vue原理

1.MVVM 2021/07/11

mvc模式

2.生命周期

beforeCreate(创建前)

created(创建后)

beforeMount

mounted

beforeUpdate

updated

beforeDestroy（销毁前）

destroyed（销毁后）

3.双向绑定

3.1响应式原理 2021/07/12 12:03

3.2 v-model处理

4.批量更新 2021/07/14 12:03

源码

has[id]

flushing

waiting

大致流程

4.1Vue中的nextTick 2021/07/15 10:43

vue原理

1.MVVM 2021/07/11

mvc模式

需要服务器端配合，JavaScript可以在前端修改服务器渲染后的数据。

但随着交互性的要求越来越高，MVC这种单向模式显然不够满足。MVVM模型就是借鉴MVC思想，在前端页面中，Model层使用纯JavaScript来表示，View负责显示，两者做到最大限度的分离。把Model层和View层关联起来的就是ViewModel，ViewModel把Model数据同步到view中，还负责把View层的修改同步Model。

优点：

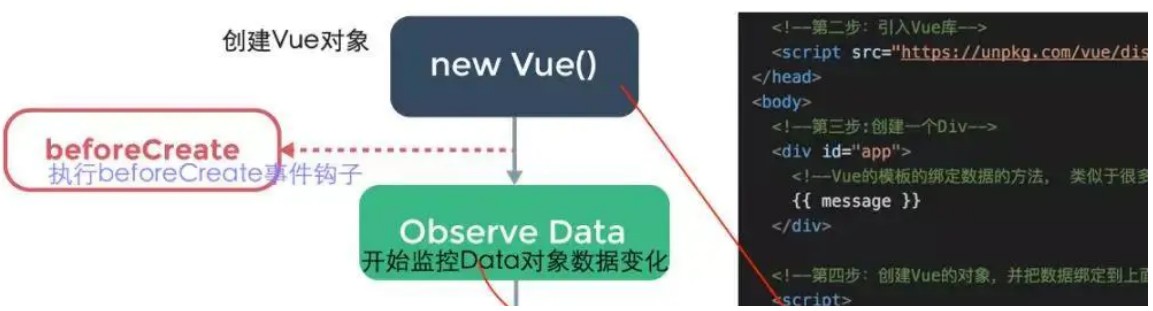
- View层变化Model层不用变化，反之
- 双向绑定，自动更新dom

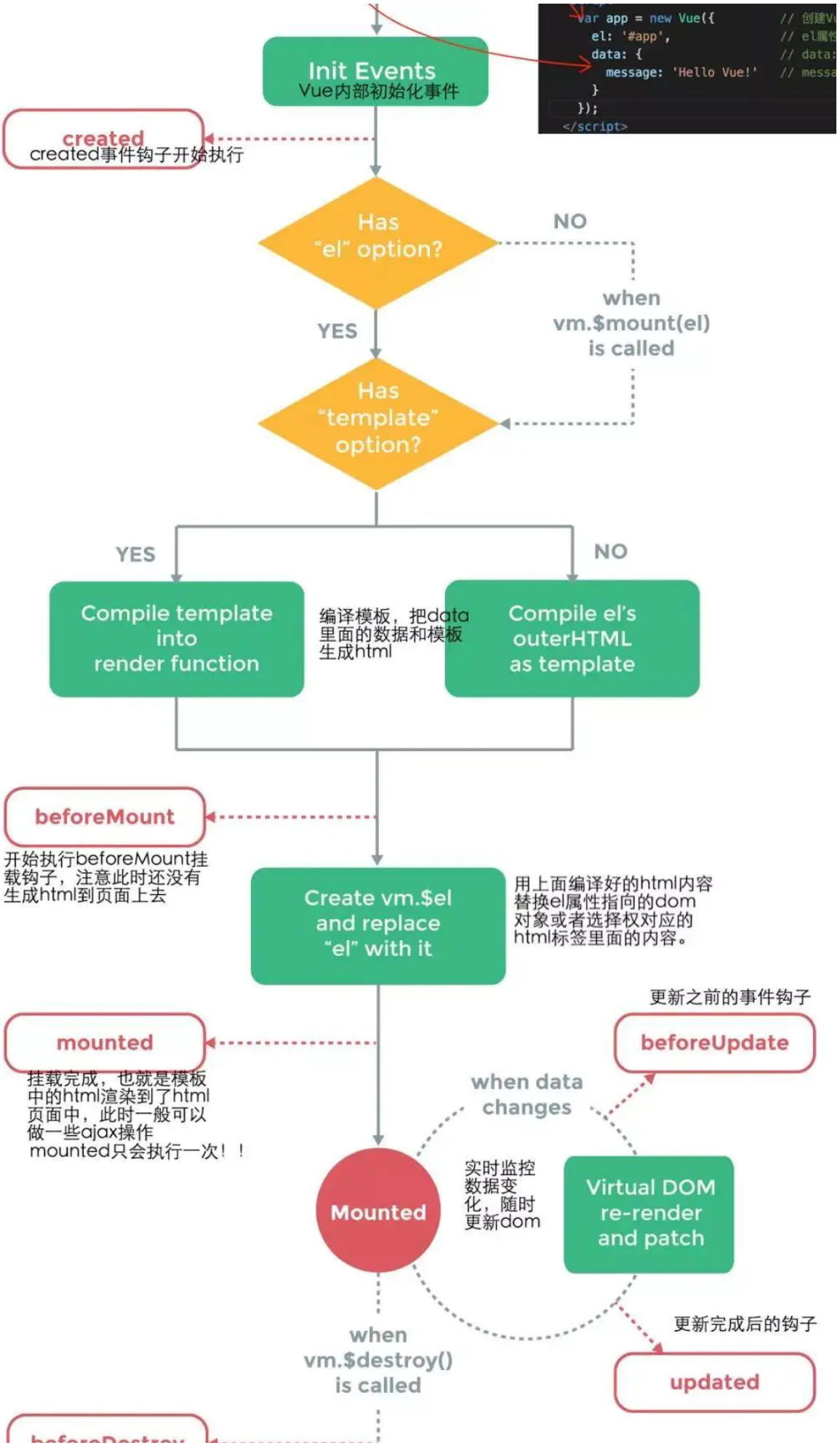
缺点

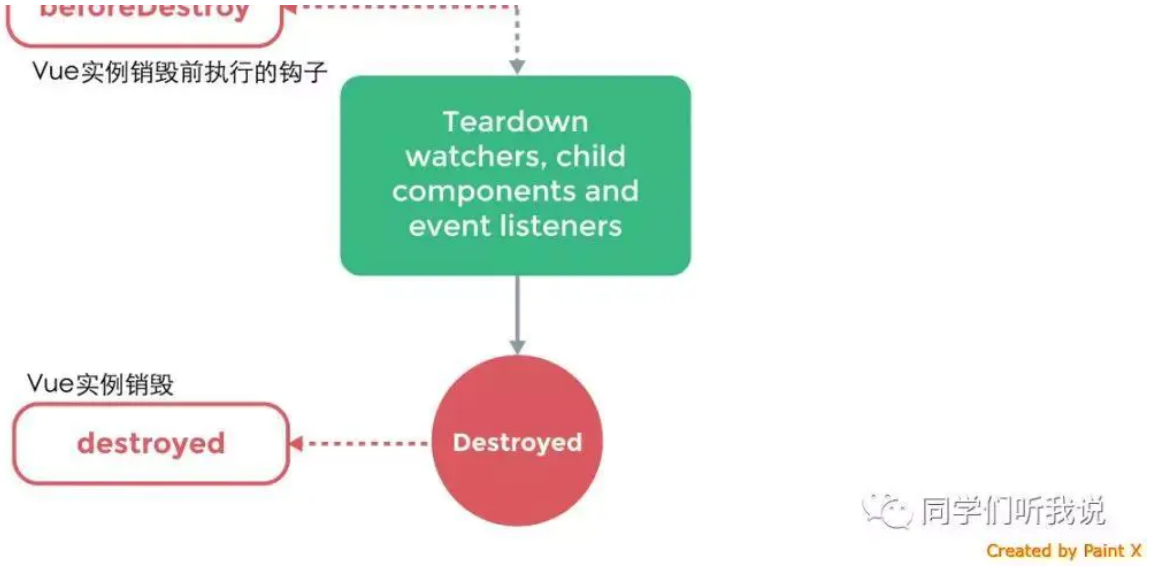
- View层出现问题后，由于View层的数据是通过指令绑定的，这些内容是没法打断点测试的。

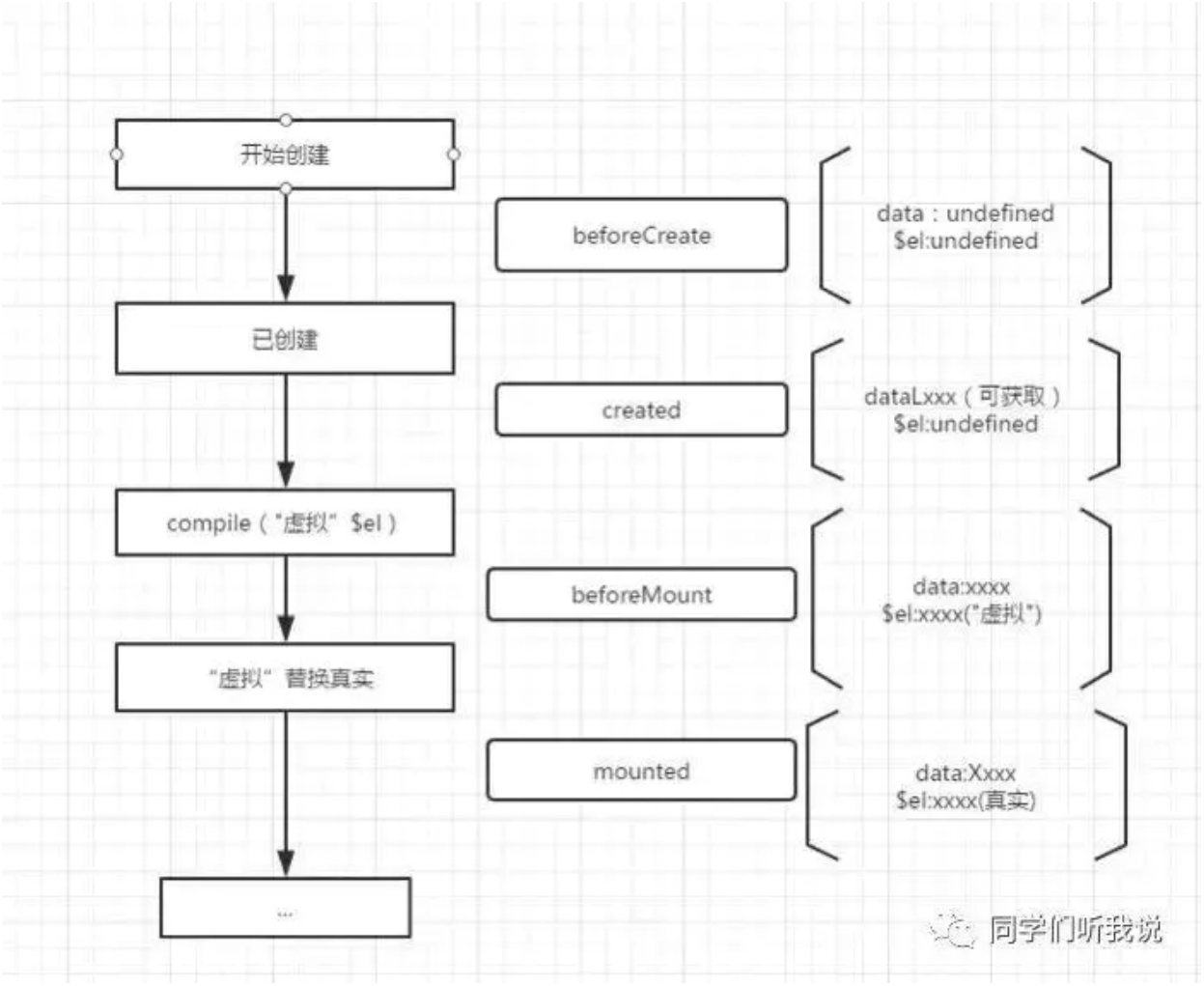
2.生命周期

Vue实例有一个完整的周期，从开始创建，初始化数据，编译模板，挂载DOM，渲染-更新-渲染，卸载等一系列过程，称为Vue实例的生命周期。
钩子就是在某个阶段给你一些处理的机会。




var app = new Vue({ // 创建V
el: '#app', // el属性
data: { // data:
message: 'Hello Vue!' // messa
}
});
</script>
Init Events
Vue内部初始化事件
created
created事件钩子开始执行
Has
"el" option?
NO
when
vm.$mount(el)
is called
YES
Has
"template"
option?
YES
NO
Compile template
into
render function
编译模板，把data
里面的数据和模板
生成html
Compile el's
outerHTML
as template
beforeMount
开始执行beforeMount挂
载钩子，注意此时还没有
生成html到页面上去
Create vm.$el
and replace
"el" with it
用上面编译好的html内容
替换el属性指向的dom
对象或者选择权对应的
html标签里面的内容。
更新之前的事件钩子
mounted
beforeUpdate
挂载完成，也就是模板
中的html渲染到了html
页面中，此时一般可以
做一些ajax操作
mounted只会执行一次！！
when data
changes
Mounted
实时监控
数据变
化，随时
更新dom
Virtual DOM
re-render
and patch
更新完成后的钩子
when
vm.$destroy()
is called
updated





beforeCreate(创建前)

在实例初始化后，数据观测和事件配置之前被调用，但此时组件的选项对象还未被创建，el和data也未初始化，因此无法访问data，methods，computed里的方法和数据。

created(创建后)

实例创建之后被调用，这一步，实例已完成以下配置：数据观测、属性和方法的运算，watch/event事件回调，完成了data数据的初始化。此时el还没有初始化，挂载阶段也未开始，$el属性也不可见，这是一个常用的生命周期，可以调用methods中的方法，改变data中的数据，通过Vue的响应式绑定体现在页面中。这个周期中是没有方法对实例化过程进行拦截的，因此假如有某些数据必须获取才能进入页面的话，并不适合在这个方法发请求，建议在组件路由钩子beforeRouteEnter中完成。

beforeMount

挂载开始之前被调用，相关的render函数首次被调用（虚拟DOM），实例已经完成如下配置：编译模板，把data与模板内容生成html，完成了el与data的初始化，注意此时还没有将生成的html挂载到页面上。

mounted

挂载完成，将生成的html挂载到页面上，此时可以做ajax请求，mounted只执行一次。

beforeUpdate

数据更新之前被调用，发生在虚拟DOM重新渲染和打补丁之前，可以在钩子中进一步的更改状态，不会触发重渲染过程。

updated

组件DOM已更新，此时可以执行依赖DOM的操作。避免在此更改状态，否则会引起重新从而导致无限循环。

beforeDestroy（销毁前）

- 这一步还可以用this获取到实例。
- 一般在这步做些重置操作，比如清除组件中的定时器和监听的dom事件。

destroyed（销毁后）

调用后，所有事件监听器被移出，所有子实例也会被销毁。

3.双向绑定

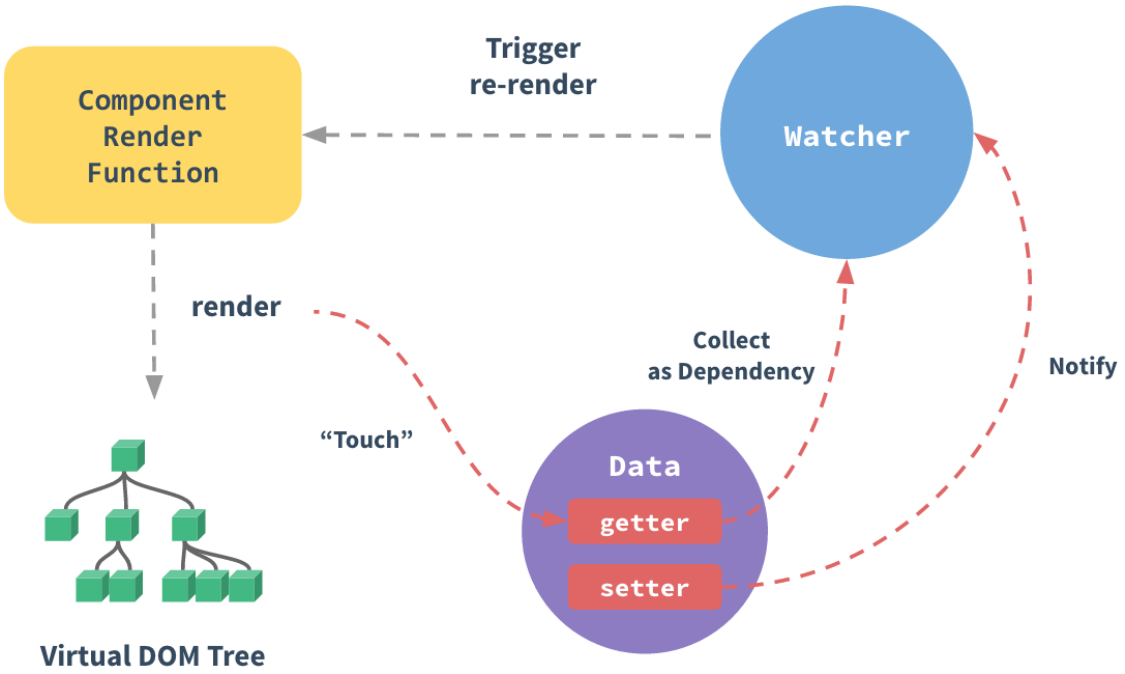
3.1响应式原理 2021/07/12 12:03

当把一个Javascript对象传入Vue实例作为data选项，Vue将遍历此对象的所有成员，并使用Object.defineProperty把这些property全部转为getter/setter。

这些getter/setter是不可见的，但是在内部它们让Vue能够追踪依赖，在property被访问和修改通知变更。

依赖收集

每个组件实例对应一个watcher，它会在组件渲染时把接触过的property记录为依赖，之后当依赖的setter触发时，会通知watcher，从而使它关联的组件重新渲染。



对于对象

Vue无法检测property的添加或移除。由于Vue对象会在初始化实例时对property转换，所以property必须存在于data对象上才能让Vue将它转换为响应式的。

```
//使用Vue实例
Vue.set
//使用Vue别名
this.$set
```

有时需要为已有对象赋值多个property，比如`Object.assign`，但是这样添加到对象上的新property不会触发更新。在这种情况下，应该用原对象与要混合进去的对象的property一起创建个新的对象。

```
this.someObj = Object.assign({},this.someobj,{a:1,b:2})
```

对于数组

Vue.set(this.arr,index,newValue)

this.arr.splice(indexOfItem,1,newValue)

异步更新队列

Vue在更新Dom时是异步的。只要侦听到数据变化，Vue将开启一个队列，并缓存在同一事件循环中发生的所有变更。如果同一个watcher被多次触发，只会被推入到队列中一次。这种在缓存时去除重复数据对于比必要的计算和DOM操作是很重要的。然后，在下一轮事件循环tick中，Vue刷新队列(已去重)工作。Vue在内部对异步队列尝试使用原生的Promise.then,MutationObserver和setImmediate,如果环境不支持，则使用setTimout()

有时为了数据变化之后等待Vue完成更新DOM，这样回调函数就会在DOM更新完成后调用。

Vue.nextTick(()=>{})

computed 2021/07/13 10:05

设计它们的初衷就是如果在模板中放入太多的逻辑会让模板难以维护。
{{message.split('').reverse().join()}}

所以对于任何复杂逻辑，则应当使用computed。

```
computed:{
   reverseMsg:function(){
      return this.message.split('').reverse().join()
   }
}
```

这里我们提供的函数用作vm.reverseMsg的getter函数。当vm.message发生变化时，reverseMsg也会跟着变化。

计算属性是基于它们的响应式依赖进行缓存的。只在相关依赖发生变化时它们才会重新求值。同样意味着下面的值不再更新，因为它不是响应式依赖。

```
computed:{
   now:function(){
```

```
        return Date.now()
    }
}
```

我们为什么需要缓存?假设有个性能开销大的计算属性A，它要遍历一个巨大的数组并做大量的计算，然后还有其他的计算属性依赖于A。如果没有缓存，将不可避免的多次执行A的getter!

计算属性是不能在默认的getter函数中进行赋值操作，例如：

```
computed:{
    foo:function(){
        this.num = 1
        return Date.now()
    }
}
```

解决方式就是接下来介绍的**计算属性的**setter

计算实现默认只有setter，不过在需要是也可以提供一个setter：

```
computed:{
    foo:{
        get:function(){
            return this.name
        },
        set:function(name){
            this.name = name
        }
    }
}
```

运行vm.foo = 'Tom'， setter函数就会被调用。

watch

当需要数据变化时需要执行异步或其他开销比较大的操作时，这个方法是最有用的。

```
<template>
    <input v-model="question">
    {{answer}}
</template>

<script>
new Vue({
    data(){
```

```
        return {
            answer:'ask a question',
            question:''
        }
    },
    watch:{
        question:function(){
            this.getAnswer()
        }
    },
    methods:{
        getAnswer(){
            this.question = 'Thinking'
            var vm = this
            axios.get('...').then(()=>{
                vm.answer = '...'
            }).catch(()=>{...})
        }
    }
})
</script>
```

watch选项允许我们执行异步操作（访问一个API），并在得到最终结果前，设置中间状态。这些都是计算属性无法做到的。

- 选项deep

为了发现对象内部值的变化，可以在选项参数中指定deep：true，注意监听数组变更是不需要这么做。

- 选项：immediate

在选项参数中指定immediate:true，将立即将表达式的当前值触发回调。

3.2 v-model处理

3.2.1表单元素级别

负责监听用户输入事件来更新数据。

v-model会忽略所有表单元素的value、checked、selected属性，初始值总是将当前活动实例的数据作为数据来源。应该通过JavaScript在组件中的data选项中声明初始值。

v-model在内部为不同的表单元素使用不同的property并抛出不同的事件：

- text和textarea元素使用其value属性和input事件。
- checkbox和radio使用checked属性和change事件。
- select 将value作为prop并将change作为事件。

多个复选框绑定到同一数组：

```
<input type="checkbox" id="jack" value="Jack" v-model="checkedNames">
<label for="jack">Jack</label>
<input type="checkbox" id="john" value="John" v-model="checkedNames">
<label for="john">John</label>
```

值绑定：

对于单选按钮，复选框，下拉框，v-model通常绑定的值是静态字符串(复选框可以是布尔值)

修饰符：

- **.lazy**

v-model每次在input事件触发后将输入框的值与绑定的数据进行同步。可以给v-model添加lazy修饰符，从而转为在输入框失去焦点时进行同步。

- **.number**

将输入值转为数值类型，可以给v-model添加number修饰符。因为即使在type='number'时，输入元素值也总返回字符串。

- **.trim**

过滤用户输入的首尾空白字符，可以给v-model添加trim修饰符。

3.2.2组件级别（手动实现）

```
<input v-model="searchText"/>
```

等价于：

```
<input :value="searchText @input="searchText=$event.target.value"/>
```

当用在组件时，v-model使用方式为这样：

```
<custom-input :inputValue="searchText" @updata:inputValue=$event/>
```

custom-input组件的实现：

- 通过props属性接受参数inputValue。
- 在其input事件被触发时，将新值通过自定义的updata:inputValue事件抛出。

写成代码是这样的：

```
Vue.component('custom-input',{
    props:['inputValue'],
    template:`
        <input :value="inputValue"
@input="$emit('updata:inputValue',$event.target.value)"/>
    `
})
```

现在custom-input组件就可以完美的运行起来了。

另一种方法是使用computed property的功能来定义getter和setter。get方法应返回inputValue，set则触发相应的事件。

4.批量更新 2021/07/14 12:03

连续3次触发了mutation，那么watch中的cb会执行几次呢？答案是1次。

接下来围绕上面这个例子介绍的就是Vue中的批量更新：

首先，msg这个key是通过object.defineproperty监听了的，当key被set时，触发了这个key对应的watcher事件中的update方法。

那么watcher是什么呢？

watcher就类似于一个key的观察者，当key被set时，会调用自身原型上的update方法。这个方法最终目的是调用实例上的cb。即 console.log('监听到mutation')。

问题的关键就是监听到一个set引起的mutation就立即同步执行一次cb吗？

那么如果cb是一个性能开销很大的回调函数，那么就会引起性能问题了。

所以update方法一定是异步的：

Vue的做法是把cb放在微任务队列或者宏任务队列，具体放在哪个队列，要看当前运行环境是否支持Promise、MutationObserver,setImmediate这几个相当于微任务的队列，不支持就用settimeout调用cb放在宏任务中。

不管放在宏任务中还是微任务中，cb总是等待所以同步代码执行完成后才去执行，这里涉及到EventLoop。就是先调用宏任务，每个宏任务有个延时队列，比如settimeout的回调函数就会被放在宏任务的延时队列中，延时队列也是异步操作。然后等待同步代码执行完成后，就去看当前宏任务的微任务队列，依次执行，当前微任务队列如果都出队之后就去执行延时队列中的回调函数。

源码

```
function queueWatcher (watcher) {
  var id = watcher.id;
  if (has[id] == null) {
    has[id] = true;
    if (!flushing) {
      queue.push(watcher);
    } else {
      // if already flushing, splice the watcher based on its id
      // if already past its id, it will be run next immediately.
      var i = queue.length - 1;
      while (i > index && queue[i].id > watcher.id) {
        i--;
      }
      queue.splice(i + 1, 0, watcher);
    }
    // queue the flush
    if (!waiting) {
      waiting = true;
      nextTick(flushSchedulerQueue);
    }
  }
}
```

has[id]

我们执行了3次set操作，对应的三次key中的watcher中的update方法，而这三次每次传的watcher实例都是同一个watcher实例，所以第一次has[id]为null，第一次置为true之后，后面的两次就就直接结束了。通俗解释就是，第一次触发mutation，就已经加入到队列中，后面就会处理。同一个watcher的mutation就不再接待了。

flushing

为了让quene队列保证watcher按顺序排列，例如下面通俗的例子：

- 第一次watcher3进入if分支，被推入到队列最后面然后执行对应的回调。此时变量flushing变为true，表示当前watcher3执行完毕，可以让新的watcher1进入。队列顺序为[watcher3]
- watcher1进入的是else分支，queue中所有的watcher是按照ID升序排列的，此时队列顺序为[watcher1,watcher3]，执行watcher1对应的回调

waiting

保证完成一次对queue的遍历之前不会开启新的遍历。

大致流程

相同的watcher只会被推入queue中一次，然后一次加入把flushSchedulerQueue加入task队列，一次遍历queue调用watcher的cb，即一次输出“监听到mutation”。

4.1Vue中的nextTick 2021/07/15 10:43

```
/* @flow */
    /* globals MutationObserver */
    import { noop } from 'shared/util'
    import { handleError } from './error'
    import { isIE, isIOS, isNative } from './env'
    export let isUsingMicroTask = false
    const callbacks = []
    let pending = false
    function flushCallbacks () {
      pending = false
      const copies = callbacks.slice(0)
      callbacks.length = 0
      for (let i = 0; i < copies.length; i++) {
        copies[i]()
      }
    }
    // Here we have async deferring wrappers using microtasks.
```

```
    // In 2.5 we used (macro) tasks (in combination with microtasks).
    // However, it has subtle problems when state is changed right before repaint
    // (e.g. #6813, out-in transitions).
    // Also, using (macro) tasks in event handler would cause some weird
behaviors
    // that cannot be circumvented (e.g. #7109, #7153, #7546, #7834, #8109).
    // So we now use microtasks everywhere, again.
    // A major drawback of this tradeoff is that there are some scenarios
    // where microtasks have too high a priority and fire in between supposedly
    // sequential events (e.g. #4521, #6690, which have workarounds)
    // or even between bubbling of the same event (#6566).
    let timerFunc
    // The nextTick behavior leverages the microtask queue, which can be accessed
    // via either native Promise.then or MutationObserver.
    // MutationObserver has wider support, however it is seriously bugged in
    // UIWebView in iOS >= 9.3.3 when triggered in touch event handlers. It
    // completely stops working after triggering a few times... so, if native
    // Promise is available, we will use it:
    /* istanbul ignore next, $flow-disable-line */
    if (typeof Promise !== 'undefined' && isNative(Promise)) {
      const p = Promise.resolve()
      timerFunc = () => {
        p.then(flushCallbacks)
        // In problematic UIWebViews, Promise.then doesn't completely break, but
        // it can get stuck in a weird state where callbacks are pushed into the
        // microtask queue but the queue isn't being flushed, until the browser
        // needs to do some other work, e.g. handle a timer. Therefore we can
        // "force" the microtask queue to be flushed by adding an empty timer.
         if (isIOS) setTimeout(noop)
      }
      isUsingMicroTask = true
    } else if (!isIE && typeof MutationObserver !== 'undefined' && (
      isNative(MutationObserver) ||
      // PhantomJS and iOS 7.x
      MutationObserver.toString() === '[object MutationObserverConstructor]'
    )) {
      // Use MutationObserver where native Promise is not available,
      // e.g. PhantomJS, iOS7, Android 4.4
      // (#6466 MutationObserver is unreliable in IE11)
      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)) {
  // Fallback to setImmediate.
  // Technically it leverages the (macro) task queue,
  // but it is still a better choice than setTimeout.
  timerFunc = () => {
    setImmediate(flushCallbacks)
  }
} else {
  // Fallback to setTimeout.
  timerFunc = () => {
    setTimeout(flushCallbacks, 0)
  }
}
export function nextTick (cb?: Function, ctx?: Object) {
  let _resolve
  callbacks.push(() => {
    if (cb) {
      try {
        cb.call(ctx)
      } catch (e) {
        handleError(e, ctx, 'nextTick')
      }
    } else if (_resolve) {
      _resolve(ctx)
    }
  })
  if (!pending) {
    pending = true
    timerFunc()
  }
  // $flow-disable-line
  if (!cb && typeof Promise !== 'undefined') {
    return new Promise(resolve => {
      _resolve = resolve
    })
  }
}
```

核心之处是timeFunc的实现，有以下重要的几点：

- 使用了Promise（内部含有异步代码，resolve/then..），MutationObserver微任务的异步延迟包装器。
    - 首先判断是否支持Promise，支持的话利用Promise来执行回调函数。

- 如果不支持Promise，再判断是否支持Mutationobserver。如果执行，那么生成一个Observer来观察文本节点发生的变化，从而执行所有的回调函数。
- 如果Promise、Mutationobserver都不支持，那么使用settimeout

- flushCallbacks
  - 会被放入微任务或者宏任务中将要执行的回调函数的集合。

5.性能优化

5.1利用cache缓存

Vue官网解释：

```
computed: {
  timeMessage: {
    cache: false,
    get: function () {
      return Date.now() + this.message
    }
  }
}
```

在Vue未来的大版本中，计算属性的cache属性将会被移除，把不缓存的计算属性转换为函数可以得到相同的结果：

```
methods:{
    getTimeMessage: function () {
      return Date.now() + this.message
    }
}
```

手写代码理解：

```
<template>
  {{ballType()}}
  {{BookType()}}
</template>
<script>
data(){
  return{
    a:'足球',
    b:'三国',
```

```
  }
 },
mounted(){

},
methods:{
  ballType(){
    return this.myCach2('球类')(this.a)
  },
  BookType(){
    return this.myCach2('书类')(this.b)
  },
  myCach2(type){
    return (val)=>{
      return `type is ${type},val is ${val}`
    }
  }
</script>
```

cache缓存函数使用的闭包形式，比如this.myCach2这个函数性能开销很大，那么我们就可以使用返回一个**闭包函数**，这时就将这个闭包函数给缓存起来了，当this.a变化时也就是再次读取时，直接读取this.myCach2保存的数据，就不用在重新执行this.myCach2。