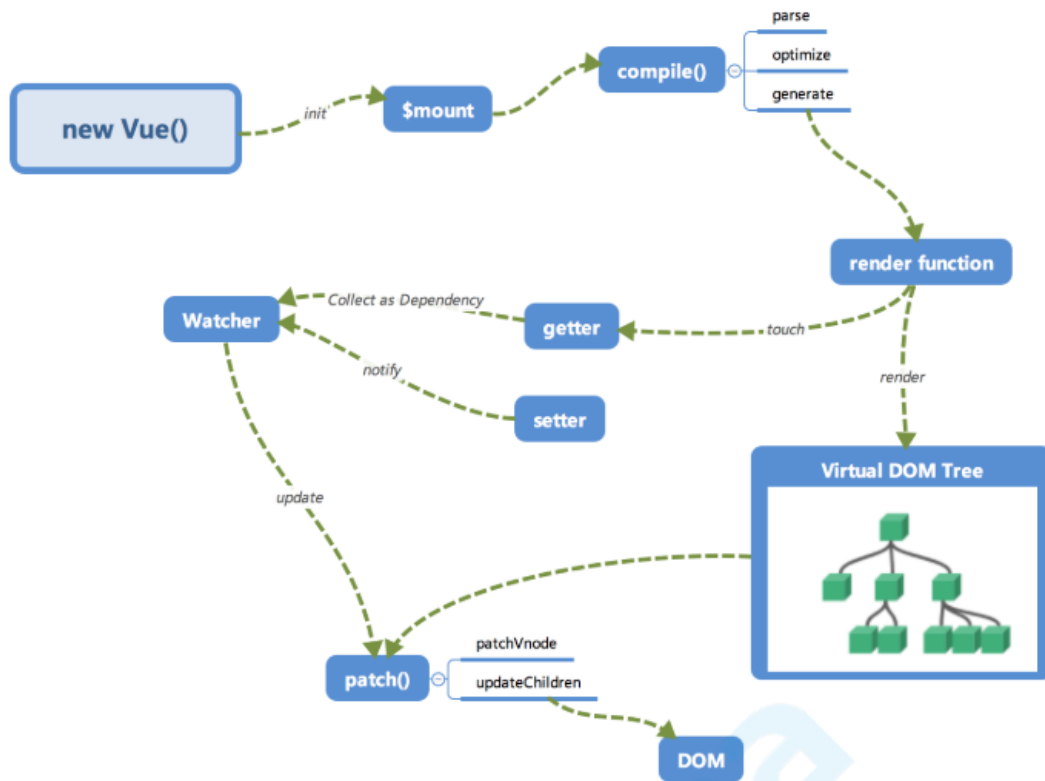


## 作业

1. 尝试去看看vue-router的[源码](#)，并解答：
  - 参数获取
  - 嵌套路由的解决方式
2. 完成kvue中getters部分

```
class Store {  
  constructor(options = {}) {  
    options.getters && this.handleGetters(options.getters);  
  }  
  
  handleGetters(getters) {  
    this.getters = {}; // 定义this.getters  
    // 遍历getters选项，为this.getters定义property  
    // 属性名就是选项中的key，只需定义get函数保证其只读性  
    Object.keys(getters).forEach(key => {  
      Object.defineProperty(this.getters, key, {  
        get: () => { // 注意依然是箭头函数  
          return getters[key](this.state);  
        }  
      });  
    });  
  }  
}
```

## Vue工作流程



## 初始化

初始化data、props、事件等

## 挂载

执行编译，首次渲染、创建和追加过程

## 编译

编译模块分为三个阶段：parse、optimize、generate

## 数据响应式

渲染函数执行时会触发getter进行依赖收集，将来数据变化时会触发setter进行更新

## 虚拟dom

Vue2开始支持Virtual DOM，通过JS对象描述dom，数据变更时映射为dom操作

```

// dom
<div name="开课吧" style="color:red" @click="xx">
  <a> click me</a>
</div>

// vdom
{
  tag: 'div',
  props:{
    name: '开课吧',
    style:{color:red},
    onClick:xx
  }
  children: [

```

```

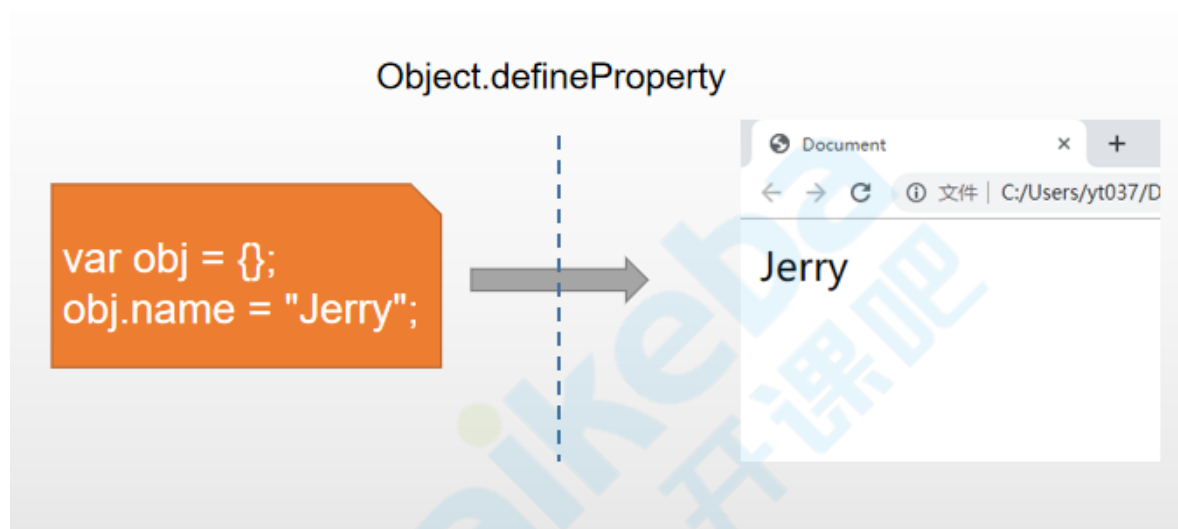
    {
      tag: 'a',
      text: 'click me'
    }
  ]
}

```

## 更新视图

数据修改时监听器会执行更新，通过对比新旧vdom，得到最小修改，就是 patch

## Vue 2响应式的原理：defineProperty



```

<div id="app">
  <p>你好, <span id='name'></span></p>
</div>
<script>
  var obj = {};
  Object.defineProperty(obj, "name", {
    get() {
      console.log('获取name')
      return document.querySelector('#name').innerHTML;
    },
    set(nick) {
      console.log('设置name')
      document.querySelector('#name').innerHTML = nick;
    }
  });

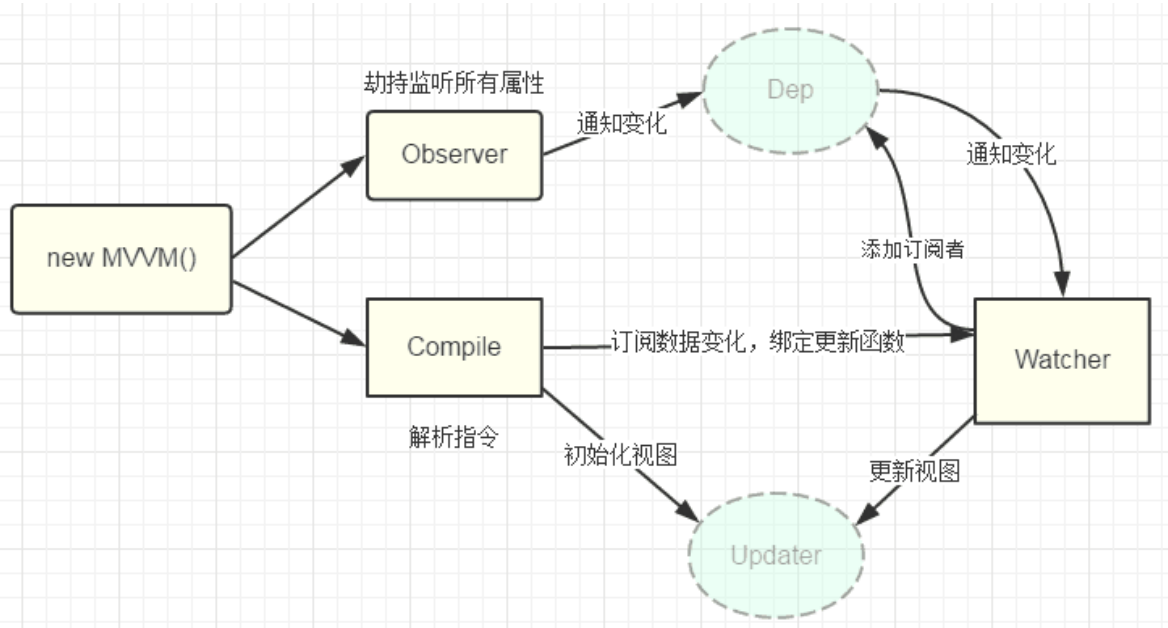
  obj.name = "Jerry";
  console.log(obj.name)
</script>

```

## 实现自己的Vue

简化版架构图

开课吧web全栈架构师



```
// 创建kvue.js
// new KVue({
//   data: {
//     msg: 'hello, vue'
//   }
// })

class KVue {
  constructor(options) {
    // 保存选项
    this.$options = options;
    // 传入data选项
    this.$data = options.data;
    // 响应化
    this.observe(this.$data);
  }

  observe(value) {
    if (!value || typeof value !== "object") {
      return;
    }
    // 遍历, 执行数据响应式
    Object.keys(value).forEach(key => {
      this.defineReactive(value, key, value[key]);
    });
  }

  defineReactive(obj, key, val) {
    // 递归
    this.observe(val);
    // 给obj定义属性
    Object.defineProperty(obj, key, {
      get() {
        return val;
      },
    });
  }
}
```

```

    set(newVal) {
      if (newVal === val) {
        return;
      }
      val = newVal;
      console.log(`${key}属性更新了`);
    }
  });
}
}

// 创建index.html
<script src="kvue.js"></script>
<script>
  const app = new KVue({
    data: {
      test: "I am test",
      foo: {
        bar: "bar"
      }
    }
  });

  app.$data.test = "hello, kaikeba!";
  app.$data.foo.bar = "oh my bar";
</script>

```

为\$data做代理, kvue.js

```

observe(value) {
  //...

  Object.keys(value).forEach(key => {
    this.defineReactive(value, key, value[key]);
    // 代理data中的属性到vue根上
    this.proxyData(key);
  });
}

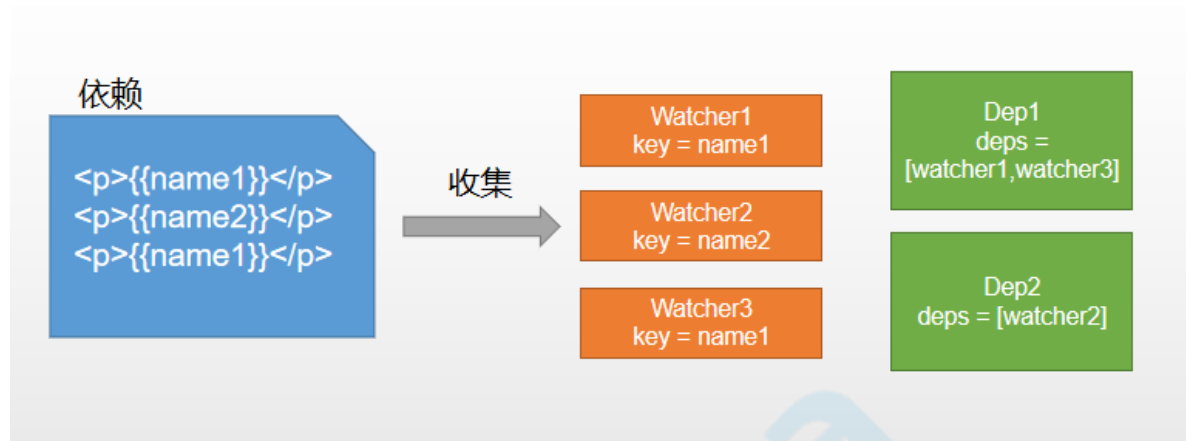
// 在vue根上定义属性代理data中的数据
proxyData(key) {
  Object.defineProperty(this, key, {
    get() {
      return this.$data[key];
    },
    set(newVal) {
      this.$data[key] = newVal;
    }
  });
}

```

测试代码的变化

```
<script>
  app.test = "hello, kaikeba!";
  app.foo.bar = "oh my bar";
</script>
```

## 依赖收集与追踪



看下面案例，理出思路：

```
new vue({
  template:
    `<div>
      <span>{{name1}}</span>
      <span>{{name2}}</span>
      <span>{{name1}}</span>
    </div>`,
  data: {
    name1: 'name1',
    name2: 'name2',
    name3: 'name3'
  },
  created(){
    this.name1="开课吧"
    this.name3="Jerry"
  }
});
```

创建Dep, kvue.js

```
// kvue.js
class Dep {
  constructor () {
    // 存数所有的依赖
    this.deps = []
  }

  // 在deps中添加一个监听器对象
  addDep (dep) {
    this.deps.push(dep)
  }
}
```

```

// 通知所有监听器去更新视图
notify() {
  this.deps.forEach(dep => dep.update());
}
}

```

创建Watcher, kvue.js

```

// 监听器：负责更新视图
class watcher {
  constructor (vm, key) {
    // 在new一个监听器对象时将该对象赋值给Dep.target，在get中会用到
    Dep.target = this
    this.vm = vm;
    this.key = key;
  }

  // 更新视图的方法
  update() {
    console.log(`属性${this.key}更新了`);
  }
}

```

依赖收集, kvue.js

```

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

  // 每次defineReactive创建一个Dep实例
  const dep = new Dep()

  Object.defineProperty(obj, key, {
    get() {
      // 将Dep.target（即当前的Watcher对象存入Dep的deps中）
      Dep.target && dep.addDep(Dep.target);
      return val
    },
    set(newVal) {
      if (newVal === val) return
      // 在set的时候触发dep的notify来通知所有的watcher对象更新视图
      dep.notify()
    }
  })
}

```

测试代码

```

class KVue {
  constructor(options) {
    // ...

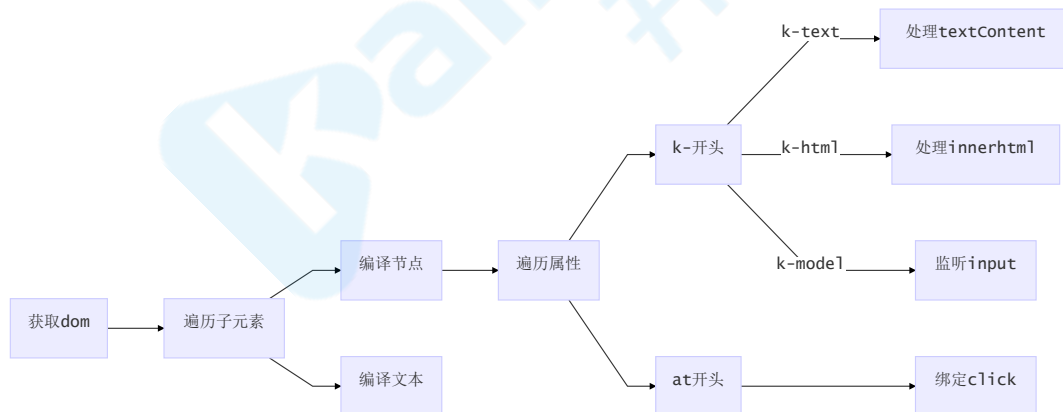
    // 新建一个watcher观察者对象，这时候Dep.target会指向这个watcher对象
    new watcher(this, 'test');
    // 访问get函数，为了触发依赖收集
    this.test
    new watcher(this, 'foo.bar');
    this.foo.bar
  }
}

```

## 编译compile

核心任务：

1. 获取并遍历DOM树
2. 文本节点：获取{}格式的内容并解析
3. 元素节点：访问节点特性，截获k-和@开头内容并解析



目标功能

```

<body>
  <div id="app">
    <p>{{name}}</p>
    <p k-text="name"></p>
    <p>{{age}}</p>
    <p>
      {{doubleAge}}
    </p>
    <input type="text" k-model="name">

```



```

    <button @click="changeName">呵呵</button>
  <div k-html="html"></div>
</div>
<script src='./compile.js'></script>
<script src='./kvue.js'></script>

<script>
  const kaikeba = new KVue({
    el: '#app',
    data: {
      name: "I am test.",
      age: 12,
      html: '<button>这是一个按钮</button>'
    },
    created() {
      console.log('开始啦')
      setTimeout(() => {
        this.name = '我是测试'
      }, 1500)
    },
    methods: {
      changeName() {
        this.name = '哈喽，开课吧'
        this.age = 1
      }
    }
  })
</script>
</body>

```

## compile.js

- 根据节点类型编译，compile.js

```

// 编译器用法: new Compile(el, vm)
class Compile {
  constructor(el, vm) {
    this.$vm = vm;
    this.$el = document.querySelector(el);

    if (this.$el) {
      // 把$el里面内容搬到片段里面操作保证效率
      this.$fragment = this.node2Fragment(this.$el);
      // 执行编译，替换动态内容
      this.compile(this.$fragment);
      // 重新放回到$el中
      this.$el.appendChild(this.$fragment);
    }
  }

  // 把el里面内容搬到fragment里面操作保证效率
  node2Fragment(el) {
    const fragment = document.createDocumentFragment();
    let child;
    while ((child = el.firstChild)) {

```

```

        fragment.appendChild(child);
    }
    return fragment;
}

// 递归遍历el，分别处理元素节点和插值表达式
compile(el) {
    const childNodes = el.childNodes;
    Array.from(childNodes).forEach(node => {
        if (this.isElement(node)) {
            console.log("编译元素" + node.nodeName);
        } else if (this.isInterpolation(node)) {
            console.log("编译插值文本" + node.textContent);
        }
        // 递归子节点
        if (node.childNodes && node.childNodes.length > 0) {
            this.compile(node);
        }
    });
}

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

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

```

测试代码

```

class KVue {
    constructor(options) {
        // ...
        // watcher测试代码不需要了
        // new watcher();
        // this.$data.test;
        // new watcher();
        // this.$data.foo.bar;

        // 执行编译
        new Compile(options.el, this)
    }
}

```

- 执行编译, compile.js

```

compile(el) {
    const childNodes = el.childNodes;
    Array.from(childNodes).forEach(node => {
        if (this.isElement(node)) {
            // console.log("编译元素" + node.nodeName);
            // 编译元素
            this.compileElement(node);
        }
    });
}

```

```

    } else if (this.isInterpolation(node)) {
      // console.log("编译插值文本" + node.textContent);
      // 编译插值
      this.compileText(node);
    }
    // ...
  });
}
// 编译元素：处理指令k-和事件@开头属性
compileElement(node) {}
compileText(node) {
  console.log(RegExp.$1);
  node.textContent = this.$vm.$data[RegExp.$1];
}

```

- 编写更新函数、创建watcher，真正实现依赖收集， compile.js

```

// compile.js
compileText(node) {
  // console.log(RegExp.$1);
  // node.textContent = this.$vm.$data[RegExp.$1];
  // 调用更新函数
  const exp = RegExp.$1;
  this.update(node, this.$vm, exp, "text");
}
// 通用更新函数，根据指令决定调用哪个更新器
update(node, vm, exp, dir) {
  let updaterFn = this[dir + "Updater"];
  updaterFn && updaterFn(node, vm[exp]);
  new watcher(vm, exp, function(value) {
    updaterFn && updaterFn(node, value);
  });
}
// 插值文本更新器
textUpdater(node, value) {
  node.textContent = value;
}

```

- 更新Watcher， kvue.js

```

class Watcher {
  constructor(vm, key, cb) {

    this.vm = vm;
    this.key = key;
    // 传入更新函数cb
    this.cb = cb;

    // 设定将当前watcher实例到Dep.target
    Dep.target = this;
    this.vm[this.key]; // 读取vm的属性触发依赖收集
    Dep.target = null;
  }
}

```

```

// 更新
update() {
  // console.log('视图更新啦! ');
  this.cb.call(this.vm, this.vm[this.key]);
}
}

```

- 编译元素, compile.js

```

compileElement(node) {
  //获取节点属性
  let nodeAttrs = node.attributes;
  Array.from(nodeAttrs).forEach(attr => {
    // 规定: 指令以 k-xxx 命名
    let attrName = attr.name; // 属性名称
    let exp = attr.value; // 表达式
    if (this.isDirective(attrName)) {
      let dir = attrName.substring(2); // 截取指令名称
      // 执行指令解析
      this[dir] && this[dir](node, this.$vm, exp);
    }
  });
}

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

text(node, vm, exp) {
  this.update(node, vm, exp, "text");
}

```

## 拓展

- 数组响应化怎么做

## 作业

- 实现v-html、v-model、@click