> 可能已经注意到了 other-component(:msg="msg") 被转化成了 。 mpvue 在运行时会从根组件开始把所有的组件实例数据合并成一个树形的数据，然后通过 setData 到 appData ，$c 是 $children 的缩写 。至于那个 0 则是我们的 compiler 处理过后的一个标记，会为每一个子组件打一个特定的不重复的标记 。 树形数据结构如下

```js
// 这儿数据结构是一个数组 ， index 是动态的
{
  $child: {
    '0'{
      // ... root data
      $child: {
        '0': {
          // ... data
          msg: 'Hello Vue.js!',
          $child: {
            // ...data
          }
        }
      }
    }
  }
}
```

### wxss

> 这个部分的处理同 web 的处理差异不大， 唯一不同在于通过配置生成 .css 为 .wxss ， 其中的对于 css 的若干处理，在 postcss-mpvue-wxss 和 px2rpx-loader 这两部分的文档中又详细的介绍

- 推荐和小程序一样，将 app.json / page.json 放到页面入口处，使用 copy-webpack-plugin copy 到对应的生成位置。

> 这部分内容来源于 app 和 page 的 entry 文件， 通常习惯是 main.js ，你需要在你的入口文件中 export default { config: {} } ， 这才能被我们的 loader 识别为这是一个配置， 需要写成 json 文件

```js
import Vue from 'vue';
import App from './app';

const vueApp = new Vue(App);
vueApp.$mount();

// 这个是我们约定的额外的配置
export default {
    // 这个字段下的数据会被填充到 app.json / page.json
    config: {
        pages: [ 'static/calendar/calendar', '^pages/list/list'], // Will be
        window: {
            backgroundTextStyle: 'light',
            navigationBarBackgroundColor: '##455A73',
            navigationBarTitleText: '美团汽车票 ',
            navigationBarTextStyle: '##fff'
        }
    }
};
```

# 六、React

## 1 React 中 keys 的作用是什么?

> Keys 是 React 用于追踪哪些列表中元素被修改 、被添加或者被移除的辅助标识

- 在开发过程中， 我们需要保证某个元素的 key 在其同级元素中具有唯一性 。在 React Diff 算法中 React 会借助元素的 Key 值来判断该元素是新近创建的还是被移动而来的元素，从而减少不必要的元素重渲染 。此外， React 还需要借助 Key 值来判断元素与本地状态的关联关系， 因此我们绝不可忽视转换函数中 Key 的重要性

## 2 传入 setState 函数的第二个参数的作用是什么?

> 该函数会在 setState 函数调用完成并且组件开始重渲染的时候被调用， 我们可以用该函数来监听渲染是否完成:

```js
this.setState(
  { username: 'tylermcginnis33' },
  () => console.log( 'setState has finished and the component has re-rendere
)
```

```js
this.setState((prevState, props) => {
  return {
    streak: prevState.streak + props.count
  }
})
```

## 3 React 中 refs 的作用是什么

- Refs 是 React 提供给我们的安全访问 DOM 元素或者某个组件实例的句柄
- 可以为元素添加 ref 属性然后在回调函数中接受该元素在 DOM 树中的句柄，该值会作为回调函数的第一个参数返回

## 4 在生命周期中的哪一步你应该发起 AJAX 请求

> 我们应当将AJAX 请求放到 componentDidMount 函数中执行， 主要原因有下

- React 下一代调和算法 Fiber 会通过开始或停止渲染的方式优化应用性能， 其会影响到 componentWillMount 的触发次数 。对于 componentWillMount 这个生命周期函数的调用次数会变得不确定， React 可能会多次频繁调用 componentWillMount 。如果我们将 AJAX 请求放到 componentWillMount 函数中，那么显而易见其会被触发多次， 自然也就不是好的选择。
- 如果我们将 AJAX 请求放置在生命周期的其他函数中， 我们并不能保证请求仅在组件挂载完毕后才会要求响应 。如果我们的数据请求在组件挂载之前就完成， 并且调用了 setState 函数将数据添加到组件状态中，对于未挂载的组件则会报错 。而在 componentDidMount 函数中进行 AJAX 请求则能有效避免这个问题

## 5 shouldComponentUpdate 的作用

> shouldComponentUpdate 允许我们手动地判断是否要进行组件更新，根据组件的应用场景设置函数的合理返回值能够帮我们避免不必要的更新

## 6 如何告诉 React 它应该编译生产环境版

> 通常情况下我们会使用 Webpack 的 DefinePlugin 方法来将 NODE_ENV 变量值设置为 production 。编译版本中 React 会忽略 propType 验证以及其他的告警信息， 同时还会降低代码库的大小， React 使用了 Uglify 插件来移除生产环境下不必要的注释等信息

## 7 概述下 React 中的事件处理逻辑

> 为了解决跨浏览器兼容性问题， React 会将浏览器原生事件（ Browser Native Event ）封装为合成事件（ SyntheticEvent ）传入设置的事件处理器中 。这里的合成事件提供了与原生事件相同的接口，不过它们屏蔽了底层浏览器的细节差异，保证了行为的一致性 。另外有意思的是， React 并没有直接将事件附着到子元素上， 而是以单一事件监听器的方式将所有的事件发送到顶层进行处理 。这样 React 在更新 DOM 的时候就不需要考虑如何去处理附着在 DOM 上的事件监听器， 最终达到优化性能的目的

## 8 createElement 与 cloneElement 的区别是什么

> createElement 函数是 JSX 编译之后使用的创建 React Element 的函数， 而 cloneElement 则是用于复制某个元素并传入新的 Props

## 9 redux中间件

> 中间件提供第三方插件的模式， 自定义拦截 action -> reducer 的过程。变为 action -> middlewares -> reducer 。这种机制可以让我们改变数据流， 实现如异步 action ， action 过滤， 日志输出，异常报告等功能

- redux-logger ： 提供日志输出
- redux-thunk ： 处理异步操作
- redux-promise ： 处理异步操作， actionCreator 的返回值是 promise

## 10 redux有什么缺点

- 一个组件所需要的数据，必须由父组件传过来， 而不能像 `flux` 中直接从 `store` 取。
- 当一个组件相关数据更新时， 即使父组件不需要用到这个组件， 父组件还是会重新 `render` ， 可能会有效率影响， 或者需要写复杂的 `shouldComponentUpdate` 进行判断。

## 11 react组件的划分业务组件技术组件?

- 根据组件的职责通常把组件分为UI组件和容器组件。
- UI 组件负责 UI 的呈现， 容器组件负责管理数据和逻辑。
- 两者通过 `React-Redux` 提供 `connect` 方法联系起来

## 12 react生命周期函数

初始化阶段

- `getDefaultProps` :获取实例的默认属性
- `getInitialState` :获取每个实例的初始化状态
- `componentWillMount` ： 组件即将被装载 、渲染到页面上
- `render` :组件在这里生成虚拟的 `DOM` 节点
- `omponentDidMount` :组件真正在被装载之后

运行中状态

- `componentWillReceiveProps` :组件将要接收到属性的时候调用
- `shouldComponentUpdate` :组件接受到新属性或者新状态的时候（可以返回false，接收数据后不更新， 阻止 `render` 调用，后面的函数不会被继续执行了)
- `componentWillUpdate` :组件即将更新不能修改属性和状态
- `render` :组件重新描绘
- `componentDidUpdate` :组件已经更新

销毁阶段

- `componentWillUnmount` :组件即将销毁

## 13 react性能优化是哪个周期函数

> shouldComponentUpdate 这个方法用来判断是否需要调用render方法重新描绘dom。因为dom的描绘非常消耗性能， 如果我们能在 shouldComponentUpdate方 法中能够写出更优化的 dom diff 算法， 可以极大的提高性能

## 14 为什么虚拟dom会提高性能

> 虚拟 dom 相当于在 js 和真实 dom 中间加了一个缓存，利用 dom diff 算法避免了没有必要的 dom 操作，从而提高性能

具体实现步骤如下

- 用 JavaScript 对象结构表示 DOM 树的结构；然后用这个树构建一个真正的 DOM 树，插到文档当中
- 当状态变更的时候， 重新构造一棵新的对象树 。然后用新的树和旧的树进行比较，记录两棵树差异
- 把2所记录的差异应用到步骤1所构建的真正的 DOM 树上， 视图就更新

## 15 diff算法?

- 把树形结构按照层级分解， 只比较同级元素。
- 给列表结构的每个单元添加唯一的 key 属性， 方便比较。
- React 只会匹配相同 class 的 component （这里面的 class 指的是组件的名字）
- 合并操作，调用 component 的 setState 方法的时候，React 将其标记为 - dirty . 到每一个事件循环结束，React 检查所有标记 dirty 的 component 重新绘制.
- 选择性子树渲染 。开发人员可以重写 shouldComponentUpdate 提高 diff 的性能

## 16 react性能优化方案

- 重写 shouldComponentUpdate 来避免不必要的dom操作
- 使用 production 版本的 react.js
- 使用 key 来帮助 React 识别列表中所有子组件的最小变化

## 16 简述flux 思想

> Flux 的最大特点，就是数据的"单向流动"。

- 用户访问 View
- View 发出用户的 Action
- Dispatcher 收到 Action ，要求 Store 进行相应的更新
- Store 更新后，发出一个 "change" 事件
- View 收到 "change" 事件后，更新页面

## 17 说说你用react有什么坑点?

1. JSX做表达式判断时候，需要强转为boolean类型

> 如果不使用 !!b 进行强转数据类型，会在页面里面输出 0 。

```
render() {
  const b = 0;
  return <div>
    {
      ! !b && <div>这是一段文本</div>
    }
  </div>
}
```

2. 尽量不要在 componentWillRevicesProps 里使用 setState，如果一定要使用，那么需要判断结束条件，不然会出现无限重渲染，导致页面崩溃

3. 给组件添加ref时候，尽量不要使用匿名函数， 因为当组件更新的时候， 匿名函数会被当做新的prop处理，让ref属性接受到新函数的时候，react内部会先清空ref，也就是会以null为回调参数先执行一次ref这个props，然后在以该组件的实例执行一次ref，所以用匿名函数做ref的时候，有的时候去ref赋值后的属性会取到null

4. 遍历子节点的时候，不要用 index 作为组件的 key 进行传入

## 18 我现在有一个button ， 要用react在上面绑定点击事件，要怎么做?

```
class Demo {
  render() {
    return <button onClick={(e) => {
      alert('我点击了按钮')
```

```
    }}>
      按钮
    </button>
  }
}
```

你觉得你这样设置点击事件会有什么问题吗?

> 由于 onClick 使用的是匿名函数，所有每次重渲染的时候，会把该 onClick 当做一个新的 prop 来处理，会将内部缓存的 onClick 事件进行重新赋值，所以相对直接使用函数来说， 可能有一点的性能下降

修改

```js
class Demo {

  onClick = (e) => {
    alert('我点击了按钮')
  }

  render() {
    return <button onClick={this.onClick}>
      按钮
    </button>
  }
```

## 19 react 的虚拟dom是怎么实现的

> 首先说说为什么要使用 Virturl DOM ， 因为操作真实 DOM 的耗费的性能代价太高，所以 react 内部使用 js 实现了一套dom结构，在每次操作在和真实dom之前，使用实现好的diff算法，对虚拟dom进行比较， 递归找出有变化的dom节点，然后对其进行更新操作 。为了实现虚拟 DOM ， 我们需要把每一种节点类型抽象成对象，每一种节点类型有自己的属性，也就是prop，每次进行 diff 的时候， react 会先比较该节点类型，假如节点类型不一样，那么 react 会直接删除该节点，然后直接创建新的节点插入到其中，假如节点类型一样，那么会比较 prop 是否有更新，假如有 prop 不一样，那么 react 会判定该节点有更新，那么重渲染该节点，然后在对其子节点进行比较，一 层一层往下， 直到没有子节点

## 20 react 的渲染过程中，兄弟节点之间是怎么处理的？也就是key值不一样的时候

> 通常我们输出节点的时候都是map一个数组然后返回一个 `ReactNode` ， 为了方便 `react` 内部进行优化， 我们必须给每一个 `reactNode` 添加 `key` ， 这个 `key prop` 在设计值处不是给开发者用的， 而是给react用的，大概的作用就是给每一个 `reactNode` 添加一个身份标识， 方便react进行识别，在重渲染过程中， 如果key一样，若组件属性有所变化，则 `react` 只更新组件对应的属性；没有变化则不更新， 如果key不一样，则react先销毁该组件，然后重新创建该组件

## 21 那给我介绍一下react

1. 以前我们没有jquery的时候， 我们大概的流程是从后端通过ajax获取到数据然后使用jquery生成dom结果然后更新到页面当中，但是随着业务发展， 我们的项目可能会越来越复杂，我们每次请求到数据， 或则数据有更改的时候， 我们又需要重新组装一次dom结构，然后更新页面， 这样我们手动同步dom和数据的成本就越来越高， 而且频繁的操作dom，也使我我们页面的性能慢慢的降低。
2. 这个时候mvvm出现了， mvvm的双向数据绑定可以让我们在数据修改的同时同步dom的更新，dom的更新也可以直接同步我们数据的更改， 这个特定可以大大降低我们手动去维护dom更新的成本， mvvm为react的特性之一， 虽然react属于单项数据流， 需要我们手动实现双向数据绑定。
3. 有了mvvm还不够， 因为如果每次有数据做了更改，然后我们都全量更新dom结构的话，也没办法解决我们频繁操作dom结构(降低了页面性能)的问题， 为了解决这个问题， react内部实现了一套虚拟dom结构，也就是用js实现的一套dom结构，他的作用是讲真实dom在js中做一套缓存，每次有数据更改的时候， react内部先使用算法，也就是鼎鼎有名的diff算法对dom结构进行对比，找到那些我们需要新增 、更新 、删除的dom节点，然后一次性对真实DOM进行更新， 这样就大大降低了操作dom的次数 。 那么diff算法是怎么运作的呢，首先，diff针对类型不同的节点，会直接判定原来节点需要卸载并且用新的节点来装载卸载的节点的位置；针对于节点类型相同的节点，会对比这个节点的所有属性， 如果节点的所有属性相同，那么判定这个节点不需要更新， 如果节点属性不相同，那么会判定这个节点需要更新， react会更新并重渲染这个节点。
4. react设计之初是主要负责UI层的渲染， 虽然每个组件有自己的state，state表示组件的状态， 当状态需要变化的时候， 需要使用setState更新我们的组件，但是， 我们想通过一个组件重渲染它的兄弟组件， 我们就需要将组件的状态提升到父组件当中，让父组件的状态来

控制这两个组件的重渲染， 当我们组件的层次越来越深的时候，状态需要一直往下传， 无疑加大了我们代码的复杂度， 我们需要一个状态管理中心 ，来帮我们管理我们状态state。

5. 这个时候， redux出现了， 我们可以将所有的state交给redux去管理， 当我们的某一个state有变化的时候，依赖到这个state的组件就会进行一次重渲染， 这样就解决了我们的我们需要一直把state往下传的问题 。redux有action、reducer的概念，action为唯一修改state的来源， reducer为唯一确定state如何变化的入口， 这使得redux的数据流非常规范， 同时也暴露出了redux代码的复杂，本来那么简单的功能，却需要完成那么多的代码。
6. 后来，社区就出现了另外一套解决方案，也就是mobx， 它推崇代码简约易懂， 只需要定义一个可观测的对象，然后哪个组价使用到这个可观测的对象， 并且这个对象的数据有更改，那么这个组件就会重渲染， 而且mobx内部也做好了是否重渲染组件的生命周期shouldUpdateComponent，不建议开发者进行更改， 这使得我们使用mobx开发项目的时候可以简单快速的完成很多功能， 连redux的作者也推荐使用mobx进行项目开发 。但是，随着项目的不断变大， mobx也不断暴露出了它的缺点，就是数据流太随意， 出了bug之后不好追溯数据的流向， 这个缺点正好体现出了redux的优点所在，所以针对于小项目来说，社区推荐使用mobx，对大项目推荐使用redux

# 七、Vue

## 1 对于MVVM的理解

> `MVVM` 是 `Model-View-ViewModel` 的缩写

- `Model` 代表数据模型，也可以在 `Model` 中定义数据修改和操作的业务逻辑。
- `View` 代表 `UI` 组件， 它负责将数据模型转化成 `UI` 展现出来。
- `ViewModel` 监听模型数据的改变和控制视图行为 、处理用户交互， 简单理解就是一个同步View 和 `Model` 的对象， 连接 `Model` 和 `View`

> - 在 `MVVM` 架构下， `View` 和 `Model` 之间并没有直接的联系， 而是通过 `ViewModel` 进行交互， `Model` 和 `ViewModel` 之间的交互是双向的， 因此 `View` 数据的变化会同步到Model中， 而Model 数据的变化也会立即反应到 `View` 上。
> - `ViewModel` 通过双向数据绑定把 `View` 层和 `Model` 层连接了起来， 而 `View` 和 `Model` 之间的同步工作完全是自动的， 无需人为干涉， 因此开发者只需关注业务逻辑，不需要手动操作DOM，不需要关注数据状态的同步问题， 复杂的数据状态维护完全由 `MVVM` 来统一管理

## 2 请详细说下你对vue生命周期的理解

> 答：总共分为8个阶段创建前/后， 载入前/后，更新前/后，销毁前/后

- 创建前/后： 在 beforeCreate 阶段， vue 实例的挂载元素 el 和数据对象 data 都为 undefined ， 还未初始化 。在 created 阶段， vue 实例的数据对象 data 有了，el还没有
- 载入前/后：在 beforeMount 阶段， vue 实例的 $el 和 data 都初始化了，但还是挂载之前为虚拟的 dom 节点， data.message 还未替换 。在 mounted 阶段， vue 实例挂载完成， data.message 成功渲染。
- 更新前/后： 当 data 变化时，会触发 beforeUpdate 和 updated 方法
- 销毁前/后：在执行 destroy 方法后，对 data 的改变不会再触发周期函数，说明此时 vue 实例已经解除了事件监听以及和 dom 的绑定，但是 dom 结构依然存在

什么是vue生命周期?

- 答： Vue 实例从创建到销毁的过程，就是生命周期 。从开始创建 、初始化数据 、编译模板 、挂载Dom→渲染 、更新→渲染 、销毁等一系列过程，称之为 Vue 的生命周期。

vue生命周期的作用是什么?

- 答：它的生命周期中有多个事件钩子 ，让我们在控制整个Vue实例的过程时更容易形成好的逻辑。

vue生命周期总共有几个阶段?

- 答：它可以总共分为 8 个阶段：创建前/后 、载入前/后 、更新前/后 、销毁前/销毁后。

第一次页面加载会触发哪几个钩子?

- 答：会触发下面这几个 beforeCreate 、 created 、 beforeMount 、 mounted 。

DOM 渲染在哪个周期中就已经完成?

- 答： DOM 渲染在 mounted 中就已经完成了

## 3 Vue实现数据双向绑定的原理：Object.defineProperty()

- vue 实现数据双向绑定主要是：采用数据劫持结合发布者-订阅者模式的方式， 通过 Object.defineProperty() 来劫持各个属性的 setter ， getter ，在数据变动时发布消息给订阅者，触发相应监听回调 。当把一个普通 Javascript 对象传给 Vue 实例来作为它的 data 选项时，Vue 将遍历它的属性，用 Object.defineProperty() 将它们转为 getter/setter 。用户看不到 getter/setter ，但是在内部它们让 Vue 追踪依赖，在属性被访问和修改时通知变化。
- vue的数据双向绑定 将 MVVM 作为数据绑定的入口 ，整合 Observer ， Compile 和 Watcher 三者， 通过 Observer 来监听自己的 model 的数据变化， 通过 Compile 来解析编译模板指令 （ vue 中是用来解析 {{}} ）， 最终利用 watcher 搭起 observer 和 Compile 之间的通信桥梁， 达到数据变化 — >视图更新；视图交互变化 （ input ） — > 数据 model 变更双向绑定效果。

## 4 Vue组件间的参数传递

父组件与子组件传值

> 父组件传给子组件：子组件通过 props 方法接受数据；

- 子组件传给父组件： $emit 方法传递参数

非父子组件间的数据传递，兄弟组件传值

> eventBus ，就是创建一个事件中心，相当于中转站， 可以用它来传递事件和接收事件 。项目比较小时，用这个比较合适 （虽然也有不少人推荐直接用 VUEX ， 具体来说看需求)

## 5 Vue的路由实现： hash模式 和 history模式

- hash 模式：在浏览器中符号 “#” ，#以及#后面的字符称之为 hash ，用 window.location.hash 读取 。特点： hash 虽然在 URL 中，但不被包括在 HTTP 请求中；用来指导浏览器动作，对服务端安全无用， hash 不会重加载页面 。
- history 模式：h istory 采用 HTML5 的新特性；且提供了两个新方法： pushState() ， replaceState() 可以对浏览器历史记录栈进行修改， 以及 popState 事件的监听到状态变更

## 5 vue路由的钩子函数

> 首页可以控制导航跳转， beforeEach ， afterEach 等，一 般用于页面 title 的修改 。一些需要登录才能调整页面的重定向功能。

- beforeEach 主要有3个参数 to ， from ， next 。
- to ： route 即将进入的目标路由对象。
- from ： route 当前导航正要离开的路由。
- next ： function 一定要调用该方法 resolve 这个钩子 。执行效果依赖n ext 方法的调用参数 。可以控制网页的跳转

## 6 vuex是什么？怎么使用？ 哪种功能场景使用它?

- 只用来读取的状态集中放在 store 中； 改变状态的方式是提交 mutations ， 这是个同步的事物； 异步逻辑应该封装在 action 中。
- 在 main.js 引入 store ， 注入 。新建了一个目录 store ， … export
- 场景有：单页应用中， 组件之间的状态 、音乐播放 、登录状态 、加入购物车


- state ： Vuex 使用单一状态树,即每个应用将仅仅包含一个 store 实例，但单一状态树和模块化并不冲突 。存放的数据状态，不可以直接修改里面的数据。
- mutations ： mutations 定义的方法动态修改 Vuex 的 store 中的状态或数据
- getters ：类似 vue 的计算属性， 主要用来过滤一些数据。
- action ： actions 可以理解为通过将 mutations 里面处里数据的方法变成可异步的处理数据的方法， 简单的说就是异步操作数据 。 view 层通过 store.dispath 来分发 action


> modules ：项目特别复杂的时候， 可以让每一个模块拥有自己的 state 、 mutation 、 action 、 getters ，使得结构非常清晰， 方便管理


## 7 v-if 和 v-show 区别

- 答： v-if 按照条件是否渲染， v-show 是 display 的 block 或 none ；

## 8 $route 和 $router 的区别

- $route 是“路由信息对象”，包括 path ， params ， hash ， query ， fullPath ， matched ， name 等路由信息参数。
- 而 $router 是“路由实例”对象包括了路由的跳转方法，钩子函数等

## 9 如何让CSS只在当前组件中起作用?

> 将当前组件的 <style> 修改为 <style scoped>

## 10 <keep-alive></keep-alive> 的作用是什么?

- <keep-alive></keep-alive> 包裹动态组件时，会缓存不活动的组件实例,主要用于保留组件状态或避免重新渲染

> 比如有一个列表和一个详情，那么用户就会经常执行打开详情=>返回列表=>打开详情 ...这样的话列表和详情都是一个频率很高的页面，那么就可以对列表组件使用 <keep-alive></keep-alive> 进行缓存， 这样用户每次返回列表的时候，都能从缓存中快速渲染， 而不是重新渲染

## 11 指令v-el的作用是什么?

> 提供一个在页面上已存在的 DOM 元素作为 Vue 实例的挂载目标.可以是 CSS 选择器，也可以是一个 HTMLElement 实例,

## 12 在Vue中使用插件的步骤

- 采用 ES6 的 import ... from ... 语法或 CommonJS 的 require() 方法引入插件
- 使用全局方法 Vue.use( plugin ) 使用插件,可以传入一个选项对象 Vue.use(MyPlugin, { someOption: true })

## 13 请列举出3个Vue中常用的生命周期钩子函数?

- created：实例已经创建完成之后调用,在这一步,实例已经完成数据观测, 属性和方法的运算, watch/event 事件回调. 然而, 挂载阶段还没有开始, $el 属性目前还不可见
- mounted： el 被新创建的 vm.$el 替换， 并挂载到实例上去之后调用该钩子 。如果 root 实例挂载了一个文档内元素， 当 mounted 被调用时 vm.$el 也在文档内。

activated ：keep-alive 组件激活时调用

## 14 vue-cli 工程技术集合介绍

问题一： 构建的 vue-cli 工程都到了哪些技术，它们的作用分别是什么?

- vue.js ： vue-cli 工程的核心， 主要特点是 双向数据绑定 和 组件系统。
- vue-router ： vue 官方推荐使用的路由框架。
- vuex ：专为 Vue.js 应用项目开发的状态管理器， 主要用于维护 vue 组件间共用的一些 变量 和 方法。
- axios （或者 fetch 、 ajax ）： 用于发起 GET 、或 POST 等 http 请求， 基于 Promise 设计。
- vuex 等：一个专为 vue 设计的移动端 UI 组件库。
- 创建一个 emit.js 文件，用于 vue 事件机制的管理。
- webpack ：模块加载和 vue-cli 工程打包器。

问题二：vue-cli 工程常用的 npm 命令有哪些?

- 下载 node_modules 资源包的命令：

```
npm install
```

- 启动 vue-cli 开发环境的 npm命令：

```
npm run dev
```

- vue-cli 生成 生产环境部署资源 的 npm 命令：

```
npm run build
```

- 用于查看 vue-cli 生产环境部署资源文件大小的 npm 命令：

```
npm run build --report
```

> 在浏览器上自动弹出一个 展示 vue-cli 工程打包后 app.js 、manifest.js 、 vendor.js 文件里面所包含代码的页面 。可以具此优化 vue-cli 生产环境部署的静态资源，提升 页面 的加载速度

## 15 NextTick

> nextTick 可以让我们在下次 DOM 更新循环结束之后执行延迟回调，用于获得更新后的 DOM

## 16 vue的优点是什么?

- 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
- 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
- 可测试 。界面素来是比较难于测试的， 而现在测试可以针对 ViewModel 来写

## 17 路由之间跳转?

声明式（标签跳转)

```
<router-link :to="index">
```

编程式（ js跳转)

```
router.push('index')
```

## 18 实现 Vue SSR

其基本实现原理

- app.js 作为客户端与服务端的公用入口， 导出 Vue 根实例，供客户端 entry 与服务端 entry 使用 。客户端 entry 主要作用挂载到 DOM 上， 服务端 entry 除了创建和返回实例， 还进行路由匹配与数据预获取。
- webpack 为客服端打包一个 Client Bundle ， 为服务端打包一个 Server Bundle 。

- 服务器接收请求时，会根据 url ，加载相应组件， 获取和解析异步数据，创建一个读取 Server Bundle 的 BundleRenderer ，然后生成 html 发送给客户端。
- 客户端混合，客户端收到从服务端传来的 DOM 与自己的生成的 DOM 进行对比， 把不相同的 DOM 激活，使其可以能够响应后续变化， 这个过程称为客户端激活 。为确保混合成功，客户端与服务器端需要共享同一套数据 。在服务端， 可以在渲染之前获取数据，填充到 stroe 里， 这样，在客户端挂载到 DOM 之前， 可以直接从 store 里取数据 。首屏的动态数据通过 window.__INITIAL_STATE__ 发送到客户端

> Vue SSR 的实现， 主要就是把 Vue 的组件输出成一个完整 HTML ，vue-server-renderer 就是干这事的

- Vue SSR 需要做的事多点（输出完整 HTML)， 除了 complier -> vnode ， 还需如数据获取填充至 HTML 、客户端混合（ hydration ）、缓存等等 。 相比于其他模板引擎（ ejs ， jade 等)， 最终要实现的目的是一样的，性能上可能要差点

## 19 Vue 组件 data 为什么必须是函数

- 每个组件都是 Vue 的实例。
- 组件共享 data 属性， 当 data 的值是同一个引用类型的值时， 改变其中一个会影响其他

## 20 Vue computed 实现

- 建立与其他属性（如： data 、 Store ）的联系；
- 属性改变后， 通知计算属性重新计算

> 实现时， 主要如下

- 初始化 data ， 使用 Object.defineProperty 把这些属性全部转为 getter/setter 。
- 初始化 computed ，遍历 computed 里的每个属性，每个 computed 属性都是一个 watch 实例 。每个属性提供的函数作为属性的 getter ，使用 Object.defineProperty 转化。
- Object.defineProperty getter 依赖收集 。用于依赖发生变化时，触发属性重新计算。
- 若出现当前 computed 计算属性嵌套其他 computed 计算属性时， 先进行其他的依赖收集

## 21 Vue complier 实现

- 模板解析这种事，本质是将数据转化为一段 `html` ， 最开始出现在后端， 经过各种处理吐给前端 。随着各种 `mv*` 的兴起，模板解析交由前端处理。
- 总的来说， `Vue complier` 是将 `template` 转化成一个 `render` 字符串。

> 可以简单理解成以下步骤：

- `parse` 过程，将 `template` 利用正则转化成 `AST` 抽象语法树 。
- `optimize` 过程，标记静态节点，后 `diff` 过程跳过静态节点，提升性能。
- `generate` 过程，生成 `render` 字符串

## 22 怎么快速定位哪个组件出现性能问题

> 用 `timeline` 工具 。 大意是通过 `timeline` 来查看每个函数的调用时常，定位出哪个函数的问题，从而能判断哪个组件出了问题

# 八、框架通识

## 1 MVVM

> `MVVM` 由以下三个内容组成

- `View` ： 界面
- `Model` ： 数据模型
- `ViewModel` ： 作为桥梁负责沟通 `View` 和 `Model`

> 在 `JQuery` 时期， 如果需要刷新 `UI` 时， 需要先取到对应的 `DOM` 再更新 `UI` ， 这样数据和业务的逻辑就和页面有强耦合。

> MVVM

> 在 MVVM 中， UI 是通过数据驱动的，数据一旦改变就会相应的刷新对应的 UI ， UI 如果改变，也会改变对应的数据 。这种方式就可以在业务处理中只关心数据的流转， 而无需直接和页面打交道 。 ViewModel 只关心数据和业务的处理，不关心 View 如何处理数据，在这种情况下， View 和 Model 都可以独立出来，任何一方改变了也不一定需要改变另一方， 并且可以将一些可复用的逻辑放在一个 ViewModel 中，让多个 View 复用这个 ViewModel 。

- 在 MVVM 中， 最核心的也就是数据双向绑定，例如 Angluar 的脏数据检测， Vue 中的数据劫持。

脏数据检测

> 当触发了指定事件后会进入脏数据检测， 这时会调用 $digest 循环遍历所有的数据观察者， 判断当前值是否和先前的值有区别， 如果检测到变化的话，会调用 $watch 函数，然后再次调用 $digest 循环直到发现没有变化 。循环至少为二次， 至多为十次。

> 脏数据检测虽然存在低效的问题，但是不关心数据是通过什么方式改变的，都可以完成任务，但是这在 Vue 中的双向绑定是存在问题的 。并且脏数据检测可以实现批量检测出更新的值， 再去统一更新 UI ，大大减少了操作 DOM 的次数 。所以低效也是相对的， 这就仁者见仁智者见智了。

数据劫持

> Vue 内部使用了 Object.defineProperty() 来实现双向绑定， 通过这个函数可以监听到 set 和 get 的事件。

```js
var data = { name: 'yck' }
observe(data)
let name = data.name // -> get value
data.name = 'yyy' // -> change value

function observe(obj) {
  // 判断类型
  if ( !obj || typeof obj !== 'object') {
    return
  }
  Object.keys(obj).forEach(key => {
```

```
    defineReactive(obj, key, obj [key])
  })
}

function defineReactive(obj, key, val) {
  // 递归子属性
  observe(val)
  Object.defineProperty(obj, key, {
    enumerable: true,
    configurable: true,
    get: function reactiveGetter() {
      console.log( 'get value')
      return val
    },
    set: function reactiveSetter(newVal) {
      console.log( 'change value')
      val = newVal
    }
  })
}
```

> 以上代码简单的实现了如何监听数据的 `set` 和 `get` 的事件，但是仅仅如此是不够的， 还需要在适当的时候给属性添加发布订阅

html

```
<div>
    {{name}}
</div>
```

> 在解析如上模板代码时， 遇到 就会给属性 `name` 添加发布订阅。

js

```
// 通过 Dep 解耦
class Dep {
  constructor() {
    this.subs = []
  }
  addSub(sub) {
    // sub 是 Watcher 实例
    this.subs.push(sub)
  }
  notify() {
    this.subs.forEach(sub => {
```