☰ 逐行剖析 Vue.js 源码

1. 前言

上一篇文章中我们介绍了 `Object` 数据的变化侦测方式，本篇文章我们来看一下对 `Array` 型数据的变化 `Vue` 是如何进行侦测的。

为什么 `Object` 数据和 `Array` 型数据会有两种不同的变化侦测方式?

这是因为对于 `Object` 数据我们使用的是 `JS` 提供的对象原型上的方法 `Object.defineProperty` ，而这个方法是对象原型上的，所以 `Array` 无法使用这个方法，所以我们需要对 `Array` 型数据设计一套另外的变化侦测机制。

在这里，有的同学就有疑问了，我用 `Object.defineProperty` 同样可以监测到 `Array` 型数据的变化呀，例如如下代码:

```js
let obj = { arr: [1, 2, 3] }
Object.defineProperty(obj.arr, '0', {
  set: function (newValue) {
    console.log('数据被修改了')
    value = newValue
  },
  get: function () {
    console.log('数据被读取了')
  }
})

obj.arr[0]        // 数据被读取了
obj.arr[0] = 5    // 数据被修改了
```

为什么还要对 `Array` 型数据设计一套另外的变化侦测机制呢?

这个问题问的好。对，这个例子没有错， `Array` 本质上也是 `Object` ，我们甚至可以把一个 `Array` 看作是如下样子的 `Object` (这样显然是不准确的，但是便于我们理解):

```js
let arr = [1,2,3]
// =>
let arrObj = {
    "0":1,
    "1":2,
```

可以看到，如果把 `arr` 看成 `arrObj` ，那么我们就可以使用 `Object.defineProperty` 来监测 `arr` 的变化。另外我们还知道， `Object.defineProperty` 监测 `Object` 型数据时是给 `Object` 型数据的每个 `key/value` 添加上了 `getter` 和 `setter` ，这样，对于 `Object` 型数据我们在通过 `key` 值取值或设置值时就可以被监测到。

同理，我们仔细观察，数组 `arr` 的索引值恰好就是 `arrObj` 的 `key` 值，所以我们通过数组的索引值来操作数组时是可以用 `Object.defineProperty` 监测到的。**但是，数组并不是只能由索引值来操作数组，更常用的操作数组的方法是使用数组原型上的一些方法如 `push` , `shift` 等来操作数组，当使用这些数组原型方法来操作数组时，`Object.defineProperty` 就监测不到了，所以 `Vue` 对 `Array` 型数据单独设计了数据监测方式。**

万变不离其宗，虽然对 `Array` 型数据设计了新的变化侦测机制，但是其根本思路还是不变的。那就是：还是在获取数据时收集依赖，数据变化时通知依赖更新。

下面我们就通过源码来看看 `Vue` 对 `Array` 型数据到底是如何进行变化侦测的。

2. 在哪里收集依赖

首先还是老规矩，我们得先把用到 `Array` 型数据的地方作为依赖收集起来，那么第一问题就是该在哪里收集呢?

其实 `Array` 型数据的依赖收集方式和 `Object` 数据的依赖收集方式相同，都是在 `getter` 中收集。那么问题就来了，不是说 `Array` 无法使用 `Object.defineProperty` 方法吗？无法使用怎么还在 `getter` 中收集依赖呢?

其实不然，我们回想一下平常在开发的时候，在组件的 `data` 中是不是都这么写的：

```js
data(){
  return {
    arr:[1,2,3]
  }
}
```

想想看， `arr` 这个数据始终都存在于一个 `object` 数据对象中，而且我们也说了，谁用到了数据谁就是依赖，那么要用到 `arr` 这个数据，是不是得先从 `object` 数据对象中获取一下

总结一句话就是：**Array型数据还是在getter中收集依赖。**

3. 使Array型数据可观测

上一章节中我们知道了 `Array` 型数据还是在 `getter` 中收集依赖，换句话说就是我们已经知道了 `Array` 型数据何时被读取了。

回想上一篇文章中介绍 `Object` 数据变化侦测的时候，我们先让 `Object` 数据变的可观测，即我们能够知道数据什么时候被读取了、什么时候发生变化了。同理，对于 `Array` 型数据我们也得让它变的可观测，目前我们已经完成了一半可观测，即我们只知道了 `Array` 型数据何时被读取了，而何时发生变化我们无法知道，那么接下来我们就来解决这一问题：当 `Array` 型数据发生变化时我们如何得知?

3.1 思路分析

`Object` 的变化时通过 `setter` 来追踪的，只有某个数据发生了变化，就一定会触发这个数据上的 `setter` 。但是 `Array` 型数据没有 `setter` ，怎么办?

我们试想一下，要想让 `Array` 型数据发生变化，那必然是操作了 `Array` ，而 `JS` 中提供的操作数组的方法就那么几种，我们可以把这些方法都重写一遍，在不改变原有功能的前提下，我们为其新增一些其他功能，例如下面这个例子：

```js
let arr = [1,2,3]
arr.push(4)
Array.prototype.newPush = function(val){
  console.log('arr被修改了')
  this.push(val)
}
arr.newPush(4)
```

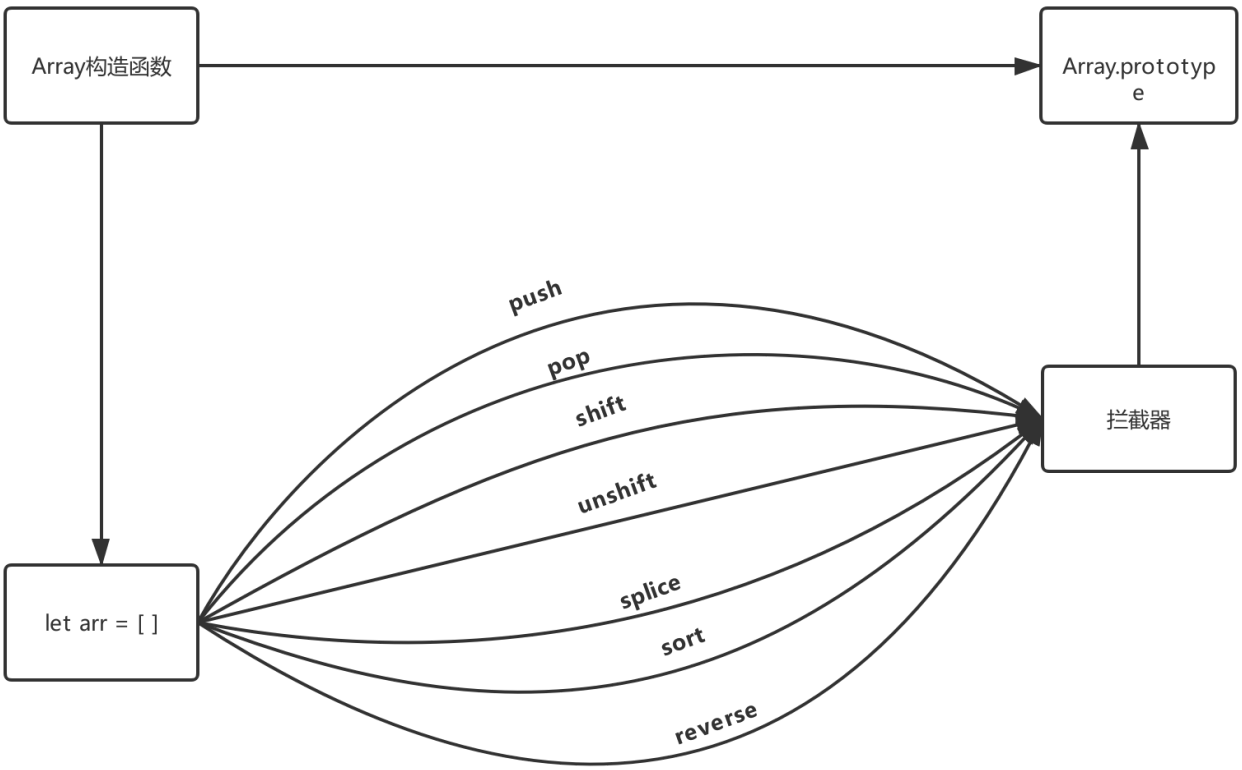
在上面这个例子中，我们针对数组的原生 `push` 方法定义个一个新的 `newPush` 方法，这个 `newPush` 方法内部调用了原生 `push` 方法，这样就保证了新的 `newPush` 方法跟原生 `push` 方法具有相同的功能，而且我们还可以在新的 `newPush` 方法内部干一些别的事情，比如通知变化。

是不是很巧妙？ `Vue` 内部就是这么干的。

☰ 逐行剖析 Vue.js 源码

3.2 数组方法拦截器

基于上一小节的思想，在 `Vue` 中创建了一个数组方法拦截器，它拦截在数组实例与 `Array.prototype` 之间，在拦截器内重写了操作数组的一些方法，当数组实例使用操作数组方法时，其实使用的是拦截器中重写的方法，而不再使用 `Array.prototype` 上的原生方法。如下图所示：



经过整理， `Array` 原型中可以改变数组自身内容的方法有7个，分别是： `push` , `pop` , `shift` , `unshift` , `splice` , `sort` , `reverse` 。那么源码中的拦截器代码如下：

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

const arrayProto = Array.prototype
// 创建一个对象作为拦截器
export const arrayMethods = Object.create(arrayProto)

// 改变数组自身内容的7个方法
const methodsToPatch = [
```

☰ 逐行剖析 Vue.js 源码

```
  'unshift',
  'splice',
  'sort',
  'reverse'
]

/**
 * Intercept mutating methods and emit events
 */
methodsToPatch.forEach(function (method) {
  const original = arrayProto[method]      // 缓存原生方法
  Object.defineProperty(arrayMethods, method, {
    enumerable: false,
    configurable: true,
    writable: true,
    value:function mutator(...args){
      const result = original.apply(this, args)
      return result
    }
  })
})

```

在上面的代码中，首先创建了继承自 `Array` 原型的空对象 `arrayMethods` ，接着在 `arrayMethods` 上使用 `object.defineProperty` 方法将那些可以改变数组自身的7个方法遍历逐个进行封装。最后，当我们使用 `push` 方法的时候，其实用的是 `arrayMethods.push` ，而 `arrayMethods.push` 就是封装的新函数 `mutator` ，也就后说，实标上执行的是函数 `mutator` ，而 `mutator` 函数内部执行了 `original` 函数，这个 `original` 函数就是 `Array.prototype` 上对应的原生方法。 那么，接下来我们就可以在 `mutato` r函数中做一些其他的事，比如说发送变化通知。

3.3 使用拦截器

在上一小节的图中，我们把拦截器做好还不够，还要把它挂载到数组实例与 `Array.prototype` 之间，这样拦截器才能够生效。

其实挂载不难，我们只需把数据的 `__proto__` 属性设置为拦截器 `arrayMethods` 即可，源码实现如下：

逐行剖析 Vue.js 源码

```
  constructor (value) {
    this.value = value
    if (Array.isArray(value)) {
      const augment = hasProto
        ? protoAugment
        : copyAugment
      augment(value, arrayMethods, arrayKeys)
    } else {
      this.walk(value)
    }
  }
}
// 能力检测：判断__proto__是否可用，因为有的浏览器不支持该属性
export const hasProto = '__proto__' in {}

const arrayKeys = Object.getOwnPropertyNames(arrayMethods)

/**
 * Augment an target Object or Array by intercepting
 * the prototype chain using __proto__
 */
function protoAugment (target, src: Object, keys: any) {
  target.__proto__ = src
}

/**
 * Augment an target Object or Array by defining
 * hidden properties.
 */
/* istanbul ignore next */
function copyAugment (target: Object, src: Object, keys: Array<string>)
  for (let i = 0, l = keys.length; i < l; i++) {
    const key = keys[i]
    def(target, key, src[key])
  }
}
```

上面代码中首先判断了浏览器是否支持 `__proto__` ，如果支持，则调用 `protoAugment` 函数把 `value.__proto__ = arrayMethods` ；如果不支持，则调用 `copyAugment` 函数把拦截器中重写的7个方法循环加入到 `value` 上。

测。

4. 再谈依赖收集

4.1 把依赖收集到哪里

在第二章中我们说了，数组数据的依赖也在 `getter` 中收集，而给数组数据添加 `getter/setter` 都是在 `Observer` 类中完成的，所以我们也应该在 `Observer` 类中收集依赖，源码如下：

```js
// 源码位置：/src/core/observer/index.js
export class Observer {
  constructor (value) {
    this.value = value
    this.dep = new Dep()     // 实例化一个依赖管理器，用来收集数组依赖
    if (Array.isArray(value)) {
      const augment = hasProto
        ? protoAugment
        : copyAugment
      augment(value, arrayMethods, arrayKeys)
    } else {
      this.walk(value)
    }
  }
}
```

上面代码中，在 `Observer` 类中实例化了一个依赖管理器，用来收集数组依赖。

4.2 如何收集依赖

在第二章中我们说了，数组的依赖也在 `getter` 中收集，那么在 `getter` 中到底该如何收集呢？这里有一个需要注意的点，那就是依赖管理器定义在 `Observer` 类中，而我们需要在 `getter` 中收集依赖，也就是说我们必须在 `getter` 中能够访问到 `Observer` 类中的依赖管理器，才能把依赖存进去。源码是这么做的：

```js
function defineReactive (obj,key,val) {
  let childOb = observe(val)
  Object.defineProperty(obj, key, {
```

☰ 逐行剖析 Vue.js 源码

```
      get(){
        if (childOb) {
          childOb.dep.depend()
        }
        return val;
      },
      set(newVal){
        if(val === newVal){
          return
        }
        val = newVal;
        dep.notify()   // 在setter中通知依赖更新
      }
    })
}

/**
 * Attempt to create an observer instance for a value,
 * returns the new observer if successfully observed,
 * or the existing observer if the value already has one.
 * 尝试为value创建一个0bserver实例，如果创建成功，直接返回新创建的Observer实例。
 * 如果 Value 已经存在一个Observer实例，则直接返回它
 */
export function observe (value, asRootData){
  if (!isObject(value) || value instanceof VNode) {
    return
  }
  let ob
  if (hasOwn(value, '__ob__') && value.__ob__ instanceof Observer) {
    ob = value.__ob__
  } else {
    ob = new Observer(value)
  }
  return ob
}
```

在上面代码中，我们首先通过 `observe` 函数为被获取的数据 `arr` 尝试创建一个 `Observer` 实例，在 `observe` 函数内部，先判断当前传入的数据上是否有 `__ob__` 属性，因为在上篇文章中说了，如果数据有 `__ob__` 属性，表示它已经被转化成响应式的了，如果没有则表示该数据还不是响应式的，那么就调用 `new Observer(value)` 将其转化成响应式的，并把数据对应的 `Observer` 实例返回。

4.3 如何通知依赖

到现在为止，依赖已经收集好了，并且也已经存放好了，那么我们该如何通知依赖呢?

其实不难，在前文说过，我们应该在拦截器里通知依赖，要想通知依赖，首先要能访问到依赖。要访问到依赖也不难，因为我们只要能访问到被转化成响应式的数据 `value` 即可，因为 `vaule` 上的 `__ob__` 就是其对应的 `Observer` 类实例，有了 `Observer` 类实例我们就能访问到它上面的依赖管理器，然后只需调用依赖管理器的 `dep.notify()` 方法，让它去通知依赖更新即可。源码如下：

```js
/**
 * Intercept mutating methods and emit events
 */
methodsToPatch.forEach(function (method) {
  const original = arrayProto[method]
  def(arrayMethods, method, function mutator (...args) {
    const result = original.apply(this, args)
    const ob = this.__ob__
    // notify change
    ob.dep.notify()
    return result
  })
})
```

上面代码中，由于我们的拦截器是挂载到数组数据的原型上的，所以拦截器中的 `this` 就是数据 `value` ，拿到 `value` 上的 `Observer` 类实例，从而你就可以调用 `Observer` 类实例上面依赖管理器的 `dep.notify()` 方法，以达到通知依赖的目的。

OK，以上就基本完成了 `Array` 数据的变化侦测。

5. 深度侦测

在前文所有讲的 `Array` 型数据的变化侦测都仅仅说的是数组自身变化的侦测，比如给数组新增一个元素或删除数组中一个元素，而在 `Vue` 中，不论是 `Object` 型数据还是 `Array` 型数据所实现的数据变化侦测都是深度侦测，所谓深度侦测就是不但要侦测数据自身的变化，还要侦测数据中所有子数据的变化。举个例子：

☰ 逐行剖析 Vue.js 源码

```
  {
    name:'NLRX',
    age:'18'
  }
]
```

数组中包含了一个对象，如果该对象的某个属性发生了变化也应该被侦测到，这就是深度侦测。

这个实现起来比较简单，源码如下：

```js
export class Observer {
  value: any;
  dep: Dep;

  constructor (value: any) {
    this.value = value
    this.dep = new Dep()
    def(value, '__ob__', this)
    if (Array.isArray(value)) {
      const augment = hasProto
        ? protoAugment
        : copyAugment
      augment(value, arrayMethods, arrayKeys)
      this.observeArray(value)   // 将数组中的所有元素都转化为可被侦测的响应式
    } else {
      this.walk(value)
    }
  }

  /**
   * Observe a list of Array items.
   */
  observeArray (items: Array<any>) {
    for (let i = 0, l = items.length; i < l; i++) {
      observe(items[i])
    }
  }
}

export function observe (value, asRootData){
  if (!isObject(value) || value instanceof VNode) {
    return
  }
```

```
    ob = value.__ob__
  } else {
    ob = new Observer(value)
  }
  return ob
}
```

在上面代码中，对于 `Array` 型数据，调用了 `observeArray()` 方法，该方法内部会遍历数组中的每一个元素，然后通过调用 `observe` 函数将每一个元素都转化成可侦测的响应式数据。

而对应 `object` 数据，在上一篇文章中我们已经在 `defineReactive` 函数中进行了递归操作。

6. 数组新增元素的侦测

对于数组中已有的元素我们已经可以将其全部转化成可侦测的响应式数据了，但是如果向数组里新增一个元素的话，我们也需要将新增的这个元素转化成可侦测的响应式数据。

这个实现起来也很容易，我们只需拿到新增的这个元素，然后调用 `observe` 函数将其转化即可。我们知道，可以向数组内新增元素的方法有3个，分别是： `push` 、 `unshift` 、 `splice` 。我们只需对这3中方法分别处理，拿到新增的元素，再将其转化即可。源码如下：

```js
/**
 * Intercept mutating methods and emit events
 */
methodsToPatch.forEach(function (method) {
  // cache original method
  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    // 如果是push或unshift方法，那么传入参数就是新增的元
        break
      case 'splice':
        inserted = args.slice(2) // 如果是splice方法，那么传入参数列表中下标为
        break
```

```
    // notify change
    ob.dep.notify()
    return result
  })
})
```

在上面拦截器定义代码中，如果是 `push` 或 `unshift` 方法，那么传入参数就是新增的元素;如果是 `splice` 方法，那么传入参数列表中下标为2的就是新增的元素，拿到新增的元素后，就可以调用 `observe` 函数将新增的元素转化成响应式的了。

7. 不足之处

前文中我们说过，对于数组变化侦测是通过拦截器实现的，也就是说只要是通过数组原型上的方法对数组进行操作就都可以侦测到，但是别忘了，我们在日常开发中，还可以通过数组的下标来操作数据，如下：

```js
let arr = [1,2,3]
arr[0] = 5;        // 通过数组下标修改数组中的数据
arr.length = 0     // 通过修改数组长度清空数组
```

而使用上述例子中的操作方式来修改数组是无法侦测到的。 同样， `Vue` 也注意到了这个问题， 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。

8. 总结

在本篇文章中，首先我们分析了对于 `Array` 型数据也在 `getter` 中进行依赖收集；其次我们发现，当数组数据被访问时我们轻而易举可以知道，但是被修改时我们却很难知道，为了解决这一问题，我们创建了数组方法拦截器，从而成功的将数组数据变的可观测。接着我们对数组的依赖收集及数据变化如何通知依赖进行了深入分析；最后我们发现 `Vue` 不但对数组自身进行了变化侦测，还对数组中的每一个元素以及新增的元素都进行了变化侦测，我们也分析了其实现原理。

以上就是对 `Array` 型数据的变化侦测分析。

逐行剖析 Vue.js 源码

← Object的变化侦测

Vue中的虚拟DOM →