深入理解计算属性（compouted）与监听器(watch)

作用机制上，watch 和 computed 都是以 Vue 的依赖追踪机制为基础的，它们都试图处理这样一件事情：当某一个数据（称它为依赖数据）发生变化的时候，所有依赖这个数据的“相关”数据“自动”发生变化，也就是自动调用相关的函数去实现数据的变动。

首先来说一说 Computed
Vue 中 computed 的本质—lazy Watch
首先，先假设传入这样的一组 computed：

```
//先假设有两个 data: data_one 和 data_two
computed:{
    isComputed:function(){
        return this.data_one + 1;
    },
    isMethods:function(){
        return this.data_two + this.data_one;
    }
}
```

我们知道，在 new Vue()的时候会做一系列初始化的操作，Vue 中的 data,props，methods,computed 都是在这里初始化的：

```
export function initState (vm) {
  vm._watchers = []
  const opts = vm.$options
  if (opts.props) initProps(vm, opts.props) //初始化 props
  if (opts.methods) initMethods(vm, opts.methods) //初始化 methods
  if (opts.data) {
    initData(vm) //初始化 data
  } else {
    observe(vm._data = {}, true /* asRootData */)
  }
  if (opts.computed) initComputed(vm, opts.computed) //初始化 computed
  if (opts.watch && opts.watch !== nativeWatch) {
    initWatch(vm, opts.watch) //初始化 initWatch
```

```
  }
}
```

重点说说 initComputed()这个函数。const computedWatcherOptions = { lazy: true } //用于传入 Watcher 实例的一个对象

```
function initComputed (vm, computed {
  //声明一个 watchers，同时挂载到 Vue 实例上
  const watchers = vm._computedWatchers = Object.create(null)
  //是否是服务器渲染
  const isSSR = isServerRendering()

  //遍历传入的 computed
  for (const key in computed) {
    //userDef 是 computed 对象中的每一个方法
    const userDef = computed[key]
    const getter = typeof userDef === 'function' ? userDef : userDef.get
    if (process.env.NODE_ENV !== 'production' && getter == null) {
      warn(
        `Getter is missing for computed property "${key}".`,
        vm
      )
    }

    //如果不是服务端渲染的，就创建一个 Watcher 实例
    if (!isSSR) {
      // create internal watcher for the computed property.
      watchers[key] = new Watcher(
        vm,
        getter || noop,
        noop,
        computedWatcherOptions
      )
    }

    if (!(key in vm)) {
      //如果 computed 中的 key 没有在 vm 中，通过 defineComputed 挂载上去
```

```
        defineComputed(vm, key, userDef)
      } else if (process.env.NODE_ENV !== 'production') {
        //后面都是警告 computed 中的 key 重名的
        if (key in vm.$data) {
          warn(`The computed property "${key}" is already defined in data.`,
vm)
        } else if (vm.$options.props && key in vm.$options.props) {
          warn(`The computed property "${key}" is already defined as a prop.`,
vm)
        }
      }
    }
}
```

在 initComputed 之前，我们看到声明了一个 computedWatcherOptions 的对象，这个对象是实现"lazy Watcher"的关键。接下来看 initComputed，它先声明了一个名为 watchers 的空对象，同时在 vm 上也挂载了这个空对象。之后遍历计算属性，并把每个属性的方法赋给 userDef，如果 userDef 是 function 的话就赋给 getter，接着判断是否是服务端渲染，如果不是的话就创建一个 Watcher 实例。Watcher 实例我也在上一篇文章分析过，就不逐行分析了，不过需要注意的是，这里新建的实例中我们传入了第四个参数，也就是 computedWatcherOptions，这时，Watcher 中的逻辑就有变化了：

```
 if (options) {
    this.deep = !!options.deep
    this.user = !!options.user
    this.lazy = !!options.lazy
    this.sync = !!options.sync
 } else {
    this.deep = this.user = this.lazy = this.sync = false
 }
```

这里的 options 指的就是 computedWatcherOptions，当我们走 initData 的逻辑的时候，options 并不存在，所以 this.lazy = false，但当我们有了 computedWatcherOptions 后，this.lazy = true。同时，后面还有这样一段代码：this.dirty = this.lazy，dirty 的值也为 true 了。this.value = this.lazy

```
        ? undefined
        : this.get()
```

这段代码我们可以知道，当 lazy 为 false 时，返回的是 undefined 而不是 this.get()方法。也就是说，并不会执行 computed 中的两个方法：（请看我

开头写的 computed 示例）function(){
  return this.data_one + 1;
}
function(){
  return this.data_two + this.data_one;
}
这也就意味着，computed 的值还并没有更新。而这个逻辑也就暂时先告一段落。二. defineProperty 让我们再回到 initComputed 函数中来：if (!(key in vm)) {
  //如果 computed 中的 key 没有在 vm 中，通过 defineComputed 挂载上去
  defineComputed(vm, key, userDef)
}
可以看到，当 key 值没有挂载到 vm 上时，执行 defineComputed 函数：//一个用来组装 defineProperty 的对象
```
const sharedPropertyDefinition = {
  enumerable: true,
  configurable: true,
  get: noop,
  set: noop
}

export function defineComputed (
  target: any,
  key: string,
  userDef: Object | Function
) {
  //是否是服务端渲染，注意这个变量名 => shouldCache
  const shouldCache = !isServerRendering()
  if (typeof userDef === 'function') {
    //如果 userDef 是 function，给 sharedPropertyDefinition.get 也就是当前 key 的 getter
    //赋上 createComputedGetter(key)
    sharedPropertyDefinition.get = shouldCache
      ? createComputedGetter(key)
      : userDef
    sharedPropertyDefinition.set = noop
```

```
  } else {
    //否则就使用 userDef.get 和 userDef.set 赋值
    sharedPropertyDefinition.get = userDef.get
      ? shouldCache && userDef.cache !== false
        ? createComputedGetter(key)
        : userDef.get
      : noop
    sharedPropertyDefinition.set = userDef.set
      ? userDef.set
      : noop
  }
  if (process.env.NODE_ENV !== 'production' &&
      sharedPropertyDefinition.set === noop) {
    sharedPropertyDefinition.set = function () {
      warn(
        `Computed property "${key}" was assigned to but it has no setter.`,
        this
      )
    }
  }
  //最后，我们把这个 key 挂载到 vm 上
  Object.defineProperty(target, key, sharedPropertyDefinition)
}
```

defineComputed 中，先判断是否是服务端渲染，如果不是，说明计算属性是需要缓存的，即 shouldCache 是为 true 。接下来，判断 userDef 是否是函数，如果是就说明是我们常规 computed 的用法，将 getter 设为 createComputedGetter(key)的返回值。如果不是函数，说明这个计算属性是我们自定义的，需要使用 userDef.get 和 userDef.set 来为 getter 和 setter 赋值了，这个 else 部分我就不详细说了，不会到自定义 computed 的朋友可以看文档计算属性的 setter。最后，将 computed 的这个 key 挂载到 vm 上，当你访问这个计算属性时就会调用 getter。

```
function createComputedGetter (key) {
  return function computedGetter () {
    const watcher = this._computedWatchers && this._computedWatchers[key]
    if (watcher) {
      if (watcher.dirty) {
```

```
            watcher.evaluate()
        }
        if (Dep.target) {
            watcher.depend()
        }
        return watcher.value
    }
  }
}
```

最后我们来看 createComputedGetter 这个函数，他返回了一个函数 computedGetter，此时如果 watcher 存在的情况下，判断 watcher.dirty 是否存在，根据前面的分析，第一次新建 Watcher 实例的时候 this.dirty 是为 true 的，此时调用 watcher.evaluate()：function evaluate () {

```
    this.value = this.get()
    this.dirty = false
```

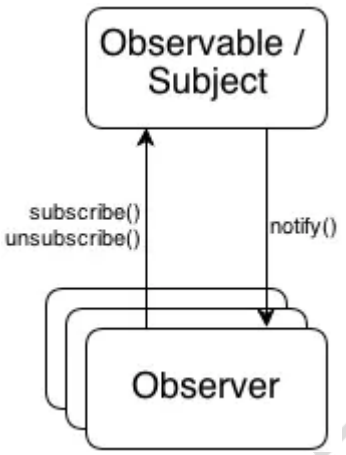
} this.get()实际上就是执行计算属性的方法。之后将 this.dirty 设为 false。另外，当我们执行 this.get()时是会为 Dep.target 赋值的，所以还会执行 watcher.depend()，将计算属性的 watcher 添加到依赖中去。最后返回 watcher.value，终于，我们获取到了计算属性的值，完成了 computed 的初始化。 计算属性的缓存——lazy Watcher，也就是"lazy watcher"。还记得 Vue 官方文档是这样形容 computed 的：我们可以将同一函数定义为一个方法而不是一个计算属性。两种方式的最终结果确实是完全相同的。然而，不同的是计算属性是基于它们的依赖进行缓存的。计算属性只有在它的相关依赖发生改变时才会重新求值。这就意味着只要 message 还没有发生改变，多次访问 reversedMessage 计算属性会立即返回之前的计算结果，而不必再次执行函数。回顾之前的代码，我们发现只要不更新计算属性的中 data 属性的值，在第一次获取值后，watch.lazy 始终为 false，也就永远不会执行 watcher.evaluate()，所以这个计算属性永远不会重新求值，一直使用上一次获得（也就是所谓的缓存）的值。一旦 data 属性的值发生变化，根据我们知道会触发 update()导致页面重新渲染（这部分内容有点跳，不清楚的朋友一定先弄懂 data 数据绑定的原理），重新 initComputed，那么 this.dirty = this.lazy = true，计算属性就会重新取值。OK，关于 computed 的原理部分到这里就分析完了。

接下来看看 Watch

Watch 的实现是基于 JavaScript 观察者 (发布/订阅) 模式

定义

观察者模式定义了对象之间一对对多的依赖关系，当一个对象改变了状态，它的所有依赖会被通知，然后自动更新。
和其他模式相比，这种模式又增加了一个原则:
在相互作用的对象之间进行松散耦合设计
所以主要的想法是我们要有一个大的对象来处理订阅（Subject/Observable），以及很多对象（Observers）被订阅然后等待事件触发



接下来基于这种模式来实现一个 Watch。
先分析一下原生 Vue 中的 watch 是如何工作的

```
const v = new Vue({
data:{
   a: 1,
   b: {
      c: 3
   }
}
})
// 实例方法$watch，监听属性"a"
v.$watch("a",()=>console.log("你修改了 a"))
                  //当 Vue 实例上的 a 变化时$watch 的回调
setTimeout(()=>{
v.a = 2
// 设置定时器，修改 a
},1000);
```

这个过程大概分为三部分：实例化 Vue、调用$watch 方法、属性变化，触

发回调

一、实例化 Vue

```
class Vue { //Vue 对象
    constructor (options) {
      this.$options=options;
      let data = this._data=this.$options.data;
      Object.keys(data).forEach(key=>this._proxy(key));
      // 拿到 data 之后，我们循环 data 里的所有属性，都传入代理函数中
      observe(data,this);
    }
    $watch(expOrFn, cb, options){   //监听赋值方法
      new Watcher(this, expOrFn, cb);
      // 传入的是 Vue 对象
    }

    _proxy(key) { //代理赋值方法
      // 当未开启监听的时候，属性的赋值使用的是代理赋值的方法
      // 而其主要的作用，是当我们访问 Vue.a 的时候，也就是 Vue 实例的属性时，我们返回的是 Vue.data.a 的属性而不是 Vue 实例上的属性
      var self = this
      Object.defineProperty(self, key, {
        configurable: true,
        enumerable: true,
        get: function proxyGetter () {
          return self._data[key]
          // 返回 Vue 实例上 data 的对应属性值
        },
        set: function proxySetter (val) {
          self._data[key] = val
        }
      })
    }
}
```

注意这里的 Object.defineProperty (obj, key , option) 方法

总共参数有三个，其中 option 中包括 set(fn), get(fn), enumerable(boolean), configurable(boolean)

set 会在 obj 的属性被修改的时候触发，而 get 是在属性被获取的时候触发，（其实属性的每次赋值，每次取值，都是调用了函数）；

constructor ：Vue 实例的构造函数，传入参数（options）的时候，constructor 就会被调用，让 Vue 对象和参数 data 产生关联，让我们可以通过 this.a 或者 vm.a 来访问 data 属性，建立关联之后，循环 data 的所有键名，将其传入到_proxy 方法

$watch：实例化 Watcher 对象

_proxy：这个方法是一个代理方法，接收一个键名，作用的对象是 Vue 对象，

当我们在 new Vue 的时候，传进去的 data 很可能包括子对象，例如在使用 Vue.data.a = {a1:1 , a2:2 }的时候，这种情况是十分常见的，但是刚才的_proxy 函数只是循环遍历了 key，如果我们要给对象的子对象增加 set 和 get 方法的时候，最好的方法就是递归；

方法也很简单，如果有属性值 == object，那么久把他的属性值拿出来，遍历一次，如果还有，继续遍历，代码如下：

```
function defineReactive (obj, key, val) {//  类似_proxy 方法，循环增加 set 和 get 方法，只不过增加了 Dep 对象和递归的方法
    var dep = new Dep()
     var childOb = observe(val)
     //这里的 val 已经是第一次传入的对象所包含的属性或者对象，会在 observe 进行筛选，决定是否继续递归
     Object.defineProperty(obj, key, {//这个 defineProperty 方法，作用对象是每次递归传入的对象，会在 Observer 对象中进行分化
        enumerable: true,
        configurable: true,
        get: ()=>{
```

```
                if(Dep.target){//这里判断是否开启监听模式（调用 watch）
                    dep.addSub(Dep.target)//调用了，则增加一个 Watcher
对象
                }
                return val//没有启用监听，返回正常应该返回 val
            },
        set:newVal=> {var value =    val
                if (newVal === value) {//新值和旧值相同的话，return
                    return
                }
                val = newVal
                childOb = observe(newVal)
                                    //这里增加 observe 方法的原因是，
当我们给属性赋的值也是对象的时候，同样要递归增加 set 和 get 方
法
                dep.notify()
                                    //这个方法是告诉 watch，你该行动
了
        }
      })
    }
    function observe (value, vm) {//递归控制函数
        if (!value || typeof value !== 'object') {//这里判断是否为对象，
如果不是对象，说明不需要继续递归
            return
        }
        return new Observer(value)//递归
    }
```

Opserver 对象是使用 defineReactive 方法循环给参数 value 设置 set 和 get 方法，同时顺便调了 observe 方法做了一个递归判断，看看是否要从 Opserver 对象开始再来一遍。

Dep 起到连接的作用：

```
class Dep {
    constructor() {
     this.subs = []    //Watcher 队列数组
    }
```

```
    addSub(sub){
      this.subs.push(sub) //增加一个 Watcher
    }
  notify(){
      this.subs.forEach(sub=>sub.update()) //触发 Watcher 身上的 update 回调（也就是你传进来的回调）
    }
}
Dep.target = null //增加一个空的 target，用来存放 Watcher

接下来来实现 Watcher
class Watcher { // 当使用了$watch 方法之后，不管有没有监听，或者触发监听，都会执行以下方法
  constructor(vm, expOrFn, cb) {
    this.cb = cb   //调用$watch 时候传进来的回调
    this.vm = vm
    this.expOrFn = expOrFn //这里的 expOrFn 是你要监听的属性或方法也就是$watch 方法的第一个参数（为了简单起见，我们这里补考录方法，只考虑单个属性的监听）
    this.value = this.get()//调用自己的 get 方法，并拿到返回值
  }
  update(){   // 还记得 Dep.notify 方法里循环的 update 么？
    this.run()
  }
  run(){//这个方法并不是实例化 Watcher 的时候执行的，而是监听的变量变化的时候才执行的
    const   value = this.get()
    if(value !==this.value){
      this.value = value
      this.cb.call(this.vm)//触发你穿进来的回调函数，call 的作用，我就不说了
    }
  }22                get(){ //向 Dep.target 赋值为 Watcher
    Dep.target = this   //将 Dep 身上的 target 赋值为 Watcher 对象
    const value = this.vm._data[this.expOrFn];//这里拿到你要监听的值，在变化之前的数值
    // 声明 value，使用 this.vm._data 进行赋值，并且触发_data[a]的 get
```

事件

```
        Dep.target = null
        return value
    }
 }
```

Watcher 在实例化的时候，重点在于 get 方法，我们来分析一下，get 方法首先把 Watcher 对象赋值给 Dep.target，随后又有一个赋值,const value = this.vm._data[this.exOrFn],之前所做的就是修改了 Vue 对象的 data(_data)的所有属性的 get 和 set？,而 Vue 对象也作为第一个参数，传给了 Watcher 对象，这个 this.vm._data 里的所有属性，在取值的时候，都会触发之前 defineReactive 方法.

回过头来再看看 get：

```
function defineReactive (obj, key, val) {
    /*.......*/
    Object.defineProperty(obj, key, {
  /*.......*/
  get: ()=>{
    if(Dep.target){ //触发这个 get 事件之前，我们刚刚对 Dep.target 赋值为 Watcher 对象
       dep.addSub(Dep.target)//这里会把我们刚赋值的 Dep.target(也就是 Watcher 对象)添加到监听队列里
    }
    return val
  },
  /*.......*/
 }
}
```

在吧 Watcher 对象放再 Dep.subs 数组中之后，new Watcher 对象所执行的任务就告一段落，此时我们有：

1.Dep.subs 数组中，已经添加了一个 Watcher 对象，

2.Dep 对象身上有 notify 方法，来触发 subs 队列中的 Watcher 的 update 方法，

3.Watcher 对象身上有 update 方法可以调用 run 方法可以触发最终我

们传进去的回调。

当我们调用 set 函数触发 notify 方法

```
Set (newVal) {
  var value =    val
  if (newVal === value) {
     return
  }
  val = newVal
  childOb = observe(newVal)
  dep.notify()//触发 Dep.subs 中所有 Watcher.update 方法
}
```

这样一个简易的 watch 就封装好了。