☰ 逐行剖析 Vue.js 源码

1. 前言

在上一篇文章中，我们知道：数据驱动视图的关键点则在于我们如何知道数据发生了变化，只要知道数据在什么时候变了，那么问题就变得迎刃而解，我们只需在数据变化的时候去通知视图更新即可。

要想知道数据什么时候被读取了或数据什么时候被改写了，其实不难， `JS` 为我们提供了 `Object.defineProperty` 方法，通过该方法我们就可以轻松的知道数据在什么时候发生变化。

2. 使Object数据变得“可观测”

数据的每次读和写能够被我们看的见，即我们能够知道数据什么时候被读取了或数据什么时候被改写了，我们将其称为数据变的‘可观测’。

要将数据变的‘可观测’，我们就要借助前言中提到的 `Object.defineProperty` 方法了，在本文中，我们就使用这个方法使数据变得“可观测”。

首先，我们定义一个数据对象 `car` ：

```js
let car = {
  'brand':'BMW',
  'price':3000
}
```

我们定义了这个 `car` 的品牌 `brand` 是 `BMW` ,价格 `price` 是3000。现在我们可以通过 `car.brand` 和 `car.price` 直接读写这个 `car` 对应的属性值。但是，当这个 `car` 的属性被读取或修改时，我们并不知情。那么应该如何做才能够让 `car` 主动告诉我们，它的属性被修改了呢?

接下来，我们使用 `Object.defineProperty()` 改写上面的例子：

```js
let car = {}
let val = 3000
Object.defineProperty(car, 'price', {
  enumerable: true,
  configurable: true,
```

逐行剖析 Vue.js 源码

```
    },
    set(newVal){
      console.log('price属性被修改了')
      val = newVal
    }
  })
```

通过 `Object.defineProperty()` 方法给 `car` 定义了一个 `price` 属性，并把这个属性的读和写分别使用 `get()` 和 `set()` 进行拦截，每当该属性进行读或写操作的时候就会触发 `get()` 和 `set()` 。如下图：

```
> car.price
  price属性被读取了
< 3000
> car.price = 5000
  price属性被修改了
< 5000
```

可以看到， `car` 已经可以主动告诉我们它的属性的读写情况了，这也意味着，这个 `car` 的数据对象已经是“可观测”的了。

为了把 `car` 的所有属性都变得可观测，我们可以编写如下代码：

```js
// 源码位置：src/core/observer/index.js

/**
 * Observer类会通过递归的方式把一个对象的所有属性都转化成可观测对象
 */
export class Observer {
  constructor (value) {
    this.value = value
    // 给value新增一个__ob__属性，值为该value的Observer实例
    // 相当于为value打上标记，表示它已经被转化成响应式了，避免重复操作
    def(value,'__ob__',this)
    if (Array.isArray(value)) {
      // 当value为数组时的逻辑
      // ...
```

☰ 逐行剖析 Vue.js 源码

```
    }

    walk (obj: Object) {
      const keys = Object.keys(obj)
      for (let i = 0; i < keys.length; i++) {
        defineReactive(obj, keys[i])
      }
    }
  }
  /**
   * 使一个对象转化成可观测对象
   * @param { Object } obj 对象
   * @param { String } key 对象的key
   * @param { Any } val 对象的某个key的值
   */
  function defineReactive (obj,key,val) {
    // 如果只传了obj和key，那么val = obj[key]
    if (arguments.length === 2) {
      val = obj[key]
    }
    if(typeof val === 'object'){
        new Observer(val)
    }
    Object.defineProperty(obj, key, {
      enumerable: true,
      configurable: true,
      get(){
        console.log(`${key}属性被读取了`);
        return val;
      },
      set(newVal){
        if(val === newVal){
            return
        }
        console.log(`${key}属性被修改了`);
        val = newVal;
      }
    })
  }
```

在上面的代码中，我们定义了 `observer` 类，它用来将一个正常的 `object` 转换成可观测的 `object` 。

然后判断数据的类型，只有 `object` 类型的数据才会调用 `walk` 将每一个属性转换成 `getter/setter` 的形式来侦测变化。 最后，在 `defineReactive` 中当传入的属性值还是一个 `object` 时使用 `new observer(val)` 来递归子属性，这样我们就可以把 `obj` 中的所有属性（包括子属性）都转换成 `getter/seter` 的形式来侦测变化。 也就是说，只要我们将一个 `object` 传到 `observer` 中，那么这个 `object` 就会变成可观测的、响应式的 `object` 。

`observer` 类位于源码的 `src/core/observer/index.js` 中。

那么现在，我们就可以这样定义 `car` ：

```js
let car = new Observer({
  'brand':'BMW',
  'price':3000
})
```

这样， `car` 的两个属性都变得可观测了。

3. 依赖收集

3.1 什么是依赖收集

在上一章中，我们迈出了第一步：让 `object` 数据变的可观测。变的可观测以后，我们就能知道数据什么时候发生了变化，那么当数据发生变化时，我们去通知视图更新就好了。那么问题又来了，视图那么大，我们到底该通知谁去变化？总不能一个数据变化了，把整个视图全部更新一遍吧，这样显然是不合理的。此时，你肯定会想到，视图里谁用到了这个数据就更新谁呗。对！你想的没错，就是这样。

视图里谁用到了这个数据就更新谁，我们换个优雅说法：我们把"谁用到了这个数据"称为"谁依赖了这个数据",我们给每个数据都建一个依赖数组(因为一个数据可能被多处使用)，谁依赖了这个数据(即谁用到了这个数据)我们就把谁放入这个依赖数组中，那么当这个数据发生变化的时候，我们就去它对应的依赖数组中，把每个依赖都通知一遍，告诉他们："你们依赖的数据变啦，你们该更新啦！"。这个过程就是依赖收集。

3.2 何时收集依赖？何时通知依赖更新？

明白了什么是依赖收集后，那么我们到底该在何时收集依赖？又该在何时通知依赖更新？

会触发 `getter` 属性，那么我们就可以在 `getter` 中收集这个依赖。同样，当这个数据变化时会触发 `setter` 属性，那么我们就可以在 `setter` 中通知依赖更新。

总结一句话就是：**在getter中收集依赖，在setter中通知依赖更新。**

3.3 把依赖收集到哪里

明白了什么是依赖收集以及何时收集何时通知后，那么我们该把依赖收集到哪里？

在3.1小节中也说了，我们给每个数据都建一个依赖数组，谁依赖了这个数据我们就把谁放入这个依赖数组中。单单用一个数组来存放依赖的话，功能好像有点欠缺并且代码过于耦合。我们应该将依赖数组的功能扩展一下，更好的做法是我们应该为每一个数据都建立一个依赖管理器，把这个数据所有的依赖都管理起来。OK，到这里，我们的依赖管理器 `Dep` 类应运而生，代码如下：

```js
// 源码位置：src/core/observer/dep.js
export default class Dep {
  constructor () {
    this.subs = []
  }

  addSub (sub) {
    this.subs.push(sub)
  }
  // 删除一个依赖
  removeSub (sub) {
    remove(this.subs, sub)
  }
  // 添加一个依赖
  depend () {
    if (window.target) {
      this.addSub(window.target)
    }
  }
  // 通知所有依赖更新
  notify () {
    const subs = this.subs.slice()
    for (let i = 0, l = subs.length; i < l; i++) {
      subs[i].update()
    }
  }
}
```

逐行剖析 Vue.js 源码

```
 * Remove an item from an array
 */
export function remove (arr, item) {
  if (arr.length) {
    const index = arr.indexOf(item)
    if (index > -1) {
      return arr.splice(index, 1)
    }
  }
}
```

在上面的依赖管理器 `Dep` 类中，我们先初始化了一个 `subs` 数组，用来存放依赖，并且定义了几个实例方法用来对依赖进行添加，删除，通知等操作。

有了依赖管理器后，我们就可以在getter中收集依赖，在setter中通知依赖更新了，代码如下：

```js
function defineReactive (obj,key,val) {
  if (arguments.length === 2) {
    val = obj[key]
  }
  if(typeof val === 'object'){
    new Observer(val)
  }
  const dep = new Dep()  //实例化一个依赖管理器，生成一个依赖管理数组dep
  Object.defineProperty(obj, key, {
    enumerable: true,
    configurable: true,
    get(){
      dep.depend()    // 在getter中收集依赖
      return val;
    },
    set(newVal){
      if(val === newVal){
          return
      }
      val = newVal;
      dep.notify()   // 在setter中通知依赖更新
    }
  })
}
```

4. 依赖到底是谁

通过上一章节，我们明白了什么是依赖？何时收集依赖？以及收集的依赖存放到何处？那么我们收集的依赖到底是谁？

虽然我们一直在说"谁用到了这个数据谁就是依赖"，但是这仅仅是在口语层面上，那么反应在代码上该如何来描述这个"谁"呢？

其实在 `Vue` 中还实现了一个叫做 `Watcher` 的类，而 `Watcher` 类的实例就是我们上面所说的那个"谁"。换句话说就是：谁用到了数据，谁就是依赖，我们就为谁创建一个 `Watcher` 实例。在之后数据变化时，我们不直接去通知依赖更新，而是通知依赖对应的 `Watch` 实例，由 `Watcher` 实例去通知真正的视图。

`Watcher` 类的具体实现如下：

```js
export default class Watcher {
  constructor (vm,expOrFn,cb) {
    this.vm = vm;
    this.cb = cb;
    this.getter = parsePath(expOrFn)
    this.value = this.get()
  }
  get () {
    window.target = this;
    const vm = this.vm
    let value = this.getter.call(vm, vm)
    window.target = undefined;
    return value
  }
  update () {
    const oldValue = this.value
    this.value = this.get()
    this.cb.call(this.vm, this.value, oldValue)
  }
}

/**
 * Parse simple path.
 * 把一个形如'data.a.b.c'的字符串路径所表示的值，从真实的data对象中取出来
 * 例如：
 * data = {a:{b:{c:2}}}
```

☰ 逐行剖析 Vue.js 源码

```
const bailRE = /[^\w.$]/
export function parsePath (path) {
  if (bailRE.test(path)) {
    return
  }
  const segments = path.split('.')
  return function (obj) {
    for (let i = 0; i < segments.length; i++) {
      if (!obj) return
      obj = obj[segments[i]]
    }
    return obj
  }
}
```

谁用到了数据，谁就是依赖，我们就为谁创建一个 `Watcher` 实例，在创建 `Watcher` 实例的过程中会自动的把自己添加到这个数据对应的依赖管理器中，以后这个 `Watcher` 实例就代表这个依赖，当数据变化时，我们就通知 `Watcher` 实例，由 `Watcher` 实例再去通知真正的依赖。

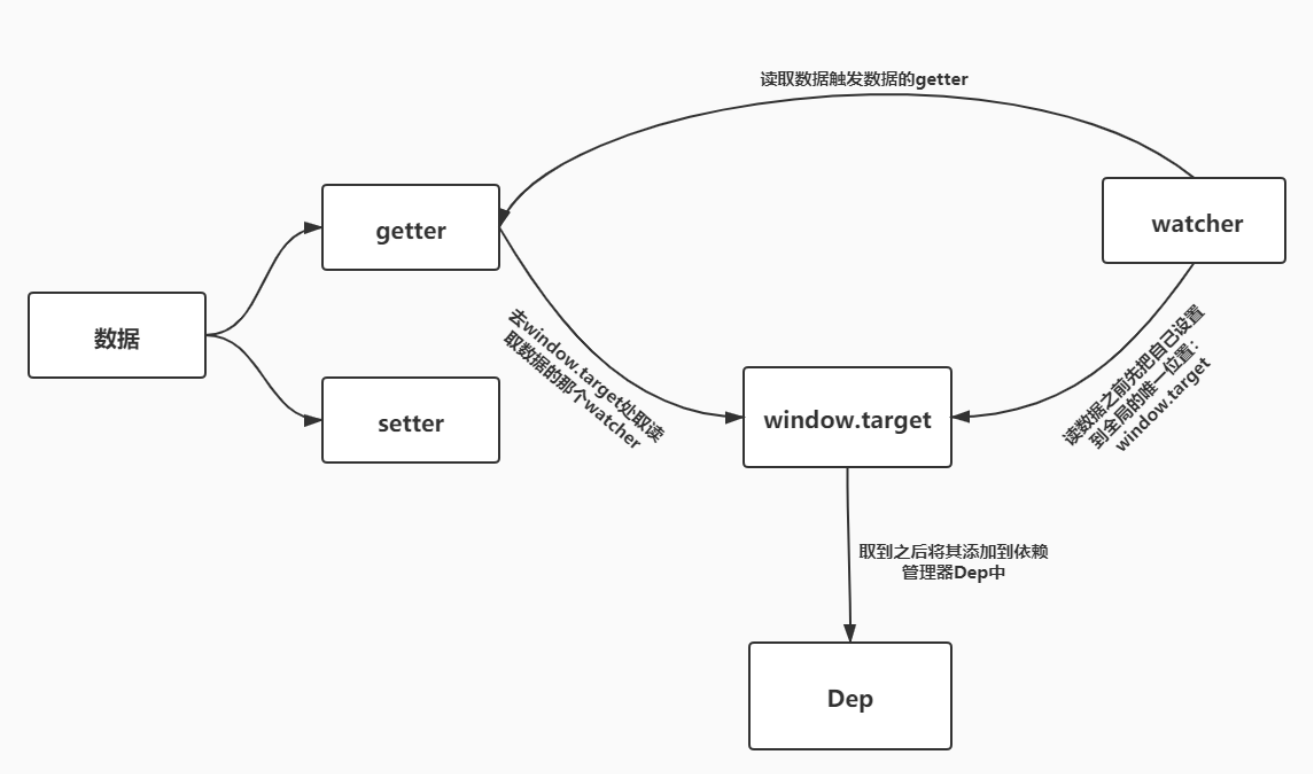
那么，在创建 `Watcher` 实例的过程中它是如何的把自己添加到这个数据对应的依赖管理器中呢?

下面我们分析 `Watcher` 类的代码实现逻辑：

1. 当实例化 `Watcher` 类时，会先执行其构造函数；
2. 在构造函数中调用了 `this.get()` 实例方法；
3. 在 `get()` 方法中，首先通过 `window.target = this` 把实例自身赋给了全局的一个唯一对象 `window.target` 上，然后通过 `let value = this.getter.call(vm, vm)` 获取一下被依赖的数据，获取被依赖数据的目的是触发该数据上面的 `getter` ，上文我们说过，在 `getter` 里会调用 `dep.depend()` 收集依赖，而在 `dep.depend()` 中取到挂载 `window.target` 上的值并将其存入依赖数组中，在 `get()` 方法最后将 `window.target` 释放掉。
4. 而当数据变化时，会触发数据的 `setter` ，在 `setter` 中调用了 `dep.notify()` 方法，在 `dep.notify()` 方法中，遍历所有依赖(即watcher实例)，执行依赖的 `update()` 方法，也就是 `Watcher` 类中的 `update()` 实例方法，在 `update()` 方法中调用数据变化的更新回调函数，从而更新视图。

简单总结一下就是： `Watcher` 先把自己设置到全局唯一的指定位置（ `window.target` ），然后读取数据。因为读取了数据，所以会触发这个数据的 `getter` 。接着，在 `getter` 中就会从全局唯一的那个位置读取当前正在读取数据的 `Watcher` ，并把这个 `watcher` 收集到

关系流程图，如下图：



以上，就彻底完成了对 `Object` 数据的侦测，依赖收集，依赖的更新等所有操作。

5. 不足之处

虽然我们通过 `Object.defineProperty` 方法实现了对 `object` 数据的可观测，但是这个方法仅仅只能观测到 `object` 数据的取值及设置值，当我们向 `object` 数据里添加一对新的 `key/value` 或删除一对已有的 `key/value` 时，它是无法观测到的，导致当我们对 `object` 数据添加或删除值时，无法通知依赖，无法驱动视图进行响应式更新。

当然， `Vue` 也注意到了这一点，为了解决这一问题， `Vue` 增加了两个全局API: `Vue.set` 和 `Vue.delete` ，这两个API的实现原理将会在后面学习全局API的时候说到。

6. 总结

首先，我们通过 `Object.defineProperty` 方法实现了对 `object` 数据的可观测，并且封装了 `Observer` 类，让我们能够方便的把 `object` 数据中的所有属性（包括子属性）都转换成 `getter/seter` 的形式来侦测变化。

☰ 逐行剖析 Vue.js 源码

最后，我们为每一个依赖都创建了一个 `Watcher` 实例，当数据发生变化时，通知 `Watcher` 实例，由 `Watcher` 实例去做真实的更新操作。

其整个流程大致如下：

1. `Data` 通过 `observer` 转换成了 `getter/setter` 的形式来追踪变化。
2. 当外界通过 `Watcher` 读取数据时，会触发 `getter` 从而将 `Watcher` 添加到依赖中。
3. 当数据发生了变化时，会触发 `setter` ，从而向 `Dep` 中的依赖（即Watcher）发送通知。
4. `Watcher` 接收到通知后，会向外界发送通知，变化通知到外界后可能会触发视图更新，也有可能触发用户的某个回调函数等。

在 GitHub 上编辑此页

上次更新: 3/24/2020, 5:37:47 AM

← 综述 Array的变化侦测 →