技术分享—丁时一（Vue2.x原理系列）

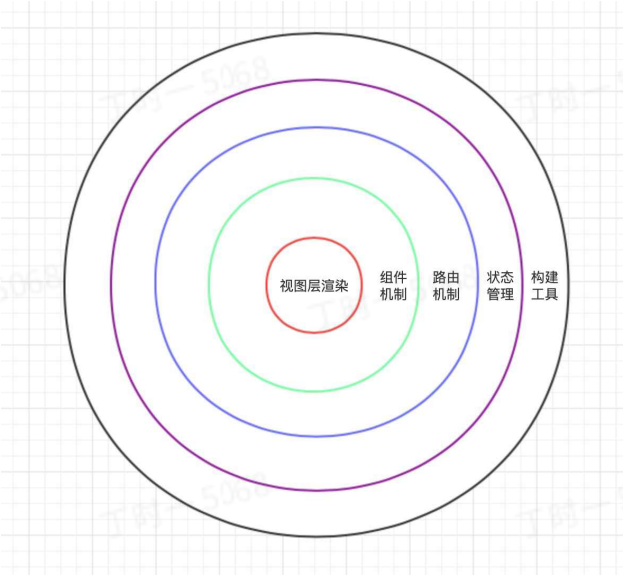
> 理解原理可以让我们写代码的时候更有信心，并且能够以不同的形式去写代码，在调试的时候也可以帮我们快速定位。本文档主要是理解一些渲染的大概流程和一些api的实现。

简介

Vue.js(之后简称Vue)是一款友好的渐进式的javascript框架。能够帮助我们创建可维护性和可测试性更强的代码。对比之前类似于Jquery库等，随着应用程序变得复杂过后，我们需要更加频繁的去操作DOM，由于缺乏正规的组织形式，我们的代码变得非常难以维护。但Vue就能够很好的帮我们解决这个问题：通过描述状态和DOM之间的映射关系、就可以将状态渲染成DOM呈现在用户界面中。

何为渐进式

关于渐进式，我也是最近才了解到的。**所谓渐进式，就是把框架分层**。简单来说，就是一开始不需要你完全掌握它的全部功能特性，可以后续逐步增加功能。没有多做职责之外的事情。



虚拟DOM

Vue1.0

在Vue1.0的时候，还没有虚拟DOM，这时候使用的方案是每个节点都绑定一个watcher，当状态发生变化的时候，就会在一定程度上知道哪些节点使用了这个状态，从而对这些节点进行更新操作，根本不需要去进行比对。

但这样做有一定的代价，因为颗粒度太细了，每一个绑定都有一个对应的watcher来观察状态的变化，这样就会造成一些内存的开销和一些依赖追踪的开销，当状态被越多的节点使用时候，开销就越大。对于一个大型项目来说，这个开销就是非常大的。

Vue1.0响应式实现demo

Vue1.0响应式实现

Vue2.x后

在vue2.0引入虚拟Dom之后，就选择了一个中等颗粒度的方案：组件级别是一个wacher实例，就是说即便一个组件内部有10个节点使用了某个状态，但也只有一个watcher来观察这个状态的变化，所以当状态发生变化的时候，只能通知到组件，然后组件内部通过虚拟DOM去对比(diff)与渲染(patch)。

什么是虚拟DOM?

虚拟DOM（virtual dom），就是使用javascript对象来描述一个dom（Vnode），毕竟我们直接去操作dom的话，代价是非常大的，所以我们就使用虚拟DOM先把各种属性、变化等都描述好，然后再根据这个对象去渲染真实的DOM。举个例子:

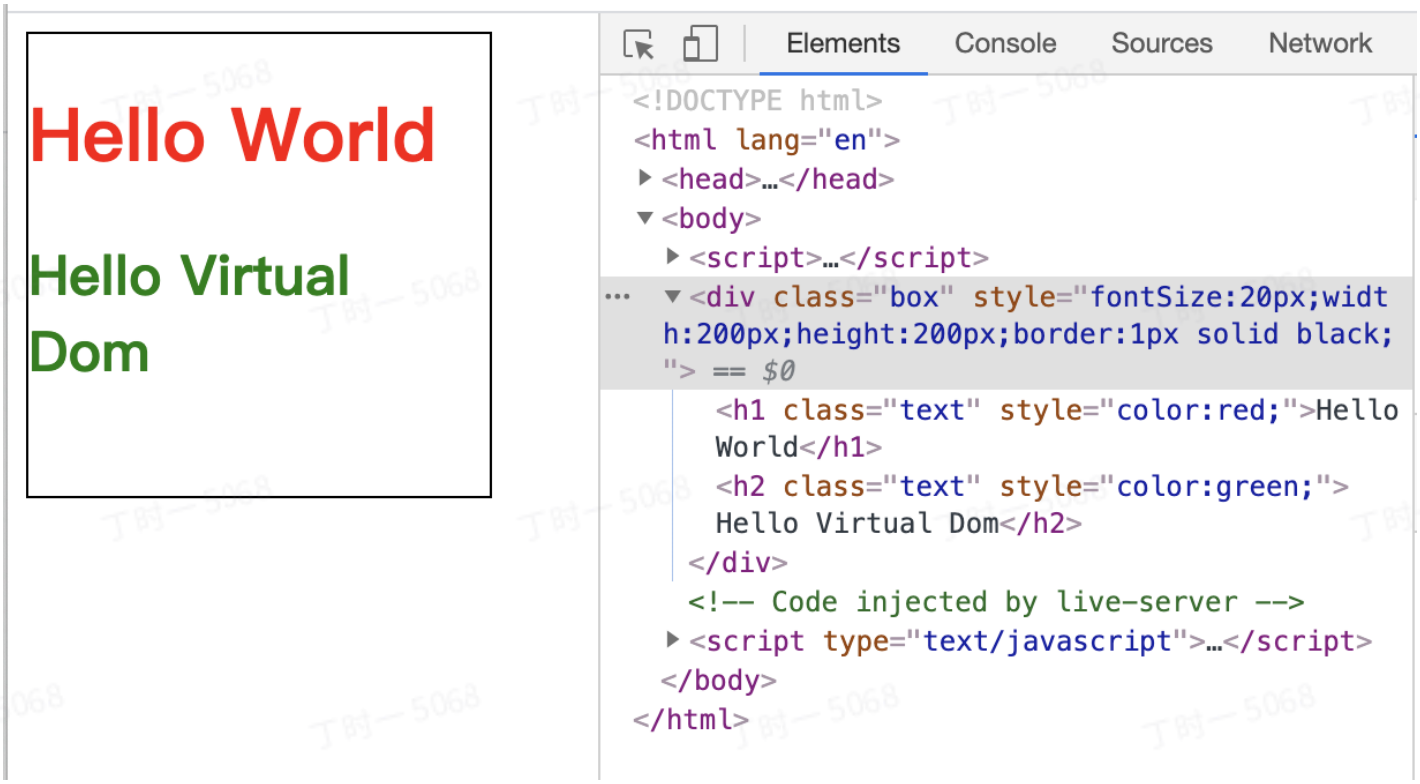
JavaScript

```
const Element = {
    tag: 'div',
    props: {
        class: 'box',
        style: {
            fontSize: "20px",
            width: "200px",
            height: "200px",
            border: "1px solid black"
        }
    },
    text: "",
    children: [{
            tag: 'h1',
            props: {
                class: 'text',
                style: {
                    color: 'red'
                }
```

```
            },
            text: "",
            children: [{
                text: "Hello World",
            }]
        },
        {
            tag: 'h2',
            props: {
                class: 'text',
                style: {
                    color: 'green'
                }
            },
            text: '',
            children: [{
                text: "Hello Virtual Dom",
            }]
        }
    ]
}
console.log(Element);

//创建节点
function createElement(element) {
    const {
        tag,
        children,
        text
    } = element;
    //判断是否是标签节点
    if (typeof tag === 'string') {
        element.el = document.createElement(tag);
        children.map((child) => {
            element.el.appendChild(createElement(child))
        })
    } else {
        console.log('文本节点');
        element.el = document.createTextNode(text)
    }
    updateProps(element)
    return element.el
}

//根据props添加attribute
```

```
function updateProps(element) {
    const el = element.el;
    const newProps = element.props || {}
    for (let key in newProps) {
        if (key === "style") {
            let styleStr = "";
            for (let sKey in newProps.style) {
                styleStr += `${sKey}:${newProps.style[sKey]};`
            }
            el.setAttribute('style', styleStr)
        } else if (key === 'class') {
            el.className = newProps[key];
        } else {
            el.setAttribute(key, newProps[key])
        }
    }
}
document.body.appendChild(createElement(Element))
```

运行后的效果就是这样的:



template -> vNode

这里可能有一些疑问，我们编写的时候写的是标签，上面代码中的`Element`是怎么来的呢？这就要说到Vue的模版解析了，这里不多讲（因为自己也不是特别清楚，就说说大概：

获取模版

Vue会根据传入的options，然后判断获取template。由下面的代码可知，优先级render > template > el,

当获取到template之后，调用compileToFunctions并且传入template获取render函数，这个过程中就涉及到template->ast->render函数的转换了。

JavaScript

```
`/src/platforms/web/entry-runtime-with-compiler.js`
Vue.prototype.$mount = function (
  el?: string | Element,
  hydrating?: boolean
): Component {
  el = el && query(el)
  //省略。。。。
  const options = this.$options
  // resolve template/el and convert to render function
  if (!options.render) {
    let template = options.template
    if (template) {
      if (typeof template === 'string') {
        if (template.charAt(0) === '#') {
          template = idToTemplate(template)
        }
      } else if (template.nodeType) {
        template = template.innerHTML
      } else
      //省略
      }
    } else if (el) {
      template = getOuterHTML(el)
    }
    if (template) {
      //根据template获取render函数。
      const { render, staticRenderFns } = compileToFunctions(template, {
      //省略
      }, this)
      options.render = render
     }
     return mount.call(this, el, hydrating)
```

```
}


function getOuterHTML (el: Element): string {
  if (el.outerHTML) {
    return el.outerHTML
  } else {
    const container = document.createElement('div')
    container.appendChild(el.cloneNode(true))
    return container.innerHTML
  }
}
```

模版编译(template->AST)

这一步就是根据获取的template根据许多正则表达式，把获取的template解析成ast树,源码比较复杂，下面用一个Demo进行阐述。

HTML

```
<!--源码路径src/compiler/parser/html-parser.js-->
<!--简易Demo-->
<body>
    <script type="text/template" id="ast">
        <div>
            <h1>你好</h1>
            <ul>
                <li>你好啊A</li>
                <li>好的B</li>
                <li>收到了C</li>
            </ul>
        </div>
    </script>
    <script src="./parse.js"></script>
    <script>
        var str = document.getElementById('ast').innerHTML
        console.log(str)
        parse(str)
    </script>
</body>
```

JavaScript

```
//parse.js
function parse(templateStr) {
    // 当前遍历到的位置
    var index = 0
    // 栈1保存标签
    var stack1 = []
    // 栈2保存结果
    var stack2 = [{
        children: []
    }]
    // 剩余的模板字符串（即没遍历到的）
    var lastStr = templateStr
    // 匹配开始标签
    var startTagReg = /^<([a-z]+[1-6]?)>/
    // 匹配结束标签
    var endTagReg = /^<\/([a-z]+[1-6]?)>/
    // 这里只匹配开始标签到结束标签之间的文字，不匹配结束标签到开始标签之间的文字
    var wordTagReg = /^([^\>]+)<\/([a-z]+[1-6]?)>/
    // var wordTagReg = /^>([^\>]+)<\/([a-z]+[1-6]?)>/

    while (index < templateStr.length) {
        if (startTagReg.test(lastStr)) {
            var startTag = lastStr.match(startTagReg)[1]
            // console.log('找到开始标签', startTag)
            // console.log(index)
            index += startTag.length + 2
            stack1.push({
                tag: startTag,
                children: [],
                isRoot: true
            })
            stack2.push({
                tag: startTag,
                children: []
            })
        } else if (endTagReg.test(lastStr)) {
            var endTag = lastStr.match(endTagReg)[1]
            // console.log('找到结束标签', endTag)
            var c = stack2.pop()
            stack2[stack2.length - 1].children.push(c)
            index += endTag.length + 3
        } else if (wordTagReg.test(lastStr)) {
            var wordTag = lastStr.match(wordTagReg)[1]
            index += wordTag.length
            if (!/^\s+$/.test(wordTag)) {
```

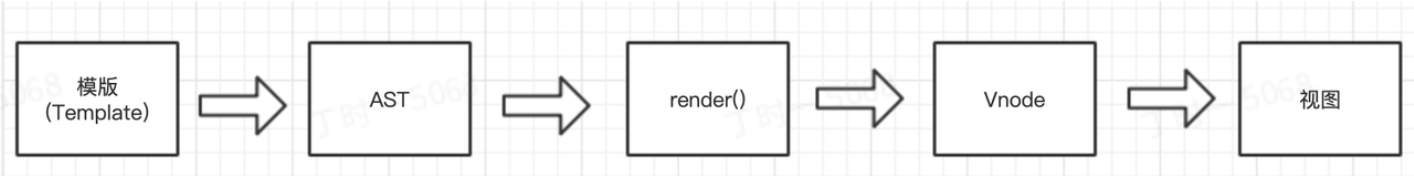
```
                stack2[stack2.length - 1].children.push({
                    text: wordTag
                })
            } else {
                // console.log('全是空字符串')
            }
        } else {
            // 啥都没找到
            // console.log('没匹配到标签以及开始到结束标签之间的文字', lastStr)
            index++
        }
        //截取没有解析的剩下的字符串
        lastStr = templateStr.substring(index)
    }
    console.log(stack1)
    console.log(stack2)
```

这个DEMO并没有template->ast->render->Vnode的过程。

真实的过程应该是根据template解析出ast 、然后通过with的方式生成一个作用域。这个function就是render函数，然后render函数执行的时候，就会根据原型上的_c、_v、_s等方法去生成对应的Vnode。

JavaScript

```
 //ast
    _c("p", {
        staticClass: "foo"
    }, [
        _v(_s(text))
    ])

 //render
function render() {
    with(this) {
        return _c('div', {
            attrs: {
                "data": 111
            }
        },
        [_v(111)])
    }
}

//Vue.prototype._c = ....
//Vue.prototype._v = ....
//执行render  通过_v,_c,_s等生成vnode也就是最开始的Element了。
```



当状态发生变化的时候，我们完全可以重新再重复上面的过程，但是这样太费时间了，我们可以在状态变化的时候，将新的和旧的虚拟DOM做一个比较（diff），只去更新不同的地方。

Vue2.x响应式原理

Object.defineProperty简介

从ES5开始,所有的对象都具备了属性描述符.

JavaScript

```
let person = {
   name:"丁时一"
  };

Object.defineProperty(person, "name", {
  configurable: true,
  writable: true,
  enumerable: true,
  value: "丁时一",
});
```

如你所见,这个普通的对象属性对应的属性描述符,可不仅仅只是一个"丁时一",它还包含另外三个特性,`configurable`,`writable`,`enumerable`。

writable

> 定义属性是否可写，在非严格模式下,修改会静默失败,但在严格模式下,就会抛出TypeError错误!

JavaScript

```
"use strict";
let person = {};

Object.defineProperty(person, "name", {
  configurable: true,
  writable: false, //不可写!
  enumerable: true,
  value: "丁时一",
});

console.log(person.name); //丁时一
person.name = "Hello World"; //在严格模式下 Cannot assign to read only property "name" of object "#<Object>"
console.log(person.name); //丁时一
```

enumerable

> 控制属性是否会出现在对象的枚举当中,比如说for...in循环，如果enumerable为false那么这个属性就不会出现在枚举当中

JavaScript

```
let person = {};
    Object.defineProperty(person, "name", {
      configurable: true,
      writable: true,
      enumerable: true, //可枚举
      value: "丁时一",
    });
    Object.defineProperty(person, "age", {
      configurable: true,
      writable: true,
      enumerable: false, //不可枚举
      value: 20,
    });
    console.log(person); //{name: "丁时一", age: 20}
    for (let i in person) {
      //并没有age  因为age不可枚举
      console.log(i, person[i]); //name 丁时一
    }
```

configurable

> configurable表示属性是否可配置。只要属性是可配置的,就可以使用Object.defineProperty来修改属性描述符。

JavaScript

```
let myobj={
  a:3
}
myobj.a=4
console.log(myobj.a);//4
Object.defineProperty(myobj,"a",{
  configurable:false,//不可配置!
  writable:true,
  value:4

myobj.a=5
console.log(myobj.a);//5

Object.defineProperty(myobj, "a", {
  configurable: true,
  writable: true,
});//TypeError
```

不管是否处于严格模式,尝试修改一个不可配置的属性描述符都会发生错误,所以把configurable设置为false是单向操作!

一个属性被定义为不可配置之后，再次调用Object.defineProperty()并修改除writable以外的属性都会导致错误

tips:在configurable为false的情况下,还是可以把writable从true改为false,但是由false改为true就会导致错误!

get、set(主要)

> 每个属性都可以使用get、set来拦截属性的读取和设置

JavaScript

```
const obj = {
    name: '丁时一'
}
let val = obj.name
Object.defineProperty(obj, 'name', {
    configurable: true,
    enumerable: true,
    get() {
        console.log("读取属性name");
        return val
    },
    set(newVal) {
        console.log('设置属性name');
        val= newVal
    }
})
obj.name='233'
console.log(obj.name);
```

收集依赖

知道了Object.defineProperty有set和get操作的时候，我们就可以利用这个做点事情了。当我们读取一个属性的值的时候，我们就把要和这个属性相关的操作(更新等)放到一个框里面，当我们修改这个属性的时候，我们就去执行这个框里面的操作。

来看一个简单的例子:

JavaScript

```javascript
const data = {
    message: "hello world",
    height: "170",
    weight: "115"
}
Object.keys(data).forEach(key => {
    defineReactive(data, key, data[key])
})
let Target = null;

function $watch(exp, fn) {
    // 将 Target 的值设置为 fn
    Target = fn
    // 读取字段值，触发 get 函数
    data[exp]
}
function defineReactive(data, key, val) {
    //这个dep就是拿来装与对应属性相关的操作的
    const dep = [];
    Object.defineProperty(data, key, {
        get() {
            dep.push(Target)
            return val
        },
        set(newVal) {
            if (newVal === val) {
                return
            }
            val = newVal
        }
    })
}
//第一个依赖
$watch('name', () => {
    console.log("name 被修改了，我可以执行一些和name相关的东西1");
})
//第二个依赖
$watch('name', () => {
    console.log("name 被修改了，我可以执行一些和name相关的东西2");
})
```

在这里我们对一组数据进行循环监测，当我们去获取属性值的时候，就把依赖收集起来。

触发依赖

当我们改变属性值的时候，我们就去执行我们收集的依赖。

JavaScript

```
        set(newVal) {
            if (newVal === val) {
                return
            }
            //触发依赖
            dep.forEach(fn => fn())
            val = newVal
        }

 //对值进行修改，
data.message="丁时一"
```

运行后结果如下图所示:

可以看到我们收集到的依赖在我们的属性发生变化的时候就执行了。

那么设想一下:如果我们的节点使用到了这个属性，那么我们就把更新对应节点的依赖给收集起来，然后属性改变的时候，我们就去执行这些更新，是不是就可以达到响应式的效果了？

源码导读

确实Vue是这么做的，现在我们来跟着源码从new Vue()入手简单过一遍收集依赖和触发依赖的过程:

options.data的传递过程

JavaScript

```
1.
`/src/core/instance/index.js`
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)//执行_init方法
  }
```

JavaScript

```
2.
//现在我们找到_init定义的地方
`/src/core/instance/init.js`
export function initMixin (Vue: Class<Component>) {
  Vue.prototype._init = function (options?: Object) {
    const vm: Component = this
    //省略 一些选项合并策略。。。
    initLifecycle(vm)
    initEvents(vm)
    initRender(vm)
    callHook(vm, 'beforeCreate')
    initInjections(vm) // resolve injections before data/props
    initState(vm)
    initProvide(vm) // resolve provide after data/props
    callHook(vm, 'created')
  }
  //省略。。。
}
//可以看到，在_init当中做了很多初始化的操作，其中就包括initState(vm)
```

JavaScript

```
3.
//进入到initState();
`/src/core/instance/state.js`
export function initState (vm: Component) {
  vm._watchers = []
  const opts = vm.$options
  if (opts.props) initProps(vm, opts.props)
  if (opts.methods) initMethods(vm, opts.methods)
  if (opts.data) {
    initData(vm)
  } else {
    observe(vm._data = {}, true /* asRootData */)
  }
  if (opts.computed) initComputed(vm, opts.computed)
  if (opts.watch && opts.watch !== nativeWatch) {
    initWatch(vm, opts.watch)
  }
}
//在initState中也做了许多选项的初始化，包括props，methods，computed，watch。这里我们直接看initData()
```

JavaScript

```
4.
`/src/core/instance/state.js`
function initData (vm: Component) {
  let data = vm.$options.data
  //这里判断data 是 函数韩式对象，如果是函数的话，那么得到函数的执行结果，也就是return 的那个对象
  data = vm._data = typeof data === 'function'
    ? getData(data, vm)
    : data || {}

  // 省略....
  const keys = Object.keys(data)s
  let i = keys.length
  while (i--) {
  //省略。。。
    const key = keys[i]
  //对data做一层代理 就是this.message 访问的就是this.data.message
    proxy(vm, `_data`, key)
  }
  // observe data
  observe(data, true /* asRootData */)
}
```

这个方法里我省略了很多，这个方法里面主要就是把传进来的data 使用Observer进行响应式处理

JavaScript

```
5.
`src/core/instance/state.js`
export function observe (value: any, asRootData: ?boolean): Observer | void {
  let ob: Observer | void
  ob = new Observer(value)
  return ob
}
```

JavaScript

```
6.
`/src/core/observer/index.js`
export class Observer {
  value: any;
  dep: Dep;
  vmCount: number; // number of vms that have this object as root $data

  constructor (value: any) {
    this.value = value
    this.dep = new Dep()
    this.vmCount = 0
    def(value, '__ob__', this)
    if (Array.isArray(value)) {
    //对数组的处理   先省略。。。
    } else {
    //先不考虑data是数组的情况下，那么我们就去执行walk方法了
      this.walk(value)
    }
  }
  walk (obj: Object) {//这个obj就是data
    const keys = Object.keys(obj)
    for (let i = 0; i < keys.length; i++) {
    //这里面对data进行一个遍历，然后把obj 和 对应的键传入defineReactive
      defineReactive(obj, keys[i])
    }
  }
  /**
   * 对数组的处理
   */
  observeArray (items: Array<any>) {
  }
}
```

加入响应式系统defineReactive

还记得我们自己去实现收集依赖的过程吗，那个defineReactive和这里的是一个作用，现在我们来看看这个函数到底做了什么。`defineReactive` 函数的核心就是 **将数据对象的数据属性转换为访问器属性，**即为数据对象的属性设置一对 `getter/setter`，但其中做了很多处理边界条件的工作。`defineReactive` 接收五个参数，但是在 `walk` 方法中调用 `defineReactive` 函数时只传递

了前两个参数，即数据对象和属性的键名。我们看一下 `defineReactive` 的函数体，首先定义了 `dep` 常量，它是一个 `Dep` 实例对象：

JavaScript

```
`/src/core/observer/index.js`
export function defineReactive (
  obj: Object,
  key: string,
  val: any,
  customSetter?: ?Function,
  shallow?: boolean
) {
//每个属性对应一个 框 用来装收集的依赖
  const dep = new Dep()

  const getter = property && property.get
  const setter = property && property.set

  let childOb = !shallow && observe(val)
  Object.defineProperty(obj, key, {
    enumerable: true,
    configurable: true,
    get: function reactiveGetter () {
      const value = getter ? getter.call(obj) : val
      if (Dep.target) {
      //对依赖的收集  相当于push
        dep.depend()
        if (childOb) {
          childOb.dep.depend()
          if (Array.isArray(value)) {
            dependArray(value)
          }
        }
      }
      return value
    },
    set: function reactiveSetter (newVal) {
      const value = getter ? getter.call(obj) : val
      //如果前后数据一样就没必要再触发依赖了
      if (newVal === value || (newVal !== newVal && value !== value)) {
        return
      }
      //如果之前有set的话，那么在这里执行一次
```

```
      if (setter) {
        setter.call(obj, newVal)
      } else {
        val = newVal
      }

      childOb = !shallow && observe(newVal)
      //notify就是去循环执行收集的依赖，然后进行一个update->render->patch的过程
      dep.notify()
    }
  })
}
```

问题：现在我们什么时候会去触发这个get呢？(什么时候收集的依赖)

想必大家都了解过Watcher,这个就可以看成我们的依赖了。当我们对一个对象，组件，渲染函数进行观察的时候，就需要使用Watcher构造函数了，这里简述一下触发收集依赖的过程:

JavaScript

```
export default class Watcher {
  vm: Component;
  expression: string;
  cb: Function;
  //省略
  constructor (
    vm: Component,
    expOrFn: string | Function,
    cb: Function
  ) {
    this.vm = vm
    if (isRenderWatcher) {
      vm._watcher = this
    }
    vm._watchers.push(this)
    // parse expression for getter
    if (typeof expOrFn === 'function') {
      this.getter = expOrFn
    } else {
      this.getter = parsePath(ex    pOrFn)
    }
    this.value = this.lazy
      ? undefined
      : this.get()
```

```
  }

  /**
   * Evaluate the getter, and re-collect dependencies.
   */
  get () {
    pushTarget(this)
    let value
    const vm = this.vm
    try {
      value = this.getter.call(vm, vm)
    } catch (e) {
      if (this.user) {
        handleError(e, vm, `getter for watcher "${this.expression}"`)
      } else {
        throw e
      }
    }
    //省略
    return value
  }
   addDep (dep: Dep) {
    const id = dep.id
    if (!this.newDepIds.has(id)) {
      this.newDepIds.add(id)
      this.newDeps.push(dep)
      if (!this.depIds.has(id)) {
        dep.addSub(this)
      }
    }
  }

}
```

问题：触发依赖的过程

> 当属性发生变化的时候，就会触发属性的set，set里面执行了一个`dep.notify`,这个刚发就是对dep里面收集的依赖（watcher）进行遍历，然后调用其update方法进行相应的更新操作。

JavaScript

```
 dep.notify() --- >

 class Dep{
   notify () {
    for (let i = 0, l = subs.length; i < l; i++) {
      //subs[i]就是一个个的watcher
      subs[i].update()
    }
  }
}
class Watcher{
  update () {
    /* istanbul ignore else */
    if (this.lazy) {
      this.dirty = true
    } else if (this.sync) {
    // 一些cb的触发
      this.run()
    } else {
      //nextTick有关?
      queueWatcher(this)
    }
  }
}
```

数组的处理

在上面说响应式数据的处理的时候，关于数组的处理还没有进行一个阐述。下面这段代码是Observer构造函数里面的，当我们需要观测的数据是一个数组的时候，我们就会去做一些对数组的响应式的处理。

JavaScript

```
if (Array.isArray(value)) {
  const augment = hasProto
    ? protoAugment
    : copyAugment
  augment(value, arrayMethods, arrayKeys)
  this.observeArray(value)
} else {
  this.walk(value)
}
```

拦截数组变异方法的思路

现在我们有这么一种思路，把原本的数组的方法保存下来，再去重写这个方法，在这个重写的方法里面执行之前保存的方法，然后就可以在这个重写的方法里面去做一些其他的事情了。

JavaScript

```
function sayHello(){
    console.log("hello")
}
const originSayHello  = sayHello;
sayHello = function(){
    console.log("做一些其他的事情");
    //执行原本的方法
    originSayHello();
}
```

这样去做的话，既能保证原来的效果，又能在途中去做一些其他的事情。Vue在对数组进行处理的时候，是把数组的一些方法给保存下来，然后重写，重写的时候做一些响应式的处理，我们来看看Vue怎么去做的吧：

JavaScript

```
const arrayProto = Array.prototype
export const arrayMethods = Object.create(arrayProto)

const methodsToPatch = [
  'push',
  'pop',
  'shift',
  'unshift',
  'splice',
  'sort',
  'reverse'
]

methodsToPatch.forEach(function (method) {
  // 缓存之前的方法
  const original = arrayProto[method]
  def(arrayMethods, method, function mutator (...args) {
    const result = original.apply(this, args)
    const ob = this.__ob__
    let inserted
    switch (method) {
      case 'push':
      case 'unshift':
        inserted = args
        break
      case 'splice':
        inserted = args.slice(2)
        break
    }
    if (inserted) ob.observeArray(inserted)
    // notify change
    ob.dep.notify()
    return result
  })
})
```

Vue先缓存了7种数组的方法，并且把这些方法加到array数据,__proto__上面去，所以当使用数组的这些方法的时候，实际上是在执行对应的mutator在这些方法里面，会对不同的执行进行不同的处理，

如果有插入新的数据，那么就需要把这些新的数据加到响应式系统当中

JavaScript

```
if (inserted) ob.observeArray(inserted)
```

当数据发生改变，那么就需要触发对应的依赖。

JavaScript

```
ob.dep.notify()
```

因为数组里面可能包含的还是一些对象，所以在对数组进行观察的时候，需要再对数组进行一个遍历观察，就和普通对象一样的了。

JavaScript

```
observeArray (items: Array<any>) {
    for (let i = 0, l = items.length; i < l; i++) {
    //对数组项的观察和普通的对象的观察是一样的。
      observe(items[i])
    }
  }
```

为什么使用变异数组，而不直接使用Object.defineProperty去做监控呢?

其实Object.defineProperty是可以监听数组的变化的,比如对每个下标进行一个观察，那么使用下标去获取指和设置值的时候就会触发一些get和set:

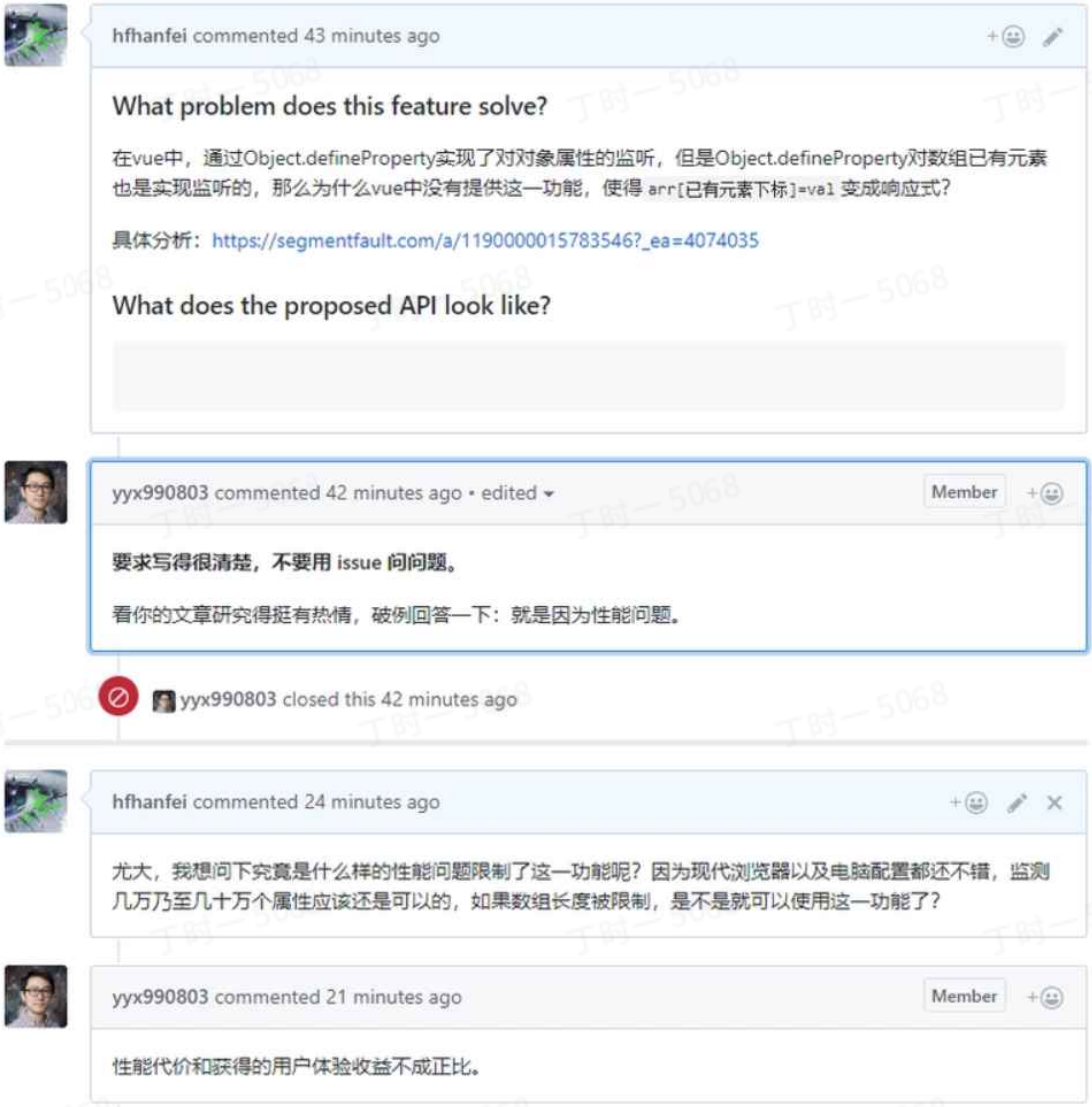
JavaScript

```
let Target = null;//暂存依赖
const array = [1, 2, 3, 4, 5];
for (let i in array) {
    defineReactive(array, i, array[i]);
}
function $watchArray(exp, fn) {
    Target = fn;
    //取值触发get
    array[exp];
}

function defineReactive(data, key, val) {
    //这个dep就是拿来装与对应属性相关的操作的
    const dep = [];
    Object.defineProperty(data, key, {
        get() {
            dep.push(Target)
            return val
        },
        set(newVal) {
            if (newVal === val) {
                return
            }
            dep.forEach(fn => fn())
            val = newVal
        }
    })
}

$watchArray('0', () => {
    console.log('数组的第一项改变');
})
$watchArray('1', () => {
    console.log('数组的第二项改变');
})

array[0] = 2222;//触发set
array[1] = 1111;//触发set
```

数组的第一项改变

数组的第二项改变

那为什么不这么去做呢？而是使用变异数组的方式，因为：



没了。

一些API的实现

$set实现

用法

vm.$set(target, propertyName/index, value)

{Object | Array} target

{string | number} propertyName/index

{any} value

向响应式对象中添加一个 property，并确保这个新 property 同样是响应式的，且触发视图更新。它必须用于向响应式对象上添加新 property，因为 Vue 无法探测普通的新增 property

JavaScript

```
1    Vue.prototype.$set = set
```

我们来看看 `set` 方法把

> core/observer/index.js

JavaScript

```
export function set (target: Array&lt;any&gt; | Object, key: any, val: any): any
{
  //如果是数组 则使用splice进行    因为splice本身就是响应式的
  if (Array.isArray(target) &amp;&amp; isValidArrayIndex(key)) {
    target.length = Math.max(target.length, key)
    target.splice(key, 1, val)
    return val
  }
  //如果时set原的属性 直接赋值就好了 会触发响应式
  if (key in target &amp;&amp; !(key in Object.prototype)) {
    target[key] = val
    return val
  }
  //到了这里就说明不是
  const ob = (target: any).__ob__
  if (target._isVue || (ob &amp;&amp; ob.vmCount)) {
    return val
  }
  if (!ob) {
    target[key] = val
    return val
  }
  //并且对这个属性进行响应式处理
  defineReactive(ob.value, key, val)
  ob.dep.notify()
  return val
}
```

我们先全部的看一看这些代码，之后再进行拆分讲解。

进入set方法，会有一个if判断，判断target是否是==数组==，如果是数组，并且index也是合法的，那么我们就使用 `splice` 方法对齐进行更新就好了，因为Vue2.x在处理数组的时候，使用的变异方法里面包含splice，那么我们直接使用splice对齐进行更新，也能触发响应式。

JavaScript

```
//如果是数组 则使用splice进行    因为splice本身就是响应式的
if (Array.isArray(target) &amp;&amp; isValidArrayIndex(key)) {
  target.length = Math.max(target.length, key)
  target.splice(key, 1, val)
  return val
}
```

那如果target不是数组，我们就进入下一个判断：如果这个key本来就是target上的属性，我们就直接修改这个属性就好了，会自动触发响应式。

JavaScript

```
//如果时set原的属性 直接赋值就好了 会触发响应式
if (key in target &amp;&amp; !(key in Object.prototype)) {
  target[key] = val
  return val
}
```

那如果这个key不是target上的属性，那么就是要新增属性了:在新增属性之前，我们取到target.__ob__，

JavaScript

```
const ob = (target: any).__ob__
if (target._isVue || (ob &amp;&amp; ob.vmCount)) {
  return val
}
```

那这个`__ob__`又是什么，我们来到`Observer`类中，发现target.ob就是observer实例，

JavaScript

```
export class Observer {
  constructor (value: any) {
    this.value = value
    this.dep = new Dep()
    this.vmCount = 0
    //value  的 __ob__就是observer实例 __ob__上有dep
    def(value, "__ob__", this)
   }
}
```

然后是下面一段代码,判断是否是根data和Vue实例，这两个上是不可以加数据的。

接下来就判断target `是否是响应式的`：如果target不是响应式的，也就是ob不存在，那么新增的属性也没必要做响应式，直接加上去就好了

JavaScript

```
  if (!ob) {
    target[key] = val
    return val
  }
```

那如果target是响应式的呢？正好有一个方法可以将属性做成响应式的，那就是`defineReactive`，我们直接调用

`defineReactive`把属性添加到target上，然后再调用ob.dep.notify()触发响应式就ok了。

JavaScript

```
  defineReactive(ob.value, key, val)
  ob.dep.notify()
```

$delete实现

用法

vm.$delete(target, propertyName/index)

{Object | Array} target

{string | number} propertyName/index

删除对象的 property。如果对象是响应式的，确保删除能触发更新视图。这个方法主要用于避开 Vue 不能检测到 property 被删除的限制，但是你应该很少会使用它

JavaScript

```
Vue.prototype.$delete= del
```

老规矩,我们先来整体看看`delete`吧。

JavaScript

```
export function del (target: Array&lt;any&gt; | Object, key: any) {
  if (Array.isArray(target) &amp;&amp; isValidArrayIndex(key)) {
    target.splice(key, 1)
    return
  }
  const ob = (target: any).__ob__
  if (target._isVue || (ob &amp;&amp; ob.vmCount)) {
    return
  }
  if (!hasOwn(target, key)) {
    return
  }
  delete target[key]
  if (!ob) {
    return
  }
  ob.dep.notify()
}
```

相比于`set`，`delete`的步骤要少了许多

首先进入del函数也是一个对target的判断：如果target是数组，并且index合法的话，同样的手段，使用`splice`进行删除。

JavaScript

```
  if (Array.isArray(target) &amp;&amp; isValidArrayIndex(key)) {
    target.splice(key, 1)
    return
  }
```

如果target不是数组，首先也是需要获取__ob__,关于__ob__在set的时候已经讲述过了，接着判断是否是根data或者Vue实例。如果是的话，直接返回，不做任何操作

```JavaScript
const ob = (target: any).__ob__
if (target._isVue || (ob &amp;amp;&amp;amp; ob.vmCount)) {
  return
}
```

接下来判断key属性是否是target上的属性，如果不是，那么就没有删除的必要了，不做任何操作，直接退出。

```JavaScript
if (!hasOwn(target, key)) {
  return
}
```

那么这些条件都不满足，那么我们就需要删除对象上的key属性了,使用`delete`关键字，将属性删除。本来应该进行触发响应式的，但如果target本来就不是响应式数据，那么就没必要进行响应式的触发，所以接着就是一个if判断，如果ob不存在（不是响应式数据），那么删除key属性之后，直接退出。否则执行`ob.dep.notify`触发响应式。

```JavaScript
delete target[key]
 if (!ob) {
   return
 }
 ob.dep.notify()
```

computed实现

Vue的computed选项的实现

watch实现

Vue选项的watch原理

对比Vue3.0（响应式实现方面）

Vue2.x响应式缺点

- 遍历递归，消耗大 ---- 递归观察属性 observer(val)
- 新增/删除属性，需要额外实现单独的API $delete $set
- 数组，需要额外实现 变异数组
- ……

Proxy简介

ECMAScript 6 入门 - Proxy

下面是 Proxy 支持的拦截操作一览，一共 13 种。

- **get(target, propKey, receiver)**：拦截对象属性的读取，比如`proxy.foo`和`proxy['foo']`。
- **set(target, propKey, value, receiver)**：拦截对象属性的设置，比如`proxy.foo = v`或`proxy['foo'] = v`，返回一个布尔值。
- **has(target, propKey)**：拦截`propKey in proxy`的操作，返回一个布尔值。
- **deleteProperty(target, propKey)**：拦截`delete proxy[propKey]`的操作，返回一个布尔值。
- **ownKeys(target)**：拦截`Object.getOwnPropertyNames(proxy)`、`Object.getOwnPropertySymbols(proxy)`、`Object.keys(proxy)`、`for...in`循环，返回一个数组。该方法返回目标对象所有自身的属性的属性名，而`Object.keys()`的返回结果仅包括目标对象自身的可遍历属性。
- **getOwnPropertyDescriptor(target, propKey)**：拦截`Object.getOwnPropertyDescriptor(proxy, propKey)`，返回属性的描述对象。
- **defineProperty(target, propKey, propDesc)**：拦截`Object.defineProperty(proxy, propKey, propDesc)`、`Object.defineProperties(proxy, propDescs)`，返回一个布尔值。
- **preventExtensions(target)**：拦截`Object.preventExtensions(proxy)`，返回一个布尔值。
- **getPrototypeOf(target)**：拦截`Object.getPrototypeOf(proxy)`，返回一个对象。
- **isExtensible(target)**：拦截`Object.isExtensible(proxy)`，返回一个布尔值。
- **setPrototypeOf(target, proto)**：拦截`Object.setPrototypeOf(proxy, proto)`，返回一个布尔值。如果目标对象是函数，那么还有两种额外操作可以拦截。
- **apply(target, object, args)**：拦截 Proxy 实例作为函数调用的操作，比如`proxy(...args)`、`proxy.call(object, ...args)`、`proxy.apply(...)`。
- **construct(target, args)**：拦截 Proxy 实例作为构造函数调用的操作，比如`new proxy(...args)`。

使用ES6的 Proxy 进行数据响应化，解决上述Vue2痛点

Proxy可以在目标对象上加一层拦截/代理，外界对目标对象的操作，都会经过这层拦截

相比 `Object.defineProperty` ，Proxy支持的对象操作十分全面：get、set、has、deleteProperty、ownKeys、defineProperty……等等

Vue3.0分享

Vue3.0分享

Proxy响应式简易demo

JavaScript

```
class Dep {
    constructor(val) {
        //收集依赖
        this.effects = new Set();
        //ref
        this._val = val
    }

    get value() {
        this.depend()
        return this._val
    }
    set value(newVal) {
        this._val = newVal
        dep.notice()
    }

    depend() {
        //收集依赖
        if (Dep.target) {
            this.effects.add(Dep.target)
        }
    }

    notice() {
        //触发依赖
        this.effects.forEach(effect => effect())
    }
}

// const dep = new Dep(10);

function effect(fn) {
    Dep.target = fn;
```

```
    fn()
    Dep.target = null;
}


const targetMap = new Map();

function getDep(target, key) {
    let deps = targetMap.get(target);
    if (!deps) {
        deps = new Map();
        targetMap.set(target, deps)
    }
    let dep = deps.get(key);
    if (!dep) {
        dep = new Dep();
        deps.set(key, dep)
    }
    return dep
}


function reactive(data) {
    return new Proxy(data, {
        get(target, key) {
            // console.log(`${key}获取`);
            //获取这个target
            const proxyTarget = Reflect.get(target, key);
            if (typeof proxyTarget === 'object') {
                return reactive(target[key])
            }
            const dep = getDep(target, key)
            dep.depend()
            return Reflect.get(target, key)
        },
        set(target, key, val) {
            // console.log(`${key}设置`);
            const dep = getDep(target, key)
            const result = Reflect.set(target, key, val)
            dep.notice()
            return result
        }
    })
}
```

```
const data = {
    message: 'hello world',
    age: 21,
    info: {
        weight: 116
    },
    array: [1, 2, 3, 4]
}
const user = reactive(data);

//测试代码
let b = 10
effect(() => {
    b = user.age + 20
    console.log(b);
})
let c;
effect(() => {
    c = user.message + " hello vue3"
    console.log(c);
})
let d;
effect(() => {
    d = user.info.weight + 10
    console.log(d);
})

let e;
effect(() => {
    c = user.array[0] + 10;
    console.log(c,'array');
})

user.age = 3
user.message = "11111"
user.info.weight = 110
user.array[0] = 20
```

相比Vue2.x的响应式，不用遍历属性的Object.defineProperty了，proxy代理的是整个对象，并且也可以监听数组下标的变化了。。。。除了这些，基础Proxy的功能，还能检测到属性的删减等操作

谢谢～