数据响应式处理过程

- initState()=>initData()=>observe()
  1. 从Vue构造函数开始找到_data处理的开始src/core/instance/index.js中的this._init()
  2. this._init()在src/core/instance/init.js中的initMixin()进行初始化，initMixin()中调用initState初始化vm 的 _props/methods/_data/computed/watch，初始化实例的状态
  3. 在initState方法中调用initData()对_data进行初始化，将data属性注入到vue实例上
  4. 初始化 _data，组件中 data 是函数，调用函数返回结果
  5. 调用observe对data进行响应式处理
- observe(value)
  - 位置：src/core/observe/index.js
  - 功能：
    - 判断value是否是对象，如果不是对象直接返回
    - 判断value对象是否有__ob__，如果有直接返回
    - 如果没有，创建observe对象
    - 返回observe对象
- Observer类
  - 位置：src/core/observe/index.js
  - 功能：
    - 给value对象定义不可枚举的__ob__属性，记录当前的observer对象，def(value, '__ob__', this)
    - 数组的响应式处理
      1. 通过方法 this.observeArray(value)为数组中的每一个对象创建一个 observer 实例
      2. 通过arrayMethods补充数组的 'push','pop','shift','unshift','splice','sort', 'reverse'方法
    - 对象的响应式处理，调用walk方法。遍历每一个属性，设置为响应式数据，调用defineReactive()

      ```
      遍历每一个属性，设置为响应式数据，调用defineReactive()
        for (let i = 0; i < keys.length; i++) {
          defineReactive(obj, keys[i])
        }
      ```
- defineReactive
  - 位置：src/core/observe/index.js
  - 功能：
    - 为每一个属性创建dep依赖对象实例

      ```
      const dep = new Dep()
      ```
    - 如果当前属性的值是对象，调用observe

      ```
      判断是否递归观察子对象，并将子对象属性都转换成 getter/setter，返回子观察对象
      let childOb = !shallow && observe(val)
      ```

- 定义getter
  - 收集依赖

```
// 如果存在当前依赖目标，即 watcher 对象，则建立依赖
if (Dep.target) {
  dep.depend()
  // 如果子观察目标存在，建立子对象的依赖关系
  if (childOb) {
    childOb.dep.depend()
    // 如果属性是数组，则特殊处理收集数组对象依赖
    if (Array.isArray(value)) {
      dependArray(value)
    }
  }
}
```

  - 返回属性的值
- 定义setter
  - 保存新值
  - 如果新值是对象，调用observe

```
// 如果预定义的 getter 存在则 value 等于getter 调用的返回值，否则直接赋予属性值
    const value = getter ? getter.call(obj) : val
// 如果新值等于旧值或者新值旧值为NaN则不执行
    if (newVal === value || (newVal !== newVal && value !== value)) {
      return
    }
// 如果没有 setter 直接返回
    if (getter && !setter) return
// 如果预定义setter存在则调用，否则直接更新新值
    if (setter) {
      setter.call(obj, newVal)
    } else {
      val = newVal
    }
    // 如果新值是对象，观察子对象并返回 子的 observer 对象
    childOb = !shallow && observe(newVal)
```

  - 派发更新（发送通知），调用dep.notify（）

```
    // 派发更新(发布更改通知)
    dep.notify()
```

- 收集依赖

- 在Watcher对象的get方法中调用pushTarget，记录Dep.target属性

```
src/core/observer/watcher.js
pushTarget(this)

src/core/observer/dep.js
// Dep.target 用来存放目前正在使用的watcher
// 全局唯一，并且一次也只能有一个watcher被使用
Dep.target = null
const targetStack = []
// 入栈并将当前 watcher 赋值给 Dep.target
// 父子组件嵌套的时候先把父组件对应的 watcher 入栈，
// 再去处理子组件的 watcher，子组件的处理完毕后，再把父组件对应的 watcher 出栈，继续操作
export function pushTarget (target: ?Watcher) {
  targetStack.push(target)
  Dep.target = target
}
```

- 访问data中的成员的时候收集依赖，访问data成员会触发defineReactive的getter中收集依赖

```
dep.depend()=>

// 将观察对象和 watcher 建立依赖
  depend () {
    if (Dep.target) {
      // 如果 target 存在，把 dep 对象添加到 watcher 的依赖中
      Dep.target.addDep(this)
    }
  }
```

- 把属性对应的watcher对象添加到dep的subs数组中，为属性收集依赖
- 如果该属性对应的是一个对象，创建childOb对象，目的是子对象添加和删除成员时发送通知

- 数据发生变化Watcher的整个过程

- 会在setter属性中触发dep.notify()，调用watcher对象的update方法

```
// 发布通知
notify () {
  const subs = this.subs.slice()
  if (process.env.NODE_ENV !== 'production' && !config.async) {
    subs.sort((a, b) => a.id - b.id)
  }
  // 调用每个订阅者的update方法实现更新
  for (let i = 0, l = subs.length; i < l; i++) {
    subs[i].update()
  }
}
```

```
src/core/observer/watcher.js
update () {
  if (this.lazy) {
    this.dirty = true
  } else if (this.sync) {
    this.run()
  } else {
    queueWatcher(this)
  }
}
```

- queueWatcher()判断watcher是否被处理，如果没有的话添加到queue队列中，并调用flushScheduleQueue()
- flushScheduleQueue()
  - queue.sort((a, b) => a.id - b.id)
  - 触发beforeUpdate钩子函数
  - 调用wacher.run()
    - run()=>get()=>getter()=>updateComponent更新视图
  - 清空上一次的依赖，重置watcher中的一些状态（出栈操作）
  - 触发actived钩子函数
  - 触发update钩子函数