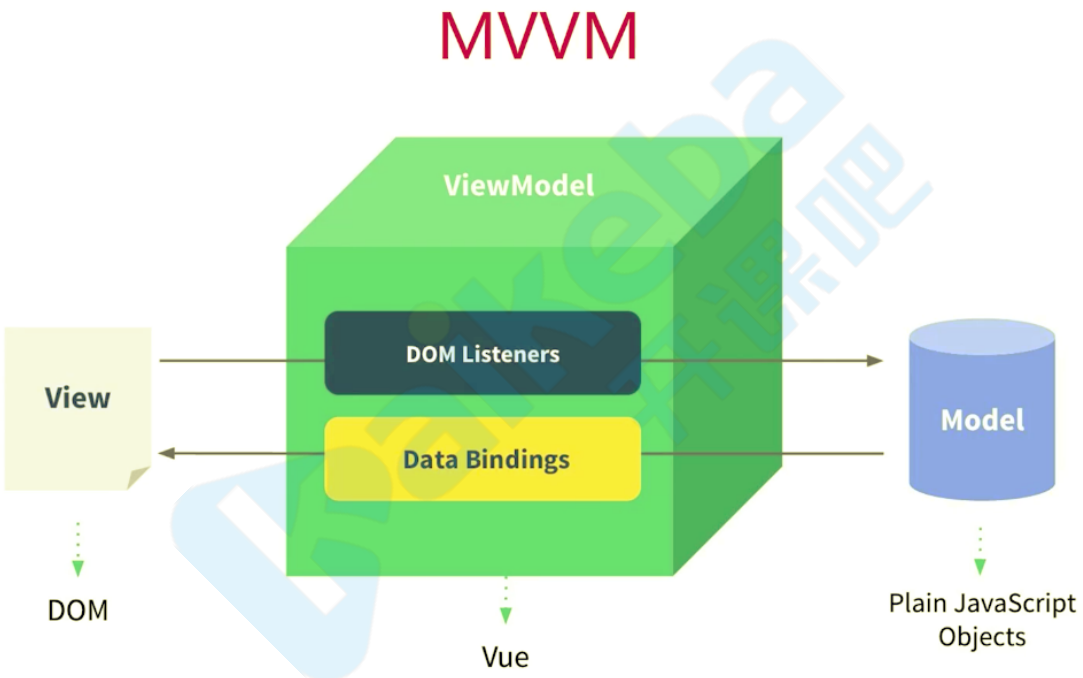
手写Vue

复习

https://www.processon.com/view/link/5e146d6be4b0da16bb15aa2a

理解Vue的设计思想

- MVVM模式



MVVM框架的三要素：**数据响应式、模板引擎及其渲染**

数据响应式：监听数据变化并在视图中更新

- Object.defineProperty()
- Proxy

模版引擎：提供描述视图的模版语法

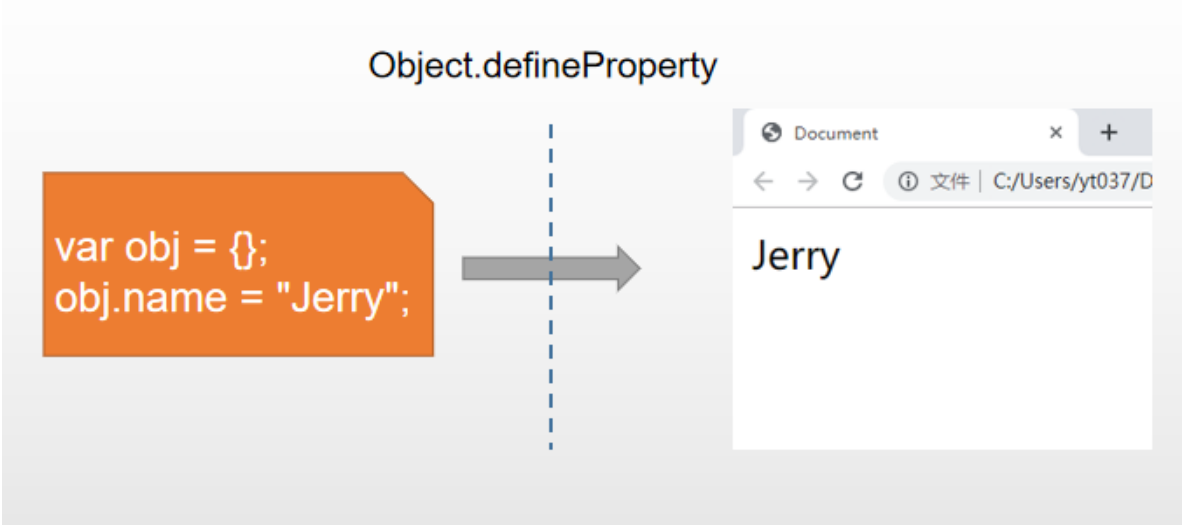
- 插值：{{}}
- 指令：v-bind，v-on，v-model，v-for，v-if

渲染：如何将模板转换为html

- 模板 => vdom => dom

数据响应式原理

数据变更能够响应在视图中，就是数据响应式。vue2中利用`Object.defineProperty()`实现变更检测。



简单实现

```
const obj = {}

function defineReactive(obj, key, val) {
  Object.defineProperty(obj, key, {
    get() {
      console.log(`get ${key}:${val}`);
      return val
    },
    set(newVal) {
      if (newVal !== val) {
        console.log(`set ${key}:${newVal}`);
        val = newVal
      }
    }
  })
}

defineReactive(obj, 'foo', 'foo')
obj.foo
obj.foo = 'fooooooooooo'
```

结合视图

```
<!DOCTYPE html>
<html lang="en">
<head></head>
<body>
  <div id="app"></div>
  <script>
    const obj = {}

    function defineReactive(obj, key, val) {
      Object.defineProperty(obj, key, {
        get() {
          console.log(`get ${key}:${val}`);
```

```
          return val
        },
        set(newVal) {
          if (newVal !== val) {
            val = newVal
            update()
          }
        }
      })
    }

    defineReactive(obj, 'foo', '')
    obj.foo = new Date().toLocaleTimeString()

与视图结合的函数 function update() {
      app.innerText = obj.foo
    }

    setInterval(() => {
      obj.foo = new Date().toLocaleTimeString()
    }, 1000);
  </script>
</body>
</html>
```

遍历需要响应化的对象

```
// 对象响应化：遍历每个key，定义getter、setter
function observe(obj) {
  if (typeof obj !== 'object' || obj == null) {
    return
  }
  Object.keys(obj).forEach(key => {
    defineReactive(obj, key, obj[key])
  })
}


const obj = {foo:'foo',bar:'bar',baz:{a:1}}

observe(obj)
obj.foo
obj.foo = 'fooooooooooo'
obj.bar
obj.bar = 'barrrrrrrrrr'
obj.baz.a = 10 // 嵌套对象no ok
```

解决嵌套对象问题

```
function defineReactive(obj, key, val) {
    observe(val)
    Object.defineProperty(obj, key, {
    //...
```

解决赋的值是对象的情况

```
obj.baz = {a:1}
obj.baz.a = 10 // no ok
```

```
set(newVal) {
    if (newVal !== val) {
        observe(newVal) // 新值是对象的情况
        notifyUpdate()
```

如果添加/删除了新属性无法检测

```
obj.dong = 'dong'
obj.dong // 并没有get信息
```

```
function set(obj, key, val) {
  defineReactive(obj, key, val)
}
```

测试

```
set(obj, 'dong', 'dong')
obj.dong
```

`defineProperty()`不支持数组

这里留个作业：解决数组数据的响应化

Vue中的数据响应化

目标代码

kvue.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <meta http-equiv="X-UA-Compatible" content="ie=edge">
  <title>Document</title>
</head>
<body>
  <div id="app">
    <p>{{counter}}</p>
  </div>
```

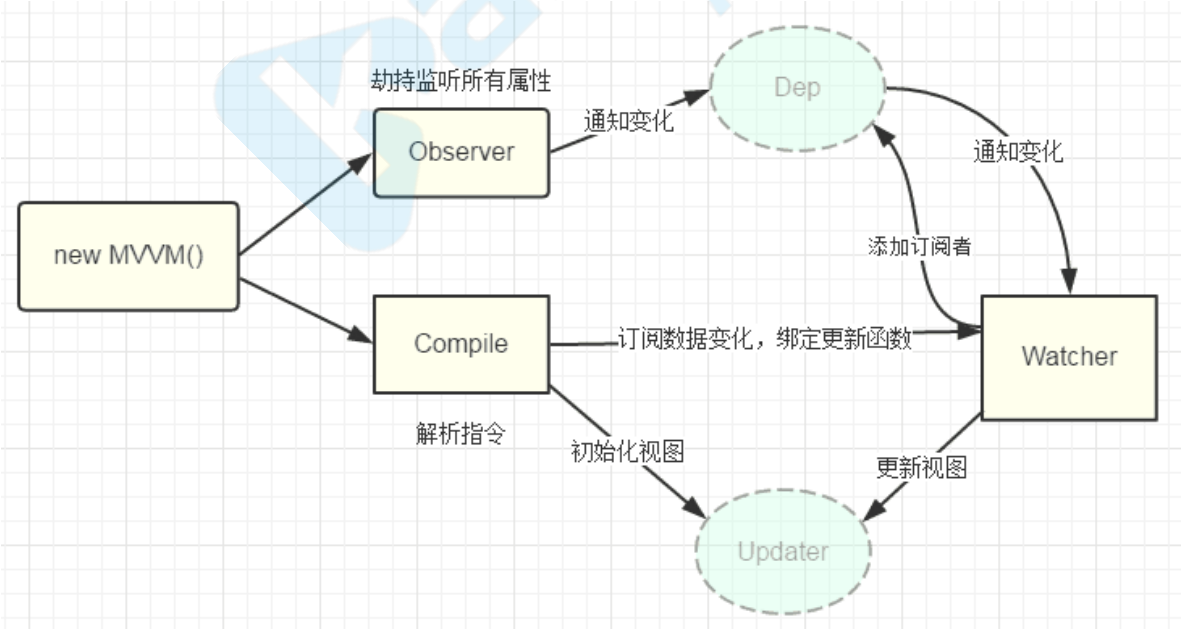
```
  <script src="node_modules/vue/dist/vue.js"></script>

  <script>
    const app = new Vue({
      el:'#app',
      data: {
        counter: 1
      },
    })
    setInterval(() => {
      app.counter++
    }, 1000);

  </script>
</body>
</html>
```

原理分析

1. `new Vue()`首先执行初始化，对data执行响应化处理，这个过程发生在Observer中
2. 同时对模板执行编译，找到其中动态绑定的数据，从data中获取并初始化视图，这个过程发生在Compile中
3. 同时定义一个更新函数和Watcher，将来对应数据变化时Watcher会调用更新函数
4. 由于data的某个key在一个视图中可能出现多次，所以每个key都需要一个管家Dep来管理多个Watcher
5. 将来data中数据一旦发生变化，会首先找到对应的Dep，通知所有Watcher执行更新函数



涉及类型介绍

- KVue：框架构造函数
- Observer：执行数据响应化（分辨数据是对象还是数组）
- Compile：编译模板，初始化视图，收集依赖（更新函数、watcher创建）
- Watcher：执行更新函数（更新dom）
- Dep：管理多个Watcher，批量更新

KVue

框架构造函数：执行初始化

- 执行初始化，对data执行响应化处理，kvue.js

```
function observe(obj) {
  if (typeof obj !== 'object' || obj == null) {
    return
  }

  new Observer(obj)
}

function defineReactive(obj, key, val) {}

class KVue {
  constructor(options) {
    this.$options = options;
    this.$data = options.data;
    进行data的响应化操作
    observe(this.$data)
  }
}

进行数据响应化 class Observer {
  constructor(value) {
    this.value = value
    this.walk(value);
  }

  walk(obj) {
    Object.keys(obj).forEach(key => {
      defineReactive(obj, key, obj[key])
    })
  }
}
```

- 为$data做代理　　代理函数用于方便用户直接访问$data中的数据

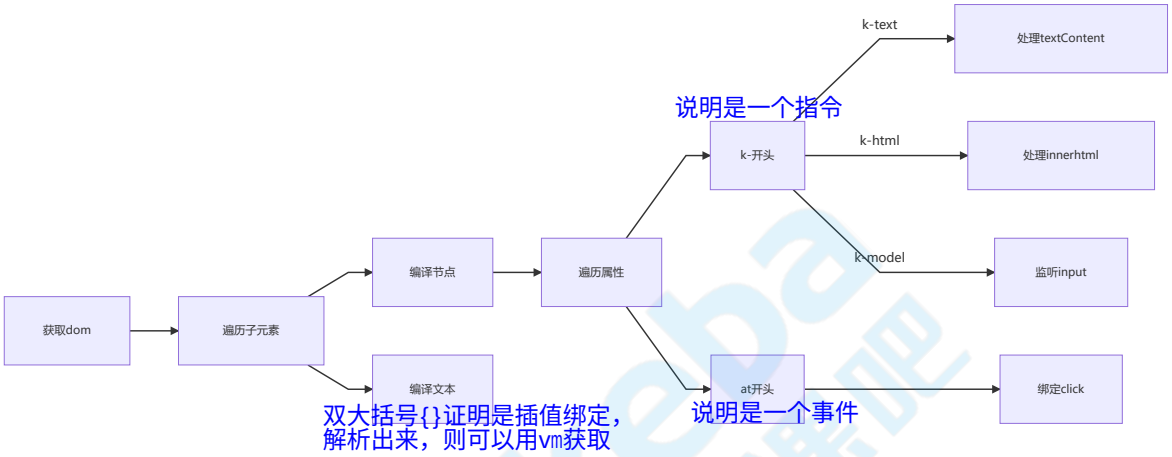
```
class KVue {
  constructor(options) {
    // 。。。
    proxy(this, '$data')
  }
}

function proxy(vm) {
  Object.keys(vm.$data).forEach(key => {
    Object.defineProperty(vm, key, {
      get() {
        return vm.$data[key];
      },
```

```
      set(newVal) {
        vm.$data[key] = newVal;
      }
    });
  })
}
```

编译 - Compile

编译模板中vue模板特殊语法，初始化视图、更新视图



初始化视图

根据节点类型编译，compile.js

编译器：
递归遍历dom树
判断节点类型，如果是文本，则判断是否插值绑定
如果是元素，则遍历其属性，判断是否是指令或事件，然后递归其子元素

```
class Compile {
  constructor(el, vm) {
    this.$vm = vm;    Vue实例，由于处于编译阶段，所以此时的$vm中只有$data，可以理解为直接对应的是$data
    this.$el = document.querySelector(el); 元素的数组

    if (this.$el) {
      this.compile(this.$el);
    }
  }

  compile(el) {
    const childNodes = el.childNodes;
    Array.from(childNodes).forEach(node => {
      if (this.isElement(node)) {
        console.log("编译元素" + node.nodeName);    元素有nodeName用于判断
      } else if (this.isInterpolation(node)) {
        console.log("编译插值文本" + node.textContent);    插值文本有文本内容
      }
      if (node.childNodes && node.childNodes.length > 0) {    元素有children则递归遍历
        this.compile(node);
      }
    });
  }
```

Array.from()方法可以将元素的数组转换为真正的数组(childNodes有可能只是一个类数组)

```
  isElement(node) {
    return node.nodeType == 1;
  }

  isInterpolation(node) {
    return node.nodeType == 3 && /\{\{(.*)\}\}/.test(node.textContent);
  }
}
```

nodeType==3 证明是文本

转义符转义两个{}，其中()是用于在RegExp中分组
用RegExp.$1可以获取第一组内容

编译插值，compile.js

```
compile(el) {
    // ...
        } else if (this.isInerpolation(node)) {
            // console.log("编译插值文本" + node.textContent);
            this.compileText(node);
        }
    });
}
```

此方法是用于当出现{{counted}}的时候显示的是counter的内容

```
compileText(node) {
    console.log(RegExp.$1);
    node.textContent = this.$vm[RegExp.$1];
}
```

key

value(此时的$vm中已经有对应data)

编译元素

```
compile(el) {
    //...
    if (this.isElement(node)) {
        // console.log("编译元素" + node.nodeName);
        this.compileElement(node)
    }
}
compileElement(node) {
    let nodeAttrs = node.attributes;
    Array.from(nodeAttrs).forEach(attr => {
        let attrName = attr.name;
        let exp = attr.value;
        if (this.isDirective(attrName)) {
            let dir = attrName.substring(2);
            this[dir] && this[dir](node, exp);
        }
    });
}

isDirective(attr) {
    return attr.indexOf("k-") == 0;
}
```

1.节点是元素(假设此时只有 ，则node为P)
2.遍历其属性列表

规定：指令以k- 开头，k-xx=oo

指令名称

k-xx

指令的表达式对应$data中的key
由于处于编译阶段，所以此时的$vm中只有$data，可以理解为直接对应的是$data

oo

xx

执行指令

等于if(this[dir]){
    this[dir](node,exp)
}

根据
此时的this[dir]为this.text

k-text的情况

```
text(node, exp) {
    node.textContent = this.$vm[exp];
}
```

对应；

与上述compileText对比，compileText直接用RegExp的$，而text则使用动态获取this.$vm[exp]

this.$vm[exp]

```
}
```

k-html

```
html(node, exp) {
    node.innerHTML = this.$vm[exp]
}
```

> 留个作业：实现事件和双绑

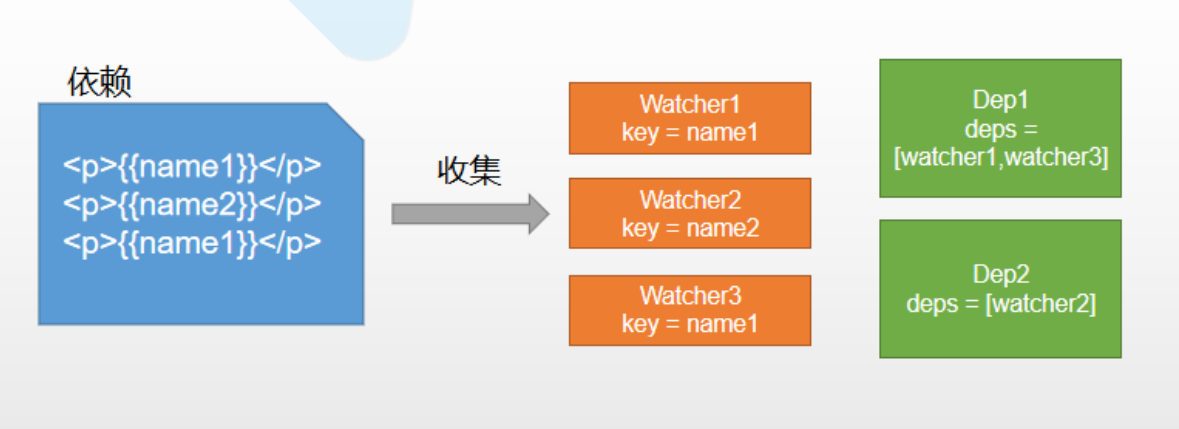
依赖收集

视图中会用到data中某key，这称为依赖。同一个key可能出现多次，每次都需要收集出来用一个Watcher来维护它们，此过程称为依赖收集。

多个Watcher需要一个Dep来管理，需要更新时由Dep统一通知。

看下面案例，理出思路：

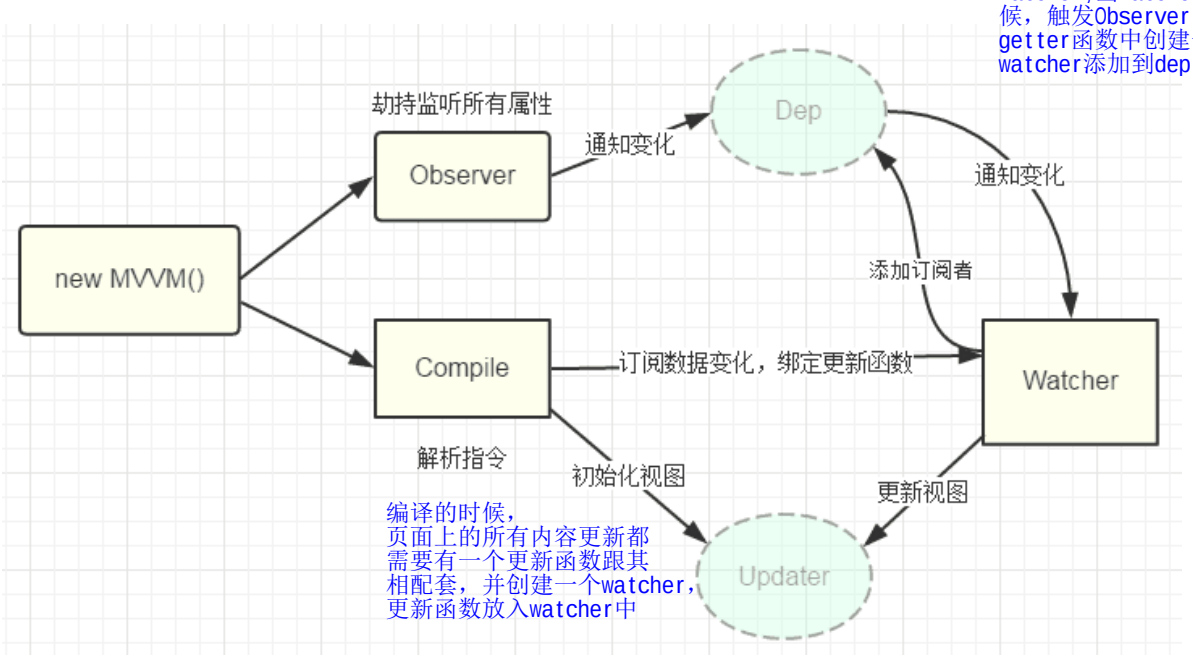
```
new Vue({
    template:
        `<div>
            <p>{{name1}}</p>
            <p>{{name2}}</p>
            <p>{{name1}}</p>
         <div>`,
    data: {
        name1: 'name1',
        name2: 'name2'
    }
});
```



实现思路

1. defineReactive时为每一个key创建一个Dep实例
2. 初始化视图时读取某个key，例如name1，创建一个watcher1
3. 由于触发name1的getter方法，便将watcher1添加到name1对应的Dep中
4. 当name1更新，setter触发时，便可通过对应Dep通知其管理所有Watcher更新

可以全局创建一个变量，保存起当前watcher,当watcher读取某个key的时候，触发Observer的getter函数，在getter函数中创建一个dep，并将该watcher添加到dep中



创建Watcher，kvue.js

```
const watchers = [];//临时用于保存watcher测试用

// 监听器：负责更新视图      观察者：保存更新函数，值发生变化时调用更新函数
class Watcher {
  constructor(vm, key, updateFn) {
    // kvue实例
    this.vm = vm;
    // 依赖key
    this.key = key;
    // 更新函数
    this.updateFn = updateFn;

    // 临时放入watchers数组
    watchers.push(this)
  }

  // 更新                    watcher自身的更新函数，call()方法表示绑定到vm上下文，并将最新的值传给update让其更新
  update() {
    this.updateFn.call(this.vm, this.vm[this.key]);
  }                         此处的update函数会返回一个对应key的最新value
}
```

编写更新函数、创建watcher

```
// 调用update函数执插值文本赋值
compileText(node) {
    // console.log(RegExp.$1);
    // node.textContent = this.$vm[RegExp.$1];
    this.update(node, RegExp.$1, 'text')
}

//元素
text(node, exp) {
    this.update(node, exp, 'text')
}
```

```
}

//元素
html(node, exp) {
    this.update(node, exp, 'html')
}

update(node, exp, dir) {
    const fn = this[dir+'Updater']
    fn && fn(node, this.$vm[exp])
    new Watcher(this.$vm, exp, function(val){
        fn && fn(node, val)
    })
}

textUpdater(node, val) {
    node.textContent = val;
}
htmlUpdater(node, val) {
    node.innerHTML = val
}
```

*根据上面定义的watcher类，其update函数会返回一个对应key的最新值，所以此处的val代表watcher返回的最新值

声明Dep　　用于依赖、管理某个key的所有watcher　　*需要将Dep和key关联起来

```
class Dep {
    constructor () {
        this.deps = []
    }

    addDep (dep) {
        this.deps.push(dep)
    }

    notify() {
        this.deps.forEach(dep => dep.update());
    }
}
```

此处的dep代表的是每一个watcher实例

由于此处的dep代表的是每一个watcher实例
而watcher实例身上有update方法
所以此处可以执行dep.update()

创建watcher时触发getter

```
class Watcher {
  constructor(vm, key, updateFn) {
    Dep.target = this;
    this.vm[this.key];
    Dep.target = null;
  }
}
```

依赖收集，创建Dep实例

```
defineReactive(obj, key, val) {
  this.observe(val);

  //创建一个dep和当前key一一对应，也就是说当执行defineReactive的时候，传入key同时创建了一个dep
  const dep = new Dep()
```

```
  Object.defineProperty(obj, key, {
    get() {  *依赖收集发生在getter里面
      Dep.target && dep.addDep(Dep.target);
      return val
    },
    set(newVal) {
      if (newVal === val) return
      dep.notify()
    }
  })
}
```

作业

- 实现数组响应式
```
1 找到数组原型
2 覆盖数组中的方法，为他们添加更新方法，使其可以通知更新
3 将得到的新原型设置到数组实例原型上
```
- 完成后续k-model、@xx