

深入剖析Vue源码 - 基础的数据代理机制



AI代码助手上线啦
选中代码，体验AI替你一键
快速解读代码
立即体验

不做祖国的韭菜 2019-04-03 3,715 阅读12分钟



简单回顾一下这个系列的前两节，前两节花了大量的篇幅介绍了 **Vue** 的选项合并，选项合并是 **Vue** 实例初始化的开始，**Vue** 为开发者提供了丰富的选项配置，而每个选项都严格规定了合并的策略。然而这只是初始化中的第一步，这一节我们将对另一个重点的概念深入的分析，他就是**数据代理**，我们知道 **Vue** 大量利用了代理的思想，而除了响应式系统外，还有哪些场景也需要进行数据代理呢？这是我们这节分析的重点。

2.1 数据代理的含义

数据代理的另一个说法是数据劫持，当我们在访问或者修改对象的某个属性时，数据劫持可以拦截这个行为并进行额外的操作或者修改返回的结果。而我们知道 **Vue** 响应式系统的核心就是数据代理，代理使得数据在访问时进行依赖收集，在修改更新时对依赖进行更新，这是响应式系统的核心思路。而这一切离不开 **Vue** 对数据做了拦截代理。然而响应式并不是本节讨论的重点，这一节我们将看看数据代理在其他场景下的应用。在分析之前，我们需要掌握两种实现数据代理的方法：**Object.defineProperty** 和 **Proxy**。

2.1.1 Object.defineProperty

官方定义：**Object.defineProperty()** 方法会直接在一个对象上定义一个新属性，或者修改一个对象的现有属性，并返回这个对象。

基本用法：



复制代码

1. 数据描述符，它拥有四个属性配置

- `configurable`：数据是否可删除，可配置
- `enumerable`：属性是否可枚举
- `value`：属性值,默认为 `undefined`
- `writable`：属性是否可读写



AI代码助手上线啦
选中代码，体验AI替你一键
快速解读代码
立即体验

2. 存取描述符，它同样拥有四个属性选项

- `configurable`：数据是否可删除，可配置
- `enumerable`：属性是否可枚举
- `get`：一个给属性提供 `getter` 的方法，如果没有 `getter` 则为 `undefined`。
- `set`：一个给属性提供 `setter` 的方法，如果没有 `setter` 则为 `undefined`。

需要注意的是：数据描述符的 `value`, `writable` 和 存取描述符中的 `get`, `set` 属性不能同时存在，否则会抛出异常。有了 `Object.defineProperty` 方法，我们可以方便的利用存取描述符中的 `getter/setter` 来进行数据的监听,这也是响应式构建的雏形。`getter` 方法可以让我们在访问数据时做额外的操作处理，`setter` 方法使得我们可以在数据更新时修改返回的结果。看看下面的例子,由于设置了数据代理，当我们访问对象 `o` 的 `a` 属性时，会触发 `getter` 执行钩子函数，当修改 `a` 属性的值时，会触发 `setter` 钩子函数去修改返回的结果。



复制代码

```
1 var o = {}
2 var value;
3 Object.defineProperty(o, 'a', {
4   get() {
5     console.log('获取值')
6     return value
7   },
8   set(v) {
9     console.log('设置值')
10    value = v
11  }
12 })
13 o.a = 'sss'
14 // 设置值
15 console.log(o.a)
16 // 获取值
17 // 'qqq'
18
```

```

1  var arr = [1,2,3];
2  arr.forEach((item, index) => {
3      Object.defineProperty(arr, index, {
4          get() {
5              console.log('数组被getter拦截')
6              return item
7          },
8          set(value) {
9              console.log('数组被setter拦截')
10             return item = value
11         }
12     })
13 })
14
15 arr[1] = 4;
16 console.log(arr)
17 // 结果
18 数组被setter拦截
19 数组被getter拦截
20 4

```



AI代码助手上线啦

选中代码，体验AI替你一键快速解读代码

立即体验

显然，**已知长度的数组是可以通过索引属性来设置属性的访问器属性的。但是数组的添加确实无法进行拦截，这个也很好理解，不管是通过 `arr.push()` 还是 `arr[10] = 10` 添加的数据，数组所添加的索引值并没有预先加入数据拦截中，所以自然无法进行拦截处理。这个也是使用 `Object.defineProperty` 进行数据代理的弊端。为了解决这个问题，`Vue` 在响应式系统对数组的方法进行了重写，间接的解决了这个问题，详细细节可以参考后续的响应式系统分析。

另外如果需要拦截的对象属性嵌套多层，如果没有递归去调用 `Object.defineProperty` 进行拦截，深层次的数据也依然无法监测。

2.1.2 Proxy

为了解决像数组这类无法进行数据拦截，以及深层次的嵌套问题，`es6` 引入了 `Proxy` 的概念，它是真正在语言层面对数据拦截的定义。和 `Object.defineProperty` 一样，`Proxy` 可以修改某些操作的默认行为，但是不同的是，`Proxy` 针对目标对象会创建一个新的实例对象，并将目标对象代理到新的实例对象上。本质的区别是后者会创建一个新的对象对原对象做代理，外界对原对象的访问，都必须先通过这层代理进行拦截处理。而拦截的结果是**我们只要通过操作新的实例对象就能间接的操作真正的目标对象了**。针对

`Proxy`，下面是基础的写法：



```

4         console.log('获取值')
5         return Reflect.get(target, key, receiver)
6     },
7     set(target, key, value, receiver) {
8         console.log('设置值')
9         return Reflect.set(target, key, value, receiver)
10    }
11 })
12
13 nobj.a = '代理'
14 console.log(obj)
15 // 结果
16 设置值
17 {a: "代理"}

```



AI代码助手上线啦
选中代码，体验AI替你一键
快速解读代码
立即体验

上面的 `get`, `set` 是 `Proxy` 支持的拦截方法，而 `Proxy` 支持的拦截操作有13种之多，具体可以参照[ES6-Proxy](#)文档,前面提到，`Object.defineProperty` 的 `getter` 和 `setter` 方法并不适合监听拦截数组的变化，那么新引入的 `Proxy` 又能否做到呢？我们看下面的例子。



复制代码

```

1  var arr = [1, 2, 3]
2  let obj = new Proxy(arr, {
3      get: function (target, key, receiver) {
4          // console.log("获取数组元素" + key);
5          return Reflect.get(target, key, receiver);
6      },
7      set: function (target, key, receiver) {
8          console.log('设置数组');
9          return Reflect.set(target, key, receiver);
10     }
11 })
12 // 1. 改变已存在索引的数据
13 obj[2] = 3
14 // result: 设置数组
15 // 2. push, unshift添加数据
16 obj.push(4)
17 // result: 设置数组 * 2 (索引和length属性都会触发setter)
18 // // 3. 直接通过索引添加数组
19 obj[5] = 5
20 // result: 设置数组 * 2
21 // // 4. 删除数组元素
22 obj.splice(1, 1)
23

```

2.2 initProxy



AI代码助手上线啦

选中代码，体验AI替你一键快速解读代码

立即体验

数据拦截的思想除了为构建响应式系统准备，它也可以为**数据进行筛选过滤**。在合并选项后，`vue` 接下来会为 `vm` 实例设置一层代理，这层代理可以为**数据筛选**，这个过程究竟怎么发生的，我们看代码的实现。



复制代码

```
1 Vue.prototype._init = function(options) {
2   // 选项合并
3   ...
4   {
5     // 对vm实例进行一层代理
6     initProxy(vm);
7   }
8   ...
9 }
```

`initProxy` 的实现如下：



复制代码

```
1 // 代理函数
2 var initProxy = function initProxy (vm) {
3
4   if (hasProxy) {
5     var options = vm.$options;
6     var handlers = options.render && options.render._withStripped
7       ? getHandler
8       : hasHandler;
9     // 代理vm实例到vm属性_renderProxy
10    vm._renderProxy = new Proxy(vm, handlers);
11  } else {
12    vm._renderProxy = vm;
13  }
14 };
```

首先是判断浏览器是否支持原生的 `proxy` 。



复制代码

```
1 var hasProxy =
2   typeof Proxy !== 'undefined' && isNative(Proxy);
```

理 2. 参数 `options.render._withStripped` 代表着什么, `getHandler` 和 `hasHandler` 何理解为模板数据的访问进行数据筛选过滤。到底有什么数据需要过滤。4 才会建立这层代理, 那么在旧的浏览器, 非法的数据又将如何展示。



AI代码助手上线啦
选中代码, 体验AI替你一键
快速解读代码
立即体验

带着这些疑惑, 我们接着往下分析。

2.2.1 触发代理

源码中 `vm._renderProxy` 的使用出现在 `Vue` 实例的 `_render` 方法中, `Vue.prototype._render` 是将渲染函数转换成 `Virtual DOM` 的方法, 这部分是关于实例的挂载和模板引擎的解析, 笔者并不会在这一章节中深入分析, 我们只需要先有一个认知, ** `Vue` 内部在 `js` 和真实 `DOM` 节点中设立了一个中间层, 这个中间层就是 `Virtual DOM`, 遵循 `js -> virtual -> 真实dom` 的转换过程, 而 `Vue.prototype._render` 是前半段的转换, **当我们调用 `render` 函数时, 代理的 `vm._renderProxy` 对象便会访问到。



复制代码

```
1 Vue.prototype._render = function () {
2   ...
3   // 调用vm._renderProxy
4   vnode = render.call(vm._renderProxy, vm.$createElement);
5 }
```

那么代理的处理函数又是什么? 我们回过头看看代理选项 `handlers` 的实现。 `handlers` 函数会根据 `options.render._withStripped` 的不同执行不同的代理函数, 当使用类似 `webpack` 这样的打包工具时, 通常会使用 `vue-loader` 插件进行模板的编译, 这个时候 `options.render` 是存在的, 并且 `_withStripped` 的属性也会设置为 `true` (关于编译版本和运行时版本的区别可以参考后面章节), 所以此时代理的选项是 `hasHandler`, 在其他场景下, 代理的选项是 `getHandler`。 `getHandler`, `hasHandler` 的逻辑相似, 我们只分析使用 `vue-loader` 场景下 `hasHandler` 的逻辑。另外的逻辑, 读者可以自行分析。



复制代码

```
1 var hasHandler = {
2   // key in obj或者with作用域时, 会触发has的钩子
3   has: function has (target, key) {
4     ...
5   }
6 };
```

```
1 var obj = {
2   a: 1
3 }
4 var nObj = new Proxy(obj, {
5   has(target, key) {
6     console.log(target) // { a: 1 }
7     console.log(key) // a
8     return true
9   }
10 })
11
12 with(nObj) {
13   a = 2
14 }
```



AI代码助手上线啦
选中代码，体验AI替你一键快速解读代码
立即体验

那么这两个触发条件是否跟 `_render` 过程有直接的关系呢？答案是肯定的。 `vnode = render.call(vm._renderProxy, vm.$createElement);` 的主体是 `render` 函数，而这个 `render` 函数就是包装成 `with` 的执行语句，**在执行 `with` 语句的过程中，该作用域下变量的访问都会触发 `has` 钩子，这也是模板渲染时之所有会触发代理拦截的原因。 **我们通过代码来观察 `render` 函数的原形。

```
1 var vm = new Vue({
2   el: '#app'
3 })
4 console.log(vm.$options.render)
5
6 //输出，模板渲染使用with语句
7 f anonymous() {
8   with(this){return _c('div',{attrs:{"id":"app"}},[_v(_s(message)+_s(_test))])}
9 }
```



复制代码

2.2.2 数据过滤

我们已经大致知道了 `Proxy` 代理的访问时机，那么设置这层代理的作用又在哪里呢？首先思考一个问题，我们通过 `data` 选项去设置实例数据，那么这些数据可以随着个人的习惯任意命名吗？显然不是的，如果你使用 `js` 的关键字(像 `Object,Array,NaN`)去命名,这是不被允许的。另一方面，`Vue` 源码内部使用了以 `$_` 作为开头的内部变量，所以以 `$_` 开头的变量名也是不被允许的，这就构成了数据过滤监测的前提。接下来我们具体看 `hasHandler` 的细节实现。


```

4      // isAllowed用来判断模板上出现的变量是否合法。
5      var isAllowed = allowedGlobals(key) ||
6          (typeof key === 'string' && key.charAt(0) === '_' && !(key in
7          // _和$开头的变量不允许出现在定义的数据中，因为他是vue内部保留属性的
8          // 1. warnReservedPrefix: 警告不能以$ _开头的变量
9          // 2. warnNonPresent: 警告模板出现的变量在vue实例中未定义
10         if (!has && !isAllowed) {
11             if (key in target.$data) { warnReservedPrefix(target, key); }
12             else { warnNonPresent(target, key); }
13         }
14         return has || !isAllowed
15     }
16 };
17
18 // 模板中允许出现的非vue实例定义的变量
19 var allowedGlobals = makeMap(
20     'Infinity,undefined,NaN,isFinite,isNaN,' +
21     'parseFloat,parseInt,decodeURI,decodeURIComponent,encodeURIComponent,' +
22     'Math,Number,date,Array,Object,Boolean,String,RegExp,Map,Set,JSON,Intl,' +
23     'require' // for Webpack/Browserify
24 );

```



AI代码助手上线啦
选中代码，体验AI替你一键
快速解读代码
立即体验

首先 `allowedGlobals` 定义了 `javascript` 保留的关键字，这些关键字是不允许作为用户变量存在的。

`(typeof key === 'string' && key.charAt(0) === '_' && !(key in target.$data))` 的逻辑对以 `$_` 开头，或者是否是 `data` 中未定义的变量做判断过滤。这里对未定义变量的场景多解释几句，前面说到，代理的对象 `vm.renderProxy` 是在执行 `_render` 函数中访问的，而在使用了 `template` 模板的情况下，`render` 函数是对模板的解析结果，换言之，之所以会触发数据代理拦截是因为模板中使用了变量，例如 `<div>{{message}}</div>`。而如果我们在模板中使用了未定义的变量，这个过程就被 `proxy` 拦截，并定义为不合法的变量使用。

我们可以看看两个报错信息的源代码(是不是很熟悉):

```

1 // 模板使用未定义的变量
2 var warnNonPresent = function (target, key) {
3     warn(
4         "Property or method \"" + key + "\" is not defined on the instance but " +
5         'referenced during render. Make sure that this property is reactive, ' +
6         'either in the data option, or for class-based components, by ' +
7         'initializing the property. ' +
8         'See: https://vuejs.org/v2/guide/reactivity.html#Declaring-Reactive-Properties.',
9         target
10    );

```



复制代码



```

16     "Property \"" + key + "\" must be accessed with \"$data.\" + key + "\" because " +
17     'properties starting with "$" or "_" are not proxied in the Vue instance to
18     'prevent conflicts with Vue internals' +
19     'See: https://vuejs.org/v2/api/#data',
20     target
21   );
22 };

```



AI代码助手上线啦
选中代码，体验AI替你一键
快速解读代码
立即体验

分析到这里，前面的疑惑只剩下最后一个问题。只有在浏览器支持 `proxy` 的情况下，才会执行 `initProxy` 设置代理，那么在不支持的情况下，数据过滤就失效了，此时非法的数据定义还能正常运行吗？我们先对比下面两个结论。



复制代码

```

1  // 模板中使用_开头的变量，且在data选项中有定义
2  <div id="app">{{_test}}</div>
3  new Vue({
4    el: '#app',
5    data: {
6      _test: 'proxy'
7    }
8  })

```

1. 支持 `proxy` 浏览器的结果

2. 不支持 `proxy` 浏览器的结果

其他实现细节不在本节讨论范围内。



AI代码助手上线啦
选中代码，体验AI替你一键快速解读代码
立即体验

```
1 function initData(vm) {
2   vm._data = typeof data === 'function' ? getData(data, vm) : data || {}
3   if (!isReserved(key)) {
4     // 数据代理，用户可直接通过vm实例返回data数据
5     proxy(vm, "_data", key);
6   }
7 }
8
9 function isReserved (str) {
10   var c = (str + '').charCodeAt(0);
11   // 首字符是$, _的字符串
12   return c === 0x24 || c === 0x5F
13 }
```

`vm._data` 可以拿到最终 `data` 选项合并的结果，`isReserved` 会过滤以 `$_` 开头的变量，`proxy` 会为实例数据的访问做代理，当我们访问 `this.message` 时，实际上访问的是 `this._data.message`，而有了 `isReserved` 的筛选，即使 `this._data._test` 存在，我们依旧无法在访问 `this._test` 时拿到 `_test` 变量。这就解释了为什么会有变量没有被声明的语法错误，而 `proxy` 的实现，又是基于上述提到的 `Object.defineProperty` 来实现的。



复制代码

```
1 function proxy (target, sourceKey, key) {
2   sharedPropertyDefinition.get = function proxyGetter () {
3     // 当访问this[key]时，会代理访问this._data[key]的值
4     return this[sourceKey][key]
5   };
6   sharedPropertyDefinition.set = function proxySetter (val) {
7     this[sourceKey][key] = val;
8   };
9   Object.defineProperty(target, key, sharedPropertyDefinition);
10 }
```

2.3 小结

这一节内容，详细的介绍了数据代理在 `Vue` 的实现思路和另一个应用场景，数据代理是一种设计模式，也是一种编程思想，`Object.defineProperty` 和 `Proxy` 都可以实现数据代理，但是他们各有优劣，前者兼容



例如，你从数据库读取数据，并把它的数据存入数据库，然后你把它存入数据库，这也有助于你从数据库中读取数据。



AI代码助手上线啦

选中代码，体验AI替你一键快速解读代码

立即体验

- 深入剖析Vue源码 - 选项合并(上)
- 深入剖析Vue源码 - 选项合并(下)
- 深入剖析Vue源码 - 数据代理，关联子父组件
- 深入剖析Vue源码 - 实例挂载,编译流程
- 深入剖析Vue源码 - 完整渲染过程
- 深入剖析Vue源码 - 组件基础
- 深入剖析Vue源码 - 组件进阶
- 深入剖析Vue源码 - 响应式系统构建(上)
- 深入剖析Vue源码 - 响应式系统构建(中)
- 深入剖析Vue源码 - 响应式系统构建(下)
- 深入剖析Vue源码 - 来，跟我一起实现diff算法!
- 深入剖析Vue源码 - 揭秘Vue的事件机制
- 深入剖析Vue源码 - Vue插槽，你想了解的都在这里!
- 深入剖析Vue源码 - 你了解v-model的语法糖吗?
- 深入剖析Vue源码 - Vue动态组件的概念，你会乱吗?
- 彻底搞懂Vue中keep-alive的魔法(上)
- 彻底搞懂Vue中keep-alive的魔法(下)

标签： Vue.js

本文收录于以下专栏



Vue源码解析 专栏目录

发布关于Vue源码解析的文章

68 订阅 · 18 篇文章

订阅

上一篇 深入剖析Vue源码 - 选项合并(下)

下一篇 深入剖析Vue源码 - 完整挂载流程和...

评论 2



最热

最新

王小金Ryan 前端工程师

存取描述符中是不是重复出现了configurable和enumerable??

1年前

👍 点赞

💬 评论

...

追33333

vue-loader场景下应该是getHandler把

4年前

👍 点赞

💬 评论

...



AI代码助手上线啦

选中代码，体验AI替你一键快速解读代码

立即体验

目录

收起 ^

2.1 数据代理的含义

2.1.1 Object.defineProperty

2.1.2 Proxy

2.2 initProxy

2.2.1 触发代理

2.2.2 数据过滤

2.3 小结

相关推荐

深入剖析Vue源码 - 响应式系统构建(上)

22k阅读 · 96点赞

来，送你一本免费的Vue源码解析！

2.4k阅读 · 27点赞

深入剖析Vue源码 - 完整挂载流程和模板编译

7.0k阅读 · 42点赞

深入剖析Vue源码 - 响应式系统构建(下)

2.7k阅读 · 19点赞



精选内容

JavaScript 的 “new Function”：你不知道的黑魔法，让代码更灵活！

烛阴 · 34阅读 · 0点赞

发布第五天，我的开源项目突破 1.7 K Star！

ConardLi · 955阅读 · 34点赞

京东一面：postMessage 如何区分不同类型的消息 🤔🤔🤔

Moment · 314阅读 · 4点赞

🔥 Vue2 vs Vue3 的 h 函数终极指南：从入门到源码级深度解析

鱼樱前端 · 35阅读 · 0点赞

🔥 稀土掘金 稀土掘金 稀土掘金 稀土掘金 稀土掘金 稀土掘金 稀土掘金 稀土掘金 稀土掘金 稀土掘金



AI代码助手上线啦

选中代码，体验AI替你一键快速解读代码

立即体验

为你推荐

深入剖析Vue源码 - 组件基础

不做祖国的韭菜 5年前 👁 6.5k 👍 67 💬 11

Vue.js

vue源码分析-基础的数据代理检测

yyzzabc123 2年前 👁 86 👍 点赞 💬 评论

前端 Vue.js

vue源码分析-基础的数据代理检测

yyzzabc123 2年前 👁 46 👍 点赞 💬 评论

Vue.js

vue源码分析-基础的数据代理检测

yyzzabc123 2年前 👁 26 👍 点赞 💬 评论

Vue.js

深入剖析Vue源码 - 来，跟我一起实现diff算法！

不做祖国的韭菜 5年前 👁 6.0k 👍 80 💬 7

Vue.js

深入剖析Vue源码 - 组件进阶

不做祖国的韭菜 5年前 👁 3.6k 👍 22 💬 6

Vue.js

vue源码分析-基础的数据代理检测

yyzzabc123 2年前 👁 19 👍 点赞 💬 评论

Vue.js

深入剖析Vue源码 - 完整渲染过程

不做祖国的韭菜 5年前 👁 7.7k 👍 67 💬 11

Vue.js



深入剖析Vue源码 - 选项合并(下)

不做祖国的韭菜 6年前 3.7k 26 2

Retrofit 源码剖析-深入

wzgiceman 8年前 1.1k 47 1



AI代码助手上线啦
选中代码，体验AI替你一键
快速解读代码
立即体验

深入剖析Vue源码 - 你了解v-model的语法糖吗？

不做祖国的韭菜 5年前 14k 161 5

Vue.js

深入剖析Vue源码 - Vue插槽，你想了解的都在这里！

不做祖国的韭菜 5年前 15k 96 8

Vue.js

深入剖析Vue源码 - Vue动态组件的概念，你会乱吗？

不做祖国的韭菜 5年前 9.5k 66 6

Vue.js

深入剖析Vue源码 - 响应式系统构建(中)

不做祖国的韭菜 5年前 2.9k 22 3

Vue.js