Vue面试题

代码性能优化

减少Object.defineProperty的执行次数

- vue2使用他来订阅data中的所有属性
- vue3使用proxy，解决了如下问题
  - vue2的defineProperty不能察觉属性的添加与删除
  - vue2的defineProperty在数组的所有方法都是自己实现的
  - proxy可以做到
- **只有需要在页面中响应式显示的属性，才放到data中**

只显示一次的变量处理

- 也可以用v-once

```
   data(){
        return {
        // 应针对里面属性不修改
             val:Object.freeze({a:1})
        }
    }
```

一次都不显示的变量处理

```
  // 直接给this.挂载属性使用，如 this.timer

   this.timer = setInterval();
```

keep-alive

- 避免频繁的创建与销毁组件

- `include` - 字符串或正则表达式。只有名称匹配的组件会被缓存。
- `exclude` - 字符串或正则表达式。任何名称匹配的组件都不会被缓存。
- `max` - 数字。最多可以缓存多少组件实例。

keep-alive实例

- 处理方案

```
<keep-alive :include="cacheComponents" max="20">
        <router-view/>
</keep-alive>

<script>
    import cacheComponents from 'config'; //组件名
['01_defineProperty','02_freeze']
    export default {
        name:'main',
        data(){
            return {
                cacheComponents
            }
        }
    }
</script>
```

- max 会根据队列，后进后出，先取消缓存前面的

v-for和v-if为什么不能在一起

- 为什么要在一起?
- v-for执行权重较高

```
<div v-if="data">
    {{ data.xxx }}
</div>
<!--业务需要-->
<div v-for="item in data" v-if="item.show">

</div>
<!--不好的示例：循环次数 * v-if的判断-->
<div v-for="item in data" v-if="data">

</div>
```

- v-for 第一层含义是判断元素是否存在
- v-for执行权重较高, 循环次数 * v-if的判断， 影响性能
- 跟for原理一致，会先判断被循环的元素是否存在，因此不用单独v-if判断元素

```
  <!--不好的方式，n次判断v-if-->
 <div v-for="u in users" :key="u.name" v-if="u.active">
      {{ u.name }}
    </div>
  <!--中等方式，n次js判断-->

<div v-for="u in getActiveUser(users)" :key="u.name">
      {{ u.name }}
 </div>
  <!--高等方式compunted，n次js判断，this.xxx相关属性原值不变，可以缓存-->
 <div v-for="u in activeUsers" :key="u.name">
      {{ u.name }}
 </div>
```

- 总结：computed方式
    - 1. 解决元素存在与否的判断
      2. 执行元素次数的js判断
      3. 原值相同，不会执行，走缓存

减少diff时不必要的DOM操作

- 索引作为key时，key会随着元素增加和移除而改变
- 在diff时，如果告诉了框架 老DOM中的张三 === 新DOM中的李四
- 就会造成不必要的DOM操作
- 尽可能给与准确的key作为diff对比的依据，否则会造成多余的操作（修改），
    - vue会尽可能的就地复用元素，对比出差异，产生修改

v-model语法糖

- @input
- :value
- 语法糖2:

```
<!--父组件中-->
<text-document v-bind:title.sync="doc.title"></text-document>
<!--子组件中-->
$emit('update:title',值)
```

自定义v-model属性名

```
// 子组件定义v-model
Vue.component('base-checkbox', {
  model: {
     // v-model 传递:checked
    prop: 'checked',
      // v-model 绑定@change
    event: 'change'
  },
  // 接收
  props: ['checked']

  template: `
    <input
      type="checkbox"
      v-bind:checked="checked"
      v-on:change="$emit('change', $event.target.checked)"
    >
  `
})
```

减少渲染层级

减少DOM层级

- vue-fragment

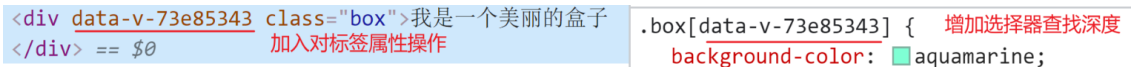
```
import Fragment from 'vue-fragment'
Vue.use(Fragment.Plugin)

// or
import { Plugin } from 'vue-fragment'
Vue.use(Plugin)
```

```
// …

export const MyComponent {
  template: '
  <fragment>
    <input type="text" v-model="message">
    <span>{{ message }}</span>
  </fragment>
  ',
  data() { return { message: 'hello world }}
}
```

减少css层级

- 远离scoped
- 组件的style会将内容作为style标签，动态插入
- 

- 大型项目 使用普通css，特定个例使用scoped css
- 对于template周期以外生成的结构，需要使用/deep/ 来查找
  - 比如第三方的UI库的某些样式
  - 比如v-html中的元素

分析消耗

- 1. 动态创建style标签 : 性能消耗
  2. 浏览器解析多个style标签: DOM阻塞渲染
  3. 给元素添加属性：性能消耗
  4. 浏览器css选择器根据属性查找：阻塞css渲染
- 解决：尽可能少些scoped样式，而直接使用传统方式-全局

函数式组件

- 函数式： pure纯函数，只接收参数，返回数据，没有自身的状态
- function (n1,n2) { let xx = 1; return n1 + n2 }
- 默认组件有状态：自身数据data，生命周期（生命周期初始化）
- 没有生命周期 -> 没有初始化生命周期的js开销

```
<!--方式1:也可以使用模板的方式-->
<template functional>
    <div>
      <!-- 当前作用域就是context作用域`props` `children` `slots`:
 `scopedSlots` `data``parent`: `listeners` -->
        {{data.attrs.text}}
        <slot></slot>
    </div>
</template>
<!--方式2:-->
export default {
    functional:true,
```

```
    render(h,context){
        return h('div',context.props.text,context.children)
    }
}
```

参数列表

组件需要的一切都是通过 `context` 参数传递，它是一个包括如下字段的对象：

- `props`：提供所有 prop 的对象
- `children`：VNode 子节点的数组
- `slots`：一个函数，返回了包含所有插槽的对象
- `scopedSlots`：(2.6.0+) 一个暴露传入的作用域插槽的对象。也以函数形式暴露普通插槽。
- `data`：传递给组件的整个数据对象，作为 `createElement` 的第二个参数传入组件
- `parent`：对父组件的引用
- `listeners`：(2.3.0+) 一个包含了所有父组件为当前组件注册的事件监听器的对象。这是 `data.on` 的一个别名。
- `injections`：(2.3.0+) 如果使用了 `inject` 选项，则该对象包含了应当被注入的 property。

高阶

原样向外导出

```
// index.js
export * from "./RecordSelected";
```

封装组件代码

```
/**
 *
 * @param {Component} 任意被包装的组件
 * @returns {VNode} 渲染后的内容
 */
export const RecordSelected = function(WrappedComponent) {
  return {
    props: {
      ...WrappedComponent.props,
      diffKey: { type: String, default: () => "id" },
      change: { type: Function },
    },
    created() {
      this.selected = [];
      this.plusEvents = {
        "on-select-all": this.toggleAll.bind(this, true),
        "on-select-all-cancel": this.toggleAll.bind(this, false),
        "on-select": (selection, row) => {
          this.toggleState(true, row);
        },
        "on-select-cancel": (selection, row) => {
```

```
          this.toggleState(false, row);
        },
      };
    },
    /**
     * 更新前确保是否选中
     */
    beforeUpdate() {
      this.data.forEach((info) => {
        info._checked = this.selected
          .map((data) => data[this.diffKey])
          .includes(info[this.diffKey]);
      });
    },
    beforeDestroy() {
      this.selected.length = 0;
    },
    /**
     * 包装函数渲染接受的组件
     * @param {Function} 内置渲染函数
     * @returns VNode
     */
    render(h,context) {
      return h(WrappedComponent, {
        on: { ...this.$listeners, ...this.plusEvents },
        props: this.$props,
        attrs: this.$attrs,
        scopedSlots: this.$scopedSlots,
      });
    },
    methods: {
      /**
       * 功能：初始化已选数组
       * @returns {undefined} - 不返回任何数据
       */
      initKey() {
        this.selected.length = 0;
      },
      toggleAll(yesOrNo, selection) {
        selection.forEach((data) => {
          this.toggleState(yesOrNo, data);
        });
      },
      /**
       * 功能：对比已选数组是否存在
       * @param {Any} value - 对比的值
       * @param {Boolean} type - 操作类型boolean true为添加, false为移除
       * @returns {Boolean} - 返回元素是否操作成功
       */
      toggleState(type = true, value) {
        let index = this.selected.findIndex((sData) => {
          return value[this.diffKey] === sData[this.diffKey];
        });
        // 没有该元素且需要添加
        if (index == -1 && type) {
          this.selected.push(value);
          this.$emit("change", this.selected);
          return true;
```

```
        } else if (index != -1 && !type) {
          // 有该元素，且移出
          this.selected.splice(index, 1);
          this.$emit("change", this.selected);
          return true;
        }
        this.$emit("change", this.selected);
        return false;
      },
      /**
       * 功能：判断数据是否在已选数组中存在
       * @param {Any} value - 对比的值
       * @returns {Boolean}} - 返回是否包含该Id
       */
      hasId(value) {
        return this.selected.map((data) => data[this.diffKey]).includes(value);
      },
      /**
       * 向外暴露的函数，外部最终通过调用该函数获取选中的集合数据
       * this.$refs.xxx.getSelection();
       * @returns {Array} - 返回所有选中的数组
       */
      getSelection() {
        return this.selected;
      },
    },
  };
};
```

高阶组件应用

```
<template>
    <Table  :diffKey="id" @change="dosomething">
        <column></column>
        ...
    </table>
</template>
import { SceneResize, RecordSelected } from "../../components/middleware";
import {Table} from 'element-ui';
 components: {
    Table: RecordSelected(Table),
 },
```

- 总结：开闭原则，尽可能不要改别人东西
    - 复用性，提取公共的中间件
    - 利用高阶组件的思想实现中间层转化
    - 中间件创建的时候接收并转化响应的数据
    - 在render的时候，直接传递或加工传递

render(createElement,|context[函数式])函数

```
// @returns {VNode}
createElement(
  // {String | Object | Function}
  // 一个 HTML 标签字符串，组件选项对象，或者一个返回值
  // 类型为 String/Object 的函数，必要参数
  'div',

  // {Object}
  // 一个包含模板相关属性的数据对象
  // 这样，您可以在 template 中使用这些属性。可选参数。
  {
    // (详情见下一节)
  },

  // {String | Array}
  // 子节点 (VNodes)，由 `createElement()` 构建而成，
  // 或使用字符串来生成“文本节点”。可选参数。
  [
    createElement('h1', '一则头条'),
    createElement(MyComponent, {
      props: {
        someProp: 'foobar'
      }
    })
  ]
)
```

相关属性

- createElement(组件,如下)

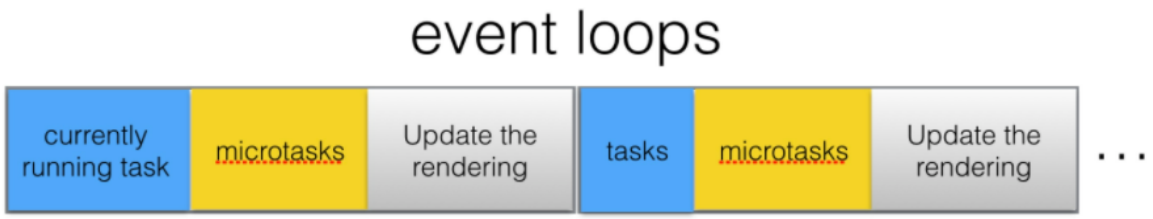
```
{
  // 与 `v-bind:class` 的 API 相同，
  // 接受一个字符串、对象或字符串和对象组成的数组
  'class': {
    foo: true,
    bar: false
  },
  // 与 `v-bind:style` 的 API 相同，
  // 接受一个字符串、对象，或对象组成的数组
  style: {
    color: 'red',
    fontSize: '14px'
  },
  // 普通的 HTML attribute
  attrs: {
    id: 'foo'
  },
  // 组件 prop
  props: {
    myProp: 'bar'
  },
```

```
  // DOM property
  domProps: {
    innerHTML: 'baz'
  },
  // 事件监听器在 `on` 内，
  // 但不再支持如 `v-on:keyup.enter` 这样的修饰器。
  // 需要在处理函数中手动检查 keyCode。
  on: {
    click: this.clickHandler
  },
  // 仅用于组件，用于监听原生事件，而不是组件内部使用
  // `vm.$emit` 触发的事件。
  nativeOn: {
    click: this.nativeClickHandler
  },
  // 自定义指令。注意，你无法对 `binding` 中的 `oldValue`
  // 赋值，因为 Vue 已经自动为你进行了同步。
  directives: [
    {
      name: 'my-custom-directive',
      value: '2',
      expression: '1 + 1',
      arg: 'foo',
      modifiers: {
        bar: true
      }
    }
  ],
  // 作用域插槽的格式为
  // { name: props => VNode | Array<VNode> }
  scopedSlots: {  // 当前组件 <slot></slot><slot name="content"></slot>
    default: props => createElement('span', props.text),
    content:props => createElement('button', props.text),
  },
  // 如果组件是其它组件的子组件，需为插槽指定名称
  slot: 'name-of-slot',  // 目前没卵用
  // 其它特殊顶层 property
  key: 'myKey',
  ref: 'myRef',
  // 如果你在渲染函数中给多个元素都应用了相同的 ref 名，
  // 那么 `$refs.myRef` 会变成一个数组。
  refInFor: true
}
```

```
<!--以上配置等同于-->
    <我这个子组件 key="myKey" ref="myRef">
        <template>
            <span>11</span>
        </template>
         <template v-slot:content>
            <button>11</button>
        </template>
    </我这个子组件>
```

异步

事件循环



微任务&宏任务

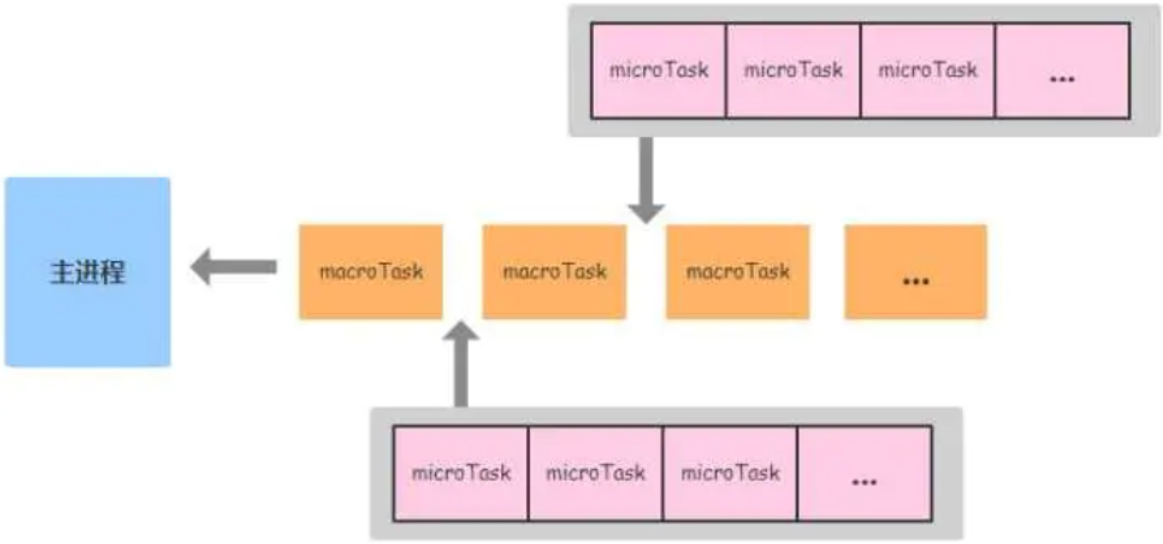
- 宏任务是宿主调用的
- 微任务是js调用的

macrotasks|task:

setTimeout，setInterval， setImmediate，requestAnimationFrame, I/O ，UI渲染

microtasks:

Promise， process.nextTick， Object.observe， MutationObserver



this.$nextTick(()=>{ 下次周期后执行 })

想象Vue的框架代码

```
// 1.启动
new Vue();

// 2.执行构造函数
function Vue() {
    // 3.初始化
    // 4.运行你的代码  this.$nextTick(()=>{   在5.更新后执行}  )
    // 5.更新
}
```

总结

- Vue的更新【数据】利用事件循环中的微任务，在代码中无论调用微任务还是宏任务，都会处于前者更新数据的之后，因此即可实现$nextTick，最终主线程发现所有任务执行完毕，开始各种检查

```
当 Event loop 执行完 Microtasks 后，会判断 document 是否需要更新。因为浏览器是 60Hz 的刷新率，每 16ms 才会更新一次。
然后判断是否有 resize 或者 scroll ，有的话会去触发事件，所以 resize 和 scroll 事件也是至少 16ms 才会触发一次，并且自带节流功能。
判断是否触发了 media query
更新动画并且发送事件
判断是否有全屏操作事件
执行 requestAnimationFrame 回调
执行 IntersectionObserver 回调，该方法用于判断元素是否可见，可以用于懒加载上，但是兼容性不好
更新界面
以上就是一帧中可能会做的事情。如果在一帧中有空闲时间，就会去执行 requestIdleCallback 回调。
```

- 浏览器的一帧说的就是一次完整的重绘。
- requestIdleCallback(function)

React fiber优化

- 将任务切片成一个个小的小任务，在每次更新的时候，那一帧的空闲时间去处理，利用requestIdleCallback函数，任务比较平滑
- fiber据说研发了3年

笔试题

```
console.log('script start');   // 1

async function async1() { // es7的async  就是es6 generator的语法糖，  和Promise是一个级别微任务
  await async2();
  console.log('async1 end');  // 微任务 5
}
async function async2() {
  console.log('async2 end');  // 2
}
async1();

setTimeout(function() {    // 宏任务 8
  console.log('setTimeout')
}, 0)

new Promise(resolve => {
  console.log('Promise');    // 3
  resolve()
})
  .then(function() {
```

```
    console.log('promise1')   // 微任务 6
  })
  .then(function() {
    console.log('promise2') //  // 微任务 7
  })

console.log('script end') //  4
```