EffectiveFE-Engineering

作者：冰红茶

工作了一段时间后，发现自己在代码高效化和工程化方面欠债太多，所以想记录和总结用以提升效率的最佳实践^_^

参考书籍：

高效前端：《Web 高效编程与优化实践》作者：李银城

《前端工程化 体系设计与实践》作者：周俊鹏

一、HTML/CSS 优化

遵循几条大原则：

> - 能用 HTML/CSS 优化结束战斗的勿用 JS
> - 尽量简练

1.1 巧用伪类

1) hover

> - 高亮：hover 与 opacity 配合

```
.title:hover { opacity: 0.5; }

<h1 class="title">你好</h1>
```

> - 显示子菜单
>   - 这里有一个问题，两个组件需要紧邻着，否则如果存在间隙的话两个组件 hover 的过程变得不连续，显示就会变得失效。
>   - 但是实际业务中，需要两个紧邻组件中的是需要缝隙的，这时候可以使用透明伪元素解决问题

```
ul, li { display: inline-block; margin: 0; padding: 0; } li { margin-
right:
10px; } ul li:last-of-type { margin-right: 0; } .select { display: none; }
.select::before { display: block; content: ''; height: 10px; opacity: 0; }
.select:hover { display: block; } .title:hover + .select{ display: block;
}

<div class="title">你好</div>
<ul class="select">
  <li>选择1</li>
  <li>选择2</li>
  <li>选择3</li>
  <li>选择4</li>
</ul>
```

2) checked

> - 修改 radio/checkbox 的样式

```
        input[type="radio"] + span {
            display: inline-block;
            padding: 3px;
            width: 6px;
            height: 6px;
            border: 1px solid #000;
            border-radius: 50%;
            background: transparent;
            background-clip: content-box;
            transition: all 0.5s;
        }
        input[type="radio"]:checked + span{
            background: #000;
            background-clip: content-box;
        }
        input[type="radio"]  + span + label {
            display: inline-block;
            font-size: 12px;
        }

        <input id="radio1" type="radio" name="singleSelect"></input>
        <span></span>
        <label for="radio1">选择1</label>
        <input id="radio2" type="radio" name="singleSelect"></input>
        <span></span>
        <label for="radio2">选择2</label>
```

3) 前向伪类选择器 nth-last-of-type(n)

> - 多列宽度自适应

```
ul, li { display: inline-block; margin: 0; padding: 0; } ul { width: 100%;
}
li:first-of-type:nth-last-of-type(2), li:first-of-type:nth-last-of-type(2)
~ li
{ width: 50%; } li:first-of-type:nth-last-of-type(3),
li:first-of-type:nth-last-of-type(3) ~ li { width: 33.3%; }
li:first-of-type:nth-last-of-type(4), li:first-of-type:nth-last-of-type(4)
~ li
{ width: 25%; }

<ul class="select">
  <li>选择1</li>
  <li>选择2</li>
  <li>选择3</li>
  <li>选择4</li>
</ul>
```

1.2 HTML 标签

1) 画一个三角形

> - 利用不同 border 边的透明度

```
.triangle { width: 0; height: 0; border-left: 10px solid transparent;
border-right: 10px solid transparent; border-bottom: 10px solid red; }

<div class="triangle"></div>
```

2) 尽可能使用伪元素

> - 伪元素原生计算值是 inline
> - 输入框的不可读可以使用伪元素进行覆盖
> - CSS 计数器 count

```
        .counterReset {
            counter-reset: fruit 1;
        }
        .counterReset input:checked {
            counter-increment: fruit;
        }
        .total::after {
            content: counter(fruit);
            font-size: 14px;
            color: red;
        }

        <div class="counterReset">
            <label><input type="checkbox"></input>香蕉</label>
            <label><input type="checkbox"></input>苹果</label>
        </div>
        <p>你选择了<span class="total"></span>个水果</p>
```

二、js 优化

2.1 几个原则和模式

1) 避免使用全局变量

> -

2) 改变样式

> - 常见的方法是直接使用 getComputedStyle, 添加内联 style 的方式，但是这种方式不好，每次都要添加多个样式，而且不能复用，最佳实践是先把需要实现的样式用 class 实现，然后再用 JS addClass 的方式进行实现

3) 避免使用重复代码

> - 重复代码 -> 封装成函数 -> 封装成模块 -> 封装成库 -> 封装成 SDK
> - 使用策略模式有利于高内聚低耦合，也能体现开闭原则（即对拓展是开放的，对修改是封闭的）

```
        model: {
            low: function() {
                // low speed
            },
            middle: function() {
                // middle speed
            },
            high: function() {
                // high speed
            }
        }
        use(model['middle']);
```

3) 访问者模式

> -

```
function vistor() {}
vistor.prototype.eventName = [];
vistor.prototype.registry = {};
vistor.prototype.on = function () {
  this.eventName.push(arguments[0]);
  this.registry[arguments[0]] = arguments[1];
};
vistor.prototype.emit = function () {
  let eventName = arguments[0];
  let a = Array.from(arguments);
  a.shift();
  this.registry[eventName](...a);
};
```

4) 不要滥用闭包

> - 闭包的作用是可以使子作用域访问父作用域的变量，同时不用闭包内的变量不可见。
> - 子作用域访问上层的作用域需要花费较多的时间，做好直接把父作用域的变量作为函数的参数传进去

2.2 其他优化策略

1) 其他优化策略

> - 使用三目运算符
> - 不要出现魔数，即函数的参数含义不明显，可以先在函数前面把参数重新定义一下名称再传进去
> - Object.assign()合并对象
> - 减少使用 forEach，map 等遍历函数，多使用 includes(), filter(), find()等数组方法
> - 使用 async/await 替代 promise 和 callback hell, 对于一些 callback hell 可以先包装成 promise 再使用 async/await

2.3 lodash的使用

1) Array方法

> - chunk(array, [size=1]) 根据数量分割数组
> - difference(array, [values]) 筛选不相同的元素
> - 升级版 加了一个迭代器 differenceBy(array, [values], [iteratee=_.identity])
> - 不要出现魔数，即函数的参数含义不明显，可以先在函数前面把参数重新定义一下名称再传进去
> - Object.assign()合并对象
> - 减少使用forEach，map等遍历函数，多使用includes(), filter(), find()等数组方法
> - 使用async/await 替代promise和callback hell, 对于一些callback hell可以先包装成promise再使用async/await

---

三、Vue

3.1 八股文

1) `compute` 和 `watch`有什么区别

项目	`compute`	`watch`
异步	不支持	支持
缓存	支持	不支持
流	*一个数据 <- 多个数据*	*行为 <- 一个数据*
属性	`get`(默认)和`set`	`handler`、`immediate`、`deep`
参数	无	`curVal`、`prevVal`

注意：当依赖的属性变化时，computed 不会立即重新计算生成新的值，而是先标记为脏数据，当下次 computed 被获取时候，才会进行重新计算并返回。

2) diff 算法

- 是否是相同的节点，如果节点不同(key 和 sel 节点的选择器)，直接替换
- 如果节点相同，分析子节点的 5 种情况，进行不同的处理
    - `oldVnode === vnode`
    - `oldVnode`有子节点`vnode`没有
    - `oldVnode`没有子节点`vnode`有
    - 都有文本节点
    - 都有子节点
- 递归处理子节点
- 比较时为同层级比较，直接把时间复杂度从 O(3) -> O(1)
- 比较的时候是从首尾向中间进行，一旦`StartIdx > EndIdx`表明 oldCh 和 newCh 至少有一个已经遍历完了，就会结束比较。如果有 key，还会从用 key 生成的对象 oldKeyToIdx 中查找匹配的节点，所以为节点设置 key 可以更高效的利用 dom

3) 生命周期

执行链：父`beforeCreate` => 父`created` => 父`beforeMount` => 子`beforeCreate` => 子`created` => 子`beforeMount` => 子`mounted` => 父`mounted` 父`beforeUpdate` => 子`beforeUpdate` => 子`updated` => 父`updated`

周期	执行顺序	特点
`beforeCreate`	先父后子	可以访问`vm.$parent`和`vm.$createElement`
`created`	先父后子	可以访问`data`、`props`、`methods`、`computed`、`watch`、`inject`
`beforeMount`	先父后子	获取并可以访问`vm.$el`(el 提供的真实节点)，在这之前 template 模板已导入渲染函数编译。而当前阶段虚拟 Dom 已经创建完成，即将开始渲染。在此时也可以对数据进行更改，不会触发 updated。
`mounted`	先子后父	`render`函数 -> `vnode` -> 真实节点
`beforeDestory`	先父后子	

周期	执行顺序	特点
destoryed	先子后父	删除vm, 销毁vm._watcher, 删除数据observer中的引用
beforeUpdate	先父后子	
updated	先子后父	

4) 双向绑定

> - 观察者模式 一个主题多个观察者

```
// 主题，接收状态变化，触发每个观察者
class Subject {
  constructor(state) {
    this.state = state;
    this.observers = [];
  }
  getState() {
    return this.state;
  }
  setState(state) {
    this.state = state;
    this.notifyAllObservers();
  }
  attach(observer) {
    this.observers.push(observer);
  }
  notifyAllObservers() {
    this.observers.forEach((observer) => {
      observer.update();
    });
  }
}

// 观察者，等待被触发
class Observer {
  constructor(name, subject) {
    this.name = name;
```

```
    this.subject = subject;
    this.subject.attach(this);
  }
  update() {
    console.log(`${this.name} update, state: ${this.subject.getState()}`);
  }
}

// 测试代码
let s = new Subject();
let o1 = new Observer("o1", s);
let o2 = new Observer("o2", s);
let o3 = new Observer("o3", s);

s.setState(1);
s.setState(2);
s.setState(3);
```

- 发布订阅者模式

```
  var pubsub = (() => {
      var topics = {};
      function on(topic,fn){
          if (!topics[topic]) topics[topic] = [];
          topics[topic].push(fn);
      }
      function emit(topic,...args){
          if (!topics[topic]) return;
          topics[topic].forEach(fn => fn(...args);
      }
      return {
          on,
          emit
      }
  })()
```

模式	特点
观察者模式	主题和观察者需要相互关联，观察者拥有 update 方法 一对多
发布订阅者模式	发布者和订阅者不需要直接联系 多对多 比较简单，多作为库来使用

- 对象监听方法

```
function activeObject(obj) {
  Object.keys(obj).forEach((key) => {
    let val = obj[key];
    let subject = null,
      watcher = null;
```

```
    Object.defineProperty(obj, key, {
      enumerable: true,
      configurable: true,
      get: () => {
        if (!subject) {
          subject = new Subject(val);
          watcher = new Observer(key, subject);
        }
        return subject.getState();
      },
      set: (value) => {
        if (val !== value) {
          val = value;
          subject && subject.setState(val);
        }
      },
    });
  });
}
```

- 数组窃听方法

```
const methods = ["push", "pop"];

function activeArray(obj) {
  const wrapArrayPrototype = Object.create(Array.prototype);
  subject = new Subject(obj);
  watcher = new Observer(obj, subject);
  methods.forEach((method) => {
    wrapArrayPrototype[method] = function (...args) {
      const result = Array.prototype[method].call(this, ...args);
      subject.setState(result);
      return result;
    };
  });
  obj.__proto__ = wrapArrayPrototype;
}
```

- 综合

```
function activeData(obj) {
  const type = Object.prototype.toString.call(obj).slice(8, -1);
  if (type === "Object") {
    activeObject(obj);
    Object.values(obj).forEach((child) => activeData(child));
  } else if (type === "Array") {
    activeArray(obj);
    obj.forEach((child) => activeData(child));
```

```
    }
  }
```

- Watcher 订阅者是 Observer 和 Compile 之间通信的桥梁，主要做的事情是:
  - 实例化时往主题 subject 里面添加自己
  - 必须有一个 update()方法
  - 待属性变动 subject.notice()通知时，能调用自身的 update()方法，并触发 Compile 中绑定的回调。
- 缺点：无法监听对象的属性的创建和删除，可以使用`this.$set`

5) 预编译

- render 函数 > templates 模板 > el 属性挂载元素 outerHTML
- 在包含单文件组件的项目中，使用 webpack 打包时已经将单文件组件中的模板预先编译成了渲染函数
- 也存在实例化 vue 但是没有 render、templates、el 的情况，就是使用 vue 作为 eventbus 使用时
- 编译时 先转化为 AST 树，在转化为渲染函数，最后返回 Vnode 节点

构建模式	运行时机	webpack 配置	特点
运行时构建	vue 实例化创建节点且存在 render 函数属性时	默认或者`alias: {'vue$': 'vue/dist/vue.runtime.common.js'}`	删除了模板的编译功能，无法支持带`template`属性的 Vue 实例选项
独立构建	vue 实例化创建节点并且不存在 render 函数属性时	`alias: {'vue$': 'vue/dist/vue.common.js'}`	需要完整的模板编译功能

6) 组件间通讯

对象	方法
父子	props 和$emit
多层嵌套	`provide`和`inject` 或者`eventbus ( = new vue())`
状态共享`Vue.observable`	`const store = Vue.observable({ count: 0 }); const mutations = {setCount(count) {store.count = count;}};`
vue 实例(`$on` 和 `$emit`)	`vue.$on vue.$emit vue.$off`
其他 `$ref`/`$parent`/`$children`	`this.$refs.list.getList()`

7) 指令

名称	正常写法	缩写	特点
组件数据绑定	`v-bind:props`	`:props`	
插槽	`v-slot:name`	`#name`	获取插槽作用域 `v-slot:name="scope"`
方法绑定	`v-on:func`	`@func`	获取额外参数和子组件通讯参数 `@callback=handleChange(index, $event)`
双向绑定	`v-model`	-	语法糖，等同于 `<Child :value="value" @input="handleInputValue"></Child>` 子组件必须 emit input 事件：`props: {value: Number}` `$emit('input', value)` ，当然了，你也可以手动修改参数名和方法名，使用`model`字段:`{prop: 'checked',event: 'change'}`
只渲染一次	v-once	-	-
循环	v-for	-	
判断	`v-if` `v-else-if` `v-else`	-	根据表达式的值的真假条件，销毁或重建元素
是否显示	`v-show`	-	根据表达式之真假值，切换元素的 `display` CSS 属性节点还在文档中
innerHTML	`v-html`	-	更新元素的 `innerHTML`
textContent	`v-text`	-	更新元素的 `textContent`

8) 自定义指令

```
// 入口
import Auth from './utils/auth';
Vue.use(Auth);

// auth.js 提供给install方法
const AUTH_LIST = ['admin']

function checkAuth(auths) {
    return AUTH_LIST.some(item => auths.includes(item))
}

function install(Vue, options = {}) {
    Vue.directive('auth', {
        componentUpdated(el, binding) {
            if (!checkAuth(binding.value)) {
```

```
                    el.parentNode && el.parentNode.removeChild(el)
                }
            }
            // bind：只调用一次，指令第一次绑定到元素时调用。在这里可以进行一次性的初始化设置。

            // inserted：被绑定元素插入父节点时调用 (仅保证父节点存在，但不一定已被插入文档中)。

            // update：所在组件的 VNode 更新时调用，但是可能发生在其子 VNode 更新之前。指令的值可能发生了改变，也可能没有。

            // componentUpdated：指令所在组件的 VNode 及其子 VNode 全部更新后调用。

            // unbind：只调用一次，指令与元素解绑时调用。
        })
}

export default { install }

// 组件使用时
<button v-auth="['user']">提交</button>
```

9) 事件修饰符

名称	特点
.stop	阻止事件冒泡
.capture	使用事件捕获模式
.prevent	阻止默认事件
.self	事件只在自己身上发生时才触发，如果触发其他元素通过冒泡或者捕获等方式不会被触发，当自身触发后依然会往外进行冒泡
.once	事件只发生一次
.sync	数据双向绑定，父组件`<Child :value="total" v-on:update:change="total = $event"/>`子组件`$emit('update:change', value)`
表单修饰符`.lazy`, `.trim`, `.number`	配合 v-model 使用, `.number`如果输入的第一个字符是数字，那就只能输入数字，否则他输入的就是普通字符串。
.passive	当页面滚动的时候就会一直触发 onScroll 事件，这个其实是存在性能问题的，尤其是在移动端，当给他加上 .passive 后触发的就不会那么频繁了。
鼠标按钮修饰符	：鼠标左键点击；`.right`：鼠标右键点击；`.middle`：鼠标中键点击；

名称	特点
键盘按键修饰符	`.enter` `.tab` `.delete` (捕获“删除”和“退格”键) `.esc` `.space` `.up` `.down` `.left` `.right`,`.exact` 修饰符允许你控制由精确的系统修饰符组合触发的事件。
串联事件修饰符	串联使用事件修饰符的时候，需要注意其顺序，同样 2 个修饰符进行串联使用，顺序不同，结果大不一样。@click.prevent.self 会阻止所有的点击事件，而 @click.self.prevent 只会阻止对自身元素的点击。

10) 混入 mixins

> - 混入的先被执行，组件数据部分后执行，如果有重复属性以组件数据为准

11) 自定义插件

```
MyPlugin.install = function (Vue, options) {
    // 1. 添加全局方法或 property
    Vue.myGlobalMethod = function () {
        // 逻辑...
    }

    // 2. 添加全局资源
    Vue.directive('my-directive', {
        bind (el, binding, vnode, oldVnode) {
        // 逻辑...
        }
        ...
    })

    // 3. 注入组件选项
    Vue.mixin({
        created: function () {
        // 逻辑...
        }
        ...
    })

    // 4. 添加实例方法
    Vue.prototype.$myMethod = function (methodOptions) {
        // 逻辑...
    }
}
```

12) 过滤器

```
<!-- 在双花括号中 -->
{{ message | filterA | filterB }}
```

```
<!-- 在 `v-bind` 中 -->
<div v-bind:id="rawId | formatId"></div>
```

```
// 局部
filters: {
    capitalize: function (value) {
        if (!value) return ''
        value = value.toString()
        return value.charAt(0).toUpperCase() + value.slice(1)
    }
}

// 全局
Vue.filter('capitalize', function (value) {
    if (!value) return ''
    value = value.toString()
    return value.charAt(0).toUpperCase() + value.slice(1)
})

new Vue({
    // ...
})
```

13) nextTick 与更新循环

- 在 Vue 更新数据的时候，视图不会立即更新，因为在数据更新过程中同一变量可能被修改多次，所以会有一个批处理的过程，保留最后一次修改变量的结果，并把最终结果更新视图。
- 步骤
    - 同步修改数据, Vue 开启一个异步队列，并缓冲在此事件循环中发生的所有数据改变。如果同一个 watcher 被多次触发，只会被推入到队列中一次
    - 查找异步队列，推入执行栈，执行 callback[事件循环]并更新视图,（`promise.then`或者 HTML5 的`MutationObserver`，如果环境不支持就使用`setTimeout(fn, 0)`）
    - nextTick 拿到更新后视图，在同一事件循环中，如果存在多个 nextTick，将会按最初的执行顺序进行调用；
- 官方文档说明：注意 mounted 不会承诺所有的子组件也都一起被挂载。如果你希望等到整个视图都渲染完毕，可以用 vm.$nextTick

```
    mounted: function () {
        this.$nextTick(function () {
            // Code that will run only after the
            // entire view has been rendered
        })
    }
```

13) vue-loader 是什么

> - vue 文件的一个加载器，跟 template/js/style 转换成 js 模块。

3.2 性能优化

1) 在 map 循环中添加不同的 key 值，就地复用

2) 对于不变的对象使用 Object.freeze

3) `v-cloak`解决页面闪烁问题

> - v-cloak 指令保持在元素上直到关联实例结束编译，利用它的特性，结合 CSS 的规则 `[v-cloak] { display: none }` 一起使用就可以隐藏掉未编译好的 Mustache 标签，直到实例准备完毕，但是个人认为加个 loading 体验会更好

```
// template 中
<div class="#app" v-cloak>
  <p>{{value.name}}</p>
</div>

// css 中 [v-cloak] { display: none; }
```

4) `v-once` 和 `v-pre` 提升性能

> - `v-pre` 给我们去决定要不要跳过这个元素和它的子元素的编译过程。可以用来显示原始 Mustache 标签。跳过大量没有指令的节点会加快编译。
> - `v-once` 只会渲染一次，后面的重新渲染都会被跳过

5) 使用函数式组件

> - 无状态，无数据响应，无生命周期，没有 instance 实例, 只会根据传进来的 props 进行数据渲染，基本的骨架如下

```
Vue.component("my-component", {
  functional: true, // 必要
  // Props 是可选的
  props: {
    // ...
  },
  // 为了弥补缺少的实例
  // 提供第二个参数作为上下文
  render(createElement, context) {
    return createElement("div", context.data, [
      context.scopedSlots.default({
        a: 1, // 作为插槽的作用域参数
      }),
    ]);
  },
});
```

// 或者

```
<template functional>
  <button class="btn btn-primary" v-bind="data.attrs" v-on="listeners">
    <p v-for="item in props.items" @click="props.handleClick(item);">
      {{ item }}
    </p>
    <slot />
  </button>
</template>
<script>
  export default {
    props: ["level"],
  };
</script>
```

```
// 或者 https://juejin.im/post/6872128694639394830
// 根据不同的情况渲染不同的组件
var EmptyList = {
  /* ... */
};
var TableList = {
  /* ... */
};
var OrderedList = {
  /* ... */
};
var UnorderedList = {
  /* ... */
};

Vue.component("smart-list", {
  functional: true, // 声明 functional: true，表明它是一个函数式组件
  props: {
    items: {
      type: Array,
      required: true,
    },
    isOrdered: Boolean,
  },
  // 为了弥补缺少的实例
  // 提供第二个参数作为上下文
  render: function (createElement, context) {
    // 组件中所有的一切都是通过 context 传递的
    // 根据不同的情况渲染不同的组件
    function appropriateListComponent() {
      var items = context.props.items;

      if (items.length === 0) return EmptyList;
```

```
      if (typeof items[0] === "object") return TableList;
      if (context.props.isOrdered) return OrderedList;

      return UnorderedList;
    }

    return createElement(
      appropriateListComponent(),
      context.data, // 传递给组件的整个数据对象
      context.children // `VNode` 子节点的数组
    );
  },
});
```

3.3 原则与规范

1) 数据与视图分离

3.4 小技巧

1) 父子组件透传

> - 属性透传v-bind="$props"或者v-bind="$attrs"

```
<template>
  <child-component v-bind="$props" />
</template>

<script>
  import ChildComponent from "@/components/ChildComponent";

  export default {
    props: {
      // 注意这里的校验props
      ...ChildComponent.options.props,
    },
  };
</script>
```

> - 对象透传 也可传递某一特定对象的属性，与provide和inject的区别：provide和inject绑定并不是可响应的

```
<!-- obj = {name: '', id: ''} -->
<Child v-bind="obj"></Child>
<!-- 等价于 -->
<Child :name="obj.name" :id="obj.id"></Child>
```

- 事件监听透传 `v-bind="$listeners"` 但不包括.native 修饰器的

2) 作用域插槽

```
<!-- 子组件 -->
<div>
    <slot name="head" :id="id"><slot>
    <slot name="footer" :item="item"><slot>
</div>

<!-- 父组件 -->
<child>
    <template v-slot:head="scope">{{scope.id}}<template>
    <template v-slot:footer="{item}">{{item}}<template>
</child>
```

3) 动态指令参数

- `<div @[event]="handleChange"></div>`

4) `hookEvent` 的使用

- 可以在模板中监听子组件的生命周期钩子，好处是可以不破坏第三方的源码的同时监听其生命周期
- `<ThirdPart @hook:updated="handleUpdated"></ThirdPart>`
- 也可以使用`vm.$on('hooks:beforeDestory', cb)` 或者 `vm.$once('hooks:beforeDestory', cb)`，可以使代码的可读性更好

5) `watch`

- watch 有一个特点，初始化变量的是时候是不会执行回调的，可以使用`immediate: true`
- `deep: true``可以进行深度监听，但有时 ☝🏻 监听某一层，可以这样写

```
    watch: {
        'obj.a': {
            handler(newVal, oldVal) {
            },
        }
    }
```

6) 渲染函数中使用 JSX

3.5 vue3.0 的特点

1) 性能比 2.0 快 1.3~2 倍

- diff 算法优化
  - vue2.0 的 VNode 比较是全量的，vue3.0 只比较 PatchFlag 标记标记节点，静态节点不比较
  - cachehandlers 事件侦听缓存 vue2.0 的事件绑定是动态的，每次都会重新创建，vue3.0 会缓存不变的事件

2) 使用`typescript`重构

3) `Tree shaking support`

4) `Composition API`

5) 自定义渲染 API `Custom Renderer API`

6) 更先进的组件

- `Fragment Teleport(Protal) Suspense`

7) `v-model`统一双向数据流，删除`.sync`

8) `v-if`、`v-for`优先级问题

- 在 2.x 是`v-for`优先级高，在 3.0 中`v-if`的优先级高

9) 去掉`functional: true`

```
import { h } from "vue";

const FuncComp = (props, context) => {
  return h(`h${props.name}`, context.attrs, context.slots);
};

FuncComp.props = ["level"];

export default FuncComp;
```

10) vue 文件结构

- `beforeCreate`和`created`钩子使用`setup`函数替代
- props 解构会使其丧失响应式的
- 一个组件中可写多个 v-model 指令

```
<!--
            作者：宫小白
            链接：https://juejin.im/post/6874314855281590280
            来源：掘金
            著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
```

```
        -->
<!-- 父组件 -->
<test01 v-model:foo="a" v-model:bar="b"></test01>
<!-- 子组件 -->
<template>
  <div>{{num2}}</div>
  <input
    type="text"
    :value="foo"
    @input="$emit('update:foo',$event.target.value)"
  />
  <input
    type="text"
    :value="bar"
    @input="$emit('update:bar',$event.target.value)"
  />
</template>
<script>
  import { ref, reactive, computed, watch, onMounted, onUpdated,
onUnmounted, provide, inject } from "vue";
  export default {
      props: {
          data: String,
      },
      emits: ["update:foo", "update:bar"],, // 用于v-model
      setup (props, context) {
          provide('xx','1234')
    const data=inject('xx', 该参数为默认值);
          const num = ref(1);
          const obj = reactive({
              name: "gxb",
              age: 18,
              num,
          });
          const num2 = computed(() => num.value + 1);
          const num3 = computed({
              get: () => num,
              set: value => num.value = value
          });
          watch(num, (name, preName) => {
              console.log(`new ${name}---old ${preName}`);
          });
          // 监听多个
          watch([num, ()=>obj.name], ([newNum, newName], [oldNum,
oldName]) => {
              console.log(`new ${(newNum)},${(newName)}---old
${(oldNum)},${oldName}`);
          });

          // 生命周期
          onBeforeMounted(() => {
              console.log('beforeMounted!')
          });
          onMounted(() => {
```

```
            console.log('mounted!')
        });
        onUpdated(() => {
            console.log('updated!')
        });
        onUnmounted(() => {
            console.log('unmounted!')
        });
        return { num, obj, num2, num3 };
    },
};
</script>
```

11) `Teleport` 传送门

> - 把节点挂载到 body 上

```
<teleport to="body">
  <div v-if="flag">
    <div>模态框</div>
  </div>
</teleport>
```

12) Fragments

> - 原来 template 节点下只能放一个节点，现在可以放多个

四、React

4.1 八股文

1) 单向数据流

> - `view` -> `action` -> `store` -> `reducer` -> `store` -> `view`
> - `view` `dispatch` 一个 `action`，`store`根据`action`的类型`reducer`一个`new state`，`store`拿到`new state`后更新`view`
> - redux 更新视图使用了订阅发布模式

2) `setState`是同步还是异步

> - `setState`只在合成事件和钩子函数中是“异步”的，在原生事件和 `setTimeout` 中都是同步的。
> - setState 的“异步”并不是说内部由异步代码实现，其实本身执行的过程和代码都是同步的，只是合成事件和钩子函数的调用顺序在更新之前，导致在合成事件和钩子函数中没法立马拿到更新后的值，形式了所谓的“异步”，此外可以通过 `setState(newState, cb)` 中的 cb 拿到更新后的结果。
> - 一句话总结：`react`管得到的就是异步 管不到的就是同步

发生时机	特点
批量更新	创建一个异步队列updateQueue，通过 firstUpdate 、 lastUpdate 、 lastUpdate.next 去维护这个队列，相同的key会被覆盖，只保留最后一个更新，这样的话就可以避免多次更新同一个state
合成事件	合成事件的代码放在try里面执行，此时去读state里面的值还是以前的，所以就会造成异步的错觉，最后执行finally的时候次啊回执行performSyncWork方法，更新state并渲染视图
生命周期	如果在componentDidMount中执行SetState，需要在执行完componentDidmount后才去commitUpdateQueue更新
原生事件	没有走合成事件的逻辑，并不像合成事件或钩子函数中被return，而直接走performSyncWork去更新，所以当在原生事件中setState后，能同步拿到更新后的state值
setTimeout	基于event Loop的模型下，没有被 react 包装过，setTimeout中里去setState总能拿到最新的state值

3) 通讯

> - 方式|特点 父子|props 兄弟|父 state 子 props 跨层级通信|Provider，Consumer和Context 发布订阅模式|eventbus on emit 全局状态管理工具|Redux或者Mobx

```
// util.js
import React from "react";
let { Consumer, Provider } = React.createContext(); //创建 context 并暴露Consumer和Provider模式
export { Consumer, Provider };
```

```
<!-- 父组件 -->
<!-- 导入 Provider -->
import {Provider} from "../../utils/context"

<Provider value="{name}">
  <div>
    <p>父组件定义的值:{name}</p>
    <Child />
  </div>
</Provider>
```

```
// 导入Consumer
import { Consumer } from "../../utils/context";
function Son(props) {
  return (
    //Consumer容器,可以拿到上文传递下来的name属性,并可以展示对应的值
```

```
      <Consumer>
        {(name) => (
          <div
            style={{
              border: "1px solid blue",
              width: "60%",
              margin: "20px auto",
              textAlign: "center",
            }}
          >
            // 在 Consumer 中可以直接通过 name 获取父组件的值
            <p>子组件。获取父组件的值:{name}</p>
          </div>
        )}
      </Consumer>
    );
}
export default Son;
```

4) 为什么使用框架而不是原生

- *组件化* react的组件化可以做到函数级别的原子组件
- *天然分层* MVVM模式，代码解耦更容易读写
- *开发效率* 不必手动更新 DOM，提高开发效率
- *生态* 数据流管理结构和 UI 库都有成熟的解决方案

5) redux的middleware机制

- 使用applyMiddleware API
- 借鉴 koa 的洋葱圈模型

```
        // 手动包装dispatch
        getDispatchWrapper(store) {
            let next = store.dispatch;
            return action => {
                // before TODO
                const result = next(action);
                // after TODO
                return result;
            }
        }

        // middlewares = [getDispatchWrapper1, getDispatchWrapper2, ...];
        function applyMiddleware(middlewares) {
            middlewares
            .reverse()
            .forEach(getDispatchWrapper => store.dispatch =
getDispatchWrapper(store));
        }
```

> - 上面的做法是每次更新 store.dispatch 方法的引用，只想一个新的函数，此外还有一种方式进行链式调用，使用 next 作为传参代替 store.dispatch

```
// 改进 克里希化getDispatchWrapper
const middle = (store) => (next) => (action) => {
  // before TODO
  console.log("dispatching", action);

  const result = next(action);

  // after TODO
  console.log("next state", store.getState());

  return result;
};

// middlewares = [getDispatchWrapper1, getDispatchWrapper2, ...];
function applyMiddleware(middlewares) {
  middlewares
    .reverse()
    .reduce((ret, middle) => middle(store)(ret), store.dispatch);
}
```

6) thunk

> - 判断action：如果是function类型，就调用这个function（并传入dispatch和getState及extraArgument为参数），而不是任由让它到达 reducer，因为 reducer 是个纯函数，Redux 规定到达 reducer 的 action 必须是一个 plain object 类型。

```
function createThunkMiddleware(extraArgument) {
  return ({ dispatch, getState }) => (next) => (action) => {
    if (typeof action === "function") {
      return action(dispatch, getState, extraArgument);
    }

    return next(action);
  };
}

const thunk = createThunkMiddleware();
thunk.withExtraArgument = createThunkMiddleware;

export default thunk;
```

7) react-redux

> - 工作原理

    - 获取 state, connect 通过 context 获取 Provider 中的 store, store.getState()获取整个 store tree 上所有 state
    - 包装原组件，将mapStateToProps,mapDispatchToProps已属性的形式传入WrappedComponent，mapStateToProps订阅更新，mapDispatchToProps发布更新
    - 监听 store tree，如果 state 变化了就调用 this.setState()触发视图更新
- 从 dispatch -> reduce -> getState 这条流里面如果没有使用异步控制的话，可以同步拿到最新的state
- 从 dispatch -> reduce -> connect -> initSubscription -> trySubscribe-> props 这条流里面，使用了setState的方法，所以会表现出【异步】

8) 组件/逻辑复用以及各自优缺点

方式	优点	缺点
mixin	-	mixin跟组件之间存在隐式依赖，依赖关系不透明，增加维护成本，特别是多个mixin共存的情况下，状态增加不可预测性；属性之间会进行打平，增加不可预测性
HOC	通过从外层传props到组件的方式，不更改组件的 state，降低耦合度;传入的参数跟返回组件自身的参数具有天然的层级结构，降低复杂度	扩展性限制:无法从外部访问子组件的 state，因此无法通过shouldComponentUpdate过滤掉不必要的更新（React.PureComponent可以解决这个问题）;Ref 传递问题被阻断（React.forwardRef可以解决）；命名冲突
React Hooks	简洁、解耦、组合、函数友好	学习成本、写法上有限制（不能出现在条件、循环中），React.memo并不能完全替代 shouldComponentUpdate（因为拿不到 state change，只针对 props change）

9) HOC的理解

- HOC本身不是一个component,而是一个function
- 输入的参数是component，返回也是一个component
- 不是react的 API，而是一种基于 React 特性形成的设计模式
- 使用的场景redux中的connect，react-router中的withRouter
- 应用
    - props 的增强
    - 鉴权
    - 生命周期劫持

```
import React, { createContext } from "react";
const { Provider, Consumer } = createContext();
const getNewComp = (Comp, newProps) => {
  return (props) =>
    props.login ? (
      <Consumer>
        {(value) => <Comp {...{ ...props, ...newProps, ...value }} />}
```

```
        </Consumer>
      ) : (
        <NoRight />
      );
};

// Search是一个子组件
const SuperSearch = getNewComp(Search, { a: 1 });
const SuperInput = getNewComp(Input, { a: 2 });
```

```
<Provider value="{{b:" 3}}>
  <SuperSearch name="search" login="{true}" />
  <SuperInput name="input" />
</Provider>
```

- 缺点：多层嵌套调试会很麻烦，可以劫持 props，如果不约定可能会造成冲突

9) React.forwardRef

- 一般来讲，ref 不能用于函数组件，因为函数组件没有实例，不能获取组件对象
- 但是现在有需求：获取函数组件内部某个元素的 dom，那咋办？React.forwardRef应运而生

```
import React, {PureComponent, forwardRef, createRef} from 'react';
const Comp = forwardRef((props, ref) => <span ref={ref}>nihao</span>;
export default class extends PureComponent {
    constructor(props) {
        super(props);
        this.title = createRef();
    }

    componentDidMount() {
        this.props.init();
        console.log(this.title.current);
    }
    render() {
        return <Comp ref={this.title} />
    }
}
```

10) fiber如何理解

- 单线程调度算法
- React 16以前使用reconcilation用的是递归，中断困难，而fiber用的是循环
- 一种将 recocilation分拆成多个小任务，可以随时停止，恢复。停止恢复的时机取决于当前的一帧（16ms）内，还有没有足够的时间允许计算。

> - 时间分片正是基于可随时打断、重启的 Fiber 架构,可打断当前任务,优先处理紧急且重要的任务,保证页面的流畅运行。

11) 生命周期

> - 16.0 版本以前渲染是同步的，16.0 版本以后是异步的，这意味着在 render 函数之前的所有函数都有可能被执行多次，所以这也是`UNSAVE_componentWillMount`，`UNSAFE_componentWillReceiveProps`，`UNSAFE_componentWillUpdate`，被标注为不安全的原因

生命周期	特点
`constructor`	`super(props)`，否则我们无法在构造函数里拿到 `this`
`getDerivedStateFromProps`	静态函数，无法获取 this，根据新的 props 和当前的 state 来调整新的 state。
`UNSAVE_componentWillMount`	在 reader 之前，同步调用 setState 不会引发渲染，此方法是服务端渲染唯一会调用的生命周期函数。常用于当支持服务器渲染时，需要同步获取数据的场景。
`render`	期望是一个纯函数，任何跟数据相关的逻辑请放在 componentDidMount 和 componentDidUpdate 中
`React Updates DOM and refs`	-
`componentDidMount`	适合网络请求和添加订阅。如果直接调用`setState`。它将触发额外渲染，但此渲染会发生在浏览器更新屏幕之前。如此保证了即使在 `render`两次调用的情况下，用户也不会看到中间状态。
`UNSAFE_componentWillReceiveProps`	考虑到因为父组件引发渲染可能要根据 props 更新 state 的需要而设立的，会在已挂载的组件接收新的 `props` 之前被调用
`getDerivedStateFromProps`	替代了`UNSAFE_componentWillReceiveProps`
`shouldComponentUpdate`	`shouldComponentUpdate(nextProps, nextState) {}`根据此函数的返回值来判断是否进行重新渲染，`true` 表示重新渲染，`false` 表示不重新渲染，默认返回 `true`，可以作为性能优化的手段。但是官方提倡我们使用内置的 `PureComponent` 来减少重新渲染的次数，而不是手动编写 `shouldComponentUpdate` 代码。`PureComponent` 内部实现了对 `props` 和 `state` 进行浅层比较。
`UNSAFE_componentWillUpdate`	初始渲染不会调用此方法。但是你不能此方法中调用 `this.setState`，否则就无限循环了

生命周期	特点
getSnapshotBeforeUpdate	替代UNSAFE_componentWillUpdate，在 render 之后，在更新之前（如：更新 DOM 之前）被调用。给了一个机会去获取 DOM 信息，计算得到并返回一个 snapshot，这个 snapshot 会作为 componentDidUpdate 的第三个参数传入。如果你不想要返回值，请返回 null，不写的话控制台会有警告。 getSnapshotBeforeUpdate 方法是在 UNSAFE_componentWillUpdate 后（如果存在的话），在 React 真正更改 DOM 前调用的，它获取到组件状态信息更加可靠。还有一个十分明显的好处：它调用的结果会作为第三个参数传入 componentDidUpdate，避免了 UNSAFE_componentWillUpdate 和 componentDidUpdate 配合使用时将组件临时的状态数据存在组件实例上浪费内存，getSnapshotBeforeUpdate 返回的数据在 componentDidUpdate 中用完即被销毁，效率更高。
componentDidUpdate	-
componentWillUnmount	执行一些清理操作，如定时器，订阅，网络请求，不要 setState，因为没有效果
componentDidCatch	componentDidCatch(error, info) {}如果发生错误，你可以通过调用 setState使用 componentDidCatch渲染降级 UI，但在未来的版本中将不推荐这样做。可以使用静态 getDerivedStateFromError来处理降级渲染
getDerivedStateFromError	static getDerivedStateFromError(error) {}此生命周期会在后代组件抛出错误后被调用。它将抛出的错误作为参数，并返回一个值以更新 state。渲染阶段调用，因此不允许出现副作用

```
        // 作者：LeviDing
        // 链接：https://juejin.im/post/6844904199923187725
        // 来源：掘金
        // 著作权归作者所有。商业转载请联系作者获得授权，非商业转载请注明出处。
        getSnapshotBeforeUpdate(prevProps, prevState) {
            console.log('#enter getSnapshotBeforeUpdate');
            return 'foo';
        }

        componentDidUpdate(prevProps, prevState, snapshot) {
            console.log('#enter componentDidUpdate snapshot = ',
snapshot);
        }
```

4.2 性能优化

-

4.3 原则与规范

- import 顺序
  - 标准模块
  - 第三方模块
  - 自己代码导入（组件）
  - 特定于模块的导入（例如 CSS，PNG 等）
  - 仅用于测试的代码

4.4 小技巧

1) `Portal`

- 将组件挂载于父组件以外的组件或者节点
- `ReactDom.createProtal(Comp, targetCom);`

2) Fragment

- 此节点作为容器不渲染，可以简写为`<></>`
- 不支持 key 和属性。

3) StrictMode

- 仅在开发模式下运行的检查工具
- 检查过时的 API，不安全的生命周期，意外的副作用, 使用废弃的 findDOMNode
  `<StrictMode></StrictMode>`

五、webpack

5.1 八股文

1) 相关概念

- `Entry` 打包入口
- `Module` 模块 一切文件皆可视为模块 从入口开始递归所有模块
- `Chunk` 代码块 一个`chunk`由多个模块组合而成，用于代码的合并与分割
- `Loader` 模块转换器 用于将模块的原内容按照去求转换成新内容
- `Plugin` 拓展插件
- `Output` 输出

2) 构建过程

- 从`Entry`出发依次递归寻找`Module`，利用`Loader`并辅助以`plugin`对`Module`进行转换，最后以`entry`为单位进行分组，其依赖会被打到同一个`chunk`，并输出文件

3) 配置属性

- entry 入口
  - 可以是字符串、数组或者对象，如果是字符串、数组，最后只会输出一个chunk，且使用Output.library时只有最后一个入口文件的模块被导出
  - 也可以写成同步函数或者返回promise的异步函数
- output 配置如何输出
  - filename vs chunkFilename Entry的键值对键值，chunkFilename 非Entry入口的chunk名称，比如动态加载或者CommonChunkPlugin(提取第三方库和公共模块)
  - path vs publishPath path表示打包出来的目录 publishPath表示打包后需要上传服务器的地址
  - library vs libraryTarget library表示导出库的名称 libraryTarget导出方式，比如var/commonjs/commonjs2/this/window/global/umd/libraryExport 表示导出的子模块，默认default
- module
  - 使用loader的test include 和exclude可以减少搜索范围加快速度
  - 使用noParse可以避免递归一些没有依赖模块的文件，比如jQuery, noParse: /jquery/, //不去解析jquery中的依赖库
- resolve 配置寻找模块的代码
  - alias 路径别名
  - mainFields 优先使用那份模块的代码（在package.json里面对应目录）比如：mainFields: ['jsnext:main', 'browser', 'main'];
  - extensions 文件路径后缀优先级 extensions: ['.ts', '.js', 'json'];
  - modules 配置webpack在哪里寻找第三方模块，默认只会在node_modules里面找，如果有很多需要导入的文件在src/components文件夹中，可以配置modules: ['node_modules', 'src/components']，这样可以直接使用import button from 'Button'进行导入
- plugin 配置拓展插件
  -
- devServer

4) sourceMap

- cheap 不包含列信息，且不包含loader信息
- cheap-mudule 不包含列信息，包含loader信息
- inline 把sourceMap以hash字符串的形式写进文件中，一般不会在生产环境中使用
- 在开发环境中，webpack是不支持sourceMap的，需要使用source-map-loader进行加载，且要写在最前面避免其他loader对sourcemap进行转换 enforce: 'pre'

5.2 构建速度优化

1) 多线程压缩

- webpack3 happy-pack
- webpack4 uglifyjs-webpack-plugin | parallel-uglify-plugin | terser-webpack-plugin

2) DLLPlugin预编译

- 创建一个manifest.json文件，DllReferencePlugin使用它来映射依赖项

3) 开启缓存

- 开启`babel-loader`缓存（`babel-loader?cacheDirectory=true`)
- 开启terser-webpack-plugin缓存
- 使用`hard-source-webpack-plugin`提升模块转换阶段缓存

4) 缩小构建目标

- include
- resolve - alias
- resolve - modules
- resolve - extensions
- resolve - mainFields: ['main'] // package.json指定的入口文件 `jsnext:main` `browser` `main`

5.3 优化使用体验

1) 监听文件自动刷新 watch

- 原理 定时获取文件的最后编辑时间，每次保存最新的最后编辑时间，下次更新的时候与上次比较，如果不相同则认为文件发生了变化。但是文件发生了变化也不会第一时间告知监听者，而是先缓存起来，收集一段时间后再一次性告诉监听者，而这个时间可以设置，避免频繁更新。
- 自动刷新浏览器的原理
  - 借助浏览器拓展去通过浏览器的接口去刷新，比如LiveEdit插件
  - 向要开发的网页中注入客户端代码，通过代理客户端刷新整个页面
  - 将要开发的网页装进一个iframe中，通过刷新iframe去看到最新的效果

2) 开启模块热更新

- 在不刷新页面的情况下更新目标节点
- 原理：源码发生变化的时候，只需要重新编译发生变化的模块，再替换掉相应的老模块
- HMR的优点在于可以保存应用的状态，提高开发效率
- 底层原理 Server端使用webpack-dev-server去启动本地服务，内部实现主要使用了webpack、express、websocket。
  - 使用express启动本地服务，当浏览器访问资源时对此做响应。
  - 服务端和客户端使用websocket实现长连接
  - webpack监听源文件的变化，即当开发者保存文件时触发webpack的重新编译。每次编译都会生成hash值、已改动模块的json文件、已改动模块代码的js文件。编译完成后通过socket向客户端推送当前编译的hash戳，客户端的websocket监听到有文件改动推送过来的hash戳，会和上一次对比。一致则走缓存，不一致则通过ajax和jsonp向服务端获取最新资源

> - 使用内存文件系统去替换有修改的内容实现局部刷新
> - 为什么使用JSONP而不用socke通信获取更新过的代码？因为通过socket通信获取的是一串字符串需要再做处理。而通过JSONP获取的代码可以直接执行。

5.4 优化输出质量

1) 区分环境

2) 压缩代码

3) 使用tree shaking

4) 提取公共代码

> - 好处：base.js一旦被用户浏览器缓存，那么在任何页面都不需要重新下载一份，提升客户体验
> - 业务代码.js
> - common.js
> - base.js 所有页面都会用的到的基础库，例如react和react.dom

5) 分割代码按需加载

> - `import(*)` 语法
> - 用在路由切换的场合用得比较多

6) Scope Hoisting

7) 输出分析

六、Axios

6.1 八股文

1) 相关概念

> - Axios 是一个基于 Promise 的 HTTP 客户端，拥有以下特性：
>
>> - 支持promise API
>> - 能够拦截请求和响应
>> - 能够转换请求和相应数据
>> - 能够取消请求和自动转换JSON数据
>> - 客户端支持防御CSRF攻击
>> - 同时支持浏览器和node环境

2) 拦截器

> - `axios.interceptors.request`和`axios.interceptors.response`对象提供的`use`方法

```
        // 添加请求拦截器
        axios.interceptors.request.use(function (config) {
```

```
        config.headers.token = 'added by interceptor';
        return config;
    });

    // 添加响应拦截器
    axios.interceptors.response.use(function (data) {
        data.data = data.data + ' - modified by interceptor';
        return data;
    });
```

- 实现原理
  - 任务注册

```
    // lib/core/Axios.js
    function Axios(instanceConfig) {
        this.defaults = instanceConfig;
        this.interceptors = {
            request: new InterceptorManager(),
            response: new InterceptorManager()
        };
    }

    // lib/core/InterceptorManager.js
    function InterceptorManager() {
        this.handlers = [];
    }

    InterceptorManager.prototype.use = function use(fulfilled,
rejected) {
        this.handlers.push({
            fulfilled: fulfilled,
            rejected: rejected
        });
        // 返回当前的索引，用于移除已注册的拦截器
        return this.handlers.length - 1;
    };
s
```

  - 任务编排 请求拦截是倒序，相应拦截是顺序

```
    // lib/core/Axios.js
    Axios.prototype.request = function request(config) {
        config = mergeConfig(this.defaults, config);

        // 省略部分代码
        var chain = [dispatchRequest, undefined];
        var promise = Promise.resolve(config);
```

```
            // 任务编排
            this.interceptors.request.forEach(function
unshiftRequestInterceptors(interceptor) {
                chain.unshift(interceptor.fulfilled,
interceptor.rejected);
            });

            this.interceptors.response.forEach(function
pushResponseInterceptors(interceptor) {
                chain.push(interceptor.fulfilled, interceptor.rejected);
            });

            // 任务调度
            while (chain.length) {
                promise = promise.then(chain.shift(), chain.shift());
            }

            return promise;
        };
```

- 任务调度

```
// lib/core/Axios.js
Axios.prototype.request = function request(config) {
    // 省略部分代码
    var promise = Promise.resolve(config);
    while (chain.length) {
        promise = promise.then(chain.shift(), chain.shift());
    }
}
```

七、web性能优化

7.1 css 优化

1) 概念

- 是指一个元素外观的改变所触发的浏览器行为，浏览器会根据元素的新属性重新绘制，使元素呈现新的外观。这个过程就是重绘。重排必定会引发重绘，但重绘不一定会引发重排
- 常见的会引起重绘的属性 color、border-style、visibility、background、text-decoration、background-image、background-position、background-repeat、outline-color、outline、outline-style、border-radius、outline-width、box-shadow、background-size

2) 减少reflow对性能的影响的建议

- 不要一条一条地修改 DOM 的样式，预先定义好 class，然后修改 DOM 的 className
- 把 DOM 离线后修改，比如：先把 DOM 给 display:none (有一次 Reflow)，然后你修改100次，然后再把它显示出来

> - 不要把 DOM 结点的属性值放在一个循环里当成循环里的变量
> - 尽可能不要修改影响范围比较大的 DOM
> - 为动画的元素使用绝对定位 absolute / fixed
> - 不要使用 table 布局，可能很小的一个小改动会造成整个 table 的重新布局

7.2 图片延迟

```
        <!DOCTYPE html>
        <html lang="en">
        <head>
            <meta charset="UTF-8">
            <title>Lazyload 1</title>
            <style>
                img {
                display: block;
                margin-bottom: 50px;
                height: 200px;
            }
            </style>
        </head>
        <body>
            <img src="images/loading.gif" data-src="images/1.png">
            <img src="images/loading.gif" data-src="images/2.png">
            <img src="images/loading.gif" data-src="images/3.png">
            <img src="images/loading.gif" data-src="images/4.png">
            <img src="images/loading.gif" data-src="images/5.png">
            <img src="images/loading.gif" data-src="images/6.png">
            <img src="images/loading.gif" data-src="images/7.png">
            <img src="images/loading.gif" data-src="images/8.png">
            <img src="images/loading.gif" data-src="images/9.png">
            <img src="images/loading.gif" data-src="images/10.png">
            <img src="images/loading.gif" data-src="images/11.png">
            <img src="images/loading.gif" data-src="images/12.png">
            <script>
                function lazyload() {
                    var images = document.getElementsByTagName('img');
                    var len    = images.length;
                    var n      = 0;      //存储图片加载到的位置，避免每次都从第一
张图片开始遍历
                    return function() {
                        var seeHeight =
document.documentElement.clientHeight;
                        var scrollTop = document.documentElement.scrollTop
|| document.body.scrollTop;
                        for (var i = n; i < len; i++) {
                            if (images[i].offsetTop < seeHeight +
scrollTop) {
                                if (images[i].getAttribute('src') ===
'images/loading.gif') {
                                images[i].src =
images[i].getAttribute('data-src');
                            }
```

```
                            n = n + 1;
                            }
                        }
                    }
                }
                var loadImages = lazyload();
                loadImages();           //初始化首页的页面图片
                window.addEventListener('scroll', loadImages, false);
            </script>
        </body>
        </html>
```

八、lerna

8.1 介绍

1) 用于管理多个存在依赖关系的包

2) 目录结构

- packages(目录)
- lerna.json(配置文件)
- package.json(工程描述文件)
- packages
  - module-1
    - package.json(工程描述文件)
  - module-2
    - package.json(工程描述文件)

2) 基本工作流

> - `lerna init`