Vue响应式原理

> 昨天看完后面这部分太晚了，后面写的有点睁不开眼了。今天就再总结下响应式原理，如果明天下班早会继续总结下逼格也很高的 `vdom`

先撸为敬

```
const Observer = function(data) {
  for (let key in data) {
    defineReactive(data, key);
  }
}

const defineReactive = function(obj, key) {
  const dep = new Dep();
  let val = obj[key];
  Object.defineProperty(obj, key, {
    enumerable: true,
    configurable: true,
    get() {
      console.log('in get');
      dep.depend();
      return val;
    },
    set(newVal) {
      if (newVal === val) {
        return;
      }
      val = newVal;
      dep.notify();
    }
  });
}

const observe = function(data) {
  return new Observer(data);
}

const Vue = function(options) {
  const self = this;
  if (options && typeof options.data === 'function') {
    this._data = options.data.apply(this);
  }

```

```
  this.mount = function() {
    new Watcher(self, self.render);
  }

  this.render = function() {
    with(self) {
      _data.text;
    }
  }

  observe(this._data);
}

const Watcher = function(vm, fn) {
  const self = this;
  this.vm = vm;
  Dep.target = this;

  this.addDep = function(dep) {
    dep.addSub(self);
  }

  this.update = function() {
    console.log('in watcher update');
    fn();
  }

  this.value = fn();
  Dep.target = null;
}

const Dep = function() {
  const self = this;
  this.target = null;
  this.subs = [];
  this.depend = function() {
    if (Dep.target) {
      Dep.target.addDep(self);
    }
  }

  this.addSub = function(watcher) {
    self.subs.push(watcher);
  }

  this.notify = function() {
    for (let i = 0; i < self.subs.length; i += 1) {
      self.subs[i].update();
```

```
    }
  }
}

const vue = new Vue({
  data() {
    return {
      text: 'hello world'
    };
  }
})

vue.mount(); // in get
vue._data.text = '123'; // in watcher update /n in get
```

这里我们用不到100行的代码，实现了一个简易的vue响应式。当然，这里如果不考虑期间的过程，我相信，40行代码之内可以搞定。但是我这里不想省略，为什么呢？我怕你把其中的过程自动忽略掉，怕别人问你相关东西的时候，明明自己看过了，却被怼的哑口无言。总之，我是为了你好，多喝热水。

Dep的作用是什么？

> 依赖收集器，这不是官方的名字蛤，我自己瞎B起的，为了好记。

用两个例子来看看依赖收集器的作用吧。

- 例子1，无所吊味的渲染是不是没必要？

  ```
  const vm = new Vue({
      data() {
          return {
              text: 'hello world',
              text2: 'hey',
          }
      }
  })
  ```

  当 `vm.text2` 的值发生变化时，会再次调用 `render`，而 `template` 中却没有使用 `text2`，所以这里处理 `render` 是不是毫无意义？

  针对这个例子还记得我们上面模拟实现的没，在 `Vue` 的 `render` 函数中，我们调用了本次渲染相关的值，所以，与渲染无关的值，并不会触发 `get`，也就不会在依赖收集器中添加到监听(`addSub` 方法不会触发)，即使调用 `set` 赋值， `notify` 中的 `subs` 也是空的。OK，继续回归demo，来一小波测试去印证下我说的吧。

```
const vue = new Vue({
  data() {
    return {
      text: 'hello world',
      text2: 'hey'
    };
  }
})

vue.mount(); // in get
vue._data.text = '456'; // nothing
vue._data.text2 = '123'; // in watcher update /n in get
```

- 例子2，多个Vue实例引用同一个data时，通知谁？是不是应该俩都通知？

```

let commonData = {
  text: 'hello world'
};

const vm1 = new Vue({
  data() {
    return commonData;
  }
})

const vm2 = new Vue({
  data() {
    return commonData;
  }
})

vm1.mount(); // in get
vm2.mount(); // in get
commonData.text = 'hey' // 输出了两次 in watcher update /n in get
```

老规矩，自己代入进去试试。

希望通过这两个例子，你已经大概清楚了 `Dep` 的作用，有没有原来就那么回事的感觉？有就对了。总结一下吧(以下依赖收集器实为 `Dep`)：

- `vue` 将 `data` 初始化为一个 `Observer` 并对对象中的每个值，重写了其中的 `get` 、 `set` 、 `data` 中的每个 `key` ，都有一个独立的依赖收集器。
- 在 `get` 中，向依赖收集器添加了监听
- 在mount时，实例了一个 `Wathcer` ，将收集器的目标指向了当前 `Watcher`

- 在 `data` 值发生变更时，触发 `set` ，触发了依赖收集器中的所有监听的更新，来触发 `Watcher.update`

OK，今天的内容就到这里吧，好梦。