Vuejs全家桶原理

课堂目标

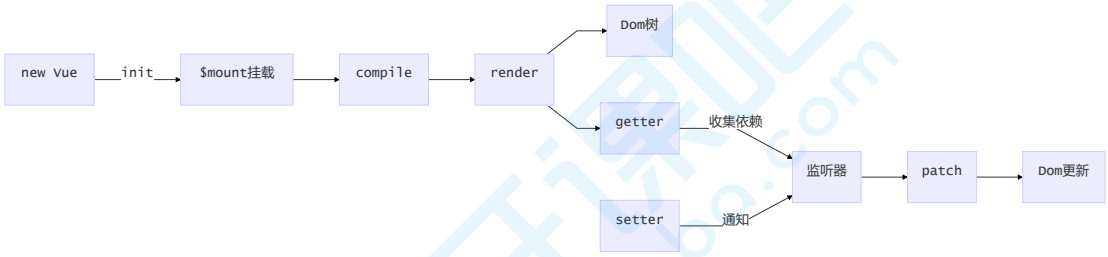
知识要点

为什么要懂原理

编程其实和武侠世界是比较像的，每一个入门的程序员，都幻想自己有朝一日，神功大成，青衣长剑，救民于水火，但其实大部分人一开始的学习方式就错了，导致一直无法进入到高手的行列，究其原因，就是过于看中招式，武器 而忽略了内功的修炼，所以任你慕容复有琅环玉洞的百家武学，还是被我乔峰一招制敌，这就是内功差距

武学之道，切勿贪多嚼不烂，博而不精不如一招鲜吃遍天 编程亦是如此，源码，就是内力修炼的捷径

Vue工作机制



初始化

在 `new Vue()` 之后。 Vue 会调用进行初始化，会初始化生命周期、事件、 props、 methods、 data、 computed 与 watch 等。其中最重要的是通过 `Object.defineProperty` 设置 `setter` 与 `getter` ，用来实现「**响应式**」以及「**依赖收集**」，后面会详细讲到，这里只要知道即可。

初始化之后调用 `$mount` 会挂载组件

编译

编译模块分为三个截断

1. parse
    1. 使用正则解析template中的vue的指令(v-xxx) 变量等等 形成语法树AST
2. optimize
    1. 标记一些静态节点，用作后面的性能优化，在diff的时候直接略过

3. generate
    1. 把第一部生成的AST 转化为渲染函数 render function

响应式

这一块是vue最核心的内容

getter和setter待会咱们会演示代码，初始化的时候通过defineProperty进行绑定，设置通知的机制

当编译生成的渲染函数被实际渲染的时候，会触发getter进行依赖收集，在数据变化的时候，触发setter进行更新

虚拟dom

Virtual DOM 是react首创，Vue2开始支持，就是用 JavaScript 对象来描述dom结构，数据修改的时候，我们先修改虚拟dom中的数据，然后数组做diff，最后再汇总所有的diff，力求做最少的dom操作，毕竟js里对比很快，而真实的dom操作太慢

```
// vdom
{
    tag: 'div',
    props:{
        name:'开课吧',
        style:{color:red},
        onClick:xx
    }
    children: [
        {
            tag: 'a',
            text: 'click me'
        }
    ]
}
```

```
<div name="开课吧" style="color:red" @click="xx">
    <a>
        click me
    </a>
</div>
```

更新视图

数据修改触发setter，然后监听器会通知进行修改，通过对比两个dom数，得到改变的地方，就是`patch`然后只需要把这些差异修改即可

下面咱们来实战一小波

defindProperity

Vue2响应式的原理

小试牛刀

```
<div id="app">
    <p>你好，<span id='name'></span></p>
</div>
<script>
    var obj = {};
    Object.defineProperty(obj, "name", {
        get: function () {
            console.log('获取name')
            return document.querySelector('#name').innerHTML;
        },
        set: function (nick) {
            console.log('设置name')
            document.querySelector('#name').innerHTML = nick;
        }
    });

    obj.name = "张无忌";
    console.log(obj.name)

</script>
```

有了思路，来新建一个类来模拟

```
class KVue {
  constructor(options) {
    this._data = options.data
    this.observer(this._data)
  }
  observer(value) {
    if (!value || (typeof value !== 'object')) {
      return
    }
    Object.keys(value).forEach((key) => {
      this.defineReactive(value, key, value[key])
    })
  }
  defineReactive(obj, key, val) {
    Object.defineProperty(obj, key, {
      enumerable: true,       /* 属性可枚举 */
      configurable: true,     /* 属性可被修改或删除 */
```

```
        get() {
          return val
        },
        set(newVal) {
          if (newVal === val) return
          this.cb(newVal)
        }
      })
    }

    cb(val) {
      console.log("更新数据了",val)
    }
  }


  let o = new KVue({
    data: {
      test: "I am test."
    }
  })
  o._data.test = "hello,kaikeba"
```

依赖收集与追踪

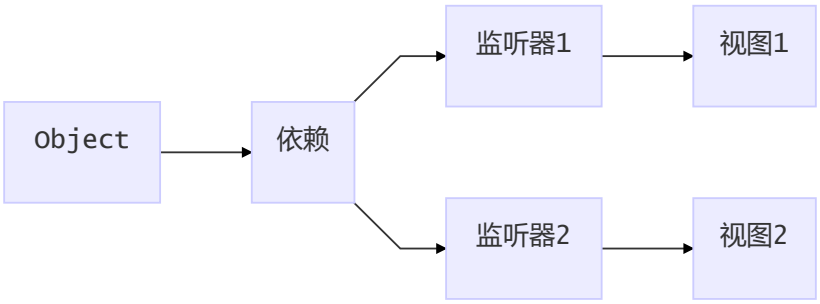
vue大概代码是酱紫

```
new Vue({
    template:
        `<div>
            <span>{{name1}}</span>
            <span>{{name2}}</span>
        <div>`,
    data: {
        text1: 'name1',
        text2: 'name2',
        text3: 'name3'
    },
    created(){
        this.name1="开课吧"
        this.name3="蜗牛"
    }
});
```

text1被修改，所以视图更新，但是text3视图没用到，所以不需要更新，如何实现呢，就需要我们的依赖收集小朋友

```
// 依赖收集小朋友
class Dep {
    constructor () {
        // 存数所有的依赖
        this.deps = []
    }

    // 在deps中添加一个监听器对象
    addDep (dep) {
        this.deps.push(dep)
    }

    // 通知所有监听器去更新视图
    notify () {
        this.deps.forEach((dep) => {
            dep.update()
        })
    }
}
```

```
//
class Watcher {
    constructor () {
        // 在new一个监听器对象时将该对象赋值给Dep.target，在get中会用到
        Dep.target = this
    }

    // 更新视图的方法
    update () {
        console.log("视图更新啦～")
    }
}
```



我们在增加了一个 Dep 类的对象，用来收集 Watcher 对象。读数据的时候，会触发 reactiveGetter 函数把当前的 Watcher 对象（存放在 Dep.target 中）收集到 Dep 类中去。

写数据的时候，则会触发 reactiveSetter 方法，通知Dep 类调用 notify 来触发所有 watcher 对象的update 方法更新对应视图

```
    constructor(options) {
      this._data = options.data
      this.observer(this._data)

      // 新建一个Watcher观察者对象，这时候Dep.target会指向这个Watcher对象
      new Watcher();
      // 在这里模拟render的过程，为了触发test属性的get函数
      console.log('模拟render，触发test的getter', this._data.test);

    }



    defineReactive(obj, key, val) {

      const dep = new Dep()

      Object.defineProperty(obj, key, {
        enumerable: true,
        configurable: true,
        get: function reactiveGetter() {
          // 将Dep.target（即当前的Watcher对象存入Dep的deps中）
          dep.addDep(Dep.target)
          return val
        },
        set: function reactiveSetter(newVal) {
```

```
            if (newVal === val) return
            // 在set的时候触发dep的notify来通知所有的watcher对象更新视图
            dep.notify()
          }
        })
      }
```

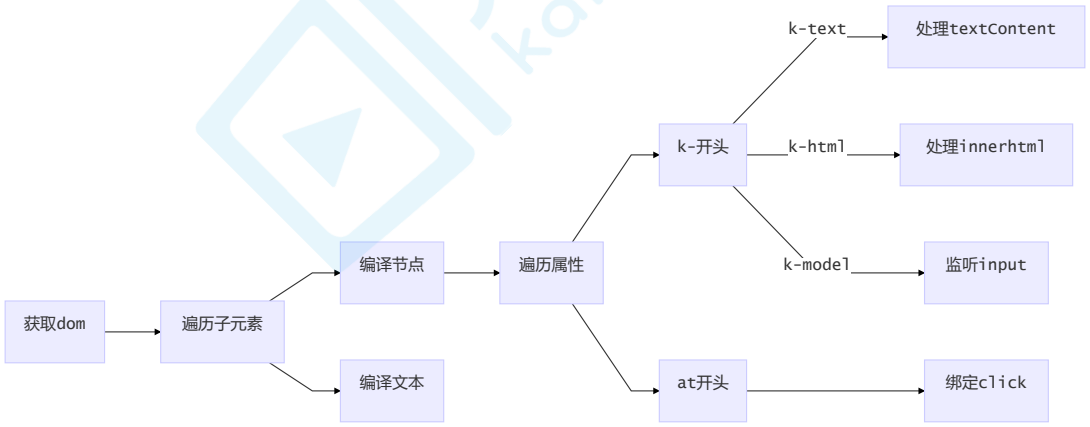
检查点

首先observer进行依赖收集，把Watcher放在Dep中，数据变化的时候调用Dep的notify方法通知watcher进行视图更新

###

编译compile

核心逻辑 获取dom，遍历dom，获取{{}}格式的变量，以及每个dom的属性，截获k-和@开头的 设置响应式



talk is cheap show me the ~~money~~ code

目标功能

```
<body>
  <div id="app">
    <p>{{name}}</p>
    <p k-text="name"></p>
    <p>{{age}}</p>
    <p>
      {{doubleAge}}
    </p>
    <input type="text" k-model="name">
    <button @click="changeName">呵呵</button>
    <div k-html="html"></div>
  </div>
  <script src='./compile.js'></script>
  <script src='./kaikeba-vue.js'></script>


  <script>
    let kaikeba = new KVue({
      el:'#app',
      data: {
        name: "I am test.",
        age:12,
        html:'<button>这是一个按钮</button>'
      },
      created(){
        console.log('开始啦')
        setTimeout(()=>{
          this.name='我是蜗牛'

        }, 1500)
      },
      methods:{
        changeName(){
          this.name = '哈喽，开课吧'
          this.age=1
          this.id = 'xx'
          console.log(1,this)

        }
      }
    })
  </script>
</body>
```

compile.js

```
class Compile {
    constructor(el,vm) {
      this.$vm = vm
      this.$el = document.querySelector(el)
      if (this.$el) {
        this.$fragment = this.node2Fragment(this.$el)
        this.compileElement(this.$fragment)
        this.$el.appendChild(this.$fragment)
      }
    }
    node2Fragment(el) {
        // 新建文档碎片 dom接口
      let fragment = document.createDocumentFragment()
      let child
      // 将原生节点拷贝到fragment
      while (child = el.firstChild) {
        fragment.appendChild(child)
      }
      return fragment
    }
    compileElement(el) {
      let childNodes = el.childNodes

      Array.from(childNodes).forEach((node) => {
        let text = node.textContent
        // 表达式文本
        // 就是识别{{}}中的数据
        let reg = /\{\{(.*)\}\}/
        // 按元素节点方式编译
        if (this.isElementNode(node)) {
          this.compile(node)
        } else if (this.isTextNode(node) && reg.test(text)) {
            // 文本 并且有{{}}
          this.compileText(node, RegExp.$1)

        }
        // 遍历编译子节点
        if (node.childNodes && node.childNodes.length) {
          this.compileElement(node)
        }
      })
    }

    compile(node) {
      let nodeAttrs = node.attributes
      Array.from(nodeAttrs).forEach( (attr)=>{
        // 规定：指令以 v-xxx 命名
        // 如 <span v-text="content"></span> 中指令为 v-text
        let attrName = attr.name    // v-text
        let exp = attr.value // content
        if (this.isDirective(attrName)) {
            let dir = attrName.substring(2) // text
            // 普通指令
```

```
            this[dir] && this[dir](node, this.$vm, exp)
        }
        if(this.isEventDirective(attrName)){
            let dir = attrName.substring(1) // text
            this.eventHandler(node, this.$vm, exp, dir)

        }
      })
    }
    compileText(node, exp) {
      this.text(node, this.$vm, exp)
    }

    isDirective(attr) {
      return attr.indexOf('k-') == 0
    }

    isEventDirective(dir) {
      return dir.indexOf('@') === 0
    }

    isElementNode(node) {
      return node.nodeType == 1
    }

    isTextNode(node) {
      return node.nodeType == 3
    }
    text(node, vm, exp) {
        this.update(node, vm, exp, 'text')
    }

    html(node, vm, exp) {
        this.update(node, vm, exp, 'html')
    }

    model(node, vm, exp) {
        this.update(node, vm, exp, 'model')
        let val = vm.exp
        node.addEventListener('input', (e)=>{
            let newValue = e.target.value
            vm[exp] = newValue
            val = newValue
        })
    }

    update(node, vm, exp, dir) {
        let updaterFn = this[dir + 'Updater']
        updaterFn && updaterFn(node, vm[exp])
        new Watcher(vm, exp, function(value) {
            updaterFn && updaterFn(node, value)
        })
    }
```

```
    // 事件处理
    eventHandler(node, vm, exp, dir) {
        let fn = vm.$options.methods && vm.$options.methods[exp]
        if (dir && fn) {
            node.addEventListener(dir, fn.bind(vm), false)
        }
    }
    textUpdater(node, value) {
        node.textContent = value
    }

    htmlUpdater(node, value) {
        node.innerHTML = value
    }

    modelUpdater(node, value) {
        node.value = value
    }
}
```

入口文件

```
class KVue {
  constructor(options) {
    this.$data = options.data
    this.$options = options
    this.observer(this.$data)
    // 新建一个Watcher观察者对象，这时候Dep.target会指向这个Watcher对象
    // new Watcher()
    // 在这里模拟render的过程，为了触发test属性的get函数
    console.log('模拟render，触发test的getter', this.$data)
    if(options.created){
      options.created.call(this)
    }
    this.$compile = new Compile(options.el, this)
  }
  observer(value) {
    if (!value || (typeof value !== 'object')) {
      return
    }
    Object.keys(value).forEach((key) => {
      this.proxyData(key)
      this.defineReactive(value, key, value[key])
```

```
    })
  }
  defineReactive(obj, key, val) {

    const dep = new Dep()
    Object.defineProperty(obj, key, {
      enumerable: true,
      configurable: true,
      get() {
        // 将Dep.target（即当前的Watcher对象存入Dep的deps中）

        Dep.target && dep.addDep(Dep.target)
        return val
      },
      set(newVal) {
        if (newVal === val) return
        val = newVal

        // 在set的时候触发dep的notify来通知所有的Watcher对象更新视图
        dep.notify()
      }
    })
  }
  proxyData(key) {

    Object.defineProperty(this, key, {
      configurable: false,
      enumerable: true,
      get() {

        return this.$data[key]
      },
      set(newVal) {
        this.$data[key] = newVal
      }
    })
  }

}
```

依赖收集 Dep

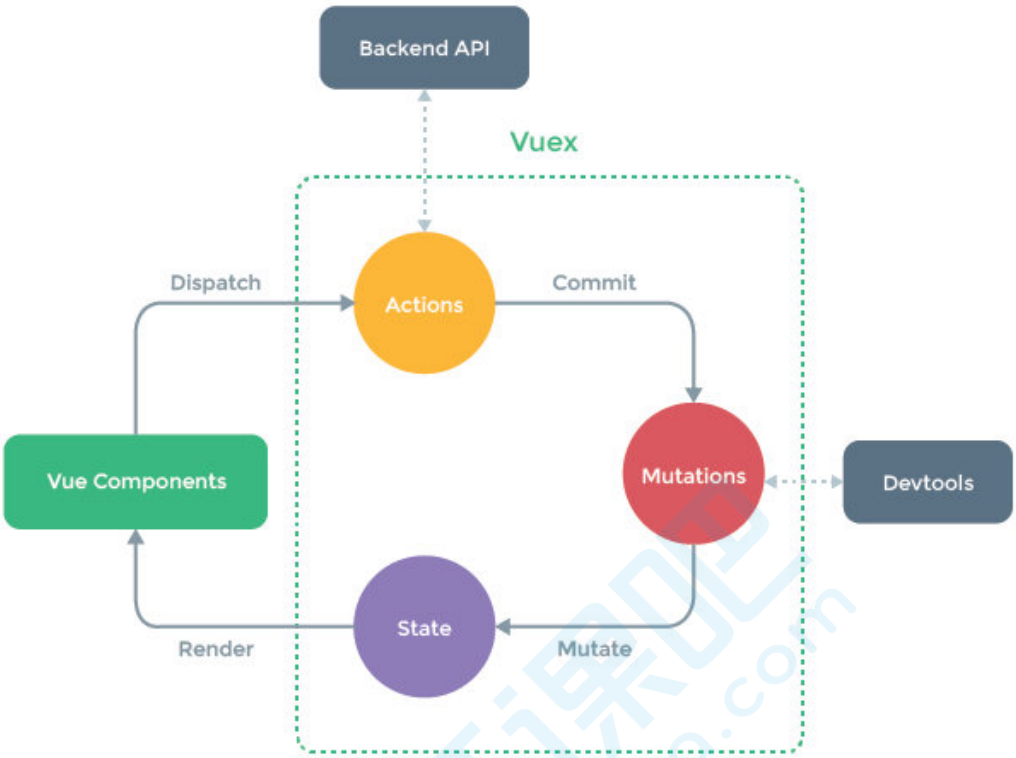
```
class Dep {
  constructor() {
    // 存数所有的依赖
    this.deps = []
  }

  // 在deps中添加一个监听器对象
  addDep(dep) {
    this.deps.push(dep)
```

```
  }
  depend() {
    Dep.target.addDep(this)
  }
  // 通知所有监听器去更新视图
  notify() {
    this.deps.forEach((dep) => {
      dep.update()
    })
  }
}
```

监听器

```
// 监听器
class Watcher {
  constructor(vm, key, cb) {
    // 在new一个监听器对象时将该对象赋值给Dep.target，在get中会用到
    // 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
    this.cb = cb
    this.vm = vm
    this.key = key
    this.value = this.get()
  }
  get() {
    Dep.target = this
    let value = this.vm[this.key]
    return value
  }
  // 更新视图的方法
  update() {
      this.value = this.get()
      this.cb.call(this.vm, this.value)
  }
}
```

vuex




```
class KVuex {
    constructor (options) {
        this.state = options.state
        this.mutations = options.mutations
        this.actions = options.actions
        // 借用vue本身的响应式的通知机制
        // state 会将需要的依赖收集在 Dep 中
        this._vm = new KVue({
            data: {
                $state: state
            }
        })
    }

    commit (type, payload, _options) {
        const entry = this.mutations[type]
        entry.forEach(handler=>handler(payload))
    }

    dispatch (type, payload) {
        const entry = this.actions[type]

        return entry(payload)
```

```
    }
}
```

https://github.com/vuejs/vuex

vue-router

使用

```
const routes = [
    { path: '/', component: Home },
    { path: '/book', component: Book },
    { path: '/movie', component: Movie }
]

const router = new VueRouter(Vue, {
    routes
})

new Vue({
    el: '#app',
    router
})
```

```
class VueRouter {
    constructor(Vue, options) {
        this.$options = options
        this.routeMap = {}
        this.app = new Vue({
            data: {
                current: '#/'
            }
        })

        this.init()
        this.createRouteMap(this.$options)
        this.initComponent(Vue)
    }

    // 初始化 hashchange
    init() {
        window.addEventListener('load', this.onHashChange.bind(this), false)
        window.addEventListener('hashchange', this.onHashChange.bind(this), false)
    }

    createRouteMap(options) {
        options.routes.forEach(item => {
```

```
            this.routeMap[item.path] = item.component
        })
    }

    // 注册组件
    initComponent(Vue) {
        Vue.component('router-link', {
            props: {
                to: String
            },
            template: '<a :href="to" rel="external nofollow" rel="external nofollow" >
<slot></slot></a>'
        })

        const _this = this
        Vue.component('router-view', {
            render(h) {
                var component = _this.routeMap[_this.app.current]
                return h(component)
            }
        })
    }

    // 获取当前 hash 串
    getHash() {
        return window.location.hash.slice(1) || '/'
    }

    // 设置当前路径
    onHashChange() {
        this.app.current = this.getHash()
    }
}
```

vue3.0展望

1. 重写虚拟dom
2. 静态树提升
3. 使用Proxy观察者机制 取代Obect.DefineProperty
4. 体积更小 压缩后大概10KB
5. 可维护性，很多包解耦
6. 全面支持TS
7. 实验性知的Time Slicing 和hooks支持

进阶思考

1. 如何监听到数组push的

2. 思考组件化

回顾