Vue 常考进阶知识点

响应式原理

Vue 内部使用了 `Object.defineProperty()` 来实现数据响应式，通过这个函数可以监听到 `set` 和 `get` 的事件。

```
var data = { name: 'yck' }
observe(data)
let name = data.name // -> get value
data.name = 'yyy' // -> change value

function observe(obj) {
  // 判断类型
  if (!obj || typeof obj !== 'object') {
    return
  }
  Object.keys(obj).forEach(key => {
    defineReactive(obj, key, obj[key])
  })
}

function defineReactive(obj, key, val) {
  // 递归子属性
  observe(val)
  Object.defineProperty(obj, key, {
    // 可枚举
    enumerable: true,
    // 可配置
    configurable: true,
    // 自定义函数
    get: function reactiveGetter() {
      console.log('get value')
      return val
    },
    set: function reactiveSetter(newVal) {
      console.log('change value')
      val = newVal
    }
  })
}
```

以上代码简单的实现了如何监听数据的 `set` 和 `get` 的事件，但是仅仅如此是不够的，因为自定义的函数一开始是不会执行的。只有先执行了依赖收集，才能在属性更新的时候派发更新，所以接下来我们需要先触发依赖收集。

```
<div>
    {{name}}
</div>
```

在解析如上模板代码时，遇到 `{{name}}` 就会进行依赖收集。

接下来我们先来实现一个 `Dep` 类，用于解耦属性的依赖收集和派发更新操作。

```
// 通过 Dep 解耦属性的依赖和更新操作
class Dep {
  constructor() {
    this.subs = []
  }
  // 添加依赖
  addSub(sub) {
    this.subs.push(sub)
  }
  // 更新
  notify() {
    this.subs.forEach(sub => {
      sub.update()
    })
  }
}
// 全局属性，通过该属性配置 Watcher
Dep.target = null
```

以上的代码实现很简单，当需要依赖收集的时候调用 `addSub`，当需要派发更新的时候调用 `notify`。

接下来我们先来简单的了解下 Vue 组件挂载时添加响应式的过程。在组件挂载时，会先对所有需要的属性调用 `Object.defineProperty()`，然后实例化 `Watcher`，传入组件更新的回调。在实例化过程中，会对模板中的属性进行求值，触发依赖收集。

因为这一小节主要目的是学习响应式原理的细节，所以接下来的代码会简略的表达触发依赖收集时的操作。

```
class Watcher {
  constructor(obj, key, cb) {
    // 将 Dep.target 指向自己
    // 然后触发属性的 getter 添加监听
    // 最后将 Dep.target 置空
```

```
    Dep.target = this
    this.cb = cb
    this.obj = obj
    this.key = key
    this.value = obj[key]
    Dep.target = null
  }
  update() {
    // 获得新值
    this.value = this.obj[this.key]
    // 调用 update 方法更新 Dom
    this.cb(this.value)
  }
}
```

以上就是 `Watcher` 的简单实现，在执行构造函数的时候将 `Dep.target` 指向自身，从而使得收集到了对应的 `Watcher`，在派发更新的时候取出对应的 `Watcher` 然后执行 `update` 函数。

接下来，需要对 `defineReactive` 函数进行改造，在自定义函数中添加依赖收集和派发更新相关的代码。

```
function defineReactive(obj, key, val) {
  // 递归子属性
  observe(val)
  let dp = new Dep()
  Object.defineProperty(obj, key, {
    enumerable: true,
    configurable: true,
    get: function reactiveGetter() {
      console.log('get value')
      // 将 Watcher 添加到订阅
      if (Dep.target) {
        dp.addSub(Dep.target)
      }
      return val
    },
    set: function reactiveSetter(newVal) {
      console.log('change value')
      val = newVal
      // 执行 watcher 的 update 方法
      dp.notify()
    }
  })
}
```

以上所有代码实现了一个简易的数据响应式，核心思路就是手动触发一次属性的 getter 来实现依赖收集。

现在我们就来测试下代码的效果，只需要把所有的代码复制到浏览器中执行，就会发现页面的内容全部被替换了。

```
var data = { name: 'yck' }
observe(data)
function update(value) {
  document.querySelector('div').innerText = value
}
// 模拟解析到 `{{name}}` 触发的操作
new Watcher(data, 'name', update)
// update Dom innerText
data.name = 'yyy'
```

Object.defineProperty 的缺陷

以上已经分析完了 Vue 的响应式原理，接下来说一点 `Object.defineProperty` 中的缺陷。

如果通过下标方式修改数组数据或者给对象新增属性并不会触发组件的重新渲染，因为 `Object.defineProperty` 不能拦截到这些操作，更精确的来说，对于数组而言，大部分操作都是拦截不到的，只是 Vue 内部通过重写函数的方式解决了这个问题。

对于第一个问题，Vue 提供了一个 API 解决

```
export function set (target: Array<any> | Object, key: any, val: any): any {
  // 判断是否为数组且下标是否有效
  if (Array.isArray(target) && isValidArrayIndex(key)) {
    // 调用 splice 函数触发派发更新
    // 该函数已被重写
    target.length = Math.max(target.length, key)
    target.splice(key, 1, val)
    return val
  }
  // 判断 key 是否已经存在
  if (key in target && !(key in Object.prototype)) {
    target[key] = val
    return val
  }
  const ob = (target: any).__ob__
  // 如果对象不是响应式对象，就赋值返回
  if (!ob) {
    target[key] = val
    return val
  }
  // 进行双向绑定
  defineReactive(ob.value, key, val)
  // 手动派发更新
  ob.dep.notify()
```

```
  return val
}
```

对于数组而言，Vue 内部重写了以下函数实现派发更新

```
// 获得数组原型
const arrayProto = Array.prototype
export const arrayMethods = Object.create(arrayProto)
// 重写以下函数
const methodsToPatch = [
  'push',
  'pop',
  'shift',
  'unshift',
  'splice',
  'sort',
  'reverse'
]
methodsToPatch.forEach(function (method) {
  // 缓存原生函数
  const original = arrayProto[method]
  // 重写函数
  def(arrayMethods, method, function mutator (...args) {
  // 先调用原生函数获得结果
    const result = original.apply(this, args)
    const ob = this.__ob__
    let inserted
    // 调用以下几个函数时，监听新数据
    switch (method) {
      case 'push':
      case 'unshift':
        inserted = args
        break
      case 'splice':
        inserted = args.slice(2)
        break
    }
    if (inserted) ob.observeArray(inserted)
    // 手动派发更新
    ob.dep.notify()
    return result
  })
})
```

编译过程

想必大家在使用 Vue 开发的过程中，基本都是使用模板的方式。那么你有过「模板是怎么在浏览器中运行的」这种疑虑嘛?

首先直接把模板丢到浏览器中肯定是不能运行的，模板只是为了方便开发者进行开发。Vue 会通过编译器将模板通过几个阶段最终编译为 `render` 函数，然后通过执行 `render` 函数生成 Virtual DOM 最终映射为真实 DOM。

接下来我们就来学习这个编译的过程，了解这个过程中大概发生了什么事情。这个过程其中又分为三个阶段，分别为：

1. 将模板解析为 AST
2. 优化 AST
3. 将 AST 转换为 `render` 函数

在第一个阶段中，最主要的事情还是通过各种各样的正则表达式去匹配模板中的内容，然后将内容提取出来做各种逻辑操作，接下来会生成一个最基本的 AST 对象

```
{
    // 类型
    type: 1,
    // 标签
    tag,
    // 属性列表
    attrsList: attrs,
    // 属性映射
    attrsMap: makeAttrsMap(attrs),
    // 父节点
    parent,
    // 子节点
    children: []
}
```

然后会根据这个最基本的 AST 对象中的属性，进一步扩展 AST。

当然在这一阶段中，还会进行其他的一些判断逻辑。比如说对比前后开闭标签是否一致，判断根组件是否只存在一个，判断是否符合 HTML5 Content Model 规范等等问题。

接下来就是优化 AST 的阶段。在当前版本下，Vue 进行的优化内容其实还是不多的。只是对节点进行了静态内容提取，也就是将永远不会变动的节点提取了出来，实现复用 Virtual DOM，跳过对比算法的功能。在下一个大版本中，Vue 会在优化 AST 的阶段继续发力，实现更多的优化功能，尽可能的在编译阶段压榨更多的性能，比如说提取静态的属性等等优化行为。

最后一个阶段就是通过 AST 生成 `render` 函数了。其实这一阶段虽然分支有很多，但是最主要的目的就是遍历整个 AST，根据不同的条件生成不同的代码罢了。

NextTick 原理分析

`nextTick` 可以让我们在下次 DOM 更新循环结束之后执行延迟回调，用于获得更新后的 DOM。

在 Vue 2.4 之前都是使用的 microtasks，但是 microtasks 的优先级过高，在某些情况下可能会出现比事件冒泡更快的情况，但如果都使用 macrotasks 又可能会出现渲染的性能问题。所以在新版本中，会默认使用 microtasks，但在特殊情况下会使用 macrotasks，比如 v-on。

对于实现 macrotasks ，会先判断是否能使用 `setImmediate` ，不能的话降级为 `MessageChannel` ，以上都不行的话就使用 `setTimeout`

```
if (typeof setImmediate !== 'undefined' && isNative(setImmediate)) {
  macroTimerFunc = () => {
    setImmediate(flushCallbacks)
  }
} else if (
  typeof MessageChannel !== 'undefined' &&
  (isNative(MessageChannel) ||
    // PhantomJS
    MessageChannel.toString() === '[object MessageChannelConstructor]')
) {
  const channel = new MessageChannel()
  const port = channel.port2
  channel.port1.onmessage = flushCallbacks
  macroTimerFunc = () => {
    port.postMessage(1)
  }
} else {
  macroTimerFunc = () => {
    setTimeout(flushCallbacks, 0)
  }
}
```

以上代码很简单，就是判断能不能使用相应的 API。