框架通识

1 MVVM

> `MVVM` 由以下三个内容组成

- `View`：界面
- `Model`：数据模型
- `ViewModel`：作为桥梁负责沟通 `View` 和 `Model`

> 在 `JQuery` 时期，如果需要刷新 `UI` 时，需要先取到对应的 `DOM` 再更新 `UI`，这样数据和业务的逻辑就和页面有强耦合。

MVVM

在 `MVVM` 中，`UI` 是通过数据驱动的，数据一旦改变就会相应的刷新对应的 `UI`，`UI` 如果改变，也会改变对应的数据。这种方式就可以在业务处理中只关心数据的流转，而无需直接和页面打交道。`ViewModel` 只关心数据和业务的处理，不关心 `View` 如何处理数据，在这种情况下，`View` 和 `Model` 都可以独立出来，任何一方改变了也不一定需要改变另一方，并且可以将一些可复用的逻辑放在一个 `ViewModel` 中，让多个 `View` 复用这个 `ViewModel`。

- 在 `MVVM` 中，最核心的也就是数据双向绑定，例如 `Angluar` 的脏数据检测，`Vue` 中的数据劫持。

脏数据检测

> 当触发了指定事件后会进入脏数据检测，这时会调用 `$digest` 循环遍历所有的数据观察者，判断当前值是否和先前的值有区别，如果检测到变化的话，会调用 `$watch` 函数，然后再次调用 `$digest` 循环直到发现没有变化。循环至少为二次 ，至多为十次。

> 脏数据检测虽然存在低效的问题，但是不关心数据是通过什么方式改变的，都可以完成任务，但是这在 `Vue` 中的双向绑定是存在问题的。并且脏数据检测可以实现批量检测出更新的值，再去统一更新 `UI`，大大减少了操作 `DOM` 的次数。所以低效也是相对的，这就仁者见仁智者见智了。

数据劫持

> `Vue` 内部使用了 `Object.defineProperty()` 来实现双向绑定，通过这个函数可以监听到 `set` 和 `get` 的事件。

```
1 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, {
    enumerable: true,
    configurable: true,
    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，还需要在适当的时候给属性添加发布订阅

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

> 在解析如上模板代码时，遇到 `{name}` 就会给属性 `name` 添加发布订阅。

```
// 通过 Dep 解耦
class Dep {
  constructor() {
    this.subs = []
  }
```

```
  addSub(sub) {
    // sub 是 Watcher 实例
    this.subs.push(sub)
  }
  notify() {
    this.subs.forEach(sub => {
      sub.update()
    })
  }
}
// 全局属性，通过该属性配置 Watcher
Dep.target = null

function update(value) {
  document.querySelector('div').innerText = value
}

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

> 接下来,对 `defineReactive` 函数进行改造

```
```

```
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()
    }
  })
}
```

> 以上实现了一个简易的双向绑定，核心思路就是手动触发一次属性的 `getter` 来实现发布订阅的添加

Proxy 与 Object.defineProperty 对比

> `Object.defineProperty` 虽然已经能够实现双向绑定了，但是他还是有缺陷的。

- 只能对属性进行数据劫持，所以需要深度遍历整个对象 对于数组不能监听到数据的变化
- 虽然 `Vue` 中确实能检测到数组数据的变化，但是其实是使用了 `hack` 的办法，并且也是有缺陷的。

```
const arrayProto = Array.prototype
export const arrayMethods = Object.create(arrayProto)
// hack 以下几个函数
const methodsToPatch = [
  'push',
  'pop',
  'shift',
  'unshift',
```

```
  'splice',
  'sort',
  'reverse'
]
methodsToPatch.forEach(function (method) {
  // 获得原生函数
  const original = arrayProto[method]
  def(arrayMethods, method, function mutator (...args) {
    // 调用原生函数
    const result = original.apply(this, args)
    const ob = this.__ob__
    let inserted
    switch (method) {
      case 'push':
      case 'unshift':
        inserted = args
        break
      case 'splice':
        inserted = args.slice(2)
        break
    }
    if (inserted) ob.observeArray(inserted)
    // 触发更新
    ob.dep.notify()
    return result
  })
})
```

> 反观 `Proxy` 就没以上的问题，原生支持监听数组变化，并且可以直接对整个对象进行拦截，所以 `Vue` 也将在下个大版本中使用 `Proxy` 替换 `Object.defineProperty`

```
let onWatch = (obj, setBind, getLogger) => {
  let handler = {
    get(target, property, receiver) {
      getLogger(target, property)
      return Reflect.get(target, property, receiver);
    },
    set(target, property, value, receiver) {
      setBind(value);
      return Reflect.set(target, property, value);
    }
  };
  return new Proxy(obj, handler);
};
```

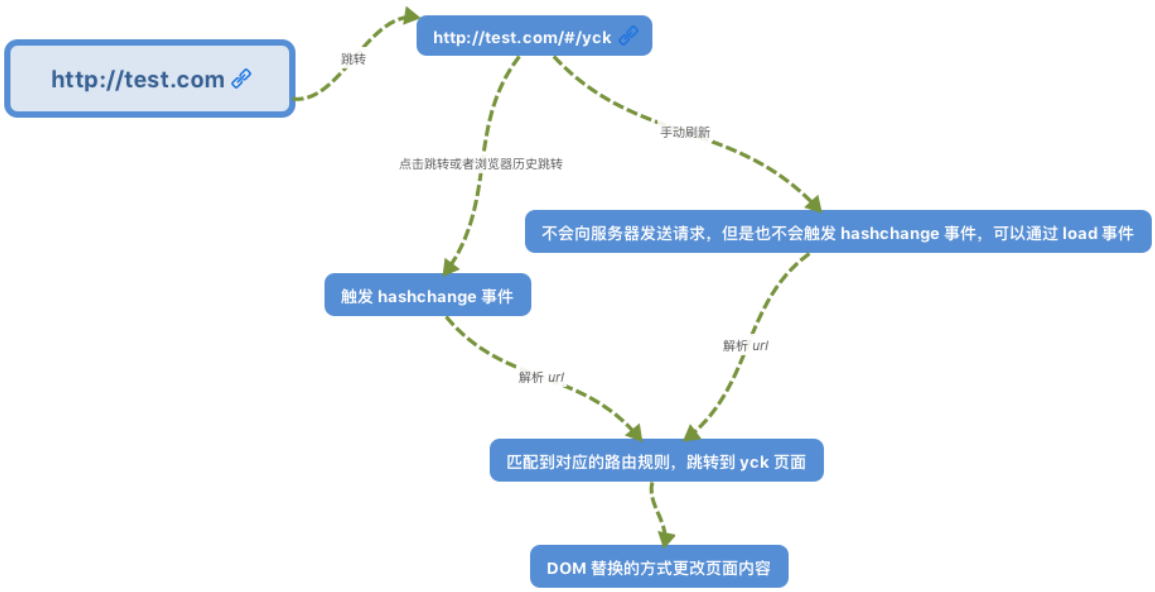
```
14
15 let obj = { a: 1 }
16 let value
17 let p = onWatch(obj, (v) => {
18   value = v
19 }, (target, property) => {
20   console.log(`Get '${property}' = ${target[property]}`);
21 })
22 p.a = 2 // bind `value` to `2`
23 p.a // -> Get 'a' = 2
```

2 路由原理

> 前端路由实现起来其实很简单，本质就是监听 `URL` 的变化，然后匹配路由规则，显示相应的页面，并且无须刷新。目前单页面使用的路由就只有两种实现方式

- `hash` 模式
- `history` 模式

> `www.test.com/##/` 就是 `Hash URL`，当 `##` 后面的哈希值发生变化时，不会向服务器请求数据，可以通过 `hashchange` 事件来监听到 `URL` 的变化，从而进行跳转页面。



> `History`模式是 `HTML5` 新推出的功能，比之 `Hash URL` 更加美观

3 Virtual Dom

为什么需要 Virtual Dom

> 众所周知，操作 `DOM` 是很耗费性能的一件事情，既然如此，我们可以考虑通过 `JS` 对象来模拟 `DOM` 对象，毕竟操作 `JS` 对象比操作 `DOM` 省时的多

```
// 假设这里模拟一个 ul，其中包含了 5 个 li
[1, 2, 3, 4, 5]
// 这里替换上面的 li
[1, 2, 5, 4]
```

> 从上述例子中，我们一眼就可以看出先前的 `ul` 中的第三个 `li` 被移除了，四五替换了位置。

- 如果以上操作对应到 `DOM` 中，那么就是以下代码

```
// 删除第三个 li
ul.childNodes[2].remove()
// 将第四个 li 和第五个交换位置
let fromNode = ul.childNodes[4]
let toNode = node.childNodes[3]
let cloneFromNode = fromNode.cloneNode(true)
let cloenToNode = toNode.cloneNode(true)
ul.replaceChild(cloneFromNode, toNode)
ul.replaceChild(cloenToNode, fromNode)
```

> 当然在实际操作中，我们还需要给每个节点一个标识，作为判断是同一个节点的依据。所以这也是 `Vue` 和 `React` 中官方推荐列表里的节点使用唯一的 `key` 来保证性能。

- 那么既然 `DOM` 对象可以通过 `JS` 对象来模拟，反之也可以通过 `JS` 对象来渲染出对应的 `DOM`
- 以下是一个 `JS` 对象模拟 `DOM` 对象的简单实现

```
export default class Element {
  /**
   * @param {String} tag 'div'
   * @param {Object} props { class: 'item' }
   * @param {Array} children [ Element1, 'text']
   * @param {String} key option
   */
  constructor(tag, props, children, key) {
    this.tag = tag
    this.props = props
```

```
    if (Array.isArray(children)) {
      this.children = children
    } else if (isString(children)) {
      this.key = children
      this.children = null
    }
    if (key) this.key = key
  }
  // 渲染
  render() {
    let root = this._createElement(
      this.tag,
      this.props,
      this.children,
      this.key
    )
    document.body.appendChild(root)
    return root
  }
  create() {
    return this._createElement(this.tag, this.props,
this.children, this.key)
  }
  // 创建节点
  _createElement(tag, props, child, key) {
    // 通过 tag 创建节点
    let el = document.createElement(tag)
    // 设置节点属性
    for (const key in props) {
      if (props.hasOwnProperty(key)) {
        const value = props[key]
        el.setAttribute(key, value)
      }
    }
    if (key) {
      el.setAttribute('key', key)
    }
    // 递归添加子节点
    if (child) {
      child.forEach(element => {
        let child
        if (element instanceof Element) {
          child = this._createElement(
            element.tag,
            element.props,
```

```
            element.children,
            element.key
          )
        } else {
          child = document.createTextNode(element)
        }
        el.appendChild(child)
      })
    }
    return el
  }
}
```

Virtual Dom 算法简述

- 既然我们已经通过 `JS` 来模拟实现了 `DOM`，那么接下来的难点就在于如何判断旧的对象和新的对象之间的差异。
- `DOM` 是多叉树的结构，如果需要完整的对比两颗树的差异，那么需要的时间复杂度会是 `O(n ^ 3)`，这个复杂度肯定是不能接受的。于是 `React` 团队优化了算法，实现了 `O(n)` 的复杂度来对比差异。
- 实现`O(n)` 复杂度的关键就是只对比同层的节点，而不是跨层对比，这也是考虑到在实际业务中很少会去跨层的移动 `DOM` 元素

> 所以判断差异的算法就分为了两步

- 首先从上至下，从左往右遍历对象，也就是树的深度遍历，这一步中会给每个节点添加索引，便于最后渲染差异
- 一旦节点有子元素，就去判断子元素是否有不同

Virtual Dom 算法实现

树的递归

- 首先我们来实现树的递归算法，在实现该算法前，先来考虑下两个节点对比会有几种情况
- 新的节点的 `tagName` 或者 `key` 和旧的不同，这种情况代表需要替换旧的节点，并且也不再需要遍历新旧节点的子元素了，因为整个旧节点都被删掉了
- 新的节点的 `tagName` 和 `key`（可能都没有）和旧的相同，开始遍历子树
- 没有新的节点，那么什么都不用做

```
import { StateEnums, isString, move } from './util'
import Element from './element'

export default function diff(oldDomTree, newDomTree) {
  // 用于记录差异
  let pathchs = {}
```

```
  // 一开始的索引为 0
  dfs(oldDomTree, newDomTree, 0, pathchs)
  return pathchs
}

function dfs(oldNode, newNode, index, patches) {
  // 用于保存子树的更改
  let curPatches = []
  // 需要判断三种情况
  // 1.没有新的节点，那么什么都不用做
  // 2.新的节点的 tagName 和 `key` 和旧的不同，就替换
  // 3.新的节点的 tagName 和 key（可能都没有） 和旧的相同，开始遍历子树
  if (!newNode) {
  } else if (newNode.tag === oldNode.tag && newNode.key === oldNode.key) {
    // 判断属性是否变更
    let props = diffProps(oldNode.props, newNode.props)
    if (props.length) curPatches.push({ type: StateEnums.ChangeProps, props })
    // 遍历子树
    diffChildren(oldNode.children, newNode.children, index, patches)
  } else {
    // 节点不同，需要替换
    curPatches.push({ type: StateEnums.Replace, node: newNode })
  }

  if (curPatches.length) {
    if (patches[index]) {
      patches[index] = patches[index].concat(curPatches)
    } else {
      patches[index] = curPatches
    }
  }
}
```

判断属性的更改

> 判断属性的更改也分三个步骤

- 遍历旧的属性列表，查看每个属性是否还存在于新的属性列表中
- 遍历新的属性列表，判断两个列表中都存在的属性的值是否有变化
- 在第二步中同时查看是否有属性不存在与旧的属性列列表中

```
```

```
function diffProps(oldProps, newProps) {
  // 判断 Props 分以下三步骤
  // 先遍历 oldProps 查看是否存在删除的属性
  // 然后遍历 newProps 查看是否有属性值被修改
  // 最后查看是否有属性新增
  let change = []
  for (const key in oldProps) {
    if (oldProps.hasOwnProperty(key) && !newProps[key]) {
      change.push({
        prop: key
      })
    }
  }
  for (const key in newProps) {
    if (newProps.hasOwnProperty(key)) {
      const prop = newProps[key]
      if (oldProps[key] && oldProps[key] !== newProps[key]) {
        change.push({
          prop: key,
          value: newProps[key]
        })
      } else if (!oldProps[key]) {
        change.push({
          prop: key,
          value: newProps[key]
        })
      }
    }
  }
  return change
}
```

判断列表差异算法实现

> 这个算法是整个 `Virtual Dom` 中最核心的算法，且让我一一为你道来。 这里的主要步骤其实和判断属性差异是类似的，也是分为三步

- 遍历旧的节点列表，查看每个节点是否还存在于新的节点列表中
- 遍历新的节点列表，判断是否有新的节点
- 在第二步中同时判断节点是否有移动

> PS：该算法只对有 `key` 的节点做处理

```
function listDiff(oldList, newList, index, patches) {
  // 为了遍历方便，先取出两个 list 的所有 keys
  let oldKeys = getKeys(oldList)
```

```
  let newKeys = getKeys(newList)
  let changes = []

  // 用于保存变更后的节点数据
  // 使用该数组保存有以下好处
  // 1.可以正确获得被删除节点索引
  // 2.交换节点位置只需要操作一遍 DOM
  // 3.用于 `diffChildren` 函数中的判断，只需要遍历
  // 两个树中都存在的节点，而对于新增或者删除的节点来说，完全没必要
  // 再去判断一遍
  let list = []
  oldList &&
    oldList.forEach(item => {
      let key = item.key
      if (isString(item)) {
        key = item
      }
      // 寻找新的 children 中是否含有当前节点
      // 没有的话需要删除
      let index = newKeys.indexOf(key)
      if (index === -1) {
        list.push(null)
      } else list.push(key)
    })
  // 遍历变更后的数组
  let length = list.length
  // 因为删除数组元素是会更改索引的
  // 所有从后往前删可以保证索引不变
  for (let i = length - 1; i >= 0; i--) {
    // 判断当前元素是否为空，为空表示需要删除
    if (!list[i]) {
      list.splice(i, 1)
      changes.push({
        type: StateEnums.Remove,
        index: i
      })
    }
  }
  // 遍历新的 list，判断是否有节点新增或移动
  // 同时也对 `list` 做节点新增和移动节点的操作
  newList &&
    newList.forEach((item, i) => {
      let key = item.key
      if (isString(item)) {
        key = item
```

```
      }
      // 寻找旧的 children 中是否含有当前节点
      let index = list.indexOf(key)
      // 没找到代表新节点，需要插入
      if (index === -1 || key == null) {
        changes.push({
          type: StateEnums.Insert,
          node: item,
          index: i
        })
        list.splice(i, 0, key)
      } else {
        // 找到了，需要判断是否需要移动
        if (index !== i) {
          changes.push({
            type: StateEnums.Move,
            from: index,
            to: i
          })
          move(list, index, i)
        }
      }
    })
  return { changes, list }
}

function getKeys(list) {
  let keys = []
  let text
  list &&
    list.forEach(item => {
      let key
      if (isString(item)) {
        key = [item]
      } else if (item instanceof Element) {
        key = item.key
      }
      keys.push(key)
    })
  return keys
}
```

遍历子元素打标识

> 对于这个函数来说，主要功能就两个

- 判断两个列表差异
  - 给节点打上标记
  - 总体来说，该函数实现的功能很简单

```
function diffChildren(oldChild, newChild, index, patches) {
  let { changes, list } = listDiff(oldChild, newChild, index, patches)
  if (changes.length) {
    if (patches[index]) {
      patches[index] = patches[index].concat(changes)
    } else {
      patches[index] = changes
    }
  }
  // 记录上一个遍历过的节点
  let last = null
  oldChild &&
    oldChild.forEach((item, i) => {
      let child = item && item.children
      if (child) {
        index =
          last && last.children ? index + last.children.length + 1 : index + 1
        let keyIndex = list.indexOf(item.key)
        let node = newChild[keyIndex]
        // 只遍历新旧中都存在的节点，其他新增或者删除的没必要遍历
        if (node) {
          dfs(item, node, index, patches)
        }
      } else index += 1
      last = item
    })
}
```

渲染差异

> 通过之前的算法，我们已经可以得出两个树的差异了。既然知道了差异，就需要局部去更新 `DOM` 了，下面就让我们来看看 `Virtual Dom` 算法的最后一步骤

这个函数主要两个功能

- 深度遍历树，将需要做变更操作的取出来
- 局部更新 `DOM`

```
let index = 0
```

```
export default function patch(node, patchs) {
  let changes = patchs[index]
  let childNodes = node && node.childNodes
  // 这里的深度遍历和 diff 中是一样的
  if (!childNodes) index += 1
  if (changes && changes.length && patchs[index]) {
    changeDom(node, changes)
  }
  let last = null
  if (childNodes && childNodes.length) {
    childNodes.forEach((item, i) => {
      index =
        last && last.children ? index + last.children.length +
1 : index + 1
      patch(item, patchs)
      last = item
    })
  }
}

function changeDom(node, changes, noChild) {
  changes &&
    changes.forEach(change => {
      let { type } = change
      switch (type) {
        case StateEnums.ChangeProps:
          let { props } = change
          props.forEach(item => {
            if (item.value) {
              node.setAttribute(item.prop, item.value)
            } else {
              node.removeAttribute(item.prop)
            }
          })
          break
        case StateEnums.Remove:
          node.childNodes[change.index].remove()
          break
        case StateEnums.Insert:
          let dom
          if (isString(change.node)) {
            dom = document.createTextNode(change.node)
          } else if (change.node instanceof Element) {
            dom = change.node.create()
          }
```

```
          node.insertBefore(dom, node.childNodes[change.index])
          break
        case StateEnums.Replace:
          node.parentNode.replaceChild(change.node.create(), node)
          break
        case StateEnums.Move:
          let fromNode = node.childNodes[change.from]
          let toNode = node.childNodes[change.to]
          let cloneFromNode = fromNode.cloneNode(true)
          let cloenToNode = toNode.cloneNode(true)
          node.replaceChild(cloneFromNode, toNode)
          node.replaceChild(cloenToNode, fromNode)
          break
        default:
          break
      }
    })
}
```

Virtual Dom 算法的实现也就是以下三步

- 通过 `JS` 来模拟创建 `DOM` 对象
- 判断两个对象的差异
- 渲染差异

```
let test4 = new Element('div', { class: 'my-div' }, ['test4'])
let test5 = new Element('ul', { class: 'my-div' }, ['test5'])

let test1 = new Element('div', { class: 'my-div' }, [test4])

let test2 = new Element('div', { id: '11' }, [test5, test4])

let root = test1.render()

let pathchs = diff(test1, test2)
console.log(pathchs)

setTimeout(() => {
  console.log('开始更新')
  patch(root, pathchs)
  console.log('结束更新')
}, 1000)
```