talk is cheap , show me the code!

1. v-if和v-for谁的优先级高？

- 测试代码

```
<head>
    <script src="https://cdn.bootcdn.net/ajax/libs/vue/2.6.12/vue.min.js"></script>
</head>
<body>
    <div id="app">
        <h1>v-if 和 v-for 谁的优先级更高? </h1>

        <p v-for="item in children" :key='item' v-if="isFolder"> {{item.title}} </p>
    </div>
    <script>
        const app = new Vue({
            el:'#app',
            data(){
```

```
                return {
                    children: [
                        {title: 'foo'},
                        {title: 'bar'}
                    ]
                }
            },
            computed :{
                isFolder(){
                    return this.children && this.children.length > 0
                }
            }
        })
        console.log(app.$options.render)

    </script>
</body>
```

- 打印结果

```
(function anonymous() {
    with (this) {
        return _c('div', {
            attrs: {
                "id": "app"
            }
        }, [_c('h1', [_v("v-if 和 v-for 谁的优先级更高? ")]), _v(" "), _l((children), function(item) {
            return (isFolder) ? _c('p', {
                key: item
            }, [_v(" " + _s(item.title) + " ")]) : _e()
        })], 2)
    }
}
)

```

> 可以看到 `_l` 里面嵌套着 `(isFolder) ? _c : ...` 的三元判断，说明同级的时候， `v-for` 的优先级大于 `v-if` 。

> 另外我们在vue的源码里面的 `src\compiler\codegen\index.js` 里面发现这么一段代码

```
if (el.staticRoot && !el.staticProcessed) {
 return genStatic(el, state)
} else if (el.once && !el.onceProcessed) {
```

```
 return genOnce(el, state)
} else if (el.for && !el.forProcessed) {
 return genFor(el, state)
} else if (el.if && !el.ifProcessed) {
 return genIf(el, state)
} else if (el.tag === 'template' && !el.slotTarget && !state.pre) {
 return genChildren(el, state) || 'void 0'
} else if (el.tag === 'slot') {
 return genSlot(el, state)
} else {
 // component or element
}
```

也可以说明 `v-for` 的优先级比 `v-if` 高。

怎么进行性能优化

```
<head>
    <script src="https://cdn.bootcdn.net/ajax/libs/vue/2.6.12/vue.min.js"></script>
</head>
<body>
    <div id="app">
        <h1>v-if 和 v-for 谁的优先级更高? </h1>
        <!-- 性能优化的代码 -->
        <template v-if='isFolder'>
            <p v-for="item in children" :key='item'> {{item.title}} </p>
        </template>
    </div>
    <script>
        const app = new Vue({
            el:'#app',
            data(){
                return {
                    children: [
                        {title: 'foo'},
                        {title: 'bar'}
                    ]
                }
            },
            computed :{
                isFolder(){
                    return this.children && this.children.length > 0
                }
            }
        })
        console.log(app.$options.render)

    </script>
```

- 打印结果

```
(function anonymous() {
    with (this) {
        return _c('div', {
            attrs: {
                "id": "app"
            }
        }, [_c('h1', [_v("v-if 和 v-for 谁的优先级更高? ")]), _v(" "), (isFolder) ?
_l((children), function(item) {
            return _c('p', {
                key: item
            }, [_v(" " + _s(item.title) + " ")])
        }) : _e()], 2)
    }
}
)

```

> 可以通过打印 `app.$options.render` 的渲染结果,可以知道，当 `v-if` 的层级比 `v-for` 高的时候，会优先判断 `v-if` 如果不成立则直接渲染空的子节点,从而避免不必要的列表渲染达到性能优化的效果。

2. Vue的组件data为什么要是个函数而Vue的根实例没有这个限制？

- 测试代码

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
  <script src="https://cdn.jsdelivr.net/npm/vue@2.6.14/dist/vue.js"></script>
</head>

<!--
Vue的组件的data必须是一个函数，而Vue的根实例不需要。

 -->
<body>
  <div id="app">
    <!--根实例的data是共享的-->
    <button @click="$data.a++">{{a}}</button>
```

```
    <hr />
    <button @click='$data.a++'>{{a}}</button>

    <!-- 组件的data是相互隔离的 -->
    <h1>函数data</h1>
    <func-add > </func-add>
    <func-add></func-add>
    <hr />
    <h1>对象data</h1>
    <obj-add > </obj-add>
    <obj-add></obj-add>
    <hr />
  </div>

  <script>
    Vue.component('func-add',{
      data(){
        return {
          num:1
        }
      },
      template:`<button @click='num++'>{{num}}</button>`
    })

    Vue.component('obj-add', {
      data:{
        num1: 1   // 会报错vue.js:634 [Vue warn]: The "data" option should be a function that returns a per-instance value in component definitions.
      },
      template:`<button @click='num1++'>n{{num1}}</button>`
    })
    const vm = new Vue({
      el:'#app',
      data: {
        a: 1
      },

    })
  </script>
</body>
</html>
```

> 打开这个文件，我们会发现返回obj的组件是没办法正常渲染的，并且会报错，而跟组件跟返回函数的组件则没有任何问题，但是返回obj的根实例，点击的时候是同时发生变化的，而函数组件是各自变化的。而且即时我们把根组件的data改成返回一个data点击的时候，依然会相互影响，因为是同一个根组件实例，共享data。这也说明了Vue是一个单例模式。

然后，我们也可以到源码中去寻找答案，在`src\core\instance\init.js`里面有这么一段代码

```
 if (options && options._isComponent) {
   // optimize internal component instantiation
   // since dynamic options merging is pretty slow, and none of the
   // internal component options needs special treatment.
   initInternalComponent(vm, options)
 } else {
   /**
       * 合并options
       */
      vm.$options = mergeOptions(
        resolveConstructorOptions(vm.constructor),
        options || {},
        vm
      )
    }
```

说明根实例跟组件合并option配置项的时候的处理方法是不一致的,两个处理方法的源码如下：

```

export function initInternalComponent (vm: Component, options:
InternalComponentOptions) {
  const opts = vm.$options = Object.create(vm.constructor.options)
  // doing this because it's faster than dynamic enumeration.
  const parentVnode = options._parentVnode
  opts.parent = options.parent
  opts._parentVnode = parentVnode

  const vnodeComponentOptions = parentVnode.componentOptions
  opts.propsData = vnodeComponentOptions.propsData
  opts._parentListeners = vnodeComponentOptions.listeners
  opts._renderChildren = vnodeComponentOptions.children
  opts._componentTag = vnodeComponentOptions.tag

  if (options.render) {
    opts.render = options.render
    opts.staticRenderFns = options.staticRenderFns
  }
}

export function resolveConstructorOptions (Ctor: Class<Component>) {
  let options = Ctor.options
  if (Ctor.super) {
    const superOptions = resolveConstructorOptions(Ctor.super)
    const cachedSuperOptions = Ctor.superOptions
    if (superOptions !== cachedSuperOptions) {
```

```
      // super option changed,
      // need to resolve new options.
      Ctor.superOptions = superOptions
      // check if there are any late-modified/attached options (#4976)
      const modifiedOptions = resolveModifiedOptions(Ctor)
      // update base extend options
      if (modifiedOptions) {
        extend(Ctor.extendOptions, modifiedOptions)
      }
      options = Ctor.options = mergeOptions(superOptions, Ctor.extendOptions)
      if (options.name) {
        options.components[options.name] = Ctor
      }
    }
  }
  return options
}
```

@没看懂系列呜呜呜~~

3. key的原理和应用

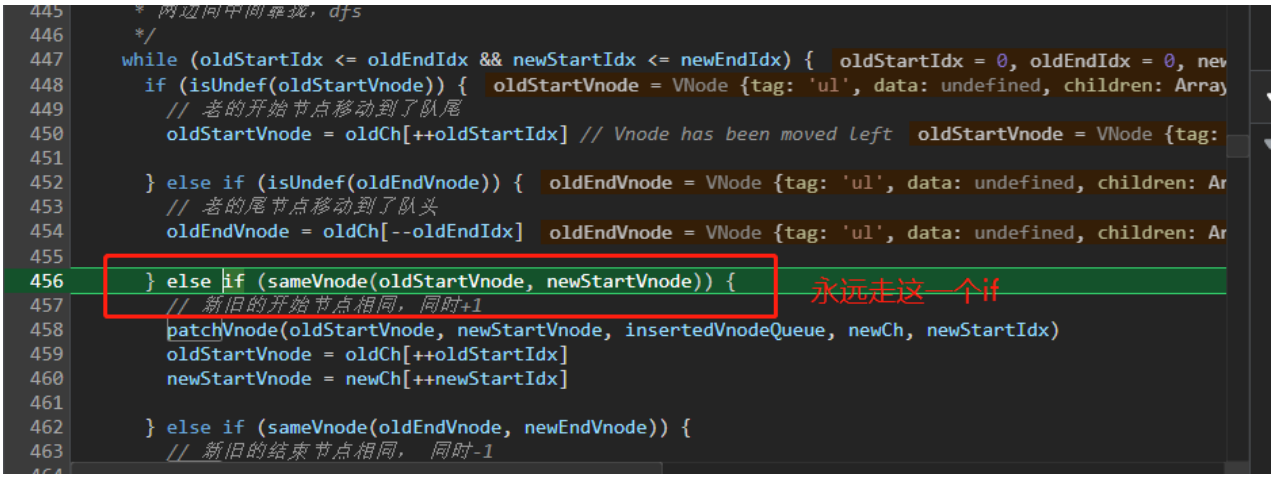
不使用key的时候

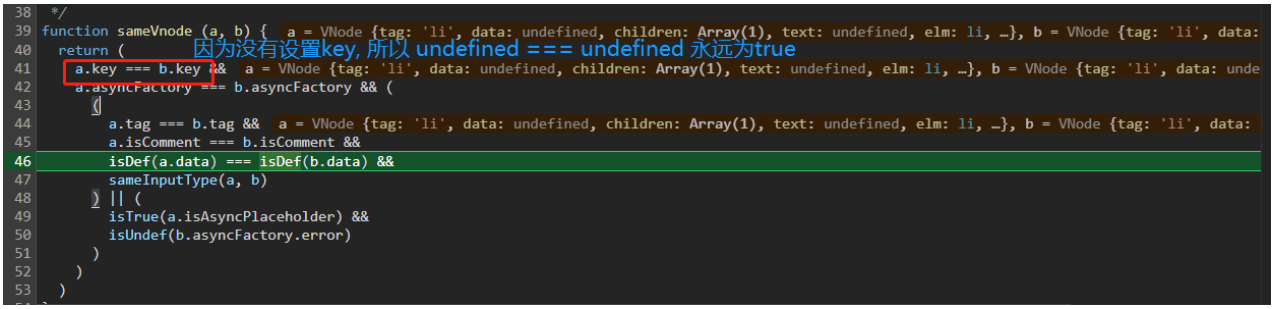
- 测试代码

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <script src='../dist/vue.js'></script>
</head>
<body>

    <!-- key的作用和原理 -->
    <div id="app">
       <ul>
           <li v-for='a in arr'  >{{a}}</li>
       </ul>
    </div>

    <script>
```

```
        const app = new Vue({
            el:'#app',
            data(){
                return {
                    arr:['a','b','c','d','e']
                }
            },
            mounted(){
                setTimeout(()=>{
                    this.arr.splice(2,0,'f')
                }, 3000)
            }
        })
        console.log(app.$options.render)
    </script>
</body>
</html>
```

- 测试结果





所以他的循环是这样的

```
// 第一次循环patch a
a b c d e
a b c f d e



// 第二次循环patch b
b c d e
```

```
b c f d e

// 第三次循环patch c
c d e
c f d e

// 第四次循环patch a
d e
f d e

// 第五次循环patch a
e
d e

```

-
- 测试代码

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <meta http-equiv="X-UA-Compatible" content="IE=edge">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <script src='../dist/vue.js'></script>
</head>
<body>
    <!-- key的作用和原理 -->
    <div id="app">
       <ul>
           <li v-for='a in arr' :key='a' >{{a}}</li>
       </ul>
    </div>

    <script>
        const app = new Vue({
            el:'#app',
            data(){
                return {
                    arr:['a','b','c','d','e']
                }
            },
            mounted(){
                setTimeout(()=>{
                    this.arr.splice(2,0,'f')
                }, 3000)
            }
```

```
        })
        console.log(app.$options.render)
    </script>
</body>
</html>
```

通过调试，发现diff的循环是一下这样的

```
// 第一轮循环 patch a
a b c d
a

// 第二轮循环 patch b
b c d e
b f c d e

// 第三轮循环 patch e
c d e
f c d e

// 第四轮循环 patch d
c d
f c d

// 第五轮循环  patch c
c
f c


// oldch 全部处理完毕，newch剩下f，  创建f并插入

```

1. key的作用主要是为了更高效地更新虚拟DOM, 其原理是在判断是否是同一个节点的时候，Vue源码中首先判断的就是新旧节点的key, 如果不设置key的话，那么undefined === undefined 为真，然后tag等的判断也一致，导致每一个遍历节点都被判断为同一个节点，使得diff算法形同虚设，进行顺序更新，增加了dom的操作，但是如果设置了key的话，就可以有效地避免这些问题。
2. 另外，若不设置key还可能在列表更新地时候引发一些隐藏bug
3. Vue使用在使用相同标签元素地过渡切换时，也会使用到key，其目的也是为了可以让Vue区分不同节点，否则，Vue只会替换其内部属性，而不会触发到过渡效果。

4. Vue的组件模板为什么要求组件只有一个根实例

从三方面考虑：

1. new Vue({el:'#app'})
2. 单文件组件中，template下的元素div。其实就是"树"状数据结构中的"根"。
3. diff算法要求的，源码中，patch.js里patchVnode()。

> 一

实例化Vue时：

如果我在body下这样：

```
<body>
    <div id='app1'></div>
    <div id='app2'></div>
</body>
```

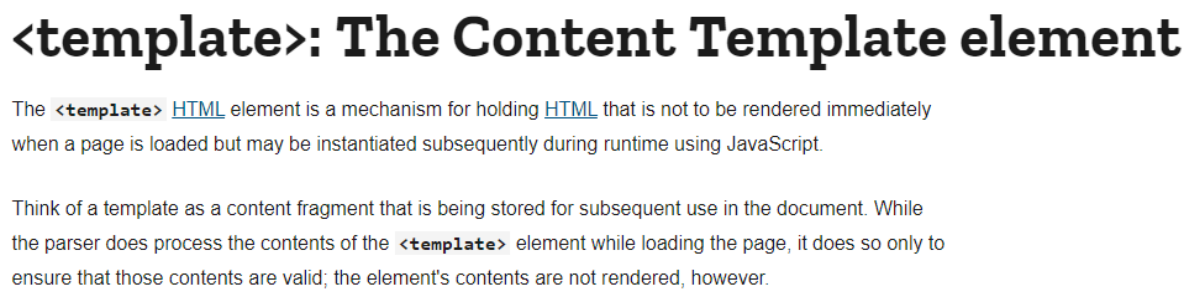
Vue其实并不知道哪一个才是我们的入口。如果同时设置了多个入口，那么vue就不知道哪一个才是这

个‘类’。

> 二

在webpack搭建的vue开发环境下，使用单文件组件时：

```
<template>
    <div> </div>
</template>
```



template这个标签，它有三个特性：

1. 隐藏性：该标签不会显示在页面的任何地方，即便里面有多少内容，它永远都是隐藏的状态，设置

了display：none；

2. 任意性：该标签可以写在任何地方，甚至是head、body、sciprt标签内；
3. 无效性：该标签里的任何HTML内容都是无效的，不会起任何作用；只能innerHTML来获取到里面

的内容。

一个vue单文件组件就是一个vue实例，如果template下有多个div那么如何指定vue实例的根入口呢，

为了让组件可以正常生成一个vue实例，这个div会自然的处理成程序的入口，通过这个根节点，来递归

遍历整个vue树下的所有节点，并处理为vdom，最后再渲染成真正的HTML，插入在正确的位置。

> 三

diff中patchvnode方法，用来比较新旧节点，我们一起来看下源码。

```
function patchVnode () {
    if (oldVnode === vnode) {
      // 新老节点一样
      return
    }
  }
```

> 可以看到里面是从一个节点开始递归的，如果有两个节点的话，那patch就不知道应该递归哪个节点了

大总结

1. v-if 与v-for谁的优先级高？

首先，我的答案是 `v-for` 的优先级是优于 `v-if` 的，因为在源码的 `src\compiler\codegen\index.js` 里面有一段 `if,else if ,else` 里面， `v-for` 是在 `v-if` 的上面,根据js的单线程顺序执行，我们可以知道，js会优先执行对 `v-for` 进行编译，然后才是 `v-if` ；

其次我曾写过 `v-if` 和 `v-for` 在同一层级的测试代码，打印 `Vue` 实例的 `$options.render` 的渲染结果，发现 `_l` 里面嵌套着我们 `v-if` 的三元运算符。所以我的答案是 `v-for` 的优先级是比 `v-if` 大的。

2. v-if和v-for同时出现的时候怎么进行性能优化？

首先我认为这里有两种情况

1. `1v-if` 的判断条件不在 `v-for` 的循环列表里面的时候，我们可以将 `v-if` 的层级提高，然后 `v-for` 在 `v-if` 的子节点里面，避免不必要的列表渲染。
2. `v-if` 的判断条件在 `v-for` 的循环列表里面，这时候我们可以利用一个计算属性来获取一个新的需要渲染的 `v-for` 循环列表，然后 `v-if` 的判断条件是新列表的是否存在以及列表的长度是否大于0，来确定是否需要进行列表渲染。

从而达到性能优化。这是我的优化方案，请问面试官有更好的优化方案吗？

3. Vue的组件data为什么要是个函数而Vue的根实例没有这个限制？

因为 `Vue` 组件是可复用的，如果使用对象形式定义 `data` ，则会导致他们共用一个 `data` 对象，那么状态的变化就会影响到所有的组件实例，很显然这是不合理的；而采用函数的形式定义，在源码中的 `initData` 时会将其作为工厂函数返回一个全新的 `data` 对象，避免多例之间的状态污染。而 `Vue` 是一个单例模式，每个 `Vue` 工程都是共用同一个 `Vue` 实例的 `data` 的，所以不需要考虑状态的隔离，也就是说， `Vue` 根实例的 `data` 可以是一个函数也可以是一个对象。

4. key的原理和作用

1. `key` 的作用主要是为了更高效地更新虚拟 `DOM`，其原理是在判断是否是同一个节点的时候，`Vue` 源码中首先判断的就是新旧节点的key, 如果不设置key的话，那么 `undefined === undefined` 为真，然后tag等的判断也一致，导致每一个遍历节点都被判断为同一个节点，使得diff算法形同虚设，进行顺序更新，增加了dom的操作，但是如果设置了key的话，就可以有效地避免这些问题。
2. 另外，若不设置key还可能在列表更新地时候引发一些隐藏bug
3. Vue使用相同标签元素过渡切换时，也会使用到key，其目的也是为了可以让Vue区分不同节点，否则，Vue只会替换其内部属性，而不会触发到过渡效果。

5. 你怎么看Vue中的diff算法

1. `diff` 算法主要是通过新旧的虚拟 `dom` 的比对，将变化的地方更新在真实的 `dom` 上
2. `vue2` 中为了降低 `Watcher` 粒度，每个组件只有一个 `Watcher` 与之对应，引入 `diff` 可以精确找到发生变化的地方
3. `Vue2` 中的 `diff` 执行的时刻是组件实例执行其更新函数时，他会比对上一次渲染的结果 `oldVnode` 与新的渲染结果 `newVnode`，此过程被尤大称为 `patch`
4. `diff` 的实现细节遵循 `深度优先，同层比较` 的策略，两个新旧节点根据自己是否有文本或者children分别进行头头，尾尾，旧头新尾，新头旧尾, 四个猜想进行比对，尝试找到相同的节点，然后递归搜索，实现两头向中间靠拢，最后批量更新中间不一致的部分，此时他还会搜索oldVnode的队列里面是否有可以重用的节点。如果没有找到相同的节点，则进行常规的遍历操作。
5. 最后 `diff` 进行比对的时候，还使用到了 `key`，`key` 不一致则判断不一致，从而加快比对效率。
6. 不知道我说的有没有不合理的地方，请你指点。（友情客串）

6. 谈一谈你对Vue组件化的理解

1. 组件是独立可复用的代码单元。在我看来，它是Vue核心特征之一，它可以使我们使用小型的组件来构建大型工程。
2. 组件分类按使用来分有：页面组件、业务组件、通用组件
3. 组件化开发可以提高应用开发效率，可对单一组件进行debug，比如说我和我的同学同时开发一个页面，如果不用组件化的思想，那我们写每一行代码之前都需要考虑另一个人有没有写这一行代码，毫无疑问的这会极大地拉低开发效率。
4. 合理划分组件，有助于提高应用性能。比如说在渲染一个大表格的时候，数据发生变化的通常是表格内容，而表格外面包裹的展示内容是不需要变化的，此时我们把表格抽离成为组件，可以在渲染的时候减少虚拟Dom的变化。
5. Vue的组件是基于配置的，我们写的组件通常是组件配置，在打包的时候，其实会被切割的，比如说template最终会先调用render函数获取AST.
6. Vue中常用的组件化技术有： 属性prop,自定义事件，插槽等，他们主要用于组件通信，扩展等

7. 组件是高内聚低耦合的、遵循单向数据流原则，即父组件传给子组件数据，而子组件需要更改数据的时候，不是直接修改数据，而是通过事件派发的方式通知父组件自行更改。
8. 请问面试官，我这样理解有什么问题吗？

7. 谈一谈对Vue设计原则的理解

- 渐进式

与其它大型框架不同的是，Vue 被设计为可以自底向上逐层应用。Vue 的核心库只关注视图层，不仅易于上手，还便于与第三方库或既有项目整合。另一方面，当与现代化的工具链以及各种支持类库结合使用时，Vue 也完全能够为复杂的单页应用提供驱动。

- 易用

Vue提供数据响应式，声明式模板语法和基于配置的组件系统的核心特性。这些使得我们只需要一点点的前端三大件基础，就能轻松写Vue的SFC应用

- 灵活性

渐进式的框架的最大优点的灵活性，如果应用足够小，我们可能只需要Vue的数据绑定就可以完成功能。当我们的项目组件变大的时候，我们才需要说去引入一些其他类库，比如说路由，状态管理以及脚手架等等。总的来说它的学习曲线相对平缓，相当灵活。

- 高效性

使用虚拟Dom和Diff算法使得我们的应用拥有最佳的性能表现。而且Vue3中引入的Proxy来做数据响应式以及脚手架Vite的基于ES6module的加载，使得Vue不管是在开发时候还是线上使用，它的体验都得到了极大的优化。

8. Vue的组件模板为什么要求组件只有一个根实例

1. `new Vue({ el:'#app' })`
2. 单文件组件中，template下的元素div。其实就是"树"状数据结构中的"根"。
3. diff算法要求的，源码中，patch.js里patchVnode()。

9. 谈一谈你对MVC_MVP_MVVM的理解

其实我们这一代前端程序员对MVC的接触并不多，因为我们入门级的前端框架就是MVVM.

而MVC还是我在学习后端开发的时候遇到的，比如说在 `Express` 的一些教材里面，它采用的前后耦合的开发方式就是传统的MVC。

M, 也就是Model, 它主要进行的是一些数据结构的定义以及数据的保存；

V, 就是views也就是视图，在Node里面，它会结合一些模板引擎就是前端的页面渲染，并返回给客户浏览器进行渲染；

C, 就是Control, 后端也称为控制层，它负责监听用户事件，进行一些逻辑处理，最后进行视图更新。

总的来说， MVC是单向数据流的一个开发模式，V到C，C到M，最后M到V, 形成一个闭环。

MVP我并没有真正接触，对它的了解全都源于网络，而我了解到的就是它把MVC中的C变成了P(presenter),

然后，各部分之间的通信，都是双向的，但是View 与 Model 不发生联系，都要通过 Presenter 传递，这就导致View 非常薄，不部署任何业务逻辑，称为"被动视图"（Passive View），即没有任何主动性，而 Presenter非常厚，所有逻辑都部署在那里。

MVVM也就是Vue采用的模式，Model- View- ViewModel三层。它是MVC的发展产物，MV的角色没有发生变化，VM是这个开发模式里面的核心，但同时又是我们无需理解细节即可进行开发的一个层面。

ViewModel需要实现一套数据响应式机制，自动响应Model中数据变化，同时Viewmodel需要实现一套更新策略自动将数据变化转换为视图更新；此外，VM还要通过事件监听响应View中用户交互修改Model中数据。这样在ViewModel中就减少了大量DOM操作代码，就像我们写Vue的时候如果没有特别需求，一般不用写原生的addEventListener。

> MVVM在保持View和Model松耦合的同时，还减少了维护它们关系的代码，使用户专注于业务逻辑，兼
>
> 顾开发效率和可维护性。

10.你了解哪些Vue的性能优化方法

1. 路由懒加载
2. 图片的懒加载，对于图片过多的页面，为了加速页面加载速度，所以很多时候我们需要将页面内未出现在可视区域

   内的图片先不做加载， 等到滚动到可视区域后再去加载（`vue-lazyload`）。
3. 使用`keep-alive`缓存页面
4. 使用`v-for`的时候，避免同时使用`v-if`,同时设置`key`
5. 动态加载第三方组件库的组件
6. `v-show`复用dom
7. 组件摧毁时进行引用标记清理

11. 你了解过哪些路由钩子

> 我使用最多的就是全局导航钩子，
>
> `router.beforeEach`常用于用户的权限认证或者一些页面加载动画的开始以及token过期强制用户重新登录等业务
>
> `router.afterEach()`页面加载动画的结束

12. Vue组件之间的通信方式