象存在新增属性

9.3.3. 队列性

队列性合并有：全部生命周期和 `watch`

JavaScript | 复制代码

```
function mergeHook (
  parentVal: ?Array<Function>,
  childVal: ?Function | ?Array<Function>
): ?Array<Function> {
  return childVal
    ? parentVal
      ? parentVal.concat(childVal)
      : Array.isArray(childVal)
        ? childVal
        : [childVal]
    : parentVal
}

LIFECYCLE_HOOKS.forEach(hook => {
  strats[hook] = mergeHook
})

// watch
strats.watch = function (
  parentVal,
  childVal,
  vm,
  key
) {
  // work around Firefox's Object.prototype.watch...
  if (parentVal === nativeWatch) { parentVal = undefined; }
  if (childVal === nativeWatch) { childVal = undefined; }
  /* istanbul ignore if */
  if (!childVal) { return Object.create(parentVal || null) }
  {
    assertObjectType(key, childVal, vm);
  }
  if (!parentVal) { return childVal }
  var ret = {};
  extend(ret, parentVal);
  for (var key$1 in childVal) {
    var parent = ret[key$1];
    var child = childVal[key$1];
    if (parent && !Array.isArray(parent)) {
      parent = [parent];
    }
    ret[key$1] = parent
      ? parent.concat(child)
      : Array.isArray(child) ? child : [child];
  }
```

```
    return ret
};
```

生命周期钩子和 `watch` 被合并为一个数组，然后正序遍历一次执行

9.3.4. 叠加型

叠加型有： `component` 、 `directives` 、 `filters`

JavaScript | 复制代码

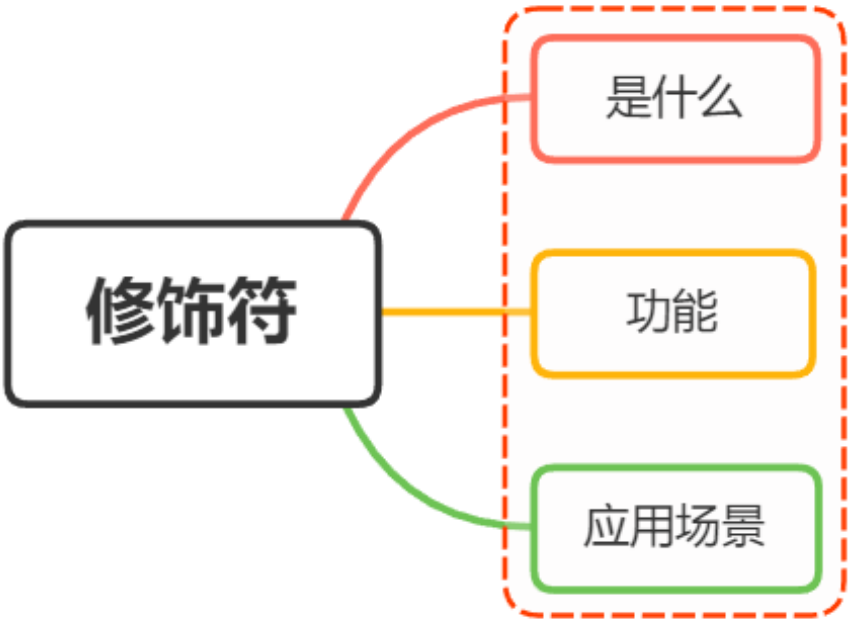
```
strats.components=
strats.directives=

strats.filters = function mergeAssets(
    parentVal, childVal, vm, key
) {
    var res = Object.create(parentVal || null);
    if (childVal) {
        for (var key in childVal) {
            res[key] = childVal[key];
        }
    }
    return res
}
```

叠加型主要是通过原型链进行层层的叠加

9.4. 小结：

- 替换型策略有 `props` 、 `methods` 、 `inject` 、 `computed` ，就是将新的同名参数替代旧的参数
- 合并型策略是 `data` ，通过 `set` 方法进行合并和重新赋值
- 队列型策略有生命周期函数和 `watch` ，原理是将函数存入一个数组，然后正序遍历依次执行
- 叠加型有 `component` 、 `directives` 、 `filters` ，通过原型链进行层层的叠加

10. Vue常用的修饰符有哪些有什么应用场景



10.1. 修饰符是什么

在程序世界里，修饰符是用于限定类型以及类型成员的声明的一种符号

在 `Vue` 中，修饰符处理了许多 `DOM` 事件的细节，让我们不再需要花大量的时间去处理这些烦恼的事情，而能有更多的精力专注于程序的逻辑处理

`vue` 中修饰符分为以下五种：

- 表单修饰符
- 事件修饰符
- 鼠标按键修饰符
- 键值修饰符
- v-bind修饰符

10.2. 修饰符的作用

10.2.1. 表单修饰符

在我们填写表单的时候用得最多的是 `input` 标签，指令用得最多的是 `v-model`

关于表单的修饰符有如下：

- lazy

- trim
- number

10.2.1.1. lazy

在我们填完信息，光标离开标签的时候，才会将值赋予给 `value` ，也就是在 `change` 事件之后再进行信息同步

JavaScript | 复制代码

```
<input type="text" v-model.lazy="value">
<p>{{value}}</p>
```

10.2.1.2. trim

自动过滤用户输入的首空格字符，而中间的空格不会过滤

JavaScript | 复制代码

```
<input type="text" v-model.trim="value">
```

10.2.1.3. number

自动将用户的输入值转为数值类型，但如果这个值无法被 `parseFloat` 解析，则会返回原来的值

JavaScript | 复制代码

```
<input v-model.number="age" type="number">
```

10.2.2. 事件修饰符

事件修饰符是对事件捕获以及目标进行了处理，有如下修饰符：

- stop
- prevent
- self
- once
- capture
- passive
- native

10.2.2.1. stop

阻止了事件冒泡，相当于调用了 `event.stopPropagation` 方法

```
JavaScript | 复制代码
<div @click="shout(2)">
  <button @click.stop="shout(1)">ok</button>
</div>
//只输出1
```

10.2.2.2. prevent

阻止了事件的默认行为，相当于调用了 `event.preventDefault` 方法

```
JavaScript | 复制代码
<form v-on:submit.prevent="onSubmit"></form>
```

10.2.2.3. self

只当在 `event.target` 是当前元素自身时触发处理函数

```
JavaScript | 复制代码
<div v-on:click.self="doThat">...</div>
```

> 使用修饰符时，顺序很重要；相应的代码会以同样的顺序产生。因此，用 `v-on:click.prevent.self` 会阻止**所有的点击**，而 `v-on:click.self.prevent` 只会阻止对元素自身的点击

10.2.2.4. once

绑定了事件以后只能触发一次，第二次就不会触发

```
JavaScript | 复制代码
<button @click.once="shout(1)">ok</button>
```

10.2.2.5. capture

使事件触发从包含这个元素的顶层开始往下触发

```JavaScript
<div @click.capture="shout(1)">
    obj1
<div @click.capture="shout(2)">
    obj2
<div @click="shout(3)">
    obj3
<div @click="shout(4)">
    obj4
</div>
</div>
</div>
</div>
// 输出结构: 1 2 4 3
```

10.2.2.6. passive

在移动端，当我们在监听元素滚动事件的时候，会一直触发 `onscroll` 事件会让我们的网页变卡，因此我们使用这个修饰符的时候，相当于给 `onscroll` 事件整了一个 `.lazy` 修饰符

```JavaScript
<!-- 滚动事件的默认行为 (即滚动行为) 将会立即触发 -->
<!-- 而不会等待 `onScroll` 完成  -->
<!-- 这其中包含 `event.preventDefault()` 的情况 -->
<div v-on:scroll.passive="onScroll">...</div>
```

> 不要把 `.passive` 和 `.prevent` 一起使用,因为 `.prevent` 将会被忽略，同时浏览器可能会向你展示一个警告。
>
> `passive` 会告诉浏览器你不想阻止事件的默认行为

10.2.2.7. native

让组件变成像 `html` 内置标签那样监听根元素的原生事件，否则组件上使用 `v-on` 只会监听自定义事件

```JavaScript
<my-component v-on:click.native="doSomething"></my-component>
```

> 使用.native修饰符来操作普通HTML标签是会令事件失效的

10.2.3. 鼠标按钮修饰符

鼠标按钮修饰符针对的就是左键、右键、中键点击，有如下：

- left 左键点击
- right 右键点击
- middle 中键点击

JavaScript | 复制代码

```
<button @click.left="shout(1)">ok</button>
<button @click.right="shout(1)">ok</button>
<button @click.middle="shout(1)">ok</button>
```

10.2.4. 键盘修饰符

键盘修饰符是用来修饰键盘事件（`onkeyup`，`onkeydown`）的，有如下：

`keyCode` 存在很多，但 `vue` 为我们提供了别名，分为以下两种：

- 普通键（enter、tab、delete、space、esc、up...）
- 系统修饰键（ctrl、alt、meta、shift...）

JavaScript | 复制代码

```
// 只有按键为keyCode的时候才触发
<input type="text" @keyup.keyCode="shout()">
```

还可以通过以下方式自定义一些全局的键盘码别名

JavaScript | 复制代码

```
Vue.config.keyCodes.f2 = 113
```

10.2.5. v-bind修饰符

v-bind修饰符主要是为属性进行操作，用来分别有如下：

- async
- prop

- camel

10.2.5.1. async

能对 `props` 进行一个双向绑定

```JavaScript
//父组件
<comp :myMessage.sync="bar"></comp>
//子组件
this.$emit('update:myMessage',params);
```

以上这种方法相当于以下的简写

```JavaScript
//父亲组件
<comp :myMessage="bar" @update:myMessage="func"></comp>
func(e){
 this.bar = e;
}
//子组件js
func2(){
  this.$emit('update:myMessage',params);
}
```

使用 `async` 需要注意以下两点：

- 使用 `sync` 的时候，子组件传递的事件名格式必须为 `update:value` ，其中 `value` 必须与子组件中 `props` 中声明的名称完全一致
- 注意带有 `.sync` 修饰符的 `v-bind` 不能和表达式一起使用
- 将 `v-bind.sync` 用在一个字面量的对象上，例如 `v-bind.sync="{ title: doc.title }"` ，是无法正常工作的

10.2.5.2. props

设置自定义标签属性，避免暴露数据，防止污染HTML结构

```JavaScript
<input id="uid" title="title1" value="1" :index.prop="index">
```

10.2.5.3. camel

将命名变为驼峰命名法，如将 `view-Box` 属性名转换为 `viewBox`

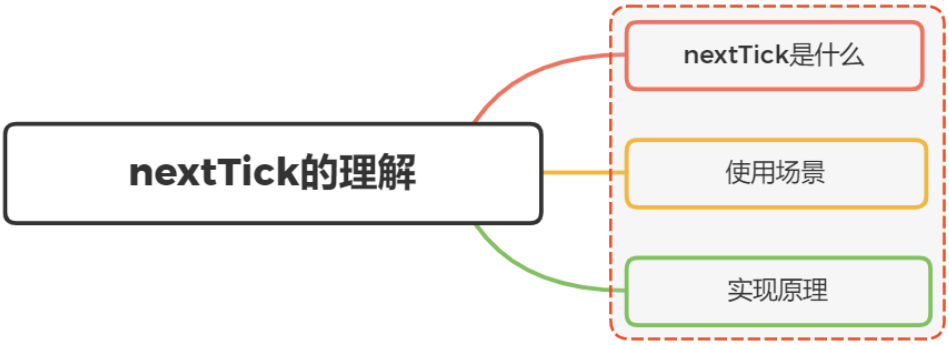
```JavaScript
<svg :viewBox="viewBox"></svg>
```

10.3. 应用场景

根据每一个修饰符的功能，我们可以得到以下修饰符的应用场景：

- .stop：阻止事件冒泡
- .native：绑定原生事件
- .once：事件只执行一次
- .self ： 将事件绑定在自身身上，相当于阻止事件冒泡
- .prevent：阻止默认事件
- .caption：用于事件捕获
- .once：只触发一次
- .keyCode：监听特定键盘按下
- .right：右键

11. Vue中的$nextTick有什么作用?



11.1. NextTick是什么

官方对其的定义

> 在下次 DOM 更新循环结束之后执行延迟回调。在修改数据之后立即使用这个方法，获取更新后的 DOM

什么意思呢?

我们可以理解成，`Vue` 在更新 `DOM` 时是异步执行的。当数据发生变化，`Vue` 将开启一个异步更新队列，视图需要等队列中所有数据变化完成之后，再统一进行更新

举例一下

`Html` 结构

HTML 复制代码

```html
<div id="app"> {{ message }} </div>
```

构建一个 `vue` 实例

JavaScript 复制代码

```javascript
const vm = new Vue({
  el: '#app',
  data: {
    message: '原始值'
  }
})
```

修改 `message`

JavaScript 复制代码

```javascript
this.message = '修改后的值1'
this.message = '修改后的值2'
this.message = '修改后的值3'
```

这时候想获取页面最新的 `DOM` 节点，却发现获取到的是旧值

JavaScript 复制代码

```javascript
console.log(vm.$el.textContent) // 原始值
```

这是因为 `message` 数据在发现变化的时候，`vue` 并不会立刻去更新 `Dom`，而是将修改数据的操作放在了一个异步操作队列中

如果我们一直修改相同数据，异步操作队列还会进行去重

等待同一事件循环中的所有数据变化完成之后，会将队列中的事件拿来进行处理，进行 `DOM` 的更新

11.1.1.1. 为什么要有nexttick

举个例子

JavaScript | 复制代码

```
{{num}}
for(let i=0; i<100000; i++){
    num = i
}
```

如果没有 `nextTick` 更新机制，那么 `num` 每次更新值都会触发视图更新(上面这段代码也就是会更新10万次视图)，有了 `nextTick` 机制，只需要更新一次，所以 `nextTick` 本质是一种优化策略

11.2. 使用场景

如果想要在修改数据后立刻得到更新后的 `DOM` 结构，可以使用 `Vue.nextTick()`

第一个参数为：回调函数（可以获取最近的 `DOM` 结构）

第二个参数为：执行函数上下文

JavaScript | 复制代码

```
// 修改数据
vm.message = '修改后的值'
// DOM 还没有更新
console.log(vm.$el.textContent) // 原始的值
Vue.nextTick(function () {
  // DOM 更新了
  console.log(vm.$el.textContent) // 修改后的值
})
```

组件内使用 `vm.$nextTick()` 实例方法只需要通过 `this.$nextTick()` ，并且回调函数中的 `this` 将自动绑定到当前的 `Vue` 实例上

```javascript
this.message = '修改后的值'
console.log(this.$el.textContent) // => '原始的值'
this.$nextTick(function () {
    console.log(this.$el.textContent) // => '修改后的值'
})
```

`$nextTick()` 会返回一个 `Promise` 对象，可以是用 `async/await` 完成相同作用的事情

```javascript
this.message = '修改后的值'
console.log(this.$el.textContent) // => '原始的值'
await this.$nextTick()
console.log(this.$el.textContent) // => '修改后的值'
```

11.3. 实现原理

源码位置：`/src/core/util/next-tick.js`

`callbacks` 也就是异步操作队列

`callbacks` 新增回调函数后又执行了 `timerFunc` 函数，`pending` 是用来标识同一个时间只能执行一次

JavaScript | 复制代码

```javascript
export function nextTick(cb?: Function, ctx?: Object) {
  let _resolve;

  // cb 回调函数会经统一处理压入 callbacks 数组
  callbacks.push(() => {
    if (cb) {
      // 给 cb 回调函数执行加上了 try-catch 错误处理
      try {
        cb.call(ctx);
      } catch (e) {
        handleError(e, ctx, 'nextTick');
      }
    } else if (_resolve) {
      _resolve(ctx);
    }
  });

  // 执行异步延迟函数 timerFunc
  if (!pending) {
    pending = true;
    timerFunc();
  }

  // 当 nextTick 没有传入函数参数的时候，返回一个 Promise 化的调用
  if (!cb && typeof Promise !== 'undefined') {
    return new Promise(resolve => {
      _resolve = resolve;
    });
  }
}
```

`timerFunc` 函数定义，这里是根据当前环境支持什么方法则确定调用哪个，分别有：

`Promise.then` 、 `MutationObserver` 、 `setImmediate` 、 `setTimeout`

通过上面任意一种方法，进行降级操作

JavaScript | 复制代码

```javascript
export let isUsingMicroTask = false
if (typeof Promise !== 'undefined' && isNative(Promise)) {
  //判断1：是否原生支持Promise
  const p = Promise.resolve()
  timerFunc = () => {
    p.then(flushCallbacks)
    if (isIOS) setTimeout(noop)
  }
  isUsingMicroTask = true
} else if (!isIE && typeof MutationObserver !== 'undefined' && (
  isNative(MutationObserver) ||
  MutationObserver.toString() === '[object MutationObserverConstructor]'
)) {
  //判断2：是否原生支持MutationObserver
  let counter = 1
  const observer = new MutationObserver(flushCallbacks)
  const textNode = document.createTextNode(String(counter))
  observer.observe(textNode, {
    characterData: true
  })
  timerFunc = () => {
    counter = (counter + 1) % 2
    textNode.data = String(counter)
  }
  isUsingMicroTask = true
} else if (typeof setImmediate !== 'undefined' && isNative(setImmediate))
{
  //判断3：是否原生支持setImmediate
  timerFunc = () => {
    setImmediate(flushCallbacks)
  }
} else {
  //判断4：上面都不行，直接用setTimeout
  timerFunc = () => {
    setTimeout(flushCallbacks, 0)
  }
}
```

无论是微任务还是宏任务，都会放到 `flushCallbacks` 使用

这里将 `callbacks` 里面的函数复制一份，同时 `callbacks` 置空

依次执行 `callbacks` 里面的函数

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

小结：

1. 把回调函数放入callbacks等待执行
2. 将执行函数放到微任务或者宏任务中
3. 事件循环到了微任务或者宏任务，执行函数依次执行callbacks中的回调

12. Vue实例挂载的过程

12.1. 思考

我们都听过知其然知其所以然这句话

那么不知道大家是否思考过 `new Vue()` 这个过程中究竟做了些什么？

过程中是如何完成数据的绑定，又是如何将数据渲染到视图的等等

12.2. 分析

首先找到 `vue` 的构造函数

源码位置：src\core\instance\index.js

JavaScript 复制代码

```
function Vue (options) {
  if (process.env.NODE_ENV !== 'production' &&
    !(this instanceof Vue)
  ) {
    warn('Vue is a constructor and should be called with the `new` keyword')
  }
  this._init(options)
}
```

`options` 是用户传递过来的配置项，如 `data、methods` 等常用的方法

`vue` 构建函数调用 `_init` 方法，但我们发现本文件中并没有此方法，但仔细可以看到文件下方定定义了很多初始化方法

JavaScript 复制代码

```
initMixin(Vue);     // 定义 _init
stateMixin(Vue);    // 定义 $set $get $delete $watch 等
eventsMixin(Vue);   // 定义事件  $on  $once $off $emit
lifecycleMixin(Vue);// 定义 _update  $forceUpdate  $destroy
renderMixin(Vue);   // 定义 _render 返回虚拟dom
```

首先可以看 `initMixin` 方法，发现该方法在 `Vue` 原型上定义了 `_init` 方法

源码位置：src\core\instance\init.js

JavaScript | 复制代码

```javascript
Vue.prototype._init = function (options?: Object) {
    const vm: Component = this
    // a uid
    vm._uid = uid++
    let startTag, endTag
    /* istanbul ignore if */
    if (process.env.NODE_ENV !== 'production' && config.performance && mark) {
      startTag = `vue-perf-start:${vm._uid}`
      endTag = `vue-perf-end:${vm._uid}`
      mark(startTag)
    }

    // a flag to avoid this being observed
    vm._isVue = true
    // merge options
    // 合并属性，判断初始化的是否是组件，这里合并主要是 mixins 或 extends 的方法
    if (options && options._isComponent) {
      // optimize internal component instantiation
      // since dynamic options merging is pretty slow, and none of the
      // internal component options needs special treatment.
      initInternalComponent(vm, options)
    } else { // 合并vue属性
      vm.$options = mergeOptions(
        resolveConstructorOptions(vm.constructor),
        options || {},
        vm
      )
    }
    /* istanbul ignore else */
    if (process.env.NODE_ENV !== 'production') {
      // 初始化proxy拦截器
      initProxy(vm)
    } else {
      vm._renderProxy = vm
    }
    // expose real self
    vm._self = vm
    // 初始化组件生命周期标志位
    initLifecycle(vm)
    // 初始化组件事件侦听
    initEvents(vm)
    // 初始化渲染方法
    initRender(vm)
    callHook(vm, 'beforeCreate')
```

```
    // 初始化依赖注入内容，在初始化data、props之前
    initInjections(vm) // resolve injections before data/props
    // 初始化props/data/method/watch/methods
    initState(vm)
    initProvide(vm) // resolve provide after data/props
    callHook(vm, 'created')

    /* istanbul ignore if */
    if (process.env.NODE_ENV !== 'production' && config.performance && mark) {
      vm._name = formatComponentName(vm, false)
      mark(endTag)
      measure(`vue ${vm._name} init`, startTag, endTag)
    }
    // 挂载元素
    if (vm.$options.el) {
      vm.$mount(vm.$options.el)
    }
  }
```

仔细阅读上面的代码，我们得到以下结论：

- 在调用 `beforeCreate` 之前，数据初始化并未完成，像 `data` 、 `props` 这些属性无法访问到
- 到了 `created` 的时候，数据已经初始化完成，能够访问 `data` 、 `props` 这些属性，但这时候并未完成 `dom` 的挂载，因此无法访问到 `dom` 元素
- 挂载方法是调用 `vm.$mount` 方法

`initState` 方法是完成 `props/data/method/watch/methods` 的初始化

源码位置：src\core\instance\state.js