第02课：灵活的 data，死板的 props

事件驱动

在前端来说数据驱动式框架，必然离不开事件驱动，事件驱动一定程度上弥补了数据驱动的不足，在 dom操作 的时代通常都是这样操作：

> 通过特定的选择器查找到需要操作的节点 -> 给节点添加相应的事件监听

响应用户操作，效果是这样：

> 用户执行某事件（点击，输入，后退等等） -> 调用 JavaScript 来修改节点

这种模式对业务来说是没有什么问题，但是从开发成本和效率来说会比较力不从心，在业务系统越来越庞大的时候，就显得复杂了。另一方面，找节点和修改节点这件事，效率本身就很低，因此出现了数据驱动模式。

数据驱动

> 读取模板，同时获得数据，并建立 VM(view-model) 的抽象层 -> 在页面进行填充

要注意的是， `MVVM` 对应了三个层， `M - Model` ，可以简单的理解为数据层； `V - View` ，可以理解为视图，或者网页界面； `VM - ViewModel` ，一个抽象层，简单来说可以认为是 V 层中抽象出的数据对象，并且可以与 `V 和 M 双向互动` （一般实现是基于双向绑定，双向绑定的处理方式在不同框架中不尽相同）。

> 用户执行某个操作 -> 反馈到 VM 处理（可以导致 Model 变动） -> VM 层改变，通过绑定关系直接更新页面对应位置的数据

可以简单地理解：数据驱动不是操作节点的，而是通过虚拟的抽象数据层来直接更新页面。主要就是因为这一点，数据驱动框架才得以有较快的运行速度（因为不需要去折腾节点），并且可以应用到大型项目。

Vue 模式

Vue 通过 {{}} 绑定文本节点， data 里动态数据与 Props 静态数据进行一个映射关系，当 data 中的属性或者 props 中的属性有变动，以上两者里的每个数据都是行为操作需要的数据或者模板 view 需要渲染的数据，一旦其中一个属性发生变化，则所有关联的行为操作和数据渲染的模板上的数据同一时间进行同步变化，这种基于数据驱动的模式更简便于大型应用开发。只要合理的组织数据和代码，就不会显得后续皮软。

何为动态数据 data，何为静态数据 props

相同点

两者选项里都可以存放各种类型的数据，当行为操作改变时，所有行为操作所用到和模板所渲染的数据同时都会发生同步变化。

不同点

Data 被称之为动态数据的原因，在各自实例中，在任何情况下，我们都可以随意改变它的数据类型和数据结构，不会被任何环境所影响。

Props 被称之为静态数据的原因，在各自实例中，一旦在初始化被定义好类型时，基于 Vue 是单向数据流，在数据传递时始终不能改变它的数据类型。

更为关键地是，对数据单向流的理解， props 的数据都是通过父组件或者更高层级的组件数据或者字面量的方式进行传递的，不允许直接操作改变各自实例中的 props 数据，而是需要通过别的手段，改变传递源中的数据。

data 选项

当一个实例创建的时候， Vue 会将其响应系统的数据放在 data选项中 ， 当这些属性的值发生改变时，视图将会产生“响应”，即匹配更新为新的值。初始定行的行为代码也都会随着响应系统进行一个映射。

而 data 选项中的数据在实例中可以任意改变，不受任何影响，前提必须数据要跟逻辑相辅相成。

初始化映射

```
<template>
   <div>
      <p v-if='boolean'>true</p>
```

```
        <p v-for='value in obj'>{{value}}</p>
        <p v-for='item in list'>{{item}}</p>
        <p>{{StringMsg}}</p>
        <p>{{NumberMsg}}</p>
    </div>
</template>
<script>
    export default {
      data () {
        return  {
           obj : {a:'1',b:'2',c:'3'},
           list:['a','b','c'],
           boolean : true,
           StringMsg : 'hello vue',
           NumberMsg : 2.4,
        }
      }
    }
</script>
```

运行代码时，在 data选项 里定义了五种数据类型，通过指令和 {{}} 进行渲染，证实了 data选项 里可以 定义任何数据类型 。

视图与数据映射


```
<template>
    <div>
        <p>{{StringMsg}}</p>
        <p>{{NumberMsg}}</p>
        <button @click='changeData'>改变数据</button>
    </div>
</template>
<script>
    export default {
      data () {
        return  {
           StringMsg : 'hello vue',
           NumberMsg : 2.4
        }
      },
      methods: {
        changeData () {
          this.StringMsg = 2.4;
          this. NumberMsg = 'hello vue'
        }
      }
    }
</script>
```

每个.vue 的文件则就是一个实例，在 data 中定义了两种数据：

- String 类型
- Number 类型

同时还定义了一个 changeData 事件。

在运行代码时候， data选项 已经进入了 Vue的响应系统 里， model层 (数据层)与 view层 (视图层)进行了对应的映射，任何数据类型都可以定义。

当用户发生点击操作的时候，同时可以把 StringMsg，NumberMsg 的数据对调， 充分说明了， 无论值和类形都可以进行随意转换 。

行为与数据的映射


```
<template>
   <div>
      <p>{{StringMsg}}</p>
      <p>{{NumberMsg}}</p>
      <button @click='changeData'>改变数据</button>
      <button @click='findData'>查看数据</button>
   </div>
</template>
<script>
   export default {
     data () {
       return  {
          StringMsg : 'hello vue',
          NumberMsg : 2.4
       }
     },
     methods: {
       changeData () {
         this.StringMsg = 2.4;
         this.NumberMsg = 'hello vue'
       },
       findData () {
         console.log(`StringMsg: ${this.StringMsg}`)
         console.log(`NumberMsg: ${this.NumberMsg}`)
       }
     }
   }
</script>
```

改变数据以后，通过点击 findData 事件来进行验证，虽然在初始化定义好了行为数据的检测代码，但是当数据在执行 findData 之前先执行 changeData，一旦改变 data 选项里的数据时，findData 里对应的数据同时也会进行相应的映射。

this.StringMsg //=> 2.4

this.NumberMsg //=>’hello vue’

总结：
* data 选项里的数据是灵活的
* 可以定义任何数据类型
* 也可以改变成任何数据类型
* 当数据变化时，视图和行为绑定的数据都会同步改变

props

使用 props 传递数据作用域是孤立的，它是父组件通过模板传递而来，想接收到父组件传来的数据，需要通过 props选项 来进行接收。

子组件需要显示的声明接收父组件传递来的数据的 数量， 类型， 初始值 。

简单的接收可以通过数组的形式来进行接收。

父组件

```
<template>
   <div>
      <demo :msg='msgData' :math = 'mathData' ></demo>
   </div>
</template>
<script>
import Demo from './Demo.vue'
   export default {
     data () {
       return  {
          msgData:'从父组件接收来的数据',
          mathData : 2
       }
     },
     components : {
       Demo
     }
   }
</script>
```

子组件

```
<template>
  <div>
     <p>{{msg}}</p>
     <p>{{math}}</p>
  </div>
</template>

<script>
export default {
  name: 'demo',
  props: [ 'msg' , 'math'],
}
</script>
```

在子组件中需要通过显示定义好需要从父组件中接收那些数据。

同样的在父组件中在子组件模板中过 v-bind 来传递子组件中需要显示接收的数据。

语法：
： == v-bind(是封装的语法糖)： msg = msgData
* msg 第一个参数必须要与子组件的 props 同名
* msgData 则是父组件中需要向子组传递的数据

props 可以显示定义一个或一个以上的数据，对于接收的数据，可以是各种数据类型，同样也可以传递一个函数。

父组件

```
<template>
   <div>
      <demo :fn = 'myFunction'  ></demo>
   </div>
</template>
<script>
import Demo from './Demo.vue'
   export default {
     components : {
       Demo
     },
     methods: {
       myFunction () {
           console.log('vue')
       }
```

```
    }
  }
</script>
```

子组件

```
<template>
  <div>
     <button @click='fn'>按钮</button>
  </div>
</template>

<script>
export default {
  name: 'demo',
  props: [ 'fn' ],
}
</script>
```


同样，在父组件中也可以向子组件中传递一个 function，在子组件同样也可以执行父组件传递过来的 myFunction 这个函数。

字面量语法和动态语法

对于字面量语法和动态语法，初学者在父组件模板中向子组件中传递数据时加和不加 v-bind 有什么区别，同时会引起什么错语等问题会感觉迷惑。

在子组件模板上传递数据时加 v-bind 意味着什么？

`v-bind:msg = 'msg'` 通过 v-bind 进行传递数据并且传递的数据并不是一个字面量，双引号里的解析的是 一个表达式 ，同样也可以是实例上定义的数据和方法(其实就是引用一个变量）”。

`msg='11111'` 没有 v-bind 的模式下只能传递一个字面量，这个字面量只限于 String 类量，字符串类型。

注意：

虽然通过字面量模式下，传任何类型都会被转成字符串类型，但是在子件接收的时候可以通过 typeof 去进行类型检测。

字面量写法除了 String 类型

想通过字面量进行数据传递时，如果想传递 非String类型 ， 必须 props 名前要加上 v-bind ， 内部通过实例寻找，如果实例方没有此属性和方法， 则默认为对应的数据类型 。

```
:msg='11111' //number
:msg='true' //bootlean
:msg='()=>{console.log(1)} //function
:msg='{a:1} //object
```

子组件模板 props 定义问题

1.0版本时

HTML 特性是不区分大小写的，所以当使用的不是字符串模板，camelCased (驼峰式) 命名的 prop 需要转换为相对应的 kebab-case (短横线隔开式) 命名。

注意

由于文档上仍然有这句话，经过测试后，无论是不是字符串模板，camelCased (驼峰式) 和 kebab-case (短横线隔开式) 两者都可以。

建议

为了直观性，规范性还是推荐 kebab-case (短横线隔开式)。

对象传递简写

props 原子化可以让整体代码逻辑和向外暴露需要传递数据的接口非常清晰，但是同样可以把子组件需要接收的 props 在父组件中以一个对象进行传递。

当传递的数量一旦多到已经让原子化不再结构清晰的时候，通过一个对象传递显得更为简洁明了。

父组件

```
<template>
   <div>
      <demo v-bind= 'msg'  ></demo>
```

```
    </div>
  </template>
  <script>
  import Demo from './Demo.vue'
    export default {
      components : {
        Demo
      },
      data () {
        return {
          msg : {a:1,b:2}
        }
      }
    }
  </script>
```

子组件

```
  <template>
    <div>
      <button>按钮</button>
    </div>
  </template>

  <script>
  export default {
    name: 'demo',
    props: ['a','b'],
    created () {
      console.log(this.a)
      console.log(this.b)
    },
  }
  </script>
```


<demo v-bind= 'msg' ></demo> 内部发生了什么?

在子组件模板内部对其 进行了一个封装 ， 把其展开则跟 props 原子化原理是一个原理

```
<demo :a='a' :b='b' ></demo>
```

通常情况下建议使用第二种，props 原子化。

不可变的 props

在 data 选项中，当前实例（当前组件中改动）可以 任意改变data选项里的数据， Vue 传递数据时是基于 数据单向流动 ，子组件不能改变当前实例中的 props 任何属性，需要通知父组件改变相应的值，重新改变。

直接改变 props 数据

```
<template>
  <div>
     <button @click='changeProps'>按钮</button>
  </div>
</template>

<script>
export default {
  name: 'demo',
  props: ['msg'],
  methods: {
    changeProps () {
      this.msg = 'new msg'
    }
  }
}
</script>
```


直接改变 props 时会发生一个警告报错

> [Vue warn]: Avoid mutating a prop directly since the value will be overwritten whenever the parent component re-renders. Instead, use a data or computed property based on the prop's value. Prop being mutated: "msg"

防止数据的 不可控性 ，不能显示的直接改变，父组件的传递来的数据和子组接 props 接收的数据也是同步响应的，一旦父组件向下传递的数据改变时，prop 接收的数据值也会同样发生变化。

单向数据流的原因也是如此，就像河流一样，水只会从高向低流，想让水的质量改变，只有从源头改变。

父组件改动

```
<template>
   <div>
      <demo :msg = 'msg'  ></demo>
      <button @click='msg = "new vue"'>按钮</button>
```

```
    </div>
  </template>
  <script>
  import Demo from './Demo.vue'
    export default {
      components : {
        Demo
      },
      data () {
        return {
          msg : 'vue'
        }
      }
    }
  </script>
```

在父组件中初始化传递过后，想要改变子组件的数据，可以通过再次改变向子组件传递的 msg 数据，子组件渲染的视图同样会跟着同步改动。

一次性传递，过渡改动

虽然 props 是不可改动的，上面的 case 是父组件进行改变自身实例的的数据，这个实现很简单，有时经过一次数据传递，不需用父组件再次传递，因为一些需求需要改动 props 数据，可以用过渡的方法，让其转换为一个可变的数据。

过渡到 data 选项中

```
props: ['msg'],
data: function () {
  return { myMsg: this.msg }
}
```

在 data 选项里通过 myMsg 接收 props msg
数据，相当于对 myMsg = msg 进行一个赋值操作，不仅拿到了 myMsg 的数据，而且也可以改变 myMsg 数据。

this.myMsg = ‘new Vue’ myMsg 会发生相应的改变。

一次性传递，过滤处理

依然是通过 props 一次性接收，想对接收的 prop 进行一些过滤操作再次进行视图渲染，可以在一些计算属性中进行操作，可以 computed 监听 props 里的数据变化，经过过滤操

作返回一个需要的值。

```
props:['msg']
computed : {
    computedMsg () {
        return this.msg + 1
    }
}
```

注意：
在 JavaScript 中对象和数组是引用类型，指向同一个内存空间，如果 prop 是一个对象或数组，在子组件内部改变它会影响父组件的状态。不要对父组件传递来的引用类型数据进行过滤。

> 下篇导读

本章对 props 和 data 的用法理解已经进行了全面的讲解，通过再次改变传递数据时是在父组件的实例里进行实施的。往往特定的需求和一些组件封装触发传递的命令并不能直接在父组件执行，需要子组件通知上层组件。

再近一步说，子组件改变不了父组件传递的数据，但是子组件可以用通信的方式，通知子组件改动，因此 `$on`， `$emit`， v-on 深入理解这三者关系尤为重要！