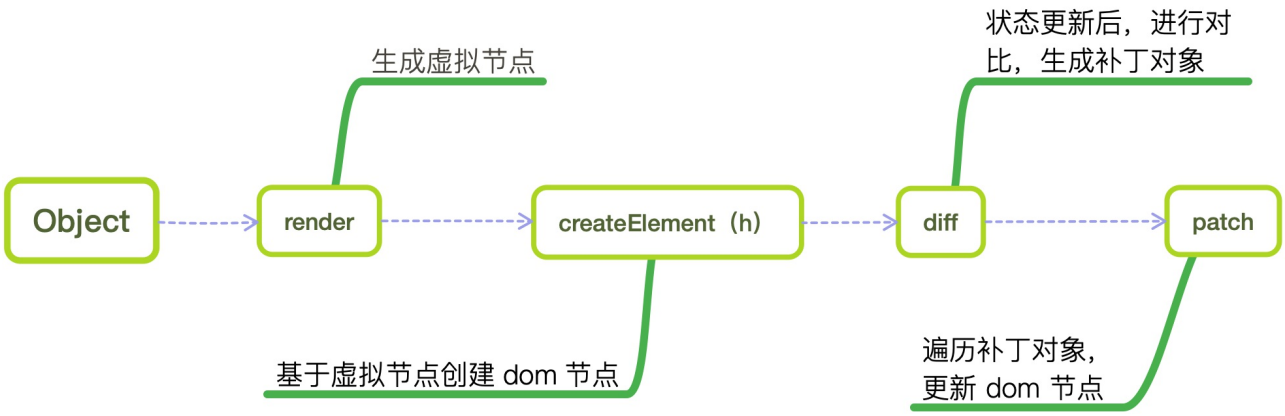
更灵活的组件：Render 函数与 Functional Render

Vue.js 2.x 与 Vue.js 1.x 最大的区别就在于 2.x 使用了 Virtual DOM（虚拟 DOM）来更新 DOM 节点，提升渲染性能。

一般来说，我们写 Vue.js 组件，模板都是写在 <template> 内的，但它并不是最终呈现的内容，template 只是一种对开发者友好的语法，能够一眼看到 DOM 节点，容易维护，在 Vue.js 编译阶段，会解析为 Virtual DOM。

与 DOM 操作相比，Virtual DOM 是基于 JavaScript 计算的，所以开销会小很多。下图演示了 Virtual DOM 运行的过程：



正常的 DOM 节点在 HTML 中是这样的：

```
<div id="main">
  <p>文本内容</p>
  <p>文本内容</p>
</div>
```

用 Virtual DOM 创建的 JavaScript 对象一般会是这样的：

```
const vNode = {
  tag: 'div',
  attributes: {
    id: 'main'
  },
  children: [
    // p 节点
  ]
}
```

vNode 对象通过一些特定的选项描述了真实的 DOM 结构。

在 Vue.js 中，对于大部分场景，使用 template 足以应付，但如果想完全发挥 JavaScript 的编程能力，或在一些特定场景下（后文介绍），需要使用 Vue.js 的 Render 函数。

Render 函数

正如上文介绍的 Virtual DOM 示例一样，Vue.js 的 Render 函数也是类似的语法，需要使用一些特定的选项，将 template 的内容改写成一个 JavaScript 对象。

对于初级前端工程师，或想快速建站的需求，直接使用 Render 函数开发 Vue.js 组件是要比 template 困难的，原因在于 Render 函数返回的是一个 JS 对象，没有传统 DOM 的层级关系，配合上 if、else、for 等语句，将节点拆分成不同 JS 对象再组装，如果模板复杂，那一个 Render 函数是难读且难维护的。所以，绝大部分组件开发和业务开发，我们直接使用 template 语法就可以了，并不需要特意使用 Render 函数，那样只会增加负担，同时也放弃了 Vue.js 最大的优势（React 无 template 语法）。

很多学习 Vue.js 的开发者在遇到 Render 函数时都有点”躲避“，或直接放弃这部分，这并没有问题，因为不用 Render 函数，照样可以写出优秀的 Vue.js 程序。不过，Render 函数并没有想象中的那么复杂，只是配置项特别多，一时难以记住，但归根到底，Render 函数只有 3 个参数。

来看一组 template 和 Render 写法的对照：

```
<template>
  <div id="main" class="container" style="color:
red">
    <p v-if="show">内容 1</p>
    <p v-else>内容 2</p>
  </div>
</template>
<script>
  export default {
    data () {
      return {
        show: false
      }
    }
  }
</script>
```

```
export default {
  data () {
    return {
      show: false
    }
  },
  render: (h) => {
    let childNode;
    if (this.show) {
      childNode = h('p', '内容 1');
    } else {
      childNode = h('p', '内容 2');
    }

    return h('div', {
      attrs: {
        id: 'main'
      },
      class: {
        container: true
      },
      style: {
        color: 'red'
      }
    }, [childNode]);
  }
}
```

这里的 h，即 createElement，是 Render 函数的核心。可以看到，template 中的 **v-if / v-else** 等指令，都被 JS 的 **if / else** 替代了，那 **v-for** 自然也会被 **for** 语句替代。

h 有 3 个参数，分别是：

1. 要渲染的元素或组件，可以是一个 html 标签、组件选项或一个函数（不常用），该参数为必填项。示例：

```
// 1. html 标签
h('div');
// 2. 组件选项
import DatePicker from '../component/date-picker.vue';
h(DatePicker);
```

2. 对应属性的数据对象，比如组件的 props、元素的 class、绑定的事件、slot、自定义指令等，该参数是可选的，上文所说的 Render 配置项多，指的就是这个参数。该参数的完整配置和示例，可以到 Vue.js 的文档查看，没必要全部记住，用到时查阅就好：[createElement 参数](https://cn.vuejs.org/v2/guide/render-function.html#createElement-参数)。

3. 子节点，可选，String 或 Array，它同样是一个 h。示例：

```
[
  '内容',
  h('p', '内容'),
  h(Component, {
    props: {
      someProp: 'foo'
    }
  })
]
```

约束

所有的组件树中，如果 vNode 是组件或含有组件的 slot，那么 vNode 必须唯一。以下两个示例都是**错误**的：

```
// 局部声明组件
const Child = {
  render: (h) => {
    return h('p', 'text');
  }
}

export default {
  render: (h) => {
    // 创建一个子节点，使用组件 Child
    const ChildNode = h(Child);

    return h('div', [
      ChildNode,
      ChildNode
    ]);
  }
}
```

```
{
  render: (h) => {
    return h('div', [
      this.$slots.default,
      this.$slots.default
    ])
  }
}
```

重复渲染多个组件或元素，可以通过一个循环和工厂函数来解决：

```
const Child = {
  render: (h) => {
    return h('p', 'text');
  }
}

export default {
  render: (h) => {
    const children = Array.apply(null, {
      length: 5
    }).map(() => {
      return h(Child);
    });
    return h('div', children);
  }
}
```

对于含有组件的 slot，复用比较复杂，需要将 slot 的每个子节点都克隆一份，例如：

```
{
  render: (h) => {
    function cloneVNode (vnode) {
      // 递归遍历所有子节点，并克隆
      const clonedChildren = vnode.children &&
vnode.children.map(vnode => cloneVNode(vnode));
      const cloned = h(vnode.tag, vnode.data,
clonedChildren);
      cloned.text = vnode.text;
      cloned.isComment = vnode.isComment;
      cloned.componentOptions =
vnode.componentOptions;
      cloned.elm = vnode.elm;
```

```
      cloned.context = vnode.context;
      cloned.ns = vnode.ns;
      cloned.isStatic = vnode.isStatic;
      cloned.key = vnode.key;

      return cloned;
    }

    const vNodes = this.$slots.default ===
undefined ? [] : this.$slots.default;
    const clonedVNodes = this.$slots.default ===
undefined ? [] : vNodes.map(vnode =>
cloneVNode(vnode));

    return h('div', [
      vNodes,
      clonedVNodes
    ])
  }
}
```

在 Render 函数里创建了一个 cloneVNode 的工厂函数，通过递归将 slot 所有子节点都克隆了一份，并对 VNode 的关键属性也进行了复制。

深度克隆 slot 并非 Vue.js 内置方法，也没有得到推荐，属于黑科技，在一些特殊的场景才会使用到，正常业务几乎是用不到的。比如 iView 组件库的穿梭框组件 Transfer，就用到了这种方法：

它的使用方法是：

```
<Transfer
    :data="data"
    :target-keys="targetKeys"
    :render-format="renderFormat">
  <div :style="{float: 'right', margin: '5px'}">
    <Button size="small"
@click="reloadMockData">Refresh</Button>
  </div>
</Transfer>
```

示例中的默认 slot 是一个 Refresh 按钮，使用者只写了一遍，但在 Transfer 组件中，是通过克隆 VNode 的方法，显示了两遍。如果不这样做，就要声明两个具名 slot，但是左右两个的逻辑可能是完全一样的，使用者就要写两个一模一样的 slot，这是不友好的。

> Render 函数的基本用法还有很多，比如 v-model 的用法、事件和修饰符、slot 等，读者可以到 Vue.js 文档阅读。[Vue.js 渲染函数 (https://cn.vuejs.org/v2/guide/render-function.html)](https://cn.vuejs.org/v2/guide/render-function.html)

Render 函数使用场景

上文说到，一般情况下是不推荐直接使用 Render 函数的，使用 template 足以，在 Vue.js 中，使用 Render 函数的场景，主要有以下 4 点：

1. 使用两个相同 slot。在 template 中，Vue.js 不允许使用两个相同的 slot，比如下面的示例是错误的：

   ```
   <template>
     <div>
       <slot></slot>
       <slot></slot>
     </div>
   </template>
   ```

   解决方案就是上文中讲到的**约束**，使用一个深度克隆 VNode 节点的方法。

2. 在 SSR 环境（服务端渲染），如果不是常规的 template 写法，比如通过 Vue.extend 和 new Vue 构造来生成的组件实例，是编译不过的，在前面小节也有所介绍。回顾上一节的 `$Alert` 组件的 notification.js 文件，当时是使用 Render 函数来渲染 Alert 组件，如果改成另一种写法，在 SSR 中会报错，对比两种写法：

```
// 正确写法
import Alert from './alert.vue';
import Vue from 'vue';

Alert.newInstance = properties => {
  const props = properties || {};

  const Instance = new Vue({
    data: props,
    render (h) {
      return h(Alert, {
        props: props
      });
    }
  });

  const component = Instance.$mount();
  document.body.appendChild(component.$el);

  const alert = Instance.$children[0];

  return {
    add (noticeProps) {
      alert.add(noticeProps);
    },
    remove (name) {
      alert.remove(name);
    }
  }
};

export default Alert;
```

```
// 在 SSR 下报错的写法
import Alert from './alert.vue';
import Vue from 'vue';

Alert.newInstance = properties => {
  const props = properties || {};

  const div = document.createElement('div');
  div.innerHTML = `<Alert ${props}></Alert>`;
  document.body.appendChild(div);

  const Instance = new Vue({
    el: div,
    data: props,
    components: { Alert }
  });

  const alert = Instance.$children[0];

  return {
    add (noticeProps) {
      alert.add(noticeProps);
    },
    remove (name) {
      alert.remove(name);
    }
  }
};

export default Alert;
```

3. 在 runtime 版本的 Vue.js 中，如果使用 Vue.extend 手动构造一个实例，使用 template 选项是会报错的，在第 9 节中也有所介绍。解决方案也很简单，把 template 改写为 Render 就可以了。需要注意的是，在开发独立组件时，可以通过配置 Vue.js 版本来使 template 选项可用，但这是在自己的环境，无法保证使用者的 Vue.js 版本，所以对于提供给他人用的组件，是需要考虑兼容 runtime 版本和 SSR 环境的。

4. 这可能是使用 Render 函数最重要的一点。一个 Vue.js 组件，有一部分内容需要从父级传递来显示，如果是文本之类的，直接通过 props 就可以，如果这个内容带有样式或复杂一点的 html 结构，可以使用 v-html 指令来渲染，父级传递的仍然是一个 HTML Element 字符串，不过它仅仅是能解析正常的 html 节点且有 XSS 风险。当需要最大化程度自定义显示内容时，就需要 Render 函数，它可以渲染一个完整的 Vue.js 组件。你可能会说，用 slot 不就好了？的确，slot 的作用就是做内容分发的，但在一些特殊组件中，可能 slot 也不行。比如一个表格组件 Table，它只接收两个 props：列配置 columns 和行数据 data，不过某一列的单元格，不是只将数据显示出来那么简单，可能带有一些复杂的操作，这种场景只用 slot 是不行的，没办法确定是那一列的 slot。这种场景有两种解决方案，其一就是 Render 函数，下一节的实战就是开发这样一个 Table 组件；另一种是用作用域 slot（slot-scope），后面小节也会详细介绍。

Functional Render

Vue.js 提供了一个 functional 的布尔值选项，设置为 true 可以使组件无状态和无实例，也就是没有 data 和 this 上下文。这样用 Render 函数返回虚拟节点可以更容易渲染，因为函数化组件（Functional Render）只是一个函数，渲染开销要小很多。

使用函数化组件，Render 函数提供了第二个参数 context 来提供临时上下文。组件需要的 data、props、slots、children、parent 都是通过这个上下文来传递的，比如 this.level 要改写为 context.props.level，this.$slots.default 改写为 context.children。

> 您可以阅读 Vue.js 文档—函数式组件 (https://cn.vuejs.org/v2/guide/render-function.html#函数式组件) 来查看示例。

函数化组件在业务中并不是很常用，而且也有类似的方法来实现，比如某些场景可以用 is 特性来动态挂载组件。函数化组件主要适用于以下两个场景：

- 程序化地在多个组件中选择一个；
- 在将 children、props、data 传递给子组件之前操作它们。

比如上文说过的，某个组件需要使用 Render 函数来自定义，而不是通过传递普通文本或 v-html 指令，这时就可以用 Functional Render，来看下面的示例：

1. 首先创建一个函数化组件 **render.js**：

```
// render.js
export default {
  functional: true,
  props: {
    render: Function
  },
  render: (h, ctx) => {
    return ctx.props.render(h);
  }
};
```

它只定义了一个 props：render，格式为 Function，因为是 functional，所以在 render 里使用了第二个参数 ctx 来获取 props。这是一个中间文件，并且可以复用，其它组件需要这个功能时，都可以引入它。

2. 创建组件：

```
<!-- my-component.vue -->
<template>
  <div>
    <Render :render="render"></Render>
  </div>
</template>
<script>
  import Render from './render.js';

  export default {
    components: { Render },
    props: {
      render: Function
    }
  }
</script>
```

3. 使用上面的 my-compoennt 组件：

```
<!-- demo.vue -->
<template>
  <div>
    <my-component :render="render"></my-
component>
  </div>
</template>
<script>
  import myComponent from '../components/my-
component.vue';

  export default {
    components: { myComponent },
    data () {
      return {
        render: (h) => {
          return h('div', {
            style: {
              color: 'red'
            }
          }, '自定义内容');
        }
      }
    }
  }
</script>
```

这里的 render.js 因为只是把 demo.vue 中的 Render 内容过继，并无其它用处，所以用了 Functional Render。

就此例来说，完全可以用 slot 取代 Functional Render，那是因为只有 render 这一个 prop。如果示例中的 <Render> 是用 v-for 生成的，也就是多个时，用 一个 slot 是实现不了的，那时用

Render 函数就很方便了，后面章节会专门介绍。

结语

如果想换一种思路写 Vue.js，就试试 Render 函数吧，它会让你“又爱又恨”！

注：本节部分内容参考了《Vue.js 实战》（清华大学出版社），部分代码参考 iView (https://github.com/iview/iview/blob/2.0/src/components/tr