**MVVM:**

**M: model 数据模型,从后端获取到的数据**

**V: View 视图页面,展示给用户看到的页面**

**VM: ViewModel,把Model和View联系起来,Vue 起的就是这个桥梁作用**

**当数据发生变化时,视图页面也会发生变化: 视图改变,数据也会发生改变.这就是双向数据绑定**

**MVVM优点**

**MVVM模式和MVC模式一样，主要目的是分离视图（View）和模型（Model），有几大优点:**

**1. 低耦合。视图（View）可以独立于Model变化和修改，一个ViewModel可以绑定到不同的"View"上，当View变化的时候Model可以不变，当Model变化的时候View也可以不变。**

**2. 可重用性。你可以把一些视图逻辑放在一个ViewModel里面，让很多view重用这段视图逻辑。**

**3. 独立开发。开发人员可以专注于业务逻辑和数据的开发（ViewModel），设计人员可以专注于页面设计，使用Expression Blend可以很容易设计界面并生成xaml代码。**

**4. 可测试。界面素来是比较难于测试的，测试可以针对ViewModel来写。**

**v-if和v-show**

**区别**

* **1.手段：v-if是通过控制dom节点的存在与否来控制元素的显隐；v-show是通过设置DOM元素的display样式，block为显示，none为隐藏；**
* **2.编译过程：v-if切换有一个局部编译/卸载的过程，切换过程中合适地销毁和重建内部的事件监听和子组件；v-show只是简单的基于css切换；**
* **3.编译条件：v-if是惰性的，如果初始条件为假，则什么也不做；只有在条件第一次变为真时才开始局部编译（编译被缓存？编译被缓存后，然后再切换的时候进行局部卸载); v-show是在任何条件下（首次条件是否为真）都被编译，然后被缓存，而且DOM元素保留；**
* **4.性能消耗：v-if有更高的切换消耗；v-show有更高的初始渲染消耗；**

**使用场景**

**基于以上区别，因此，如果需要非常频繁地切换，则使用 v-show 较好；如果在运行时条件很少改变，则使用 v-if 较好。**

**总结**

**v-if判断是否加载，可以减轻服务器的压力，在需要时加载,但有更高的切换开销;v-show调整DOM元素的CSS的dispaly属性，可以使客户端操作更加流畅，但有更高的初始渲染开销。如果需要非常频繁地切换，则使用 v-show 较好；如果在运行时条件很少改变，则使用 v-if 较好。**

**computed 和 watch 的区别**

**1. computed是根据依赖的值的变化计算得出新的状态,计算的得到的值需要挂载到Vue实例上;**

**而 watch 是监听已经存在的状态,所有 watch 也可以监听 computed 的属性变化**

**2. computed计算得到的值具有缓冲性,当依赖发生变化时,才会去计算新的状态;**

**watch当数据发生变化才会执行**

**3. 从适用场景来说**

**computed 适用于一个数据受多个数据影响的场景**

**而 watch 适用于一个数据影响多个数据的场景**

**computed 用于同步数据的处理**

**watch 可以处理异步数据,复杂的逻辑**

**生命周期函数(钩子函数 hook)**

**- beforeCreated ———— 创建Vue实例之前**

**在数据观测和初始化事件还未开始，VUE实例的挂载元素$el和数据对象都为undefined**

**可以进行的事件：加loading事件**

**- created ———— Vue实例创建后**

**完成数据观测，属性和方法的运算，初始化事件，$el属性还没有显示出来**

**可以进行的事件：结束loading，请求数据为mounted作准备**

**- beforeMount ———— 实例挂载到DOM之前**

**在挂载开始之前被调用，相关的render函数首次被调用。实例已完成以下的配置：编译模板，把data里面的数据和模板生成html。注意此时还没有挂载html到页面上。（$el和data都已经初始化，但仍是虚拟DOM节点）**

**- mounted ———— 实例挂载后**

**$el被新创建的vm.$el 替换，并挂载到实例上去之后调用。实例已完成以下的配置：用上面编译好的html内容替换el属性指向的DOM对象。完成模板中的html渲染到html页面中。（VUE实例挂载完成，data.filter成功渲染）**

**可以进行的事件：配合路由钩子使用**

**- beforeUpdate ———— 更新前**

**在数据更新之前调用，发生在虚拟DOM重新渲染和打补丁之前。可以在该钩子中进一步地更改状态，不会触发附加的重渲染过程。**

**- Updated ———— 更新后**

**在由于数据更改导致的虚拟DOM重新渲染和打补丁之后调用。调用时，组件DOM已经更新，所以可以执行依赖于DOM的操作。然而在大多数情况下，应该避免在此期间更改状态，因为这可能会导致更新无限循环。该钩子在服务器端渲染期间不被调用。**

**- beforeDestroy ———— 销毁之前**

**在实例销毁之前调用。实例仍然完全可用。**

**- destroyed ———— 销毁之后**

**在实例销毁之后调用。调用后，所有的事件监听器会被移除，所有的子实例也会被销毁。该钩子在服务器端渲染期间不被调用。**

#### 2.VUE组件data为什么必须是函数

答：Vue组件可能存在多个实例，如果使用对象形式定义data，则会导致它们共用一个data对象，那么状态变更将会影响所有组件实例，这是不合理的。  
采用函数形式定义，在initData时会将其作为工厂函数返回全新的data对象，有效规避多实例之间状态污染问题。  
而在Vue根实例创建过程中则不存在该限制，也是因为根实例只能有一个，不需要担心这种情况。

#### 3.Vue中key的作用和工作原理

* key的作用主要是为了高效的更新虚拟DOM,其原理是vue在patch过程中通过key可以精准判断两个节点是否是同一个，从而避免频繁更新不同元素，使得整个patch过程更加高效，减少DOM操作量，提高性能。
* 另外，若不设置key还可能在列表更新时引发一些隐蔽的bug。
* vue中在使用相同标签名元素的过渡切换时，也会使用到key属性，其目的也是为了让vue可以区分它们，否则vue只会替换其内部属性而不会触发过渡效果。

#### 4.你怎么理解vue中的diff算法？

* diff算法是虚拟DOM技术的必然产物：通过新旧虚拟DOM作对比（即diff），将变化的地方更新在真实DOM上；  
  另外，也需要diff高效的执行对比过程，从而降低时间复杂度为0(N).
* vue 2.x中为了降低Watcher粒度，每个组件必须有一个Watcher与之对应，只有引入diff才能精确找到发生变化的地方。
* vue中的diff执行的时刻是组件实例执行其更新函数时，它会比对上一次渲染结果oldVnode和新的渲染结果newVnode，此过程成为patch.
* diff过程整体遵循深度优先、同层比较的策略；两个节点之间比较会更具它们是否拥有子节点或者文本节点做不同操作；  
  比较两组子节点是算法的重点，首先假设头尾节点可能相同做4次比对尝试；  
  如果没有找到相同节点才按照通用方式遍历查找，查找结束再按情况处理剩下的节点；  
  借助key通常可以非常准确的找到相同节点，因此整个patch过程非常高效。

#### 5.谈一谈对vue组件化的理解

* 组件是独立和可复用的代码组织单元。组件系统是Vue核心特性之一，它使开发者使用小型、独立和通常可复用的组件构建大型项目；
* 组件化开发能大幅提高应用开发效率、测试性、复用性等；
* 组件使用按分类有：页面组件、业务组件、通用组件；
* vue的组件是基于配置的，我们通常编写的组件是组件配置而非组件，框架后续会生成其构造函数，它们基于VueComponent,扩展于Vue;
* vue中常见组件化技术有：属性prop，自定义事件，插槽等，它们主要用于组件通信、扩展等；
* 合理的划分组件，有助于提升应用性能；
* 组件应该是高内聚、低耦合的；
* 遵循单向数据流的原则。

#### 6.谈一谈对vue设计原则的理解？

答：首先就是渐进式JavaScript框架：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时，Vue也完全能够为复杂的单页应用提供驱动。

###### 易用性

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等核心特性。  
这些使我们只需要关注应用的核心业务即可，只要会写js,html和css就能轻松编写vue应用。

###### 灵活性

渐进式框架的最大优点就是灵活性，如果应用足够小，我们可能仅需要vue核心特性即可完成功能；  
随着应用规模不断扩大，我们才可能逐渐引入路由、状态管理、vue-cli等库和工具；  
不管是应用体积还是学习难度都是一个逐渐增加的平和曲线。

###### 高效性

超快的虚拟DOM和diff算法使我们的应用拥有最佳的性能表现。  
追求高效的过程还在继续，vue3中将引入Proxy对数据响应式改进以及编译器中对于静态内容编译的改进都会让Vue更加高效。

#### 7.对MVC、MVP和MVVM的理解

###### MVC

具备着View、Controller和Model  
Model:负责保存应用数据，与后端数据进行同步  
Controller: 负责业务逻辑，根据用户行为对Model数据进行修改  
View: 负责视图展示，将Model中的数据可视化出来

###### MVP

MVP与MVC很接近，P指的是Presenter,presenter可以理解为一个中间人。  
它负责着View和Model之间的数据流动，防止view和model之间直接交流。

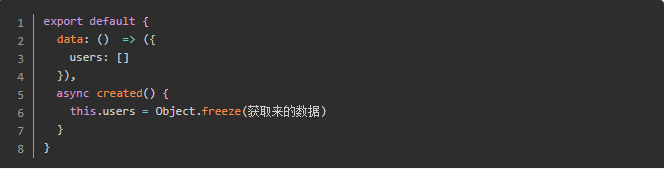
###### MVVM(Model-View-ViewModel)

ViewModel通过实现一套数据响应式机制自动响应Model中数据变化；  
同时ViewModel会实现一套更新策略自动将数据变化转换为视图更新；  
通过事件监听响应View中用户交互修改Model中数据。  
这样再ViewModel中就减少了大量DOM操作代码。  
MVVM在保持View和Model松耦合的同时，还减少了维护他们关系的代码，使用户专注于业务逻辑，兼顾开发效率和可维护性。

###### 总结

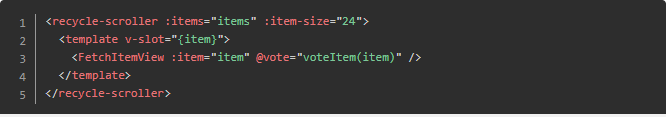
这三者都是框架模式，它们设计的目标都是为了解决Model和View的耦合问题。  
MVC模式出现较早应用在后端，在前端领域的走起也有应用。  
它们的优点是分层清晰，缺点是数据流混乱，灵活性带来的维护性问题。  
MVP模式在是MVC的进化形式，Presenter作为中间层负责MV通信，解决了两者耦合问题，但P层过于臃肿会导致维护问题。  
MVVM模式在前端领域有广泛应用，它不仅解决MV耦合问题，还同时解决了维护两者映射关系的大量繁杂代码和DOM操作代码，在提高开发效率、可读性同时还保持了优越的性能表现。

#### 8.你了解哪些Vue性能优化方法

* 路由懒加载
* 
* keep-alive缓存页面
* 
* 使用v-show 复用DOM
* v-for 遍历避免同时使用v-if
* 
* 长列表性能优化  
  如果列表是纯粹的数据展示，不会有任何变化，就不需要做响应化
* 

如果是大数据长列表，可采用虚拟滚动，只渲染少部分区域的内容

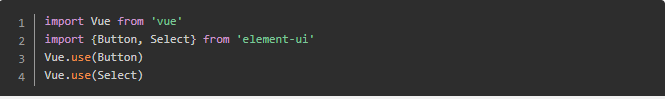
* 事件的销毁  
  Vue组件销毁时，会自动解绑它的全部指令及事件监听器，但是仅限于组件本身的事件。



* 图片懒加载  
  对于图片过多的页面，为了加载页面速度加速。所以很多时候我们需要将页面内未出现在可是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



参考项目： vue-lozyload [https://github.com/hilongjw/vue-lazyload](https://links.jianshu.com/go?to=https%3A%2F%2Fgithub.com%2Fhilongjw%2Fvue-lazyload)

* 第三方插件按需引入  
  像element-ui这样的第三方组件库可以按需引入，避免体积较大。
* 无状态的组件标记为函数式组件
* 
* 子组件分割
* 变量本地化
* 服务端渲染 - SSR

#### 9.你对Vue3.0的新特性有没有了解？

* 更快  
  1.虚拟DOM重写  
  2.优化slots的生成  
  3.静态树提升  
  4.静态属性提升  
  5.基于Proxy的响应式系统
* 更小  
  1.通过摇树优化核心库体积
* 更容易维护  
  1.TypeScript + 模块化
* 更加友好  
  1.跨平台：编译器核心和运行时核心与平台无关，使得Vue更容易与任何平台一起使用
* 更容易使用  
  1.改进的TypeScript支持，编辑器能提供强有力的类型检查和错误及肩高
* 更好的调试支持
* 独立的响应化模块
* Composition API