

开发指南篇5：Vue API 盲点解析

在了解了一些实用的开发技巧和编码理念后，我们在项目的开发过程中难免也会遇到因为不熟悉Vue API而导致的技术问题，而往往就是这样的一些问题消耗了我们大量的开发时间，造成代码可读性下降、功能紊乱甚至 **bug** 量的增加，其根本原因还是自己对Vue API的“无知”。

本文将介绍Vue 项目开发中比较难以理解并可能被你忽视的API，唯有知己知彼，才能百战不殆。

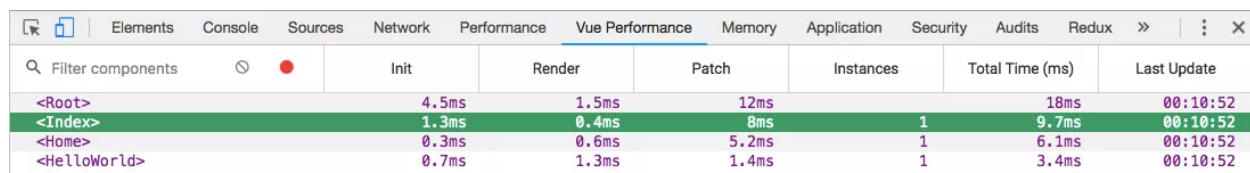
API 解析

使用performance 开启性能追踪

performance API 是Vue 全局配置API 中的一个，我们可以使用它来进行网页性能的追踪，我们可以在入口文件中添加：

```
                                javascript
if (process.env.NODE_ENV !== 'production') {
  Vue.config.performance = true;
}
```

来开启这一功能，该API（2.2.0新增）功能只适用于开发模式和支持 **performance.mark** API 的浏览器上，开启后我们可以下载[Vue Performance Devtool](#)这一chrome插件来看查看各个组件的加载情况，如图：



	Init	Render	Patch	Instances	Total Time (ms)	Last Update
<Root>	4.5ms	1.5ms	12ms		18ms	00:10:52
<Index>	1.3ms	0.4ms	8ms	1	9.7ms	00:10:52
<Home>	0.3ms	0.6ms	5.2ms	1	6.1ms	00:10:52
<HelloWorld>	0.7ms	1.3ms	1.4ms	1	3.4ms	00:10:52

从中我们可以清晰的看到页面组件在每个阶段的耗时情况，而针对耗时比较久的组件，我们便可以进行相应优化。



- performance.mark 主要用于创建标记
- performance.measure 主要用于记录两个标记的时间间隔

例如：

```
performance.mark('start'); // 创建 start 标记
performance.mark('end'); // 创建 end 标记

performance.measure('output', 'start', 'end'); // 计算两者时间间隔

performance.getEntriesByName('output'); // 获取标记，返回值是一个数组，包含了间隔时间数据
```

javascript

熟练的使用 performance 我们可以查看并分析网页的很多数据，为我们项目优化提供保障。除了上述介绍的两个方法，我们还可以使用 `performance.timing` 来计算页面各个阶段的加载情况，关于 performance.timing 的介绍可以查看我之前写的一篇文章：[利用 Navigation Timing 测量页面加载时间](#)

使用 errorHandler 来捕获异常

在浏览器异常捕获的方法上，我们熟知的一般有：`try ... catch` 和 `window.onerror`，这也是原生 JavaScript 提供给我们处理异常的方式。但是在 Vue 2.x 中如果你一如既往的想使用 `window.onerror` 来捕获异常，那么其实你是捕获不到的，因为异常信息被框架自身的异常机制捕获了，你可以使用 `errorHandler` 来进行异常信息的获取：

```
Vue.config.errorHandler = function (err, vm, info) {
  let {
    message, // 异常信息
    name, // 异常名称
    stack // 异常堆栈信息
  } = err;

  // vm 为抛出异常的 Vue 实例
  // info 为 Vue 特定的错误信息，比如错误所在的生命周期钩子
}
```

javascript

html

```
<template>
  <my-component @eventFn="doSomething"></my-component>
</template>

<script>
export default {
  methods: {
    doSomething() {
      console.log(a); // a is not defined
    }
  }
}
</script>
```

使用 Vue 中的异常捕获机制，我们可以针对捕获到的数据进行分析 and 上报，为实现前端异常监控奠定基础。关于对异常捕获的详细介绍，感兴趣的同学可以查看我的这篇文章：[谈谈前端异常捕获与上报](#)

使用 nextTick 将回调延迟到下次 DOM 更新循环之后执行

在某些情况下，我们改变页面中绑定的数据后需要对新视图进行一些操作，而这时候新视图其实还未生成，需要等待 DOM 的更新后才能获取的到，在这种场景下我们便可以使用 nextTick 来延迟回调的执行。比如未使用 nextTick 时的代码：

html

```
<template>
  <ul ref="box">
    <li v-for="(item, index) in arr" :key="index"></li>
  </ul>
</template>

<script>
export default {
  data() {
    return {
      arr: []
    }
  },
  mounted() {
```



```
    getData() {  
      this.arr = [1, 2, 3];  
      this.$refs.box.getElementsByTagName('li')[0].innerHTML = 'hello';  
    }  
  }  
}  
</script>
```

上方代码我们在实际运行的时候肯定会报错，因为我们获取 DOM 元素 li 的时候其还未被渲染，我们将方法放入 nextTick 回调中即可解决该问题：

```
                                javascript  
this.$nextTick(() => {  
  this.$refs.box.getElementsByTagName('li')[0].innerHTML = 'hello';  
})
```

当然你也可以使用 ES6 的 `async/await` 语法来改写上述方法：

```
                                javascript  
methods: {  
  async getData() {  
    this.arr = [1, 2, 3];  
  
    await this.$nextTick();  
  
    this.$refs.box.getElementsByTagName('li')[0].innerHTML = 'hello';  
  }  
}
```

那么接下来我们来分析下 Vue 是如何做到的，其源码中使用了 3 种方式：

- promise.then 延迟调用
- setTimeout(func, 0) 延迟功能
- MutationObserver 监听变化

前两种方式相信大家都比较熟悉，其都具备延迟执行的功能，我们也可以直接替换 nextTick 为这两种方式中的一种，同样可以解决问题。这里主要介绍下 [MutationObserver](#) 这一 HTML5 新特性，那么什么是 `MutationObserver` 呢？用一句话介绍就是：我们可以使用它



javascript

```
// 传入回调函数进行实例化
var observer = new MutationObserver(mutations => {
  mutations.forEach(mutation => {
    console.log(mutation.type);
  })
});

// 选择目标节点
var target = document.querySelector('#box');

// 配置观察选项
var config = {
  attributes: true, // 是否观察属性的变动
  childList: true, // 是否观察子节点的变动（指新增，删除或者更改）
  characterData: true // 是否观察节点内容或节点文本的变动
};

// 传入目标节点和观察选项
observer.observe(target, config);

// 停止观察
observer.disconnect();
```

这样我们便可以观察 id 为 box 下的 DOM 树变化，一旦发生变化就会触发相应的回调方法，实现延迟调用的功能。

使用 watch 的深度遍历和立即调用功能

相信很多同学使用 `watch` 来监听数据变化的时候通常只使用过其中的 `handler` 回调，其实其还有两个参数，便是：

- `deep` 设置为 `true` 用于监听对象内部值的变化
- `immediate` 设置为 `true` 将立即以表达式的当前值触发回调

我们来看下代码中的配置：

```
<template>
  <button @click="obj.a = 2">修改</button>
```

html



```
data() {
  return {
    obj: {
      a: 1,
    }
  },
  watch: {
    obj: {
      handler: function(newVal, oldVal) {
        console.log(newVal);
      },
      deep: true,
      immediate: true
    }
  }
}
</script>
```

以上代码我们修改了 obj 对象中 a 属性的值，我们可以触发其 watch 中的 handler 回调输出新的对象，而如果不加 `deep: true`，我们只能监听 obj 的改变，并不会触发回调。同时我们也添加了 `immediate: true` 配置，其会立即以 obj 的当前值触发回调。

在 Vue 源码中，主要使用了 [Object.defineProperty \(obj, key, option\)](#) 方法来实现数据的监听，同时其也是 Vue 数据双向绑定的关键方法之一。示例代码如下：

```
function Observer() {
  var result = null;

  Object.defineProperty(this, 'result', {
    get: function() {
      console.log('你访问了 result');
      return result;
    },
    set: function(value) {
      result = value;
      console.log('你设置了 result = ' + value);
    }
  });
}
```

javascript



```
app.result; // 你访问了 result
app.result = 11; // 你设置了 result = 11
```

我们通过实例化了 `Observer` 方法来实现了一个简单的监听数据访问与变化的功能。

`Object.defineProperty` 是 ES5 的语法，这也就是为什么 Vue 不支持 IE8 以及更低版本浏览器的主要原因。

对低开销的静态组件使用 `v-once`

Vue 提供了 `v-once` 指令用于只渲染元素和组件一次，一般可以用于存在大量静态数据组件的更新性能优化，注意是大量静态数据，因为少数情况下我们的页面渲染会因为一些静态数据而变慢。如果你需要对一个组件使用 `v-once`，可以直接在组件上绑定：

```
<my-component v-once :data="msg"></my-component>
```

html

这时候因为组件绑定了 `v-once`，所以无论 `msg` 的值如何变化，组件内渲染的永远是其第一次获取到的初始值。因此我们在使用 `v-once` 的时候需要考虑该组件今后的更新情况，避免不必要的问题产生。

使用 `$isServer` 判断当前实例是否运行于服务器

当我们的 Vue 项目中存在服务端渲染（SSR）的时候，有些项目文件可能会同时在客户端和服务端加载，这时候代码中的一些客户端浏览器才支持的属性或变量在服务端便会加载出错，比如 `window`、`document` 等，这时候我们需要进行环境的判断来区分客户端和服务端，如果你不知道 `$isServer`，那么你可能会使用 `try ... catch` 或者 `process.env.VUE_ENV` 来判断：

```
try {
  document.title = 'test';
} catch(e) {}
```

javascript

```
// process.env.VUE_ENV 需要在 webpack 中进行配置
if (process.env.VUE_ENV === 'client') {
  document.title = 'test';
}
```



javascript

```
if (this.$isServer) {  
  document.title = 'test';  
}
```

其源码中使用了 `Object.defineProperty` 来进行数据监测：

javascript

```
Object.defineProperty(Vue.prototype, '$isServer', {  
  get: isServerRendering  
});  
  
var _isServer;  
var isServerRendering = function () {  
  if (_isServer === undefined) {  
    if (!inBrowser && !inWeex && typeof global !== 'undefined') {  
      _isServer = global['process'].env.VUE_ENV === 'server';  
    } else {  
      _isServer = false;  
    }  
  }  
  return _isServer  
};
```

当我们访问 `$isServer` 属性时，其会调用 `isServerRendering` 方法，该方法会首先判断当前环境，如果在浏览器或者 Weex 下则返回 `false`，否则继续判断当前全局环境下的 `process.env.VUE_ENV` 是否为 `server` 来返回最终结果。

结语

每一门语言、一个框架都有其 API 文档，在 Vue 的项目开发过程中，很多时候当你一筹莫展之际，你可以尝试浏览一下 Vue 的 API 列表，或许你就会柳暗花明。

思考 & 作业

- 使用 `watch` 监听某一值时，同时修改该值两次会触发几次 `watch` 回调？



- 除了本文介绍的 Vue 目标外，还有哪些需要牢记并容易忽略的 API？

留言

评论将在后台进行审核，审核通过后对所有人可见

tenyiyi 前端工程师 @ 小码农

```
if (process.env.NODE_ENV !== 'production') {  
  Vue.config.performance = true;  
}
```

请问，这个process.env.NODE_ENV是在哪里配置的，我是vue-cli3.0项目，没找到定义NODE_ENV的地方，（是在vue.config.js里吗？）

▲ 0 收起评论 2月前

劳卜 前端工程师 @ TC

可以看构建基础篇3

1月前

评论审核通过后显示

评论

Honter

要是能有一篇关于SSR的就完美了

▲ 0 评论 2月前

遇白不散 自由职业者

第一问，应该是一次，数据处理是异步的。多次处理，只取最后改变的。

▲ 0 收起评论 2月前

遇白不散 自由职业者



评论审核通过后显示

评论

GentleGuo

```
<my-component :data="msg"></my-component>
```

针对上述组件，如果组件内部并没有使用传递过来的 data，那么当 msg 变化时，组件会重新渲染吗？

▲ 1 收起评论 2月前

劳卜 前端工程师 @ TC

不管使不使用组件都不会重新渲染，只会局部更新，如果没使用便不会更新
1月前

评论审核通过后显示

评论

日天哥

是handler不是hander，watch那一节错了，坑

▲ 0 收起评论 3月前

劳卜 前端工程师 @ TC

笔误，已修正
3月前

评论审核通过后显示

评论

ShineKidd

```
if (this.$isServer) {  
  document.title = 'test';  
}
```



IcyTail 前端开发

this.\$forceUpdate()

▲ 0

评论 3月前