.|\n)+?)\}\}/g

const forAliasRE = /(.*?)\s+(?:in|of)\s+(.*)/
```

advance

因为我们解析 template 采用循环进行字符串匹配的方式，所以每匹配解析完一段我们需要将已经匹配掉的去掉，头部的指针指向接下来需要匹配的部分。

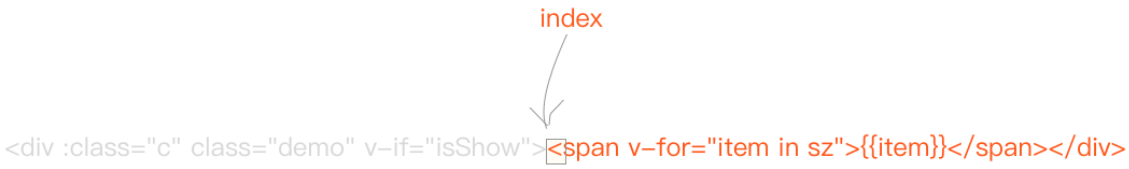
```
function advance (n) {
    index += n
    html = html.substring(n)
}
```

举个例子，当我们把第一个 div 的头标签全部匹配完毕以后，我们需要将这部分除去，也就是向右移动 43 个字符。

调用 `advance` 函数

```
advance(43);
```

得到结果



parseHTML

首先我们需要定义个 `parseHTML` 函数，在里面我们循环解析 template 字符串。

```
function parseHTML () {
    while(html) {
        let textEnd = html.indexOf('<');
        if (textEnd === 0) {
            if (html.match(endTag)) {
                //...process end tag
                continue;
            }
            if (html.match(startTagOpen)) {
                //...process start tag
                continue;
            }
        } else {
            //...process text
            continue;
        }
    }
}
```

`parseHTML` 会用 `while` 来循环解析 template ，用正则在匹配到标签头、标签尾以及文本的时候分别进行不同的处理。直到整个 template 被解析完毕。

parseStartTag

我们来写一个 `parseStartTag` 函数，用来解析起始标签（"

"部分的内容）。

```
function parseStartTag () {
    const start = html.match(startTagOpen);
    if (start) {
        const match = {
            tagName: start[1],
            attrs: [],
            start: index
        }
        advance(start[0].length);

        let end, attr
        while (!(end = html.match(startTagClose)) && (attr =
html.match(attribute))) {
            advance(attr[0].length)
            match.attrs.push({
                name: attr[1],
                value: attr[3]
```

```
            });
        }
        if (end) {
            match.unarySlash = end[1];
            advance(end[0].length);
            match.end = index;
            return match
        }
    }
}
```

首先用 `startTagOpen` 正则得到标签的头部，可以得到 `tagName`（标签名称），同时我们需要一个数组 `attrs` 用来存放标签内的属性。

```
const start = html.match(startTagOpen);
const match = {
    tagName: start[1],
    attrs: [],
    start: index
}
advance(start[0].length);
```

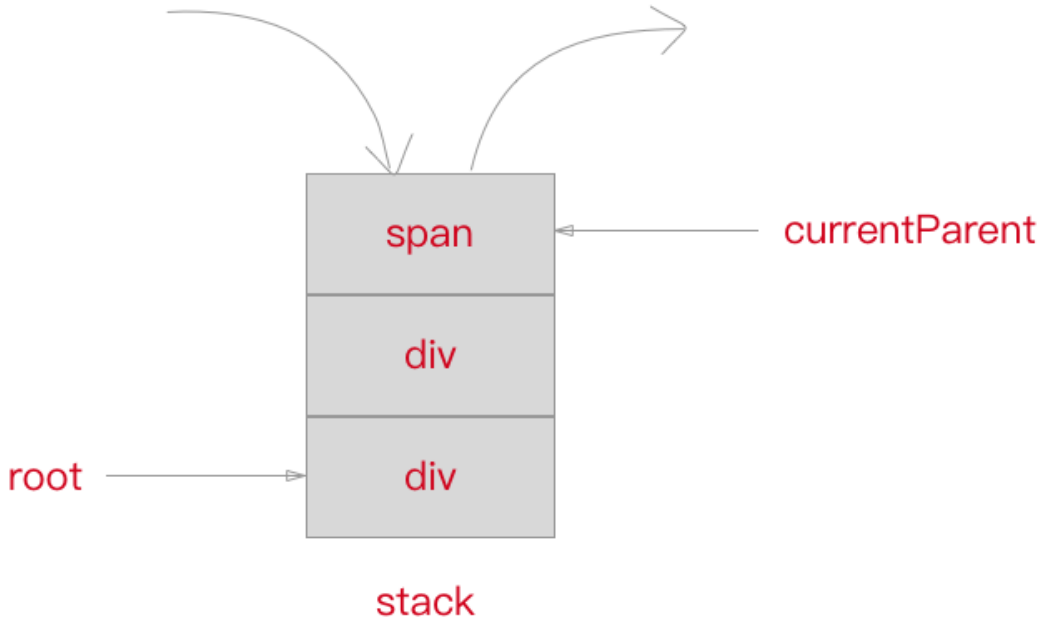
接下来使用 `startTagClose` 与 `attribute` 两个正则分别用来解析标签结束以及标签内的属性。这段代码用 `while` 循环一直到匹配到 `startTagClose` 为止，解析内部所有的属性。

```
let end, attr
while (!(end = html.match(startTagClose)) && (attr = html.match(attribute))) {
    advance(attr[0].length)
    match.attrs.push({
        name: attr[1],
        value: attr[3]
    });
}
if (end) {
    match.unarySlash = end[1];
    advance(end[0].length);
    match.end = index;
    return match
}
```

stack

此外，我们需要维护一个 **stack** 栈来保存已经解析好的标签头，这样我们可以根据在解析尾部标签的时候得到所属的层级关系以及父标签。同时我们定义一个 `currentParent` 变量用来存放当前标签的父标签节点的引用， `root` 变量用来指向根标签节点。

```
const stack = [];
let currentParent, root;
```



知道这个以后，我们优化一下 `parseHTML` ，在 `startTagOpen` 的 `if` 逻辑中加上新的处理。

```
if (html.match(startTagOpen)) {
    const startTagMatch = parseStartTag();
    const element = {
        type: 1,
        tag: startTagMatch.tagName,
        lowerCasedTag: startTagMatch.tagName.toLowerCase(),
        attrsList: startTagMatch.attrs,
        attrsMap: makeAttrsMap(startTagMatch.attrs),
        parent: currentParent,
        children: []
    }

    if(!root){
        root = element
    }

    if(currentParent){
        currentParent.children.push(element);
    }

    stack.push(element);
    currentParent = element;
    continue;
}
```

我们将 `startTagMatch` 得到的结果首先封装成 `element` ，这个就是最终形成的 AST 的节点，标签节点的 type 为 1。

```
const startTagMatch = parseStartTag();
const element = {
    type: 1,
    tag: startTagMatch.tagName,
    attrsList: startTagMatch.attrs,
    attrsMap: makeAttrsMap(startTagMatch.attrs),
    parent: currentParent,
    children: []
}
```

然后让 `root` 指向根节点的引用。

```
if(!root){
    root = element
}
```

接着我们将当前节点的 `element` 放入父节点 `currentParent` 的 `children` 数组中。

```
if(currentParent){
    currentParent.children.push(element);
}
```

最后将当前节点 `element` 压入 stack 栈中，并将 `currentParent` 指向当前节点，因为接下去下一个解析如果还是头标签或者是文本的话，会成为当前节点的子节点，如果是尾标签的话，那么将会从栈中取出当前节点，这种情况我们接下来要讲。

```
stack.push(element);
currentParent = element;
continue;
```

其中的 `makeAttrsMap` 是将 attrs 转换成 map 格式的一个方法。

```
function makeAttrsMap (attrs) {
    const map = {}
    for (let i = 0, l = attrs.length; i < l; i++) {
        map[attrs[i].name] = attrs[i].value;
    }
    return map
}
```

parseEndTag

同样，我们在 `parseHTML` 中加入对尾标签的解析函数，为了匹配如“

”。

```
const endTagMatch = html.match(endTag)
 if (endTagMatch) {
    advance(endTagMatch[0].length);
    parseEndTag(endTagMatch[1]);
    continue;
}
```

用 `parseEndTag` 来解析尾标签，它会从 stack 栈中取出最近的跟自己标签名一致的那个元素，将 `currentParent` 指向那个元素，并将该元素之前的元素都从 stack 中出栈。

这里可能有同学会问，难道解析的尾元素不应该对应 stack 栈的最上面的一个元素才对吗?

其实不然，比如说可能会存在自闭合的标签，如“
”，或者是写了“”但是没有加上“< /span>”的情况，这时候就要找到 stack 中的第二个位置才能找到同名标签。

```
function parseEndTag (tagName) {
    let pos;
    for (pos = stack.length - 1; pos >= 0; pos--) {
        if (stack[pos].lowerCasedTag === tagName.toLowerCase()) {
            break;
        }
    }

    if (pos >= 0) {
        stack.length = pos;
        currentParent = stack[pos];
    }
}
```

parseText

最后是解析文本，这个比较简单，只需要将文本取出，然后有两种情况，一种是普通的文本，直接构建一个节点 push 进当前 `currentParent` 的 children 中即可。还有一种情况是文本是如“{{item}}”这样的 Vue.js 的表达式，这时候我们需要用 `parseText` 来将表达式转化成代码。

```
text = html.substring(0, textEnd)
advance(textEnd)
let expression;
if (expression = parseText(text)) {
    currentParent.children.push({
        type: 2,
        text,
        expression
    });
} else {
    currentParent.children.push({
        type: 3,
        text,
    });
}
```

```
continue;
```

我们会用到一个 `parseText` 函数。

```
function parseText (text) {
    if (!defaultTagRE.test(text)) return;

    const tokens = [];
    let lastIndex = defaultTagRE.lastIndex = 0
    let match, index
    while ((match = defaultTagRE.exec(text))) {
        index = match.index

        if (index > lastIndex) {
            tokens.push(JSON.stringify(text.slice(lastIndex, index)))
        }

        const exp = match[1].trim()
        tokens.push(`_s(${exp})`)
        lastIndex = index + match[0].length
    }

    if (lastIndex < text.length) {
        tokens.push(JSON.stringify(text.slice(lastIndex)))
    }
    return tokens.join('+');
}
```

我们使用一个 `tokens` 数组来存放解析结果，通过 `defaultTagRE` 来循环匹配该文本，如果是普通文本直接 `push` 到 `tokens` 数组中去，如果是表达式（{{item}}），则转化成"_s(${exp})"的形式。

举个例子，如果我们有这样一个文本。

```
<div>hello,{{name}}.</div>
```

最终得到 `tokens`。

```
tokens = ['hello,', _s(name), '.'];
```

最终通过 `join` 返回表达式。

```
'hello' + _s(name) + '.';
```

processIf与processFor

最后介绍一下如何处理“`v-if`”以及“`v-for`”这样的 Vue.js 的表达式的，这里我们只简单介绍两个示例中用到的表达式解析。

我们只需要在解析头标签的内容中加入这两个表达式的解析函数即可，在这时“`v-for`”之类指令已经在属性解析时存入了 `attrsMap` 中了。

```
if (html.match(startTagOpen)) {
    const startTagMatch = parseStartTag();
    const element = {
        type: 1,
        tag: startTagMatch.tagName,
        attrsList: startTagMatch.attrs,
        attrsMap: makeAttrsMap(startTagMatch.attrs),
        parent: currentParent,
        children: []
    }

    processIf(element);
    processFor(element);

    if(!root){
        root = element
    }

    if(currentParent){
        currentParent.children.push(element);
    }

    stack.push(element);
    currentParent = element;
    continue;
}
```

首先我们需要定义一个 `getAndRemoveAttr` 函数，用来从 `el` 的 `attrsMap` 属性或是 `attrsList` 属性中取出 `name` 对应值。

```
function getAndRemoveAttr (el, name) {
    let val
    if ((val = el.attrsMap[name]) != null) {
        const list = el.attrsList
        for (let i = 0, l = list.length; i < l; i++) {
            if (list[i].name === name) {
                list.splice(i, 1)
                break
            }
        }
    }
    return val
}
```

比如说解析示例的 div 标签属性。

```
getAndRemoveAttr(el, 'v-for');
```

可有得到“item in sz”。

有了这个函数这样我们就可以开始实现 `processFor` 与 `processIf` 了。

“v-for”会将指令解析成 `for` 属性以及 `alias` 属性，而“v-if”会将条件都存入 `ifConditions` 数组中。

```
function processFor (el) {
    let exp;
    if ((exp = getAndRemoveAttr(el, 'v-for'))) {
        const inMatch = exp.match(forAliasRE);
        el.for = inMatch[2].trim();
        el.alias = inMatch[1].trim();
    }
}

function processIf (el) {
    const exp = getAndRemoveAttr(el, 'v-if');
    if (exp) {
        el.if = exp;
        if (!el.ifConditions) {
            el.ifConditions = [];
        }
        el.ifConditions.push({
            exp: exp,
            block: el
        });
    }
}
```

到这里，我们已经把 `parse` 的过程介绍完了，接下来看一下 `optimize`。

optimize

`optimize` 主要作用就跟它的名字一样，用作「优化」。

这个涉及到后面要讲 `patch` 的过程，因为 `patch` 的过程实际上是将 VNode 节点进行一层一层的比对，然后将「差异」更新到视图上。那么一些静态节点是不会根据数据变化而产生变化的，这些节点我们没有比对的需求，是不是可以跳过这些静态节点的比对，从而节省一些性能呢?

那么我们就需要为静态的节点做上一些「标记」，在 `patch` 的时候我们就可以直接跳过这些被标记的节点的比对，从而达到「优化」的目的。

经过 `optimize` 这层的处理，每个节点会加上 `static` 属性，用来标记是否是静态的。

得到如下结果。

```
{
    'attrsMap': {
        ':class': 'c',
        'class': 'demo',
```

```
        'v-if': 'isShow'
    },
    'classBinding': 'c',
    'if': 'isShow',
    'ifConditions': [
        'exp': 'isShow'
    ],
    'staticClass': 'demo',
    'tag': 'div',
    /* 静态标志 */
    'static': false,
    'children': [
        {
            'attrsMap': {
                'v-for': "item in sz"
            },
            'static': false,
            'alias': "item",
            'for': 'sz',
            'forProcessed': true,
            'tag': 'span',
            'children': [
                {
                    'expression': '_s(item)',
                    'text': '{{item}}',
                    'static': false
                }
            ]
        }
    ]
}
```

我们用代码实现一下 `optimize` 函数。

isStatic

首先实现一个 `isStatic` 函数，传入一个 node 判断该 node 是否是静态节点。判断的标准是当 type 为 2（表达式节点）则是非静态节点，当 type 为 3（文本节点）的时候则是静态节点，当然，如果存在 `if` 或者 `for` 这样的条件的时候（表达式节点），也是非静态节点。

```
function isStatic (node) {
    if (node.type === 2) {
        return false
    }
    if (node.type === 3) {
        return true
    }
    return (!node.if && !node.for);
}
```

markStatic

`markStatic` 为所有的节点标记上 `static`，遍历所有节点通过 `isStatic` 来判断当前节点是否是静态节点，此外，会遍历当前节点的所有子节点，如果子节点是非静态节点，那么当前节点也是非静态节点。

```
function markStatic (node) {
    node.static = isStatic(node);
    if (node.type === 1) {
        for (let i = 0, l = node.children.length; i < l; i++) {
            const child = node.children[i];
            markStatic(child);
            if (!child.static) {
                node.static = false;
            }
        }
    }
}
```

markStaticRoots

接下来是 `markStaticRoots` 函数，用来标记 `staticRoot`（静态根）。这个函数实现比较简单，简单来将就是如果当前节点是静态节点，同时满足该节点并不是只有一个文本节点左右子节点（作者认为这种情况的优化消耗会大于收益）时，标记 `staticRoot` 为 true，否则为 false。

```
function markStaticRoots (node) {
    if (node.type === 1) {
        if (node.static && node.children.length && !(
        node.children.length === 1 &&
        node.children[0].type === 3
        )) {
            node.staticRoot = true;
            return;
        } else {
            node.staticRoot = false;
        }
    }
}
```

optimize

有了以上的函数，就可以实现 `optimize` 了。

```
function optimize (rootAst) {
    markStatic(rootAst);
    markStaticRoots(rootAst);
}
```

generate

`generate` 会将 AST 转化成 render funtion 字符串，最终得到 render 的字符串以及 staticRenderFns 字符串。

首先带大家感受一下真实的 Vue.js 编译得到的结果。

```
with(this){
    return (isShow) ?
    _c(
        'div',
        {
            staticClass: "demo",
            class: c
        },
        _l(
            (sz),
            function(item){
                return _c('span',[_v(_s(item))])
            }
        )
    )
    : _e()
}
```

看到这里可能会纳闷了，这些 `_c`，`_l` 到底是什么？其实他们是 Vue.js 对一些函数的简写，比如说 `_c` 对应的是 `createElement` 这个函数。没关系，我们把它用 VNode 的形式写出来就会明白了，这个对接上一章写的 VNode 函数。

首先是第一层 div 节点。

```
render () {
    return isShow ? (new VNode('div', {
        'staticClass': 'demo',
        'class': c
    }, [ /*这里还有子节点*/ ])) : createEmptyVNode();
}
```

然后我们在 `children` 中加上第二层 span 及其子文本节点节点。

```
/* 渲染v-for列表 */
function renderList (val, render) {
    let ret = new Array(val.length);
    for (i = 0, l = val.length; i < l; i++) {
        ret[i] = render(val[i], i);
    }
}

render () {
    return isShow ? (new VNode('div', {
        'staticClass': 'demo',
        'class': c
    },
        /* begin */
```

```
        renderList(sz, (item) => {
            return new VNode('span', {}, [
                createTextVNode(item);
            ]);
        })
        /* end */
    )) : createEmptyVNode();
}
```

那我们如何来实现一个 `generate` 呢?

genIf

首先实现一个处理 `if` 条件的 `genIf` 函数。

```
function genIf (el) {
    el.ifProcessed = true;
    if (!el.ifConditions.length) {
        return '_e()';
    }
    return `(${el.ifConditions[0].exp})?${genElement(el.ifConditions[0].block)}:
_e()`
}
```

genFor

然后是处理 `for` 循环的函数。

```
function genFor (el) {
    el.forProcessed = true;

    const exp = el.for;
    const alias = el.alias;
    const iterator1 = el.iterator1 ? `,${el.iterator1}` : '';
    const iterator2 = el.iterator2 ? `,${el.iterator2}` : '';

    return `_l((${exp}),` +
        `function(${alias}${iterator1}${iterator2}){` +
        `return ${genElement(el)}` +
    '})';
}
```

genText

处理文本节点的函数。

```
function genText (el) {
    return `_v(${el.expression})`;
}
```

genElement

接下来实现一下 `genElement`，这是一个处理节点的函数，因为它依赖 `genChildren` 以及g `enNode`，所以这三个函数放在一起讲。

genElement会根据当前节点是否有 `if` 或者 `for` 标记然后判断是否要用 `genIf` 或者 `genFor` 处理，否则通过 `genChildren` 处理子节点，同时得到 `staticClass`、`class` 等属性。

`genChildren` 比较简单，遍历所有子节点，通过 `genNode` 处理后用“，”隔开拼接成字符串。

`genNode` 则是根据 `type` 来判断该节点是用文本节点 `genText` 还是标签节点 `genElement` 来处理。

```
function genNode (el) {
    if (el.type === 1) {
        return genElement(el);
    } else {
        return genText(el);
    }
}

function genChildren (el) {
    const children = el.children;

    if (children && children.length > 0) {
        return `${children.map(genNode).join(',')}`;
    }
}

function genElement (el) {
    if (el.if && !el.ifProcessed) {
        return genIf(el);
    } else if (el.for && !el.forProcessed) {
        return genFor(el);
    } else {
        const children = genChildren(el);
        let code;
        code = `_c('${el.tag},'{
            staticClass: ${el.attrsMap && el.attrsMap[':class']},
            class: ${el.attrsMap && el.attrsMap['class']},
        }${
            children ? `,${children}` : ''
        })`
        return code;
    }
}
```

generate

最后我们使用上面的函数来实现 `generate`，其实很简单，我们只需要将整个 AST 传入后判断是否为空，为空则返回一个 div 标签，否则通过 `generate` 来处理。

```
function generate (rootAst) {
    const code = rootAst ? genElement(rootAst) : '_c("div")'
    return {
        render: `with(this){return ${code}}`,
    }
}
```

经历过这些过程以后，我们已经把 template 顺利转成了 render function 了，接下来我们将介绍 `patch` 的过程，来看一下具体 VNode 节点如何进行差异的比对。

注：本节代码参考《template 模板是怎样通过 Compile 编译的》。

数据状态更新时的差异 diff 及 patch 机制

数据更新视图

之前讲到，在对 `model` 进行操作对时候，会触发对应 `Dep` 中的 `Watcher` 对象。`Watcher` 对象会调用对应的 `update` 来修改视图。最终是将新产生的 VNode 节点与老 VNode 进行一个 `patch` 的过程，比对得出「差异」，最终将这些「差异」更新到视图上。

这一章就来介绍一下这个 `patch` 的过程，因为 `patch` 过程本身比较复杂，这一章的内容会比较多，但是不要害怕，我们逐块代码去看，一定可以理解。

跨平台

因为使用了 Virtual DOM 的原因，Vue.js具有了跨平台的能力，Virtual DOM 终归只是一些 JavaScript 对象罢了，那么最终是如何调用不同平台的 API 的呢?

这就需要依赖一层适配层了，将不同平台的 API 封装在内，以同样的接口对外提供。

```
const nodeOps = {
    setTextContent (text) {
        if (platform === 'weex') {
            node.parentNode.setAttr('value', text);
        } else if (platform === 'web') {
            node.textContent = text;
        }
    },
    parentNode () {
        //......
    },
    removeChild () {
        //......
    },
    nextSibling () {
        //......
    },
    insertBefore () {
        //......
```

```
    }
}
```

举个例子，现在我们有上述一个 `nodeOps` 对象做适配，根据 platform 区分不同平台来执行当前平台对应的API，而对外则是提供了一致的接口，供 Virtual DOM 来调用。

一些API

接下来我们来介绍其他的一些 API，这些API在下面 `patch` 的过程中会被用到，他们最终都会调用 `nodeOps` 中的相应函数来操作平台。

`insert` 用来在 `parent` 这个父节点下插入一个子节点，如果指定了 `ref` 则插入到 `ref` 这个子节点前面。

```
function insert (parent, elm, ref) {
    if (parent) {
        if (ref) {
            if (ref.parentNode === parent) {
                nodeOps.insertBefore(parent, elm, ref);
            }
        } else {
            nodeOps.appendChild(parent, elm)
        }
    }
}
```

`createElm` 用来新建一个节点， `tag` 存在创建一个标签节点，否则创建一个文本节点。

```
function createElm (vnode, parentElm, refElm) {
    if (vnode.tag) {
        insert(parentElm, nodeOps.createElement(vnode.tag), refElm);
    } else {
        insert(parentElm, nodeOps.createTextNode(vnode.text), refElm);
    }
}
```

`addVnodes` 用来批量调用 `createElm` 新建节点。

```
function addVnodes (parentElm, refElm, vnodes, startIdx, endIdx) {
    for (; startIdx <= endIdx; ++startIdx) {
        createElm(vnodes[startIdx], parentElm, refElm);
    }
}
```

`removeNode` 用来移除一个节点。

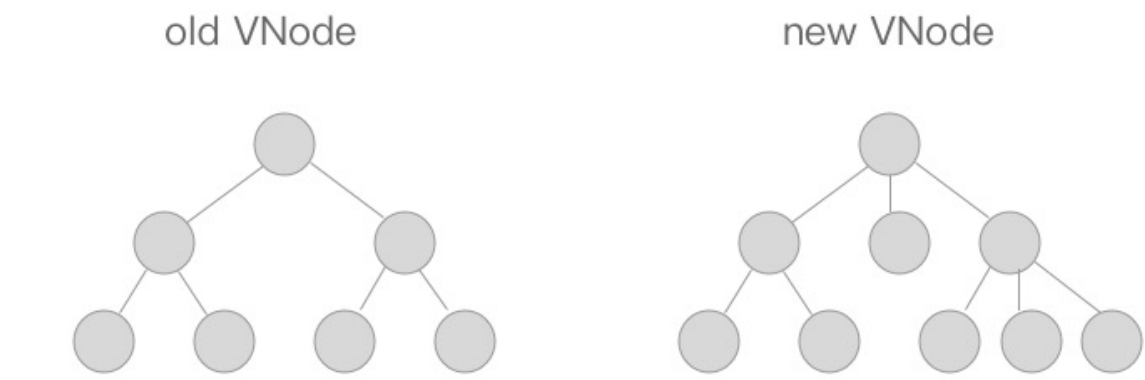
```
function removeNode (el) {
    const parent = nodeOps.parentNode(el);
    if (parent) {
        nodeOps.removeChild(parent, el);
    }
}
```

`removeVnodes` 会批量调用 `removeNode` 移除节点。

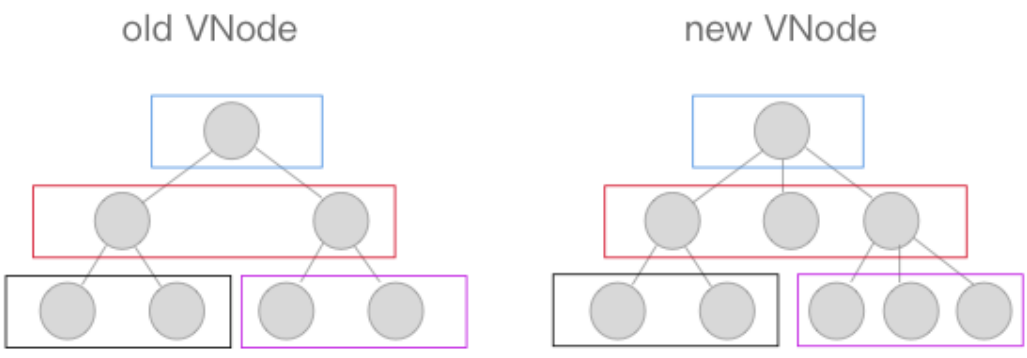
```
function removeVnodes (parentElm, vnodes, startIdx, endIdx) {
    for (; startIdx <= endIdx; ++startIdx) {
        const ch = vnodes[startIdx]
        if (ch) {
            removeNode(ch.elm);
        }
    }
}
```

patch

首先说一下 `patch` 的核心 diff 算法，我们用 diff 算法可以比对出两颗树的「差异」，我们来看一下，假设我们现在有如下两颗树，它们分别是新老 VNode 节点，这时候到了 `patch` 的过程，我们需要将他们进行比对。



diff 算法是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，如下图。



。

这张图中的相同颜色的方块中的节点会进行比对，比对得到「**差异**」后将这些「**差异**」更新到视图上。因为只进行同层级的比对，所以十分高效。

`patch` 的过程相当复杂，我们先用简单的代码来看一下。

```
function patch (oldVnode, vnode, parentElm) {
    if (!oldVnode) {
        addVnodes(parentElm, null, vnode, 0, vnode.length - 1);
    } else if (!vnode) {
        removeVnodes(parentElm, oldVnode, 0, oldVnode.length - 1);
    } else {
        if (sameVnode(oldVNode, vnode)) {
            patchVnode(oldVNode, vnode);
        } else {
            removeVnodes(parentElm, oldVnode, 0, oldVnode.length - 1);
            addVnodes(parentElm, null, vnode, 0, vnode.length - 1);
        }
    }
}
```

因为 `patch` 的主要功能是比对两个 VNode 节点，将「差异」更新到视图上，所以入参有新老两个 VNode 以及父节点的 element 。我们来逐步捋一下逻辑， `addVnodes` 、 `removeVnodes` 等函数后面会讲。

首先在 `oldVnode` （老 VNode 节点）不存在的时候，相当于新的 VNode 替代原本没有的节点，所以直接用 `addVnodes` 将这些节点批量添加到 `parentElm` 上。

```
if (!oldVnode) {
    addVnodes(parentElm, null, vnode, 0, vnode.length - 1);
}
```

然后同理，在 `vnode` （新 VNode 节点）不存在的时候，相当于要把老的节点删除，所以直接使用 `removeVnodes` 进行批量的节点删除即可。

```
else if (!vnode) {
    removeVnodes(parentElm, oldVnode, 0, oldVnode.length - 1);
}
```

最后一种情况，当 `oldVNode` 与 `vnode` 都存在的时候，需要判断它们是否属于 `sameVnode` （相同的节点）。如果是则进行patchVnode（比对 VNode ）操作，否则删除老节点，增加新节点。

```
if (sameVnode(oldVNode, vnode)) {
    patchVnode(oldVNode, vnode);
} else {
    removeVnodes(parentElm, oldVnode, 0, oldVnode.length - 1);
    addVnodes(parentElm, null, vnode, 0, vnode.length - 1);
}
```

sameVnode

上面这些比较好理解，下面我们来看看什么情况下两个 VNode 会属于 `sameVnode` （相同的节点）呢?

```
function sameVnode () {
    return (
        a.key === b.key &&
        a.tag === b.tag &&
        a.isComment === b.isComment &&
        (!!a.data) === (!!b.data) &&
        sameInputType(a, b)
    )
}

function sameInputType (a, b) {
    if (a.tag !== 'input') return true
    let i
    const typeA = (i = a.data) && (i = i.attrs) && i.type
    const typeB = (i = b.data) && (i = i.attrs) && i.type
    return typeA === typeB
}
```

`sameVnode` 其实很简单，只有当 `key`、 `tag`、 `isComment` （是否为注释节点）、 `data` 同时定义（或不定义），同时满足当标签类型为 input 的时候 type 相同（某些浏览器不支持动态修改 类型，所以他们被视为不同类型）即可。

patchVnode

之前patch的过程还剩下 `patchVnode` 这个函数没有讲，这也是最复杂的一个，我们现在来看一下。因为这个函数是在符合 `sameVnode` 的条件下触发的，所以会进行「**比对**」。

```
function patchVnode (oldVnode, vnode) {
    if (oldVnode === vnode) {
        return;
    }

    if (vnode.isStatic && oldVnode.isStatic && vnode.key === oldVnode.key) {
        vnode.elm = oldVnode.elm;
        vnode.componentInstance = oldVnode.componentInstance;
        return;
    }

    const elm = vnode.elm = oldVnode.elm;
    const oldCh = oldVnode.children;
    const ch = vnode.children;

    if (vnode.text) {
        nodeOps.setTextContent(elm, vnode.text);
    } else {
        if (oldCh && ch && (oldCh !== ch)) {
            updateChildren(elm, oldCh, ch);
        } else if (ch) {
            if (oldVnode.text) nodeOps.setTextContent(elm, '');
            addVnodes(elm, null, ch, 0, ch.length - 1);
        } else if (oldCh) {
```

```
            removeVnodes(elm, oldCh, 0, oldCh.length - 1)
        } else if (oldVnode.text) {
            nodeOps.setTextContent(elm, '')
        }
    }
}
```

首先在新老 VNode 节点相同的情况下，就不需要做任何改变了，直接 return 掉。

```
if (oldVnode === vnode) {
    return;
}
```

下面的这种情况也比较简单，在当新老 VNode 节点都是 `isStatic`（静态的），并且 `key` 相同时，只要将 `componentInstance` 与 `elm` 从老 VNode 节点"拿过来"即可。这里的 `isStatic` 也就是前面提到过的「编译」的时候会将静态节点标记出来，这样就可以跳过比对的过程。

```
if (vnode.isStatic && oldVnode.isStatic && vnode.key === oldVnode.key) {
    vnode.elm = oldVnode.elm;
    vnode.componentInstance = oldVnode.componentInstance;
    return;
}
```

接下来，当新 VNode 节点是文本节点的时候，直接用 `setTextContent` 来设置 text，这里的 `nodeOps` 是一个适配层，根据不同平台提供不同的操作平台 DOM 的方法，实现跨平台。

```
if (vnode.text) {
    nodeOps.setTextContent(elm, vnode.text);
}
```

当新 VNode 节点是非文本节点当时候，需要分几种情况。

- `oldCh` 与 `ch` 都存在且不相同时，使用 `updateChildren` 函数来更新子节点，这个后面重点讲。
- 如果只有 `ch` 存在的时候，如果老节点是文本节点则先将节点的文本清除，然后将 `ch` 批量插入插入到节点elm下。
- 同理当只有 `oldch` 存在时，说明需要将老节点通过 `removeVnodes` 全部清除。
- 最后一种情况是当只有老节点是文本节点的时候，清除其节点文本内容。

```
if (oldCh && ch && (oldCh !== ch)) {
    updateChildren(elm, oldCh, ch);
} else if (ch) {
    if (oldVnode.text) nodeOps.setTextContent(elm, '');
    addVnodes(elm, null, ch, 0, ch.length - 1);
} else if (oldCh) {
    removeVnodes(elm, oldCh, 0, oldCh.length - 1)
} else if (oldVnode.text) {
    nodeOps.setTextContent(elm, '')
}
```

updateChildren

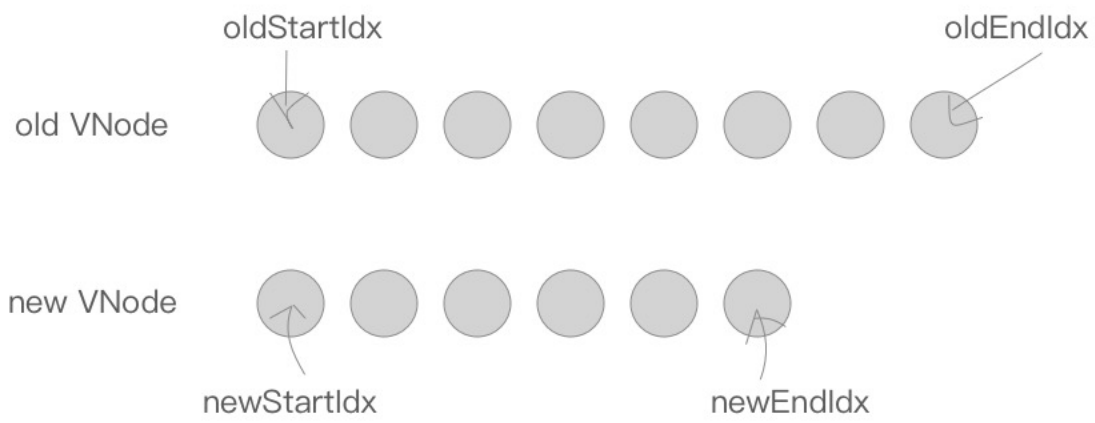
接下来就要讲一下 `updateChildren` 函数了。

```
function updateChildren (parentElm, oldCh, newCh) {
    let oldStartIdx = 0;
    let newStartIdx = 0;
    let oldEndIdx = oldCh.length - 1;
    let oldStartVnode = oldCh[0];
    let oldEndVnode = oldCh[oldEndIdx];
    let newEndIdx = newCh.length - 1;
    let newStartVnode = newCh[0];
    let newEndVnode = newCh[newEndIdx];
    let oldKeyToIdx, idxInOld, elmToMove, refElm;

    while (oldStartIdx <= oldEndIdx && newStartIdx <= newEndIdx) {
        if (!oldStartVnode) {
            oldStartVnode = oldCh[++oldStartIdx];
        } else if (!oldEndVnode) {
            oldEndVnode = oldCh[--oldEndIdx];
        } else if (sameVnode(oldStartVnode, newStartVnode)) {
            patchVnode(oldStartVnode, newStartVnode);
            oldStartVnode = oldCh[++oldStartIdx];
            newStartVnode = newCh[++newStartIdx];
        } else if (sameVnode(oldEndVnode, newEndVnode)) {
            patchVnode(oldEndVnode, newEndVnode);
            oldEndVnode = oldCh[--oldEndIdx];
            newEndVnode = newCh[--newEndIdx];
        } else if (sameVnode(oldStartVnode, newEndVnode)) {
            patchVnode(oldStartVnode, newEndVnode);
            nodeOps.insertBefore(parentElm, oldStartVnode.elm,
nodeOps.nextSibling(oldEndVnode.elm));
            oldStartVnode = oldCh[++oldStartIdx];
            newEndVnode = newCh[--newEndIdx];
        } else if (sameVnode(oldEndVnode, newStartVnode)) {
            patchVnode(oldEndVnode, newStartVnode);
            nodeOps.insertBefore(parentElm, oldEndVnode.elm, oldStartVnode.elm);
            oldEndVnode = oldCh[--oldEndIdx];
            newStartVnode = newCh[++newStartIdx];
        } else {
            let elmToMove = oldCh[idxInOld];
            if (!oldKeyToIdx) oldKeyToIdx = createKeyToOldIdx(oldCh,
oldStartIdx, oldEndIdx);
            idxInOld = newStartVnode.key ? oldKeyToIdx[newStartVnode.key] :
null;
            if (!idxInOld) {
                createElm(newStartVnode, parentElm);
                newStartVnode = newCh[++newStartIdx];
            } else {
                elmToMove = oldCh[idxInOld];
                if (sameVnode(elmToMove, newStartVnode)) {
                    patchVnode(elmToMove, newStartVnode);
                    oldCh[idxInOld] = undefined;
                    nodeOps.insertBefore(parentElm, newStartVnode.elm,
oldStartVnode.elm);
                    newStartVnode = newCh[++newStartIdx];
```

```
                } else {
                    createElm(newStartVnode, parentElm);
                    newStartVnode = newCh[++newStartIdx];
                }
            }
        }
    }

    if (oldStartIdx > oldEndIdx) {
        refElm = (newCh[newEndIdx + 1]) ? newCh[newEndIdx + 1].elm : null;
        addVnodes(parentElm, refElm, newCh, newStartIdx, newEndIdx);
    } else if (newStartIdx > newEndIdx) {
        removeVnodes(parentElm, oldCh, oldStartIdx, oldEndIdx);
    }
}
```

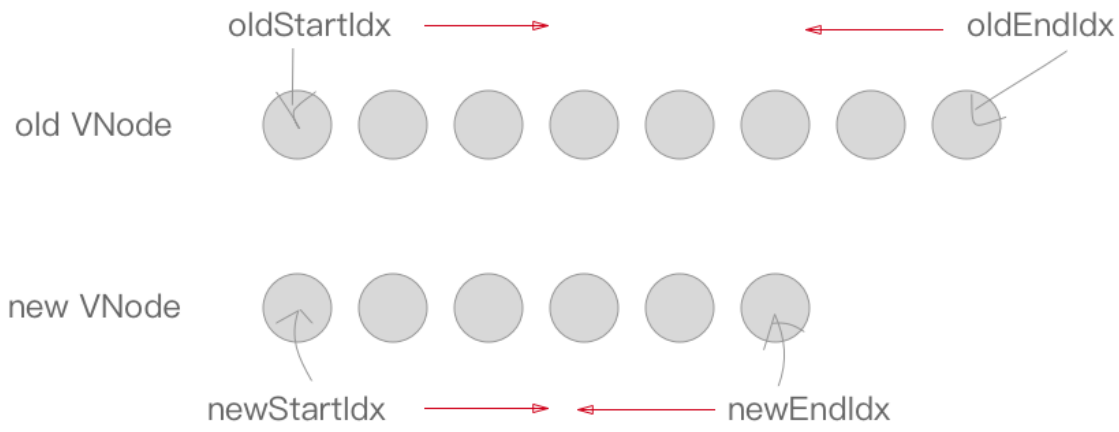
看到代码那么多先不要着急，我们还是一点一点地讲解。

首先我们定义 `oldStartIdx`、`newStartIdx`、`oldEndIdx` 以及 `newEndIdx` 分别是新老两个 VNode 的两边的索引，同时 `oldStartVnode`、`newStartVnode`、`oldEndVnode` 以及 `newEndVnode` 分别指向这几个索引对应的 VNode 节点。



接下来是一个 `while` 循环，在这过程中，`oldStartIdx`、`newStartIdx`、`oldEndIdx` 以及 `newEndIdx` 会逐渐向中间靠拢。

```
while (oldStartIdx <= oldEndIdx && newStartIdx <= newEndIdx)
```



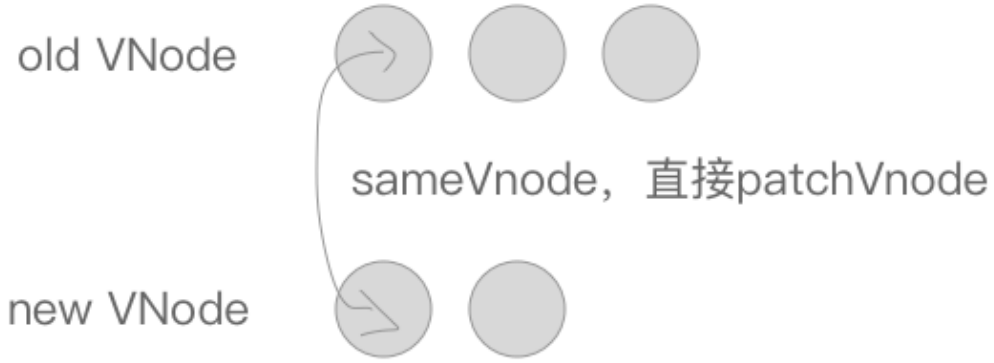
首先当 `oldStartVnode` 或者 `oldEndVnode` 不存在的时候，`oldStartIdx` 与 `oldEndIdx` 继续向中间靠拢，并更新对应的 `oldStartVnode` 与 `oldEndVnode` 的指向（注：下面讲到的 `oldStartIdx`、`newStartIdx`、`oldEndIdx` 以及 `newEndIdx` 移动都会伴随着 `oldStartVnode`、`newStartVnode`、`oldEndVnode` 以及 `newEndVnode` 的指向的变化，之后的部分只会讲 `Idx` 的移动）。

```
if (!oldStartVnode) {
    oldStartVnode = oldCh[++oldStartIdx];
} else if (!oldEndVnode) {
    oldEndVnode = oldCh[--oldEndIdx];
}
```

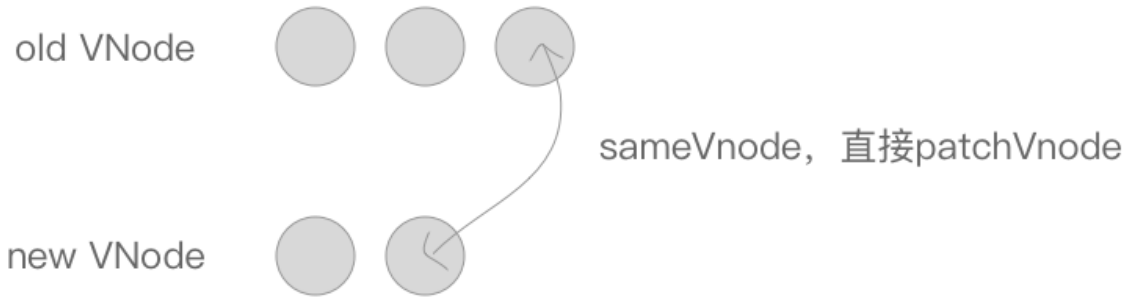
接下来这一块，是将 `oldStartIdx`、`newStartIdx`、`oldEndIdx` 以及 `newEndIdx` 两两比对的过程，一共会出现 2*2=4 种情况。

```
 else if (sameVnode(oldStartVnode, newStartVnode)) {
    patchVnode(oldStartVnode, newStartVnode);
    oldStartVnode = oldCh[++oldStartIdx];
    newStartVnode = newCh[++newStartIdx];
} else if (sameVnode(oldEndVnode, newEndVnode)) {
    patchVnode(oldEndVnode, newEndVnode);
    oldEndVnode = oldCh[--oldEndIdx];
    newEndVnode = newCh[--newEndIdx];
} else if (sameVnode(oldStartVnode, newEndVnode)) {
    patchVnode(oldStartVnode, newEndVnode);
    nodeOps.insertBefore(parentElm, oldStartVnode.elm,
nodeOps.nextSibling(oldEndVnode.elm));
    oldStartVnode = oldCh[++oldStartIdx];
    newEndVnode = newCh[--newEndIdx];
} else if (sameVnode(oldEndVnode, newStartVnode)) {
    patchVnode(oldEndVnode, newStartVnode);
    nodeOps.insertBefore(parentElm, oldEndVnode.elm, oldStartVnode.elm);
    oldEndVnode = oldCh[--oldEndIdx];
    newStartVnode = newCh[++newStartIdx];
}
```

首先是 `oldStartVnode` 与 `newStartVnode` 符合 `sameVnode` 时，说明老 VNode 节点的头部与新 VNode 节点的头部是相同的 VNode 节点，直接进行 `patchVnode`，同时 `oldStartIdx` 与 `newStartIdx` 向后移动一位。

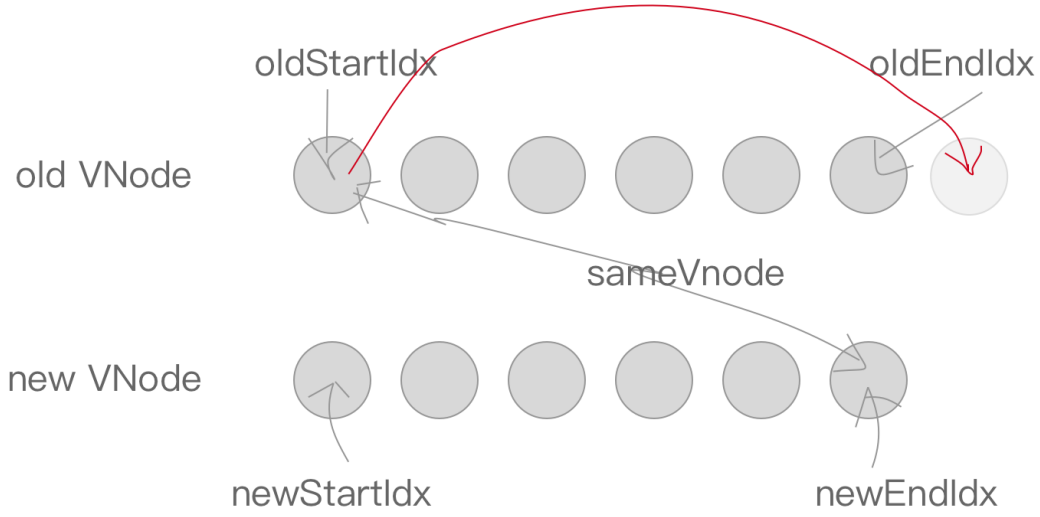


其次是 `oldEndVnode` 与 `newEndVnode` 符合 `sameVnode`，也就是两个 VNode 的结尾是相同的 VNode，同样进行 `patchVnode` 操作并将 `oldEndVnode` 与 `newEndVnode` 向前移动一位。

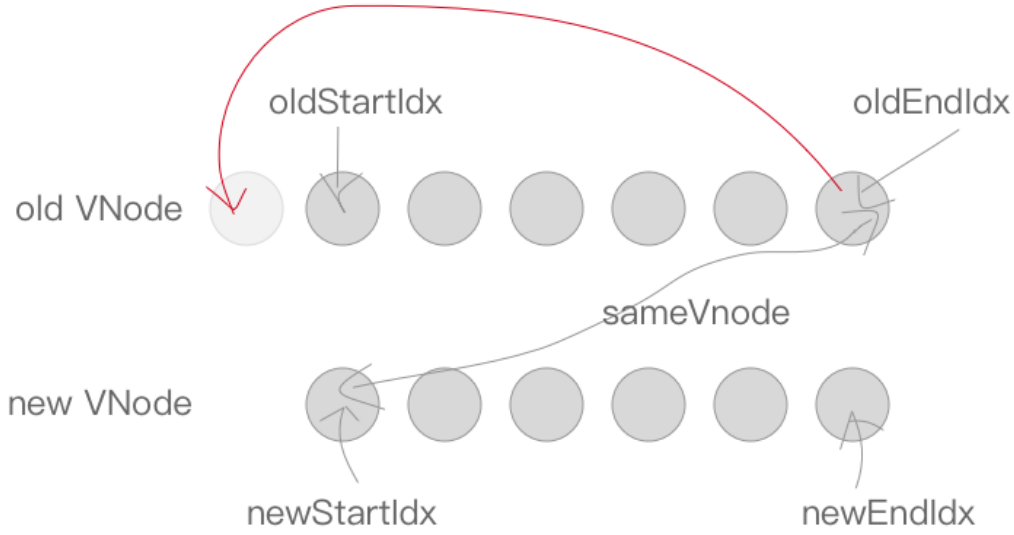


接下来是两种交叉的情况。

先是 `oldStartVnode` 与 `newEndVnode` 符合 `sameVnode` 的时候，也就是老 VNode 节点的头部与新 VNode 节点的尾部是同一节点的时候，将 `oldStartVnode.elm` 这个节点直接移动到 `oldEndVnode.elm` 这个节点的后面即可。然后 `oldStartIdx` 向后移动一位，`newEndIdx` 向前移动一位。



同理，`oldEndVnode` 与 `newStartVnode` 符合 `sameVnode` 时，也就是老 VNode 节点的尾部与新 VNode 节点的头部是同一节点的时候，将 `oldEndVnode.elm` 插入到 `oldStartVnode.elm` 前面。同样的，`oldEndIdx` 向前移动一位，`newStartIdx` 向后移动一位。



最后是当以上情况都不符合的时候，这种情况怎么处理呢?

```
else {
    let elmToMove = oldCh[idxInOld];
    if (!oldKeyToIdx) oldKeyToIdx = createKeyToOldIdx(oldCh, oldStartIdx,
oldEndIdx);
    idxInOld = newStartVnode.key ? oldKeyToIdx[newStartVnode.key] : null;
    if (!idxInOld) {
        createElm(newStartVnode, parentElm);
        newStartVnode = newCh[++newStartIdx];
    } else {
        elmToMove = oldCh[idxInOld];
        if (sameVnode(elmToMove, newStartVnode)) {
            patchVnode(elmToMove, newStartVnode);
            oldCh[idxInOld] = undefined;
            nodeOps.insertBefore(parentElm, newStartVnode.elm,
oldStartVnode.elm);
            newStartVnode = newCh[++newStartIdx];
        } else {
            createElm(newStartVnode, parentElm);
            newStartVnode = newCh[++newStartIdx];
        }
    }
}

function createKeyToOldIdx (children, beginIdx, endIdx) {
    let i, key
    const map = {}
    for (i = beginIdx; i <= endIdx; ++i) {
        key = children[i].key
        if (isDef(key)) map[key] = i
    }
    return map
}
```

`createKeyToOldIdx` 的作用是产生 `key` 与 `index` 索引对应的一个 map 表。比如说：

```
[
    {xx: xx, key: 'key0'},
    {xx: xx, key: 'key1'},
    {xx: xx, key: 'key2'}
]
```

在经过 `createKeyToOldIdx` 转化以后会变成：

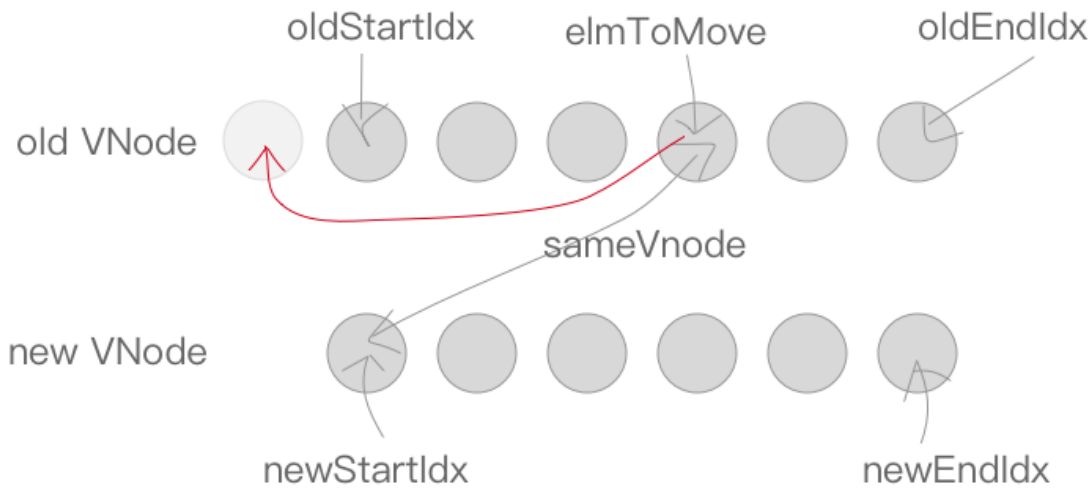
```
{
    key0: 0,
    key1: 1,
    key2: 2
}
```

我们可以根据某一个 key 的值，快速地从 `oldKeyToIdx`（`createKeyToOldIdx` 的返回值）中获取相同 key 的节点的索引 `idxInOld`，然后找到相同的节点。

如果没有找到相同的节点，则通过 `createElm` 创建一个新节点，并将 `newStartIdx` 向后移动一位。

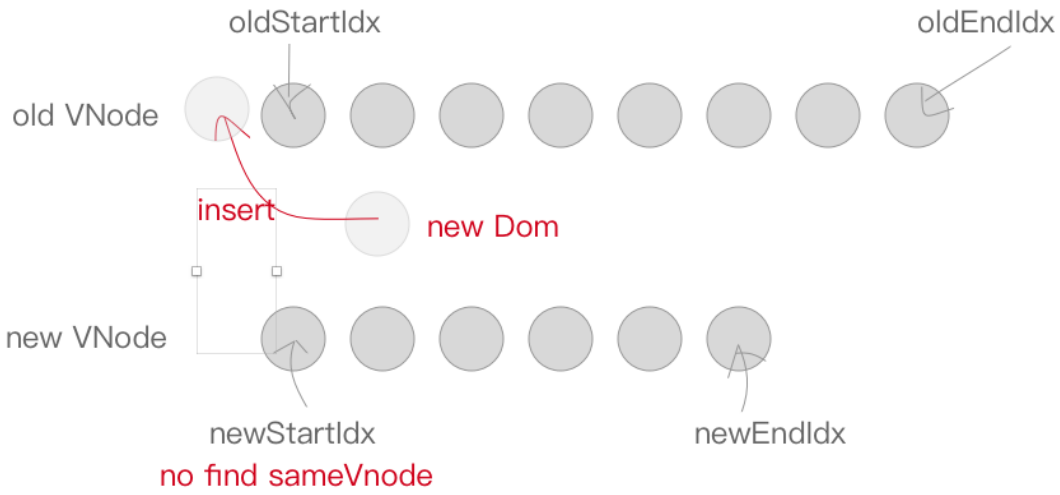
```
if (!idxInOld) {
    createElm(newStartVnode, parentElm);
    newStartVnode = newCh[++newStartIdx];
}
```

否则如果找到了节点，同时它符合 `sameVnode`，则将这两个节点进行 `patchVnode`，将该位置的老节点赋值 undefined（之后如果还有新节点与该节点key相同可以检测出来提示已有重复的 key），同时将 `newStartVnode.elm` 插入到 `oldStartVnode.elm` 的前面。同理，`newStartIdx` 往后移动一位。



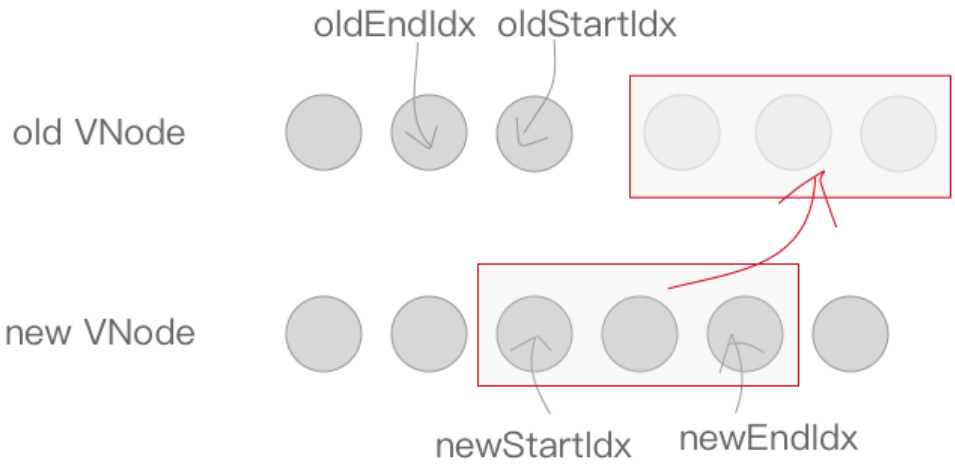
```
else {
    elmToMove = oldCh[idxInOld];
    if (sameVnode(elmToMove, newStartVnode)) {
        patchVnode(elmToMove, newStartVnode);
        oldCh[idxInOld] = undefined;
        nodeOps.insertBefore(parentElm, newStartVnode.elm, oldStartVnode.elm);
        newStartVnode = newCh[++newStartIdx];
    }
}
```

如果不符合 `sameVnode`，只能创建一个新节点插入到 `parentElm` 的子节点中，`newStartIdx` 往后移动一位。

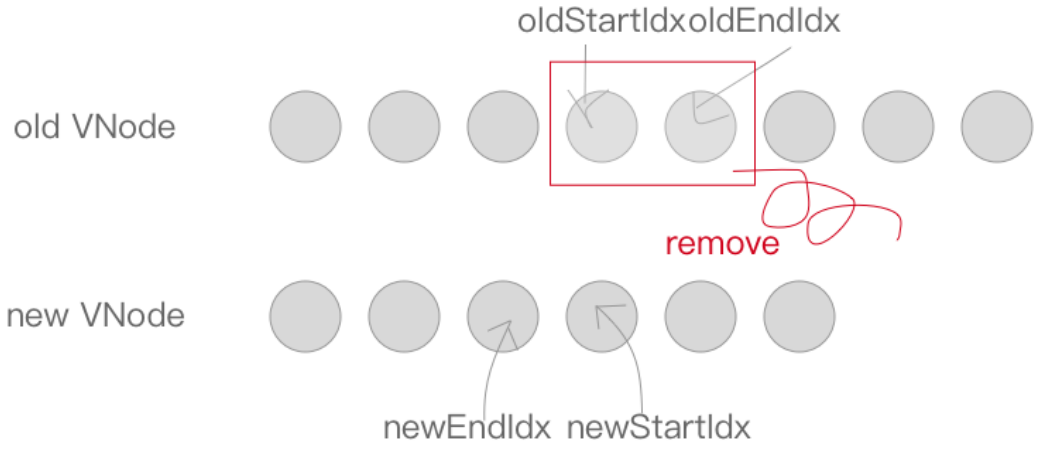



```
else {
    createElm(newStartVnode, parentElm);
    newStartVnode = newCh[++newStartIdx];
}
```

最后一步就很容易啦，当 `while` 循环结束以后，如果 `oldStartIdx > oldEndIdx`，说明老节点比对完了，但是新节点还有多的，需要将新节点插入到真实 DOM 中去，调用 `addVnodes` 将这些节点插入即可。



同理，如果满足 `newStartIdx > newEndIdx` 条件，说明新节点比对完了，老节点还有多，将这些无用的老节点通过 `removeVnodes` 批量删除即可。




```
if (oldStartIdx > oldEndIdx) {
    refElm = (newCh[newEndIdx + 1]) ? newCh[newEndIdx + 1].elm : null;
    addVnodes(parentElm, refElm, newCh, newStartIdx, newEndIdx);
} else if (newStartIdx > newEndIdx) {
    removeVnodes(parentElm, oldCh, oldStartIdx, oldEndIdx);
}
```

到这里，比对的核心实现已经讲完了，这部分比较复杂，不过仔细地梳理一下比对的过程，相信一定能够理解得更加透彻的。

注：本节代码参考《数据状态更新时的差异 diff 及 patch 机制》。

批量异步更新策略及 nextTick 原理

为什么要异步更新

通过前面几个章节我们介绍，相信大家已经明白了 Vue.js 是如何在我们修改 `data` 中的数据后修改视图了。简单回顾一下，这里面其实就是一个“`setter -> Dep -> Watcher -> patch -> 视图`”的过程。

假设我们有如下这么一种情况。

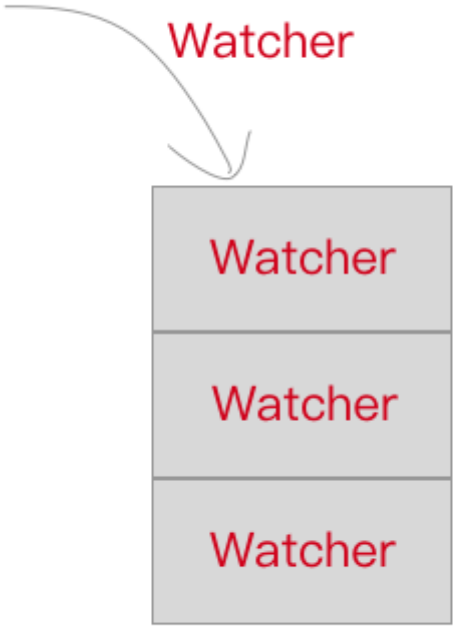
```
<template>
  <div>
    <div>{{number}}</div>
    <div @click="handleClick">click</div>
  </div>
</template>
```

```
export default {
    data () {
        return {
            number: 0
        };
    },
    methods: {
        handleClick () {
            for(let i = 0; i < 1000; i++) {
                this.number++;
            }
        }
    }
}
```

当我们按下 click 按钮的时候，`number` 会被循环增加1000次。

那么按照之前的理解，每次 `number` 被 +1 的时候，都会触发 `number` 的 `setter` 方法，从而根据上面的流程一直跑下来最后修改真实 DOM。那么在这个过程中，DOM 会被更新 1000 次！太可怕了。

Vue.js 肯定不会以如此低效的方法来处理。Vue.js在默认情况下，每次触发某个数据的 `setter` 方法后，对应的 `watcher` 对象其实会被 `push` 进一个队列 `queue` 中，在下一个 tick 的时候将这个队列 `queue` 全部拿出来 `run`（`watcher` 对象的一个方法，用来触发 `patch` 操作） 一遍。



那么什么是下一个 tick 呢?

nextTick

Vue.js 实现了一个 `nextTick` 函数，传入一个 `cb` ，这个 `cb` 会被存储到一个队列中，在下一个 tick 时触发队列中的所有 `cb` 事件。

因为目前浏览器平台并没有实现 `nextTick` 方法，所以 Vue.js 源码中分别用 `Promise` 、`setTimeout` 、`setImmediate` 等方式在 microtask（或是task）中创建一个事件，目的是在当前调用栈执行完毕以后（不一定立即）才会去执行这个事件。

笔者用 `setTimeout` 来模拟这个方法，当然，真实的源码中会更加复杂，笔者在小册中只讲原理，有兴趣了解源码中 `nextTick` 的具体实现的同学可以参考next-tick。

首先定义一个 `callbacks` 数组用来存储 `nextTick` ，在下一个 tick 处理这些回调函数之前，所有的 `cb` 都会被存在这个 `callbacks` 数组中。 `pending` 是一个标记位，代表一个等待的状态。

`setTimeout` 会在 task 中创建一个事件 `flushCallbacks` ， `flushCallbacks` 则会在执行时将 `callbacks` 中的所有 `cb` 依次执行。

```
let callbacks = [];
let pending = false;

function nextTick (cb) {
    callbacks.push(cb);

    if (!pending) {
        pending = true;
        setTimeout(flushCallbacks, 0);
    }
}

function flushCallbacks () {
    pending = false;
```

```
    const copies = callbacks.slice(0);
    callbacks.length = 0;
    for (let i = 0; i < copies.length; i++) {
        copies[i]();
    }
}
```

再写 Watcher

第一个例子中，当我们将 `number` 增加 1000 次时，先将对应的 `Watcher` 对象给 `push` 进一个队列 `queue` 中去，等下一个 tick 的时候再去执行，这样做是对的。但是有没有发现，另一个问题出现了？

因为 `number` 执行 ++ 操作以后对应的 `Watcher` 对象都是同一个，我们并不需要在下一个 tick 的时候执行 1000 个同样的 `Watcher` 对象去修改界面，而是只需要执行一个 `Watcher` 对象，使其将界面上的 0 变成 1000 即可。

那么，我们就需要执行一个过滤的操作，同一个的 `Watcher` 在同一个 tick 的时候应该只被执行一次，也就是说队列 `queue` 中不应该出现重复的 `Watcher` 对象。

那么我们给 `Watcher` 对象起个名字吧～用 `id` 来标记每一个 `Watcher` 对象，让他们看起来"不太一样"。

实现 `update` 方法，在修改数据后由 `Dep` 来调用， 而 `run` 方法才是真正的触发 `patch` 更新视图的方法。

```
let uid = 0;

class Watcher {
    constructor () {
        this.id = ++uid;
    }

    update () {
        console.log('watch' + this.id + ' update');
        queueWatcher(this);
    }

    run () {
        console.log('watch' + this.id + '视图更新啦～');
    }
}
```

queueWatcher

不知道大家注意到了没有？笔者已经将 `Watcher` 的 `update` 中的实现改成了

```
queueWatcher(this);
```

将 `Watcher` 对象自身传递给 `queueWatcher` 方法。

我们来实现一下 `queueWatcher` 方法。

```
let has = {};
let queue = [];
let waiting = false;

function queueWatcher(watcher) {
    const id = watcher.id;
    if (has[id] == null) {
        has[id] = true;
        queue.push(watcher);

        if (!waiting) {
            waiting = true;
            nextTick(flushSchedulerQueue);
        }
    }
}
```

我们使用一个叫做 `has` 的 map，里面存放 id -> true (false) 的形式，用来判断是否已经存在相同的 `Watcher` 对象 （这样比每次都去遍历 `queue` 效率上会高很多）。

如果目前队列 `queue` 中还没有这个 `Watcher` 对象，则该对象会被 `push` 进队列 `queue` 中去。

`waiting` 是一个标记位，标记是否已经向 `nextTick` 传递了 `flushSchedulerQueue` 方法，在下一个 tick 的时候执行 `flushSchedulerQueue` 方法来 flush 队列 `queue`，执行它里面的所有 `Watcher` 对象的 `run` 方法。

flushSchedulerQueue

```
function flushSchedulerQueue () {
    let watcher, id;

    for (index = 0; index < queue.length; index++) {
        watcher = queue[index];
        id = watcher.id;
        has[id] = null;
        watcher.run();
    }

    waiting  = false;
}
```

举个例子

```
let watch1 = new Watcher();
let watch2 = new Watcher();

watch1.update();
watch1.update();
watch2.update();
```

我们现在 new 了两个 `Watcher` 对象，因为修改了 `data` 的数据，所以我们模拟触发了两次 `watch1` 的 `update` 以及 一次 `watch2` 的 `update`。

假设没有批量异步更新策略的话，理论上应该执行 `Watcher` 对象的 `run`，那么会打印。

```
watch1 update
watch1视图更新啦～
watch1 update
watch1视图更新啦～
watch2 update
watch2视图更新啦～
```

实际上则执行

```
watch1 update
watch1 update
watch2 update
watch1视图更新啦～
watch2视图更新啦～
```

这就是异步更新策略的效果，相同的 `Watcher` 对象会在这个过程中被剔除，在下一个 tick 的时候去更新视图，从而达到对我们第一个例子的优化。

我们再回过头聊一下第一个例子， `number` 会被不停地进行 `++` 操作，不断地触发它对应的 `Dep` 中的 `Watcher` 对象的 `update` 方法。然后最终 `queue` 中因为对相同 `id` 的 `Watcher` 对象进行了筛选，从而 `queue` 中实际上只会存在一个 `number` 对应的 `Watcher` 对象。在下一个 tick 的时候（此时 `number` 已经变成了 1000），触发 `Watcher` 对象的 `run` 方法来更新视图，将视图上的 `number` 从 0 直接变成 1000。

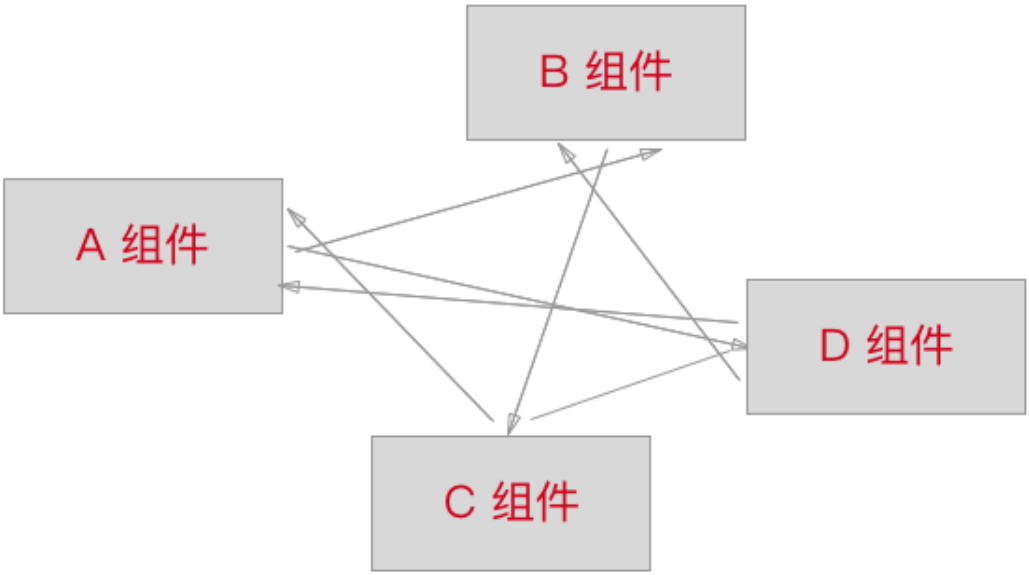
到这里，批量异步更新策略及 nextTick 原理已经讲完了，接下来让我们学习一下 Vuex 状态管理的工作原理。

注：本节代码参考《批量异步更新策略及 nextTick 原理》。

Vuex 状态管理的工作原理

为什么要使用 Vuex

当我们使用 Vue.js 来开发一个单页应用时，经常会遇到一些组件间共享的数据或状态，或是需要通过 props 深层传递的一些数据。在应用规模较小的时候，我们会使用 props、事件等常用的父子组件的组件间通信方法，或者是通过事件总线来进行任意两个组件的通信。但是当应用逐渐复杂后，问题就开始出现了，这样的通信方式会导致数据流异常地混乱。



这个时候，我们就需要用到我们的状态管理工具 Vuex 了。Vuex 是一个专门为 Vue.js 框架设计的、专门用来对于 Vue.js 应用进行状态管理的库。它借鉴了 Flux、redux 的基本思想，将状态抽离到全局，形成一个 Store。因为 Vuex 内部采用了 new Vue 来将 Store 内的数据进行「响应式化」，所以 Vuex 是一款利用 Vue 内部机制的库，与 Vue 高度契合，与 Vue 搭配使用显得更加简单高效，但缺点是不能与其他的框架（如 react）配合使用。

本节将简单介绍 Vuex 最核心的内部机制，起个抛砖引玉的作用，想了解更多细节可以参考笔者 Github 上的另一篇文章 《Vuex源码解析》或者直接阅读 Vuex源码。

安装

Vue.js 提供了一个 `Vue.use` 的方法来安装插件，内部会调用插件提供的 `install` 方法。

```
Vue.use(Vuex);
```

所以我们的插件需要提供一个 `install` 方法来安装。

```
let Vue;

export default install (_Vue) {
    Vue.mixin({ beforeCreate: vuexInit });
    Vue = _Vue;
}
```

我们采用 `Vue.mixin` 方法将 `vuexInit` 方法混淆进 `beforeCreate` 钩子中，并用 `Vue` 保存 Vue 对象。那么 `vuexInit` 究竟实现了什么呢?

我们知道，在使用 Vuex 的时候，我们需要将 `store` 传入到 Vue 实例中去。

```
/*将store放入Vue创建时的option中*/
new Vue({
    el: '#app',
    store
});
```

但是我们却在每一个 vm 中都可以访问该 `store`，这个就需要靠 `vuexInit` 了。

```
function vuexInit () {
    const options = this.$options;
    if (options.store) {
        this.$store = options.store;
    } else {
        this.$store = options.parent.$store;
    }
}
```

因为之前已经用 `Vue.mixin` 方法将 `vuexInit` 方法混淆进 `beforeCreate` 钩子中，所以每一个 vm 实例都会调用 `vuexInit` 方法。

如果是根节点（`$options` 中存在 `store` 说明是根节点），则直接将 `options.store` 赋值给 `this.$store`。否则则说明不是根节点，从父节点的 `$store` 中获取。

通过这步的操作，我们已经可以在任意一个 vm 中通过 `this.$store` 来访问 `Store` 的实例啦～

Store

数据的响应式化

首先我们需要在 `Store` 的构造函数中对 `state` 进行「响应式化」。

```
constructor () {
    this._vm = new Vue({
        data: {
            $$state: this.state
        }
    })
}
```

熟悉「响应式」的同学肯定知道，这个步骤以后，`state` 会将需要的依赖收集在 `Dep` 中，在被修改时更新对应视图。我们来看一个小例子。

```
let globalData = {
    d: 'hello world'
};
new Vue({
    data () {
        return {
            $$state: {
                globalData
            }
```

```
        }
    }
});

/* modify */
setTimeout(() => {
    globalData.d = 'hi~';
}, 1000);

Vue.prototype.globalData = globalData;
```

任意模板中

```
<div>{{globalData.d}}</div>
```

上述代码在全局有一个 `globalData`，它被传入一个 `Vue` 对象的 `data` 中，之后在任意 Vue 模板中对该变量进行展示，因为此时 `globalData` 已经在 Vue 的 `prototype` 上了所以直接通过 `this.prototype` 访问，也就是在模板中的 `{{globalData.d}}`。此时，`setTimeout` 在 1s 之后将 `globalData.d` 进行修改，我们发现模板中的 `globalData.d` 发生了变化。其实上述部分就是 Vuex 依赖 Vue 核心实现数据的“响应式化”。

讲完了 Vuex 最核心的通过 Vue 进行数据的「响应式化」，接下来我们再来介绍两个 `Store` 的 API。

commit

首先是 `commit` 方法，我们知道 `commit` 方法是用来触发 `mutation` 的。

```
commit (type, payload, _options) {
    const entry = this._mutations[type];
    entry.forEach(function commitIterator (handler) {
        handler(payload);
    });
}
```

从 `_mutations` 中取出对应的 mutation，循环执行其中的每一个 mutation。

dispatch

`dispatch` 同样道理，用于触发 action，可以包含异步状态。

```
dispatch (type, payload) {
    const entry = this._actions[type];

    return entry.length > 1
    ? Promise.all(entry.map(handler => handler(payload)))
    : entry[0](payload);
}
```

同样的，取出 `_actions` 中的所有对应 action，将其执行，如果有多个则用 `Promise.all` 进行包装。

最后

理解 Vuex 的核心在于理解其如何与 Vue 本身结合，如何利用 Vue 的响应式机制来实现核心 Store 的「响应式化」。

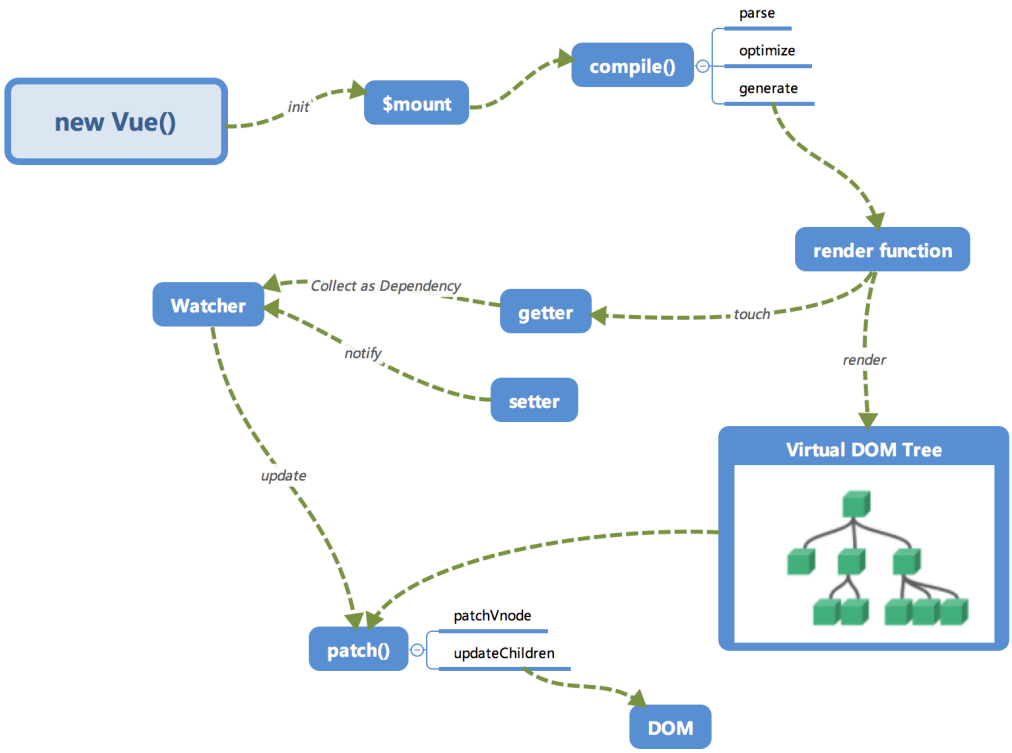
Vuex 本身代码不多且设计优雅，非常值得一读，想阅读源码的同学请看Vuex源码。

注：本节代码参考《Vuex状态管理的工作原理》。

总结 & 常见问题解答

总结

在本小册的第一节中，笔者对 Vue.js 内部运行机制做了一个全局的概览，当时通过下面这张图把 Vue.js 拆分成一个一个小模块来介绍，之后通过这一系列小节的学习，相信大家已经对 Vue.js 内部的原理有了一个更进一步的了解，对这张图也再也不会感觉到那么陌生。



每个小节中的代码都是笔者根据 Vue.js 原理单独抽离出来写成的 Demo，大家可以在我的 Gtihub 上查看完整的代码 (见 VueDemo 项目)。

本小册对 Vue.js 原理进行了初步的介绍，希望能够起到一个抛砖引玉的作用，读者读完以后，可以利用这些基础对 Vue.js 进行一个更加深入的探索，相信会有更大的收获。

常见问题

1. 怎么实现 `this._test` 改变而不是 `this._data.test` 改变触发更新?

答：其实这中间有一个**代理**的过程。

```
_proxy(options.data);

function _proxy (data) {
```

```
    const that = this;
    Object.keys(data).forEach(key => {
        Object.defineProperty(that, key, {
            configurable: true,
            enumerable: true,
            get: function proxyGetter () {
                return that._data[key];
            },
            set: function proxySetter (val) {
                that._data[key] = val;
            }
        })
    });
}
```

本质就是通过 `Object.defineProperty` 使在访问 `this` 上的某属性时从 `this._data` 中读取（写入）。

2. 能不能将依赖收集中讲到的 `dep.addSub(Dep.target)` 改成 `dep.addSub(new Watcher())` 呢?

为了便于读者理解这部分内容，我将代码做了简化，实际上一个 `Watcher` 对象可能会在多个 `Dep` 中，并不是每次 `addSub` 都是一个新的 `Watcher` 对象，需依赖 `Dep.target` 进行收集（实际上 `Dep.target` 也是通过 `Watcher` 对象的 `get` 方法调用 `pushTarget` 将自身赋值给 `Dep.target`）。

最后

从 2017 年 12 月开始写这本小册到现在差不多 2 个月的时间，虽说之前写过类似的内容，但是将 Vue.js 源码抽离成一个一个 Demo 还是花了很多时间，对于这本小册也是前前后后改了好几次才让自己满意。

因为读者的基础不一致，而小册的定位是偏向于对新手读者更加友好，所以我尽量用更加浅显易懂的方式去写这本小册的内容。希望大家可以通过这本小册初步掌握 Vue.js 的原理，掌握这些原理以后再去尝试阅读 Vue.js 源码，相信会事半功倍，也会对 Vue.js 有更深一层的理解。

End