Vue3训练营 - Element3.0开源实战

带你参加开源项目

- ## 1. Vue3.0 API 与 Element3.0组件开发实战
    - Vue3.0新语法基础
    - Compisiton-Api优势
    - Element组件的Vue3.0API更新实战
    - Vue3处理流程剖析 - 编译器、响应式数据、渲染函数
- ## 2. Vue3.0 原理剖析
    - 响应式Ractivity原理
    - 编译器原理
    - 渲染函数原理
    - Element3.0复杂特性移植
- ## 3. Element3.0项目的自动化测试、工程化、CI/CD
    - Jest测试
    - VTU
    - Element3.0单元测试实战
    - Github版本控制
    - Githook 提交信息校验
    - Eslint
    - 自动回归测试
    - GitHub Action完成CI/CD

第一天 Element组件实战

收获

- 能够使用Vue3.0 CompositionAPI
- 能够完成一个完整的PR过程
- 了解响应式原理

Vite环境搭建

https://github.com/vitejs/vite

```
yarn create vite-app vite
cd vite
yarn
yarn dev
```

Vue3官方CompositionAPI

https://v3.vuejs.org/guide/migration/render-function-api.html#overview

Vue3APi

https://v3.vuejs.org/

CompositionAPI详解

> 官方API https://composition-api.vuejs.org/api.htm

不断变化的数据

```
export default {
  name: "App",
  setup() {
    const position = reactive({});

    // 设置不断变化的数据
    window.addEventListener("mousemove", (e) => {
      console.log("mousemove:", e.pageX, e.pageY);

      position.x = e.pageX;
      position.y = e.pageY;
    });

    return { position };
  },
};
```

创建响应式数据

```
export default {
  name: "App",
  setup() {
    // 创建响应式数据
    const position = reactive({});

    // 设置不断变化的数据
    window.addEventListener("mousemove", (e) => {
      console.log("mousemove:", e.pageX, e.pageY);

      position.x = e.pageX;
      position.y = e.pageY;
    });

    return { position };
  },
};
```

定义渲染视图

```
<template>
  <div>Hello</div>
  <h1>x: {{position.x}} y: {{position.y}}</h1>
</template>
```

计算属性

```
const color = computed(() => {
  const hex = (num) => (num % 255).toString(16);
  return `#${hex(position.x) + hex(position.x) + '00'}`;
});

return { position ,color };
```

```
<h1 :style="{ background:color }">x : {{position.x}} y: {{position.y}}</h1>
```

ref拆装箱

- ref 将给定的值(确切的说是基本数据类型 ini 或 string)创建一个响应式的数据对象

```
const time = ref(0)
setInterval(() => {
  time.value = Date.now()
},1000)
```

数据监听

```
watch(
  time,
  (val, prev) => {
    console.log(`watch ${val}`);
  }
);
```

响应式副作用

```
watchEffect(() => {
  console.log("time", time.vlaue, unref(time));
});
```

事件

```
 const click = () => {
   time.value = 0;
 };
```

```
<button @click="click">Clear</button>
```

Element3.0源码实战

> 跟我一起编写Vue3版ElementUI
>
> https://juejin.im/post/6866373381424414734

获取历史版本过程

```
git log --oneline packages/button

##################################
commit 2d14bf977e986d49db7c8123a17638ae8f8004f7
Author: cuixiaorui <cui_xiaorui@126.com>
Date:   Thu Jul 30 17:43:21 2020 +0800

    refactor: use composition api refactoring button
##################################

##################################
commit 5136af71c1cc89d9c16fadf73f86ecdc76694d4a
Author: shengxinjing <316783812@qq.com>
Date:   Sat Jul 25 09:35:31 2020 +0800
##################################

# 回退为修改前
git reset —hard 2d14b
git reset --hard HEAD^
```

能力提高 刻意练习

Button旧版本

```
<template>
  <button
    class="el-button"
    @click="handleClick"
    :disabled="buttonDisabled || loading"
    :autofocus="autofocus"
    :type="nativeType"
    :class="[
      type ? 'el-button--' + type : '',
      buttonSize ? 'el-button--' + buttonSize : '',
```

```
      {
        'is-disabled': buttonDisabled,
        'is-loading': loading,
        'is-plain': plain,
        'is-round': round,
        'is-circle': circle
      }
    ]"
  >
    <i class="el-icon-loading" v-if="loading"></i>
    <i :class="icon" v-if="icon && !loading"></i>
    <span v-if="$slots.default"><slot></slot></span>
  </button>
</template>
<script>
  export default {
    name: 'ElButton',

    inject: {
      elForm: {
        default: ''
      },
      elFormItem: {
        default: ''
      }
    },

    props: {
      type: {
        type: String,
        default: 'default'
      },
      size: String,
      icon: {
        type: String,
        default: ''
      },
      nativeType: {
        type: String,
        default: 'button'
      },
      loading: Boolean,
      disabled: Boolean,
      plain: Boolean,
      autofocus: Boolean,
      round: Boolean,
      circle: Boolean
    },
```

```
    computed: {
      _elFormItemSize() {
        return (this.elFormItem || {}).elFormItemSize;
      },
      buttonSize() {
        return this.size || this._elFormItemSize || (this.$ELEMENT || {}).size;
      },
      buttonDisabled() {
        return this.disabled || (this.elForm || {}).disabled;
      }
    },

    methods: {
      handleClick(evt) {
        this.$emit('click', evt);
      }
    }
  };
</script>
```

过程版本

```
import { computed, inject, toRefs, unref, getCurrentInstance } from 'vue'

size: {
  type: String,
    default: ''
},


setup(props, ctx) {
    const { size, disabled } = toRefs(props);
    const elFormItem = inject("elFormItem", {});
    const elForm = inject("elForm", {});
    const _elFormItemSize = computed(() => unref(elFormItem.elFormItemSize));

    const buttonSize = computed(
      () =>
        size.value ||
        _elFormItemSize.value ||
        (getCurrentInstance().proxy.$ELEMENT || {}).size
    );
    const buttonDisabled = computed(
      () => disabled.value || unref(elForm.disabled)
    );
```

```
    const handleClick = (evt) => {
      ctx.emit("click", evt);
    };

    return {
      buttonSize,
      buttonDisabled,
      handleClick,
    };
},
emits: ['click'],
```

Button完全版本

```
<template>
  <button
    class="el-button"
    @click="handleClick"
    :disabled="buttonDisabled || loading"
    :autofocus="autofocus"
    :type="nativeType"
    :class="[
      type ? 'el-button--' + type : '',
      buttonSize ? 'el-button--' + buttonSize : '',
      {
        'is-disabled': buttonDisabled,
        'is-loading': loading,
        'is-plain': plain,
        'is-round': round,
        'is-circle': circle
      }
    ]"
  >
    <i class="el-icon-loading" v-if="loading"></i>
    <i :class="icon" v-if="icon && !loading"></i>
    <span v-if="$slots.default">
      <slot></slot>
    </span>
  </button>
</template>
<script>
import { computed, inject, toRefs, unref, getCurrentInstance } from 'vue'

export default {
  name: 'ElButton',
```

```
  props: {
    type: {
      type: String,
      default: 'default'
    },
    size: {
      type: String,
      default: ''
    },
    icon: {
      type: String,
      default: ''
    },
    nativeType: {
      type: String,
      default: 'button'
    },
    loading: Boolean,
    disabled: Boolean,
    plain: Boolean,
    autofocus: Boolean,
    round: Boolean,
    circle: Boolean
  },
  emits: ['click'],
  setup(props, ctx) {
    const { size, disabled } = toRefs(props)

    const buttonSize = useButtonSize(size)
    const buttonDisabled = useButtonDisabled(disabled)

    const handleClick = (evt) => {
      ctx.emit('click', evt)
    }

    return {
      handleClick,
      buttonSize,
      buttonDisabled
    }
  }
}

const useButtonSize = (size) => {
  const elFormItem = inject('elFormItem', {})

  const _elFormItemSize = computed(() => {
    return unref(elFormItem.elFormItemSize)
  })
```

```
  const buttonSize = computed(() => {
    return (
      size.value ||
      _elFormItemSize.value ||
      (getCurrentInstance().proxy.$ELEMENT || {}).size
    )
  })

  return buttonSize
}

const useButtonDisabled = (disabled) => {
  const elForm = inject('elForm', {})

  const buttonDisabled = computed(() => {
    return disabled.value || unref(elForm.disabled)
  })

  return buttonDisabled
}
</script>
```

单元测试

Button.spec.js

```
import Button from '../Button.vue'
import { mount } from '@vue/test-utils'
describe('Button.vue', () => {
  describe('props', () => {
    it('type', () => {
      const wrapper = mount(Button, {
        props: {
          type: 'primary'
        }
      })

      expect(wrapper.classes()).toContain('el-button--primary')
    })

    it('icon', () => {
```

```
    const wrapper = mount(Button, {
      props: {
        icon: 'el-icon-search'
      }
    })

    expect(wrapper.classes()).toContain('el-button--default')
  })

  it('nativeType', () => {
    const wrapper = mount(Button, {
      props: {
        nativeType: 'submit'
      }
    })

    expect(wrapper.attributes('type')).toBe('submit')
  })

  it('loading', () => {
    const wrapper = mount(Button, {
      props: {
        loading: true
      }
    })

    expect(wrapper.classes()).toContain('is-loading')
    expect(wrapper.find('.el-icon-loading').exists()).toBe(true)
  })

  it('disabled', () => {
    const wrapper = mount(Button, {
      props: {
        disabled: true
      }
    })

    expect(wrapper.classes()).toContain('is-disabled')
  })

  it('size', () => {
    const wrapper = mount(Button, {
      props: {
        size: 'medium'
      }
    })

    expect(wrapper.classes()).toContain('el-button--medium')
  })
```

```
  it('plain', () => {
    const wrapper = mount(Button, {
      props: {
        plain: true
      }
    })

    expect(wrapper.classes()).toContain('is-plain')
  })

  it('round', () => {
    const wrapper = mount(Button, {
      props: {
        round: true
      }
    })

    expect(wrapper.classes()).toContain('is-round')
  })

  it('circle', () => {
    const wrapper = mount(Button, {
      props: {
        circle: true
      }
    })

    expect(wrapper.classes()).toContain('is-circle')
  })
})

it('captures click events emitted via click', () => {
  const wrapper = mount(Button)
  wrapper.trigger('click')

  expect(wrapper.emitted('click')).toBeTruthy()
  expect(wrapper.emitted('click').length).toBe(1)
})

it('should only will trigger a click event', async () => {
  let count = 0
  const Comp = {
    template: '<div><el-button @click="handleClick"></el-button></div>',
    setup() {
      const handleClick = () => count++
      return { handleClick }
    }
  }
```

```
    const wrapper = mount(Comp, {
      global: {
        components: {
          'el-button': Button
        }
      }
    })

    await wrapper.findComponent({ name: 'ElButton' }).trigger('click')

    expect(count).toBe(1)
  })

  it("can't captures click events emitted via click when loading ", () => {
    const wrapper = mount(Button, {
      props: {
        loading: true
      }
    })
    wrapper.trigger('click')

    expect(wrapper.emitted('click')).toBeFalsy()
  })
})
```

响应式原理

[渐进式手敲Vue3.0框架]

[Vue3新特性一篇搞懂]

响应式是概念

首先我们说说什么是响应式。通过某种方法可以达到数据变了可以自由定义对应的响应就叫响应式。

```
let effective
function effect(fun) {
    effective = fun
}

function reactive(data) {
    if (typeof data !== 'object' || data === null) {
        return data
    }
    const observed = new Proxy(data, {
        get(target, key, receiver) {
            // 普通写法
            // return target[key]
```

```
            // proxy + reflect 反射
            // Reflect有返回值不报错
            let result = Reflect.get(target, key, receiver)

            // return result
            // 多层代理
            return typeof result !== 'object' ? result : reactive(result)
        },
        set(target, key, value, receiver) {
            effective()
            // 普通写法
            // target[key] = value // 如果设置不成功 没有返回
            // proxy + reflect
            const ret = Reflect.set(target, key, value, receiver)
            return ret
        },

        deleteProperty(target,key){
            const ret = Reflect.deleteProperty(target,key)
            return ret
        }

    })
    return observed
}

module.exports = {
    reactive, effect
}
```

```
<template>
  <h1 :style="{ background:color }">x : {{position.x}} y: {{position.y}}</h1>
  <h1>{{new Date(time)}}</h1>
  <h1>{{new Date(position.time)}}</h1>
  <button @click="click">Clear</button>
</template>

<script>
import { reactive, computed, ref, unref, watchEffect, watch } from "vue";
export default {
  name: "App",

  setup() {
    const position = reactive({ x: 0, y: 0 });

    // 设置不断变化的数据
    window.addEventListener("mousemove", (e) => {
```

```
      // console.log("mousemove:", e.pageX, e.pageY);

      position.x = e.pageX;
      position.y = e.pageY;
    });

    const color = computed(() => {
      const hex = (num) => (num % 255).toString(16);
      return `#${hex(position.x) + hex(position.x) + "00"}`;
    });

    const { time, click } = useTime();

    function useTime() {
      // ref
      const time = ref(0);
      setInterval(() => {
        time.value = Date.now();
      }, 1000);

      position.time = time;

      // watchEffect
      watchEffect(() => {
        console.log("time", time.vlaue, unref(time));
      });

      // watch
      watch(
        time,
        (val, prev) => {
          console.log(`watch ${val}`);
        } // getter
      );

      const click = () => {
        time.value = 0;
      };
      return { time, click };
    }

    return { position, color, time, click };
  },
};

module.exports = {
  useTime: useTime,
  useTime: useTime,
};
```

```
</script>
```

第二天 MVVM原理剖析 - Vue3.0

```

```

> 你的收获
>
> - MVVM解决什么问题
> - Vue3.0的三个组成部分
> - 编译器的组成和各自的作用

进门考

Vue3响应式实现新增加的特性是

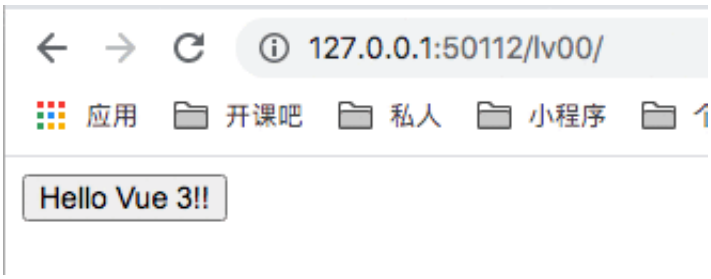
A. 数据可以新增属性

B. 数据递归不会造成影响性能

C. 对象与数组都可以实现响应式

D. 基本类型也可以实现数据响应

史前文明



想象一下如果没有MVVM框架我们要怎么实现一个这样的功能。

- 创建一个数据模型

```
const data = {
        message: 'Hello Vue 3!!'
    }
```

- 创建一个视图

```
<div id='app'>
    <input />
    <button></button>
</div>
```

- 创建一个将模型数据更新到视图上的渲染函数

```
function render() {
    // 更新视图
    document.querySelector('button').innerHTML = data.message
    document.querySelector('input').value = data.message
}
```

- 执行首次数据更新

```
// 首次数据渲染
render()
```

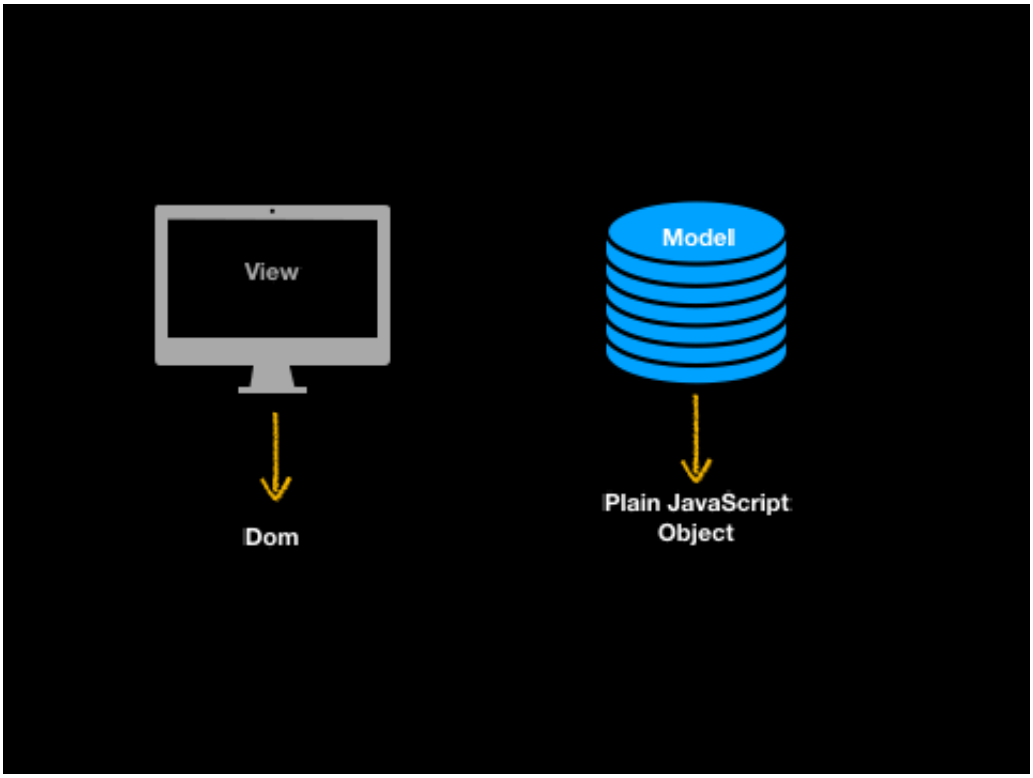
- 绑定按钮点击事件
    - 修改模型中数据： 反转字符串
    - 修改模型后重新渲染数据

```
document.querySelector('button').addEventListener('click', function () {
    data.message = data.message.split('').reverse().join('')
    render()
})
```

- 对输入项变化进行监听
    - 数据项变化时修改模型中数据
    - 修改模型后重新渲染数据

```
document.querySelector('input').addEventListener('keyup', function () {
    data.message = this.value
    render()
})
```

MVVM(Mock版)



MVVM框架其实就是在原先的View和Model之间增加了一个VM层完成以下工作。完成数据与视图的监听。我们这一步先写一个Mock版本。其实就是先针对固定的视图和数据模型实现监听。

接口定义

我们MVVM的框架接口和Vue3一模一样。

初始化需要确定

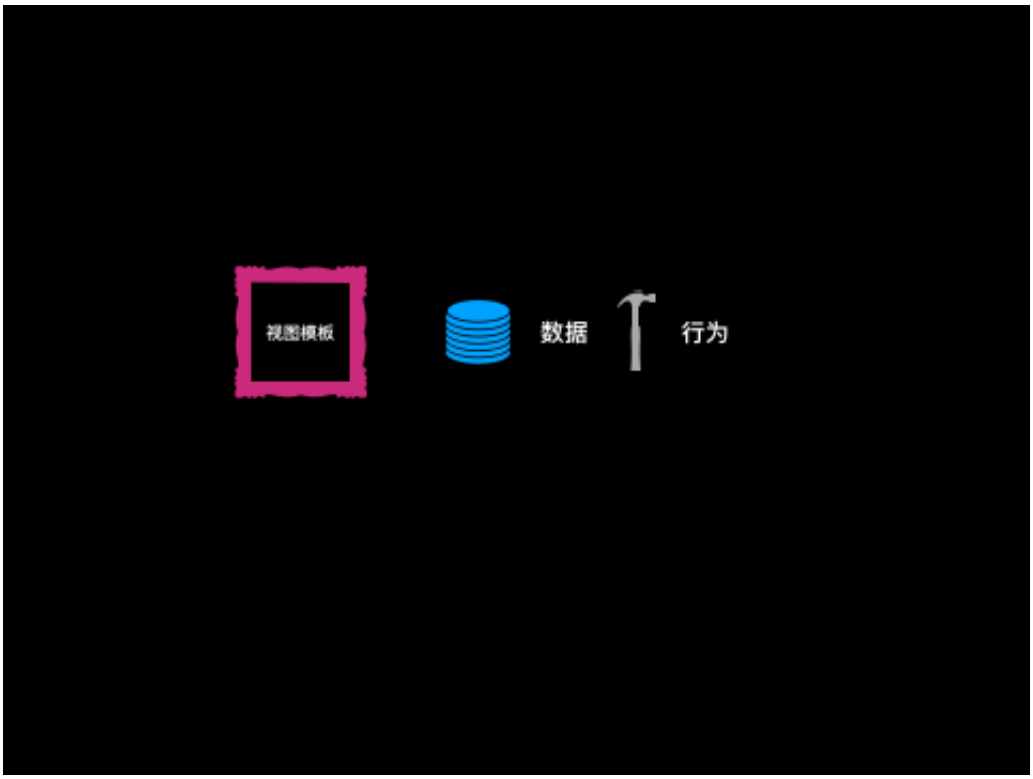
- 视图模板
- 数据模型
- 模型行为 - 比如我们希望click的时候数据模型的message会会倒序排列。

```
const App = {
  // 视图模板
  template: `
<input v-model="message"/>
<button @click='click'>{{message}}</button>
`,
  // 数据模型
  data() {
    return {
      message: 'Hello Vue 3!!'
    }
  },
  // 行为函数
  methods: {
    click() {
      this.message = this.message.split('').reverse().join('')
    }
```

```
  }
}
const {
  createApp
} = Vue
createApp(App).mount('#app')
```

程序骨架

程序执行过程大概如图：




```
const Vue = {
  createApp(config) {
    // 编译过程
    const compile = (template) => (observed, dom) => {
    }
    // 生成渲染函数
    const render = compile()

    // 定义响应函数
    let effective
    // 数据劫持
    observed = new Proxy(config.data(), {
    })

    return {
      // 初始化
      mount: function (container) {
      }
```

```
    }
  }
}
```

编译渲染函数

MVVM框架中的渲染函数是会通过视图模板的编译建立的。

```
// 编译函数
// 输入值为视图模板
const compile = (template) => {
  //渲染函数
  return (observed, dom) => {
    // 渲染过程
  }
}
```

简单的说就是对视图模板进行解析并生成渲染函数。

大概要处理以下三件事

- 确定哪些值需要根据数据模型渲染

```
// <button>{{message}}</button>
// 将数据渲染到视图
button = document.createElement('button')
button.innerText = observed.message
dom.appendChild(button)
```

- 绑定模型事件

```
// <button @click='click'>{{message}}</button>
// 绑定模型事件
button.addEventListener('click', () => {
  return config.methods.click.apply(observed)
})
```

- 确定哪些输入项需要双向绑定

```
// <input v-model="message"/>
// 创建keyup事件监听输入项修改
input.addEventListener('keyup', function () {
  observed.message = this.value
})
```

完整的代码

```
const compile = (template) => (observed, dom) => {

    // 重新渲染
    let input = dom.querySelector('input')
    if (!input) {
        input = document.createElement('input')
        input.setAttribute('value', observed.message)

        input.addEventListener('keyup', function () {
            observed.message = this.value
        })
        dom.appendChild(input)
    }
    let button = dom.querySelector('button')
    if (!button) {
        console.log('create button')
        button = document.createElement('button')
        button.addEventListener('click', () => {
            return config.methods.click.apply(observed)
        })
        dom.appendChild(button)
    }
    button.innerText = observed.message
}
```

数据监听的实现

Vue普遍走的就是数据劫持方式。不同的在于使用DefineProperty还是Proxy。也就是一次一个属性劫持还是一次劫持一个对象。当然后者比前者听着就明显有优势。这也就是Vue3的响应式原理。

Proxy/Reflect是在ES2015规范中加入的，Proxy可以更好的拦截对象行为，Reflect可以更优雅的操纵对象。
优势在于

- 针对整个对象定制 而不是对象的某个属性，所以也就不需要对keys进行遍历。
- 支持数组,这个DefineProperty不具备。这样就省去了重载数组方法这样的Hack过程。
- Proxy 的第二个参数可以有 13 种拦截方法，这比起 Object.defineProperty() 要更加丰富
- Proxy 作为新标准受到浏览器厂商的重点关注和性能优化，相比之下 Object.defineProperty() 是一个已有的老方法
- 可以通过递归方便的进行对象嵌套。

说了这么多我们先来一个小例子

```
var obj = new Proxy({}, {
    get: function (target, key, receiver) {
        console.log(`getting ${key}!`);
        return Reflect.get(target, key, receiver);
    },
    set: function (target, key, value, receiver) {
        console.log(`setting ${key}!`);
        return Reflect.set(target, key, value, receiver);
    }
})
obj.abc = 132
```

这样写如果你修改obj中的值，就会打印出来。

也就是说如果对象被修改就会得的被响应。

当然我们需要的响应就是重新更新视图也就是重新运行render方法。

首先制造一个抽象的数据响应函数

```
// 定义响应函数
let effective
observed = new Proxy(config.data(), {
  set(target, key, value, receiver) {
    const ret = Reflect.set(target, key, value, receiver)
    // 触发函数响应
    effective()
    return ret
  },
})
```

在初始化的时候我们设置响应动作为渲染视图

```
const dom = document.querySelector(container)
// 设置响应动作为渲染视图
effective = () => render(observed, dom)
render(observed, dom)
```

视图变化的监听

浏览器视图的变化,主要体现在对输入项变化的监听上，所以只需要通过绑定监听事件就可以了。

```
    document.querySelector('input').addEventListener('keyup', function () {
        data.message = this.value
    })
```

完整的代码

```
<html>
  <div id="app"></div>
</html>
<script>
  const Vue = {
    createApp(config) {
      // 编译过程
      compile = (template) => (observed, dom) => {
        // 重新渲染
        let input = dom.querySelector("input");
        if (!input) {
          input = document.createElement("input");
          input.setAttribute("value", observed.message);
          input.addEventListener("keyup", function () {
            observed.message = this.value;
          });
          dom.appendChild(input);
        }
        input.value = observed.message;
        let button = dom.querySelector("button");
        if (!button) {
          console.log("create button");
          button = document.createElement("button");
          button.addEventListener("click", () => {
            return config.methods.click.apply(observed);
          });
          dom.appendChild(button);
        }
        button.innerText = observed.message;
```

```
      };

      // 生成渲函数
      const render = compile();

      // 副作用
      let effective;

      // 数据劫持
      observed = new Proxy(config.data(), {
        set(target, key, value, receiver) {
          const ret = Reflect.set(target, key, value, receiver);
          // 触发函数响应
          effective();
          return ret;
        },
      });

      return {
        mount: function (container) {
          const dom = document.querySelector(container);
          // 设置响应动作为渲染视图
          effective = () => render(observed, dom);
          render(observed, dom);
        },
      };
    },
  };

  const App = {
    template: `<input v-model="message"/>
<button @click='click'>{{message}}</button>
    `,
    // 数据模型
    data() {
      return {
        message: "Hello Vue 3!!",
      };
    },
    // 行为函数
    methods: {
      click() {
        this.message = this.message.split("").reverse().join("");
      },
    },
  };

  const { createApp } = Vue;
  createApp(App).mount("#app");
```

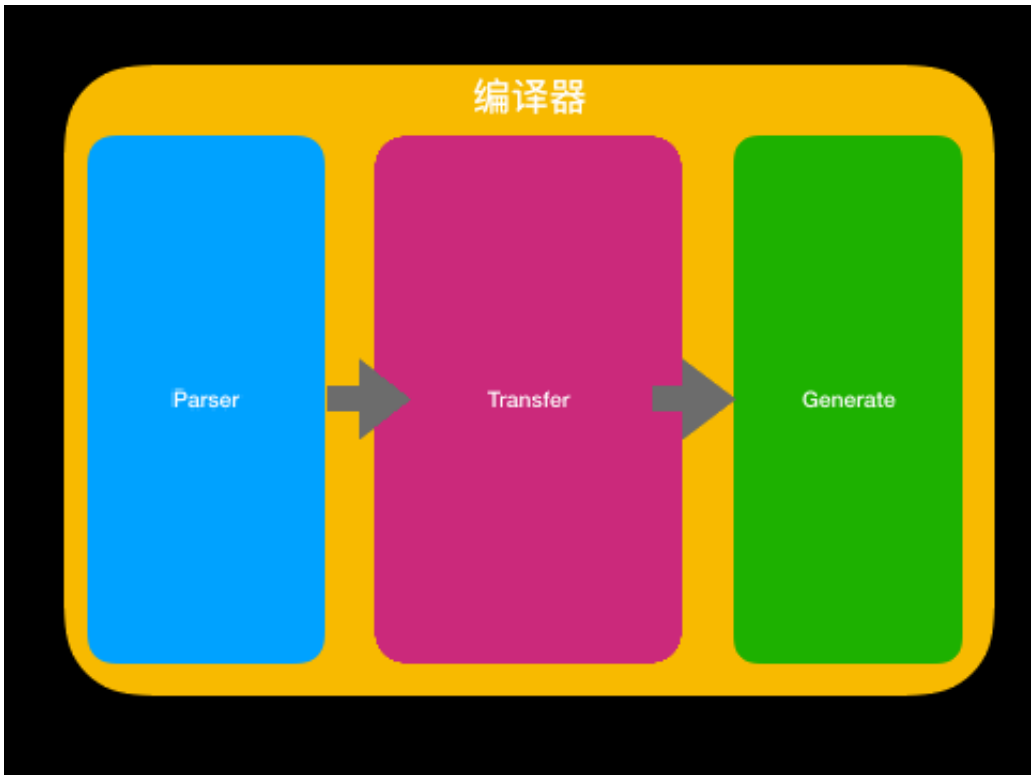
```
</script>
```

响应式数据

```
let effective
function effect(fun) {
    effective = fun
}

function reactive(data) {
    if (typeof data !== 'object' || data === null) {
        return data
    }
    const observed = new Proxy(data, {
        get(target, key, receiver) {
            // target —— 是目标对象，该对象作为第一个参数传递给 new Proxy，
            // property —— 目标属性名
            // receiver —— 如果目标属性是一个 getter 访问器属性，则 receiver 就是本次读取属性所在的 this 对象。通常，这就是 proxy 对象本身（或者，如果我们从代理继承，则是从该代理继承的对象）。现在我们不需要此参数，因此稍后将对其进行详细说明。


            // 普通写法
            // return target[key]
            // proxy + reflect 反射
            // Reflect有返回值不报错
            let result = Reflect.get(target, key, receiver)

            // return result
            // 多层代理
            return typeof result !== 'object' ? result : reactive(result)
        },
        set(target, key, value, receiver) {
            effective()
            // 普通写法
            // target[key] = value // 如果设置不成功 没有返回
            // proxy + reflect
            const ret = Reflect.set(target, key, value, receiver)
            return ret
        },

        deleteProperty(target,key){
            const ret = Reflect.deleteProperty(target,key)
            return ret
        }
```

```
    })
    return observed
}

module.exports = {
    reactive, effect
}
```

编译器Compile



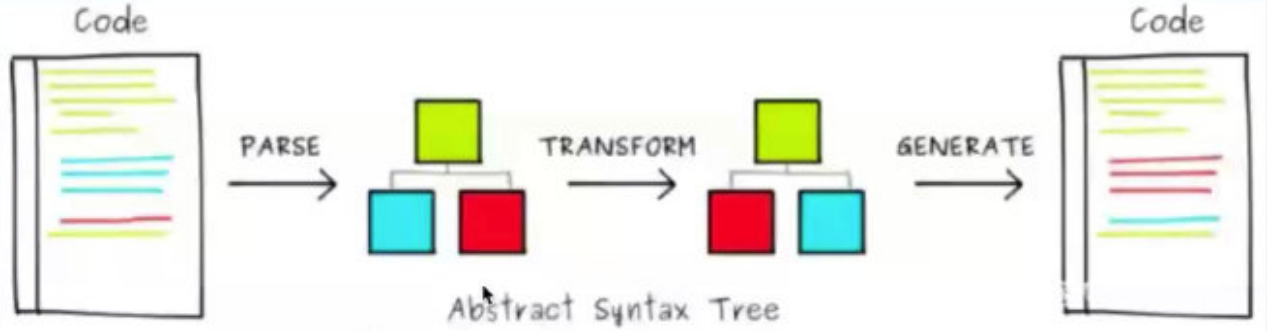
上文已经说过编译函数的功能

```
// 编译函数
// 输入值为视图模板
const compile = (template) => {
  //渲染函数
  return (observed, dom) => {
    // 渲染过程
  }
}
```

简单的说就是

- 输入：视图模板
- 输出：渲染函数

细分起来还可以分为三个个小步骤

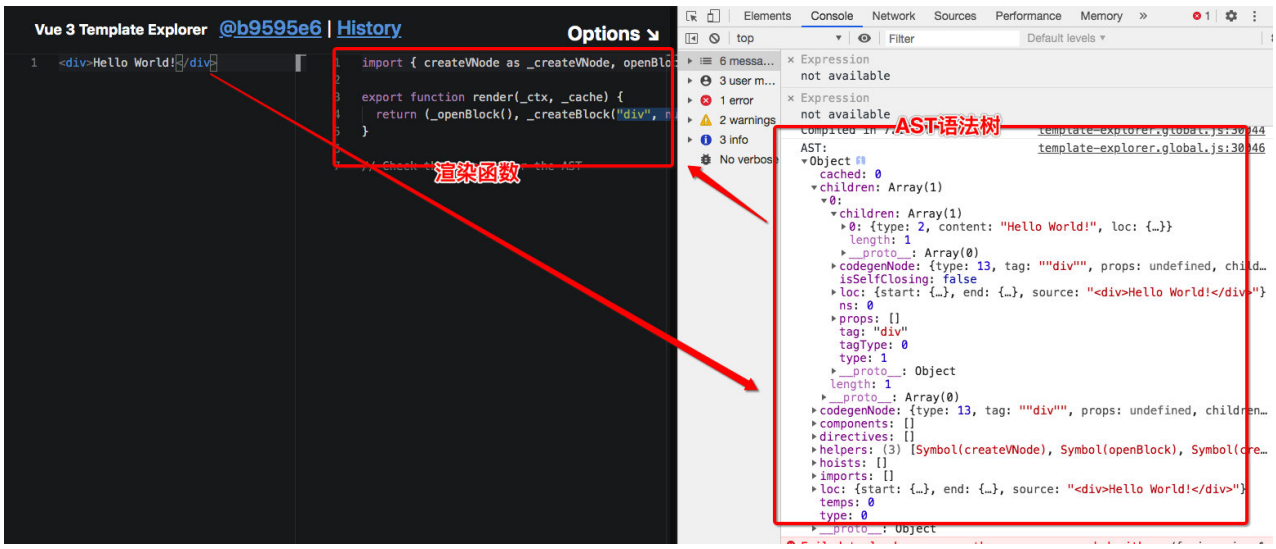


- Parse 模板字符串 -> AST(Abstract Syntax Tree)抽象语法树
- Transform 转换标记 譬如 v-bind v-if v-for的转换
- Generate AST -> 渲染函数

```
//  模板字符串 -> AST(Abstract Syntax Treee)抽象语法树
let ast = parse(template)
// 转换处理 譬如 v-bind v-if v-for的转换
ast = transfer(ast)
// AST -> 渲染函数
return generator(ast)
```

我们可以通过在线版的VueTemplateExplorer感受一下

https://vue-next-template-explorer.netlify.com/



> 编译函数解析

Parse解析器

解析器的工作原理其实就是一连串的正则匹配。

比如：

标签属性的匹配

- class="title"
- class='title'
- class=title

```
# 属性匹配
"class=abc".match(/([a-zA-Z_:][-a-zA-Z0-9_:.]*)\s*=\s*(?:"([^"]*)"|'([^']*)'|([^\s"'=<>`]+))/)

":key='abc'".match(/([a-zA-Z_:][-a-zA-Z0-9_:.]*)\s*=\s*(?:"([^"]*)"|'([^']*)'|([^\s"'=<>`]+))/)

"class='abc'".match((/([a-zA-Z_:][-a-zA-Z0-9_:.]*)\s*=\s*(?:"([^"]*)"|'([^']*)'|([^\s"'=<>`]+))/);
```

这个等实现的时候再仔细讲。可以参考一下文章。

AST解析器实战

那对于我们的项目来讲就可以写成这个样子

```
// <input v-model="message"/>
// <button @click='click'>{{message}}</button>
// 转换后的AST语法树
const parse = template => ({
    children: [{
            tag: 'input',
            props: {
                name: 'v-model',
                exp: {
                    content: 'message'
                },
            },
        },
        {
            tag: 'button',
            props: {
                name: '@click',
                exp: {
                    content: 'message'
```

```
                },
            },
            content:'{{message}}'
        }
    ],
})
```

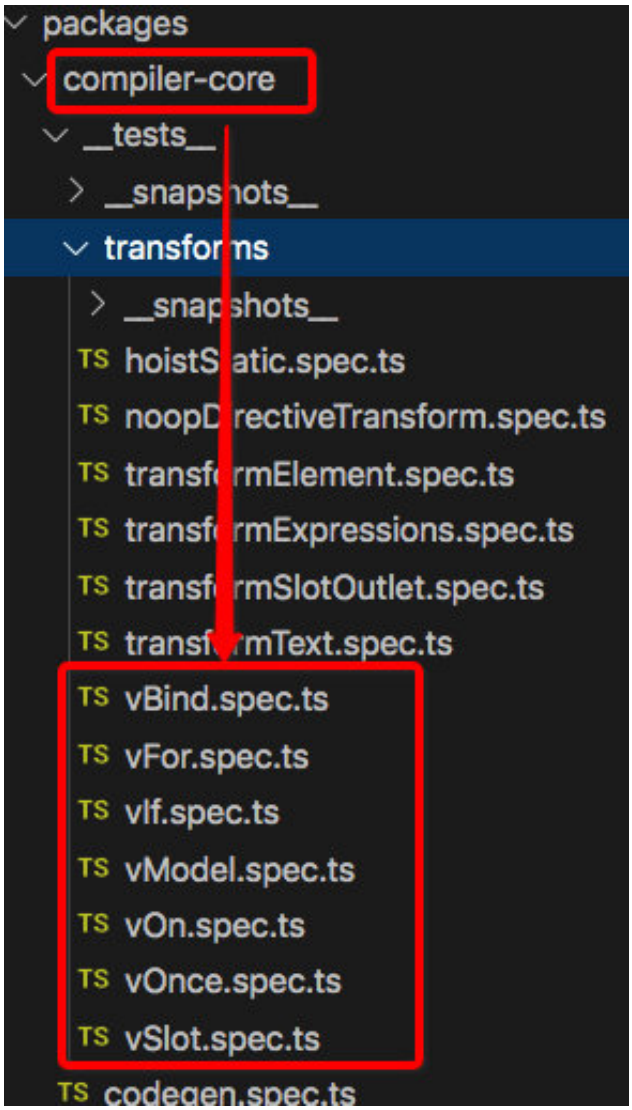
Transform转换处理

前一段知识做的是抽象语法树，对于Vue3模板的特别转换就是在这里进行。

比如：vFor、vOn

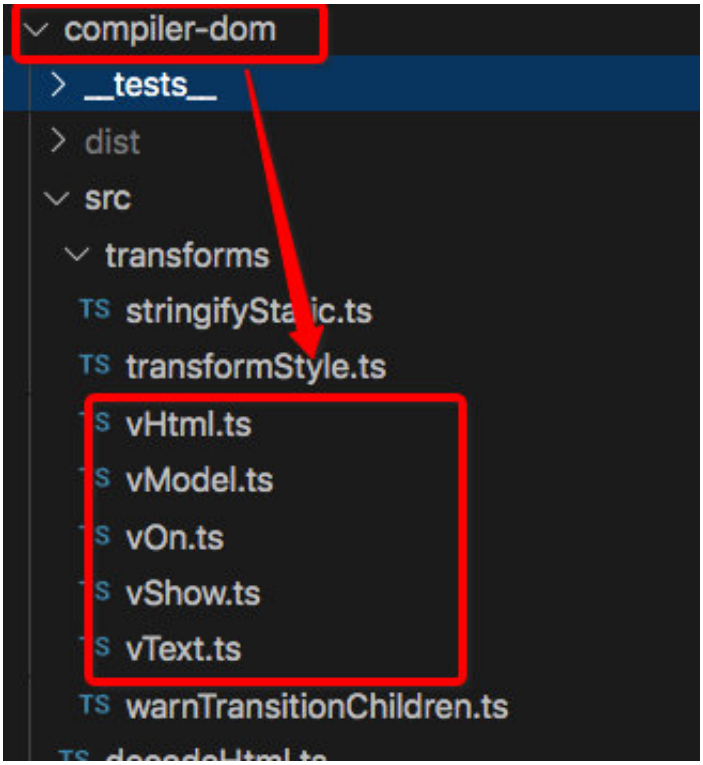
在Vue三种也会细致的分为两个层级进行处理

- compile-core 核心编译逻辑
  - AST-Parser
  - 基础类型解析 v-for 、v-on



- compile-dom 针对浏览器的编译逻辑
  - v-html
  - v-model

- v-clock




```
const transfer = ast => ({
    children: [{
            tag: 'input',
            props: {
                name: 'model',
                exp: {
                    content: 'message'
                },
            },
        },
        {
            tag: 'button',
            props: {
                name: 'click',
                exp: {
                    content: 'message'
                },
            },
            children: [{
                content: {
                    content: 'message'
                },
            }]
        }
```

```
    ],
})
```

Generate生成渲染器

生成器其实就是根据转换后的AST语法树生成渲染函数。当然针对相同的语法树你可以渲染成不同结果。比如button你希望渲染成 button还是一个svg的方块就看你的喜欢了。这个就叫做自定义渲染器。这里我们先简单写一个固定的Dom的渲染器占位。到后面实现的时候我在展开处理。

```
const generator = ast => (observed, dom) => {
    // 重新渲染
    let input = dom.querySelector('input')
    if (!input) {
        input = document.createElement('input')
        input.setAttribute('value', observed.message)
        input.addEventListener('keyup', function () {
            observed.message = this.value
        })
        dom.appendChild(input)
    }
    let button = dom.querySelector('button')
    if (!button) {
        console.log('create button')
        button = document.createElement('button')
        button.addEventListener('click', () => {
            return config.methods.click.apply(observed)
        })
        dom.appendChild(button)
    }
    button.innerText = observed.message
}
```

渲染函数

运行时

更新流程

Vue中的虚拟DOM及diff算法(https://juejin.im/post/6844903923183157261)

浏览器解析过程

- 创建DOM tree

- 创建Style Rules
- 构建Render tree
- 布局Layout
- 绘制Painting

虚拟Dom

蜗牛老师的VDom https://juejin.im/post/6844904134647234568#heading-1