Vue源码解析

Vue采用数据劫持配合发布者-订阅者模式的方式，通过Object.defineProperty来配置getter和setter来劫持数据的变化，在数据变动时，发布消息给依赖收集器，去通知观察者做出对应的回调函数，去更新视图。

MVVM 作为绑定的入口，整合Observer,Compile和Watcher三者，通过Observer来监听model数据变化，通过Compile来解析编译模板指令，最终利用Watcher搭起Observer,Compile之间的通信桥梁，达到数据数据变化 => 视图更新，视图的交互变化 => 数据model变更的双向绑定效果

# 观察者-订阅者部分

Observer 
•constructor(data) 
+0bserver(data) 
key, value) 
Dep 
* subsO 
•constructor() 
•addSub(watcher) 
•notity() 
Watcher 
•oldVal 
•constructor(vm, expr, cb) 
•update() 
•getOldVal() 

## 数据观察者

observer: 对数据进行循环遍历调用defineReactive方法

defineReactive: 对数据通过Object.defineProperty进行配置

* 在getter方法中添加订阅者
* 在setter方法调用Dep.notify通知订阅者

## 收集依赖器

添加观察者到依赖器和发布数据变化

## 订阅者

数据发生变化时，更新视图

class Watcher { 
cb) { 
this.vm = vm; 
this.expr = expr; 
this.cb = cb; 
= this.get01dVa1(); 
this . oldVa1 
get01dVa1() { 
Dep. target = this; 
// getfiii 
— compileUti1.getVa1(this .expr,this .vm); 
const oldVa1 — 
Dep. target = null; 
return oldVa1; 
update() { 
const newVa1 = compileUti1.getVa1(this this . vm) ; 
if(newVa1 this .01dVa1) { 
this . cb(newVa1) ; 

class Dep { 
constructor(){ 
this. subs — 
addSub(watcher) { 
this . subs . push(watcher) ; 
notify() { 
console. log( ' , this. subs) ; 
this . subs . forEach(w 
w.update()); 

class Observer { 
constructor(data) { 
this . observe(data); 
observe(data) { 
if(data typeof data 
'object') { 
Object. keys(data) . forEach(key { 
this . defineReactive(data, key , data [ key] ) ; 
console. log(data) ; 
// valuefrYSJdefineReativelAJff•J+.GJQfit 
defineReactive(obj, key, value) 
this . observe(value) ; 
const dep = new Dep(); 

Object. defineProperty (obj , key , { 
emumerable: true, 
configurable: false, 
get: ( ) 
// illkJ%iE'ÆfEHj, 
Dep. target dep. addSub(Dep.target); 
return value; 
set: (newVa1) => { 
this . observe(newVa1) 
if(newVa1 value) { 
value = newVa1; 
dep. notify(); 

# 编译部分

通过使用Fragment进行简单的测试，并没有实现虚拟节点转化到编译

* 节点转换 DOM Node --> Fragment

const f = document. createDocumentFragment(); 
let firstChi1d; 
while (firstChi1d = el. firstChi1d) 
f. appendChi1d(firstChi1d) ; 

* 解析指令 v-directive --> directive

const attributes = node. attributes; 
[ attributes] . forEach(attr { 
const { 
name, 
value 
} — attr; 
if (this.isDirective(name)) { 
const [ , directive] - 
split( ' 
- ' ) ; any 
- name. 
const [dirName, eventName] = directive. split( ' : ' 
compileUti1[di rName] ( node, 
value, this . vm, eventName) ; 
node. removeAttribute( 'v-' + directive); 

* 指令封装 getDirective --> parseDirective --> removeDirective

html (node, expr, vm) { 
const value = this. getVa1(expr, vm); 
new Watcher(vm, expr, (newVa1) { 
this . updater. htmlUpdater(node, newVa1); 
this . updater. htmlUpdater(node, value) 

* 数据的双向绑定 监听input时间更新数据

model(node, expr, vm) { 
const value = this .getVa1(expr, vm); 
new Watcher(vm, expr, (newVa1) { 
this . updater. modelUpdater(node, newVa1); 
node.addEventListene ('input', e { 
this. setVa1(expr, vm, e. target . value); 
this . updater. modelUpdater(node, value); 

* 数据代理 this.$data.message --> this.message

proxyData(data) { 
for(const key in data) { 
Object. defineproperty(this, key, { 
get() { 
return datal key]; 
set (newVa1) { 
data [key] 
= newVa1; 