17.2. 如何用

`vue` 中的过滤器可以用在两个地方：双花括号插值和 `v-bind` 表达式，过滤器应该被添加在 `JavaScript` 表达式的尾部，由“管道”符号指示：

JavaScript | 复制代码

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

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

17.2.1. 定义filter

在组件的选项中定义本地的过滤器

JavaScript | 复制代码

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

定义全局过滤器：

JavaScript | 复制代码

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

new Vue({
  // ...
})
```

注意：当全局过滤器和局部过滤器重名时，会采用局部过滤器

过滤器函数总接收表达式的值 (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：

Plain Text 复制代码

```
{{ 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：

Plain Text 复制代码

```
{{ 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

举个例子：

HTML 复制代码

```
<div id="app">
    <p>{{ msg | msgFormat('疯狂','--')}}</p>
</div>

<script>
    // 定义一个 Vue 全局的过滤器，名字叫做  msgFormat
    Vue.filter('msgFormat', function(msg, arg, arg2) {
        // 字符串的  replace 方法，第一个参数，除了可写一个 字符串之外，还可以定义一个正则
        return msg.replace(/单纯/g, arg+arg2)
    })
</script>
```

17.2.2. 小结：

- 部过滤器优先于全局过滤器被调用
- 一个表达式可以使用多个过滤器。过滤器之间需要用管道符“|”隔开。其执行顺序从左往右

17.3. 应用场景

平时开发中，需要用到过滤器的地方有很多，比如单位转换、数字打点、文本格式化、时间格式化之类的等

比如我们要实现将30000 => 30,000，这时候我们就需要使用过滤器

JavaScript 复制代码

```
Vue.filter('toThousandFilter', function (value) {
      if (!value) return ''
      value = value.toString()
      return .replace(str.indexOf('.') > -1 ? /(\d)(?=(\d{3})+\.)/g : /(\d)(?=(?:\d{3})+$)/g, '$1,')
})
```

17.4. 原理分析

使用过滤器

JavaScript 复制代码

```
{{ message | capitalize }}
```

在模板编译阶段过滤器表达式将会被编译为过滤器函数，主要是用过 `parseFilters` ，我们放到最后讲

JavaScript 复制代码

```
_s(_f('filterFormat')(message))
```

首先分析一下 `_f` ：

_f 函数全名是： `resolveFilter` ，这个函数的作用是从 `this.$options.filters` 中找出注册的过滤器并返回

JavaScript 复制代码

```
// 变为
this.$options.filters['filterFormat'](message) // message为参数
```

关于 `resolveFilter`

```JavaScript
import { indentity,resolveAsset } from 'core/util/index'

export function resolveFilter(id){
    return resolveAsset(this.$options,'filters',id,true) || identity
}
```

内部直接调用 `resolveAsset` ，将 `option` 对象，类型，过滤器 `id` ，以及一个触发警告的标志作为参数传递，如果找到，则返回过滤器；

`resolveAsset` 的代码如下：

```JavaScript
export function resolveAsset(options,type,id,warnMissing){ // 因为我们找的是过滤器，所以在 resolveFilter函数中调用时 type 的值直接给的 'filters',实际这个函数还可以拿到其他很多东西
    if(typeof id !== 'string'){ // 判断传递的过滤器id 是不是字符串，不是则直接返回
        return
    }
    const assets = options[type]  // 将我们注册的所有过滤器保存在变量中
    // 接下来的逻辑便是判断id是否在assets中存在，即进行匹配
    if(hasOwn(assets,id)) return assets[id] // 如找到，直接返回过滤器
    // 没有找到，代码继续执行
    const camelizedId  = camelize(id) // 万一你是驼峰的呢
    if(hasOwn(assets,camelizedId)) return assets[camelizedId]
    // 没找到，继续执行
    const PascalCaseId = capitalize(camelizedId) // 万一你是首字母大写的驼峰呢
    if(hasOwn(assets,PascalCaseId)) return assets[PascalCaseId]
    // 如果还是没找到，则检查原型链(即访问属性)
    const result = assets[id] || assets[camelizedId] || assets[PascalCaseId]
    // 如果依然没找到，则在非生产环境的控制台打印警告
    if(process.env.NODE_ENV !== 'production' && warnMissing && !result){
        warn('Failed to resolve ' + type.slice(0,-1) + ': ' + id, options)
    }
    // 无论是否找到，都返回查找结果
    return result
}
```

下面再来分析一下 `_s` ：

`_s` 函数的全称是 `toString` ,过滤器处理后的结果会当作参数传递给 `toString` 函数，最终 `toString` 函数执行后的结果会保存到 `Vnode` 中的text属性中，渲染到视图中

```JavaScript
function toString(value){
    return value == null
    ? ''
    : typeof value === 'object'
      ? JSON.stringify(value,null,2)// JSON.stringify()第三个参数可用来控制字符串里面的间距
      : String(value)
}
```

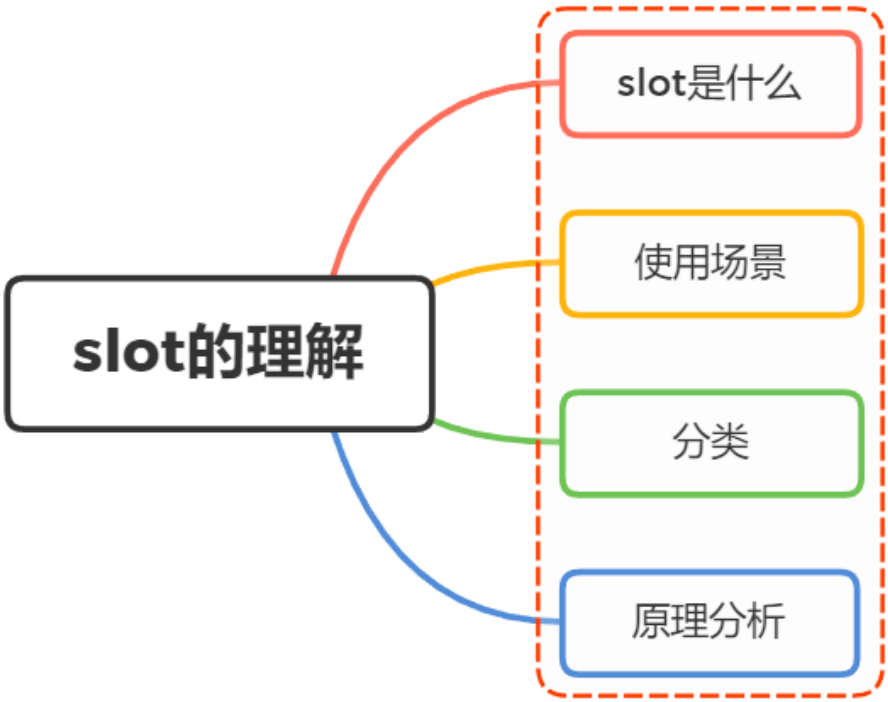
最后，在分析下 `parseFilters` ，在模板编译阶段使用该函数阶段将模板过滤器解析为过滤器函数调用表达式

```JavaScript
function parseFilters (filter) {
    let filters = filter.split('|')
    let expression = filters.shift().trim() // shift()删除数组第一个元素并将其返回，该方法会更改原数组
    let i
    if (filters) {
        for(i = 0;i < filters.length;i++){
            experssion = warpFilter(expression,filters[i].trim()) // 这里传进去的expression实际上是管道符号前面的字符串，即过滤器的第一个参数
        }
    }
    return expression
}
// warpFilter函数实现
function warpFilter(exp,filter){
    // 首先判断过滤器是否有其他参数
    const i = filter.indexof('(')
    if(i<0){ // 不含其他参数，直接进行过滤器表达式字符串的拼接
        return `_f("${filter}")(${exp})`
    }else{
        const name = filter.slice(0,i) // 过滤器名称
        const args = filter.slice(i+1) // 参数，但还多了 ‘)’
        return `_f('${name}')(${exp},${args}` // 注意这一步少给了一个 ')'
    }
}
```

17.5. 小结：

- 在编译阶段通过 `parseFilters` 将过滤器编译成函数调用（串联过滤器则是一个嵌套的函数调用，前一个过滤器执行的结果是后一个过滤器函数的参数）
- 编译后通过调用 `resolveFilter` 函数找到对应过滤器并返回结果
- 执行结果作为参数传递给 `toString` 函数，而 `toString` 执行后，其结果会保存在 `Vnode` 的 `text` 属性中，渲染到视图

18. 说说你对slot的理解？slot使用场景有哪些?



18.1. slot是什么

在HTML中 `slot` 元素，作为 `Web Components` 技术套件的一部分，是Web组件内的一个占位符

该占位符可以在后期使用自己的标记语言填充

举个栗子

```HTML
<template id="element-details-template">
  <slot name="element-name">Slot template</slot>
</template>
<element-details>
  <span slot="element-name">1</span>
</element-details>
<element-details>
  <span slot="element-name">2</span>
</element-details>
```

`template` 不会展示到页面中，需要用先获取它的引用，然后添加到 `DOM` 中，

```JavaScript
customElements.define('element-details',
  class extends HTMLElement {
    constructor() {
      super();
      const template = document
        .getElementById('element-details-template')
        .content;
      const shadowRoot = this.attachShadow({mode: 'open'})
        .appendChild(template.cloneNode(true));
    }
  })
```

在 `Vue` 中的概念也是如此

`Slot` 艺名插槽，花名“占坑”，我们可以理解为 `solt` 在组件模板中占好了位置，当使用该组件标签时候，组件标签里面的内容就会自动填坑（替换组件模板中 `slot` 位置），作为承载分发内容的出口

可以将其类比为插卡式的FC游戏机，游戏机暴露卡槽（插槽）让用户插入不同的游戏磁条（自定义内容）

放张图感受一下

18.2. 使用场景

通过插槽可以让用户可以拓展组件，去更好地复用组件和对其做定制化处理

如果父组件在使用到一个复用组件的时候，获取这个组件在不同的地方有少量的更改，如果去重写组件是一件不明智的事情

通过 `slot` 插槽向组件内部指定位置传递内容，完成这个复用组件在不同场景的应用

比如布局组件、表格列、下拉选、弹框显示内容等

18.3. 分类

`slot` 可以分来以下三种：

- 默认插槽
- 具名插槽
- 作用域插槽

18.3.1. 默认插槽

子组件用 `<slot>` 标签来确定渲染的位置，标签里面可以放 `DOM` 结构，当父组件使用的时候没有往插槽传入内容，标签内 `DOM` 结构就会显示在页面

父组件在使用的时候，直接在子组件的标签内写入内容即可

子组件 `Child.vue`

HTML 复制代码

```
<template>
    <slot>
      <p>插槽后备的内容</p>
    </slot>
</template>
```

父组件

HTML 复制代码

```
<Child>
  <div>默认插槽</div>
</Child>
```

18.3.2. 具名插槽

子组件用 `name` 属性来表示插槽的名字，不传为默认插槽

父组件中在使用时在默认插槽的基础上加上 `slot` 属性，值为子组件插槽 `name` 属性值

子组件 `Child.vue`

HTML 复制代码

```
<template>
    <slot>插槽后备的内容</slot>
  <slot name="content">插槽后备的内容</slot>
</template>
```

父组件

```HTML
<child>
    <template v-slot:default>具名插槽</template>
    <!-- 具名插槽用插槽名做参数 -->
    <template v-slot:content>内容...</template>
</child>
```

18.3.3. 作用域插槽

子组件在作用域上绑定属性来将子组件的信息传给父组件使用，这些属性会被挂在父组件 `v-slot` 接受的对象上

父组件中在使用时通过 `v-slot:` （简写：#）获取子组件的信息，在内容中使用

子组件 `Child.vue`

```HTML
<template>
  <slot name="footer" testProps="子组件的值">
          <h3>没传footer插槽</h3>
    </slot>
</template>
```

父组件

```HTML
<child>
    <!-- 把v-slot的值指定为作用域上下文对象 -->
    <template v-slot:default="slotProps">
      来自子组件数据：{{slotProps.testProps}}
    </template>
    <template #default="slotProps">
      来自子组件数据：{{slotProps.testProps}}
    </template>
</child>
```

18.3.4. 小结：

- `v-slot` 属性只能在 `<template>` 上使用，但在只有默认插槽时可以在组件标签上使用
- 默认插槽名为 `default` ，可以省略default直接写 `v-slot`

- 缩写为 `#` 时不能不写参数，写成 `#default`
- 可以通过解构获取 `v-slot={user}` ，还可以重命名 `v-slot="{user: newName}"` 和定义默认值 `v-slot="{user = '默认值'}"`

18.4. 原理分析

`slot` 本质上是返回 `VNode` 的函数，一般情况下， `Vue` 中的组件要渲染到页面上需要经过 `template -> render function -> VNode -> DOM` 过程，这里看看 `slot` 如何实现：

编写一个 `buttonCounter` 组件，使用匿名插槽

JavaScript 复制代码

```
Vue.component('button-counter', {
  template: '<div> <slot>我是默认内容</slot></div>'
})
```

使用该组件

JavaScript 复制代码

```
new Vue({
    el: '#app',
    template: '<button-counter><span>我是slot传入内容</span></button-counter>',
    components:{buttonCounter}
})
```

获取 `buttonCounter` 组件渲染函数

JavaScript 复制代码

```
(function anonymous(
) {
with(this){return _c('div',[_t("default",[_v("我是默认内容")])],2)}
})
```

`_v` 表示穿件普通文本节点， `_t` 表示渲染插槽的函数

渲染插槽函数 `renderSlot` （做了简化）

```JavaScript
function renderSlot (
  name,
  fallback,
  props,
  bindObject
) {
  // 得到渲染插槽内容的函数
  var scopedSlotFn = this.$scopedSlots[name];
  var nodes;
  // 如果存在插槽渲染函数，则执行插槽渲染函数，生成nodes节点返回
  // 否则使用默认值
  nodes = scopedSlotFn(props) || fallback;
  return nodes;
}
```

`name` 属性表示定义插槽的名字，默认值为 `default` ， `fallback` 表示子组件中的 `slot` 节点的默认值

关于 `this.$scopredSlots` 是什么，我们可以先看看 `vm.slot`

```JavaScript
function initRender (vm) {
  ...
  vm.$slots = resolveSlots(options._renderChildren, renderContext);
  ...
}
```

`resolveSlots` 函数会对 `children` 节点做归类和过滤处理，返回 `slots`

JavaScript | 复制代码

```javascript
function resolveSlots (
    children,
    context
  ) {
    if (!children || !children.length) {
      return {}
    }
    var slots = {};
    for (var i = 0, l = children.length; i < l; i++) {
      var child = children[i];
      var data = child.data;
      // remove slot attribute if the node is resolved as a Vue slot node
      if (data && data.attrs && data.attrs.slot) {
        delete data.attrs.slot;
      }
      // named slots should only be respected if the vnode was rendered in the
      // same context.
      if ((child.context === context || child.fnContext === context) &&
        data && data.slot != null
      ) {
        // 如果slot存在(slot="header") 则拿对应的值作为key
        var name = data.slot;
        var slot = (slots[name] || (slots[name] = []));
        // 如果是tempalte元素 则把template的children添加进数组中，这也就是为什么你写的template标签并不会渲染成另一个标签到页面
        if (child.tag === 'template') {
          slot.push.apply(slot, child.children || []);
        } else {
          slot.push(child);
        }
      } else {
        // 如果没有就默认是default
        (slots.default || (slots.default = [])).push(child);
      }
    }
    // ignore slots that contains only whitespace
    for (var name$1 in slots) {
      if (slots[name$1].every(isWhitespace)) {
        delete slots[name$1];
      }
    }
    return slots
}
```

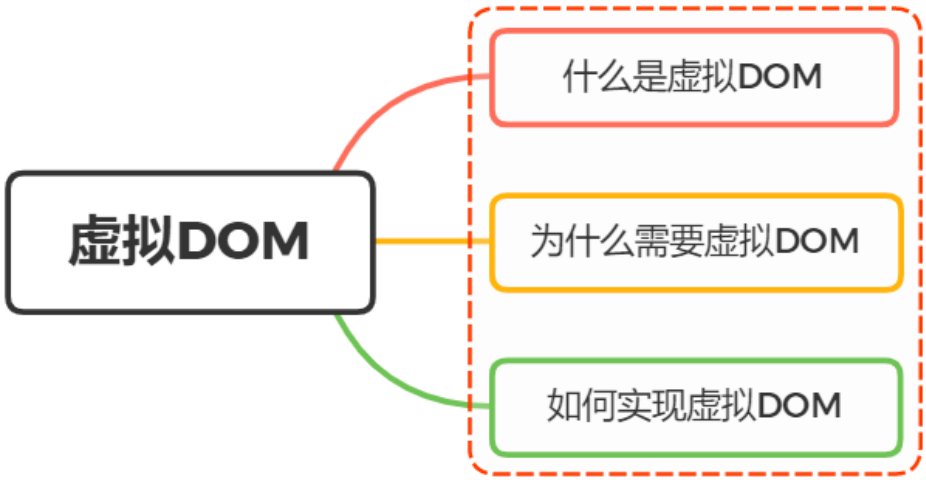
`_render` 渲染函数通过 `normalizeScopedSlots` 得到 `vm.$scopedSlots`

JavaScript | 复制代码

```
vm.$scopedSlots = normalizeScopedSlots(
  _parentVnode.data.scopedSlots,
  vm.$slots,
  vm.$scopedSlots
);
```

作用域插槽中父组件能够得到子组件的值是因为在 `renderSlot` 的时候执行会传入 `props`，也就是上述 `_t` 第三个参数，父组件则能够得到子组件传递过来的值

19. 什么是虚拟DOM？如何实现一个虚拟DOM？说说你的思路



19.1. 什么是虚拟DOM

虚拟 DOM （`Virtual DOM`）这个概念相信大家都不陌生，从 `React` 到 `Vue`，虚拟 `DOM` 为这两个框架都带来了跨平台的能力（`React-Native` 和 `Weex`）

实际上它只是一层对真实 `DOM` 的抽象，以 `JavaScript` 对象（`VNode` 节点）作为基础的树，用对象的属性来描述节点，最终可以通过一系列操作使这棵树映射到真实环境上

在 `Javascript` 对象中，虚拟 `DOM` 表现为一个 `Object` 对象。并且最少包含标签名（`tag`）、属性（`attrs`）和子元素对象（`children`）三个属性，不同框架对这三个属性的名命可能会有差别

创建虚拟 `DOM` 就是为了更好将虚拟的节点渲染到页面视图中，所以虚拟 `DOM` 对象的节点与真实 `DOM` 的属性一一照应

在 `vue` 中同样使用到了虚拟 `DOM` 技术

定义真实 `DOM`

HTML 复制代码

```html
<div id="app">
    <p class="p">节点内容</p>
    <h3>{{ foo }}</h3>
</div>
```

实例化 `vue`

JavaScript 复制代码

```javascript
const app = new Vue({
    el:"#app",
    data:{
        foo:"foo"
    }
})
```

观察 `render` 的 `render` ，我们能得到虚拟 `DOM`

JavaScript 复制代码

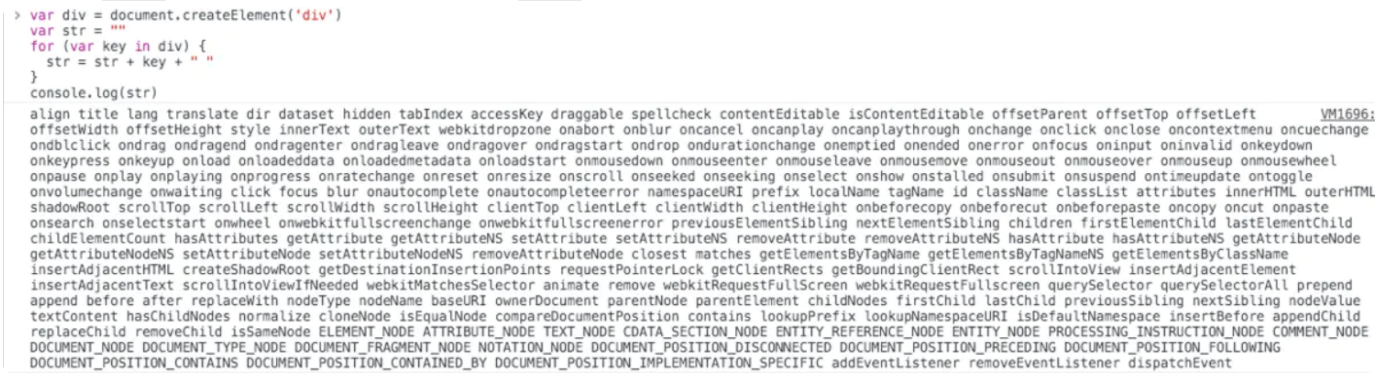
```javascript
(function anonymous(
) {
  with(this){return _c('div',{attrs:{"id":"app"}},[_c('p',{staticClass:"p"}
,
            [_v("节点内容")]),_v(" "),_c('h3',[_v(_s(foo))])])}})
```

通过 `VNode` ， `vue` 可以对这颗抽象树进行创建节点,删除节点以及修改节点的操作， 经过 `diff` 算法得出一些需要修改的最小单位,再更新视图，减少了 `dom` 操作，提高了性能

19.2. 为什么需要虚拟DOM

`DOM` 是很慢的，其元素非常庞大，页面的性能问题，大部分都是由 `DOM` 操作引起的

真实的 `DOM` 节点，哪怕一个最简单的 `div` 也包含着很多属性，可以打印出来直观感受一下：



由此可见，操作 `DOM` 的代价仍旧是昂贵的，频繁操作还是会出现页面卡顿，影响用户的体验

举个例子：

你用传统的原生 `api` 或 `jQuery` 去操作 `DOM` 时，浏览器会从构建 `DOM` 树开始从头到尾执行一遍流程

当你在一次操作时，需要更新10个 `DOM` 节点，浏览器没这么智能，收到第一个更新 `DOM` 请求后，并不知道后续还有9次更新操作，因此会马上执行流程，最终执行10次流程

而通过 `VNode` ，同样更新10个 `DOM` 节点，虚拟 `DOM` 不会立即操作 `DOM` ，而是将这10次更新的 `diff` 内容保存到本地的一个 `js` 对象中，最终将这个 `js` 对象一次性 `attach` 到 `DOM` 树上，避免大量的无谓计算

> 很多人认为虚拟 DOM 最大的优势是 diff 算法，减少 JavaScript 操作真实 DOM 的带来的性能消耗。虽然这一个虚拟 DOM 带来的一个优势，但并不是全部。虚拟 DOM 最大的优势在于抽象了原本的渲染过程，实现了跨平台的能力，而不仅仅局限于浏览器的 DOM，可以是安卓和 IOS 的原生组件，可以是近期很火热的小程序，也可以是各种GUI

19.3. 如何实现虚拟DOM

首先可以看看 `vue` 中 `VNode` 的结构

源码位置：src/core/vdom/vnode.js