作业

- v-html指令

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

htmlUpdater(node, value) {
    node.innerHTML = value;
}
```

- v-model指令

```
model(node, vm, exp) {
    this.update(node, vm, exp, "model");

    node.addEventListener("input", e => {
          vm[exp] = e.target.value;
    });
}

modelUpdater(node, value) {
    node.value = value;
}
```

- 事件监听

```
compileElement(node) {
    let nodeAttrs = node.attributes;
    Array.from(nodeAttrs).forEach(attr => {
          // ...
          // 事件处理
          if (attrName.indexOf("@") == 0) {
              // @click="handleClick"
              const dir = attrName.substring(1); // 事件名称
              // 事件监听处理
              this.eventHandler(node, this.$vm, exp, dir);
          }
    });
}

// 事件处理: 给node添加事件监听, dir-事件名称
// 通过vm.$options.methods[exp]可获得回调函数
eventHandler(node, vm, exp, dir) {
    let fn = vm.$options.methods && vm.$options.methods[exp];
    if (dir && fn) {
          node.addEventListener(dir, fn.bind(vm));
    }
}
```

今日目标

1. 调试vue项目的方式
    - 安装依赖：npm i
    - 安装打包工具：npm i rollup -g
    - 修改package.json里面dev脚本：

      ```
      "dev": "rollup -w -c scripts/config.js --sourcemap --environment
      TARGET:web-full-dev"
      ```
    - 执行打包

      ```
      npm run dev
      ```
    - 修改samples里面的文件引用新生成的vue.js
2. vue是如何启动的
3. vue响应式机制逐行分析

整理启动顺序：

platforms/web/entry-runtime-with-compiler.js

```
div#app

new Vue({
  template:dom
}).$mount('#app')
```

src\platforms\web\runtime\index.js

```
Vue.prototype.$mount = function (
  el?: string | Element,
  hydrating?: boolean
): Component {
  el = el && inBrowser ? query(el) : undefined
  return mountComponent(this, el, hydrating)
}
```

- mountComponent

src\core\index.js

```
initGlobalAPI(Vue)
```

- initGlobalAPI
    - set

    - delete
    - nextTick
    - 。。。

src\core\instance\index.js

构造函数

```
function Vue (options) {
  if (process.env.NODE_ENV !== 'production' &&
    !(this instanceof Vue)
  ) {
    warn('Vue is a constructor and should be called with the `new` keyword')
  }
  this._init(options)
}

initMixin(Vue) // 实现上面的_init这个初始化方法
stateMixin(Vue)
eventsMixin(Vue)
lifecycleMixin(Vue)
renderMixin(Vue)
```

- initMixin(Vue)

  ```
  initLifecycle(vm)
  initEvents(vm)
  initRender(vm)
  callHook(vm, 'beforeCreate')
  initInjections(vm) // resolve injections before data/props
  initState(vm)
  initProvide(vm) // resolve provide after data/props
  callHook(vm, 'created')
  ```

    - initLifecycle：$parent,$children等
    - initEvents：事件监听初始化
    - initRender：定义$createElement
    - initInjections: 获取注入数据并做响应化
    - initState：初始化props,methods,data,computed,watch等
    - initProvide：注入数据处理

- stateMixin：实现$watch,$set,$delete
- eventsMixin(Vue)：实现$emit,$on..
- lifecycleMixin(Vue)：实现_update, $forceUpdate, $destroy
- renderMixin(Vue)：_render $nextTick

数据响应式

initData src\core\instance\state.js

```
proxy()
observe(data, true /* asRootData */)
```

observe src\core\observer\index.js

```
ob = new Observer(value)
return ob
```

Observer

数组和对象响应化处理逻辑

```
  /**
   * 对象响应化
   */
  walk (obj: Object) {}

  /**
   * 数组元素响应化
   */
  observeArray (items: Array<any>) {}
```

defineReactive

数据拦截

```
Object.defineProperty(obj, key, {
    enumerable: true,
    configurable: true,
    get: function reactiveGetter () {},
    set: function reactiveSetter (newVal) {}
  })
```

Dep

watcher管理

```
depend () {
    if (Dep.target) {
        Dep.target.addDep(this)
    }
}
```

Watcher src\core\observer\watcher.js

watcher和dep互相添加引用

```
addDep (dep: Dep) {
    const id = dep.id
    if (!this.newDepIds.has(id)) {
        this.newDepIds.add(id)
        this.newDeps.push(dep)
        if (!this.depIds.has(id)) {
            dep.addSub(this)
        }
    }
}
```

watcher更新逻辑：通常情况下会执行queueWatcher，执行异步更新

```
update () {
    /* istanbul ignore else */
    if (this.lazy) {
        this.dirty = true
    } else if (this.sync) {
        this.run()
    } else {
        queueWatcher(this)
    }
}
```

queueWatcher src\core\observer\scheduler.js

推入队列，下个刷新周期执行批量任务，这是vue异步更新实现的关键

```
queue.push(watcher)

nextTick(flushSchedulerQueue)
```

nextTick将flushSchedulerQueue加入回调数组，启动timerFunc准备执行

```
callbacks.push(() => cb.call(ctx))
timerFunc()
```

timerFunc指定了vue异步执行策略，根据执行环境，首选Promise，备选依次为：MutationObserver、setImmediate、setT

数组响应式

数组比较特别，它的操作方法不会触发setter，需要特别处理

Observer

把修改过的数组拦截方法替换到当前数组对象上可以改变其行为

```
if (Array.isArray(value)) {
    if (hasProto) {
        //数组存在原型就覆盖其原型
        protoAugment(value, arrayMethods)
    } else {
        //不存在就直接定义拦截方法
        copyAugment(value, arrayMethods, arrayKeys)
    }
    this.observeArray(value)
}
```

arrayMethods src\core\observer\array.js

修改数组7个变更方法使其可以发送更新通知

```
methodsToPatch.forEach(function (method) {
  // cache original method
  const original = arrayProto[method]

  def(arrayMethods, method, function mutator (...args) {
    //该方法默认行为
    const result = original.apply(this, args)
    //得到observer
    const ob = this.__ob__
    let inserted
    switch (method) {
      case 'push':
      case 'unshift':
        inserted = args
        break
      case 'splice':
        inserted = args.slice(2)
        break
    }
    if (inserted) ob.observeArray(inserted)
    // 额外的事情是通知更新
    ob.dep.notify()
    return result
  })
})
```