基于 Proxy 的数据响应式 Reactive 原理及实现

1. 代码仓库

https://github.com/Nihiue/proxy-reactive-demo

本文的术语及设计主要参照了 vue3 的响应式原理文档，相较于官方文档，本文会更注重探究设计背后的思路以及一些关键的细节

2. 概念与基础

2.1 什么是 Proxy

Proxy - JavaScript | MDN

Proxy和传统的Getter和Setter作用类似，但是功能更全面，可以拦截对于一个对象的各类访问

```
1 const myObj = {
2   foo: "hello",
3   bar: "world"
4 };
5
6 const proxy = new Proxy(myObj, {
7   get(target, key) {
```

```
    console.log(`# get ${key}`);
    return Reflect.get(target, key);
  },
  set(target, key, value) {
    console.log(`# set ${key} to ${value}`);
    Reflect.set(target, key, value);
    return true;
  }
});

console.log(proxy.foo);
proxy.newProp = 1;
console.log(proxy.newProp);
```

这里演示了Proxy的基本用法，其输出为

```
# get foo
hello
# set newProp to 1
# get newProp
1
```

2.2 什么是 WeakMap

WeakMap - JavaScript | MDN

WeakMap 对象是一组键/值对的集合，其中的键是弱引用的

> 弱引用：即使WeakMap通过Key引用了某个对象，它也不影响这个对象垃圾回收GC

其键必须是对象，而值可以是任意的。

```
const myObj = {};

const wMap = new WeakMap();

wMap.set(myObj, {
  data: 'some meta data of myObj'
});
```

weakMap的两个关键特性 对象作为Key和对象弱引用，都是服务于一个场景。

即你有某些关联到一个对象的数据要存放，但是这些数据当且仅当这个对象存在时有意义，并且不希望这种关联阻止对象的GC。

在上面的例子中，当其他myObj的引用消失时，myObj会被GC，而此时其对应的对象

```
{ data: 'some meta data of myObj' }
```

也会一同被GC。

这个特性非常重要，会在本文中多次用到

内存管理 - JavaScript | MDN

2.3 什么是数据响应式

考虑一个场景，我们有变量 A B C

一个函数F，它接受参数 A和B，并且根据 产生一个值 C

```
let A;
let B;

let C = F(A, B);

function update() {
  C = F(A, B)
}

```

在过程式的代码中， C 的计算是一次性的，仅表示执行此行代码时的 A B 值计算出的 C。

如果 A B 的值在变化，并且我们需要获取最新的C值，我们一般需要重新调用update函数。常见的方式有

- 在所有可能改变AB的地方，手动执行 update
- 某种触发机制（比如定时器，点击事件）

在响应式系统中，我们希望AB值变化的时候，update会被自动调用。

即实现一个 watch 函数，传入声明的依赖，当依赖变化时，update被调用。

```
watch(['A', 'B'], update)
```

这样我们就可以得到一个响应式的结果值 C。

术语定义

🎨
1. A 和 B 是 update 函数的 **依赖(Dependency)**
2. update 函数称为一个 **作用(Effect)**
3. update 函数是 A 和 B 的 **订阅者(Subscribers)**

在工程中，总是要求显式声明一个作用的所有依赖是不太合理的，所以我们会实现一个能够自动分析出给定函数的所有依赖的功能，这个流程被称为**依赖收集**。

3. 设计一个响应式系统

3.1 基本思路

一个响应式系统的雏形：获取到函数的所有依赖，在依赖变化时去调用

这样我们就有了几个需要解决的问题

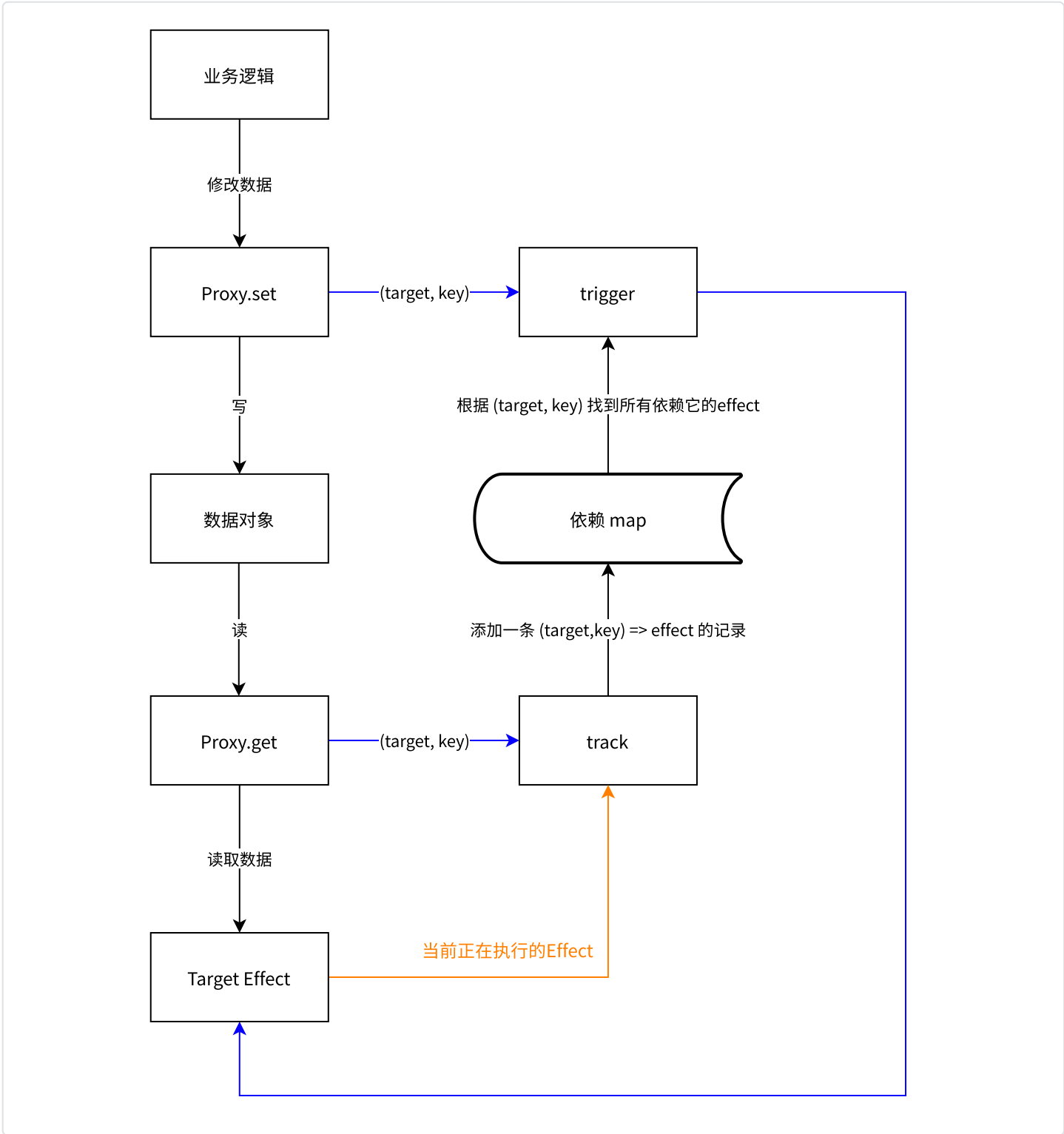
1. 依赖收集：如何获取到函数的依赖
2. 如何检测一个数据的变化
3. 管理数据和订阅者间的关系

定义几个关键流程

🐵
1. 访问数据的记法： **(target, key)**
    a. 即对于数据 foo.bar 的访问记为 (foo,bar)
2. 数据结构：**依赖 map**
    a. key是 (target, key)
    b. value是 **订阅者** 数组
3. 操作：**track 依赖收集**
    a. 当effect运行时，记录该effect所有访问过的值，写入 **依赖map**

4. 操作：**trigger 调用订阅者**
    a. 当数据变化，读取 **依赖map**，调用订阅者

整体逻辑如下



黑色：数据流　蓝色：调用流

3.2 主体框架：使用 Proxy 完成数据的响应式处理

对照2.1中的例子，实现使用Proxy包裹对象从而检测对其的读写，这里体现为reactive方法

```
1
2  function track(target, key) {
3    // 这里从简，具体参考3.4中实现
4    console.log('当前Effect访问', target, key)
5  }
6
7  function trigger(target, key) {
8     // 可以触发执行所有以该值为依赖的effect
9     console.log('触发依赖该值的所有Effect', target, key);
10 }
11
12 function createProxy(obj) {
13   const proxy = new Proxy(obj, {
14      get(target, key) {
15         track(target, key);
16         // 递归调用
17         return reactive(Reflect.get(target, key));
18       },
19       set(target, key, value) {
20         // 注意这里是先调用 Reflect.set 后调用了 trigger
21         // 为什么?
22         Reflect.set(target, key, value);
23         trigger(target, key);
24         return true;
25       }
26   });
27   return proxy ;
28 }
29
30 function reactive(obj) {
31   if (typeof obj !== 'object') {
32     // 非 Obj 类型不使用 Proxy 包裹
33     // 为什么?
34     return obj;
35   }
36   return createProxy(obj);
37 }
38
39 const data = reactive({
40    foo: 1,
41    bar: {
42        value: 1
```

```
43    },
44    tiger: [],
45    cat: {
46        meow: 1
47    }
48 });
49
50 function myFunc() {
51    if (data.foo === 1) {
52       console.log('foo is 1', data.bar.value)
53    } else {
54       console.log('foo is not 1', data.cat.meow);
55    }
56 }
57
58 console.log('# 1');
59 myFunc();
60 data.foo = 2;
61 myFunc();
```

递归包裹Proxy

这里在 Proxy的get中，递归调用了reactive， 而在reactive检测了传入值的类型

- 如果是Object，则调用createProxy返回一个包裹过的Proxy对象
- 如果是简单值（字符串，数字等）则直接返回。这里留一个扣，参考4.3

这是为了处理Object嵌套的问题，因为Proxy只能检测当前Object这一层的property访问，使用递归Proxy嵌套才可以检测内层Object的访问

依赖的动态性： 如果myFunc中有逻辑分支（if等），他的依赖是一成不变的吗?

实例中第一次执行时 data.foo === 1，收集的依赖是 [data.foo， data.bar.value]

而第二次执行， 其依赖可能会变为 [data.foo，data.cat.meow]

这就意味着：

> 🏆 必须在每次执行effect时都重新检测并更新其依赖，因为依赖是动态变化的

3.4 管理函数和依赖间的关系

先实现**依赖map**

```ts
export type KeyType = string | symbol | number;
type ObjectSubMap = Map<KeyType, Set<Function>>;

const subscribers: WeakMap<Object, ObjectSubMap> = new WeakMap();
// 使用 WeakMap 为什么?

export function getSubscribersSet(target: Object, key: KeyType) {
  let targetSubs = subscribers.get(target);

  if (!targetSubs) {
    // lazy create
    targetSubs = new Map();
    subscribers.set(target, targetSubs);
  }

  let keySubs = targetSubs.get(key);

  if (!keySubs) {
    // lazy create
    keySubs = new Set();
    targetSubs.set(key, keySubs);
  }

  return keySubs;
}
```

这里实际上分了将两层map实现，target => (key => [effect])。

这里使用了使用 TS ，借助类型系统可以更明确数据结构。

这样 getSubscribersSet 就完成了所需的映射，并且在访问时可以 lazy 创建尚不存在的 kv 结构。

然后实现 track 和 trigger 方法。

```ts
import { getSubscribersSet, KeyType } from './subscribers';

let activeEffect: Function | undefined;

export function track(target: Object, key: KeyType) {
  if (!activeEffect) {
    return;
  }
```

```
  const subscribers = getSubscribersSet(target, key);
  subscribers.add(activeEffect);
}

export function trigger(target:Object, key: KeyType) {
  getSubscribersSet(target, key).forEach(sub => sub());
}

export function watchEffect(update: Function) {
  const effect = function () {
    activeEffect = effect;
    update();
    activeEffect = undefined;
  }
  effect();
}

```

watchEffect立即执行传入的函数，同时响应式追踪其依赖，并在其依赖变更时重新运行该函数。

这里的 activeEffect 全局变量，他总是指向当前正在执行的 effect

有一个细节，为什么 watchEffect 不能直接写作

```
export function watchEffect(update: Function) {
  activeEffect = update;
  update();
  activeEffect = undefined;
  return update;
}
```

这样记录下的effect就是update本身，而不是前后有activeEffect赋值包裹的新 effect

结合 3.2 的依赖动态性，这样处理后只有第一层运行update会收集其依赖，而后面的运行并不会，这样就会漏掉依赖变化的情况

这样就可以运行一个基本的响应式示例了

```
const myObj = reactive({
  a: 1,
  b: 2
});
```

```
5
6 watchEffect(() => {
7   console.log(`a + b is ${myObj.a + myObj.b}`);
8 });
9
10 myObj.a = 2;
11
12 setInterval(() => {
13   myObj.a += 1;
14 }, 1000);
15
```

4. 性能优化及其它

4.1 ProxyMap

在2.2中，我们每次访问一个对象，都会动态为它生成一个新的Proxy，这显然有额外的性能开销

因此我们可以设计一个全局的ProxyMap

```
1
2 const proxyMap = new WeakMap();
3
4 function reactive(obj) {
5   if (typeof obj !== 'object') {
6     return obj;
7   }
8   if (!proxyMap.has(obj)) {
9     proxyMap.set(obj, createProxy(obj));
10  }
11  return proxyMap.get(obj);
12 }
```

4.2 Primitive Value 的依赖追踪

基本类型 - 术语表 | MDN

3.2 中，只考虑了 object 类型的依赖追踪，因为对其数据的访问 总是 通过访问 property 实现的，这样可以很容易使用 Proxy 来实现。

但是对于基本数据类型(Primitive value, 也称为非引用类型)， 这种实现就行不通了。 *为什么?*

解决方案是对于独立的基本类型，引入一个ref方法将其包装成为对象

```
export function ref(initVal?: any) {
  console.assert(typeof initVal !== 'object', 'should not use ref on object');

  return reactive({
    value: initVal
  });
}
```

其用法是

```
const myNumber = ref(0);

watchEffect(() => {
  console.log(`myNumber = ${myNumber.value}`);
});

myNumber.value = 8;

```

4.3 effect 调用合并 - Tick机制

在之前的trigger方法中，我们总是在 trigger 时立即调用effect

```
export function trigger(target:Object, key: KeyType) {
  getSubscribersSet(target, key).forEach(sub => sub());
}
```

这是一个符合直觉的实现，但是可能有潜在的性能浪费：

> 如果一个 effect 过于频繁地被触发，可能会导致整个系统阻塞。
>
> 比如在一个for循环中进行赋值操作。

考虑一个替代实现：

当 trigger 触发时，我们不立即执行 effect， 而是将 effect放入一个集合，并且每隔固定的时间去执行集合里的所有 effect并清空集合。

如此所有 effect 的执行都被对齐到一个 Tick ，批量执行。

如果在两次 Tick 间隔中 一个 effect 被 trigger 了多次，那么就合并了这N次执行。每个effect在Tick内，至多只会被调用一次。

时间间隔较小（100ms）时，此设计带来的非实时性很难被感知。

```
1
2  const nextTickSubs: Set<Function> = new Set();
3
4  const tickTimer = window.setInterval(function() {
5    const arr = Array.from(nextTickSubs);
6    nextTickSubs.clear();
7
8    // 为什么不直接使用 nextTickSubs.forEach ?
9    arr.forEach(function (sub) {
10     sub();
11   });
12 }, 100);
13
14 export function trigger(target:Object, key: KeyType) {
15   getSubscribersSet(target, key).forEach(function(sub) {
16     nextTickSubs.add(sub);
17   });
18 }
19
```

我们引入了一个间隔 100ms 的 tickTimer，并且使用一个 nextTickSubs 的集合缓存本次间隔中被触发的 effect。

实际上这也是 vue 采用的设计, 大家都很熟悉 $nextTick API就是来自这个机制。

5. 基于响应式系统 100 行代码实现一个迷你前端框架

响应式 VS 非响应式

1. React

   a. 显式数据变更 setState
   b. 完整重建 vdom 后 diff
2. Augular - 脏检查
   a. 在合适的实际触发脏检查
   b. 脏检查后更新有变动的视图

实现一个最小版本的 MVVM 框架，展示响应式系统的应用场景

https://github.com/Nihiue/proxy-reactive-demo/tree/master/app-demo

目标

- 类 vue API
- 支持 v-on, v-show, v-bind 等核心 dom 指令
- 支持 watch 机制
- 支持扩展 directive

简单起见，不实现

- vdom 及模板引擎
- 组件机制
- v- for 及 v-if 等与vdom关系密切的功能

具体的测试用例参考

前端框架测试应用

关键点

1. 所有对于dom的操作都是数据驱动的，因此他们都是effect
2. watchEffect是所有数据响应式的基础，比如 computed， watch 等机制
3. app对象上和模板产生的所有effect对象都需要绑定上下文 this，以及一些约定好的参数

需要熟悉的API

1. **使用 new Function 来创建一个函数**

```
1 const test = new Function('a', 'b', 'return a + b');
```

```
2
3 // 等效于
4
5 const test = function(a, b) {
6   return a + b;
7 }
```

可以看到 new Function 支持以字符串输入创建一个function，这带来了更多动态性：

 可以在运行时创建任意函数

2. Function.bind 方法

大家可能比较熟悉的是bind用于绑定this, 它会替换指定函数中的this指向并返回一个新函数

```
1 const that = {
2     a: 1,
3     b: 2
4 }
5
6 function add() {
7    return this.a + this.b;
8 }
9
10 add.bind(that)();
```

实际上bind还有个用法是提前绑定函数的某些参数，只留下剩余的参数可变，形成一个偏函数

比如

```
1 functon add (a, b) {
2     return a + b;
3 }
4
5 const myNewFunction = add.bind(null, 100);
6
7 // 等效的 myNewFunction
8
9 const myNewFunction(b) {
10   return 100 + b;
11 }
```

可以看到，add 的第一个参数a被固定为100了。这样新函数就只能计算 100 + ？ 的值了

其实 this 也可看作函数的参数，可以认为所有的 javascript 函数都有一个隐藏的 this 形参，不过其值是系统提供的。

这样看来 bind 的作用就是其字面解释： 提前绑定函数的参数值。

bind提供了另一层的动态性：

> 🎉 给定一个Function，我们可以通过bind去固定其参数，改变其行为

实现 Dom 操作

实现数据变动到操作 dom 的机制

产出为 render 函数，这些 render 函数就是响应式数据的effect

实现 v-bind

考虑以下场景

```
const app = {
  data: reative({
     foo: 'hello, world'
  })
};

/* HTML

<span x-bind:textContent="data.foo"></span>


*/
```

data.foo 的值需要被绑定到 dom 元素的 textContent 属性，那么我们实现一个最简单的版本

```
const el = document.querySelector('[x-bind:textContent]');
```

```
2
3 function update() {
4    el.textContent = app.data.foo;
5 }
6
7 watchEffect(update);
```

可以工作，但是缺乏动态性，如果要 bind 别的参数就不行了

上点难度，来一个支持任意参数的

因为是x-bind:*, 所以就不能使用 css selector 了

```
1
2   const allEls= Array.from(document.querySelectorAll('*'));
3
4   allEls.forEach(function (el) {
5     el.getAttributeNames().forEach(function(attr) {
6       if (!attr.startsWith('x-bind:')) {
7         return;
8       }
9       let [ prefix, propName ] = attr.split(':');
10      const attrVal = el.getAttribute(attr);
11
12      propName = toCamelcase(propName);
13
14      const func = new Function(
15          'data',
16          '$el',
17          `$el['${propName}'] = ${attrVal}`
18      ).bind(app, data, el);
19
20      watchEffect(func);
21
22    }
23 }
24
```

> 所有的dom attribute都是小写的
>
> 那么textContent就要写成 text-content, 再交给 toCamelcase 变回来

这里使用了 newFunction 和 bind，看起来有些变得复杂了

我们以 x-bind:text-content="data.foo"为例看看这段代码怎么执行

```
prefix = "v-bind"
propName = "textContent"
attrVal = "data.foo"

const func = (function(data, $el) {
  $el['textContent'] = data.foo
}).bind(app, data, el)

watchEffect(func);
```

这里的 new Function 配合 bind，自然地注入了 data 和 this

更进一步， 我们可以把注入写成

```
 const func = new Function(
          '{ data, methods }',
          '$el',
          `$el['${propName}'] = ${attrVal}`
      ).bind(app, app, el);
```

使用一个解构，注入app 上更多的东西

特殊处理 bind:class 和 bind:style

对于class和style的bind，它的实现不仅仅是简单赋值，考虑以下用法

```
<span x-bind:class={red: data.foo === true} />
```

首先，attrVal是个字典，其次，dom操作class的api是 $el.classList ，接口也不兼容

所以new Function是传入的body就需要特殊定制

```
let functionBody = `$el['${propName}'] = ${attrVal}`;

if (propName === 'class') {
```

```
    functionBody = `
       const classDict = ${attrVal};
       Object.keys(classDict).forEach(function (className) {
         $el.classList[classDict[className] ? 'add' : 'remove'](className);
       });
    `;
}

 const func = new Function(
    '{ data, methods }',
    '$el',
    functionBody
 ).bind(app, app, el);
```

bind:style的处理与其相似，不再赘述

实现 v-on

有了实现 bind 的基础，实现 on 是比较简单的，直接给出其实现

```
  // x-on:click="methods.handleClick"

  const [ prefix, eventName ] = attr.split(':');
  const attrVal = el.getAttribute(attr);

  const func = new Function(
      '{ data, methods }',
      '$event',
      attrVal
  ).bind(app, app);

  el.addEventListener(eventName, func);
```

注意， 我们虽然声明了一个 $event 的形参，却没有bind值给他

是因为 eventListener 的第一个参数就是 $event，正好留出这个空

实现 v-show

```
  const attrVal = el.getAttribute(attr);

  const func = new Function(
```

```
      '{ data, methods }',
      '$el',
      `$el.style.display = (${attrVal}) ? 'initial' : 'none'`
  ).bind(app, app, el);
```

整理 && 小结

看了以上几个指令，发现 new Function 再 bind 的过程大同小异，所以可以抽象一个方法

统一为所有render函数绑定 this 和 { methods, data }

```
function renderFunction({ args, body, values }, appThis) {
  // args: 新函数的形参
  // body: 新函数的函数体
  // values: 绑定到形参的值

  // 这里自动添加了一个新的形参 { methods, data },  并且给其提供值为 appThis
  // 目的是在模板函数中可以不通过 this 直接访问这两个属性，使得模板更加干净

  return (
        new Function('{ methods, data }', ...args, body)
      ).bind(appThis, appThis, ...values);
}
```

以x-bind为例， 其用法如下，看起来更加清晰

```
renderFunction({
    args: ['$el'],
    body: `$el.${propName} = ${attrVal}`,
    values: [ el ]
  })
```

另外，我们实现一个initDom方法， 将各种指令的调用抽象为接口统一的函数

registerRender 接受一个函数，稍后会对所有调用 registerRender 的函数调用 watchEffect

```
function handleOn(appThis, registerRender, el, attr) {
    const func = renderFunction(/ *  */);
```

```
    registerRender(func);
}
function handleShow(appThis, registerRender, el, attr) {}
function handleBind(appThis, registerRender, el, attr) {}

function initDom(rootEl, app) {
  const renderFuncArray = [];

  function registerRender(func, desc = '') {
    func.effect_debug_info = desc;
    renderFuncArray.push(func);
  }

  const attrHandlers = {
    'x-bind': handleBind,
    'x-on': handleOn,
    'x-show': handleShow
  };

   Array.from(rootEl.querySelectorAll('*')).forEach(function (el) {
    el.getAttributeNames().forEach(function(attr) {
      if (!attr.startsWith('x-')) {
        return;
      }

      Object.keys(attrHandlers).forEach(function(prefix) {
        if (attr.startsWith(prefix)) {
          attrHandlers[prefix](appThis, registerRender, el, attr);
          el.removeAttribute(attr);
        }
      });
    });

  return renderFuncArray;

}
```

实现 frameworkInit

methods 绑定 this

对于 methods中的所有函数，都需要绑定 this 为 app

```
function frameworkInit(mountSel, app) {
  Object.keys(app.methods).forEach(function (name) {
    if (typeof app.methods[name] === 'function') {
      app.methods[name] = app.methods[name].bind(app);
    }
  });
}
```

实现 watch

watch中的对应函数，同样需要绑定 this，不在赘述

watch的响应式部分的原理和 dom操作类似，直接给出实现

```

 Object.keys(app.watch).forEach(function (val) {
    const func = new Function(`this.watch['${val}'](this.${val})`).bind(app);
    func.effect_debug_info = `watch => ${val}`;
    watchEffect(func);
 });
```

调用 initDom 操作

```
 const renderFuncs = initDom(document.querySelector(mountSel), app);
 renderFuncs.forEach(render => {
   watchEffect(render);
 });
```