# Vue 数据绑定、数据劫持、数据双向绑定实现原理

**Dep：订阅器**

1. 它的实例在什么时候创建？
   1. 初始化的时候对 data 属性进行数据劫持的时候创建。
2. 创建的实例个数是多少？
   1. 与 data 中的属性一一对应。
   2. 每次对 data 中的一个属性进行数据劫持的时候都会创建一个 Dep 实例，实例中保存着使用了该属性的 Watcher，也保存了一个通知 Watcher 的方法，该方法让那些使用了该属性的 Watcher 都进行数据更新。

**Watcher：订阅者**

1. 它的实例什么时候创建？
   1. 初始化的时候解析模板表达式和一般指令时创建。
2. 创建的实例个数是多少？
   1. 与模板表达式和一般指令一一对应。
   2. 每次解析模板表达式和一般指令时都会创建一个 Watcher 实例。
   3. 每次创建的这个新的 Watcher 实例都会去获取表达式的 value，间接出发了获取属性的 get 方法。通过这种方式建立 Watcher 和 Dep 之间的关系。

**Dep 与 Watcher 之间的关系：多对多的关系**

一个 data 属性 🡺 一个 Dep 🡺 n 个 Watcher （模板中有多个表达式用到了该属性的时候）

一个表达式 🡺 一个 Watcher 🡺 n 个 Dep （多层表达式，例 a.b.c）

**Dep 与 Watcher 之间的关系什么时候建立？**

在创建每一个 Watcher 的时候建立，通过调用 Watcher 的 get 方法。

**Watcher 和 Dep 如何相互存储对方 ？**

每个 Watcher 实例都有一个实例属性 depIds，该属性值是一个对象，格式为：

{ 0: Dep, 1: Dep }

每个 Dep 实例都有一个实例属性 subs，该属性值是一个数组，存放每一个 Watcher，格式为：[ Watcher, Watcher, Watcher ]

初始化

1. 当创建一个 MVVM 实例的时候，将 data 数据保存在了 \_data 中，并对 data 实现数据代理，即将 data 中的所有属性放到 vm 对象上，这样通过 vm.xxx 即可访问到 data.xxx 的数据。具体实现是调用了 MVVM 原型上的 \_proxy 函数，通过 Object.defineProperty 实现。
2. 然后对 data 中的数据开启 Observer 监视，只有在 data 为对象的时候创建Observer 实例
   1. Observer 实例创建后就执行了 Observer 构造函数中的初始化代码，执行了 walk 函数，该函数遍历了 data 中的所有属性，并最后调用 defineReactive函数以实现响应式数据绑定。
   2. 在defineReactive 函数的首行创建一个 Dep 订阅器实例，因此进入 Dep 构造函数。
      1. 在该 Dep 实例上有两个实例属性，id 和 subs，id 用于标识，subs 用于存放 Watcher，首个 Dep 实例的 id 为 0，往后每个 Dep 实例的 id 都会 ++，subs 初始为空数组。（注：Dep 有一个静态属性 target，初始为 null）
   3. Dep 实例的初始化代码运行完成后回到 defineReactive 函数，运行数据绑定的核心代码，通过 Object.defineProperty 实现，其中在属性的 get 方法中建立 Dep 和 Watcher 的关系。在属性的 set 方法中判断该属性值是否发生了变化，如果发生了变化则通知 Dep 订阅器，执行 Dep 原型身上的 notify 方法。
3. Observer 实例的初始化代码执行完毕之后，创建 Compile 对象的实例
   1. Compile 的实例初始化代码主要做的事情对传入的el 控制的区域进行编译，对该区域中的指令、模板表达式中的数据渲染出来。
      1. 渲染是通过 DocumentFragment 实现，首先将 el 中的所有子节点都拷贝到一个 fragment 身上
      2. 然后对这个 fragment 进行遍历，判断每一个元素节点身上是否有指令，文本节点中是否有模板表达式，有的话均执行相应的方法，最终通过 updater 更新数据。那如果这个节点还有子节点的话就会进行递归调用，这样就实现了 el 区域中所有节点的遍历。
      3. 遍历过程中遇到带指令的会判断该指令是普通指令还是事件指令，事件指令则直接绑定对应的事件，普通指令则均又调用了一个 bind 函数。
      4. 在 bind 函数中调用了相应指令对应的 updater 方法去实现数据更新。但又同时创建了一个 Watcher 实例，因此执行 Wactch 实例的初始化代码。
         1. 在每一个 Watcher 实例身上都有几个实例属性，其中关键的有：
            1. This.cb = cb 保存回调函数
            2. This.exp = exp 保存表达式（例：wife.name.high）
            3. This.depIds = {} 保存 Dep 实例
            4. This.value = this.get() 保存旧的属性值，跟新值做对比使用
         2. 在运行到 this.val = this.get() 时，调用了 get 方法。
            1. 在 get 方法中首先将 Dep.target 设置为 this，也就是Watcher
            2. 然后调用 Watcher 原型身上的 getVMVal 方法，获取其 this.exp 保存的字符串对应的值，需要去 vm 实例身上的 \_data 中获得，当去取 \_data 中的数据的时候实际上调用了该属性的 get 方法，而在这个 get 方法中建立了 Dep 和 Watcher 的关系。

在这个获取属性值的 get 方法中首先判断了 Dep.target，由于之前已经设置成了 this，也就是 Watcher，因此执行 Dep 实例的 depend 方法。

该 depend 方法使用了 Dep.target.addDep(this)，由于 Dep.target 是 Watcher，因此调用了 Watcher 的 addDep 方法，并传入 Dep 对象。

addDep 方法中调用了 dep 的 addSub 方法，将 Watcher 对象保存在 dep 的 subs 数组中，并将该 dep 的信息也保存在了 Watcher 的depIds 对象上。

depend 方法调用结束，返回属性值。

* + - * 1. 最终将获取到的值存在 this.value 身上

更新数据

1. 当 data 中数据发生改变时，会调用属性的 set 方法，在该方法中调用 dep.notify()，遍历所有订阅者。
2. 在 notify 方法中通过遍历自身的 subs 属性调用所有订阅者 Watcher 的 update 方法
3. 在 update 中最终执行了 Wather 的回调函数，实现数据更新。

数据双向绑定实现

1. 在 Compile 初始化遍历过程中，遇到 v-model 指令时，会执行对应的 model 方法，该方法同样调用了 bind 方法实现数据的初始化更新和创建了 Watcher。
2. 在 bind 方法执行完后为该元素节点添加 input 事件监听，不用 change 事件是因为 change 事件在离开焦点的时候出发。
3. 在该事件监听处理函数中为 data 中相应的属性设置成新变化的值，这样就会调用属性的 set 方法
4. Set 方法然后通知订阅器 Dep，然后在通知每个订阅者 Watcher 进行数据更新

画图示例

