04 可复用性 & 组合

目录(Catalog)

生词(New Words)

- **anchor ['æŋkə] --n.锚 v.抛锚**
    - anchor point 锚点
    - anchor text 锚文本, 链接锚文本
- **anonymous [ə'nɒnɪməs] --adj.匿名的, 无名的**
    - It is an anonymous letter. 这时一封匿名信.
- **capitalize ['kæpɪtəlaɪz] --vt.首字母大写; 使资本化; 估计...的假值**
    - capitalize operation. 资本化运作.

前置知识

- `Mixin`(混入) 是 js 中实现**对象组合**最流行的一种模式. 在一个 mixin 方法中, 一个对象接收来自另一个对象的属性和方法, 许多 js 库中都有类似的 mixin 方法:

    ```
    function mixin(receiver, supplier) {
        Object.keys(supplier).forEach(function(key) {
            receiver[key] = supplier[key];
        })
        return receiver;
    }
    ```

    此笔记来自: `JS-book-learning/《深入理解ES6》/chapter04_扩展对象的功能性/chapter04-扩展对象的功能性.md`

内容(Content)

4.1 混入

4.1.1 基础

- 混入(mixin) 提供了一种非常灵活的方式, 来分发 Vue 组件中的可复用功能. 一个混入对象可以包含任意组件选项. 当组件使用混入对象时, 所有混入对象的选项将被“混合”进入该组件本身的选项.

  例子:

```
// - 定义一个混入对象
var myMixin = {
    created; function() {
        this.hello()
    },
    methods: {
        hello: function() {
            console.log('Hello from mixin!')
        }
    }
};
// - 定义一个使用混入对象的组件
var Component = Vue.extend({
    mixins: [myMixin]
});
var component = new Component();    // => "Hello from mixin!"
```

  下面给一个单文件组件的示例:

```
<template>
    <div class="default-div">
    <h2>4.1.1 混入--基础</h2>
    <p>{{mixinValue}}</p>
    </div>
</template>
<script>
    const myMixin = {
        data() {
            return {
                mixinValue: ""
            }
        },
        created: function() {
            this.hello()
        },
        methods: {
            hello: function() {
                this.mixinValue = "Hello from mixin!"
            }
        }
    };
    export default {
        mixins: [myMixin],
        name: 'Demo41',
```

```
        data() {
            return {
            }
        },
    }
</script>
<style scoped></style>
```

4.1.2 选项合并

- 当组件和混入对象含有同名选项时, 这些选项将以恰当的方式进行“合并”.
    - (1) `data`(数据对象)在内部会进行递归合并, 当发生冲突时以**组件**数据优先.
    - (2) 同名钩子函数将合并为一个数组, 因此都将被调用. 另外, 混入对象的钩子将在组件自身钩子之前调用.
    - (3) 值为对象的选项, 例如 `methods`, `components` 和 `directives`, 将被合并为同一个对象. 两个对象键名冲突时, 取组件对象的键值对.

这里把上面 4.1.1 单文件组件的实例稍微改一下, 看下测试:

```
<template>
    <div class="default-div">
    <h2>4.1.1 混入--基础</h2>
    <p>{{mixinValue}}</p>
    </div>
</template>
<script>
    const myMixin = {
        data() {
            return {
                mixinValue: "",
                messge: 'hello',
                foo: 'abc'
            }
        },
        created: function() {
            this.hello()
        },
        methods: {
            hello: function() {
                this.mixinValue = "Hello from mixin!"
            },
            foos: function() {
                console.log('foo')
            },
            conflicting: function() {
                console.log('from mixin')
            }
        }
    };
    export default {
        mixins: [myMixin],
```

```
        name: 'Demo411',
        data() {
            return {
                message: 'Hello Vue',
                bar: 'def'
            }
        },
        created: function() {
            // => {"message": "Hello Vue", "bar": "def",
            //     "mixinValue": "Hello from mixin!", "foo": "abc"}
            // - tip: 注意这里的的 console 会输出 2 次, 因为利用 mixin
            //   合并, 肯定要把每个对象中的属性/方法都执行一遍.
            console.log(JSON.stringify(this.$data))
        },
        methods: {
            bars: function() {
                console.log('bar')
            },
            conflicting: function() {
                console.log('from self')
            }
        }
    }
</script>
<style scoped></style>
```

注意: `Vue.extend()` 也使用同样的策略进行合并.

4.1.3 全局混入

- 混入也可以进行全局注册. 使用时格外小心! 一旦使用全局混入, 它将影响每一个之后创建的 Vue 实例. 使用恰当时, 这可以用来为自定义选项注入处理逻辑.

```
// - 为自定义的选项 'myOption' 注入一个处理器
Vue.mixin({
    created; function() {
        var myOption = this.$options.myOption;
        if (myOption) {
            console.log(myOption)
        }
    }
})
new Vue({
    myOption: 'hello!'
})
// => "hello!"
```

WARNING: 请谨慎使用全局混入, 因为它会影响每个单独创建的 Vue 实例 (包括第三方组件). 大多数情况下, 只应当应用于自定义选项, 就像上面示例一样. 推荐将其作为插件发布, 以避免重复应用混入.

4.1.4 自定义选项合并策略

- 自定义选项将使用默认策略, 即简单地覆盖已有值. 如果想让自定义选项以自定义逻辑合并, 可以向 `Vue.config.optionMergeStrategies` 添加一个函数:

```
Vue.config.optionMergeStrategies.myOption = function(toVal, fromVal) {
    // - 返回合并后的值.
}
```

对于多数值为对象的选项, 可以使用与 `methods` 相同的合并策略:

```
var strategies = Vue.config.optionMergeStrategies;
strategies.myOption = strategies.methods
```

可以在 Vuex 1.x 的混入策略里找到一个更高级的例子:

```
const merge = Vue.config.optionMergeStrategies.computed;
Vue.config.optionMergeStrategies.vuex = function(toVal, fromVal) {
    if (!toVal) return fromVal;
    if (!fromVal) return toVal;
    return {
        getters: merge(toVal.getters, fromVal.getters),
        state: merge(toVal.state, fromVal.state),
        actions: merge(toVal.actions, fromVal.actions)
    }
}
```

tip: 我也没弄懂怎么使用的....., 以后看完源码再回来添加注释.

4.2 自定义指令

4.2.1 简介

- *Additional Info* : `指令` 的知识见: `./01-基础.md -> 1.4 模板语法 -> 1.4.2 指令`.
- 除了核心功能默认内置的指令 (v-model 和 v-show), Vue 也允许注册自定义指令. 注意, 在 Vue2.0 中, 代码复用和抽象的主要形式是组件. 然而, 有的情况下, 你仍然需要对普通 DOM 元素进行底层操作, 这时候就会用到自定义指令. 举个聚焦输入框的例子, 如下:

当页面加载时, 该元素将获得焦点(注意: `autofocus` 在移动端 Safari 上不工作). 事实上, 只要你在打开这个页面后还没点击过任何内容, 这个输入框就应当还是处于聚焦状态. 现在让我们用自定义指令来实现这个功能:

```
<template>
    <div class="default-div">
    <h2>4.2.1 自定义指令</h2>
        <input type="text" v-model="val" v-focus>
```

```
    </div>
</template>

<script>
    export default {
        name: 'Demo411',
        data() {
            return {
                val: '自定义指令'
            }
        },
        directives: {
            focus: {
                // - 自定义指令的定义: 当被绑定的元素已插入其父节点时调用
                inserted: function(el) {
                    el.focus()
                }
            }
        }
    }
</script>
<style scoped></style>
```

4.2.2 钩子函数

- 一个指令可对象可以提供如下几个钩子函数(均为可选):
  - (1) `bind`: 只调用一次, 指令第一次绑定到元素时调用. 在这里可以进行一次性的初始化设置.
  - (2) `inserted`: 当绑定的元素已插入其父节点时调用(仅保证父节点存在, 但不一定已被插入文档中) (示例: 见上面的 `4.2.1`)
  - (3) `update`: 所在组件的 VNode 更新时调用, **但是可能发生在其子 VNode 更新之前**. 指令的值可能发生了改变, 也可能没有. 但是你可以通过比较更新前后的值来忽略不必要的模板更新(详细的钩子函数参数见下.)
    - **NOTICE:** 我们会在 `4.3.2.1 虚拟 DOM` 讨论 `渲染函数` (`4.3 渲染函数 & JSX`) 时介绍更多 VNodes 的细节.
  - (4) `componentUpdated`: 指令所在组件的 VNode 及其子 VNode 全部更新后调用.
  - (5) `unbind`: 只调用一次, 指令与元素解绑时调用.

接下来我们来看一下钩子函数的参数(即: `el`, `binding`, `vnode` 和 `oldVnode`)

4.2.3 钩子函数参数

- 自定义指令钩子函数会被传入一下参数:
  - (1) `el`: 自定义指令所绑定的元素, 可以用来直接操作 DOM.
  - (2) `binding`: 一个对象, 包含以下 property(属性):
    - `name`: 自定义指令名, 不包含 `v-` 前缀.

- `value`：自定义指令的绑定值，例如：`v-my-directive="1 + 1"` 中，绑定值为 2.
- `oldValue`：自定义指令绑定的前一个值，仅在 `update` 和 `componentUpdated` 钩子中可用. 无论值是否改变都可用.
- `expression`：指令绑定值的字符串形式. 例如 `v-my-directive="1 + 1"` 中，表达式为 `"1 + 1"`.
- `arg`：传给自定义指令的参数，可选. 例如 `v-my-directive:foo` 中，参数为 `"foo"`.
- `modifiers`：一个包含修饰符的对象. 例如：`v-my-directive.foo.bar` 中，修饰符对象为 `{foo: true, bar: true}`.

- (3) `vnode`：Vue 编译生成的虚拟节点. 移步 **VNode API** 来了解更多详情.
- (4) `oldVnode`：上一个虚拟节点，仅在 `update` 和 `componentUpdated` 钩子中可用.

WARNING: 除了 `el` 之外，其他参数都应该是只读的，切勿进行修改. 如果需要在钩子之间共享数据，建议通过元素的 `dataset` 来进行.

这是一个使用了这些 property(属性) 的自定义钩子样例:

```
<template>
    <div class="default-div">
    <h2>4.2.3 钩子函数参数</h2>
    <div v-demo:foo.a.b="message"></div>
    </div>
</template>
<script>
    export default {
        name: 'Demo423',
        data() {
            return {
                message: 'Hello!'
            }
        },
        directives: {
            demo: {
                // - 自定义指令的定义
                bind: function(el, binding, vnode) {
                    var s = JSON.stringify;
                    // - name: 为自定义指令名.
                    // - value: 为给自定义指令传递的值.
                    // - expression: 指令绑定值的字符串形式.
                    // - arg: 传递给自定义指令的参数, 这里为 `foo`
                    // - modifiers: 修饰符对象, 此处是 `{a:true, b:true}`
                    // - vnode: Vue 编译后生成的虚拟节点.
                    el.innerHTML =
                        'name: ' + s(binding.name) + '<br>' +
                        'value: ' + s(binding.value) + '<br>' +
                        'expression: ' + s(binding.expression) + '<br>' +
                        'argument: ' + s(binding.arg) + '<br>' +
                        'modifiers: ' + s(binding.modifiers) + '<br>' +
                        'vnode keys: ' + Object.keys(vnode).join(', ')
                }
```

```
            }
        }
    }
</script>
<style scoped>
    .default-div {
        min-height: 420px;
    }
</style>
```

4.2.3.1 动态指令参数

- 指令的参数可以是动态的. 例如, 在 `v-mydirective:[arguments]="value"` 中, `argument` 参数可以根据组件实例数据进行更新! 这使得自定义指令可以在应用中被灵活使用.
    - **TIP:** 通过上面这个 `[arguments]` 是不是发现了 Vue 这里的动态指令参数, 和 ES6 中新增的 `可计算属性名(Computed Property Name)` 的语法几乎是一样的? `可计算属性名` 笔记参考: `../../../JS-book-learning/《深入理解ES6》/chapter04_扩展对象的功能性`

例如你想要创建一个自定义指令, 用来通过固定布局将元素固定在页面中, 我们可以像这样创建一个通过指令值来更新竖直位置像素值的自定义指令:

```
<template>
    <div class="default-div">
    <h2>4.2.3.1 钩子函数参数 -- 动态指令参数</h2>
    <div id="fixed-ele" ref="fixedEle">
        <p>Scroll down the page</p>
        <p v-pin="200">Stick me 200px from the top of the page</p>
    </div>
    </div>
</template>
<script>
    export default {
        name: 'Demo4231',
        data() {
            return {}
        },
        directives: {
            pin: {
                bind: function(el, binding, vnode) {
                    el.style.position = 'fixed';
                    el.style.top = binding.value + 'px';
                }
            }
        }
    }
</script>
<style scoped></style>
```

上面的示例把该元素固定在距离页面顶部 200 像素的位置. 但如果场景是我们需要把元素固定在左侧而不是顶部又该怎么办呢? 这时使用**动态参数**就可以非常方便地根据每个组件实例来进行更新. 我们更该一下上面的示例:

```
<template>
    <div class="default-div">
    <h2>4.2.3.1 钩子函数参数 -- 动态指令参数</h2>
    <div id="fixed-ele" ref="fixedEle">
        <p>Scroll down inside this section ↓</p>
        <p v-pin:[direction]="200">I am pinned onto the page at 200px to the
left.</p>
    </div>
    </div>
</template>
<script>
    export default {
        name: 'Demo4231',
        data() {
            return {
                direction: 'left'
            }
        },
        directives: {
            pin: {
                bind: function(el, binding, vnode) {
                    el.style.position = 'fixed';
                    var s = (binding.arg == 'left' ? 'left': 'top')
                    el.style[s] = binding.value + 'px';
                }
            }
        }
    }
</script>
<style scoped></style>
```

这样这个自定义指令现在的灵活性就足以支持一些不同的用例了.

4.2.4 函数简写

- 在很多时候, 你可能想在 `bind` 和 `update` 时触发相同行为, 而不关心其它的钩子. 比如这样写:

```
<p v-color-swatch="red">Color changed</p>
```

```
export default {
    name: 'Demo4231',
    data() {
        return {
            red: 'red'
        }
    },
    directives: {
        "color-swatch": function(el, binding, vnode) {
            el.style.color = binding.value;
        }
    }
}
```

4.2.5 对象字面量

- 如果指令需要多个值, 可以传入一个 JS 对象字面量. 记住, 指令函数能够接受所有合法的 JS 表达式.

```
<div v-demo="{color: 'white', text: 'Hello!'}"></div>
```

```
export default {
    name: 'Demo4231',
    data() {
        return {
            red: 'red'
        }
    },
    directives: {
        "color-swatch": function(el, binding, vnode) {
            console.log(binding.value.color);       // => "white"
            console.log(binding.value.text);       // => "Hello!"
        }
    }
}
```

4.3 渲染函数 & JSX

4.3.1 基础

- Vue 推荐在绝大多数情况下使用模板来创建你的 HTML. 然而在一些场景中, 你真的需要 JavaScript 的完全编程的能力. 这时你可以用渲染函数, 它比模板更接近编译器.

  让我们深入一个简单的例子, 这个例子里 render 函数很实用. 假设我们要生成一些带锚点的标题:

  NOTICE: 下面给出的是单文件组件中的写法

```
<template>
    <div class="default-div">
    <h2>4.3.1 渲染函数 & JSX -- 基础</h2>
        <AnchoredHeading :level="level">Hello world!</AnchoredHeading>
    </div>
</template>
<script>
    import AnchoredHeading from './anchored-heading';
    export default {
        name: 'Demo423',
        data() {
            return {
                level: 2
            }
        },
        components: {
            AnchoredHeading,
        }
    }
```

```
</script>
<style scoped></style>
```

```
<!-- anchored-heading.vue -->
<template>
    <h1 v-if="level == 1">
        <slot></slot>
    </h1>
    <h2 v-else-if="level == 2">
        <slot></slot>
    </h2>
    <h3 v-else-if="level == 3">
        <slot></slot>
    </h3>
    <h4 v-else-if="level == 4">
        <slot></slot>
    </h4>
    <h5 v-else-if="level == 5">
        <slot></slot>
    </h5>
    <h6 v-else-if="level == 6">
        <slot></slot>
    </h6>
</template>
<script>
    export default {
        name: 'AnchoredHeading',
        props: {
            level: {
                type: Number,
                required: true
            }
        }
    }
</script>
```

这里用模板并不是最好的选择: 不但代码冗长, 而且在每一个级别的标题中重复书写了 `<slot></slot>`, 在要插入锚点元素时还要再次重复.

虽然模板在大多数组件中都非常好用, 但是显然在这里它就不合适了. 那么, 我们来尝试使用 `render` 函数重写上面的例子:

```
// - anchored-heading.jsx
export default {
    name: 'AnchoredHeading',
    render: function(createElement) {
        return createElement(
            'h' + this.level,       // - 标签名
            this.$slots.default     // - 子节点数组
        )
    },
    props: {
```

```
            level: {
                type: Number,
                required: true
            }
        }
    }
```

看起来简单多了! 这样代码精简很多, 但是需要非常熟悉 Vue 的实例 property. 这这个例子中, 你需要知道, 像组件中传递不带 `v-slot` 指令的子节点时, 比如 `anchored-heading` 中的 `Hello world!`, 这些子节点被存储在组件实例中的 `$slots.default` 中. 如果你还没有理解, 在深入渲染函数之前推荐阅读 实例 property API

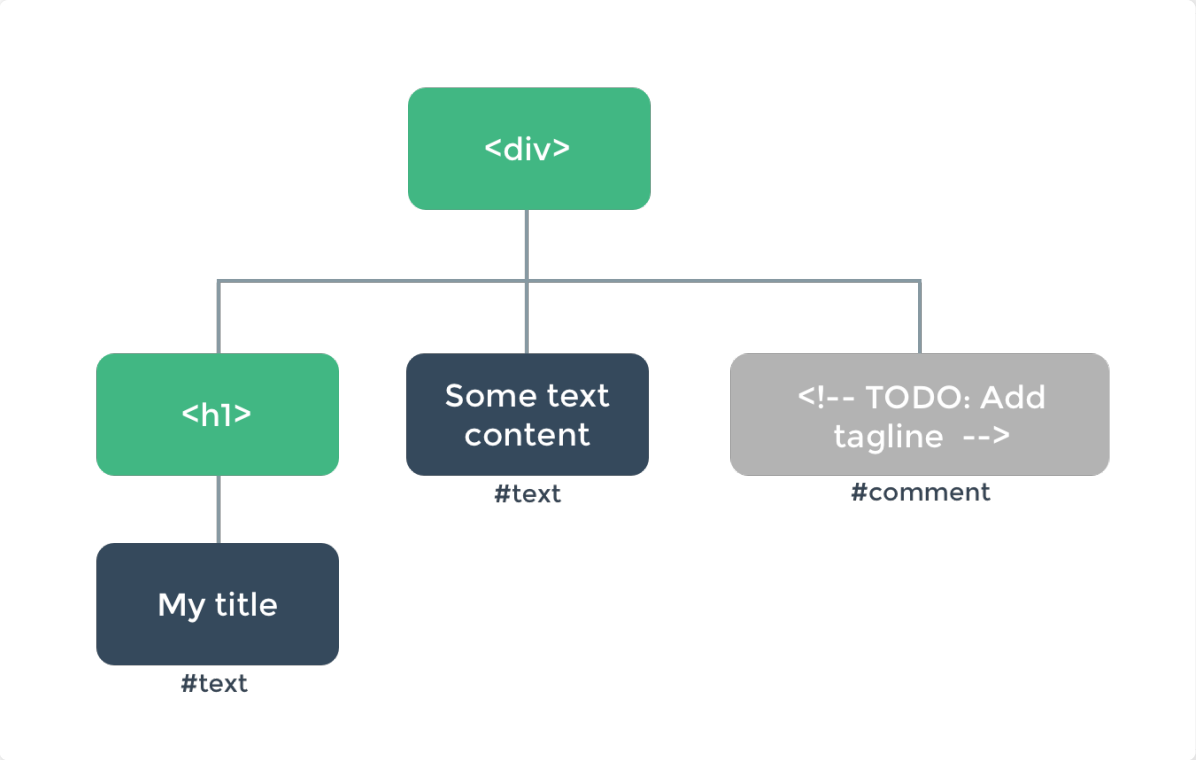
4.3.2 节点, 树以及虚拟 DOM

- 在深入渲染函数之前, 了解一些浏览器的工作原理是很重要的. 以下面这段 HTML 为例:

```
<div>
    <h1>My title</h1>
    Some text content
    <!-- TODO: Add tagline -->
</div>
```

当浏览器读到这些代码时, 它会建立一个 DOM 节点树 来保持追踪所有内容, 如同你会画一张家谱树来追踪家庭成员的发展一样.

上述 HTML 对应的 DOM 节点树如下图所示:



每个元素都是一个节点. 每段文字也是一个节点. 甚至注释也都是节点. 一个节点就是页面的一个部分. 就像家谱树一样, 每个节点都可以有孩子节点 (也就是说每个部分可以包含其它的一些部分).

高效地更新所有这些节点会是比较困难的, 不过所幸你不必手动完成这个工作. 你只需要告诉 Vue 你希望页面上的 HTML 是什么, 这可以是在一个模板里:

```
<h1>{{ blogTitle }}</h1>
```

或者一个渲染函数里:

```
render: function(createElement) {
    return createElement('h1', this.blogTitle)
}
```

在这两种情况下, Vue 都会自动保持页面的更新, 即便 `blogTitle` 发生了改变.

4.3.2.1 虚拟 DOM

- Vue 通过建立一个虚拟 DOM 来追踪自己要如何改变真实 DOM. 请仔细看这行代码:

```
return createElement('h1', this.blogTitle)
```

createElement 到底会返回什么呢? 其实不是一个实际的 DOM 元素. 它更准确的名字可能是 createNodeDescription(创建节点描述), 因为它所包含的信息会告诉 Vue 页面上需要渲染什么样的节点, 包括及其子节点的描述信息. 我们把这样的节点描述为 "虚拟节点 (virtual node)", 也常简写它为 "VNode". "虚拟 DOM" 是我们对由 Vue 组件树建立起来的整个 VNode 树的称呼.

4.3.3 `createElement` 参数

- 接下来你需要熟悉的是如何在 `createElement` 函数中使用模板中的那些功能. 这里是 `createElement` 接受的参数:

```
// @returns {VNode}
createElement(
    // {String | Object | Function}
    // - 一个 HTML 标签名, 组件选项对象, 或者 resolve 了上述任何一种的一个
    //   async 函数. 必填项.
    'div',

    // {Object}
    // - 一个与模板中 attribute 对应的数据对象. 可选
    {
        // - 详情见下一节
    },

    // {String | Array}
    // - 子级虚拟节点(VNodes), 由 `createElement()` 构建而成,
    //   也可以使用字符串来生成 "文本虚拟节点". 可选
    [
        '先写一些文字',
        createElement('h1', '一则头条'),
        createElement(MyComponent, {
            props: {
                someProp: 'foobar'
            }
        })
    ]
)
```

4.3.3.1 深入数据对象

- 有一点需要注意: 正如 `v-bind:class` 和 `v-bind:style` 在模板语法中会被特别对待一样, 它们在 VNode 数据对象中也有对应的顶层字段. 该对象也允许你绑定普通的 HTML attribute, 也允许绑定如 `innerHTML` 这样的 DOM property (这会覆盖 `v-html` 指令).

```
{
    // - 与 `v-bind:class` 的 API 相同, 接受一个字符串,
    //   对象或字符串和对象组成的数组
    class: {
        foo: true,
        bar: false
    },
    // - 与 `v-bind:style` 的 API 相同,
    //   接受一个字符串, 对象, 或对象组成的数组
    style: {
        color: 'red',
        fontSize: '14px'
    },
    // - 普通的 HTML 特性(attribute)
    attrs: {
        id: 'foo'
    },
    // - 组件 prop
    props: {
        myProp: 'bar'
    },
    // - DOM property
    domProps: {
        innerHTML: 'baz'
    },
    // - 事件监听器在 `on` 内, 但不再支持如 `v-on:keyup.enter` 这样的修饰器.
    //   需要在处理函数中手动检查 keyCode.
    on: {
        click: this.clickHandler
    },
    // - 仅用于组件, 用于监听原生事件, 而不是组件内部使用 `vm.$emit`触发的事件.
    nativeOn: {
        click: this.nativeClickHandler
    },
    // - 自定义指令. 注意, 你无法对 `binding` 中的 `oldValue` 赋值,
    //   因为 Vue 已经自动为你进行了同步.
    directives: [
        {
            name: 'my-custom-directive',
            value: '2',
            expression: '1 + 1',
            arg: 'foo',
            modifiers: {
                bar: true
            }
        }
```

```
    ],
    // - 作用域插槽的格式为
    // {name: props => VNode | Array<VNode>}
    scopedSlots: {
        default: props => createElement('span', props.text)
    },
    // - 如果组件是其它组件的子组件, 需为插槽指定名称
    slot: 'name-of-slot',
    // - 其它特殊顶层 property
    key: 'myKey',
    ref: 'myRef',
    // - 如果你在渲染函数中给多个元素都应用了相同的 ref 名, 那么 `$refs.myRef`
    //   会变成一个数组.
    refInFor: true
}
```

4.3.3.2 完整示例

- 有了这些知识, 我们现在完善本章节开始实现的那个 `.jsx` 格式的组件. 下面是更新后的代码:

```
var getChildrenTextContent = function(children) {
    return children.map(function(node) {
        return node.children
            ? getChildrenTextContent(node.children)
            : node.text
    }).join('')
}
export default {
    name: 'AnchoredHeading',
    render: function(createElement) {
        // - 创建 kebab-case 风格的 ID
        // - `\W` 匹配除了字母, 数字, 下划线之外的字符.
        var headingId = getChildrenTextContent(this.$slots.default)
            .toLowerCase()
            .replace(/\W+/g, '-')
            .replace(/(^-|-$)/g, '');

        return createElement(
            'h' + this.level,
            {
                style: {
                    color: '#232323'
                }
            },
            [
                createElement(
                    'a',
                    {
                        style: {
                            color: '#0366d6'
                        },
                        attrs: {
```

```
                            name: headingId,
                            href: '#' + headingId
                        }
                    },
                    this.$slots.default
                )
            ]
        )
    },
    props: {
        level: {
            type: Number,
            required: true
        }
    }
}
```

4.3.3.3 约束

- 组件树中的所有 VNode 必须是唯一的. 这意味着, 下面的渲染函数是不合法的:

```
render: function (createElement) {
    var myParagraphVNode = createElement('p', 'hi')
    return createElement('div', [
        // 错误 - 重复的 VNode
        myParagraphVNode,
        myParagraphVNode
    ])
}
```

如果你真的需要重复很多次的元素/组件, 你可以使用工厂函数来实现. 例如, 下面这渲染函数用完全合法的方式渲染了 20 个相同的段落:

```
render: function(createElement) {
    return createElement(
        'div',
        // - 注意: 在 Array 上使用 Array.apply() 我们比较少见,
        //   平时用的较多的是 function.apply(); 此处会创建一个拥有
        //   20 个 undefined 的数组.
        Array.apply(null, {length: 20}).map(function() {
            return createElement('p', 'hi')
        })
    )
}
```

4.3.4 使用 JavaScript 代替模板功能

4.3.4.1 `v-if` 和 `v-for`

- 只要在原生的 JavaScript 中可以轻松完成的操作, Vue 的渲染函数就不会提供专有的替代方法. 比如, 在模板中使用的 `v-if` 和 `v-for`:

```
<ul v-if="items.length">
    <li v-for="item in items">{{ item.name }}</li>
</ul>
<p v-else>No items found.</p>
```

这些都可以在渲染函数中用 JavaScript 的 if/else 和 map 来重写:

```
export default {
    name: 'Demo4341',
    props: ['items'],
    render: function(createElement) {
        if (this.items.length) {
            return createElement('ul', this.items.map(function(item) {
                return createElement('li', item.name)
            }))
        } else {
            return createElement('p', 'No items found.')
        }
    }
}
```

4.3.4.2 `v-model`

- 渲染函数中没有与 `v-model` 的直接对应 -- 你必须自己实现响应的逻辑:

```
export default {
    name: 'Demo4342',
    render: function(createElement) {
        var self = this;
        return createElement('input', {
            domProps: {
                value: self.value
            },
            on: {
                input: function(event) {
                    self.$emit('input', event.target.value)
                }
            }
        })
    }
}
```

这就是深入底层的代价, 但与 `v-model` 相比, 这可以让你更好地控制交互细节.

4.3.4.3 事件 & 按键修饰符

- 对于 `.passive`, `.capture` 和 `.once` 这些事件修饰符, Vue 提供了响应的前缀可以用于 `on`:

修饰符	前缀
`.passive`	`&`
`.capture`	`!`
`.once`	`~`
`.capture.once` 或 `once.capture`	`~!`

例如:

```
on: {
    '!click': this.doThisInCapturingMode,
    '~keyup': this.doThisOnce,
    '~!mousemove': this.doThisOnceInCapturingMode
}
```

对于所有其他的修饰符, 私有前缀都不是必须的, 因为你可以在时间处理函数中使用事件方法:

修饰符	处理函数中的等价操作
`.stop`	`event.stopPropagation()`
`.prevent`	`event.preventDefault()`
`.self`	`if (event.target !== event.currentTarget) return`
按键: `.enter`, `.13`	`if (event.keyCode !== 13) return` (对于别的按键修饰符来说, 可将 `13` 改为 另一个按键码)
修饰符: `.ctrl`, `.alt`, `.shift`, `.meta`	`if (!event.ctrlKey) return` (将 `ctrlKey` 分别修改为 `altKey`, `shiftKey` 或者 `metaKay`)

这里有一个使用所有修饰符的例子:

```
on: {
    keyup: function(event) {
        // - 如果触发事件的元素不是事件绑定的元素则返回
        // - `event.currentTarget`: 事件处理程序当前正在处理事件的那个元素.
        if (event.target !== event.currentTarget) return;
        // - 如果按下去的不是 enter 键或者没有同时按下 shift 键, 则返回
        if (!event.shiftKey || event.keyCode !== 13) return;

        // - 阻止事件冒泡
        event.stopPropagation();

        // - 阻止该元素默认的 keyup 事件
        event.preventDefault();
```

```
    }
}
```

4.3.4.4 插槽

- (1) 你可以通过 `this.$slots` 访问静态插槽的内容, 每个插槽都是一个 VNode 数组:

```
render: function(createElement) {
    // `<div><slot></slot></div>
    return createElement('div', this.$slots.default)
}
```

Added: 插槽的内容在 `./02-深入了解组件-拆分 --> 2.4 插槽` 讲解的, 但是那时都是用单文件组件写的, 此处使用 2.4 插槽的示例, 跟改为 `*.jsx` 的写法, 以便对比查看. 代码如下:

```
<template>
    <div class="default-div">
        <h2>4.3.4.4 JSX 插槽</h2>
        <slot-jsx>
            jsx 插槽
        </slot-jsx>
    </div>
</template>
<script>
    import SlotJsx from './slot-jsx';
    export default {
        name: 'Demo4344',
        components: {
            SlotJsx,
        }
    }
</script>
```

```
// - slot-jsx.jsx
export default {
    name: 'SlotJsx',
    render: function(createElement) {
        return createElement('div', this.$slots.default)
    }
}
```

- (2) 你也可以通过 `this.$scopedSlots` 访问作用域插槽 (`./02-深入了解组件-拆分 --> 2.4.5 作用域插槽`), 每个作用域插槽都是一个返回若干 VNode 的函数. 下面是 Vue 官网文档给出的示例代码:

```
    props: ['message'],
    render: function(createElement) {
        // `<div><slot :text="message"></slot></div>
        return createElement('div', [
            this.$scopedSlots.default({
                text: this.message
            })
        ])
    }
```

Tip: 上面这几行代码当我在单文件组件中使用时很是困惑, 既然上面说是在 "作用域插槽" 中使用, 那么我就来贴出 `2.4.5 作用域插槽` 文档讲解的单文件组件示例代码, 下面是截图:

作用域插槽

```
<template>
    <div id="Demo244" class="default-div">
        <h2>2.4.5 作用域插槽 (Scoped Slot)</h2>
        <current-user>
            <template v-slot:default="slotProps">
                {{ slotProps.user.firstName }}
            </template>
        </current-user>
    </div>
</template>
<script>
    import CurrentUser from './current-user';
    export default {
        name: 'Demo245',
        components: {
            CurrentUser,
        }
    }
</script>
```

```
<template>
    <div>
        <slot v-bind:user="user">
            {{ user.lastName }}
        </slot>
    </div>
</template>
<script>
    export default {
        name: 'CurrentUser',
        data() {
            return {
                user: {
                    firstName: 'Evan',
                    lastName: 'You'
                }
            }
        }
    }
</script>
```

slotProps.user.firstName
会替换 user.lastName

有没有发现异常? 截图中作用域插槽示例目的是让父组件方便访问子组件的数据而设计的, 但是上面的 `props: ['message']` 这明显是父组件传递的 `prop`, 而且 `this.$scopedSlots.default` 访问的也是默认插槽, 所以这里.....

- (3) 如果要用渲染函数向子组件中传递作用域插槽, 可以利用 VNode 数据对象中的 `scopedSlots` 字段:

```
render: function(createElement) {
    // `<div><child v-slot="props"><span>{{ props.text}}</span></child></div>
    return createElement('div', [
        // - 在数据对象中传递 `scopedSlots`
        //   格式为 {name; props => VNode | Array<VNode>}
        scopedSlots: {
            default: function(props) {
                return createElement('span', props.text)
            }
        }
    ])
}
```

4.3.5 JSX

- 如果你写了很多 `render` 函数, 可能会觉得下面这样的代码写起来很痛苦:

```
createElement(
    'anchored-heading',
    {
        props: {
            level: 1
        }
    },
    [
        createElement('span', 'Hello'),
        ' World!`
    ]
)
```

特别是对应的模板如此简单的情况下:

```
<anchored-heading :level="1">
    <span>Hello</span> world!
</anchored-heading>
```

这就是为什么会有一个 Babel 插件, 用于在 Vue 中使用 JSX 语法, 它可以让我们回到更接近于模板的语法上.

```
import AnchoredHeading from './AnchoredHeading.vue'

new Vue({
    el: '#demo',
    render: function (h) {
        return (
        <AnchoredHeading level={1}>
            <span>Hello</span> world!
        </AnchoredHeading>
        )
    }
})
```

WARNING: 将 `h` 作为 `createElement` 的别名是 Vue 生态系统中的一个通用惯例, 实际上也是 JSX 所要求的. 从 Vue 的 Babel 插件的 3.4.0 版本开始, 我们会在以 ES2015 语法声明的含有 JSX 的任何方法和 getter 中 (不是函数或箭头函数中) 自动注入 `const h = this.$createElement`, 这样你就可以去掉 `(h)` 参数了. 对于更早版本的插件, 如果 `h` 在当前作用域中不可用, 应用会抛错.

要了解更多关于 JSX 如何映射到 JavScript, 请阅读 使用文档

4.3.6 函数式组件

- 之前创建的锚点标题组件是比较简单, 没有管理任何状态, 也没有监听任何传递给它的状态, 也没有生命周期方法. 实际上, 它只是一个接受一些 prop 的函数. 在这样的场景下, 我们可以将组件标记为 `functional`, 这意味它无状态(没有 响应式数据), 也没有实例(没有 `this` 上下文). 一个函数式组件就像这样:

```
Vue.component('my-component', {
    functional: true,
    // - Props 是可选的
    props: {
        // ...
    },
    // - 为了弥补缺少的实例提供第二个参数作为上下文
    render: function (createElement, context) {
        // ...
    }
})
```

注意: 在 2.3.0 之前的版本中, 如果一个函数式组件想要接收 prop, 则 `props` 选项是必须的. 在 2.3.0 或以上的版本中, 你可以省略 props 选项, 所有组件上的 attribute 都会被自动隐式解析为 prop.
当使用函数式组件时, 该引用将会是 HTMLElement, 因为他们是无状态的也是无实例的.

在 2.5.0 及以上版本中, 如果你使用了 单文件组件, 那么基于模板的函数式组件可以这样声明:

```
<template functional>
</template>
```

组件需要的一切都通过 `context` 参数传递, 它是一个包括如下字段的对象:

- (1) `props`: 提供所有 prop 的对象
- (2) `children`: VNode 子节点的数组
- (3) `slots`: 一个函数, 返回了包含所有插槽的对象
- (4) `scopedSlots`: (2.6.0+) 一个暴露传入的作用域插槽的对象. 也可以函数形式暴露普通插槽.
- (5) `data`: 传递给组件的整个 数据对象 (`4.3.3.1 深入对象`), 作为 `createElement` 的第二个参数传入组件.
- (6) `parent`: 对父组件的引用.
- (7) `listeners`: (2.3.0+) 一个包含了所有父组件为当前组件注册的事件监听器对象. 这是 `data.on` 的一个别名.

- (8) `injections`: (2.3.0+) 如果使用了 `inject` 选项, 则该对象包含了应当被注入的 property.

在添加 `functional: true` 之后, 需要更新我们的锚点标题组件的渲染函数, 为其增加 `context` 参数, 并将 `this.$slots.default` 更新为 `context.children`, 然后将 `this.level` 更新为 `context.props.level`.

因为函数式组件只是函数, 所以渲染开销也低很多.

在作为包装组件时它们也同样非常有用. 比如, 当你需要做这些时:

- 程序化地在多个组件中选择一个来代为渲染;
- 在将 `children`、`props`、`data` 传递给子组件之前操作它们.

下面是一个 `smart-list` 组件的例子, 它能根据传入的 prop 的值来代为渲染更具体的组件:

```
  var EmptyList = { /* ... */ }
  var TableList = { /* ... */ }
  var OrderedList = { /* ... */ }
  var UnorderedList = { /* ... */ }

  // - 由于这个示例不完整, 就不再单独创建单文件组件, 大致意思就是: `smart-list`
  //   是上面几个组件的公共组件, 上面每个组件会传递 2 个 prop 给当前
  //   `smart-list`, 当前组件再根据数据做渲染, 完成后返回渲染部分给每个调用组件.
  //   这个示例在项目中可能会是一个 3 级的组件调用, 比如:
  //
  //
  //   4.3.6.vue -- import EmptyList/TableList/../.. from 'xxx'--> import
SmartList from 'xxx'
  //
  //
  Vue.component('smart-list', {
  functional: true,
  props: {
      items: {
      type: Array,
      required: true
      },
      isOrdered: Boolean
  },
  render: function (createElement, context) {
      function appropriateListComponent () {
      var items = context.props.items

      if (items.length === 0)           return EmptyList
      if (typeof items[0] === 'object') return TableList
      if (context.props.isOrdered)      return OrderedList

      return UnorderedList
      }

      return createElement(
      appropriateListComponent(),
```

```
        context.data,
        context.children
        )
    }
    })
```

4.3.6.1 向子元素或子组件传递 `attribute` 和 事件

- 在普通组件中, 没有被定义为 prop 的特性(attribute) 会自动添加到组件的根元素上, 将已有的同名特性(attribute) 进行替换或与其进行 `智能合并(1.6 class 与 style 绑定)`.

  然而函数式组件要求你显式定义该行为:

```
Vue.component('my-functional-button', {
    functional: true,
    render: function(createElement, context) {
        // - 完全透明任何特性(attribute): 事件监听器, 子节点等.
        // - tip: 这里参考下面添加到了 functional 的单位文组件来做对比,
        //   此处说的就是 `context` 参数包含了 `button` 按钮上已定义的
        //   `class="btn btn-primary" v-bind="data.attrs"
        //   v-on=listeners` 这些已经定义的所有 HTML 特性.
        return createElement('button', context.data, context.children)
    }
})
```

  通过向 `createElement` 传入 `context.data` 作为第二个参数, 我们就把 `my-functional-button` 上面所有的特性(attribute)和事件监听器都传递下去了. 事实上这是非常透明的, 以至于那些事件甚至并不要求 `.native` 修饰符.

  如果你使用基于模板的函数时组件, 那么你还需要收到添加特性(attribute)和监听器, 因为我们可以访问到其独立的上下文内容, 所以我们可以使用 `data.attrs` 传递任何 HTML 特性(attribute), 也可以使用 `listeners`(即: `data.on` 的别名)传递任何事件监听器.

```
<template functional>
    <button
        class="btn btn-primary"
        v-bind="data.attrs"
        v-on="listeners"
    >
        <slot/>
    </button>
</template>
```

4.3.6.2 `slots()` 和 `children` 对比

- 你可能想知道为什么同时需要 `slots()` 和 `children`. `slots().default` 不是和 `children` 类似的吗? 在一些场景中, 是这样 —— 但如果是如下的带有子节点的函数式组件呢?

```
<my-functional-component>
    <p v-slot:foo>
        first
    </p>
    <p>second</p>
</my-functional-component>
```

对于这个组件, `children` 会给你两个段落标签, 而 `slots().default` 只会传递第二个匿名段落标签, `slots().foo` 会传递第一个具名段落标签. 同时拥有 `children` 和 `slots()`, 因此你可以选择让组件感知某个插槽机制, 还是简单地通过传递 `children`, 移交给其它组件去处理.

4.3.7 模板编译

- 你可能会有兴趣知道, Vue 的模板实际上被编译成了渲染函数. 这是一个实现细节, 通常不需要关心. 但如果你想看看模板的功能具体是怎样被编译的, 可能会发现会非常有意思. 下面是一个使用 Vue.compile 来实时编译模板字符串的简单示例:

```
<div>
    <header>
        <h1>I'm a template!</h1>
    </header>
    <p v-if="message">{{ message }}</p>
    <p v-else>No message.</p>
</div>
```

render:

```
function anonymous(){
    width(this) {
        return _c('div', [
            _m(0),
            (message)
                ? _c('p', [_(_s(message))])
                : _c('p', [_v("No message.")])
        ])
    }
}
```

staticRenderFns(静态渲染函数):

```
_m(0): function anonymous() {
    with(this) {
        return _c('header', [_c('h1', [_v("I'm a template!")])])
    }
}
```

4.4 插件

- 插件通常用来为 Vue 添加全局功能. 插件的功能范围没有严格的限制 -- 一般有下面几种:
    - (1) 添加全局方法或者 property. 如: **vue-custom-element**

- (2) 添加全局资源: 指令/过滤器/过渡等. 如: `vue-touch` (注: 这个作者已经不再维护了)
- (3) 通过全局混入来添加一些组件选项. 如: **vue-router**
- (4) 添加 Vue 实例方法, 通过把它们添加到 `Vue.prototype` 上实现.
- (5) 一个库, 提供自己的 API, 同时提供上面提到的一个或多个功能. 如 vue-router

4.4.1 使用插件

- 通过全局方法 `Vue.use()` 使用插件. 它需要在你调用 `new Vue()` 启动应用之前完成:

```
// - 调用 `MyPlugin.install(Vue)`
Vue.use(MyPlugin)

new Vue({
    // ... 组件选项
})
```

也可以传入一个可选的选项对象:

```
Vue.use(MyPlugin, {someOption; true})
```

Vue.use 会自动阻止多次注册相同插件, 届时即使多次调用也只会注册一次该插件.

Vue.js 官方提供的一些插件 (例如 `vue-router`) 在检测到 `Vue` 是可访问的全局变量时会自动调用 `Vue.use()`. 然而在像 CommonJS 这样的模块环境中, 你应该始终显式地调用 `Vue.use()`:

```
// - 用 Browserify 或 webpack 提供的 CommonJS 模块环境时
var Vue = require('vue');
var VueRouter = require('vue-router');

// - 不要忘了调用此方法
Vue.use(VueRouter);
```

awesome-vue 集合了大量由社区贡献的插件和库.

4.4.2 开发插件

- Vue.js 的插件应该暴露一个 `install` 方法. 这个方法的第一个参数是 `Vue` 构造器, 第二个参数是一个可选的选项对象:

```
MyPlugin.install = function(Vue, options) {
    // - 1. 添加全局方法 或 property
    Vue.myGlobalMethod = function() {
        // - 逻辑...
    };

    // - 2. 添加全局资源
    Vue.directive('my-directive', {
        bind(el, binding, vnode, oldVnode) {
            // - 逻辑
        }
        // ...
```

```
    });

    // - 3. 注入组件选项
    Vue.mixin({
        created; function() {
            // - 逻辑...
        }
        // ...
    })

    // - 4. 添加实例方法
    Vue.prototype.$myMethod = function(methodOptions) {
        // - 逻辑...
    }
}
```

4.5 过滤器

- Vue.js 允许你自定义过滤器, 可被用于一些常见的文本格式化. 过滤器可以用在 2 个地方: **双花括号插值** 和 ** `v-bind` 表达式**(后者从 2.1.0+ 开始支持). 过滤器应该被添加在 JavaScript 表达式的尾部, 由 `管道` 符号指示:

```
<!-- 在双花括号中 -->
{{ message | capitalize }}

<!-- 在 `v-bind` 中 -->
<div v-bind:id="rawId | formatId"></div>
```

你可以在一个组件的选项中定义本地的过滤器:

```
filters: {
    capitalize: function(value) {
        if (!value) return '';
        value = value.toString();
        return value.charAt(0).toUpperCase() + value.slice(1);
    }
}
```

或者在创建 Vue 实例之前全局定义过滤器:

```
Vue.filter('capitalize', function(value) {
    if (!value) return '';
    value = value.toString();
    return value.charAt(0).toUpperCase() + value.slice(1);
})
new Vue({
    // ...
})
```

当全局过滤器和局部过滤器重名时, 会采用局部过滤器. 下面这个例子用到了 `capitalize` 过滤器:

```
<template>
    <div class="default-div">
    <h2>4.5 过滤器</h2>
    <p>
        <input type="text" v-model="str">
    </p>
    {{ str | capitalize }}
    </div>
</template>
<script>
    export default {
        name: 'Demo423',
        data() {
            return {
                str: 'john'
            }
        },
        filters: {
            capitalize: function(value) {
                if (!value) return '';
                // - `charAt(num)`: 以单字符字符串的形式返回给定位置的那个字符.
                return value.charAt(0).toUpperCase() + value.slice(1);
            }
        }
    }
</script>
<style scoped></style>
```

过滤器函数总接受表达式的值(之前的操作链的结果)作为第一个参数. 在上述例子中, `capitalize` 过滤器函数将会收到 `message` 的值作为第一个参数.

过滤器可以串联

```
{{ message | filterA | filterB }}
```

在这个例子中, `filterA` 被定义为接收单个参数的过滤器函数, 表达式 `message` 的值将作为参数传入到函数中. 然后继续调用同样被定义为接收单个参数的过滤器函数 `filterB`, 将 `filterA` 的结果传递到 `filterB` 中.

过滤器是 JavaScript 函数, 因此可以接收参数:

```
{{ message | filterA('arg1', arg2) }}
```

这里, `filterA` 被定义为接收三个参数的过滤器函数. 其中 `message` 的值作为第一个参数, 普通字符串 `'arg1'` 作为第二个参数, 表达式 `arg2` 的值作为第三个参数.