****

预期的效果：

监听input的输入，input在输入的时候，会触发 watch与computed函数，并且会更新原始的input的数值。所以直接跟input相关的处理就有3处，但实际上会有连带性的触发，触发watch的input函数的时候，还会触发this.answer对应的依赖处理

看看内部是如何处理的：

Vue在初始化data的时候，会通过Object.defineProperty重新定义input的set与get访问接口，同时会创建一个记录并且保持其数据对应的依赖watcher对象的Dep对象。

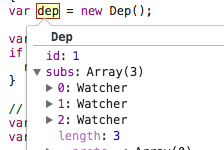
简单来说

1. 在data中定义了input，那么意味着需要对这个变量进行defineProperty的处理，并创建Dep对象

2．watch中的input函数会变成一个Watcher对象，因为它与input有关系，所以需要在data的input的Dep中保存一份引用

3．computed中的compiledMarkdown函数会变成一个Watcher对象，，因为它与input有关系，所以需要在data的input的Dep中保存一份引用

每个input都是通过闭包处理的，所以Dep会一直是存在的内部的，所以input数据的监控内部创建的Dep的结构就是如下，

，

watch与computed明明只有2个对应的Watcher对象，为什么subs会有3个呢？多增加的一个是干什么的？看后面

所以最终可以简单的梳理下更新的流程：当input数据发生变化的时候，只需要调用响应依赖的Watcher对象，Watcher对象就会负责各自的更新处理。这里面向对象的设计优势就体现出来了，将行为分布在各个对象中，并让这些对象负责自己的行为，所以每个不同Watcher对象更新各自的特点，处理各自的逻辑

更新逻辑：

vue1的 dom更新方式采用队列+直接更新的处理，这种简单粗暴。vue2在vue1的设计上，继续保留了队列的处理方式，同时结合了时下最流行的 virtual dom

更新原理：

记得在Vue1中，每个Watcher对象都会保存各自的dom节点的处理方式，通过对Watcher的的处理达到直接更新DOM的目的。Vue2因为引入的Virtual Dom的机制，所以Watcher的工作就需要变化了，大多数的Watcher不再直接负责DOM的更新操作，而只是更新数据。这里用了大多数，因为还有一个Watcher是跟Virtual Dom相关的。所以这就是在上文提到的Dep中会多一个Watcher的原因了

Virtual DOM

虚拟DOM的文章现在已经很多了，但是如何紧密结合vue中，到实际的运用是我们分析的重点，这里只是粗略下，我还要抽时间把算法看完先

原理：

简单的说，直接通过JS操作浏览器API去绘制DOM节点是很慢的，大量的页面处理中，开发者不经意就会调用更多多余或者重复的操作，这种是有性能开销的。那么有什么办法减少这种是误操作呢？就是通过一种方式能算出来最小的更新量，从而提高效率。既然要计算出对小的更新量，那么就会有对比，需要通过对新旧两个节点的对比从而计算出。DOM的操作很慢，但是JS确很快的，DOM 树上的结构、属性信息我们都可以很容易地用 JavaScript 对象表示出来，既然我们可以用JS对象表示DOM结构，那么当数据状态发生变化而需要改变DOM结构时，我们先通过JS对象表示的虚拟DOM计算出实际DOM需要做的最小变动，反过来，就可以根据这个用 JavaScript 对象表示的树结构来构建一棵真正的DOM树，操作实际DOM更新了， 从而避免了粗放式的DOM操作带来的性能问题。

根据上面的原理，Virtual DOM在实现上首先就必须先建立可以对比的JS对象，这个叫做vnode，也就是虚拟DOM了，这个对象是真实DOM结构的一个映射，通过对比更新前后vnode的变化差异diff，记录下来的不同就是我们需要对页面真正的 DOM 操作。

简单总结下，Virtual DOM 算法。包括几个步骤：

1. 用JS对象描述出DOM树的结构，然后在初始化构建中，用这个描述树去构建真正的DOM，并实际展现到页面中
2. 当有数据状态变更时，重新构建一个新的JS的DOM树，通过新旧对比DOM数的变化diff，并记录两棵树差异
3. 把步骤2中对应的差异通过步骤1重新构建真正的DOM，并重新渲染到页面中，这样整个虚拟DOM的操作就完成了，视图也就更新了

看到这里可以简单总结下，Watcher与Virtual DOM的关系，：

1. Watcher 是来决定你要不要更新这个dom
2. 虚拟DOM是用来找出怎么以最小的代价来更新

Vue2中对应的逻辑：

这里不会涉及算法，并非这章的重点，主要看下整个更新过程中，虚拟DOM逻辑是怎么配合工作的。

继续input的数据流向，之前讲到了input中的Dep是保存了3个Watcher对象的引用，其中会有一个Watcher是跟整个页面的渲染有关系的，这个就是用来封装vnode的处理。

当遍历Dep这个保存Watcher数组的时候，会把Watcher加入到一个异步的队列中进行处理

代码进行了简化

function queueWatcher(watcher) {

var id = watcher.id;

if (has[id] == null) {

has[id] = true;

queue.push(watcher);

nextTick(flushSchedulerQueue);

}

}

function flushSchedulerQueue() {

queue.sort(function(a, b) { return a.id - b.id; });

for (index = 0; index < queue.length; index++) {

watcher = queue[index];

id = watcher.id;

has[id] = null;

watcher.run();

}

}

这里很关键的一个点就是针对queue进行了排序，原因就是其中有一个Wacher是保存了vnode了，因为最后一步才是vnode的对比更新。必须让前面的Watcher更新数据完毕后，最后vnode才能做真正的对比，不过computed的Wacher不会加入到这个队列中，它会再编译树中动态的执行。

啪啦啪啦，当前面的Watcher执行完毕后，调到最后一个Watcher，可以看到对应的代码

vm.\_update(vm.\_render(), hydrating);

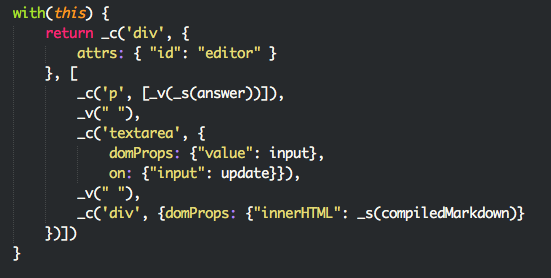
1. 通过vm.\_render方法构建vnode
2. 通过vm.\_update 对比vnode，并渲染到页面中

vm.\_render

初始化的时，会通过构建出来的JS描述树，生成初始vnode，去绘制初始页面。每次DOM变化的时候，我们还是需要重新构建这个描述树，通过这个描述树去构建新的vnode

这个描述树生成相当复杂，vue2内部专门会有一个AST是干这个事的

对应的结构是这样的，这个可以其实就是真实DOM树的一个结构映射了：



但是这个结构是可执行的，可编译的，通过with的方式改变this的上下文，动态执行每个可执行的代码部分，并把每个节点部分都编译成vnode，组成一个有对应层次结构的vnode对象

举例来说

div是最外层的vnode

div有子节点=> p，生成对应vnode

p有子节点=>文本节点answer，生成对应vnode

每个vnode会保存每个对应节点一些计算信息，比如tag、data、 children、text这些都是用于后面的比对计算的

vm.\_update

通过render拿到了vnode，然后通过update对比vnode绘制到页面

update这个方法内部有段代码

vm.$el = vm.\_\_patch\_\_(prevVnode, vnode);

从这个字面意思就明显知道，更新补丁，用于对比新旧2个vnode，

vue2有个专门的patch文件用于vnode的对比策略，patch内部会细分很多策略出来

1. 如果vnode不存在但是oldVnode存在，就意味着要销毁
2. 如果oldVnode不存在但是vnode存在，说明意图是要创建新节点
3. 当vnode和oldVnode都存在时，就需要更新了

每一种策略都对应的不同的处理方式，更新才意味着需要对比新旧的vnode，首先是需要判断下两个节点是否值得比较，在这个例子里面只改变了属性input与answer的值，所以，这里是属于同节点内的属性变更的，所以检测vnode的变化也是相对最简单，递归子节点，通过patchVnode检测每个节点属性的变化

if(sameVnode(oldStartVnode, newStartVnode)) {

patchVnode(oldStartVnode, newStartVnode, insertedVnodeQueue);

oldStartVnode = oldCh[++oldStartIdx];

newStartVnode = newCh[++newStartIdx];

}

当对比到差异时，例如文本answer被改变，那么对应的vnode在对比的时候，就能找到差异，然后重新设置值，此刻的node就是真实的DOM引用的，如果改变了textContent就意味着页面上呈现的数据就直接被改变了

if (oldVnode.text !== vnode.text) {

nodeOps.setTextContent(elm, vnode.text);

}

function setTextContent (node, text) {

node.textContent = text;

}

因为不是具体的算法分析，所以不会一段代码一段代码的去句斟字酌了，整段分析都是基于这个简单的代码，所以在实现上很多地方是有偏差的，不能以偏概全，但是通过这个文章，相对你对vue2的内部逻辑应该是有一个初步的认识